SLAC-PUB-11138
April 2005

The BaBar Data Reconstruction Control System

Antonio Ceseracciu, *Member, IEEE*, Martino Piemontese, Francesco Safai Tehrani, Teela M. Pulliam, Fulvio Galeazzi

***Abstract*— The BaBar experiment is characterized by extremely high luminosity and very large volume of data produced and stored, with increasing computing requirements each year. To fulfill these requirements a Control System has been designed and developed for the offline distributed data reconstruction system.**

The control system described in this paper provides the performance and flexibility needed to manage a large number of small computing farms, and takes full benefit of OO design. The infrastructure is well isolated from the processing layer, it is generic and flexible, based on a light framework providing message passing and cooperative multitasking. The system is distributed in a hierarchical way: the top-level system is organized in farms, farms in services, and services in subservices or code modules. It provides a powerful Finite State Machine framework to describe custom processing models in a simple regular language.

This paper describes the design and evolution of this control system, currently in use at SLAC and Padova on ∼450 CPUs organized in 9 farms.

***Index Terms*— distributed systems, control system, high energy physics software.**

I. The BaBar Data Reconstruction and the Control System

A. BaBar Prompt Reconstruction system

THE Prompt Reconstruction (PR) system is part of the software for the *BaBar* experiment. Its tasks are to: generate calibration data; and process the incoming data from the detector, performing a complete reconstruction of physical events producing results ready for physics analysis. These tasks are done in two separate *passes*: the Prompt Calibration (PC) pass, and the Event Reconstruction (ER) pass. The need to automate this process pushed for the creation of a first Control System, and later for a new design. This new design and implementation is the subject of this paper.

The data obtained from the detector Data Acquisition System (DAQ) is stored in files using a format known as "extended tagged container" (XTC). There is a 1-to-1 correspondence between data runs and XTC files. Those files are then archived in the mass storage system. The typical data rate of XTC production is 600 GB/day; the integrated amount of stored XTC data is about 270 TB, at the time of writing. The Prompt Calibration pass calculates and stores the new calibrations for each run into the conditions database, presented in [1]. These calibrations are then used by the Event Reconstruction pass of the run which produces fully reconstructed events and writes them to the event store database. The Prompt Calibration and Event Reconstruction passes are processed by farms of typically 32 and 100 CPUs respectively.

Each farm processes one run at a time. For each run, an instance of a daemon known as the Logging Manager (LM, see [2]) is started on a dedicated server. It distributes the events read sequentially from an XTC file to multiple reconstruction processes running on the farm machines (nodes). The Prompt Calibration farm must process runs in sequential order, as the calibration uses information from the previous runs, while the Event Reconstruction farm is only constrained to process runs that have been calibrated. For a detailed discussion of the BaBar Prompt Reconstruction model see [4] and [7].

B. The need for automation: Control System

The reconstruction software provides all the tools necessary to process a run in a computing farm. Using the tools by hand is impractical, given the high data rate of BaBar production. The basic purpose of the Control System was to automate the production procedures, in order to obtain a system able to process long sequences of data runs without continuous input from human operators. The first control system was then realized using some of the most popular tools available for system level automation: the perl programming language and shell scripting. This first Control System is described in [5].

This necessity driven development process caused the control system scripts to grow in complexity. "Quick and dirty" programming approaches, typical of system level scripting, raised manageability issues; local improvements easily caused global disruption. The behavior of the system became difficult to understand and control. Even though the system was able to properly satisfy its functionality requirements, these flexibility concerns motivated the design and implementation of a different system.

II. The new Control System

The lessons taught by the development and operation of the old system established the functional requirements for the new Control System, but also a specific set of software requirements:

- flexible: able to execute a variety of processing models, distributing processes and tasks on different machines as decided by the user;

A. Ceseracciu is with the Stanford Linear Accelerator Center, 2575 Sand Hill Road M/S 97, Menlo Park, CA 94025, USA (email: antony@slac.stanford.edu); was with Università di Padova, Padova, Italy.

F. Safai Tehrani is with INFN Roma c/o Universitá' di Roma "La Sapienza", P.le A. Moro 2, 00185 Roma, Italy; was with the Stanford Linear Accelerator Center, Menlo Park, CA, USA

M. Piemontese, Flat 3, 75 Gauden Road, SW4 6LJ, London, UK; was with the Stanford Linear Accelerator Center, Menlo Park, CA, USA

T.M. Pulliam is with the Ohio State University, Columbus, Ohio, USA.

