FORTE: ISON ROBOTIC TELESCOPE CONTROL SOFTWARE

V. Kouprianov[(1)] **and I. Molotov**[(2)]

[(1)]*Central (Pulkovo) Observatory of Russian Academy of Sciences, 196140, 65/1 Pulkovskoye Ave., Saint Petersburg, Russia, Email: v.k@bk.ru*

[(1)]*University of North Carolina at Chapel Hill, Department of Physics and Astronomy, Phillips Hall, CB #3255, 120 E. Cameron Ave., Chapel Hill, NC 27599-3255, USA, Email: vkoupr@email.unc.edu*

[(2)]*Keldysh Institute for Applied Mathematics, 125047, 4 Miusskaya Sq., Moscow, Russia, Email: im62@mail.ru*

ABSTRACT

Space surveillance has a number of important features that make it very different from most of the classical fields of observational astronomy. Among those, we can mention a demand for extremely accurate timing, complex tracking modes, dynamic scheduling, high data acquisition rate, and often unusual telescope setup that may include multiple optical channels working simultaneously. One needs to take all this into account to ensure an adequate telescope control system (TCS) software design.

Since 2011, the International Scientific Optical Network (ISON) project started a transition to the new standard TCS software called FORTE (**F**acility for **O**perating **R**obotic **TE**lescopes) that is now in the course of active development and testing and that should gradually replace the obsolete ISON software throughout the whole network to overcome the numerous design flaws and limitations of the latter.

This new software has Python-based distributed client-server architecture that makes it extremely flexible an scalable to a wide range of sensor apertures and configurations. A tight integration with Apex package for astronomical image analysis helps to automate the complex calibration and maintenance tasks and provides access to stellar catalogs and orbital data. A customizable high-level object-oriented modular approach allows one to easily configure the package for use in some very peculiar sensor configurations, like a new 6-channel 3-mount barrier sensor, in a completely transparent way.

We describe the basic design principles of FORTE and show in detail how it meets these and other requirements of ground-based optical space surveillance.

Keywords: optical measurements; sensors; telescope control software.

1. INTRODUCTION

The International Scientific Optical Network (ISON) project [4], established in 2000's as a joint effort of professional and amateur astronomers and engineers, is an independent coordinated worldwide network of optical sensors, mostly of small apertures but fast response and large fields of view (FOV). Currently it is operated and managed by Keldysh Institute for Applied Mathematics (KIAM) in Moscow and provides most of Russia's measurements of space objects in high orbits. These data form the basis of the KIAM catalog of Earth-orbiting objects. Successful every-day operation of ISON, among other factors, strongly depends on the software used to control the sensors and acquire these data.

As it was already noted in [3], the field of space surveillance and tracking (SST) imposes certain very specific requirements on the capabilities of the telescope control system (TCS). They include:

- Accurate timing, down to the fractions of millisecond for low Earth orbits.
- Tight scheduling, including the ability to acquire images within the specified time frame.
- Fast and variable-rate real-time target tracking.
- Accommodating complex sensor layouts, like multiple optical channels attached to the same mount or multiple mounts acting synchronously.
- Integration with the image processing pipeline.
- Networked architecture for the real-time communication with the data analysis center and with other nodes of the sensor network.

Most of these requirements come from the nature of Earth-orbiting objects – in particular, their fast apparent motion – and from cost effectiveness and sensor efficiency considerations. Among other quite general TCS requirements that are not specific to SST but still must be

met by any modern general-purpose TCS, we can mention the following:

- Distributed client-server architecture.
- Modular design that separates the hardware- and client-independent core from both the low-level hardware access components ("backend") and the high-level observatory control code ("frontend").
- Extensive scripting and automation capabilities.
- Detailed telemetry and logging.

Currently there are three major publicly available platforms aimed at providing hardware-independent observatory control and automation: RTS2 (`http://www.rts2.org`), INDI (`http://indilib.org`), and ASCOM (`http://ascom-standards.org/`). They are open-source, support a vast majority of the advanced amateur grade astronomical equipment, and are widely used in the "classical" astronomical applications involving deep-sky and slow-moving Solar system objects. Unfortunately, an attempt to make them compliant with the SST TCS specifications listed above would require a substantial modification of their code and even their architecture.

That's why we have made a decision in 2011 to create a new all-purpose TCS platform from scratch that would be flexible enough to meet all current and future ISON requirements and serve as the new standard software suite replacing the previous set of TCS software components. Since then, the new platform FORTE (**F**acility for **O**perating **R**obotic **TE**lescopes) has reached its maturity and became the software of choice to drive all new ISON sensors.

