1 Revisions

Titel: DABC: Programmer Manual

Document	Date	Editor	Revision	Comment
DABC-prog	2009-01-04	Hans G.Essel	1.0.0	First scetch

Contents

2 *DABC* Programmer Manual: Overview

[programmer/prog-overview.tex]

2.1 Role and functionality of the objects

2.1.1 Modules

All processing code runs in module objects. There are two general types of modules: The ***dabc::ModuleSync*** and the ***dabc::ModuleAsync***.

2.1.1.1 Class *dabc::ModuleSync*

Each module is executed by a dedicated working thread. The thread executes a method *MainLoop()* with arbitrary code, which *may block* the thread. In blocking calls of the framework (resource or port wait), optionally command callbacks may be executed implicitely ("non strictly blocking mode"). In the "strictly blocking mode", the blocking calls do nothing but wait. A *timeout* may be set for all blocking calls; this can optionally throw an exception when the time is up. On timeout with exception, either the *MainLoop()* is left and the exception is then handled in the framework thread; or the *MainLoop()* itself catches and handles the exception. On state machine commands (e.g. Halt or Suspend), the blocking calls are also left by exception, thus putting the mainloop thread into a stopped state.

2.1.1.2 Class *dabc::ModuleAsync*

Several modules may be run by a *shared working thread*. The thread processes an *event queue* and executes appropriate *callback functions* of the module that is the receiver of the event. Events are fired for data input or output, command execution, and if a requested resource (e.g. memory buffer) is available. **The callback functions must never block the working thread**. Instead, the callback must **return** if further processing requires to wait for a requested resource. Thus each callback function must check the available resources explicitly whenever it is entered.

2.1.2 Commands

A module may register commands in the constructor and may define command actions by overwriting a virtual command callback method in a subclass implementation of ***dabc::Command***.

2.1.3 Parameters

A module may register ***dabc::Parameter*** objects. Parameters are accessible by name; their values can be monitored and optionally changed by the controls system.

2.1.4 Manager

The modules are organized and controlled by one manager.

- The manager is an object manager that owns and keeps all registered basic objects into a folder structure. Each object has a direct backpointer to the manager.

- The manager has an independent manager thread. This thread may execute manager commands for configuration, and all high priority commands like state machine switching that must be executed immediately even if the mainloop of the modules is blocked.
- The manager receives and dispatches external module commands to the destination module where they are queued and eventually executed by the module thread (see above).
- The manager defines interface methods to the control system. This covers registering, sending, and receiving of commands; registering, updating, unregistering of parameters; error logging and global error handling. These methods must be implemented in subclass of ***dabc::Manager*** that knows the specific controls framework.
- The manager is a singleton instance per executable. It is persistent independent of the application's state.

2.1.5 Memory and buffers

Data in memory is referred by ***dabc::Buffer*** objects. Allocated memory areas are kept in ***dabc::MemoryPool*** objects.

- A buffer contains a list of references to scattered memory fragments, or it may have an empty list of references. Auxiliary class ***dabc::Pointer*** offers methods to transparently treat the scattered fragments from the user point of view (concept of "virtual contiguous buffer"). Moreover, the user may also get direct access to each of the fragments.
- The buffers are provided by one or several memory pools which preallocate reasonable memory from the operating system. A memory pool may keep several sets, each set for a different configurable memory size. Each memory pool is identified by a unique ***dabc::PoolHandle*** object which is used e.g. by the module to communicate with the memory pool.
- A new buffer may be requested from a memory pool by size. Depending on the module type and mode, this request may block until an appropriate buffer is available; or it may return an error value if it can not be fulfilled, resp. The delivered buffer has at least the requested size, but may be larger. A buffer as delivered by the memory pool is contiguos.
- Several buffers may refer to the same fragment of memory. Therefore, the memory as owned by the memory pool has a reference counter which is incremented for each buffer that refers to any of the contained fragments. When a user frees a buffer object, the reference counters of the referred memory blocks are decremented. If a reference counter becomes zero, the memory is marked as "free" in the memory pool.

2.1.6 Ports

Buffers are entering and leaving a module through ***dabc::Port*** objects.

