Dataflow Engine in DAQ Backbone DABC

J. Adamczewski-Musch, H. G. Essel, N. Kurz and S. Linev

Abstract—**New experiments at FAIR require new concepts of data acquisition systems. So rather than building hardware trigger configurations with strict latency limitations, self-triggered electronic systems will be utilized. Their front-end components will be time synchronized and will provide data furnished with time stamps. Data streams from a multitude of such components will be forwarded via a powerful sorting network to an event building farm. The Data Acquisition Backbone Core (DABC) is designed as a general purpose software framework for the implementation of such data acquisition systems. It is written in C++. Recently, we made its first version available. In this paper we describe the mechanisms of the data flow engine of DABC.**

Index Terms—**Data acquisition, software packages**

I. Introduction

DURING the design phase of data acquisition systems for the new experiments of the FAIR facility at GSI, a new concept evolved [1]. This concept comprises self-triggered front-end systems delivering time stamped data. The sorting of the data over the network cannot be based on events, since the latter are not defined at that point, but on data packets of time slices. An event definition is possible only after these time slice fragments are recomposed at the receiver nodes of the network. Afterwards the event data can be processed for filtering, compression, on-line analysis and storage. This processing will need large processing farms with high performance networks. The global structure of such a system is shown in Figure 1. To prove these concepts, we developed a DAQ framework for two main applications. The first one is a prototype of a high speed network event building system. The second application is a test bed for a full readout chain starting from the detectors, digitizers, readout controllers, data combiners up to the event building, filtering, and archiving.

The requirements for the test-bed use-case turned out to be the same as for a general purpose DAQ framework, since a variety of front-ends had to be integrated, some from legacy systems, others being not ready yet or even unknown. The Data Acquisition Backbone Core DABC is a C++ software library running on standard PCs. It organizes the data flow between various components of a distributed system in an efficient way using a data flow engine that combines data over fast networks. Application specific software that handles all kind of front-end systems can be plugged in. (The DABC does not cover the development of such front-ends). A general description of the DABC design and of some performance results has been published [2]. Bidirectional data rates of ~500 Mbytes/s per node have been measured on a Linux cluster with 100 nodes connected via InfiniBand [3].

Manuscript received May 20, 2009; revised January 28, 2010. This work was supported by the European Community-Research Infrastructure Activity under the FP6 "Structuring the European Research Area" program (Hadron physics, contract number RII3-CT-2004-506078).

J. Adamczewski-Musch, H. G. Essel , N. Kurz and S. Linev are with GSI Helmholtzzentrum für Schwerionenforschung, Planckstr 1, 64291 Darmstadt, Germany. The corresponding author is S. Linev (phone: +49-6159-711338, fax: +49-6159-712986, e-mail: S.Linev@gsi.de).

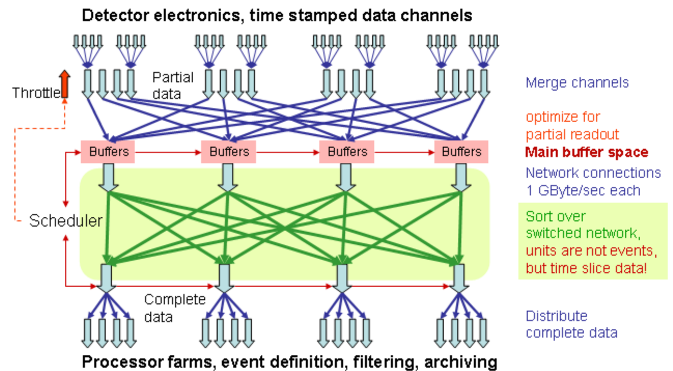


Fig. 1. Overall data acquisition scheme.

II. DABC Data Flow Engine

Figure 2 shows a topology example of a DABC based system. Front-end systems send data over PCI or Ethernet into PCs, where they are processed by application specific plug-ins. The networking (InfiniBand or Ethernet) is provided by the DABC.

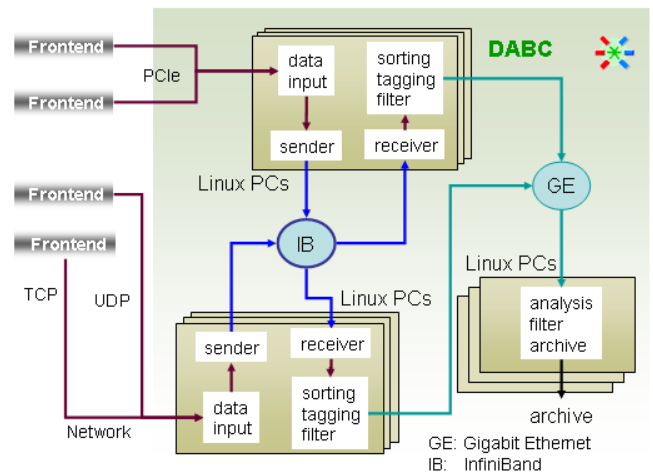


Fig. 2. Example of a DABC topology.

