S4P Programmer's Guide

A guide for programmers of the Simple, Scalable, Script-Based, Science Processor (S4P)

November 2005

Document Version 1.0.0

Chris Lynnes, NASA

Table of Contents

1. Introduction

Data processing costs too much. This fact was brought home during the development of the systems to process data from the Terra platform. Integration of the science algorithms has turned out to be more difficult than expected, leading to higher costs, reduced capability and schedule slips. This has motivated an evolution in EOSDIS toward Science Investigator-led Processing Systems (SIPS). In mid-1998, the Goddard Earth Sciences DAAC began developing a simplified processing system as a risk mitigation and a potential resource for future SIPS.

Many science processing systems have simply grown up around the algorithms they run. Although simple and robust, they often are specific to the algorithm. On the other hand, the EOSDIS Core System was designed to be general, resulting in a large, complex mix of commercial and custom software. Recent successful large systems include the SeaWiFS Data Processing System and the TRMM Science and Data Information System. While developed for specific disciplines, they are in fact relatively easy to generalize to other algorithms. One thing that these have in common is the use of commercial databases (often Sybase), and in most cases commercial system management tools (e.g. AutoSys).

In contrast, many simpler systems, such as the EROS Data Center AVHRR 1KM system, rely on a simple directory structure to drive processing, with directories representing different stages of production. The system passes input data to a directory, and the output data is placed in a "downstream" directory.

The GES DAAC's Simple Scalable Script-based Science Processing System (S4P) is based on the latter concept, but with modifications to allow varied science algorithms and improve portability. It uses a factory assembly line paradigm: when work orders arrive at a station, an executable is run, and output work orders are sent to downstream stations. This document is targeted at system designers who are planning or considering the use of S4P to implement an automated processing system. It comprises:

- A description of the basic philosophy
- A description of the overall architecture
- How to apply S4P to automated science processing
- A "getting started" guide
- How to design S4P stations
- Descriptions of existing S4P systems
- Upcoming features

1.1 Key Premises

Following are key premises that went into the design of S4P:

- Designing a *simple* system for processing science data is not rocket science.
- Most of the operability and performance gains targeted by fancy processing systems are not realized due to the complexity of the resulting system.
- More Code = More Bugs.
- A lot of time, money and smarts went into developing the operating system you're running on, so think twice before reinventing a facility that is already present (or mostly present) in the operating system.
- Most of the human effort expended in an automated processing system is spent troubleshooting; hence a system should be "Designed for Trouble".
- Commodity resources (disk and CPU) have decreased in cost to a point where they can compensate for less than optimized usage. Currently, 1 staff-year = ~0.5 TB disk or 4 GFLOPS.
- If you have enough extra resources, you don't have to worry so much about resource management, scheduling, or even planning. You just process when the input data come in.

1.2 Design Goals

Based on the above premises, the following design goals were used in development:

- Simplicity: Above all, the system is designed to be simple. This translates to a number of tactics:

    1. Using an interpreted language (Perl)
    2. Limiting the variety of design entities (objects, interfaces, etc.)
    3. Limiting the amount of code.
    4. Using the operating system wherever possible.

- Transparency: The outcome of the Design For Trouble principle is a system that is transparent to the operators. That is to say that an any given time, an operator can determine where problems have appeared in the system and can easily drill down to find the source of the trouble. It also means that all information passing through interfaces is accessible to the operator (hence an emphasis on passing information as files rather than inter-process communications.)

1.3 Related Information

S4PM is the biggest application of S4P for which documentation available at http://disc.gsfc.nasa.gov/techlab/s4pm/.

2. Architectural Overview

2.1 Assembly-Line Paradigm

The process of using science algorithms to create products is modeled as a factory assembly line, with a number of **stations** along the way. Two key improvements on a real factory are: (1) multiple instances of a job can be executed simultaneously at a given station, and (2) the output can be sent to more than one downstream station simultaneously.

