The BaBar Prompt Reconstruction Manager: A Real Life Example of a Constructive Approach to Software Development.

Francesco Safai Tehrani
for the BaBar Prompt Reconstruction and Computing Groups

Istituto Nazionale di Fisica Nucleare, Sezione di RomaI, P.za A. Moro 2, I-00185 Roma, Italy

Abstract

The need for a working system to automate the Online Prompt Reconstruction (OPR) subsystem has led the members of the OPR group to develop their system using the software cycles development model. Starting with a complete design, they've implemented the system in each cycle using multi-paradigm methods, combining different techniques and languages and enhancing the working system gradually while using it in the production phase. The OPR software has evolved from a set of shell scripts, which were run manually, into a set of partially automated Perl/shell scripts. These will be transformed into a completely automated system based on Perl/C++/Java code with a CORBA distributed architecture, able to execute parallel/multiple PR jobs on a farm of Unix machines.

Keywords: software design, PERL, object oriented, OO, BaBar, CORBA, patterns

1 Introduction: the Prompt Reconstruction system

The Online Prompt Reconstruction (OPR [1]) system is one of the subsystems that is part of the software for the *BaBar* experiment [2]. Its task is to process the incoming data from the DAQ, performing a complete reconstruction so that the results will be immediately available for analysis. It also generates the rolling calibrations, which will be used to adapt the detector's behaviour to optimize the data taking.

The processing phase executes a full reconstruction on the data, storing the results into Objectivity [3]. The data obtained from the detector are stored in a file using a format known as "tagged container" (XTC file). That file is then stored on HPSS, available for any further processing. The file must be staged in on disk before being processed [1].

A farm of Sun Ultra 5 machines is being used, arranged in a star topology. A central machine running a process known as the Logging Manager (LM, see [6]) distributes the events to the remote machines which physically run the reconstruction code. When a remote machine finishes processing an event, it requests a new event from the LM. Each farm machine continues requesting new events until the LM finds the EOF of the XTC file (figure 1).

The policy for the processing of the XTC files has been defined, but it keeps evolving. It is related to the processing rate achieved by the OPR, which is currently (as of December 1999) around 55 Hz, while a rate of 100 Hz is requested to keep up with the DAQ. The reasons for such a difference are numerous but outside the scope of this paper [14].

The experiment's software model requires OPR to automatically process incoming data shortly after they are available. This requires a high level of automation for the whole process. Some parallel control tasks provide for the integrity of the system (performance-wise and from the data storage viewpoint) and for the bookkeeping.

[1] The original policy was for OPR to process the file before storing it on HPSS and we might switch back to that policy in the future.

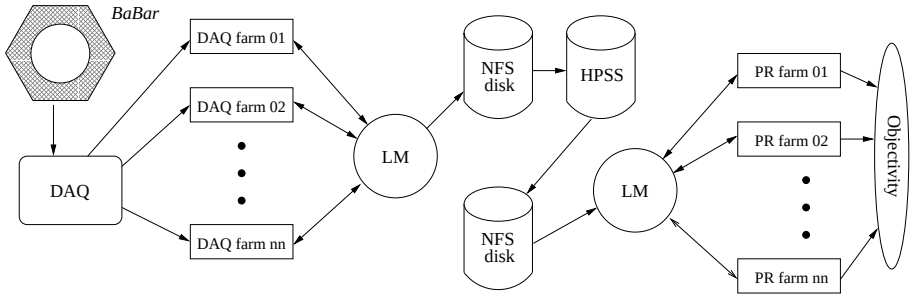

Figure 1: Relations between the LM and the DAQ

A complete design for the OPR system already exists, but the design of some of its subsystems is still evolving. The relationships between the main parts of the OPR system have been designed (see figure 2), but the fine structure is still being refined and developed. The Global Farm Manager (GFM) starts an instance of the PRM (Prompt Reco Manager) for each job. The PRM then starts the Logging Manager and, using the services provided the Global Farm Daemon (GFD) starts the Prompt Reco Daemon (PRD) on each machine of the farm. The PRD starts the Prompt Reco Framework (PRF) which performs the reconstruction writing the results into Objectivity.

