COMMON TEMPLATE AND ORGANISATION FOR CERN BEAM INSTRUMENTATION FRONT END SOFTWARE UPGRADE

A. Guerrero, S. Jackson,
CERN, Geneva, Switzerland

Abstract

To enable the use of PS and SPS as LHC injectors, and to comply with the standards and directives laid down by the CERN Accelerator Controls group (AB/CO), most of the existing software associated with Beam Instrumentation required re-engineering. Faced with the magnitude of this software upgrade task and the necessary new developments, we designed a set of internal rules, generic tools and templates. These facilitate, and in some cases automate, the development of the entire software chain from the real-time to the user interfaces. The benefits of our approach, and results obtained so far, have been assessed by AB/CO and our framework is now considered as the starting point to develop the CERN accelerator standard for front-end software. This initial project and its first applications will be presented.

INTRODUCTION

Historically, CERN has relied heavily on computing technology to provide the control system and instrumentation necessary for controlling its complex operation.

The recent need to (re) develop many systems for the LHC era prompted the 'Beam Instrumentation Software Common Template & Organisation' (BISCoTO [1]) project, which aimed to provide a flexible environment, allowing the rapid creation of common generic real-time systems.

REAL-TIME SERVER ARCHITECTURE

Over the years, many programming languages have been used in AB/BDI (CERN Beam Diagnostic and Instrumentation Group) [2]. Real-time servers have been developed with languages such as Modula 2, C or C^{++}. Based on available resources, it was decided that servers would be developed in C, and corresponding Graphical User Interfaces (GUI) in Java. C was chosen because C++ was not mature enough on our LYNXOS 2.5.1 version and also because some future framework users were not ready for C++.

Real-time servers in AB/BDI have been implemented using varying architectures. To standardise, the BISCoTO skeleton architecture was devised, encapsulating as many generic features as possible whilst minimising system specific features to avoid redundancy. The aim was to give developers an operational real-time server, which requires tuning for an instrument's needs. Figure 1 shows the skeleton's architecture, and the processes (indicated by ovals) involved. Inter-process communication is via message queues and shared memory. Using a process-based architecture rather than threads simplified diagnostics and maintenance and allowed each subsystem to be developed by different team members. On the other hand, the use of message queues implied an additional overhead between subsystems sharing information. This burden was considered acceptable, but care was taken to minimise the effects of these overheads.

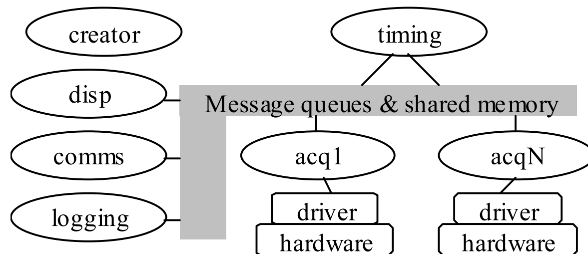

Figure 1: Real-time architecture.

The Creator process

The creator is charged with starting all processes in a specific order. In order to synchronise the launching sequence, the creator is informed by the launched processes of their readiness via a flag in shared memory. Once the process has been created, it is surveyed in case of abnormal termination. Should this happen, the creator may restart the process depending on its configuration.

The Dispatcher (disp) process

The dispatcher acts as an intermediary for forwarding messages to one or more processes. For example, if 2 acquisition processes need to be informed of an event from the timing process, they could subscribe directly. Instead, they subscribe to the dispatcher, which makes a single subscription on their behalf. The advantage of this dispatcher layer is the possibility for a task to subscribe to events coming from entirely different event sources (i.e. machine timing, specific hardware, user command) and 'block wait' for all these in one call.

The Comms process

This process manages all clients' requests for data or actions from the BISCoTO server.

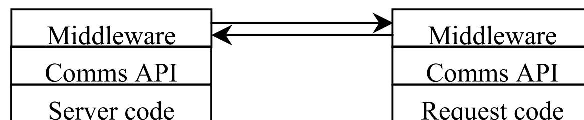

Figure 2: Detachment of client and server request from middleware by intermediate API

The comms process was designed so that the request code, i.e. the server code that handles the client's request, is detached from the middleware on which the request was made. This allows any BISCoTO system to be migrated to any middleware without modifications to the server's code. We achieved this via an intermediate 'middleware neutral' API as shown in fig 2. To introduce a new middleware, a new Comms process is written for *all* systems, hooking into their request code. Detaching the code from the middleware also gives the possibility to start several Comms processes simultaneously, allowing the benchmarking of different middlewares, as well as providing several communication mediums in case a particular middleware fails or legacy constraints. At present Comms processes exist for SLEquip [3], CMW [4], and raw TCP/IP.

The Logging process

To make sense of asynchronous logs from a real time multiple-process architecture, logging needs to be controlled. Writing to log files is a lengthy operation that can't be afforded by real-time processes. Buffering messages and flushing later is okay, but the death of a process could result in the loss of information. The logging process was created to solve this by providing a mechanism for processes to log in a fast reliable way. A process opens a log instance in shared memory, containing the various attributes, such as verbosity, filename, maximum file size, etc. To log, the message and a pointer to this log information is sent via a message queue. The dedicated logging process (running at low priority), then writes to the file when CPU time is available.

A standardised log message format, allowing 4 logging levels ('Status', 'Warning', 'Non-Fatal' and 'Fatal'), and time-stamping was created. The 'LogController' GUI, is then used to explore 'logs' known to a system's logging process. With the standard log format, the GUI can filter one or several merged logs based on severity or keywords. The GUI can also instruct the logging process to discard messages with a severity lower than a given level, thus allowing run-time verbosity control.

