Fermi National Accelerator Laboratory

FERMILAB-Conf-94/104

Data Flow Manager for DART

D. Berg et al

Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510

April 1994

Presented at the *Conference on Computing in High Energy Physics 94*,
San Francisco, California, April 21-27, 1994

Disclaimer

Data Flow Manager for DART*

David Berg, Dennis Black, Dave Slimmer
Online Systems Department/Computing Division

Jürgen Engelfried
E781
Physics Department/Research Division

Vivian O'Dell
Special Assignments/Computing Division

Fermi National Accelerator Laboratory
P.O. Box 500
Batavia, IL 60510

Abstract

The DART Data Flow Manager (**dfm**) integrates a buffer manager with a requester/provider model for scheduling work on buffers. Buffer lists, representing built events or other data, are queued by service requesters to service providers.

Buffers may be either internal (reside on the local node), or external (located elsewhere, e.g., dual ported memory). Internal buffers are managed locally. Wherever possible, **dfm** moves only addresses of buffers rather than buffers themselves.

1 Introduction

DART [1], a collaboration of fixed target experiments, developed a buffer management system [2] that will satisfy their varied requirements [3] with a single coherent facility.

dfm requester tasks assemble lists of pointers to buffers, representing, for example, various fragments of an event, and schedule them to **dfm** provider tasks, which perform services such as logging events to tape. Each experiment designs its own requesters and providers, building on a framework of **dfm** functions.

A tool, **snapshot**, was also developed for debugging and performance measurement. This tool runs as a separate process that spies on **dfm**'s control structures.

2 Requirements

The basic requirements for **dfm** were that:

- The data flow manager should be almost invisible to the experimenter, making the use and integration relatively easy.
- There must be well defined interfaces between the data flow manager and other data acquisition components, e.g., run control, logger, Hoist [4].
- The data flow manager must either avoid, or be able to detect and gracefully recover from, deadlocks, insufficient resources, etc.
- The data flow manager should impose no strict real time requirements on Unix workstations used as filter farms. The occasional slow analysis of an event must not halt the overall flow. Even if a processor completely drops out, data acquisition must not halt (though it may slow proportionately).
- Similarly, flow control problems with back end workstations must not interfere

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

with data acquisition.

- There should be no restrictions on the content of buffers.
- Data buffers should be externally tagged with characteristics that can be used to control flow, buffer selection and distribution among sinks, etc.
- A data buffer may be represented to the buffer manager directly, as a collection of data to be moved, or indirectly, as the address of the data.
- Beyond those necessary for fault tolerance, there are no requirements for dynamic re-configuration during a run.
- In addition to the buffer manager routines being used by programs written in C (or C++), there must also be FORTRAN bindings.

3 Buffers

Local buffers are managed as a pool that is pre-allocated using a distribution of discrete sizes specific to each experiment. A request for a buffer is made by size, and satisfied by the smallest available buffer meeting the requirement.

There are three kinds of buffers. There may be any number of buffer classes defined for each kind, up to an aggregate of 255 classes. Classes are defined by the experiment as needed. The three kinds are:

- *external* - buffers not managed by **dfm**.
- *internal* - local buffers, including buffer lists, managed by **dfm**.
- *sub-buffer* - a local buffer may be divided arbitrarily into sub-buffers; each such parent buffer defines a separate class.

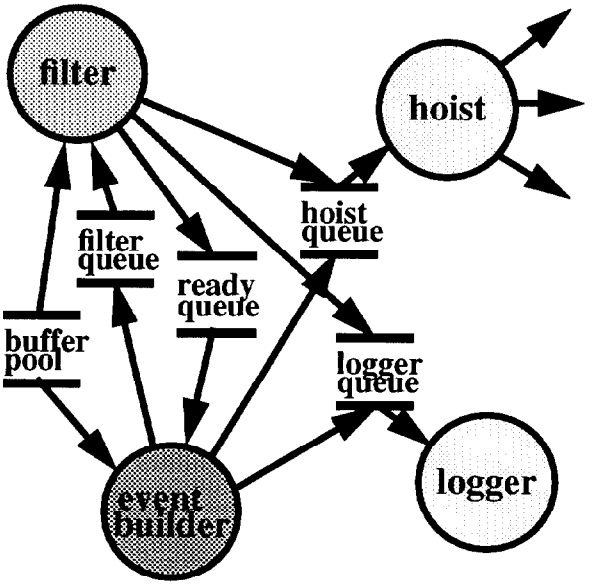
Associated with each internal buffer is a buffer descriptor consisting of the buffer class and buffer use count. The buffer descriptor physically precedes the buffer in memory.