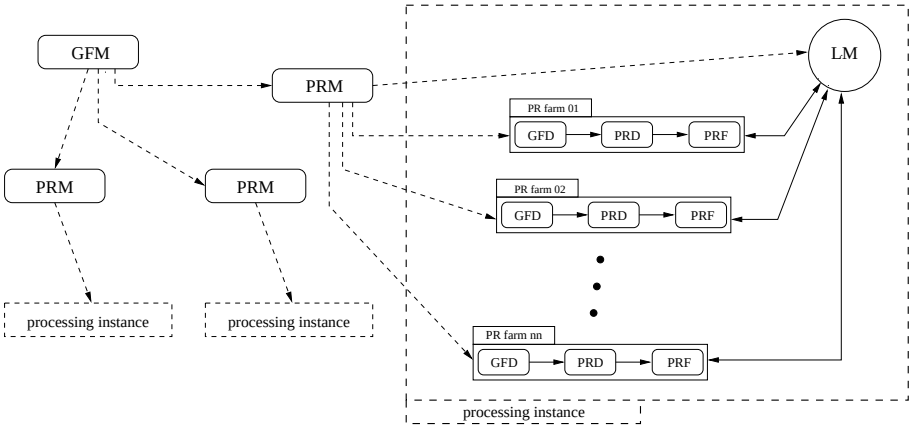

Figure 2: The PR architecture

The PRM package is the part that manages the OPR processing, scheduling jobs and checking their execution. It is currently composed of a set of scripts that perform the tasks shown in figure 3. It has been developed over the course of about seven months and is still evolving.

2 The language for the PRM: PERL

The PRM package in its current status contains a mixture of shell and PERL scripts and in the future will contain C++ and/or Java code. The choice of the various languages is, naturally, related to the group member's knowledge and experience and to the specific tasks that we implemented.

A core part of shell scripts already existed when the PRM development began (see section 4), and it seemed natural to continue using them. The language of choice for the early stages of development was PERL [4]: it is a scripting language that is available (due to its Open Source licensing policy [2]) on various platforms, including most Unix flavours.

[2]To be precise, PERL's licence is called the "Artistic Licence" and is a "kinder and gentler version of the GNU

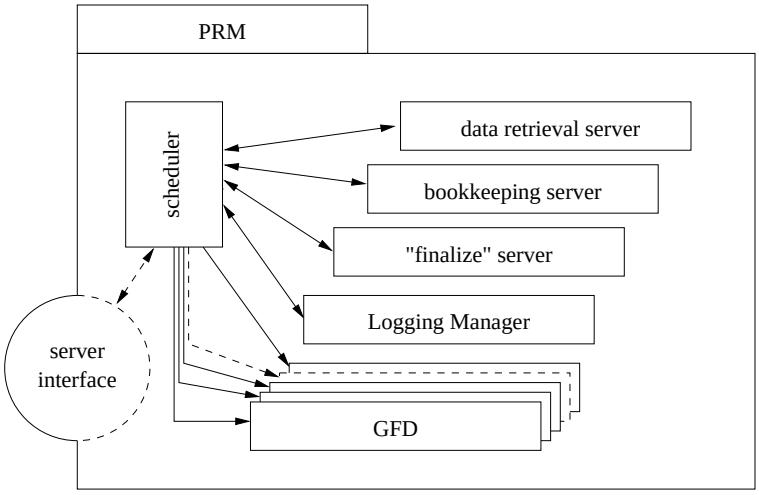

Figure 3: The fine structure of the PRM

It has evolved over the course of years from a language dedicated to data reporting and information extraction into an extremely powerful language with a multi-paradigm (procedural and object oriented) architecture and a vast array of libraries or, more precisely, modules available for the implementation of complex programs.

PERL has been used to build wrappers around the core scripts, thus isolating the complexity of the system and developing a common interface that abstracts the PRM functionalities from the underlying structure. The use of PERL has made it possible to implement a powerful control system, requiring flexible data extraction from log files and parsing of the output of various system commands.

The system currently handles all the automatic bookkeeping for the completed jobs, using the DBI (DataBase Interface, see [7, 8] and chapter 12 of [10]) mechanism: an object oriented abstract layer which uses a standard interface, plus a driver that is specific for the DB system. Drivers for the most common DB systems, like Oracle [16] and mySQL [17], are available.