The building network can be used bi-directionally. Each node can serve both as a sender of partial data, and as a receiver of complete data (event or time slice). Inside DABC processes, software *modules* (*workers*) process the data of one or several data streams. Data streams propagate through *ports*, which are connected by *transports* and *devices* as shown in Figure 3. If necessary, application specific codes are implemented in the module, transport, and device classes (network transports and devices are already implemented in the DABC core). Workers run in threads. DABC applications may differ considerably regarding the need for CPU power and data bandwidth. Depending on the underlying hardware - like number of CPUs per node or like the kind of network - the optimum number of workers and threads, and the assignment of workers to threads may also vary.

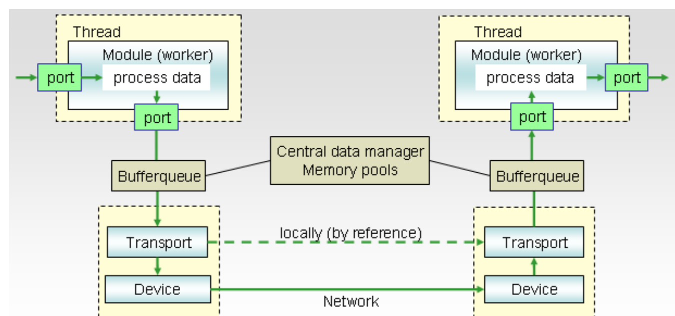


Fig. 3. Data flow engine of DABC.

In the following we describe in more detail the thread and memory management, and the data transport concept of DABC.

III. DABC Techniques

A. Processes or Threads

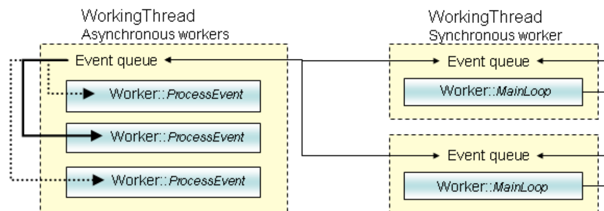
The current development of computer architectures leads to more and more CPU cores per node. To use these new architectures efficiently, the application code should run in parallel in multiple execution units like processes or threads. The big advantage of threads is that they share the address space and that they provide efficient synchronization mechanisms. It is more inconvenient to manage shared memory for processes and inter-process communication. Therefore, the DABC uses multiple threads, mutexes and condition variables. The correct locking of shared resources and the synchronization of jobs (when needed) are very important for any parallel executing application. One intrinsic problem of thread synchronization is the danger of deadlocks. Therefore, in the DABC, a sophisticated multi-threaded environment is implemented, which takes special care to avoid deadlocks per design. The access to the shared resources is protected with mutexes. The thread synchronization is implemented by using thread condition waits.

B. DABC Multi-threading

The basic concept of the DABC thread organization is sketched in Figure 4 (left). Each thread runs a main loop driven by *event* queues accessible to all other threads of the same process. In the following, *event* means a message triggering some action. Incoming events are queued either by the executing thread or by other threads. The main loop scans the queues and executes the entries according to priority levels by calling *ProcessEvent* functions implemented in application specific subclasses of the *worker* class. These functions do the calculations, file I/O, or networking by reacting to incoming events. When the queues are empty, the thread goes into a waiting state and does not consume any CPU resources until an event arrives. In the meantime, other threads can execute.

Workers sharing their thread with others must never block. Each event processing function must therefore return control to the main loop after a limited time. The communication between workers is also performed via events. Each worker uses thread-safe functions for event submission and processing, respectively.

To perform periodic actions in the worker without having incoming events, one may configure the timeouts, which are processed in the thread main loop and delivered in form of timeout events to the workers.



ProcessEvent	MainLoop
Several per thread (assigned by setup)	One per thread
Sequential execution	Parallel execution
Non-blocking resource requests (faster)	Blocking resource request (slower)
For low CPU requirements	For high CPU requirements (multicore!)
For high data flow (no context switch)	For low data flow (context switch)

Fig. 4. Thread organization in DABC.

Using event queues, one needs an instance that permanently delivers events to keep the workers running. Typically, the data sources deliver such events. Since the data streams are always organized in buffers, events are generated when buffers are ready to process. All functions are therefore called per buffer and execute only for a limited time.