Simply stated, a working station is a Unix or Windows directory with 2 key components:

1. A configuration file (normally `station.cfg`). The two essential items in the configuration file are a map of tasks to execute on arrival of input **work orders,** and a map of downstream stations to which output work orders are sent.
2. A daemon monitoring the station for input work orders. A program called **Stationmaster** (actually, `stationmaster.pl`) is provided for this purpose; all that need be done is to start up a Stationmaster for each station.

Once the set of stations has been established and configured, the system is started by running Stationmaster in each station of the system. When the first input work order is deposited in the upstream station, the Stationmaster does the following:

1. Detects the work order
2. Looks up the appropriate executable for that work order type
3. Makes a temporary subdirectory for executing the job
4. Changes directory into the temporary subdirectory
5. Forks off a Stationmaster process to run the executable
6. When done, it looks up the output work orders in the downstream map
7. Sends work orders to the downstream stations

The state of the system at any time is visible in three entities:

Entity	Description
Station configuration file	Describes what is run for each work order type
Input work order	Describes the input for the job
Log files	Files recording the output of Stationmaster (at the station level) and the specific executables (in the job subdirectory)

Table 2-1. Describes the three entiries involved in maintaining the system state.

2.2 Monitoring an S4P System

Monitoring is done by watching jobs appear and disappear from a given station, or conversely finding which station (directory) a job appears in, both of which are simple

tasks in the Unix or Windows system. A job log is passed along with the work order and appended at each station, providing a "drill-down" or thread-tracing capability as well.

A simple graphical user interface (`tkstat.pl`, written in Perl/Tk) is provided for this monitoring. It simply displays the number of running and failed jobs in a station (identified by the name of the job subdirectory), as well as whether a station is currently "up", i.e., has a Stationmaster daemon running. An additional program, `tkjob.pl`, is invoked by `tkstat.pl` to drilldown to the station or job level, allowing the operator to examine configuration and log files, or even to start or stop stations.

3. Application to Automated Science Processing

Many science processing systems are data-driven, in the sense that incoming data triggers the processing to be done. An example of this is the classic AVHRR 1 km system at EDC. The alternative paradigm is that of planned processing, where plans of the processing to be done are constructed before data arrival. The EOSDIS Core System exemplifies this approach. S4P is an example of the data-driven type (and is indeed loosely based on the AVHRR system in concept.)

3.1 Simple Processing

Let us assume that an external data source is pushing data files to us. We will then process each data file through an algorithm that produces a single output data file for each input. We then process the output data file through a second algorithm which produces a single output file for each input, and finish by archiving the second output. Such a system might be implemented with only 4 stations:

1. Import Data: Detects arrival of input data, passes the full pathname on to Run Algorithm #1.
2. Run Algorithm #1: Runs the first algorithm, deletes the input data, passes the pathname of the output file to Run Algorithm #2.
3. Run Algorithm #2: Runs the second algorithm, deletes the input data, and passes the full pathname of the output file to Archive Data:
4. Archive Data: Saves the data to archive.

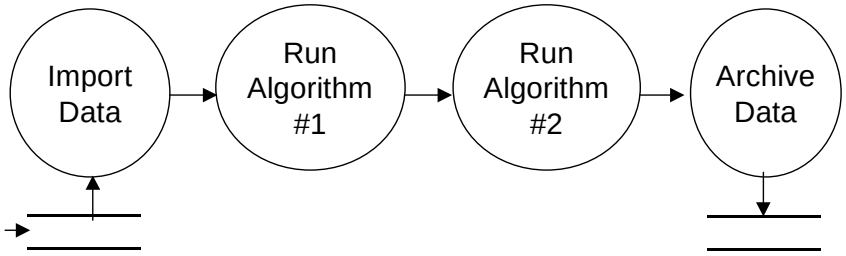

Figure 3-1. Simple processing scenario.

This system assumes very simple production rules (*i.e.,* one file in, one file out), and no resource contention (*i.e.,* more than enough disk space to go around.

