

RUN CONTROL TECHNIQUES FOR THE FERMILAB DART DATA ACQUISITION SYSTEM*

G. OLEYNIK, J. ENGELFRIED, L. MENGEL, C. MOORE, R. PORDES,
L. UDUMULA, M. VOTAVA

*Fermi National Accelerator Laboratory, PO Box 500, Batavia, IL
60510, Tel: 708-840-2430*

E. VAN DRUNEN, G. ZIOULAS

University of California at Irvine, California 92717, U.S.A.

DART is the high speed, Unix based data acquisition system being developed by the Fermilab Computing Division in collaboration with eight High Energy Physics Experiments. This paper describes DART run-control which implements flexible, distributed, extensible and portable paradigms for the control and monitoring of data acquisition systems. We discuss the unique and interesting aspects of the run-control - why we chose the concepts we did, the benefits we have seen from the choices we made, as well as our experiences in deploying and supporting it for experiments during their commissioning and sub-system testing phases. We emphasize the software and techniques we believe are extensible to future use, and potential future modifications and extensions for those we feel are not.

1 Introduction

DART [1 - 3] is a collaboration between the Fermilab Computing Division and Fermilab experiments to produce a common data acquisition (DA) system for eight experiments taking data in the '96-'97 running period. The overall DART system architecture [4-6] has been defined now for a couple of years, and has been under development and use now for 3 years. To meet the commissioning needs of the experiments, DART has been built from the "bottom up", within the architecture, so that portions of it could be deployed prior to the completion of the whole system. The complete system will be deployed at experiments prior to the start of the run.

Since DART needs to support a variety of DA architectures, it is very flexible and not constrained to any one hardware architecture. The DART experiments have a variety of different intelligent components that require control and configuration, and hence DART is a fully distributed DA system. Since we did not wish to tie ourselves down to a single computer vendor, DART software has been developed to be highly portable (to various Unix platforms, and also to VxWorks or other operating systems that use BSD sockets). We have applied our experiences from constructing past data acquisition systems to provide a more cohesive and uniform system. Because of all these reasons, DART can be fairly easily incorporated into new experiments or test-beams, and many aspects are applicable to other types of distributed system, not only data acquisition systems.

In designing such a flexible distributed system, we developed numerous concepts and techniques which enable DART software to be incorporated into future systems, and its

* This work is sponsored by DOE contract No. DE-AC02-76CH03000

concepts to be applicable to future technologies (e.g. substitute distributed applications with distributed objects in this paper).

DART run control is built around a number of distributed frameworks. For the discussions in this paper, the key frameworks and components are:

- A client-server command multicasting framework for multicasting run control commands to application “groups”
- A client-server information server from which applications fetch configuration parameters, record data acquisition history, and deposit DA statistics and rates for monitoring.
- An rlogin “multiplexor” from which all DA applications are started. From a single program, rlogin sessions are created to the DA nodes, and commands are fed to their shells.
- Scripts that use the rlogin multiplexor to start up the DA applications from a list of nodes and a set of shell scripts to run on each node.

A better understanding of these components and the rationale for their creation will be made clearer in what follows. A fairly complete description of DART run control can be found in [7].

In this paper, we will concentrate on the following specific DART run control concepts and techniques, and how we implemented them:

- node transparency - addressing communications through functional names rather than node names and using the client-server model extensively.
- replicated applications - efficiently handling applications duplicated across nodes
- application specialization - specializing a general application for a specific node
- node aliasing - making the DA independent of node names for maintenance purposes
- parallelizing - parallelizing start-up and control across nodes for minimal latency
- synchronization & draining - How the distributed DA applications are properly drained on a stop, and properly primed for data taking on a start
- application cleanup - removing all DA applications and their residue so a “fresh start” of the data acquisition system can be accomplished

While some or all of these concepts may be familiar to most, I have never seen them recorded in print in the context of HEP experiments, so hope that this exercise will be beneficial to future endeavors.