An automated data-retrieval agent for the HPSS system has been implemented. It uses a client-server architecture, based on a TCP/IP communication layer which has been realized using the IO::Socket module (see [12] and chapter 13 of [10]), an object oriented implementation of the Unix sockets.

The PRM system is monitored by a watchdog application to ensure reliability and security, allowing the operators to correct problems that might arise.

The operator interacts with the PRM using a simple command language and a client program. A graphical interface using Perl/Tk [5] might be implemented in the future, as well as other tools to allow operators and administrators to collect information about the status of the system more easily.

It is important to notice that it is possible to embed a PERL interpreter in C++ code with a technique that is similar to what is usually done with TCL [9], allowing for reuse of the current "objects" (see section 3 for a definition of "object") implemented in PERL (see chapter 6, page 370 [11]).

---

licence." It is available at `http://www.perl.com/language/misc/Artistic.html`.

3 The necessity-driven software development model

The development of the PRM, up to its current state, often has been the response to practical necessities discovered during daily operation by the users or by the administrators, and incorporated into the evolving software structure as part of the design and as part of the implementation. So it is possible to affirm that the whole evolution cycle of our code has been completely driven by necessity.

The necessity-driven software development model is a response to various needs:

- Rapid prototyping: we need a working and reliable system implemented in as little time as possible.
- Flexibility: we need a system that can evolve with our needs on a short timescale, allowing for a high level of flexibility for our ever-changing requests.
- Robust design: we need a system that can grow and adapt itself to the design of the entire OPR system in a robust way.
- Maintainability: we need a system that is easy to maintain and which lets us, thanks to the multi-paradigm approach, to incorporate sub-parts written with completely different software techniques, i.e. procedural and object oriented.
- Reliability: we have used a variety of languages to obtain such results, and although the system might appear to be written in a strictly procedural way, various object oriented design techniques have been applied to develop it.

The most evident technique used has been the implementation of VIL [3]-like structures that allow the PERL and shell scripts to interact like a dynamic network of intercommunicating "objects".

The term "object" is used here with its most general meaning: "objects" are simply entities that know how to perform a certain task. That property is not related to their physical nature; they might be scripts, Java applications, C++ applications or, more generally, CORBA [18] based applications, written in any language.

This approach lets us fully exploit the multi-paradigm/multi-language structure of our system, allowing for the use of the language that seems to be best-suited for a certain application. All we need then is to develop a VIL-like structure to wrap the script, and then allow it to act like a new "object," interacting with the other "objects" that are already present and active.

It is interesting to notice how this technique of adding a VIL-like wrapper to any program is a process of abstraction: once we have our "objects" interacting only through their interfaces, the contents of the "object" can be completely ignored by the user, restoring the "black box" concept that is fundamental in all object oriented design techniques.

Eventually, we might remove the scripts that we are using now, due to performance or security issues. Thanks to the structure of our software model, we are not forced to do that on any special time scale, until a real necessity arises that can be better addressed using a different programming language.

Another important element of this software model is the wide use of design patterns. Such structures let us describe "*communicating objects and classes that are customized to solve a general design problem in a particular context*" [13]. The fact that we are using a generalized definition of object does not reduce the power of such structures nor their application domain. Instead, it allows us to easily evolve toward a fully object oriented system without the need of changing the design that we have developed so far.

It is of extreme importance that in our abstract design (whether it is purely abstract or

[3] VIL: virtual interface layer

necessity-driven) we always use the "one object⇔one task" equation [4]. Such separation makes it possibile to evolve individual parts of the system as a response to a specific need, leaving the other parts completely untouched. Since each part of the system performs just one specific task, it is simple to delegate the execution to another "object" chosen dynamically, allowing for a simulated late binding.

From the pattern viewpoint, the PRM is a Facade (page 185 [13]), abstracting the GFM from the inner complexity of the subsystems actually involved in the management of the reconstruction process. The GFM delegates the job's execution to the PRM.

The abstraction layers between the PRM and the various subsystems use a combination of Proxy/Adapter (page 207/139 [13]) patterns which allows it to modify the interaction scheme.

The independence at implementation level of the various intercommunicating "objects" makes it easy to dynamically modify parts of the system, i.e., adding another abstraction layer to delegate a task to some other "object," without disrupting the functionalities of the whole system. The dynamic relations between "objects" at execution time make it easy to distribute tasks over a LAN (or over a WAN, if it were needed), adapting the system to our evolving hardware configuration.