2. DESIGN PRINCIPLES OF FORTE TCS

2.1. Programming Model

The first important step in designing the basic principles of a new TCS is to choose a proper programming technology. Traditionally, hardware control applications are developed in a low-level programming language like C/C++ to ensure the maximum performance when talking to the hardware. This, however, leads to a number of restrictions coming from the amount of effort required to implement or re-implement many features of a real TCS, including space-time transformations, interprocess communication (IPC), backend and frontend application programming interfaces (APIs), and many others. Although separate C/C++ libraries exist for most of these tasks, their interoperability is often poor, increasing the effort needed to put it all together. The resulting amount of code for a full-scale TCS then becomes enormous, making it hard to maintain and reducing its flexibility and scalability. Higher-level languages like Java and C# can partially help, but their static nature still limits the runtime flexibility of the system, leading to a need to separately handle all possible cases and types of the input data, which again inflates the size of the codebase. We also do not consider proprietary solutions like LabView that are sometimes used to build telescope automation systems. As a result, we end up in a selection of several modern high-level dynamic (scripting) languages. Among them, Python (`http://www.python.org`) is the most popular choice due to its extreme flexibility and a vast amount of existing libraries, especially for scientific computing.

At first glance, a dynamic interpreted scripting language like Python hardly looks appropriate for the real-time tasks like telescope control due to its relatively low performance. Its scalability is also limited by the global interpreter lock (GIL) that does not allow to run multiple tasks in parallel within a single process. However, a closer look reveals that the low performance is mostly connected with using loops (especially nested) of Python code, which is critical for computational purposes but rarely occurs in the high-level hardware control logic. Lower-level code, in turn, that involves such loops and requires low latency still can be implemented in C/C++ and easily connected to Python code via the Python/C API and extension modules. Such code is mostly associated with certain hardware communications, and its amount is in fact very limited. Most of the other loops can be avoided at all by converting them to asynchronous requests.

The GIL limitation mentioned above is also eliminated by using multiple processes. Indeed, using a separate process for each device is a direct consequence of the requirement to make a *distributed* TCS. In this paradigm, each hardware device is potentially attached to a separate computer, so its software counterpart runs on a separate machine and is connected to other devices via IPC. Hence it is not only more efficient but merely required to keep hardware device control modules in separate processes. All in all, the impact of the low performance of Python loops and GIL on the TCS efficiency is much lower than usually anticipated. Writing most of the code in a higher-level language not only saves coding time; this also helps to implement such complex tasks like automatic alignment and calibration and to make the system very configurable. A Python-based TCS comes with scripting capability built into its core, unlike many other systems that add some kind of scripting on top of the existing infrastructure. Finally, such TCS would also shares the common platform with the standard ISON image analysis infrastructure build around the Apex image analysis system [1, 2], which makes integration with the image processing pipeline very straightforward and natural.

2.2. Data Model

One more important question is the data model that would be flexible enough to accommodate all possible sensor layouts required by the real ISON demands. To achieve this, we suggest a hierarchy of abstract (logical) devices. Some of them are the purely software modules that serve as containers for other devices and coordinate their joint work. Others correspond to just a single real hardware component like a mount controller or a charge-coupled device (CCD) camera and provide the necessary abstraction layer that separates high-level functions that the device is expected to perform during observations from their implementation in the particular hardware device model, which is one of the important requirements mentioned in Section 1. This hierarchy (Figure 1) consists of the components listed below.

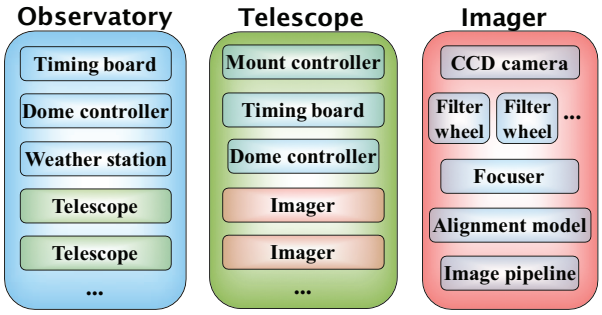

Figure 1. General sensor layout model

- *Observatory* is a top-level device that encapsulates the sensor as a whole. It is responsible for the interaction with FORTE clients (frontends) and dispatching their commands to the corresponding components of the sensor. An observatory may include a dome controller, a timing board, a weather station, and one or more telescopes.

