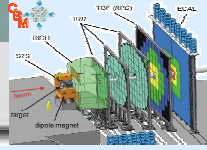
FAIR Facility for Antiproton and Ion Research



http://dabc.gsi.de

Integration of CBM readout controller into DABC framework

N. Abel[2], J. Adamczewski-Musch[1], H.G. Essel[1], U. Kebschull[2], S. Linev[1], W.F.J. Müller[1], S. Müller-Klieser[2]

[1]GSI Helmholtzzentrum für Schwerionenforschung GmbH, Darmstadt

[2]Kirchhoff Institut für Physik, Heidelberg

Motivation for developing DABC

Use cases

- Detector tests
- FE equipment tests
- High speed data transport
- Time distribution
- Switched event building
- Software evaluation
- MBS* event builder

Requirements

- build events over fast networks
- handle triggered or self-triggered front-ends
- process time stamped data streams
- provide data flow control (to front-ends)
- connect (nearly) any front-ends
- provide interfaces to plug in application codes
- connect MBS readout or collector nodes
- be controllable by several controls frameworks

DABC v.1.0 features

- Compact multi-threaded data-flow core
- Several device/application specific add-ons (plugins)
- TCP/IP (sockets) and InfiniBand (OFED verbs) as data transport
- Plugins (shared libraries) for user-specific components
- Components for constructing event-building network
- Flexible configurations with XML files
- DIM as interface for control system, other implementations are foreseen
- Generic Java-based control GUI

* Muliti Branch System: current standard DAQ at GSI

DABC core functionality

Main feature of **thread** classes in DABC:
- handling event producers
- dispatching events to processors
- timeout and command handling

Main thread classes:

dabc::WorkingThread	based on POSIX pthread library
dabc::SocketThread	event-based socket handling via select
verbs::Thread	OFED verbs as event producer
...	any other device-specific class

Processors are active elements of DABC. There are:

dabc::Module	place for user-specific code
dabc::Device	device-specific configuration
dabc::Transport	data sending/receiving
dabc::MemoryPool	*memory* management
dabc::Application	user-specific state changing
dabc::Manager	manager of everything

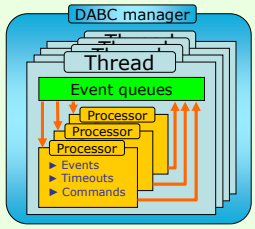
Commands usage provides a way to perform application or user specific code from any part of the system.

Commands are always executed in the context of the receiver thread allowing to avoid unnecessary mutex locking.

A command can be executed either synchronously or asynchronously to calling thread.

A command can be submitted to any object in the system, even to a remote node, via specification of the full receiver name.

A command object can contain an arbitrary number of arguments and can return several result values.



Synchronized parallel processing

Value of any **parameter** object can be setup through configuration xml file or via control interface

Configuration system of DABC allows to set application-wide default values for frequently-used parameters

Example for 4-nodes network test is just:

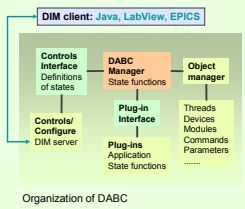
```
<?xml version="1.0"?>
<dabc version="1">
 <Context host="node01"/>
 <Context host="node02"/>
 <Context host="node03"/>
 <Context host="node04"/>
 <Default>
  <Context name="*">
   <Run>
    <Lib value="libDabcNetTest.so"/>
    <runfunc value="RunAllToAll"/>
   </Run>
  </Context>
 </Default>
</dabc>
```

dabc::MemoryPool class is used to preallocate memory blocks and share them between different components of the system. Main concept of DABC design was to provide a **zero-copy** approach, where all data transfers are done via references and memory pool takes care about correct release of memory regions via reference counters.

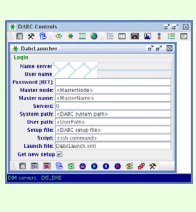
dabc::Manager class is the central class in DABC and organizes all other objects like threads, modules, factories in hierarchical structure, where all objects can be accessed by their names.

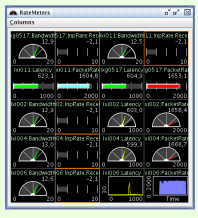
Plug-in mechanism of DABC allows to integrate different application or user-specific classes into common DAQ framework. Through different factory methods of each plugin specific thread, device, transport, application or module classes can be created.