The next important step will be to switch from the current transport/communication layer (TCP/IP) to a completely CORBA based distributed architecture. Our design approach and our software development technique have already shown that this evolution will be easy, actually resulting in a simpler system.

In fact, some of the tasks that are currently performed internally by our "objects" (like the look-up for an "object" which, in turn, provides a certain service, see figure 4) will be performed automatically by the ORB (see figure 5, thus making the communication layer completely transparent to the various "objects".

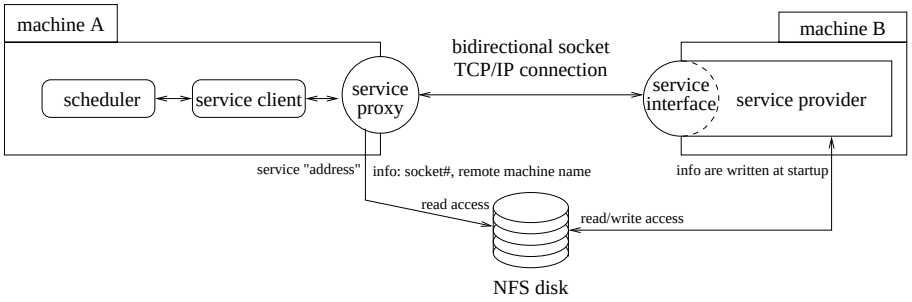

Figure 4: TCP/IP based communication scheme between "objects" in the PRM

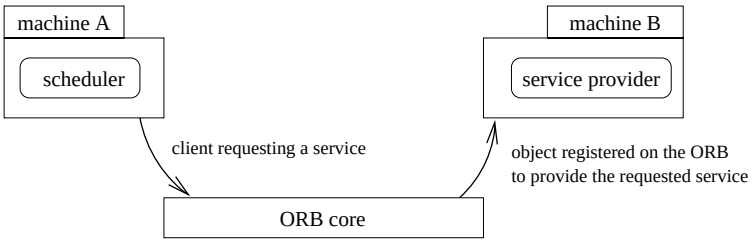

Figure 5: CORBA based communication scheme between "objects"

It is interesting to notice that other parts of the OPR system are already CORBA based, like

[4] Although if we were to violate that rule we could always implement Adapters to split tasks and responsibilities.

the LM and the GFD [15]. Due to the implementation of an Adapter, realized with a combination of PERL and C++, the interface between the PRM and the GFD is already totally transparent, allowing exchange of messages between remote objects.

The description of the development of the PRM will be made using the conventional iteration development model. In each iteration we start with a set of requests (needs) to be addressed. We evolve our system and then test to verify that the requests are met and satisfied, before moving on to the next iteration. Since our model is necessity-driven, the requests might change halfway through an iteration, so the iteration does not capture the real nature of the development process, but it is still an useful approximation.

4 Iteration 0: the original architecture

When we first began working on the PRM, the software available for running OPR jobs was a set of Bourne shell scripts. Their only function was to start the Logging Manager on a remote server, then begin the reconstruction program on all the farm machines, and then hand the control to the operator.

The system was undergoing extensive tests, and it was not possible to keep up with the data being taken by the detector. Since there was no automation, that would have required an enormous effort.

The operator had to monitor the job continuously to spot possible problems and to verify that it had been completed. Then a series of controls had to be performed before submitting a new job. The retrieval of the XTC files for processing and all the bookkeeping operations on the electronic logbook (E-logbook) were completely manual. The operator had to run some scripts to collect the information and then paste them into the E-logbook using a CGI interface. The Constants Block DB (see section 6) was automatically updated by the scripts used for the retrieval operation.

For the integrity of the system, it was, and still is, fundamental that each machine in the farm was processing one and only one job [5]. So it was again up to the operator to verify that the reconstruction program was finished on every machine before submitting a new job.

It was necessary, too, to verify that the job had not left any outstanding locks against the Objectivity database (used for the event store), otherwise the next job might have refused to start itself.

The performance of the OPR system was seriously affected by the experience of the operator, and the delay between jobs was high. This kind of manual job submission system had been developed to test the OPR chain but was completely lacking any automation features that were requested by the *BaBar* software model, requiring an operator to attend to it all the time.