Fulvio Galeazzi is with INFN Padova, c/o Dept. of Physics, Universitá di Padova, via F. Marzolo 8, 35100 Padova, Italy

Work supported in part by the Department of Energy contract DE-AC02-76SF00515
SLAC, Stanford University, Stanford, CA 94309

- reliable: complete and efficient error and exception handling;
- extensible: new modules and features can be easily added to the processing models;
- simple: both for the end user and for the developer, interfaces are kept as simple as possible;

To meet these requirements, the Control System is highly modular and distributed, avoiding as much as possible single points of failure which caused problems in the former control system. Relevant services are duplicated as needed (statically and dynamically) to enhance robustness and performance.

The Control System can be described as a dynamic network of distributed services interacting to perform all the data processing related tasks. These tasks can be summarized as follows:

- Data Flow
- Data Processing
- Distributed Networking
- Finite State Machines
- System Configuration
- System Monitoring and Alarms
- Task Automation
- User Interface

A. Engineering for flexibility: Services

A service is a high level design concept meant to separate functionality from actual implementation. The end-user will see the system in terms of the services it provides, without the need for an in depth knowledge of the internal structure. The key concept of services is topological transparency: any service meant to be visible globally in the system can be accessed from anywhere in the system. The client needs no knowledge of the location of the service on the network or of the network topology. The implementation of topological transparency is described in III-C. Also, services can be hierarchically layered in order to create higher level services which are potentially more descriptive of the problem domain and thus able to hide from the user some details of the programming interface.

III. Framework and agents

A. Introducing the Lightweight Processing Framework

The entity providing cohesion to the system is the Lightweight Processing Framework (LPF). The framework itself only provides essential services (hence the attribute "Lightweight"): cooperative multitasking, message passing and dynamic module management. Any other core task is delegated to the core framework modules, and all of the components of the Control System are coded inside framework modules. Each node participating in the farm runs at least one LPF agent, thus allowing for remote activation of services from one or more remote LPFs. This also allows dynamic restructuring of the Control System, by easily "moving" services around in a completely transparent fashion, e.g. shutting an instance of a service and creating a new one on a different LPF is transparent to the clients of that service. We can see the set of all the LPFs running in the system as a distributed service manager.

The framework has the ability to "load" services and objects through well defined interfaces. The implementation of the LPF was the first step toward the implementation of the Control System. Framework functionality is provided by a set of core modules. Dedicated modules take care of:

- message passing front-end
- dynamic module allocation
- TCP/IP inter-LPF communication
- Transparent proxy system
- Fault-tolerant naming service
- Active handling for external processes
- Internal event scheduling, cron like
- Local and global message logging
- Alarm handling

B. The Structure of the LPF

The LPF is a transparent facility: the operator never needs to interact with it directly. The uniformity of interactions between the framework and the object/service modules is guaranteed by the requirement that all the Framework-instantiable objects implement a common interface.

The LPF is an event based server with a cooperative multitasking structure, also known as non preemptive multitasking. In this processing model there is a single flow of execution in which many parts (modules) participate. Each module must be written in such a way that it performs a single set of operations, in a limited amount of time, and then returns control to the caller. After performing the initialization, the Framework reaches the "steady state", that is the main_event_loop. During this phase it cyclically transfers control to each one of the modules, thus simulating a simultaneous flow of execution of various applications. Modules interact via messages. In particular, each module generates a list of messages to be distributed to other modules as a return value. The inter-object communication becomes completely transparent by centralizing the message distribution, without the need for any additional inter process communication structure.

The modules in the Framework can be logically divided into two separate groups:

- Passive Modules: these are given control only when they receive a message.
- Active Modules: these need to be given control during every iteration.

The module interface provides simple methods to allow for transparent interaction with the Framework:

- Init: invoked after instantiation, allows modules to execute a custom startup sequence, possibly based on the configuration.
- Do: receives a message and executes the associated action returning the answer as one or more messages;
- Run: performs a set of operations to be executed during every iteration for Active Modules. Passive Modules don't define this method.

- Kill: performs the set of operations needed to remove cleanly a module from the system;

The main_event_loop of the LPF looks like this:

- begin_event_loop:
- invoke the run method of each active module and ...
- ...push the messages returned in the message queue;
- invoke the run method of the dispatcher module, passing the message queue as an argument;
- the dispatcher module dispatches the messages to the receivers and...
- ...collects the answer messages that they generate and...
- ...pushes them in the message queue;
- end_event_loop;

C. Messages and Transparent Proxies

Any inter-module communication is mediated by messages. Messages sent to a module are implicitly mapped by the LPF to remote method invocations. Message passing is strictly asynchronous.

