On-demand processing architecture for radar image products

Markus Peura*
Finnish Meteorological Institute, Development of Services

Abstract

In this paper, we present an open, light-weight production architecture that supports on-demand product generation and easy import of external code (plug-in executables). The solution is based on systematic naming policy and smart chaining of the executables.

1 Motivation

Need of control. Typically, the design of a radar data processing system is largely determined by an all-in-one software package that handles "everything" from the measurement tasks down to colourful products on end-user screens. Such solutions are sometimes problematic because mostly there is a surrounding larger system, "the main system", for which the radar data processing system should serve more like a product generator module submitted to external control, requests and data input.

On-demand product generation. Climatological services as well as many research and development tasks appreciate production systems in which products based on older products can be retrieved. Because the diversity of products available for operational meteorogical services is often large, it is not possible to archive all the products. Practically, this suggests that only raw data be archived and the system is build such that any product can be (re-)generated *on demand*.

Operational implementation and interchange of algorithms. In research and development stage, one often carries out involved computations and other experiments which are impossible with the operational software. (This is actually logical, by the definition of R&D.) Hence, this work gets done by programming experimental code and/or by applying research oriented software such as statistical toolkits. Sooner or later in international or in-house cooperation there is a need to transfer the findings to an operational system. For example in Europe, there are already many examples of common efforts on sharing the development work in radar algorithms [2] and technical processing [1].

Practically, one has three choices in transferring the result – say, an algorithm – towards operational use: Firstly, there is the literal approach: communicating a result in a technical report or scientific publication – the system owner or commercial manufacturer will eventually implement it in the software.[1] Second, one may consider code embedding: wrapping the original ad-hoc executables in a technical form (interface) supported by the operational system. Finally, a conformal approach: the R&D work applies the tools and modules on which also the operational system is based; this guarantees compatibility but may decrease flexibility.

In this paper, we will focus on the embedding approach and the conformal approach.

2 Outline of the proposed system

2.1 General properties

Next, we explain the principles and technical building blocks of a system that meets the needs described above. In fact, we wish to emphasize the involved ideas and concepts rather than our implementations (a prototype in PHP, another in Java). The amount of code required to implement the proposed system is anyway fairly small; the system could be implemented quickly in any computer language supporting string parsing and system calls (shell invocations). Further, many details presented in this paper should be seen more as suggestions of design rather than critical properties of the system.

The main features of the proposed system are:

- The architecture is truly open, and supports user's extensions (plug-ins) developed in any language capable of reading environment variables and supporting file input-output.
- The system expects that it is possible and natural to store any queried product as a file (rather than as a database entry, for example); hence this system does not fluently support querying single numbers or strings.
- Product generators (scripts or binaries) are maintained in a systematically organized directory tree; this structure by nature supports dividing human work into teams (with own file permissions etc).

*Corresponding address: `Firstname.Lastname@fmi.fi`

[1] This paper is an example of a literal approach aiming at transferring an "algorithm" to operational systems.