5 Iteration 1: the scheduler

The first step toward the implementation of the PRM system was the development of a simple "job scheduler." The requirements for this program were extremely simple. It had to be able to accept a list of jobs from a text file created by the operator and then process them in a serial way.

Some of the controls between runs were automated, so that the script was able to perform some of them during and after the job by itself, without requiring operator control. In case of problems, the script's behaviour was failsafe: it would stop and wait for the operator's intervention to solve the problem.

[5] This issue is due to the resources of the farm machines. By design, we run one reconstruction job per CPU.

It was still up to the operator to retrieve the runs to be processed from HPSS and to enter the required information into the E-logbook, while monitoring the scheduler operations by checking its logfiles.

A serious problem was that it was, at least theoretically, possible to run two or more instances of the run scheduler together, thus submitting more than one job which risked causing a deadlock situation as described earlier (see section 4).

It was, therefore, necessary to devise a locking policy to stop the scheduler from being executed more than once. To do that, we used the file-semaphore technique: the scheduler created a small text file, a "lock" file, at startup in an NFS directory that was accessible from all machines running OPR jobs. Before starting up, the scheduler would then verify the presence of the lock file, refusing to start if it were present.

A similar technique involving sockets and machine names is still in use (due to the hardware configuration of our machines), and will be completely overridden when we will switch the whole system to a CORBA based architecture. This will allow us, using CORBA nameserver features, to interact with an "object" whose physical location (machine- and socketnumber-wise) is completely unknown.

6 Iteration 2: Bookkeeping and data retrieving

The next need to satisfy was the automation of the bookkeeping procedures and the creation of a system to allow the operator to retrieve sets of XTC files.

The bookkeeping for the *BaBar* experiment is performed on two different systems:

- E-logbook: it is used to store information about job execution, number of events processed, names of Objectivity collections created, performances, problems, start time, end time and some general notes about the job. It is handled by an Oracle DB.
- Constants block DB: it is used to store information about the XTC file, such as size, filename, whether it is present or not on disk and if it has been processed or not, as well as a history of the operation performed on that file. It is handled by a mySQL DB.

Both databases needed to be updated by the scheduler. The information to be stored was derived from the execution of other shell scripts that we inherited from iteration 0. With PERL it was possible to automatically execute the scripts that were needed, parsing the output to extract the requested information. Using the DBI interface (see section 2), it was then possible to form SQL queries to update the database entries, relieving the operator of that responsibility and ensuring the continuous update of DB entries as soon as the information was available.

Using the "one object⇔one task" equation, it was perfectly natural to delegate all of the DB-related tasks to a new "object."

Another task that was natural to delegate to another "object" was the data retrieval from HPSS. The data-retrieval script acts like a server running on, for performance reasons, the remote machine whose local disks are being used to physically restore the data. The access to this machine is restricted to certain user groups with special privileges, so it was necessary to create a client program for the operator to remotely interact with this "object."

The original implementation of the client used a text-file-based message-exchange technique. The client creates a text file containing the commands that it wants the server to execute. The server polls a certain directory for such a file, and when it finds it, it executes those commands, writing the output into another file for the client to read and process.

It was possible to use this method due to the presence of an NFS area shared by all servers, but it proved to be extremely slow and often unreliable, especially if the client and the server were running under different UIDs. Often privilege problems arose on the files that got created,

resulting in potentially dangerous situations.

The data-retrieval service used a lock system analogous to the scheduler, and that enabled the client to verify whether the server was running or not. If not, it would warn the operator about the problem. It was still the operator's responsibility to make the XTC files available for processing.

The same privilege problem was met using the scheduler. The scheduler would run with the UID of the operator who started it, making it impossible for another operator to stop it, unless he had access to the `sudo kill` command.

It was then necessary to rewrite the scheduler with an explicit client-server architecture. The server task was to process the XTC files, coordinating all the other tasks that were job-related, but it was able to accept commands from a client, allowing any operator to interact with the server. Again the interaction system was file-based, and it showed the same limitations that we had met with the data-retrieval server.

In an effort to make the system more reliable, we added to the scheduler the possibility of sending email and pages to the operator. This task has been now delegated to the watchdog system, an "object" which performs many controls about system integrity.