The framework provides message passing among the modules that are loaded on a given LPF. Hence, when a client module sends a message to a service, and the service exists as a module loaded on the same LPF, the message is routed directly to that module. This mechanism satisfies the requirements for local, inter-module communication.

When a client sends a message to a service that is not represented by any module loaded on the local LPF, the transparent proxy infrastructure is activated. The wanted service is looked up by name on the remote Naming Service; if an instance of that service exists on a remote LPF, then a transparent proxy module is created on the local LPF. This proxy module advertises itself locally as the real service, and replicates its programming interface. Any message the proxy receives is forwarded to the real module, and the answer routed back to the caller. Chained operation (proxy of a proxy) is supported.

If the wanted service is not available anywhere on the system, i.e. it is not found in the Naming Service directory, then action is taken depending on a policy statically declared on a per-module basis. Depending on that policy, an error message is returned and/or an alarm is raised, or the service is instantiated on demand on the local LPF and made available to others via the Naming Service. This mechanism is designed to work in in cases where an LPF suddenly becomes unreachable: in this case, the services it was hosting are deleted from the Naming Service directory, and replacement services are instantiated as needed.

The inter-LPF messaging protocol is based on TCP and a generic serializer: messages are encoded into strings by the serializer and sent via a TCP socket. The TCP communication code is entirely contained inside a core module. This is a minimal approach designed to take advantage of the controlled environment of a typical computer cluster.

D. The Agents deployment

Each LPF agent of the new Control System can be configured to dynamically load a number of services . Figure 1 shows the architecture of the Control System for both PC and ER farms, with emphasis on the communication flow. The main structure of both passes is very similar. The main driver of the Control System is the Farm Manager (FM) service which acts as a broker between two separate levels of service: the first level (upper part of figure 1) includes all services needed for the staging (XTC) to disk of the runs and the services that take care of the scheduling of the runs; the second level (bottom part of figure 1) includes all the services that take care of the event reconstruction (ER pass) or calibration (PC pass). The Farm Manager serves also as the main access point for the user to the system. The two layers have separate services' namespaces, each one of them having a Naming Service (NS).

The reconstruction part of the single run is controlled by a Run Processing (RP) Finite State Machine service (see section III-F for the Finite State Machine abstraction and service implementation) that starts the Logging Manager [2] on the farm server and takes care of the bookkeeping; it also starts and dynamically configures a number of Node Processing (NP) Finite State Machine services . Typically on each CPU is run an instance of the event reconstruction worker process, called Elf. Each of the Node Processing Finite State Machines (see figure 2 for the description of the states of the NP) starts the Elf and waits until it has finished, then it performs integrity checks on the output data files and gathers information from the output of the Elf.

A minimal number of the Control System services has a lifespan longer than the processing time of a run, and is able to contact multiple farms. Only services that have at least one of the two requirements are hosted in this part of the system: any other service is hosted inside farms, to take advantage of the independence and isolation of the farm's environment. Such long-lived services are typically feeders, that schedule and allocate the runs to be processed to the farms, implementing policies for load balancing; and monitoring services.

The only persistent component of the Control System is the bookkeeping database. It is a relational database that stores the status of current and past processings. Currently a separate instance of this database runs at every production site, and the instances are synchronized via an automatic scheduled procedure.

E. The unified configuration and activation system

1) Formal hierarchic organization: The configuration system is designed to define the whole configuration of a large and complex computing system in a unique structure, contained in a small number of files. This centralized approach makes life easier for the user, but it can also be a potential performance and reliability problem, being a centralized service in a distributed environment.

A formal hierarchical model allows the maintenance of a global configuration, but grants the flexibility to configure any