DABC controls (Java & DIM)

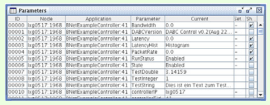


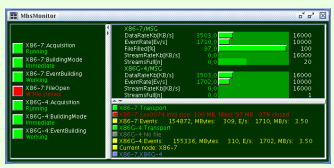
Organization of DABC



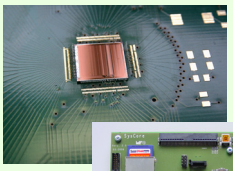


The current Java GUI is built up dynamically on the information delivered by the DIM servers running in the application threads on all nodes



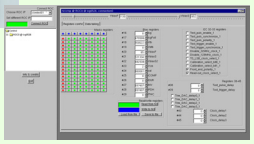


CBM readout controller



The n-XYTER is a 128 channel asynchronous, self-triggered, self-sparcifying readout ASIC developed as a front-end for neutron scattering detector applications. In FAIR experiments like CBM it is used as prototype and test-bed of front-end electronics for several kinds of detectors prototypes.

The CBM readout controller (ROC), developed in *Kirchhoff Institut für Physik, Heidelberg*, is an FPGA-based board, designed to read time-stamped data from n-XYTER and transfer that data via optical link to PC. As alternative option, Ethernet can be used for data transfer.



ROClib is a C++ library, which implements a protocol for control and data transfer from ROC to PC. Because the data transport is based on UDP/IP, the protocol cares about data retransmission in case of packets lost. There is a ROOT-based gui, used for configuration and testing of front-end boards, equipped with n-XYTER chips.

Integration with DABC, test beam Sep.08

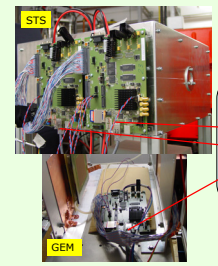
Specific *roc::Transport* class was implemented to integrate ROClib functionality into DABC. It mainly implements several methods of *dabc::DataTransport* class:

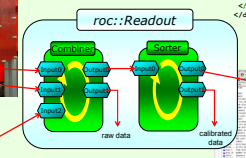
Read_Size defines size of next buffer
Read_Start request data from ROClib
Read_Complete completes reading

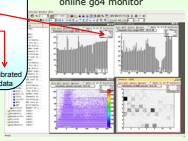
In addition, *roc::Device* class provides configuration and command execution capabilities.

Two module classes were implemented to combine and sort data from several ROCs. As output, MBS events format was used. Application class *roc::Readout* instantiates and configures all necessary components to perform readout from several ROCs. Example of configuration for readout from three ROCs is shown at the right side.

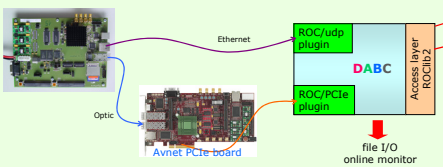
```
<?xml version="1.0"?>
<dabc version="1">
 <Context name="Readout">
  <Run>
   <Lib value="libDabcMbs.so"/>
   <Lib value="libDabcRnut.so"/>
   <Logfile value="Readout.log"/>
  </Run>
  <Application class="roc::Readout">
   <DoCalibr value="0"/>
   <NumRocs value="3"/>
   <RocIp0 value="cbmtest01"/>
   <RocIp1 value="cbmtest02"/>
   <RocIp2 value="cbmtest04"/>
   <BufferSize value="65536"/>
   <NumBuffers value="100"/>
   <TransportWindow value="30"/>
   <RawFile value="run090.lmd"/>
   <MbsServerKind value="Stream"/>
   <CalibrFile value=""/>
   <MbsFileSizeLimit value="110"/>
  </Application>
 </Context>
</dabc>
```





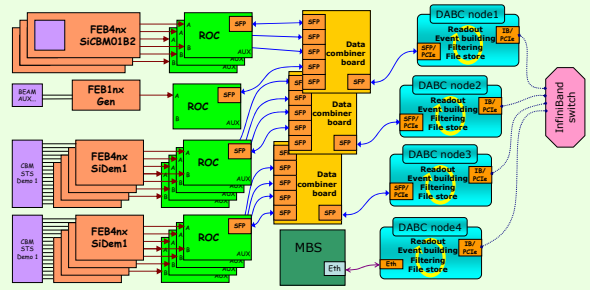


DABC as access layer for ROC



DABC will be used as access layer for ROCs, connected either via Ethernet or via fiber and PCIe board. This demonstrates the possibility to use DABC just as library.

Planed beam tests with ROC & DABC



Work supported by EU RP6 project JRA1 FutureDAQ RII3-CT-2004-506078