3.2 Dealing with Complicated Production Rules

In practice, production rules can be quite complicated: ancillary files may be optional or required; input files may be processed in groups; adjacent files may be needed for averaging, and so on. In fact, the variety of production rules is potentially boundless. As a result, such rules are not supplied as an intrinsic part of S4P. Instead, they are implemented as standalone stations, sometimes called triggering stations (Figure 3-2). In

this case, the triggering station collects information about all of the input files needed and passes that information to the algorithm run station. As such, the program run in the triggering station can be arbitrarily complicated and written in any language. It needs to fill only two criteria:

1. It is a standalone executable program (*i.e.*, binary or script).
2. It outputs a file intelligible to the corresponding Run station.

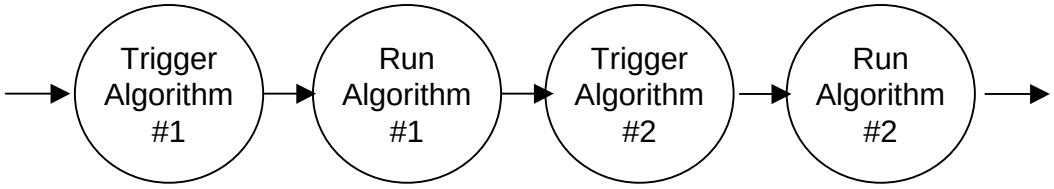

Figure 3-2. Triggering stations.

Of course, the logic in the Trigger station could be prepended to that in the corresponding Run station, thus halving the number of stations needed for the purpose. However, there are two advantages to a separate Trigger station when the rules are complicated:

1. The complicated triggering logic is separated from the algorithm, producing a more modular, maintainable system. Indeed, it can even be developed separately from the main algorithm stations.
2. The triggering can be monitored separately from the algorithm execution, which is helpful because the types of errors occurring in each one tend to be quite different.

3.3 File Tracking

Most complicated production rules tend to deal with the use of multiple files, sometimes of different data types, but also occasionally groups of files of the same data type. For instance, the MODIS Level 1B (Calibration) algorithm processes an input Level 1A file using the adjacent (leading and trailing) Level 1A files to average the calibration information, as well as the corresponding Geolocation file. For such cases, it is also important to track what files are available and whether they have been used for all of there intended purposes.

File tracking can be implemented in a number of ways. The first main category is the use of a database, either relational, DBM or otherwise. In this case, stations which make or bring in data populate the database, while Triggering stations query it for available data (Figure 3-3.)

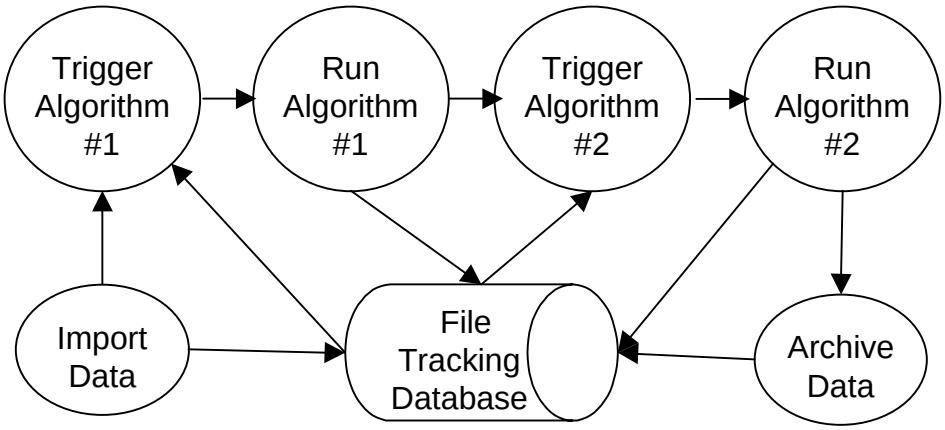

Figure 3-3. Database implementation of file tracking.

Alternatively, a station can be used to track the files in the system. In this case, all file creations or updates generate work orders which are sent to the granule station. "Queries" are accomplished by sending work orders with placeholders to the tracking station, which fills them in with the necessary file pathnames.