- *Dome* directly interacts with the underlying hardware controller operating the dome or some other type of telescope enclosure, like a sliding roof. A rotating dome has the capability to synchronize with telescope motion[1].

- *Timing board* provides the accurate UTC time to other TCS components. It is also responsible for triggering CCD exposures. Similarly to the dome, the timing board may be associated either with all telescopes or with each telescope individually. Section 3.1 gives more details on how this module is used to ensure the accurate exposure timestamping and scheduling.

- *Telescope* encapsulates a separate mount with all hardware attached to it. It always includes a mount and one or more imagers and may also have an associated dome and/or timing board. Telescope is a pure software device which main purpose is to coordinate the action of all its child components to perform high-level operations like observing the target or automatic calibration.

- *Mount controller* is responsible for pointing and tracking, according to the program of observations and independently from the particular hardware model details. Tracking rates can be updated constantly in real time or between exposures, taking into account the full sky $\leftrightarrow$ image plane transformation, including telescope misalignment and refraction. In addition to the simple (fixed or moving linearly) targets, FORTE supports any source of coordinates and velocities known to Apex, including orbital object catalogs and ephemerides given in the tabular form.

- *Imager* is a container device that corresponds to a separate optical channel of the telescope and includes all hardware responsible for taking images. It consists of strictly one CCD camera and, optionally, one or more color filter wheels (CFWs) and a focuser. Its main purpose is to coordinate their simultaneous action, e. g. make sure that no CFW or focuser motion occurs during exposure, and to provide feedback from the imaging device during automatic focusing. An imager has also the associated *image pipeline* that defines the path followed by images after their acquisition by the CCD camera, like instrumental and astrometric calibration, storage, and display (see Section 3.2). Finally, a *pointing model* is associated with each imager that accounts for the possible misalignment of optics with respect to the mount and of the mount with respect to the local horizon and meridian[2].

- *CCD camera* module controls the given CCD or other solid-state imaging technology device and is responsible for acquiring images with the given parameters, including the hardware triggering support (see Section 3.1).

- *Filter wheel* module provides the basic functionality of setting the required filter or combination of filters in the case of multiple CFWs.

- *Focuser* module controls the optional focusing device on request from a frontend or from the automatic focusing procedure. FORTE will also adjust the focus according to the thickness of the current optical filter.

The model described above accommodates all possible realistic sensor layouts used and planned by ISON and, probably, by most of the other SST projects. Examples of such complex layout are the dedicated ROSCOSMOS

[1] If a single dome is associated with the observatory that includes multiple telescopes, it follows the one which was most recently requested to synchronize with.

[2] A separate pointing model is needed for each imager because of their possible misalignment with respect to each other, like in the compound field of view systems where individual optical channels are looking in the slightly different directions then combined into a larger field of view, which is aimed at increasing the resulting sensor performance.

sensor EOP–1 (Figure 2) and the new 6-channel survey sensor consisting of three separate mounts bearing two wide-field optical systems, each one equipped with a fast CCD camera and a focuser, all mounted in the common enclosure with a sliding roof. In particular, FORTE allows one to operate the latter system both independently, as three separate two-channel telescopes, and simultaneously, as a single 6-channel telescope taking synchronous exposures that can be combined into the common very large field of view. The actual physical and logical layout is described in a text-based configuration file, whereby each device (both pure software and controlling the real hardware) gets its own unique identifier (ID) used to distinguish it from other devices of the same type. A fragment of this configuration file describing the 6-channel sensor layout is shown in Figure 3.

In this layout, each physical device has a unique ID that consists of its own unique ID within its "parent" device, accompanied by the unique ID of the parent device itself; e. g. `ebus_ccd:2@3` identifies the CCD camera attached to the second optical channel (`imager:2@3`) of the third telescope (`telescope:3`). In the case of a simple layout with just a single device of the given type, unique IDs are not needed and can be omitted. FORTE core analyzes this configuration upon initialization and builds the hierarchical sensor model dynamically.

FORTE observatory layout model is implemented by following the plugin paradigm and is based on the object-oriented data model. This means that each device type (e. g. dome, imager, or CCD camera) corresponds to a single abstract Python class that provides the interface for other devices to access its parameters and invoke a set of predefined operations (*"methods"*). To be able to control a certain hardware device model, one needs to subclass the corresponding base Python class and implement its abstract methods in the way specific to the given hardware device. Each class also has attributes that refer to its parent and child devices. This allows each device to access any other device in the hierarchy by traversing the device tree.