4 Buffer List

A buffer list is a data structure containing a set of buffer locators and control information associated with a single event or other unit of data. The pointer to this control structure is the item that is passed onto message queues between requesters and providers. Because the buffer list is itself an internal buffer, it has an associated buffer descriptor. Furthermore, buffer lists may themselves be included as buffers in other buffer lists.

5 Service Requesters and Providers

Service requesters reserve buffers and build or modify buffer lists, which they schedule to service providers. One list may be scheduled concurrently to many providers. Service providers access the lists and buffers; when a service completes, it attempts to return the buffer list to the free pool. A single process may be a requester, a provider, or a combination of both.



Requesters schedule buffer lists by specifying a service group. All scheduling is limited by a time out period, so if any queue is full, the requester will block until there is room, or until the time out.

- *broadcast* - attempt to post the list to every provider in the service group.
- *round robin* - take providers in the service group sequentially, and attempt to post the list to the next one. If the queue is full, skip to the next one.
- *next available* - maintain a ready queue of providers, and attempt to post the list to the first provider on the queue. If the ready queue is empty, or the service

queue is full, block until a provider becomes ready, or until the time out.

Providers are responsible for acquiring buffer lists from their respective queues, and blocking if their queues are empty, or until a time out. Providers may specify several optional posting requirements, which they may also change at any time. If requirements are specified, lists are posted only if they meet all requirements at the time of the request. These requirements include: *type mask, size limit,* and *frequency modulus*

Buffer use counts are maintained by the **dfm** calls in requesters and providers. They are used to track when the buffer is still in use or when it can safely be returned to the pool. The use count is incremented as a result of reserving, inserting and scheduling; it is decremented when a requester or provider has finished with the buffer. When the use count reaches zero, the buffer will be returned to the free buffer pool.

6 Semaphores

On Unix, **dfm** uses System V IPC semaphores for both mutual exclusion and synchronization. Message queues are implemented within **dfm.**

Mutual exclusion semaphores are used to insure the integrity of the information stored in the control structures, message queues, and buffer descriptors, all which are located in shared memory segments. Currently, the semaphore granularity is at the service group and buffer class level.

Synchronization semaphores are used to signal a blocked or waiting process that the status of the queue has changed. Each process using **dfm** in the system requires its own synchronization semaphore.

7 Memory

dfm uses two shared memory segments: control and data. At initialization these segments are reserved in shared memory or kernel memory, depending on the configuration of the user's system.

The control segment holds all data structures and message queues. All internal buffers, including buffer lists, are located in the data segment.

8 Message Queues

Queues and linked lists are used throughout **dfm**, in several different forms:

- Each service provider is associated with one or more message queues to which buffer lists are posted. A limited number of lists, and at most one provider, may be queued or blocked on each.
- Each service group has a message queue to which ready providers are posted, which is used for the next available distribution mode of scheduling buffer lists. A limited number of providers, and any number of requesters may be queued or blocked on each.
- Each internal buffer class has a corresponding free pool or queue and any number of requesters may be blocked on each.

9 Summary

The initial implementation of **dfm** was for VxWorks, a real-time multi-tasking kernel for embedded processors; this is currently in use for data acquisition by at least one Fermilab experiment. An implementation for Unix has been developed and is actively being integrated into several others.

References

[1] R. Pordes et al., "Fermilab's DART DA System," these proceedings.

[2] D. Berg et al., "DART Data Flow Manager Design," Fermilab Computing Division document DS-225.

[3] D. Berg et al., "DART Data Flow Requirements," Fermilab Computing Division document DS-226.

[4] Oleynik, Gene, "Integrating UNIX Workstations into Existing Online Data Acquisition Systems for Fermilab Experiments," Proceedings of Computing in High Energy Physics, 1991.