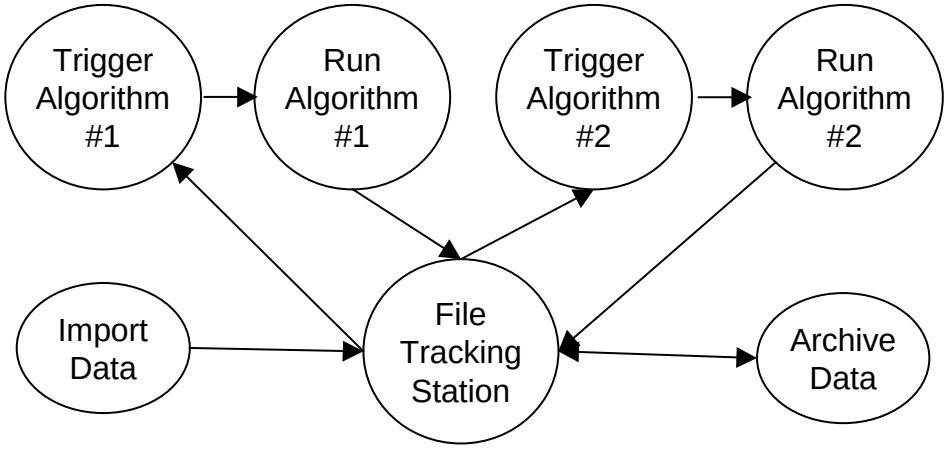

Figure 3-4. Station implementation of file tracking.

4. Getting Started

4.1 Step 1: Create your S4P directory structure.

Just use the `mkdir` command in either Unix or Windows to do this. It's a good idea (though not essential) to create all of the directories in a subdirectory of their own. Don't worry about the space used by these; the data that are processed need not be on the same disk, though it should be accessible from this area.

In designing your directory structure, you will need to make decisions about "lumping" tasks vs. "splitting" tasks. It can be simpler (especially in the beginning) to lump tasks together into a small number of stations (e.g., by having a script run them in sequence). On the other hand, this limits your ability to see what is going on within the "lump", as the monitoring granularity is at the station level. However, it makes sense to lump two tasks when one of those is a highly routine operation that is unlikely to break down.

If you have a Data Flow Diagram (DFD) or Object diagram describing your processing system, it often makes sense to have one station per process or object.

4.2 Create your station configuration files.

The station configuration file lives in the station directory. The default name for the file is `station.cfg`, although the Stationmaster can be brought up with different file names using the command line arguments (`man stationmaster` for more details).

A) Work order / executable map

This is a Perl hash (associative array) that maps each job_type to an executable to be executed when a work order of that type is received. Normally the job_type is the part of the input work order filename that comes after the "DO." Prefix. This allows several different kinds of work orders to be processed in the same station.

The executable can be either a binary executable or a script. (Actually, it can even be a Perl Module subroutine.) It can also have static arguments. The one restriction is that the input work order file will be passed to the executable as the last argument, so your executable or script must be prepared to process it as such. The work order file itself can be any format that the executable can understand. It can even be the data file itself (or a symbolic link) for simple one-in, one-out processing cases. The chapter below on Designing Stations and Systems gives more details on the various permutations.

B) Downstream work order map (**%cfg_downstream**)

This is a somewhat complicated Perl hash, as it maps output work order job_types to a list of downstream stations.

C) Other configuration parameters

$cfg_root: the root directory for the downstream stations. This allows you to specify only the last part of the directory pathname in your downstream work order map

$cfg_max_children: maximum number of child processes Stationmaster can fork off. This number is the sum of running and failed processes. A station can be made single-threaded by setting this to 0 (no children) or 1, which is useful for queuing jobs. (Only 0 is currently supported on Windows.)

Here's an example from the S4P system processing MODIS Direct Broadcast data:

```
$cfg_station_name = "Run L1A";
$cfg_root = '/usr/modis/db/s4p/';
%cfg_commands = ('MOD00' => 'perl ../run_pge.pl -l 150 -a 1 -o
/usr/modis/db/data/1');
%cfg_downstream = ('MOD01' => ['mod_pr03']);
```

4.3 Start up Stationmaster daemons.

You can do this manually, by running stationmaster.pl (from the command line or in a script), or using the tkstat.pl monitor, by clicking the button for a station (a red background means it is not currently running), and clicking the Start button in the resulting popup screen for that station. Note however, that the person who starts up a Stationmaster is the only one who can stop it (aside from the system adminstrators).