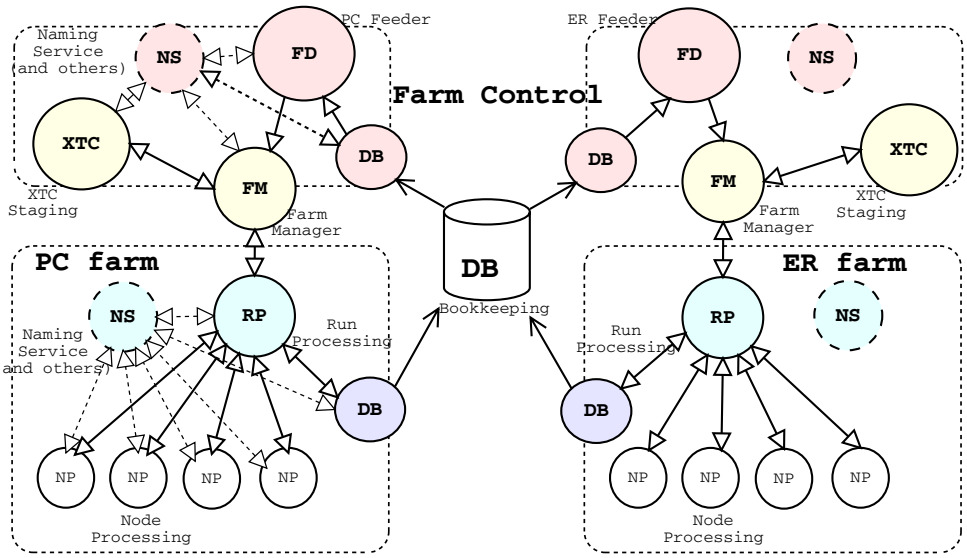

Fig. 1. The actual deployment of the Control System agents, and the partitioning between multiple farms (depicted are a Prompt Calibration (PC) and an Event Reconstruction (ER)) and farm subsystems (Farm Control, Run Processing). The dotted lines represent interaction with the naming service for transparent service name resolution, and are not printed on the right side for ease of reading.

group of computing nodes with the desired granularity. An important advantage of a formal hierarchy is that it gives the computing environment a controlled and well defined structure. A strong structure is the only way to harness the complexity of a large and heterogeneous system, as PR. A clear example is how the activation system benefits from the structure specified at configuration level, as described below.

2) Initialization system: Initialization must be completed before Activation. The Control System initialization procedure consists of just one operation: running on every node a special purpose LPF, called "BareLPF".

A BareLPF is an LPF which only runs some core modules. Its configuration is extremely simple and entirely hard-coded in its special startup file. The BareLPF listens on a fixed TCP port, which is reserved for this service. The BareLPF does not need any other external information or resources to run. The one and only duty of a BareLPF is to spawn new LPFs. A new LPF is spawned upon reception of a specific message. The BareLPF may be started at boot time or remotely by using the standard UNIX facilities like ssh [3].

3) Distributed Activation Protocol: Thanks to the hierarchical model of computation, we can define the activation protocol for a single generic case: a Configuration Manager (A) which activates a lower-level Configuration Manager (B). The activation of the whole computing environment follows by recursive application of this rule. Each Configuration Manager instance corresponds to a LPF agent.

From a local point of view, configuration is performed after activation. On the whole system scale, though, activation and configuration are performed at the same time, because the hierarchy information, needed for the activation, is specified in the configuration. The configuration sequence of an LPF includes the activation of all its children in the hierarchical tree, and cannot return until all its children report success or failure, or after its timeout expires. Depending on what is specified in the configuration, failure to activate a child LPF can lead to the failure of the parent's configuration sequence, or can be ignored, or recovered. When the configuration sequence of the Configuration Master LPF (the top level Configuration Manager) ends successfully, the whole computing system is guaranteed to be configured and ready for operation.

4) Configuration parsing and propagation: The configuration file is valid globally. New spawned LPFs could read the global configuration file, parse it, and find their own subtree. This approach would require the configuration file to be distributed to, and parsed by, every node. This centralized distribution can be a scaling issue from the network point of view; the main disadvantage though is that the distribution of the configuration files would require some software component different from the LPF, like nfs, ssh, ftp, i.e. one more potential problem. In the activation and configuration protocol design the configuration files are parsed only once, by the top level Configuration Master. The parsed configuration structure is passed to the children LPFs upon their activation. This guarantees that all the nodes configuration is coherent, i.e. built from the same files.

5) Local configuration: The configuration service exploits the hierarchical structure of the configuration data to recognize the subset of the configuration keys relevant for the local LPF, and makes them available to the LPF itself and its modules. The configuration supplied for a module is directly copied to a dedicated structure in the data space of the module itself, so that the application code in the module needs not to make any explicit access to the global configuration structure.

6) System deactivation: The activation service is also responsible for the deactivation. The deactivation subsystem is

also known as the Reaper. The deactivation information is maintained on each node by its BareLPF, inside a dedicated module. When the deactivation module sitting on a BareLPF receives the deactivation command, it checks if it carries the name of a specific service to be reaped. If it does not, the BareLPF sends to all of the LPFs active on its host a deactivation message; otherwise, only LPFs belonging to the specified service are targeted. This allows for selective service activation and deactivation on a distributed scale. An LPF upon reception of the deactivation message executes a self-destruction sequence: it stops and unloads its modules and finally exits the main LPF loop. The BareLPF remains active and ready to undergo a new activation of the system.

The deactivation sequence can be run from any LPF in which the full configuration has been loaded; this LPF becomes the Deactivation Master. It is possible to deactivate the whole system at once, i.e. to stop all LPFs running on some node defined in the configuration, even if not bound to a configured service; another option is to deactivate a single System or Service defined in the configuration.