- A module may have several input ports, output ports, or bidirectional ports. The ports are owned by the module.
- Each port uses a buffer queue of configurable length.
- The ***dabc::ModuleSync*** class *MainLoop()* function may wait for buffers to arrive at one or several input ports; this call may block. Processed buffers can be put into output ports; these calls may also block the mainloop (data backpressure mechanism).
- The ***dabc::ModuleAsync*** class has event callbacks that are executed by the framework when a buffer arrives at an input port (*processInput()*), or when an output port is ready to accept a buffer (*processOutput()*).

2.1.7 Transport

Outside the modules the ports are connected to ***dabc::Transport*** objects.

- A transport may transfer buffers between the ports of different modules (local data transport).
- A transport may connect the port to a data source (file input, network receiver, hardware readout), or to a data sink (file output, network sender).
- Two transport instances of the same kind on different nodes may connect ports of modules on different nodes (e.g. InfiniBand verbs transport).

2.1.8 Device

A transport belongs to a ***dabc::Device*** object of a corresponding type that manages it.

- A device may have one or several transports.
- A ***dabc::Device*** instance usually corresponds to an IO component (e.g. network card).
- There may be more than one ***dabc::Device*** instances of the same type in an application scope.
- The threads that run the transport functionality are created by the device. If the ***dabc::Transport*** implementation shall be able to block (e.g. on socket receive), there can be only one transport for this thread.
- A device may register parameters and define commands. This is the same functionality as available for modules.
- A device is persistent independent of the connection state of the transport. In contrast, a transport is created during *connect()* or *open()*, respectively, and deleted during *disconnect()* or *close()*, respectively.
- The device objects are owned by the global manager singleton. Transport objects are owned by the device.

2.2 User interface

2.2.1 Module

It is supposed that the data processing functionality usually is implemented by subclassing the ***dabc::ModuleSync*** base class.

- The constructor of ***dabc::ModuleSync*** subclass creates all ports, may initialize the pool handles, and may define commands and parameters.
- The virtual *ExecuteCommand()* method of ***dabc::ModuleSync*** subclass may implement the callbacks of the defined commands.
- The virtual *MainLoop()* method of ***dabc::ModuleSync*** subclass implements the processing job.
- The user code shall not directly access the memory pools to request new buffers. Instead, it can use methods of ***dabc::Module*** with a ***dabc::PoolHandle*** object as argument. These methods may block the *MainLoop()*.
- The user code can send and receive buffers from and to ports by methods of ***dabc::ModuleSync***. These methods may block the *MainLoop()*.

2.2.2 Special modules

For special set ups (e.g. Bnet), the framework provides ***dabc::Module*** subclasses with generic functionality (e.g. ***bnet::BuilderModule***, ***bnet::FilterModule***). In this case, the user specific parts like data formats are implemented by subclassing these special module classes.

- Instead of implementing *MainLoop()*, other virtual methods (e.g. *DoBuildEvent()*, *TestBuffer()*) may be implemented that are implicitly called by the superclass *MainLoop()*.
- The special base classes may provide additional methods to be used for data processing.

2.2.3 Device and transport

All data transport functionality is implemented by subclassing ***dabc::Device*** and ***dabc::Transport*** base classes.

- The ***dabc::Device*** subclass constructor may create pool handles and may define commands and parameters.
- The virtual *ExecuteCommand()* method of ***dabc::Device*** subclass may implement the callbacks of the defined commands.
- Factory method *CreateTransport()* of ***dabc::Device*** subclass defines which transport instance is created by the framework whenever this device shall be connected to the port of a module.
- The virtual *Send()* and *Recv()* methods of the ***dabc::Transport*** subclass must implement the actual transport from and to a connected port, respectively.
- To implement a simple user defined transport (e.g. read from a special socket protocol), there is another base class ***dabc::DataTransport*** (subclass of ***dabc::Transport***). This class already provides queues for (optionally) input and output buffers and a data backpressure mechanism. Instead of *Send* and *Recv()*, the user subclass must implement special virtual methods, e.g. *ReadBegin()*, *ReadComplete(buffer)* to request the next buffer of a given size from the base class, and to peform filling this buffer from the associated user device, respectively. These methods are called implicitely from the framework at the right time.

2.2.4 Factories

The set up of the application specific module and device objects is done by ***dabc::Factory*** subclasses.