From the performance viewpoint, we observed that some of the tasks performed by external "objects" were extremely costly, especially the bookkeeping ones. In this phase, we simply decided to execute them in background, without any check of their successful completion and ignoring any potential problems. This issue was addressed and solved in the next iteration.

7 Iteration 3: Sockets and further automation

Various needs had to be addressed in the iteration that led us to the current system:

- Performance: we had to parallelize as much as possible the independent tasks to ensure the lowest possible deadtime between jobs.
- Reliability: a more robust and secure system was needed. One step was to use a better and faster communication layer. Another step was to implement a watchdog agent whose only task was monitoring that everything was working correctly and, eventually, warning the operator about problems.
- Security: the need for a higher degree of system security implied that it was not possible to give certain Unix privileges to all the operators. We decided to run the server with a UID having all the necessary privileges and allowing it to accept requests from the operators (adding a security layer).
- Bookkeeping: in the previous versions, it was the operator responsibility to be careful not to reprocess data that had already been processed. It was considered fundamental to implement some kind of data quality check to prevent uncontrolled reprocessing.
- Design: we needed to normalize the software structure in such a way to make it easy to evolve toward the complete structure of the PRM.

The new system required a complete logical redesign of the structure, redefining the functional relations between the various "objects" in it. Often what was needed was the implementation of interface layers, to decouple separate "objects," thus allowing for a greater degree of flexibility in the dynamic correlations. The communication layer was chosen to be based on TCP/IP and implemented using bidirectional socket connections.

The reason for not directly using CORBA was the lack of a robust implementation of CORBA for PERL. There are various efforts in that direction (ILU [19, 20], MICO [21], COPE [22, 23]), but most of them were still in early beta stage and not completely reliable. Since we needed a reliable system over a short timescale, we chose to use well-tested tools like TCP/IP sockets and

delay the implementation of a CORBA based architecture to the next iteration.

PERL provides in the IO::Socket module (reference to the socket module) an object oriented implementation of the socket interface, allowing the module user to easily create and manipulate them. Our “objects” were then adapted to use a socket interface, through a virtual communication layer implemented via a proxy to decouple the physical communication layer from the interface (see figure 6).

