FERMILAB BEAMS DIVISION ALARMS PROCESSING SYSTEM

Seung-chan Ahn
Fermi National Accelerator Laboratory
PO Box 500
Batavia, IL 60510, USA

Abstract

ALARMS, the distributed alarms processing software for the Fermilab Beam Division is described in detail. Also reported is the experience with this new alarm processing software that has been operational for more than a year.

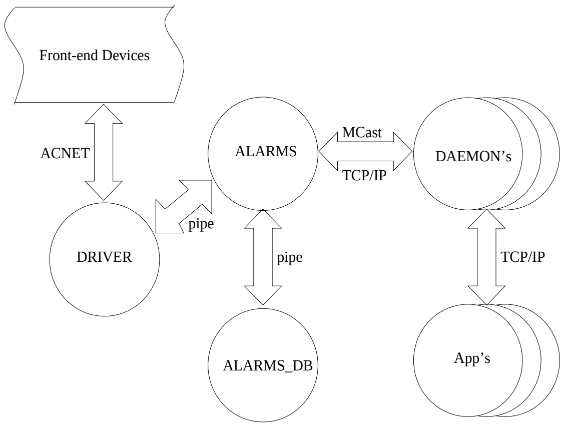
1 OVERVIEW OF THE SYSTEM

ALARMS processes the alarm messages from various devices that collectively monitor and control the Fermilab accelerator operations [1]. An alarm is a generic term indicating anything from the report of a device failure that stops the beam to a mere informational message that can be removed from display after a few seconds without much consequence. Perfect ALARMS will deliver all these messages to the intended consumers.

ALARMS is a system of distributed software components in an OS neutral environment. An obvious benefit of distributed computing is one can modify a part without affecting the rest of the system. The main part of ALARMS can be divided into 4 different functional units. Their tasks are (1) interaction with front-end computers, (2) database look-up for device properties, (3) build alarm messages, and (4) distribute messages to interested parties. While the first 3 tasks are performed in 3 separate processes on an UPS backed up node the last component is distributed over 50 plus nodes. The division of tasks roughly coincides with the division of communication methods and sources. See schematics below for illustration.

The auxiliary part consists of a suite of utility programs that inspect the alarms, monitors the ALARMS processes itself, and provides various alarms statistics for applications. Some components can talk with other programs over the network. The suite also includes a Java application program for alarm display that communicates with distributor nodes (above task 4), and with front-end computers.

ALARMS exploits the benefits of distributed computing at all levels. Each component of ALARMS makes use of threads extensively. The utilization of threads helps write a cleaner and easily maintainable code. However, it demands a careful design of the software to maintain data consistency among different computers, different



processes on a computer, and different threads within a process. To insure smooth flow of program and prevent process lock-up, it is necessary to assign thread priorities properly and control bi-directional inter-process I/O.

2 ALARMS_DRIVER

ALARMS_DRIVER directly communicates with front-end computers via the Fermilab developed ACNET protocol. Only one copy of DRIVER runs on a designated node. DRIVER initiates the whole ALARMS system activities and the flow of alarm messages.

DRIVER launches 2 subprocesses and establishes pipe connections among all; ALARMS to process alarms, and ALARMS_DB to access the device database. DRIVER exits then either of the two subprocesses exits.

DRIVER initialization proceeds as follows. After successfully launching the 2 subprocesses DRIVER waits for ALARMS to initiate the front-end nodes alarm initialization. When ALARMS is prepared to process alarms data, it instructs DRIVER to request all front-ends to refresh local alarms data. DRIVER waits for all front-end nodes to respond but it does not assume all front-ends are alive. Responding front-ends are immediately registered as valid. All subsequent alarm data from them are passed to ALARMS. DRIVER makes only a few more attempts to wake up the front-ends.

DRIVER ignores alarms from unqualified nodes. They are often test or obsolete nodes that emit at best uninteresting alarms that should not bother the accelerator operation. When a new front-end node is added to the system a separate program notifies DRIVER to reload the list of valid front-end nodes from database. For this purpose and possibly others DRIVER always leaves a TCP socket open.

DRIVER holds data when ALARMS has a large backlog of alarm data to process. When situation doesn't improve and as a result DRIVER holds significantly large number of alarm data DRIVER instructs front-ends to hold data to themselves. This provision is made as an insurance policy but it was never used in practice even in the case of power failure and recovery.

3 ALARMS

The ALARMS process builds an alarm message based on the device alarm data that front-end node assembled, and the corresponding database information obtained from ALARMS_DB. ALARMS multicasts the message using UDP protocol. ALARMS listens to itself but it does not check whether the multicast was successful. Instead ALARMS listens to DAEMON multicast messages.

ALARMS process assigns a unique sequential number to each alarm including user requests for specific actions. This sequential number is useful in particular to maintain data consistency with other components of ALARMS system. ALARMS logs all alarm data both in its raw and processed formats. The redundancy of alarm data is justified considering the low cost of storage devices.

ALARMS process also maintains a global shared memory backed up by a physical file. The shared memory contains the most recent 20000 alarms that are readily available for other processes. The same shared memory also holds process state information that external process can interrogate.

ALARMS initializes itself as follows. It first establishes shared memory and initializes all the counters. Then it sends a single BIG CLEAR message to DRIVER, which subsequently notifies all front-ends to reset the alarm data.