2 Node transparency

We set out to develop a control system in which applications communicated via functional addressing, rather than the conventional rpc-like point-to-point node addressing. What we have developed is a run control system that requires node names only once, when the DA applications are started through an rlogin multiplexor (even then these node names can be “aliased” - see below). This is accomplished through a client-server run control command multicasting framework. In this framework, applications register with any number of run control “groups”. These groups are arbitrary strings, but each establishes a communications channel. Any application can “multicast” a command to one of these groups, and all applications registered with that group, and only those, will receive the command. Appli-

cations need only know the multicasting server's address, which they get through an environment variable. We defined several general groups for use in the DA:

- trigger-manager - controls beam spill gating (computer busy)
- gateway - For intermediary sub-event collection applications
- filter - For data reduction applications
- front-end - For Intelligent readout controllers
- logger - For data logging applications
- end-of-spill - For applications needing to know about EOS
- beginning-of-spill - For applications needing to know about BOS
- snapshot - For applications that can dump "snapshot" diagnostics

and any arbitrary other groups can be dynamically added to the system by registration.

3 Replicated Applications

DART experiments have a number of replicated components. For example, in order to absorb data bandwidth, KTeV has three replicated independent event building/logging VME crates. There are two concerns here: configuring and controlling the replicated applications. For the former, we use an information client-server framework. A central server keeps configuration parameters and their values in a simple keyword/value pair indexed file. By convention, we impose upon this store a hierarchical keyword namespace, with the top of the hierarchy based upon the application name. Parameter keyword - value pairs take on the general form:

[<node>:]/<application-name>/<parameter-name> <values(s)>

where [] denotes optional node specificity (see below). We use Unix directory structure like syntax for the keyword name space, and related parameters can be grouped through this hierarchy. Combined with wild-carded (regular expression) fetches, this allows replicated applications to fetch the same configuration information very simply. Wild-carded fetches fetch a related group of parameters across the network so that they can be processed locally. This results in low latency and better utilizes the network.

For control, each of the replicated applications registers with the same run control multicasting groups, so receive the same commands, and will respond in unison to them (see the discussion on synchronization below).

4 Application Specialization

Some applications require parameters that are specific to the node they run on. KTeV is again a good example. They have an additional VME crate to which a sample of events are sent for in-depth analysis and filtering. This means the filter and logger applications on this node require different configuration parameters than the same applications in the other VME crates. Another example is our buffer management system, which may require different parameters (e.g. amount of shared memory) on different nodes.

To handle this, we have built into our information services what we call “node specificity”. Normally, an application makes a request to fetch a parameter for node “localhost”. The server will first try to fetch the parameter with the node name prepended to it: <node>:<parameter-name>, and if not found, will then try to fetch <parameter-name>. These semantics allow the application to pick up a parameter specific to its node if it exists, otherwise to pick up the “generic” parameter (if it exists). An application can also select a parameter specific to another node, or the generic only. These semantics also apply to wild-carded fetches.

If an application also must specialize in command processing, then it must join different run control groups. The required group can be communicated via configuration parameters, and thus can be specialized through configuration parameter node specificity.

5 Node Aliasing

Experiments need to keep hot spares of some processors so they can replace broken components in a plug and play fashion. Such a spare is pre-configured with a different node address, hence the changed node name needs to be made known to the system.

Node names are used in two places in DART. The first is in the rlogin multiplexor that creates rlogin sessions to the nodes that make up the DA (it does this from a list of nodes). The second is in the node specific parameters in the information server database, for which there can be many. What is needed is to be able to use aliases for the nodes in these places, and to be able to redefine the node associated with the alias once and have it automatically propagated everywhere else.

We built node aliasing into the information server and its database. We have alias (and unalias) commands. The alias command defines the association between a node and its alias, in both directions, in the configuration database. When an application connects with the information server, the server checks to see if there is an alias for it in the database. If so, it does node specificity checking for that connection with the alias instead of the node name, otherwise it uses the node name. For starting up applications with rlogin multiplexor from a list of nodes, the script that does the start-up first checks, via the information server, if the name in the list is an alias for a node, and if so, uses the associated node name to create its rlogin session.