An example of a data source is the IP socket worker class of DABC assigned to a single opened socket. Several sockets can be handled in one thread generating events when a socket can read or write, or when it is closed. The events are than delivered to the associated workers, where the callback function handles the event (processing data) and returns. Then the next event can be processed. A similar approach has been used in the implementation of thread and worker classes for InfiniBand networks.

The DABC thread system allows us to configure the assignment of workers to threads without modification. The DAQ topology can be optimized easily for given hardware setups.

C. Modules

Application specific data processing code is executed in *module* objects, which are in fact *workers* (see Figures 3 and 4). The *Module* class of DABC provides several components: input/output ports for receiving/sending data, pool handles for requesting memory, timers for generating timeout events, and parameters for the configuration and monitoring. A module can execute in both asynchronous or synchronous modes. Therefore, two alternative ways to develop data processing code are supported. The main differences are shown in Figure 4.

In order to run in an asynchronous mode (left) a module implements the event-driven processing model as described previously. In this case, the data processing code is implemented in callback functions for events such as input ready, output ready, or timeout. Several asynchronous modules executing in the same thread are in fact synchronized by events. Thus, the communication between asynchronous modules running in the same thread does not need time consuming locking and thread context switches.

An alternative way is the synchronous mode. It requires an explicit main loop, where several blocking functions can be used (s. Figure 4 right). For instance, one can block the execution of the main loop until a new portion of data arrives to the module or until a free buffer of required size is provided by the memory pool. Such blocking functions allow us to write linear code without complicated bookkeeping as it is typically required for asynchronous modules. The blocking calls in fact access the event queue. If no event for the required resource is there, the thread waits for events. It is evident that only one synchronous module can run in each thread.

It is up to the application designer to decide which mode fits best. The synchronous mode needs more threads and, therefore, more synchronization overhead while the coding is simpler. The asynchronous mode based on event processing is the only way to implement complex systems with multiple input and output ports and with irregular traffic patterns. It also provides better performance. However, the design of the callback functions is more complex.

D. Memory Management

In a high performance distributed DAQ system, one should avoid copying data as far as possible. If data have to be distributed and combined over a network, it is necessary to access and transport fragments of buffers rather than the buffers themselves. To avoid copies, one needs a gathering mechanism for the transport. This is supported by another important component of the DABC, by the memory management.

The memory space for data handling in DABC is preallocated in memory pools. Memory pools consist of contiguous blocks of configurable size. The number of blocks can be either fixed or flexibly changed on demand. On a request for free memory from any DABC component, the memory pool delivers a reference to a free memory block of the required size. Multiple references to memory fragments mapped over filled memory blocks or even overlapping fragments can be obtained for processing. The memory pool manages reference counters for each underlying memory block (see Figure 5). Each DABC component that works with memory references is responsible for releasing them. Only when all references to a memory block are released (reference counter is zero), the memory block can be used again.

To transfer data between DABC threads, only the references are exchanged. Reference objects can combine several fragments of memory together into so-called *reference lists* shown in Figure 5. This allows us to extract data subsets without copying memory. Such reference lists can be directly transferred over the network (via DMA in case of InfiniBand) or stored into files.

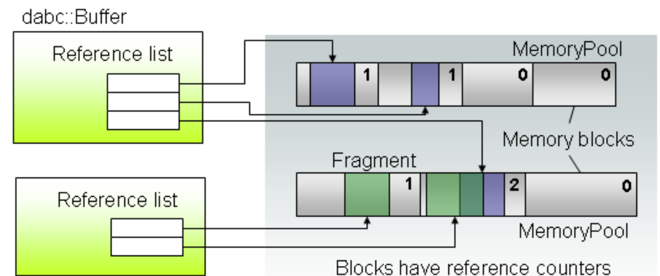


Fig. 5. Memory pool and referencing its memory by buffer objects. A buffer can have a list of references (gather list) on several memory blocks or fragments of blocks; multiple references on overlapping fragments are possible.

For any kind of DMA operations, memory pinning must be performed first. For instance, the InfiniBand library requires to register the memory prior to using it for sending or receiving data. Such registration can take a significant amount of time and should not be, therefore, performed prior to every data transfer. With the DABC memory pool, one can register memory once, before the first transfer operation starts.

E. Ports, Transports and Devices

Buffers enter and leave a module through *ports*. A module may have several input, output, or bidirectional ports. There are several module functions to process port events: when data enters an input port, when a module can send new data over an output port, or when a port is connected or disconnected.

Each port has a buffer queue with a configurable size which steers the back-pressure mechanism of the DABC. If an input port queue is full, the corresponding transport cannot add new buffers to the queue and, thus, will not read new data from its data source. The same is true for an output port queue; if it is full, no new output ready event is available for the sending module. In the synchronous mode, a call to send data over the port will just block until the connected receiver empties the port queue or until a timeout event arrives.