FORTE contains also a set of hardware APIs. Usually, they are wrappers around the original software development kit (SDK) supplied by the hardware manufacturer, or around its direct Python binding. These APIs provide a higher-level and more pythonic way of accessing the particular hardware than the original SDKs that are usually written in C/C++.

2.3. Distributed Framework

As it was noted above, one of the key requirements for a TCS is the ability to run it in a distributed computing environment. FORTE distributed core is based on a specially designed remote procedure call (RPC) mechanism that provides transparent access to all features of a device object by other devices running in the different processes and possibly on the different machines within the TCS network, as if these features were accessed locally. This allows one to spread hardware components across the network in a fully arbitrary and configurable manner. Each device is flagged as either local or remote in the observatory layout configuration (Figure 3); each telescope control workstation runs a copy of FORTE with its own local configuration. Each device is marked local on one and only one workstation which it is physically attached to.

A special DSR (driver server registry) service is used to locate devices on the network by their unique symbolic IDs. Upon initialization, each *local* device registers itself in DSR, so its actual network location – host address and TCP/IP (transmission control protocol/Internet protocol) port, assigned dynamically – is made known to other devices. The latter use a proxy object that acts exactly like a Python object that represents the device in their local address space, but in fact redirects all requests to the actual hardware device on the network. For *remote* devices (i. e. hosted on a workstation different from the current), no real driver process is created, and proxy objects are used in the same way to access the driver remotely.

Devices on the network communicate with each other using the specially designed transport protocol over TCP/IP. In addition to the ability to access the remote driver's data attributes and synchronously call its methods, FORTE RPC provides a mechanism for asynchronous method calls. This is called a *task*. When other device initiates an asynchronous call on the target device, a new thread of execution is created within the target device's process that runs the given method, and the first device receives a unique ID of the task just created. The caller may then monitor the status of execution, wait for completion and get the result of running the method, or request the task to terminate. In the latter case, if the method being called initiates other tasks itself, they are terminated too. This mechanism ensures the fastest response possible in case of emergency or when a high-priority observation request comes from the client. It also guarantees the correct cleanup of all child tasks after a premature termination.

2.4. Frontend (Client) API

FORTE is designed according to the client-server model. The server side on each TCS workstation hosts the distributed core and a set of driver processes for devices that are locally attached to this workstation (as well as the pure software devices that are declared local on this workstation). The top-level device in the sensor layout hierarchy, the *Observatory*, serves as a gateway for controlling and monitoring the sensor from the client side. This control is achieved using the same RPC mechanism that is used for internal communications between the different devices. That is, a FORTE client (frontend) has the same level of control over devices and uses the same mechanism for accessing their data attributes and calling (synchronously or asynchronously) their methods. In other words, there is no separate API for controlling FORTE

Figure 2. ROSCOSMOS sensor EOP–1

```
[observatory]
dome = plc_dome
timer = stm32_timer
telescopes = 1, 2, 3

[telescope:1]
mount = sitech_mount
imagers = 1, 2

[telescope:2]
mount = sitech_mount
imagers = 1, 2

[telescope:3]
mount = sitech_mount
imagers = 1, 2

[imager:1@1]
ccd = ebus_ccd
focuser = fli_focuser

[imager:2@1]
ccd = ebus_ccd
focuser = fli_focuser

[hardware.ccd.ebus_ccd:1@1]
local = 0

[hardware.ccd.ebus_ccd:2@1]
local = 1
device = eth1

...
```

Figure 3. Example of configuration for a complex sensor layout

from the outside other than that is used internally and that reflects the sensor model. From this point of view, a frontend is just one of the many devices on the TCS network, and the only difference is that it controls other devices, but none of them can control it.

This approach gives one the maximum flexibility possible when designing client-side applications. On the other hand, it poses a potential security risk due to the ability of the client to directly access any device within the sensor hierarchy. However, this drawback can be easily overcome by adding restrictions on the operations being performed on top of the current client protocol.

The only real difference between accessing FORTE devices internally and from the client side is that the clients are suggested to use a different transport protocol[3]. Currently the client protocol is based on the Extensible Markup Language (XML) and is readable enough by a human to be able to issue commands manually, as well as build and parse such packets by frontends written in any programming language. Figure 4 shows an example of asynchronous call (task) that initiates a series of 8 10-second exposures in R filter of a field with the given sky coordinates and the given fixed tracking rate[4].

