Noninvasive Re-architecture of Legacy Systems

Ryan Berkheimer

GST, Inc.

16 June 2017

Introduction

Organizations that deal in climatological data - particularly front line research organizations, such as NOAA's National Centers for Environmental Information (NCEI) - necessarily deal with a plethora of computing systems which collect, ingest, process, publish, and archive data. These computing systems are diverse in character, author, language, and scope. They are often created and maintained in house over years and even decades, and it is not uncommon to use a system that has been in operation for 20 years or longer, authored or altered by multiple authors of different scientific disciplines and responsibilities, which contains literally hundreds of separate files, modules, scripts, and libraries, all written in a variety of languages.

Domain Observations

As an example, the Pairwise Homogeneity Algorithm (PHA) and its encompassing GHCNM software has now been in development, production, or operation since the early 1990's - going on 30 years, over four major versions and many more minor revisions, and has had at least two primary authors and many more associated researchers and editors. Before PHA was refactored in 2014-2016, it consisted of over a dozen nested makefiles or other build files, several author-specific approaches to version control, and four distinct programming languages, at least one of which was nearly 40 years old.

As another example, the Automated Surface Observing System (ASOS) ingest system has been in development, production or operation status since the 1980's. ASOS itself is a polyglot system consisting of three tiers - a nationwide on-site meteorological sensor array network with over 900 distinct locations; many distributed regional oversight offices which collect and verify data from subsets of these stations; and a central systems oversight which collects, verifies, analyzes, and publishes all data. The ingest system operates within this third tier, collecting and processing data sent from five different ASOS related

data sources, either as input data or assistance data. It is currently under re-architecture and is the first practical case study of this publication. This system has had an unknown number of contributing authors, no software version control, sparse code-level documentation, and contains at least two distinct shell languages, a modem protocol over 30 years old, a nested manual graphical user interface system, and has over two hundred thousand lines of code written in C, FORTRAN, shell, and Java.

Domain Analysis

We present these examples as illustrations of the situation of the domain as a whole - they are not unique outliers, but instead are rather typical of many of the legacy systems currently in operation. They are the product of a truism of traditionally patterned climate data software, which operates in a fast paced and relatively new field, bases itself on an enormous breadth and depth of observational physical measurements and novel research, and relies on an even faster paced and newer field (computer science) for its expression.

Any system, natural or derived, which is a function of many unstable variables, is itself already in an unstable equilibrium, and can eventually spin out of control. Our legacy systems are an excellent example of this concept - with so many variables, iterations, technologies, authors, etc. these systems are born from an unstable condition, and over time will be pushed out of equilibrium. In practical domain specific terms this means that these systems naturally evolve to be unwieldy, hard to maintain, obtuse, poorly documented, and expensive to develop or repair.

When this non-equilibrium eventually manifests, the system owner generally has two options - repair or replacement. The option to repair has historically been preferred - not only because of the enormous resource commitments and risks of failure of developing a new system, but also because

of the qualities inherent to the climate data domain itself; it is not likely that the domain can or will stop relying on physical measurements, research efforts, collaborations, and the tools of computer science. Therefore, efforts to replace an existing system will rely on the same development conditions of the original, eventually returning to the same condition as the system it is intended to replace - a zero sum game.

While not equivalent in scope to a complete replacement effort, repair efforts of mission-critical data processing systems are themselves not trivial. These efforts traditionally include large scale project refactoring of existing code to conform to present software languages and practices in an effort to return the system to an equilibrium or system-normal state. This usually requires careful collaboration and coordination between experts in the project intent, experts in the software languages and practices currently employed, and experts in current software practices. In the case of mission-critical software, which produces data products that others rely on, it also involves providing a drop in replacement that must exactly replicate the expected behaviors of the original system - a system that may also be changing as the repairs are being made.

Problem Specification

At a fundamental level the instabilities of a system are a product of the system's heterogeneity. This is why the focus of repair efforts are essentially an attempt to reduce system heterogeneity in all project facets - this is accomplished with the introduction of version control, logging, and error handling systems, promotion of comprehensive documentation, simplification of deployment, and normalization of the codebase in both language and structure - the end goal being a revitalized, more easily managed and well behaved software system that can be maintained and built upon for an extended period of time, until the next repair.

While the goal is sound, challenges and risks quickly arise partially as a

matter of unknowns - how one function depends on another, how one module or programming language manages error handling (either handled or unhandled), how system parts handle logging. These behaviors may all be tightly coupled, requiring the entire system to be unwound at once, like a single ball of yarn. Additionally, refactoring a system might force the system into design patterns which conflict with the actual intent of the original software - moving certain languages into another might make a system more stable, but at the expense of desired CPU speed, or the introduction of a language which is not suitable for continued ease of development by the researchers who will be further developing the software.

Non-Invasive Re-Architecture

Many of these same concerns were raised while our team began developing refactoring plans for the ASOS ingest system. Realizing their broad domain prevalence, we were motivated to create a solution that could be used for this project, other existing projects that currently face the same issues, and new projects which are predicted to face these issues in the future, designing a new, general method for system development. Focusing on an architecture-first approach, this method treats a target system as a conglomerate of discrete units that interact functionally and share state in an external framework, rather than as continuous singular entity that manages state internally. This approach allows individual components to retain their essential quality while simultaneously providing them with the easily accessible behaviors characteristic of well behaved systems - centralized logging, robust error handling and recovery, simple deployment, and automatic parallelization. While nothing comes ‘for free’, we believe this approach does provide extraordinary return on investment.