To facilitate the proper initialization of DAEMON processes ALARMS exchanges messages with DAEMON using TCP/IP protocol.

Several threads of ALARMS run concurrently. The alarm receiver and message builder threads are set at the highest priority. All the other threads including the logging thread at a normal priority. All threads are cooperative, i.e., they yield whenever possible and they certainly yield while blocking for I/O.

ALARMS process logs alarms data at a fixed time interval or fixed sequential number interval, whichever comes first. Usually the data logging is done at every few minutes.

4 ALARMS_DB

ALARMS_DB process makes database queries by calling Fermilab controls library CLIB routines, which in turn make SQL queries to central database engine. CLIB caches database query results internally, and subsequent CLIB calls to the same device is equivalent to a mere memory look-up and hence extremely fast. Since BB communicates with only ALARMS and DRIVER via pipes, and no other interrupt can possibly happen, there is no reason to thread DB. The main program is encased in a single infinite loop centered on the pipe input.

DRIVER registers itself as a listener of the database changes at the program start-up. Upon the receipt of ACNET messages of device database changes, DRIVER notifies ALARMS_DB of such happenings. At that point ALARMS_DB clears the cache for the affected devices. ALARMS_DB device database is current at all times. The main device database engine we use is Sybase.

5 ALARMS_DAEMON

ALARMS_DAEMON distributes alarm messages to application programs via TCP. It can serve up to 32 applications at a time. The base data structure of DAEMON is identical to that of ALARMS. In fact the same source code is compiled into DAEMON and ALARMS. DAEMON runs on multiple nodes as a background batch process. On VMS, upon exit DAEMON sends e-mail to the program keeper and it restarts itself in the exit handler by submitting the batch job.

At startup, DAEMON is not synchronized with ALARMS. It establishes the data consistency with ALARMS as follows. First it requests ALARMS for active alarms. While ALARMS is sending active alarms stored in a queue, ALARMS may have received more alarms from the front-ends, or some alarms may have been cleared. Hence the initialization incomplete until the first multicast message is received, when DAEMON determines whether it missed any alarms (both good and bad). If necessary, DAEMON makes another request for more messages to fill up the message gap.

There is a different kind of DAEMON that maintains a complete data consistency with ALARMS (current data consistency plus history data). This DAEMON collects history data and sends out e-mail at a regular time. It is designated in a command file shared by all DAEMONs.

UDP being a connectionless protocol, some messages are lost. DAEMON detects the occurrence of missing messages by checking the sequential number of each alarm. As DAEMON discovers missing alarms it multicasts a request for rebroadcast. To reduce the network traffic, up to 2 DAEMONs send UDP multicast messages for an identical request. While they may be in different status of message reception, they attempt cooperation. DAEMON does not respond to other DAEMON messages in any other active fashion.

DAEMON maintains an open TCP socket for applications and creates a new thread for each application. Application programs send the front-end commands to DAEMON. DAEMON checks the validity of the requester and the content of command, then assembles the message and multicasts it. ALARMS picks it up and passes it to DRIVER, which assembles a correct ACNET message and sends it to the appropriate front-ends.

6 STATUS AND EXPERIENCES

Presently all the main ALARMS components run as detached background processes under VMS operating system. At the moment DAEMON was ported to two Unix platforms Solaris and FreeBSD. ALARMS is not ported due to lack of a complete CLIB equivalent in Unix systems.

The overall ALARMS has been operational for more than a year. On a typical day ALARMS processes about 10,000 alarms. About half of all alarms are caused by a couple of dozen devices.

Typical run time of ALARMS is about 2 weeks. That is, in about 2 weeks some kind of unplanned happenings (network disruption, inter-process communication error or database lookup failure) cause ALARMS terminate. When ALARMS stops running DAEMON processes recognize this fact immediately because each DAEMON keeps its own heartbeat generator that checks ALARMS heartbeat. The application programs connected to DAEMON sense the loss of heartbeat and take proper actions.

Accelerator controls front-ends come in a wide variety of flavor ranging from most modern components to legacy systems and ALARMS accommodates all.

7 OUTLOOK

The notion of distributed ALARMS resonates with the concept of OS neutrality. Along with DAEMON, porting the entire ALARMS system to Unix and NT would be a logical next step. With the advent of the Java language and its environment this next step was simple and easy.

As a start we developed an alarm display in Java under NT. It is a user configurable Java application that combines the functionality of the VMS alarm display and its related programs.

Another Java program is being designed for graphical presentation of alarm history data. The Java program runs as both applet and application.

For some critical devices with high priorities it is not enough to merely display their alarms. Machine operators do not necessarily look at the alarm display all the time. To grab the operator attention, the program needs to enunciate the alarm and operator must acknowledge it. We are currently implementing a voice synthesizer for the alarm display using IBM ViaVoice.

We are also investigating the feasibility of using Mathematica as a graphics engine for sophisticated graphical presentations of alarm data. The JDK for Mathematica and ViaVoice are currently in alpha. It may be necessary to adjust the programming details or even the design itself from time to time as the technology matures, but we believe that we are in the right direction in this venue.

8 REFERENCE

[1] S. Ahn, “Fermilab Tevatron Alarms Processing System”; ICALEPS ’97 Beijing, 1997.