In response to this request, the client receives a task ID, which can be used to monitor the progress of the task or terminate it prematurely using the similar XML requests. In Figure 4, `<target>` contains the unique ID of the device in the sensor device hierarchy, `<name>` is the name of method being called (asynchronously), and `<args>` contain the arguments of the call that describe the target object and its observation mode.

Any compliant XML client protocol packet has a one-to-one correspondence to an RPC Python call that can

[3]Using the internal transport protocol is still possible.

[4]This command is powerful enough to also automatically synchronize the dome with the telescope, to run all mutually independent sub-operations – like dome, mount, and CFW, and focuser motion – simultaneously, make sure that all such operations are completed before starting integration, and guarantee it to start at the exact moment or within the specified time limits if needed.

```
<task>
  <target>scope</target>
  <name>observe</name>
  <args>
    <arg name="target">0010+0230_1_1</arg>
    <arg name="ephem_provider">fixed</arg>
    <arg name="ha">00:10:00</arg>
    <arg name="dec">+02:30:00</arg>
    <arg name="tracking">fixed</arg>
    <arg name="tracking_rate"><list>
      <item><float>-1.57</float></item>
      <item><float>3.17</float></item>
    </list></arg>
    <arg name="exposures"><dict>
      <item name="texp">
        <float>10.0</float>
      </item>
      <item name="nexp"><int>8</int></item>
      <item name="filter">R</item>
    </dict></arg>
  </args>
</task>
```

Figure 4. Example of the XML-based client protocol

be issued internally by other devices when talking to the same device (a *Telescope*). Simple rules exist also to convert XML tags to scalar and compound Python types and back. Moreover, the client-side FORTE SDK contains a wrapper that translates XML packets back to Python and vice versa, so FORTE clients written in Python may access the devices transparently, as if they were running on the server side. For example, the packet shown in Figure 4 corresponds to the following Python call:

```
o.scope.start_task(
    "observe", target="0010+0230_1_1",
    ephem_provider="fixed",
    ha="00:10:00", dec="+02:30:00",
    tracking="fixed",
    tracking+rate=[-1.57, 3.17],
    exposures={
        "texp": 10.0, "nexp": 8,
        "filter": "R"})
```

where the first `o` refers to the top-level *Observatory* object used by Python clients as a gateway to the whole device hierarchy. This feature essentially means that a full-scale scripting facility is built into FORTE. By importing the FORTE SDK, any Python script gets access to all sensor features, from high-level observation commands like the one described above down to low-level operations on the individual devices, including telemetry. This way, the user is able to build very sophisticated custom applications for automating observations.

Indeed, this feature is also used internally by several FORTE modules in the same manner. Several high-level methods exist that automate a number of complex tasks, like the automatic alignment and focusing, as well as optimally acquiring twilight flats. These methods are essentially scripts that operate on several devices to make them work in accordance, following the predefined script logic and command flow. The only difference is that such internal "scripts" communicate with other devices via the internal RPC protocol instead of XML packets. But this is only a minor difference since the RPC works in the same way regardless of the actual protocol, so external Python scripts have the full access to exactly the same features and using the same syntax.

An example of a client application is shown in Figure 5. This is the prototype FORTE operator's graphical user interface (GUI), and it utilizes the XML protocol for both high- and low-level control of the different FORTE devices to be able to send the manual observation requests and monitor their progress and the overall TCS state.

Since the XML client protocol runs on top of a TCP/IP connection, there is no difference whether the client is located on the same computer as the server, on the same local area network (LAN), or on a remote computer over the Internet. The same approach can be used to control the sensor by the central network coordination facility or by other sensors. This gives one the maximum amount of flexibility to build the complex sensor networks.

3. OTHER IMPORTANT FEATURES OF FORTE

Software architecture described in Section 2 already solves most problems declared in Section 1. However, a number of requirements listed there are still not met by the architecture itself and need a special consideration. To help one understand how we tackle these problems, Figure 6 illustrates the structure of FORTE as a software package. Below we describe certain FORTE features that are directly related to the rest of the problems stated in Section 1.

3.1. Timing

As it was noted in Section 1, accurate timing is one of the most critical features in SST. Timestamping errors is what often limits the resulting accuracy of measurements more than optical distortions, low signal-to-noise ratio, and other sources of positional error. FORTE was designed with the maximum possible timing accuracy in mind.