4.4 Seed upstream stations with work orders.

There are three ways of doing this:

1. For data-driven processes, an S4P station can often be set up for the directory where the data (or some signal file or delivery record) come into the system. This often involves a special setting in the configuration file to recognize these "foreign" work orders, as they do not follow the normal DO.<job_type>.<job_id>.wo pattern.
2. An alternative for data driven processes is to have a program outside of S4P detect the arrival of data and create a work order for the first upstream station. Such a program may be a run via cron job, triggered by email filter, or forked off by inetd.
3. For ad hoc processing, such as reprocessing, a user interface is used to select the data input for the processing stream and then output the work orders to start the stream processing.

5. Designing Stations and Systems

This section discusses the various options for setting up stations, along with some simple examples.

5.1 Simple Filters

Perhaps the simplest station is a filter, that is a program that takes a single input file (usually from standard input) and produces a single output file (usually standard output). An example is the Unix **sed** command. The syntax from the command line is usually: `command < input > output.` In an S4P station, however, we anticipate running the same command many times, with different input files, and thus different output files. Therefore, in addition to invoking the command, we must be able to generate the output filename automatically, using say, the time or process_id, or more commonly, a permutation of the input name. One way to do this is to wrap the command in a shell or Perl script.

The following example, lets call it **run_foobar.pl**, runs the command **foobar**, using input files with the pattern `INPUT_xxx.dat` and producing output files with the pattern `OUTPUT_xxx.dat`.

```
#!/usr/bin/perl
$infile = $ARGV[0];
$outfile = $infile;
$outfile =~ s/INPUT/OUTPUT/;
system("foobar < $infile > $outfile");
```

The station configuration file (**station.cfg**) would thus have the following line:

```
%cfg_commands = ('INPUT*.dat'=>'../run_foobar.pl');
```

In this case, **run_foobar.pl** is assumed to be in the station directory. The `'..//'` prefix is used because each individual job will be run in a subdirectory created specifically for that job.

This type of script can be used for a number of basic processing setups of the filter type. Note that static arguments could be added to the **foobar** command simply by adding them to the script. For example:

```
system ("foobar –r < $infile > $outfile");
```

5.2 Simple Process Control Files

While simple filters are adequate for many cases, most science algorithms have too many tunable parameters to run with as a simple filter. Often, this problem is solved by process

control files. These can vary from simple parameter=value files to large, complex files with specialized syntax, like the ECS Process Control Files. One way to deal with these is to generate a process control template, with all parameters filled in except the input file and output file. Placeholders for the input file and output files (e.g. INSERT_INPUT_FILE_HERE and INSERT_OUTPUT_FILE_HERE) are then replaced by the station's script on execution:

```
#!/usr/bin/perl
$template_file = '../pcf_template';  # Template is in station directory
open TEMPLATE, $template_file or die "Cannot open $template_file: $!";
open PCF, '>temporary.pcf' or
    die "Cannot open temporary.pcf for writing: $!";
# Generate output filename
$output_file = $ARGV[0];
$output_file =~ s/INPUT/OUTPUT/; # Assumes input file is INPUT_xxx.dat
while (<TEMPLATE>) {
    # Input file is the argument to this script
    s/INSERT_INPUT_FILE_HERE/$ARGV[0]/;
    s/INSERT_OUTPUT_FILE_HERE/$output_file/;
    print PCF;
}
close TEMPLATE;
close PCF;
system("foobar.pl temporary.pcf");
```

5.3 Complex Production Rules

S4P does not have complex production rules built in. However, production rules of arbitrary complexity can be implemented through the use of a triggering station. This is a station whose sole purpose is to create a Process Control File, with all the necessary input and output files filled in. Such a script may access databases or perform complex calculations. Indeed, it may even be an executable binary compiled from C or Fortran. The key criteria for generating such a script are:

(1) It executes using a file as input (i.e., the input work order)
(2) It generates a full-featured process control file, all the information needed for the science algorithm to run
(3) It is self contained, in that with the information in (1) it can generate the process control file on its own

6. S4P Applications

S4P is currently being used for a number of applications at the GES DAAC. These are described below.

