Ground Station Software: *A Dynamic and Scripted Approach*

ABSTRACT

In the satellite world, there are many pieces of software used to control ground stations. Several iterations of such software exist, mainly as the result of research projects either by universities or the government. Unfortunately, these pieces of software all repeat common mistakes and little improvement in the software is made. The main goal of this paper is to provide an outline for a reusable and extensible application for the manual and automated control of networked ground stations. Essentially, the focus of this project is to address the problems perceived in existing ground station software. These problems are generally addressed through the use of a dynamic language, an object-oriented approach (everything, including primitive data types, is an object in Ruby), and the fact that the program is essentially open source because it is written in an interpreted language.

KEYWORDS

Ground station	Ruby
Scripted approach	GUI

Biswajit Panja
Computer Science, Engineering and Physics
University of Michigan-Flint, Flint, MI 48502
bpanja@umflint.edu

Bradley Schneider
Mathematics, Computer Science, and Physics
Morehead State University, Morehead, KY 40351
bradschneider@live.com

Priyanka Meharia
Accounting and Finance
Eastern Michigan University, Ypsilanti, MI 48197
pmeharia@emich.edu

INTRODUCTION

Many diverse solutions exist in the field of ground station software. However, almost all of these have failed to gain widespread acceptance and use. In other fields, software solutions see large re-use because they sufficiently address the issues at hand and provide the necessary features to accomplish a certain job. Existing ground station software fails to do that. Current trends favor unnecessarily complex code written with static languages which are not well-suited to the job. Specifically, many existing software solutions use a modular framework to facilitate re-use and flexibility. However, the modular framework isn't ideally implemented in a static language environment due mainly to its complexity and somewhat to its inefficiency. Successful software must focus on providing the most important features to the user through the simplest interface possible, and doing that means using new techniques.

PROBLEM STATEMENT

The single largest problem with existing ground station software is that it isn't as flexible as software developers try to make it. Most implementations use Java as their language of choice because it is considered a reliable cross-platform solution. While this might be true, Java is a very static language, just like most popular languages, such as C and C++. These languages are powerful and well-tested, but that alone does not qualify them as good choices for ground station software. While they are flexible languages in a sense that they can perform many diverse tasks, the paradigms and design patterns they dictate are not always desirable. Due to the fact that they are static languages, developers are forced to create complex systems to allow for the flexibility which they desire in their applications. For example, with the modular approach many projects use, software is organized into small components so that the updating, removing, or adding of a component has the smallest possible effect on the other components. This saves time by ensuring that system administrators do not have to re-compile the entire application, but rather only the new parts, which is a good thing. But in order for these components to plug into each other correctly, each piece must pass messages to the other pieces in the specified way. In a system of significant size, which categorizes most implementations, this quickly becomes very difficult. What all this boils down to is that this setup clearly has its flaws. Attempting to make a flexible piece of software requires flexible code and technology in the background; using static languages is in fact possible, as shown by existing projects, but such technology is certainly not desirable for this task.

GROUND STATION SOFTWARE DESIGN

Luckily, technology is always changing for the better, making things more efficient and user- and developer-friendly. One such technology which will improve the world of ground station software is *dynamic programming languages*. The definition of a dynamic programming language is not entirely clear, but in general the term refers to a group of high-level languages which performs at run-time operations which most languages perform at compile time, if they perform them at all. Dynamic languages are not necessarily a new thing; these or similarly designed languages have been around for decades. However, new technology has addressed issues such as efficiency which have in the past made them inferior to the more popularly used static languages. Dynamic languages come with benefits and features that static languages don't – they can extend objects and add new code at run-time, for example. These features aren't necessarily exclusive only to dynamic languages, but dynamic languages provide easy access to them while other languages would require unattractive hack-like coding if they are supported. Because of the extensibility possible in applications developed with dynamic languages, they are well-adapted for creating systems which deal with unknown or unpredictably changing components –a category of systems which includes ground stations.

THE DYNAMIC DESIGN

While choosing a new and innovative language for this project might not alone and at face value seem to lead to a new approach, a deeper consideration will show that it indeed does; for lack of a better term, this approach, which is somewhat dictated by the nature of a dynamic language, will be referred to as a d*ynamic design* or a d*ynamic approach*. It simply refers to the paradigm most closely and naturally linked with the methodology behind such dynamic languages. Additionally, in this paper the term "dynamic design" also refers to a modular design. It was earlier stated that a modular design is undesirable. This is because technologies used in other software cannot support it well. When the technology in the background changes to a dynamic

language, however, a modular design quickly becomes not only simple and easily implemented, but also natural.