The Timing and Acquisition (acq) processes

To synchronise with the beam in SPS, the software is driven by events from the SPS timing network. A BISCoTO system typically has one or more acquisition processes, each driven by these events. The skeleton has one such acquisition process, set up to respond to common SPS events, in which the developer can place instrument specific code. The ability to create multiple acquisition processes also allows a BISCoTO system to potentially control several pieces of hardware with dedicated processes.

THE CONFIGURATION GUI

To avoid modifications to the skeleton, all processes are data driven. The template is written in C, so uses structs to handle data. Historically, clients were also written in C so the same structs where referenced. Referencing these structures from Java however, is not possible, so an intermediary standard was devised to allow the mapping of data between C and Java classes. The configuration GUI handles the definition creation and the automatic generation of code mapping data between C and Java.

Limiting the number of data types

Before standardising the definitions, the number of data-types in a definition had to be reduced. Data types available only in C such as unsigned types were eliminated, whilst other types, like enumerations were standardised. Some new types allowing bit-wise manipulation were also created, ultimately based on shorts and ints.

The definition GUI

The developer uses this Java GUI to create and customise definitions for their system. If definitions change, the GUI recreates the corresponding C/Java code. With this automation, a server can be built with the skeleton, a library generated by the GUI (for data management and manipulation using dynamic querying) and user code for accessing the hardware. This allows the template code to remain generic, and maintainable by all.

The Standard configuration editor

The majority of configuration settings must be entered by the user, and read by the system on initialisation. The extension of the definition GUI to include a generic data input tool seemed logical as all data was defined in a standard way. The data input GUI, consists of a data driven table for modification of configuration data, with file selection via a tree populated with all known data files. The definition not only allows the determination of column names and maximum records for the table, but also allows generic data validation to be incorporated. The resulting data entry tool is both powerful, and more importantly free to the developer.

Special definitions and Actions

Most servers perform actions that often require parameters, and sometimes return composite data types. Developers have historically implemented their actions in an ad-hoc way, providing an API and documentation for clients. To standardise this, BISCoTO servers extend the data definition concept to trigger actions. Using a predefined action naming convention, the developer can define their actions, along with further definitions specifying input/output parameter formats, thus allowing the corresponding code generation.

Versions and advanced data management

The automatically generated libraries are released under a new version when a definition changes. This allows definitions (and corresponding data) to be changed without

affecting systems relying on the old definition. This is necessary when the developer needs to work on their server, whilst an operational instance of the server exists. Without versioning, a simple change in a definition could have side effects on the operational servers.

As well as freezing definitions, the freezing of data using 'states' is also available. A state is effectively a snapshot of all configuration data at a point in time that can be restored later. States are used in systems where configuration depends on different beam types in the SPS.

HARDWARE AND DRIVERS

Access to hardware is normally made in the acquisition process via a driver. It was clear that a lot of the driver code for accessing hardware registers could be generated from a BISCoTO definition. Automating the driver creation not only reduced the amount of bugs in drivers, but also gave other developers a standard description of the device without examining hundreds of lines of code.

Having defined the driver's registers, the developer runs a program that generates and compiles the appropriate code for the driver and its access library.

NAVIGATION BY PROPERTIES AND ACTIONS

The generated Java code follows a convention similar to that defined by Sun's Java Bean convention, meaning run-time exploration of a class via 'get' and 'set' methods is possible. This was exploited to provide a 100% data driven exploration tool (the navigator), which needs only the location and middleware ID to begin exploration. The skeleton has built in actions allowing interrogation of available properties and actions. Building a tree until it reaches a leaf, the navigator dynamically loads the Java classes that build panels allowing the *get*ting and *set*ting of properties. If the leaf is an action, the parameter classes are loaded and used to construct input/output panels for the action. Like the data input tool, the navigator is available free for all BISCoTO systems.

Replacing the hundreds of test programs created for systems in the past, the Navigator has been developed to include many features such as graphing and interaction with MS Excel. As the tool is generic, all these features are available to all systems developed under BISCoTO.

CONCLUSION

The BISCoTO project had to provide a powerful and generic framework in which a developer could rapidly develop a server. This has been achieved by making a generic template which is almost completely data driven. The project has exploited the data driven nature of the systems further, by producing *free* generic tools for the developers and users of BISCoTO systems. This product received an immediate success. It has been operationally used for every new BDI developments.

The pertinence and potential of the product is now such that it has been selected by AB/CO to be the best starting point for the Divisional standard for front end software development. This new architecture [5], named FESA for Front End Software Architecture, now fully object oriented with server framework in C++, is currently being designed and prototyped by a joint-team comprising both AB/BDI and AB-CO engineers. Its first version will be delivered to the equipment group developers in November 2003 and deployed operationally on several front-ends in the LHC injector chain in 2004.

REFERENCES

[1] http://sl-div-bi-sw.web.cern.ch/sl-div-bi-sw/Activities/GenPurp/ForUs/BISCoTO/entry.htm

[2] http://sl-div-bi.web.cern.ch/sl-div-bi

[3] http://slwww.cern.ch/~pca/equip/sl4/dum.html

[4] Kris Kostro, Jens Andersson, Steen Jensen, Franck Di Maio (CERN) N. Trofimov (IHEP, Protvino, Moscow Region):" The Controls Middleware (CMW) at CERN - Status and Usage" (these proceedings)

[5] M. Arruat, A. Guerrero, J-J. Gras, S. Jackson, M. Ludwig, J-L. Nougaret (CERN): "CERN Front End Software Architecture for Accelerator Controls" (these proceedings).