- All factories are registered and kept in the global manager. The access to the factories' functionality is done via methods of the manager.
- All factories are instantiated as static singletons when loading the libraries that defines them. The user need not to create them explicitely.
- The ***dabc::Factory*** subclasses must implement methods *CreateModule(classname)*, and *CreateDevice(classname)* , to instantiate modules or devices, respectively. The user must subclass the ***dabc::Factory*** to add own classes to the system. **Note:** the factory method for transport objects is not in this factory, but in the corresponding ***dabc::Device*** subclass.
- The manager can keep more than one factory and scans all factories to produce the requested object class.
- The framework provides several factories for predefined implementations (e.g. ***bnet::SenderModule***, ***dabc::VerbsDevice***).

2.2.5 Application plug-in

The specific application controlling code is defined in the ***dabc::ApplicationPlugin***.

- The manager has exactly one application plug-in. The user must provide a ***dabc::ApplicationPlugin*** subclass.
- The ***dabc::ApplicationPlugin*** is instantiated and registered on startup time by a global *InitPlugins()* function. This function is declared and executed in the framework, but defined in the user library and thus knows the user specific plug-in classes. This trick is necessary to decouple the executable main function (e.g. χ*DAQ*executive) from the user specific part.
- The user may implement virtual methods *UserConfigure()* , *UserEnable()*, *UserBeforeStart()*, *UserAfterStop()*, *UserHalt()* in his/her ***dabc::ApplicationPlugin*** subclass. These methods are executed by the framework state machine before or after the corresponding state transitions to do additional application specific configuration, run control, and clean-up actions. Note: all generic state machine actions (e.g. cleanup of modules and devices, starting and stopping the working processors) are already handled by the framework at the right time and need not to be invoked explicitely here.
- The application plug-in may register parameters that define the application's configuration. These parameters can be set at runtime from the configuration and controls system.
- The methods of the application plug-in should use application factories to create modules and devices by name string. However, for convenience the user may implement factory methods *CreateModule()* and *CreateDevice()* straight in his/her ***dabc::ApplicationPlugin*** subclass.
- For special DAQ topologies (e.g. Bnet), the framework offers implementations of the ***dabc::ApplicationPlugin*** containing the generic functionality (e.g. ***bnet::WorkerPlugin***, ***bnet::ClusterPlugin***). In this case, the user specific parts are implemented by subclassing these and implementing additional virtual methods (e.g. *CreateReadout()*).

2.3 Controls and configuration

2.3.1 Finite state machine

The running state of the DAQ system is ruled by a *finite state machine* on each node of the cluster.

- The manager provides an interface to switch the application state by the external control system. This may be done by calling state change methods of the manager, or by submitting state change commands to the manager.
- The finite state machine itself is not part of the manager, i.e. the manager does not necessarily check if a state transition is allowed. Instead, the controls framework provides a state machine implementation.
- Some of the application states may be propagated to the active components (modules, device objects), e.g. the Running or Ready state which correspond to the activity of the thread. Other states like Halted or Failure do not match a component state; e.g. in Halted state, all modules are deleted and thus do not have an internal state. The granularity of the control system state machine is not finer than the node application.
- There are 5 generic states to treat all set-ups:

  Halted : The application is not configured and not running. There are no modules, transports, and devices existing.

  Configured : The application is mostly configured, but not running. modules and devices are created. Local port connections are done. Remote transport connections may be not all fully connected, since some connections require active negotiations between existing modules. Thus, the final connecting is done between Configured and Ready.

  Ready : The application is fully configured, but not running (threads are stopped).

Running : The application is fully configured and running.

Failure : This state is reached when there is an error in a state transition function. Note that a run error during the Running state would not lead to Failure, but rather to stop the run in a usual way (to Ready).

- The state transitions between the 5 generic states correspond to commands of the control system for each node application:

  Configure : between Halted and Configured. Core framework automatically creates standard modules and devices. The application plug-in creates application specific objects, requests memory pools, and establishes all local port connections. Core framework instantiates the requested memory pools

  Enable : between Configured and Ready. The application plug-in may establish the necessary connections between remote ports. The framework checks if all required connections are ready.

  Start : between Ready and Running. The framework automatically starts all modules, transport and device actions.