6.1 MODIS Direct Broadcast Processing

S4P is used routinely to process the data from MODIS Direct Broadcast and SeaWiFS HRPT at the University of South Florida. The jobs are run on a Linux cluster for maximum speed, with an additional station, the S4P Router, to handle dynamic load balancing among the nodes.

6.2 TRMM Data Mining, Near Archive Data Mining

Both of these Data Mining systems seek to run user-submitted algorithms at the archive in order to reduce the volume of data that needs to be sent to the user. S4P is used as the underlying structure to implement this in an automated fashion.

Note that data mining is currently being subsumed into S4PM (see Section 6.4).

6.3 DAAC Processing

S4P is the core of S4P-Measurements (S4PM) which is used for all processing (both routine and on-demand) at the GES DAAC. It is currently the most complex S4P application.

7. S4P Features

Below are listed a number of features of the S4P system. Some already exist. Others, identified by asterisks, are in implementation or planned to begin shortly.

7.1 Stationmaster

Stationmaster is the linchpin of the system. It has a number of configurable features to provide the necessary flexibility to construct any arbitrary processing system.

- Adjustable polling interval
- Run different executables based on job type
- Child process limits
  - Total number of running and failed children
  - Number of running children*
  - Number of failed children (will stop forking jobs when too high)*
- Send output work orders to multiple downstream stations
- Send output work orders to different downstream stations based on output job type
- Run a user-defined failure analysis program on job failure, based on input job type*
- Maintain logs of the full history of a given work order (including logs for upstream stations)
- Job sorting and prioritization:
  - By work order name
  - By arrival time (FIFO)
  - By input job type

7.2 Monitoring Stations

Two tools, **tkstat** and **tkjob** work together to monitor a system of stations and control individual stations and jobs.

- Monitor a group of stations for running, failed and pending jobs
- Start and stop a station
- Job Control
  - Terminate a job
  - Suspend a job
  - Resume a job
  - Restart a failed job

In addition, station-specific scripts of various types can be developed and hooked into the **tkstat/tkjob** interface. This is done simply by associating a Button name with a command-line executable in the **station.cfg** file.

- Failure Handlers work with failed job directories, providing a way to recover from or otherwise handle a failure case. This is particularly useful for specific cleanup tasks.
- Manual Overrides work with currently running directories. These are used, for example, to release jobs that are waiting for something. (Note that this requires the scripts to do some signaling to each other in the job directory.)
- Station-specific interfaces can be attached to any station, allowing access, for example to additional GUIs.

7.2 Resource Allocation*

The TRMM Data Mining application is currently using a Resource Pool implementation of this feature. Although used there for disk allocation, it can be used for other resources once it is integrated back into the main S4P baseline.

Appendix A. Acronyms

Acronym	Meaning
AIRS	Atmospheric Infrared Sounder
AVHRR	Advanced Very High-Resolution Radiometer
CPU	Central Processing Unit
DAAC	Distributed Active Archive Center
DFD	Data Flow Diagram
ECS	EOSDIS Core System
EOSDIS	Earth Observing System Data Information System
GDAAC	GES Distributed Active Archive Center
GES	Goddard Earth Sciences
GFLOPS	Giga FLOPS (Floating Operations Per Second)
GSFC	Goddard Space Flight Center
GUI	Graphical User Interface
HRPT	High Resolution Picture Transmission (AVHRR)
MODIS	Moderate Resolution Imaging Spectroradiometer
S4P	Simple, Scalable, Script-Based, Science Processor
SeaWiFS	Sea-viewing Wide Field-of-view Sensor
SIPS	Science Investigator-Led Processing System
TB	Terrabytes
TRMM	Tropical Rainfall Measuring Mission