6 Parallelizing Application Start-up and Application Control

The rlogin multiplexor allows multiple rlogin sessions to be created in parallel (the logins proceed as a separate thread), and commands to be sent in parallel. Since we use rlogin, no software is required on the remote nodes. Commands are sent to a session via session names rather than node names, and we support sending commands to wild-carded (regular expression) session names. For instance, we construct a session’s name from its node name and operating system. Thus we can “broadcast” setenv commands to Unix shells, putenv to VxWorks shells, etc., with a single command. All proceed in parallel minimizing start-up time; the DA start-up time is determined by the node taking the longest time to start.

7 Synchronization of starting and stopping a run

Data acquisition systems typically have many levels of buffering from the front-ends down to the logging of data. It is important to make sure the data is sequentially drained from one level to the next when stopping the run in order not to lose any of the data. This is usually done in one of two ways: sending a special end-of-run event that follows the last data through the levels, or sequencing the stop command through the levels, not moving on to the next level until the previous is finished draining. We chose the latter because the end-of-run event solution is not general, and the latter was natural for the group multicasting paradigm.

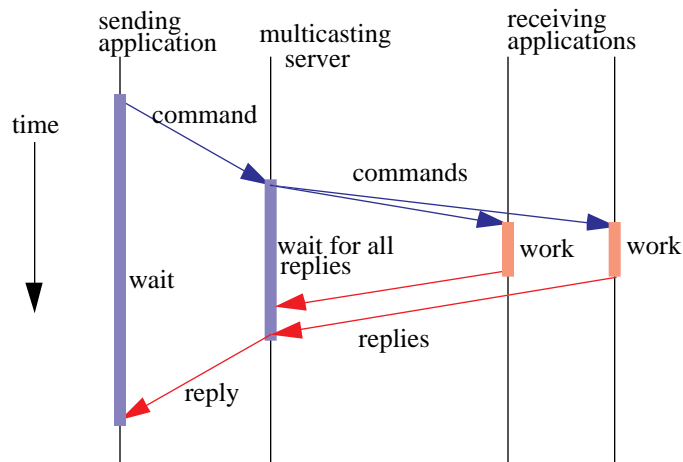


Figure 1: Synchronization with single threaded multicasting server

When a command is multicast to a group by the multicasting server, it rendezvous waiting for completion status from all members of the group (i.e. registered applications) before processing the next command. This is illustrated in the figure above. Thus draining on a stop consists of first sending the stop command to the trigger-manager group (which disables new triggers), then to the front-end group, ..., finally to the logger. Each application must in turn, not send a completion status back until it has drained all of its events. If all applications follow this rule, then they can be assured that all applications in levels before them have already been drained.

Similarly, starting a run is synchronized by sending to the logger group first, and last to the trigger-manager, which enables triggers.

8 Application cleanup

We provide a mechanism for applications to register to be automatically “cleanup up” (kill -TERM under Unix) so that a “fresh start” of part, or all, of the DA can be performed. We use guaranteed mechanisms (e.g. on Unix, registering by opening a “session” file and then

using fuser to kill all applications with the file open) to this end, and functionality is built into the rlogin multiplexor and fresh start scripts to cleanup one or all of the DA's sessions.

9 Conclusions

We have implemented a highly flexible run control, whose concepts and implementation we expect will have a long future with a wide range of application. We presented a number of ideas we used to make this possible. We will continue to build in new features and higher level user interfaces as necessary. In particular, we anticipate addressing some or all of the following:

- Graphical configuration management layer over the information services
- A “snapshot” like heartbeat of system applications
- Multi-threaded run-control command multicast server

10 References

1. G. Oleynik et al, DART - Data acquisition for the next Generation Fermilab Fixed Target Experiments, IEEE Transactions on Nuclear Science, Vol 41, No 1.
2. See Accompanying paper, “Extending DART to meet the Data Acquisition Needs of Future Experiments at Fermilab”, this conference
3. R. Pordes et al, Fermilab's DART DA System, Proceedings of CHEP94.
4. G. Oleynik R. Pordes, DART System Requirements, Fermilab CD PN-468
5. G. Oleynik R. Pordes, DART System Architecture, Fermilab CD PN-469
6. DART documents are available at URL <http://www-dart.fnal.gov:8000>
7. Fermilab DART Run Control, Proceedings Real Time '95.