DESIGN GUIDELINES

With the argument for a revolution in the form of dynamic programming languages comes the proposal for a new ground station software project. In keeping with the theme of dynamic software, a dynamic programming language and thus a dynamic design will be the best approach. For the scope of this project, two languages seem to be contestable – Ruby and Python. Both have well-developed resources and support as well as a tendency to speed up the development cycle. Both have fairly recently gained a majority of their current popularity, having lived in the shadow of other, more often used languages. These languages are not, however, to be considered too young or undeveloped. With the boom in projects utilizing Python and Ruby, resources have flourished as has the development of each language. With that in mind, these are the two best options for the development of new ground station software. Because one of the enumerated goals in the design of this project is object-orientation (at the code level), and due to the fact that Ruby's support for classes seems more elegant and to be a more central feature to the language, Ruby will be the language used. Both certainly have advantages and disadvantages, but these will not be weighed further here.

High Level Design

The following attempts to outline the guidelines for the design and implementation of the new dynamic ground station software. The vision of the ground station includes both manual and automated modes. The system will feature an easy-to-use interface to increase the simplicity and usability of the application and to aid in the scheduling of tasks. More details are described below:

Reusability across platforms- Reusability and flexibility go hand in hand. There are two essential questions that need resolution to meet this goal: How modifiable is the code, and to what extent is the code abstracted from the system hardware? Because of the possibly diverse nature of ground station implementations, the software must be as modifiable as possible to ensure its cross-platform success. One large step toward this is taken for the developer by Ruby – because the code is interpreted, the source is readily available and modifiable. This allows station administrators to modify necessary code to ensure compatibility with their particular implementation. Another requirement of the application is that it abstracts the application from the hardware. In separating the application-specific tasks from the hardware, the systems on which the code will run are greatly increased in number. Much of this is done simply through the use of a high-level language.

Multi-tiered architecture

In keeping with trends of modern internet applications, the application should follow somewhat closely a three-tiered architecture. The interface presented to the user (or the scheduler in the case of automated control) should interact with a server to retrieve information stored in the database. The top, most visible layer of the application should have no direct contact with the database and data storage level. This keeps communications standard and simple, and avoids confusing and random access to data. Also, because remote access is possible, it might be at some time necessary for a local server to access a remote database. This must also be done through the remote server, such that servers may communicate remotely, but only a local server may access a local database.

User-interface

In keeping with the theme of using advanced programming technology, the user interface will be very modern and simple to use also. Many GUI's are overly simplified, making their use more difficult than necessary. The GUI should be kept as simple to use as possible for the sake of the user, not as simple to create as possible for the sake of the developer. In the GUI, satellite and telemetry data will be easily viewed using various windows and controls. The changing of satellite parameters is also made possible by the GUI. If the satellite is not in range, the adjustments should be scheduled and made when the satellite is available. Therefore, schedule data should be presented and made manageable through means of the GUI. The ability to make a routine schedule to be executed at every pass is also crucial.

Scheduling- The ability to run the station in automated mode should be a key feature of any new ground station software, this one included. One key feature of some dynamic programming languages of which Ruby also takes advantage is *Reflection*. A program written in a language that is said to be

reflective is capable of producing or extending its own code. The GUI will interpret user events or accept textual input and in turn produce a valid script to be executed at a pre-determined time. This script could also simply be hand-coded if necessary, but the validity of such a script might be questionable. Essentially, the GUI should provide a sort of graphical coding option which allows the user to select available actions and commands from a list or by means of visual controls and widgets so as to avoid any errors in the scheduled process yet still allow for full control and maximize the use of all of a satellite's capabilities.

The above summarizes the most innovative and important aspects of the software being proposed. These are not all the innovations, but summarize the areas of development or use which are most greatly affected by the switching to a more dynamic design. What is expected of this project is that it will reach a wider audience than previous software. Previous designs have failed to be flexible. They are prematurely optimized for modification using tools not meant for the job, which detracts from what the focus of ground station software should be – providing the user with the correct tools to do what needs to be done in order to use and maintain a satellite in orbit.