Unless one is using a real-time operating system (which severely complicates the implementation and restricts the computing environment), software timing does not provide the level of accuracy necessary for SST applications. Hardware timing, in its most simple implementation, involves a camera with exposures triggered by a source that is synchronized with the Universal Coordinated Time (UTC) scale to within microseconds or better,

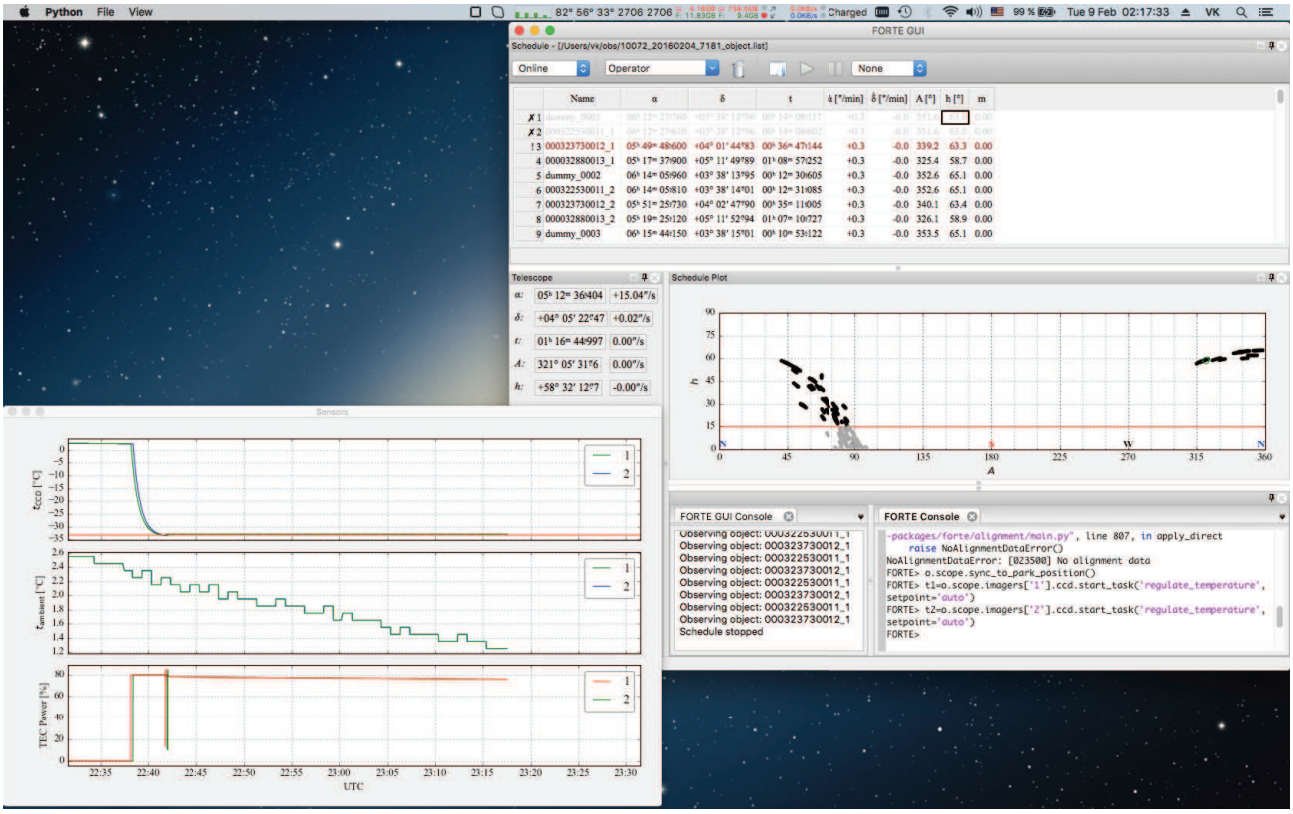

Figure 5. Prototype FORTE operator's GUI

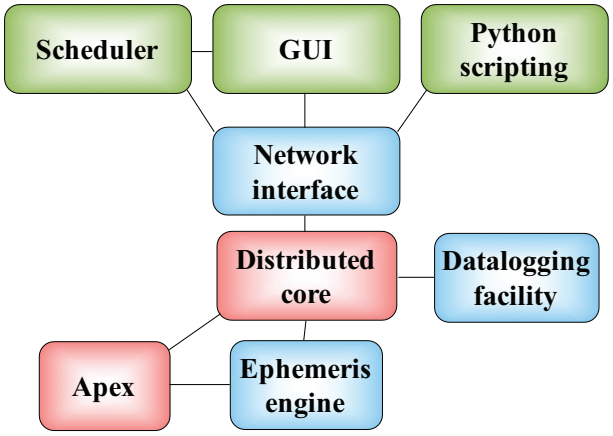

Figure 6. Structure of FORTE as a software package

like a Global Positioning System (GPS) receiver. In its simplest implementation used by ISON for years, this is just a pulse-per-second (PPS) output from a GPS receiver that triggers exposure at the moment of change of a UTC second, with the actual exposure start time measured in software by constantly polling the current exposure state and then rounding the measured time to the nearest whole second; the only limitation is that the framerate cannot exceed 1 Hz. All hardware triggering approaches, however, are still limited by the mechanical shutter and its numerous imperfections, and the only remedy is to use a fast and predictable electronic shutter like e. g. in the interline CCDs or a global shutter in CMOS (complementary metal oxide semiconductor) sensors [5]. For a traditional full-frame CCD with a mechanical shutter, the only way to achieve acceptable accuracy for medium and high Earth orbits is to rely on a sensor that monitors the process of opening and closing the shutter; for low orbits, even this seems to be insufficient. FORTE provides a flexible framework for supporting all these types of hardware timing (of course, in addition to the pure software timing, which may still be enough in certain applications).

Except for exposure timestamping, most other TCS operations (like pointing and dome control) do not require such extremely accurate timing. The normal accuracy achievable e. g. by the Network Time Protocol (NTP) is enough here. FORTE *timing board* module provides the unified interface both for querying the current UTC

time with the accuracy limited by software and for hardware timing tightly linked with the operation of a CCD camera. The interface supports multiple input and output channels, so a single timing board (like the recently designed STM32 board to be used on the modern 6-channel ISON sensors) is able to control the whole sensor or multiple cameras attached to the same mount, depending on the hardware and software configuration. The basic operations provided by the interface include the following:

- return the current software-synchronized UTC time;
- issue a trigger pulse (or a series of pulses with the given time step) to the specified output channel at the given UTC time (*mode 1*);