F. The Finite State Machine framework

A Finite State Machine (FSM) is a computation model consisting of a discrete set of states, a start state, an input alphabet, and a transition function which maps input symbols and current states to a next state. Computation begins in a special start state. It changes to next states depending on the transition function.

The processing structure of PR can be naturally described in terms of states and transitions between the states determined by well defined conditions. This offers a very flexible model to describe a processing system and realize it. A valuable benefit of the FSM model is that it provides a good view of the current status of the processing to the human operator. It also makes possible to implement default error detection and recovery procedures based on tunable timeouts and error handlers. Figure 2 shows a simplified succession of the operations (states) and transitions of the NodeProcessingFSM.

Our FSM can be considered a specialized, heavyweight framework dedicated to the application programming. We use the term heavyweight to indicate the extensive and feature rich set of programming interfaces available through this framework, as opposed to the minimal programming interface of the lightweight framework, the LPF. All the services and facilities provided by the core system are made available through the FSM, and any application code is encouraged to be coded inside FSM states. This heavyweight framework makes coding of the application layer easier, by making a rich functionality readily available.

1) The FSM Structure: A generic implementation of the FSM model is coded as a LPF module. This makes it possible to plug an FSM in an LPF at run time. FSMs are specialized dynamically by loading a particular FSM definition to an instance of the FSM module. The FSM definition is coded in a custom language in a single file.

2) The FSM Interpreter and the FSM Description Language: The FSM Description Language is a simple description language designed to describe, extend and modify FSMs with the minimum effort. It is stackless, stateless and doesn't have any control structure, only assignments and logical connectors.

The language grammar is very simple:

- `FSM: [FSM name]`: defines the symbolic name of the FSM;
- `begin: [state name]`: defines the name of state where the FSM starts;
- `state: [state name] isA: [class name]`: associates a symbolic name to a class name for the state;
- `state: [state name] onTransition: [transition name] do: [method name]`: defines a method name to be called on the object that represents the class when a certain transition is generated;
- `state: [state name] onEntry: [method name]`: defines a method to be invoked on the state object upon entering in the state;
- `state: [state name] onExit: [method name]`: defines a method to be invoked on the state object upon exiting the state;
- `state: [state name] timeout: [seconds]`: associates a generic alarm timeout to this state, to spot blocked processing early.
- `state: [state name] onTimeout: [method name]`: defines an handler to be invoked on the state object when its timeout expires.

The FSM framework takes advantage of the dynamic typed nature of the programming language to resolve at run time the referenced classes, and load them.

G. BaBar Reco Finite State Machines

The RunProcessing (RP) FSM describes the centralized part of the processing of a single run. For each processed run, a new instance of the RunProcessing FSM is created, configured, and started. This implies that all the data structures belonging to the previous runs are erased, ensuring reproducibility of the processing, which is an important consistency requirement. Together with the FSM, all its states are reloaded.

Most of the logic of the Control System is coded as states of this FSM, or as part of higher level services and then invoked by proxy states. The complexity of actual RP FSMs is thus considerable, amounting to over 40 different states. The current status of the RunProcessing FSM is often the best indication of the status of the whole processing farm. It allows the operator to monitor the processing by just observing the transitions between states. A tunable alarm timeout associated with each state is also an effective way of spotting problems that are not caught by specific alarm checks.

The RunProcessing FSM is remotely instantiated, configured and started by the FarmManager as soon as all the information needed to start processing a run is available. The first states of the RunProcessing FSM are devoted to collection of information from different sources and consistency checks; a