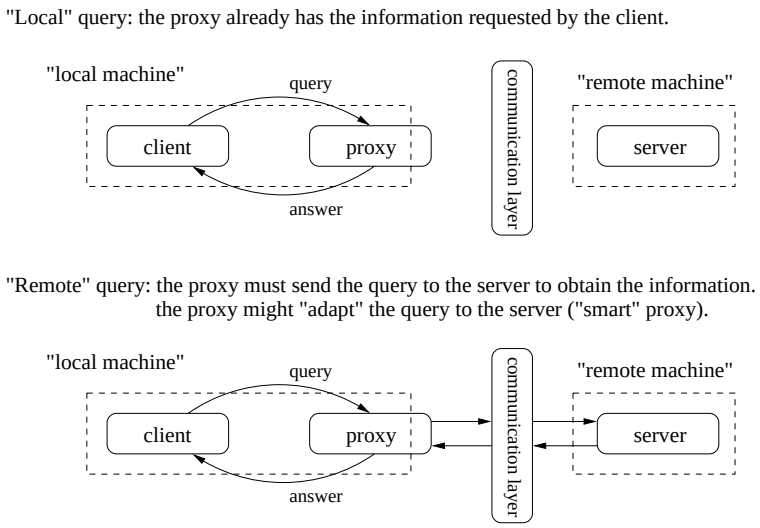

Figure 6: Client-server interaction through the proxy interface

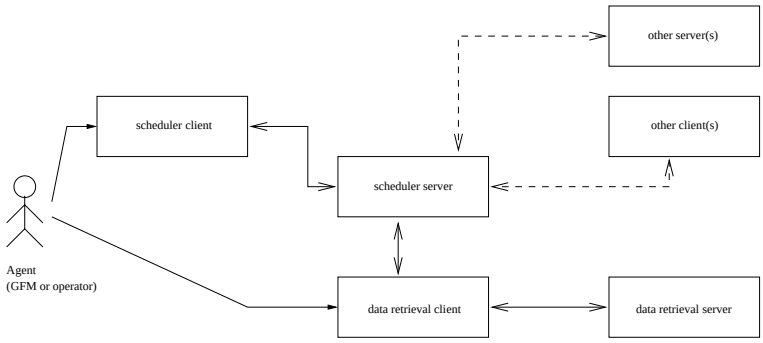

Figure 7: Interaction between “objects” in the system

The redesigned system is based on a scheduler process that runs continuously on a remote server. The operators can submit jobs using a local client, and with the same local client, they can monitor the execution of the jobs. The scheduler is able to retrieve XTC files directly, acting as a client for the data-retrieval “object,” and by policy it requests the file for the next job as soon as it starts processing the current job. This process is asynchronous, by delegation, and is an important enhancement over the previous system. With the old versions, the operator had to manually retrieve a few XTC files together, thus occupying a lot of disk space [6].

When the operator tries to submit a job, the client hands the operator’s request to the communication proxy. The proxy performs a set of controls for the job, such as verifying that the

[6] A typical XTC file ranges between five to nine GB.

operator has the privilege to perform the requested operation, that the job contains data considered interesting (i.e., data from collisions and not cosmics) and that it has not been already processed. If the job satisfies all the required parameters, the proxy submits it to the server. The server stores the job request in an execution queue, which is simply a FIFO.

The jobs are still processed in a completely sequential fashion, on a "first come, first served" basis, although recently we have introduced the possibility of modifying the queue order. In the future, we might introduce a priority parameter associated with the job and a more flexible job scheduling policy, depending on need.

To allow for different kinds of processing to be performed (i.e., different releases or different event filters), a configuration queue has been implemented as well, so that the user can specifiy a different configuration for each job. Configurations can be manipulated as jobs to change the behaviour of the scheduler system.

A watchdog agent has been created and added to the OPR system. It monitors some parameters, like available disk space, and periodically polls the other "objects" to verify that everything is working correctly. In case of problems, it is capable of sending email and pages to the operators and to the administrators. The watchdog is another step toward the complete automation of the system, slowly substituting the need for a human operator to monitor that everything is working properly. Its current main limitation is that even if it spots a problem, i.e, a server which is down, it is completely unable to fix it, i.e., restarting it. The next iteration is expected to contain some self-healing elements, and the watchdog might be used to implement that.

8 Iteration 4: PRM, the complete system

The next step will be the implementation of the full PRM system. The latest performance studies suggest that it could be possible to obtain better performance from our system if we were to run multiple instances of the PRM at the same time, partitioning the reconstruction farm into two or more sub-farms.

Currently, the structure of the farm is static: the machines are allocated at the job's start-up, and, due to our policy, it is not possible to add more machines during the processing, nor to change the farm partition dynamically [7] . This is another degree of freedom that needs to be added to the system, allowing for different priorities of processing and balancing the load on the available farm machines.

The complete system will be able to recognize the availability of new data, check whether it can process those data, and then schedule them for execution, eventually bypassing lower priority jobs. Not all of these tasks will be performed by the PRM. The PRM will be controlled by the GFM, which will have the responsibility of starting new jobs, eventually creating different instances of the PRM with different scheduling policies and different partitions of the reconstruction farm.

It might be possible that under most conditions the GFM would delegate the PRM to decide what kind of processing is optimal for the given conditions, maintaining just a basic level of control of the PRM instance, but the policy about this behaviour is still being studied.

Another important request is the reprocessing of all the data that have been taken so far with new, more reliable reconstruction code. Another farm has been created, dedicated to this new task.

The design and the implementation of the multi-instance version of the PRM will lead us a step closer to the final system and will contribute to the complete, detailed design that we need to create the PRM implementation that will fit into the global architecture of the Prompt Reco system.

[7] The mechanism for adding machines to a running job already exists. It is simply not being used.

Many parts of the system will need to be re-engineered to use a CORBA architecture, and the first step in doing that will be implementing a generic CORBA application-wrapper to adapt the existing "objects" to a different communication layer, substituting the existing socket structure.

To optimize performance, the scheduler will probably need to be rewritten in C++ or Java, while still retaining PERL code for certain tasks, such as the logfiles parsing. This can be done either by incorporating a PERL interpreter inside the C++ code or by creating an interface layer to communicate with remote scripts using the CORBA application wrapper. A version of a remote launcher based on CORBA already exists and is called the Global Farm Daemon [15].