- issue a trigger pulse (or a series of pulses with the given time step) to the specified output channel as soon as possible; return the actual UTC time of issuing the first pulse (*mode 2*);
- wait for a pulse to arrive from the specified input channel; return the UTC time of the last pulse received (*mode 3*);
- enable or disable PPS for the specified output channel (*mode 4*).

This set of operations is enough to support any combination of features of the particular hardware device model and any type of exposure scheduling. For any type of scheduling requested by the user, the main camera control routine takes most out of the actual timing board capabilities or at least does its best to simulate the required behavior in software as accurately as possible.

For example, if an exposure is scheduled to start at the given UTC moment, the programmed timing board output mode (*mode 1* above) is used if the board supports it; otherwise, the camera control module waits until just about the scheduled time and enables hardware triggering with modes 2 or 4 or uses software triggering and relies on mode 3 to measure the actual exposure start time, depending on which mode is supported by the timing board. If no specific time is scheduled, the camera control module prefers modes 2, 3 and 4 (depending on the input/output triggering capabilities of the camera and the timing board) or, if only mode 1 is available, requests an output trigger pulse at the nearest reasonable time. Similarly, if a series of equally-spaced exposures is requested, the camera control module uses the pulse sequence mode if supported and simulates it with one of the above methods otherwise. This works both for mechanical and electronic shutters. Finally, if a mechanical shutter sensor (see above) is available, its readings are sent back to the camera control module like in mode 3 and are later used by the image processing pipeline to get a more accurate estimate of mid-exposure time.

This structure is tightly integrated with the high-level support for scheduling observations. If an exposure is scheduled to begin at the given time, the pointing, dome control, imaging, and ephemeris subsystems work together to point in advance if possible and get ready to start tracking and exposing immediately before the scheduled moment. An error is generated if the calculated or the actual time to prepare for imaging (incl. pointing, dome synchronization, setting filters, refocusing, and getting the camera ready to start integration) does not allow to start the exposure in time. If a time frame is given for some exposure, the same logic makes sure that the exposure does not start before the beginning or after the end of the given interval, with maximum accuracy possible. All this logic involves a complex interaction between several FORTE devices, which would hardly be possible without the RPC framework described in Section 2.3.