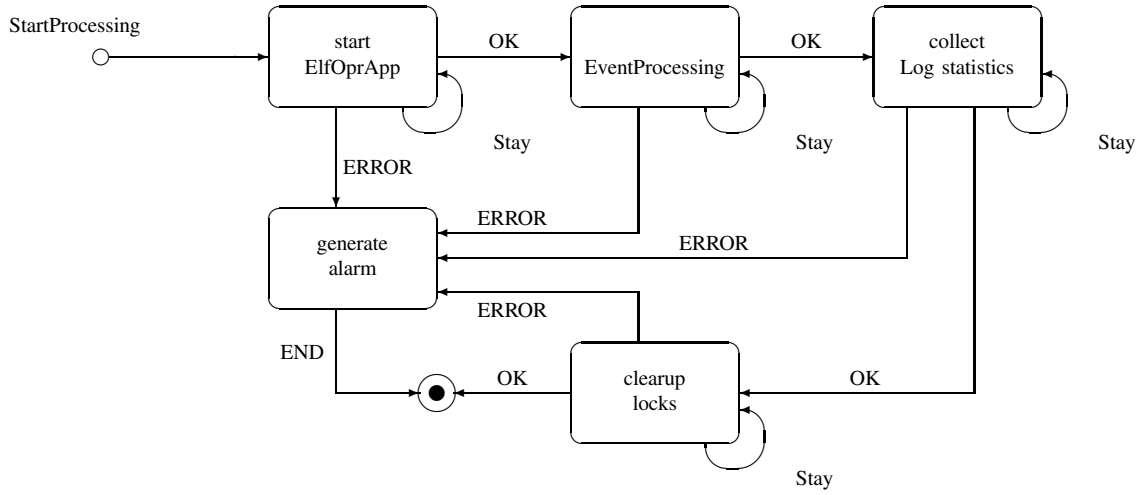

Fig. 2. A simplified view of the NodeProcessingFSM

new directory is created to host all the logfiles for the current run. Then, the Logging Manager ([2]) is started, and a monitor is attached to its log files. The dynamic configuration for the reconstruction processes (Elves) is then produced and written on disk. At this point, the duty of local processing on the nodes is delegated to the NodeProcessing FSMs running on them. They are started and the relevant part of the run-time configuration is passed to them. The RP then just waits for all of them to return. This is the phase where the farms spend most of their time, and where the nodes are actually used for distributed computation. After that, the RunProcessing FSM resumes control and takes care of consistency checking, persistent bookkeeping, and postprocessing, which includes starting the quality assurance procedures. While the FSM includes checking and recovery procedures for commonly encountered issues, the general policy is to avoid blocking if anything goes wrong, because the final consistency checking is designed to give a final response about the success of the processing. When the final state is reached, the RunProcessing FSM instance returns control to the FarmManager, and is reset before the next run.

The PostProcessing FSM is used only in the Event Reconstruction pass. The reason for that, is that its main task is to collect the event data files produced by each Elf, merge them into the final set of files to be shipped, generate bookkeeping information about the files, and ship them to the export system, to make the data available to the collaboration. Many integrity checks are performed along the way to ensure consistency of the shipped data. The PostProcessing FSM is implemented as an independent system, with no run-time interface to the RunProcessing FSM. This approach implicitly creates a decoupling buffer between the two steps, which improves flexibility and resources balancing.

H. User Interface

The user interface makes no use of the LPF and modules. It is nevertheless designed modularly to assist flexibility and maintenance.

The user interface and the control system communicate via messages. User commands prepare messages and deliver them to some LPF. Higher level commands use a discovery protocol to locate the destination LPFs. This protocol relies on the BareLPF being aware of all the services running on all the LPFs running on its host. The modular design makes the whole set of commands available by different means; currently provided are a command line interface and an integrated shell. Any command is automatically available from both. A further flexibility mechanism is supplied to code different sets of commands, called “interfaces”, into separate classes. A selection of interfaces can be loaded and changed at run-time in the command executable. Specific interfaces are dedicated to the highest level commands (`User`), farm administration (`Farm`) , debugging (`Debug`). The ability to send any kind of message to any module makes the command interface a powerful tool for interactive debugging and unit testing.

The Control System GUI is a monitoring tool. Its main panel is displayed in figure 3. It allows to inspect at a glance a set of farms, giving immediate visual feedback about the current status of each. More detailed information about a single farm is available at a mouse click, including the current RunProcessing FSM state and the nodes status (when applicable), and a continuously updated log messages history. Multiple instances of the GUI can run at the same time, even on different machines. Any communication between the GUI and the Control System is carried by regular messages. The GUI uses the perl/Tk graphics toolkit and simulates the LPF main loop thread to implement asynchronous updating of the farms status.

IV. Development

The focus on flexibility is reflected by the very controlled evolution of the project. This is detailed in figure 4.

Being a small team of developers, no formal control on the development process was used. There is however a qualitative observation we can make from looking at the development history of the two main code repositries. The core software