The DBI interface used by Perl has no equivalent in C++, but an interesting possibility could be to implement an SQL DB server using Java/JDBC [24, 25], thus allowing other "objects" in the system to store and retrieve data from the E-logbook in a completely asynchronous way.

The introduction of a CORBA architecture will relieve the code from the need to check whether the remote "objects" providing certain services are available. Such a possibility, combined with a simple use of techniques derived from the self-healing software concept, will allow for maximum flexibility and reliability of the system. The implementation of a monitor system will be greatly simplified by the availability of a common interface to standardize the procedure to query the "objects," making it possible to dynamically reconfigure the OPR system, as well as the monitor software.

9 Conclusions

The need to always have a completely working system has been the driving force for the evolution of the PRM. We started with a basic set of scripts, and developed, over time, a powerful and reliable system that is able to run in a mostly self-sufficient fashion, requiring very little effort from the operator.

The use of a necessity-driven software model has allowed us to evolve our system in a completely natural way, learning by doing and incorporating our daily experiences into the development process. This structure has been transported, due to the use of the pattern design technique and some attention to the underlying object oriented structure, into the design of the PRM system.

Many lessons have been learned and many different possibilities explored. The result is a robust and flexible design which has allowed us to create a working system. This detailed design will allow us, in the next few months, to implement the complete PRM system while still retaining the ability to incorporate and adapt new needs, as they arise.

10 Acknowledgments

I'd like to thank my fellow members of the OPR group: R. Cowan, F. di Lodovico, T. Glanzman, G. Grosdidier, S. Dasu for their contributions to this paper. Thanks to Vicki for the support and the editing.

References

1 OPR Homepage: `http://www.slac.stanford.edu/BFROOT/www/Computing/Online/PromptReco/index.html`
2 *BaBar* Homepage: `http://www.slac.stanford.edu/BFROOT/`
3 Objectivity Homepage: `http://www.objectivity.com`
4 PERL Homepage: `http://www.perl.com`
5 S. Lidie, "Perl/Tk Pocket Reference", First Edition, O'Reilly & Associates Inc., 1998.

6 S. Dasu, J. Bartelt, S. Bonneaud, T. Glanzman, T. Pavel, R. White, “Event Logging and Distribution for BaBar Online System”, CHEP98, Chicago (USA), 31 Aug-09 Sep 1998.
7 DBI: http://www.symbolstone.org/technology/perl/DBI/index.html
8 DBI: http://www.engelschall.com/ar/perldoc/pages/module/DBI.html
9 TCL Homepage: http://www.scriptics.com
10 E. Siever, S. Spainhour, N. Patwardhan, “Perl in a Nutshell”, First Edition, O’Reilly & Associates Inc., 1999.
11 L. Wall, T. Christiansen, R.L. Schwartz, “Programming Perl”, Second Edition, O’Reilly & Associates Inc., 1996.
12 IO::Socket: http://www.perl.com/pub/doc/manual/html/pod/perlipc.html
13 E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns”, Addison Wesley Longman Inc., 1994.
14 T.Glanzman, J.Bartelt, R.Cowan, S.Dasu, G.Grosdidier, F. di Lodovico, F. Safai Tehrani, “The BABAR Prompt Reconstruction System, or Getting the Results out Fast: an evaluation of nine months experience operating a near real-time bulk data production system”, CHEP2000, Padova (Italy), 7-11 Feb. 2000.
15 G. Grosdidier, S. Dasu, T. Glanzman, T. Pavel, “Sending Commands and Managing Processes across the BaBar Opr Unix Farm through C++ and CORBA”, CHEP2000, Padova (Italy), 7-11 Feb. 2000.
16 Oracle Homepage: http://www/oracle.com/
17 MySQL Homepage: http://www.mysql.com/
18 CORBA official homepage:http://www.corba.org
19 ILU Homepage: ftp://parcftp.parc.xerox.com/pub/ilu/ilu.html
20 ILU and PERL: http://www.gtk.org/~otaylor/ilu/index.html
21 MICO PERL: http://people.redhat.com/otaylor/corba-mico/
22 CORBA/PERL:http://www1.lunatech.com/cope/
23 CORBA/PERL: http://www.ve3tla.ampr.org:80/%7Eirving/
24 JAVA Homepage: http://java.sun.com/
25 JDBC Homepage: http://java.sun.com/products/jdbc/index.html