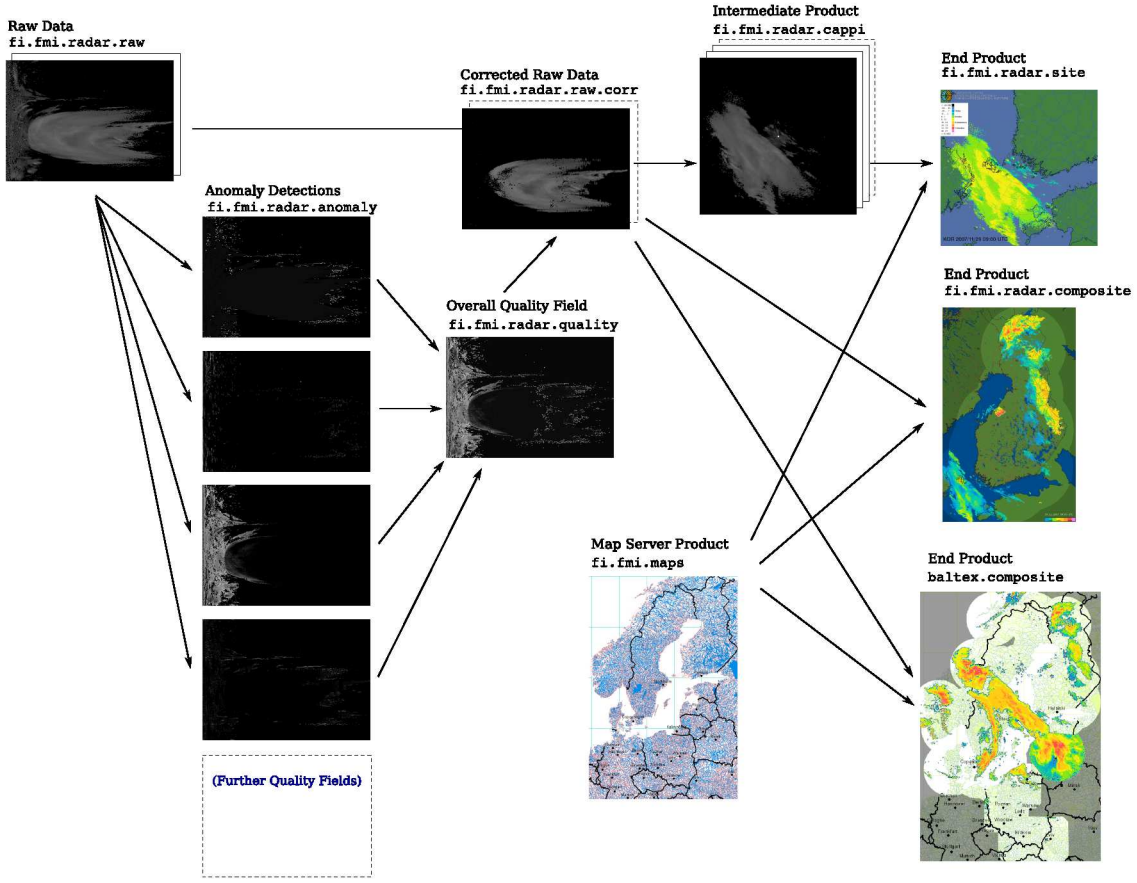

Figure 1: Examples of dependencies between end-products and measurements (raw data). In a conventional "push" system, incoming data triggers processing a series of subsequent products. Re-generating old products can be tricky or impossible. In "pull" technology, a product is requested by calling it directly, and the system generates the required "ingredients", ie. intermediate products.

- The structure of the directory containing the product generators affects directly the file naming as well as cache directory structure.
- Product file caching is an intrinsic property of the system.
- Systematic and flexible disk cache cleanup is supported.
- System users are free in designing the generator directory structure, but it is recommended to let a *hierarchy of producers* to guide the design, for example: country/institute/device/quantity.
- The system supports on-demand (pull) generation of any product for which the original data is available (archived or freshly observed);

2.2 General scheme

Next, we describe briefly the behaviour of the proposed system. In brief, a product query consists of a *product request* supplied by a client (a human user or an application) and a *response* of a server. The request is essentially a character string. The response means returning the product directly as a data stream or indirectly with a reference to it's location (a local file path or a URL). More specifically:

1. The client issues the request.
2. The server parses (analyses) the request and derives the product to be retrieved.
3. If the desired product file exists in the cache, the system returns it to the user.
   If not, the system tries to generate it by calling a product generator, providing it with all the parsed variables including (among others) a timestamp, product parameters and target file path.
4. The plug-in tries to generate the desired file, possibly involving callbacks to the system for generating the products it needs as input, recursively. In this sense, the system resembles *makefiles* applied in program code compilation.
5. The plug-in stores the resulting file in a location given by the main system. The execution returns to the system.
6. The system checks if the plug-in has generated the file. On success, it returns the file to the client. On failure, communicates the diagnostics in a preferred way (eg. error message or log entry).

2.3 Variables

A product request is essentially a character string that expresses the client's need through a set of *query variables*. There are *main variables* which are always required or automatically derived by the system, as well as *product-specific variables*.

The only compulsory main variable is the product id, `$PRODUCT`. Another important main variable is the time, given as `$TIMESTAMP` with format `YYYYMMDDhhmm`, by default the current time. (Static products like maps do not require it.) As the system is based on creating, storing and transferring files, each product has some default file format (`$FORMAT`). The product-specific variables or *product parameters* can be supplied like the main variables.

For each query, the system does some parsing with the variables. The parsing scheme is described in Fig. 2. The resulting set of variables uniquely define the product (file) to be retrieved. Practically, this means there are alternative, convenient ways to set variables. For example, instead of using `$TIMESTAMP`, a user may use (some of) variables `$YEAR`, `$MONTH`, `$DAY`, `$HOUR`, and `$MINUTE`. Further, instead of giving a set of variables one may always give just the desired filename, `$FILE` – the system will derive the other variables from that. (This feature is especially handy in http query mode, see Sec. 2.5.) Inversely, this requires the system to have a filename syntax capable of parsing `$PRODUCT`, `$TIMESTAMP`, `$FORMAT` and product parameters from `$FILE`.