Implementation

Now that the high-level design and goals have been revealed, the low-level details of the software must be clarified. The software will be coded in Ruby and utilize MySQL for database functions. The program will be modular in design. This will, as is typical, mean that modifying one piece of code affects a very small amount of other code. Where this design differs from others is that no re-compilation is necessary, since the code is interpreted. The source files must be available to run the program, so the source will also have one hundred percent accessibility, which aids in the ability to modify the code. In addition, because of language-specific features, items can even be modified at the object level without too much hassle. The new software will be best described as a loosely coupled system while being strongly modular. The software is loosely coupled because the modules can access only their own data and are aware of only data given to them from other modules. They cannot haphazardly access data from other modules. The modules may be called "strong" because each module has a very narrow andspecific function, as opposed to a weak module. This follows object-oriented principles also, such that each module will most nearly contain only one class. Each class or module has the goal of being specific and narrow. Here are some of the key modules and a brief description of each:

***GUI Front End Application* – This is the client which allows the user access to all of the ground station's software features. It is presented as a GUI application.**

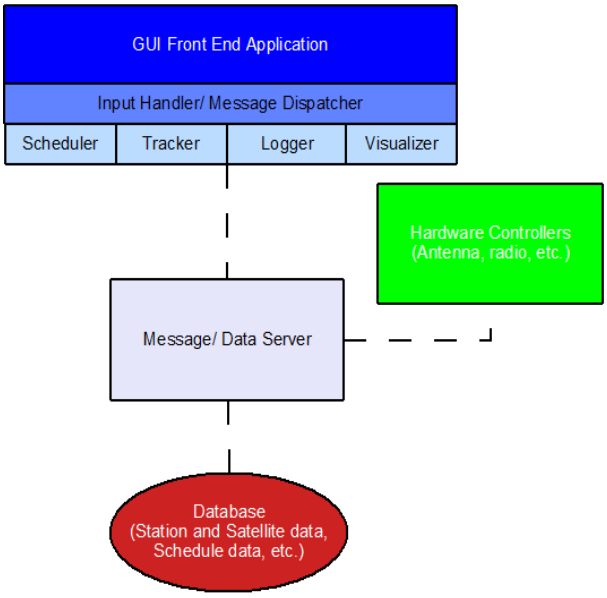
Input Handler – The input handler and message dispatcher handles all user input from the GUI application and dispatches it to the corresponding module. Modules are thus unaware of irrelevant input and only the necessary modules are sent messages.

Scheduler – The scheduler handles multiple tasks and is broken down into sub-modules. First, the scheduler executes schedules at a specified time. Second, the scheduler creates schedules through the use of the GUI application and stores them in the database.

Tracker – The tracker is responsible for predicting passes and communication time for the satellite with which the user is communicating.

Logger – The logger logs all station activity for future reference and review.

Visualizer – The visualizer is the part of the ground station software which displays satellite data through various controls and widgets.



Of course, the above diagram is very simplified. These components are those most exposed to the user, though sometimes the user does not realize it. Other components exist and will be used behind the scenes as inherited classes, for example. One instance of this is the messaging system. Each component needs to be able to send and process messages, so it is logical for each module to contain an instance of the message system class. This is not shown on the diagram because it is in a level below the components mentioned.

As mentioned, each of the components of the ground station in the diagram communicates with the others through a standard messaging interface. These messages are sent from the originating module to the message server, which then dispatches them to the appropriate destination module. In this way, the code is easily modified to fit the needs of an individual station. The developer for the station which needs modification has multiple options: 1) Modify the existing modules, 2) modify the message sending functions to redirect messages to a custom module, or 3) create entirely new modules which send messages appropriately and are therefore integrated into the existing structure seamlessly and with little effort. Because of the interpreted nature of the source, developing, testing, and debugging are incredibly quick and simple. The developer can modify multiple modules in seconds, without recompiling or any other overhead.

CONCLUSION

A quick search of scholarly resources will prove that many different projects, especially at the university level, work toward the result of creating ground station software. These projects are nearly all the same, so none of them has succeeded. There is hardly a mention in any of these projects of other software, proving that there is no consideration of what has been done and has been proven to not work. With an extensible application like the one detailed in this paper, future projects can focus on extending the capabilities of ground stations instead of rewriting the same software with a different name. Additionally, new functionality which they might desire will be easily added to software as dynamic as the one described above.

REFERENCES

i. Tuli, T., Orr, N., and Zee, R., "Low cost station design for nanosatellite missions," University of Toronto Institute for Aerospace Studies Space Flight Laboratory, 2006.

ii. Cutler, J., and Fox, A., "A Framework for robust and flexible ground station networks," Stanford University.

iii. Shirville, G., and Klofas, B., "GENSO: A global ground station network," AMSAT Symposium, October 2007.

iv. Jackson, C., and Lawrence, J., "Distributed operation of a military research microsatellite using the internet," American Institute of Aeronautics and Astronautics.

v. Bernier, S., and Barbeau, M., "A virtual ground station based on distributed components for satellite communications," Small Satellite Conference, 2001.

.