3.2. Image Processing Integration

An important FORTE feature is the image pipeline (see Section 2.2) that is essentially a user-defined set of operations on the image data and metadata. Pipelines consist of elementary operations like image calibration, display, or storage. They run sequentially, in parallel, or in any combination. They are initiated asynchronously immediately after the image readout; metadata hold a set of TCS state parameters before, during, and after integration, as well as some accompanying information like weather conditions. A certain default pipeline is associated with each optical channel of the observatory, but it can be also overridden by client applications individually for each exposure. The most basic image pipeline consists of just storing the image on disk as a flexible image transport system (FITS) file; this is what most of the simple TCS packages do. A more complex example may involve on-the-fly Apex-based image analysis of a set of images to detect tracklets and, in case of an uncorrelated detection, initiate follow-up observations on another sensor using the XML protocol described in Section 2.4.

Sharing the common programming platform with Apex allows us to use certain Apex library features by directly importing the corresponding library modules and thus making Apex an integrated part of FORTE. In particular, this refers to coordinate and time transformations used throughout FORTE and to the support for doing certain calibration tasks. For example, the automatic FORTE alignment procedure uses the Apex capabilities to detect objects, obtain astrometric solution, and calculate the celestial coordinates of the image center. Similar techniques are used by automatic focusing. Finally, the automatic twilight flat sequence obtains a number of image characteristics to maintain the necessary quality of flats and to choose the appropriate exposure time.

3.3. Datalogging

FORTE datalogger is based on the built-in Python logging facility; thus it also automatically handles log messages from all external modules that use the same facility. Various backends are supported, including disk files

with optional automatic rotation, Unix syslog daemon, Windows event log, and sockets. The actual logging configuration, including specifying destinations for different types of events and message formats, is fully defined by the user.

The same facility is used to collect the various telemetry data and hardware usage statistics, including motor revolutions, shutter cycles, voltages, and so forth. This helps to monitor sensor health and schedule maintenance.

FORTE has many other features that are out of the scope of the current paper. Our goal was to show just the basics that are most relevant to space debris observations.

4. CONCLUSIONS

Here we outlined the main requirements for a telescope control system (TCS) specific to the field of space surveillance and tracking (SST), as well as a number of general requirements for any modern TCS. The former are coming, either directly or indirectly, from the nature and specific features of objects being observed – mainly from their high apparent velocities and quickly changing orbits.

We described the new standard ISON optical sensor control system named FORTE, its basic structure, design principles, and some implementation details relevant to building a network of wide-field optical sensors in general and to the area of SST in particular. It was shown how we met all these requirements in FORTE by creating a distributed object-oriented Python core and a set of mutually interacting modules with the capability of detailed remote control and scripting and with a major focus on accurate hardware timing. FORTE is tightly integrated with the Apex-based image processing pipeline, which strongly enhances its capabilities and results in a significant improvement of space debris discovery rate and of the overall ISON performance in general.

REFERENCES

1. Devyatkin A., Gorshanov D., Kouprianov V., Verestchagina I., (2010). Apex I and Apex II software packages for the reduction of astronomical CCD observations. *Sol. Sys. Res.*, **44**(1), 68–80.
2. Kouprianov V., (2008). Distinguishing features of CCD astrometry of faint GEO objects. *Adv. Space Res.*, **41**(7), 1029–1038.
3. Kouprianov V., (2013). ISON Data Acquisition and Analysis Software. In *Proc.* 6^{th} *European Conf. Space Debris, 22–25 April 2013, Darmstadt, Germany* (Ed. L. Ouwehand), ESA SP-723, ISBN 978-92-9221-287-2, id. 21.
4. Molotov I., Agapov V., Titenko V., Khutorovsky Z., Burtsev Yu., Guseva I., Rumyantsev V., Ibrahimov M., Kornienko G., Erofeeva A., Biryukov V., Vlasjuk V., Kiladze R., Zalles R., Sukhov P., Inasaridze R., Abdullaeva G., Rychalsky V., Kouprianov V., Rusakov O., Litvinenko E., Filippov E., (2008). International scientific optical network for space debris research. *Adv. Space Res.*, **41**(7), 1022–1028.
5. Schildknecht T., Peltonen J., Sännti T., Silha J., Flohrer T., (2014). Improved Space Object Orbit Determination Using CMOS Detectors. In *Proc. Advanced Maui Optical and Space Surveillance Technologies Conference, September 9–12, 2014, Wailea, Maui, Hawaii* (Ed. S. Ryan), The Maui Economic Development Board, id. E6.