2.4 Product hierarchy

Instead of putting plug-ins to a common directory, it is handy to set up a subdirectory for each plug-in and its configuration files etc. Moreover, as groups of related plug-ins tend to have common developer groups, configuration files and other shared properties, it is wise to construct subdirectories hierarchically. Due to this "production management aspect", we recommend applying *producer based directory hierarchy* instead of hierarchies based on measurement quantity (say, wind) or application (say, www). For example, a product generator developed at FMI for computing a CAPPI image could reside in a generator directory `fi/fmi/radar/cappi`. The actual executable residing in the directory has a standard filename, say `generator.sh` (in case of a UNIX script), `generator.php` (PHP), `generator.py` (Python) or `Generator.class` (Java), and so on. The generator directory uniquely determines the respective product id `$PRODUCT`. We have preferred Java-like convention, converting slashes to periods, yielding `$PRODUCT=fi.fmi.radar.cappi` in the above example. Whatever syntax applied, the critical property is that *given a product id, the system can uniquely determine 1) the cache directory eventually containing the file and 2) the directory of the generator that can (re-)generate that product*. Some examples of product names are given in Table 1.

$PRODUCT	Description
fi.fmi.radar.cappi	FMI single-radar cappi
fi.fmi.radar.composite	Composite image
nordrad.composite.max	Nordic radar network, maximum dbz composite
baltex.raw	Radar data collected in the Baltex research project
baltrad.composite.q	Quality field of the Baltic area composite

Table 1: Examples of hypothetical product id's.

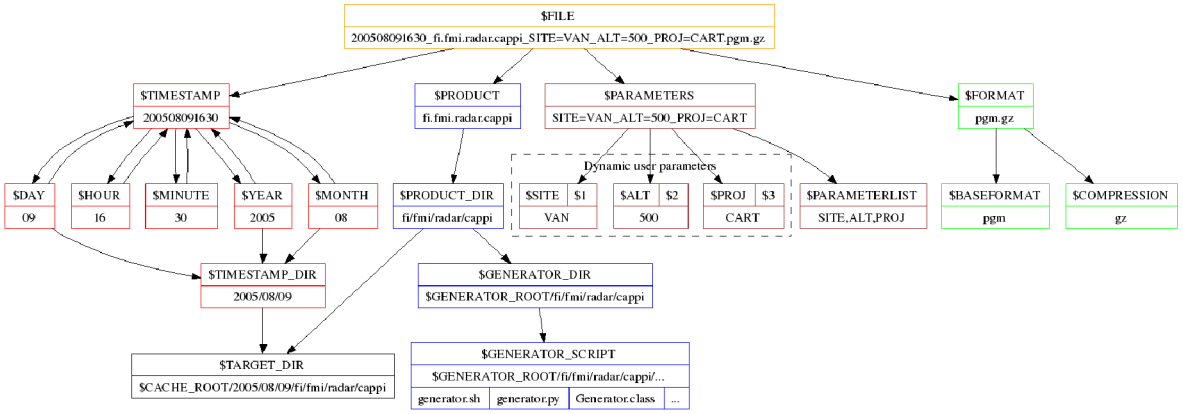

Figure 2: The proposed variable parsing scheme. For example, the user may just give `$FILE`, and all the other variables will be derived from that. If the equality sign is not desired in filenames, one can restrict to using ordered arguments (`$1`, `$2`, `$3`,...) only. Likewise, one may wish to use separators different from ours (underscore, period and comma).

2.5 Usage

We propose that the system has at least two user interfaces: a command-line and an http interface. Examples of usage are shown in Table 2. The simplest version of a command-line interface requires just programming a simple string parser, to be exectued separately for each product request. A more sophisticated solution would be to apply a memory-resident application (daemon) supporting queueing and memory caching in handling subsequent product queries.

Implementing a hypertext transfer protocol (http) interface for querying products requires configuration access to a WWW server. We recommend configuring the server such that the client can see the cache directory system openly: a request for an existent file makes the server return that file – as http does by default – while a request for a nonexistent file triggers a product generator which generates the file (and the subdirectories along the path, if needed).[2] To the client, this appears simply as a delay. After that, the file will be kept in the directory system for some preferred period, and subsequent clients will get it immediately. Hence, the directory serves simultaneously as a query means and a cache. The http interface is also handy because the protocol supports communicating error conditions. In addition, WWW servers provide some standard utilies like user access control as well as usage and error logs. Consider the question: "Which internal and external clients are using this product and when?" On the other hand, one of the limitations of the http protocol is the timeout, typically set to one minute, to prevent the server from jamming under too many cpu-demanding requests. If needed, one can implement an additional interface programming sockets directly, enabling arbitrary design for client-server interactions.