The ports are connected to *transport* objects, which are responsible either for getting data from their data source or for delivering data to a data sink, respectively. There is a special local transport subclass, which transfers data between two ports in the same process by reference. The connection of ports on different nodes is done by network transports (see

Figure 3). Presently, transports for Ethernet (TCP, UDP) and InfiniBand are implemented.

Device objects typically correspond to some physical entities like PCI cards. They manage the transports, which are interacting with these entities. A *device* is a *worker* and controls the thread for the transport to run. By configuration, transports may run in the same thread as modules, or run in their own threads without affecting directly the data processing.

IV. DABC RELEASE

DABC has been released under a GPL license [4] and can be downloaded from http://dabc.gsi.de. The structure of the released package is shown in Figure 6.

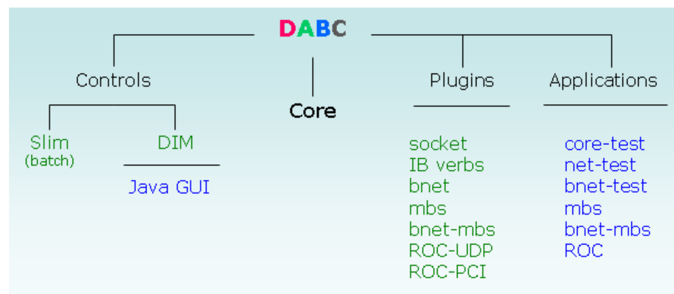


Fig. 6. Components of DABC distribution.

The core of the DABC package includes the library with the complete data-flow kernel, the DABC executable, and a shell script to start DABC applications on remote nodes. The package includes two implementations of a control interface: one is a simple batch-like control used for testing purposes (*Slim*). The other one is based on DIM [5] and provides a GUI. For setup a powerful and flexible XML-based configuration system is included.

The following functional components are provided as plugins:

- *socket* – thread, transport, and device implementation to work with IP sockets in asynchronous mode;
- *IB verbs* – thread, transport and device implementation for data transport over InfiniBand [6];
- *mbs* – transport implementation for connections to the standard GSI data acquisition system MBS [7] (http://daq.gsi.de) and modules for event building;
- *bnet* – components (applications and modules) for constructing multi-node event building networks. Applications can use these classes to construct event building networks for their own data formats.
- *bnet-mbs* – implementation of multi-node event building for MBS front-ends;
- *ROC-UDP* – device and transport classes to access a specific readout controller (ROC) via Ethernet;
- *ROC-PCI* – device and transport classes for DMA transfer from/to a PCIe board, which receives data from readout controllers via optical channels.

For each plug-in, the package provides an application directory with all files needed to run this application. In most cases, these are just configuration files. Sometimes test modules are implemented to demonstrate a specific functionality.

The plug-ins and applications provided are intended to demonstrate the functionality of DABC. They can be used as examples for users who want to develop their own classes in order to integrate their specific hardware or software into the DABC. On the DABC web site, a detailed introduction and a programmer manual are available.

V. CONCLUSION

The DABC framework can be used to develop data acquisition systems of different sizes: from small detector and readout test-beds up to high performance event building over fast networks. In addition to the conventional TCP/IP sockets, the fast and low-latency InfiniBand network is supported by DABC as well. It is possible to implement other networks.

A multitude of experiment-specific front-end systems and data processing modules can be integrated into the DABC by plug-in mechanisms.

Recently, the DABC has been used in the first detector and front-end equipment test experiments.

REFERENCES

[1] H.G.Essel, FutureDAQ for CBM: On-line event selection, *IEEE TNS* Vol.53, No.3, June 2006, pp 677-681

[2] J. Adamczewski, H. G. Essel, N. Kurz and S. Linev, Data Acquisition Backbone Core DABC, *IEEE TNS* Vol.55, No.1, February 2008, pp 251-255

[3] Data Acquisition Backbone Core DABC J. Adamczewski, H. G. Essel, N. Kurz and S. Linev 2008 *J. Phys.: Conf. Ser*. 119 022002 (8pp)

[4] http://www.gnu.org/licenses/gpl.html

[5] C. Gaspar, Distribution Information Management system DIM, - http://dim.web.cern.ch/dim

[6] OpenFabric Alliance OFED, - http://www.openfabrics.org

[7] H.G. Essel, J. Hoffmann, N. Kurz and W.Ott, The general purpose data acquisition system MBS, *IEEE TNS* Vol.47, No.2, April 2000, pp 337-339