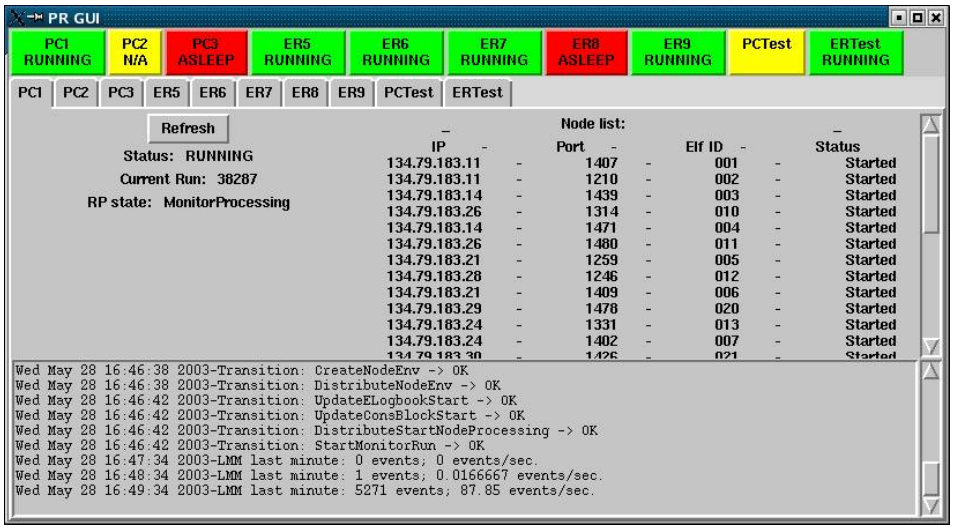

Fig. 3. The Control System GUI monitoring the processing farms at SLAC

was developed in a planned requirements - design - prototype - implement regime, with scheduled milestones and release cycles. The application software, instead, was developed in a very necessity-driven regime, because of the frequent need to quickly adapt and evolve the system. This pressure did not compromise the design qualities of the system though. We consider this a success of building flexibility by design. A more general annotation is that we found that no single formal process metodology could fit the whole system, but rather, that different kinds of code (e.g. core, application...) can be best developed within different process methodologies.

A detailed discussion about the Control System as an example of an evolutionarily efficient system is to be found in [9].

V. Conclusions

A. Performance

The Control System is not computation intensive software. This fundamental assumption supports the choice of Object Oriented (OO) programming with a dynamically typed language: *OO perl*. This combination leads to remarkably slower code than in non-OO fashion, because of the need for the interpreter to resolve symbols, like method calls, at run-time, rather than during bytecode compilation; but, it grants the greatest expressive power.

More significant performance metrics for the Control System are, indeed, reliability and flexibility. The experience of over 24 months of continuous running shows that the time lost for failure of the Control System itself is a negligible fraction of the total. More frequently, changes in the operating environment, like new revisions of tools, created conditions that the Control System was not ready to deal with, until properly adapted ("fixed"). This is inherent to the glue-like nature of the Control System itself.

B. Software design summary

The Control System is an active distributed system. This is a complex programming paradigm, that requires careful design and attention to many issues typical in concurrent programming.

This is not unnecessary complexity however. The main task of the Control System, launching and monitoring external executables, simply requires an active monitor. The only passive way to perform this task, even for simple cases like determining when an external command execution ends, is to poll for lock files and parsing the command logfiles for specific messages. This approach is less elegant and reliable: as an example, in a distributed environment it introduces a considerable burden, the dependency from the quasi-coherence (in-coherence) borne by the distributed file system.

Different software engineering techniques were exploited to manage this complexity. The main focus has been on attaining modularity by design. A very light framework (LPF) hosts all the rest of the code in form of dynamically loaded modules, including core parts, like TCP-IP intercommunication and module activation. Hence, anything coded for the Control System has to run inside a framework module. The OO modeling approach results in a classification of modules and in good code reuse among similar modules. A more structured framework, based on a Finite State Machine abstraction, supports the final processing code. This support makes it convenient to exploit the modular interface even for apparently trivial tasks, thus fostering the developers to design modular code. The main operative advantage of this design is the ease of adapting the system to changing environments. In fact, the Control System is used at any time to execute many different processing models, in different environments.

The programming interface of the framework and of most core components have remained stable over time. The maintenance and evolution of the system greatly benefit from the