Our development approach is characterized by four tenets:

- A functional-architecture framework
- An external computational workflow engine
- A specification driven, centralized control system
- A language agnostic task API

Functional Framework

A popular way of thinking and designing climatological software is as a single logical instruction set, possibly containing branches, that is provided a set of runtime properties which it uses to execute a long series of instructions, eventually producing some final output. In this pattern the logic that happens in between endpoints is generally considered as a single black box. In practice, this manifests in many modules written in a variety of languages, which are connected loosely with systems languages (such as bash) with little thought to code structure. This pattern promotes an unstructured impulsive style of development that can easily become unclear how things are glued together, where something might be failing, or where an operation might be operating incorrectly.

To correct this, our approach takes each module and defines it as a function - clearly defining inputs, outputs, where something comes from, and where it goes. We actually define three layers of functional behavior - a module level, which consists of a single module, its main function, and its inputs and outputs; a unit level, acting as a logical container of modules that consumes and produces meaningful data; and a job level, which acts as a top level container, treating units as part of a complete pipeline. This mental framework of project structure provides a clear path for mapping existing code, which may not be originally patterned uniformly, into one that is.

Workflow Engine

Once the original system is arranged into a functional framework, that framework is formalized as a topology and then fit into a workflow engine. The workflow engine is designed as distributed computational engine that can run user code, scale horizontally and vertically, and automatically provide the features of parallelization, task branching, central logging, and error recovery uniformly across job components.

In our system, the workflow engine runs job topologies, which as we described earlier, are collections of functional units arranged in a logical pipeline. To the workflow engine, each job unit is considered a single task, and tasks can be effortlessly branched in an acyclic digraph pattern within the job specification - I.e., job task 1 can provide output to both job task 2 and job task 3, which will each then progress on their own respective branch. This pattern of moving branching behavior from inside a code unit to the workflow engine automatically adds stability and error recovery to the system, while also providing a stereotyped way of handling flow behavior.

As the workflow engine runs a job and a component task becomes ready for execution, it is provided to the workflow engine queue, along with a set of inputs. The inputs are automatically parallelized so that a task-input pair will be passed on to the next virtual peer available for a given peer. Workflow peers are based on physical servers, of which there may be an arbitrary number depending on the horizontal scale needs of a particular system, and virtual peers are based on the available CPU resources of a given peer. This method of central distribution ensures that only one virtual peer manages the state of a task running a specific input at a time.

If a task fails, the exact state can be retraced, and error conditions in that case can be specified on a task, module, or job level. The workflow engine also provides output for all tasks, as well as metadata level information for the level specified (debug, info, error, etc.) to a single central log file. This provides a clear record of any issues and outputs for quality control.

Configurable Specification Control

To allow for stereotyped control of workflow engine and job behaviors, our design provides a JSON specification for all system components – all system jobs and the workflow engine can be composed without code using a simple text file containing a specification object. These JSON specs are completely configurable, allowing system operators or developers to change how a job runs or how a workflow engine manages its log or error handling behaviors without writing or modifying any code, dynamically and in real-time. JSON specification files are submitted to a running workflow engine using a command-line utility and a simple service over telnet. Job specifications are also where the initial task inputs are be specified - the system can use directly specified input or parse any file or directory tree to create inputs, reading files into records with headers. Files that are parsed across records or directories that are parsed across filenames automatically split each into its own input, which provides parallelized input to job tasks. Data parsing could easily be extended to use any type of database or distributed file system as a source of input data. A typical job specification is shown in figure 1.

```
{ "type": "async",
   "data": {"type": "directory",
            "source": "records",
            "task": "in",
            "path": "/opt/resources/data/dir/",
            "extensions": ["txt", "md"],
            "recursive": "true",
            "headers": "true",
            "delimiter": ","},
    "tasks": [{"name": "in",
                 "type": "input"}
               {"name": "get-files",
                "type": "function",
                "language": "java",
                "class": "gov.asos.ingest.content.GrabFiles",
                "constructor": {}}
               {"name": "file-output",
                "type": "output"}],
    "workflow": [["in", "get-files"]
                 ["get-files", "file-output"]]}
```

Figure 1: A typical job specification map. Jobs must contain a type, which determines the message processor; a set of tasks, which themselves contain information on the type or the base class, along with constructor parameters; a workflow, which specifies how tasks flow between each other; and a data parameter, which tells the job how to assemble initial inputs.

Task API

To run a job's tasks (which may be written in any number of languages) while sharing input and output data between them, we provide a task API that all job tasks must implement. This API is written in Java, the language we have developed our workflow engine to use. The workflow engine then dynamically loads tasks at run-time using their specification in the parent job map.

To allow other programming languages to seamlessly interact with Java, the task API uses the Java Native Interface (JNI) to provide the ability to wrap to any native, C based language (such as Fortran or C++) within a task. All tasks expose a single run method, which has a single requirement to consume and produce a map. The map itself represents a single input or data unit. Within the run method itself, the map can be used in any manner necessary, and because anything may be stored in the map, this system provides no obfuscation of existing behavior.

The task API also provides all tasks with simple debug, error, info, and log level methods, as well as the ability to kill an entire job, or a single task thread from within a task. All task log information or error information is stored in the same centralized and immutable log, which belongs to the workflow.

Summary

Together the four tenets of the framework described form a novel approach to system design in a problem space for which it is generally difficult to develop robust and agile software. Existing code bases can gain enormous benefit from its implementation, while new software projects could easily use it from the start. It fits well in the applied climatological domain. Its use does not preclude the use of other best practices - instead it complements them, allowing for targeted refactoring in specific units when problems are apparent. Its approach to functional design also promotes a clearly defined mental model of system behavior, which may provide an anchor for evolving teams over time, preventing many of the issues, which then necessarily must be solved by software repair.