[2] We have implemented this on an Apache server – triggering product generator needs two lines of code in a `.htaccess` file.

3 Hints and concluding remarks

In this paper, we have proposed an open data processing system that is fairly analogous to that of *makefiles* familiar from programming: each product requires certain other products as ingredients, which may require further ingredients, and so on. A major technical difference is that these dependence relations are not stored in a single "recipe" like a makefile, but product-to-product dependencies are distributed in the directory tree containing the product generators.

In this sense, the architecture also resembles the one applied by the geographical information systems (WMS, WFS, WCS) under the Open Geospatial Consortium (OGC) standards. In these systems, servers can be chained together, resulting in *cascading services*.

The described system works well with meteorological radar data but evidently, it would apply to geograph-

A unique description of a product — its filename: `200508091630_fi.fmi.radar.cappi_SITE=VAN_ALT=500.pgm.gz`
Command line usage, user's query: `> appserver -FILE 200508091630_fi.fmi.radar.cappi_SITE=VAN_ALT=500.pgm.gz`
Command line usage, system response (the path of the succesfully generated product file:) `/data/cache/2005/08/09/fi/fmi/radar/cappi/200508091630_fi.fmi.radar.cappi_SITE=VAN_ALT=500.pgm.gz`
Http usage, user's query *and* the path of the succesfully generated product file: `http://products.fmi.fi/query/2005/08/09/fi/fmi/radar/cappi/200508091630_fi.fmi.radar.cappi_SITE=VAN_ALT=500.pgm.gz`

Table 2: Examples of usage.

ical, biological, or medical data processing, for example.

There are some visualization and other image processing operations – like colouring, transparent layering, brightness and contrast control, zooming, animation of image series, format conversions – which could be technically implemented in the product generators but more naturally as centralized services provided by the framework system, letting plug-ins concentrate more on computations requiring substantial expertise.

The proposed system suggests, or actually requires, a proper file naming convention. Alternatively, one may consider a service involving tens of unnamed radar image products, and discussing "that radar image with greenish precpitation" in phone. Anyway, we vote for a systematic, unique, informative file naming convention. For reasons discussed above, we have preferred the hierarchical, alphabetical Java-like convention ("an inverted IP-hostname convention"). Nevertheless, one may still criticize it for producing lengthy filenames. Certainly, instead of the proposed convention one could use numerical coding – like a running index or something like International Serial Book Numbers (ISBN) – but such codes would be cryptic for human communication.

As an additional effect, the proposed naming convention suits nicely to cataloging large sets of products and search operations using pattern matching. One may consider browsing an electronic catalogue containing hundreds of products and picking wind-related products with `*.wind*`, or picking all the raw data available in an international network with `*.radar*raw*`. Also in cache cleanup configuration one can apply similar techniques combined with rules on creation times and/or access times. Further, a system manager may appericiate product dependency graphs supported by the proposed system. For example, one may wish to know how many products will be affected if certain intermediate plug-in would not survive a system upgrade, for example.

Finally, above all, we would like to underline the potential of the proposed system for international exchange of radar algorithms, visualizations and other related code snippets. Too much information remains on literal form – just like this article perhaps – because there is not too many open and flexible technical frameworks for direct exchange of innovations.

Acknowledgement

This study has been supported by the Finnish Funding Agency for Technology and Innovation (TEKES) within AULA/PIPO project.

References

[1] Dawn Harrison, Robert Scovell, Huw Lewis, and Stuart Matthews. The development of the EUMETNET OPERA radar data hub. In *Fourth European Conference on Radar in Meteorology and Hydrology (ERAD2006)*, pages 388–391. Copernicus, September 2006.

[2] Elena Saltikoff, Uta Gjertsen, Daniel Michelson, Iwan Holleman, Jörg Seltmann, Krista Odakivi, Asko Huuskonen, Harri Hohti, Jarmo Koistinen, Heikki Pohjola, and Günter Haase. Radar data quality issues in northern europe. In *Third European Conference on Radar Meteorology (ERAD04)*, pages 212–215. Copernicus Gesellschaft, October 2004.