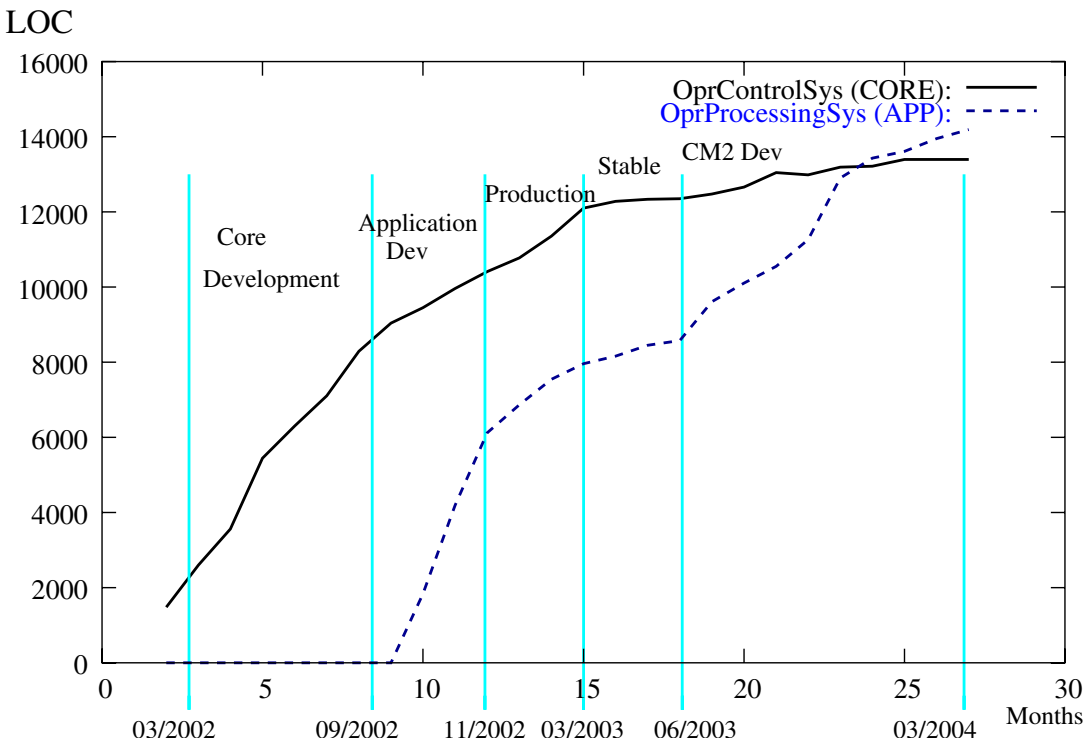

Fig. 4. The number of lines of code (LOC) in the two packages of the code repository. The plot reflects the different development phases: during Core Development (1) there is development only on the core package, at high rate. In Application Development (2) the core development slows down, while the application code grows extremely fast. Three months later the system starts to be used in production: application development slows down, and most of the new code addresses issues emerging from the real-world test. Both packages see the same amount of activity. Few months later the main development effort is considered done, and the system is Stable and works in a maintenance only regime. There is very little code contributed in this period, the curves are almost flat. This also corresponds to developers switching to other projects in the same period. Then, a major change in the experiment's Computing Model (CM2) requires a major update to the control system. It is remarkable that very little is done on the Core package, while the Application package grows fast. This shows how much the core code was reused, and how the flexibility of the core allowed a rapid adaptation of the system to a changed environment.

modular architecture, frameworks and services.

C. Acknowledgements

This work builds on a large body of development by the BaBar Computing Group, and would not have been possible without a strong collaborative effort.

REFERENCES

[1] I. Gaponenko, *CDB - Distributed Conditions Database of BaBar Experiment*, CHEP2004, Interlaken (Switzerland), 29 Sep 2004 [http://indico.cern.ch/contributionDisplay.py?contribId=316&sessionId=6 &confId=0].

[2] 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 [http://www.hep.net/chep98/PDF/47.pdf].

[3] T. Ylonen, *SSH - Secure Login Connections over the Internet*, Usenix 1996, San Jose, CA (USA) [http://www.usenix.org/ publications/library/proceedings/sec96/ylonen.html].

[4] P. Elmer et al., *Distributed Offline Data Reconstruction in BaBar*, CHEP2003, San Diego (USA), 24-28 Mar 2003 [http://www.slac.stanford.edu/econf/C0303241/proc/papers/ MODT012.PDF].

[5] F. Safai Tehrani, *The BaBar Prompt Reconstruction manager: A real life example of a constructive approach to software development*, Comput. Phys. Commun. 140, 56 (2001).

[6] http://www.objectivity.com

[7] P. Elmer, *BaBar computing: from collisions to physics results*, CHEP2004, Interlaken (Switzerland), 29 Sep 2004 [http://indico.cern.ch/contributionDisplay.py?contribId=502&sessionId=21 &confId=0].

[8] A. Ceseracciu et al., *The new BaBar Data Reconstruction Control System*, CHEP2003, San Diego (USA), 24-28 Mar 2003 [http://www.slac.stanford.edu/econf/C0303241/proc/papers/TUDT011.PDF].

[9] A. Ceseracciu, T. Pulliam, *The evolution of the distributed Event Reconstruction Control System in BaBar*, CHEP2004, Interlaken (Switzerland), 29 Sep 2004 [http://indico.cern.ch/contributionDisplay.py?contribId=210&sessionId=9 &confId=0].