  Stop : between Running and Ready. The framework automaticall stops all modules, transport and device actions, i.e. the code is suspended to wait at the next appropriate waiting point (e.g. begin of *MainLoop()*, wait for a requested resource). Note: queued buffers are not flushed or discarded on Stop !

  Halt : switches states Ready , Running , Configured, or Failure to Halted. The framework automatically deletes all registered objects (transport, device, module) in the correct order. However, the user may explicitly specify on creation time that an object shall be persistent (e.g. a device may be kept until the end of the process once it had been created)

2.3.2 Commands

The control system may send (user defined) commands to each component (module , device). Execution of these commands is independent of the state machine transitions.

2.3.3 Parameters and configuration

The *Configuration* is done using parameter objects. The manager provides an interface to register parameters to the configuration/control system.

- On application startup time, the external configuration system may set the parameters from a database, or from a configuration file (e.g. XML configuration files).
- During the application lifetime, the control system may change values of the parameters by command. However, since the set up is changed on Configure time only, it may be forbidden to change true configuration parameters except when the application is Halted. Otherwise, there would be the possibility of a mismatch between the monitored parameter values and the really running set up.
- The current parameters may be stored back to the data base of the configuration system.

2.3.4 Parameters and monitoring

The control system may use local parameter objects for *Monitoring* the components. When monitoring parameters change, the control system is updated by interface methods of the manager and may refresh the GUI representation.

2.3.5 Parameter change

The control system may change local parameter objects by command in any state to modify minor system properties independent of the configuration set up (e.g. switching on debug output, change details of processing parameters).

2.4 Package and library organisation

The complete system consists of different packages. Each package may be represented by a subproject of the source code with own namespace. There may be one or more shared libraries for each package. Main packages are as follows:

2.4.1 Core system

Defines all base classes and interfaces, like

- ***dabc::Module***
- ***dabc::Port***
- ***dabc::Buffer***
- ***dabc::Device***
- ***dabc::Transport***
- ***dabc::Command***
- ***dabc::Parameter***
- ***dabc::Manager***

and basic functionality for object organization, memory management, thread control, event communication.

2.4.2 Control system

Depends on the **Core system**. Defines functionality of state machine, command transport, parameter monitoring and modification. Interface to the **Core system** is implemented by subclass of ***dabc::Manager***.

2.4.3 Configuration System

Depends on the **Core system**; it may use functionalities of the control system. Implements the connection of configuration parameters with a database (i.e. a file in trivial case). Interface to the **Core system** is defined by subclass of ***dabc::Manager***.

2.4.4 Transport packages

One or more special transport packages: Depends on the **Core system**. It may depend on an external framework. Implements ***dabc::Device*** and ***dabc::Transport*** classes for specific data transfer mechanism, e.g. **verbs** or **tcp/ip socket**. Each transport package provides a factory to create a specific device by class name. However, the most common transport implementations may be put directly to the **Core system**, e.g. local memory transport, or socket; the corresponding factory is part of the **Core system** then.

2.4.5 Bnet package

Depends on the **Core system**. Implements special modules to cover a generic event builder network. Defines interfaces (virtual methods) of the special Bnet modules to implement user specific code in subclasses. The Bnet package provides a factory to create specific Bnet modules by class name. It also provides application plug-ins to define generic functionalities for worker nodes (***bnet::WorkerPlugin***) and controller nodes (***bnet::ControllerPlugin***).

2.4.6 Application packages

Depend on the **Core system**; may depend on several ransport packages; may depend on the Bnet package; may depend on other application packages. Implement experiment specific modules that actually run the DAQ. May implement ***dabc::Device*** and ***dabc::Transport*** classes for special data input or output. May provide a factory to create a specific module or device by class name. Provides implementation of ***dabc::ApplicationPlugin*** that defines the application set-up; this may be a subclass of specific existing application plug-ins (e.g. subclass of ***bnet::WorkerPlugin***).

The DABC distribution will provide the

- **Core**
- **Control**
- **Configuration**
- **Bnet**
- **Transport packages** for **verbs** and **socket**

As examples, there are several application packages, with configuration data for different cases:

- Simple *MBS* event builder (one node, one or more *MBS* input channels)
- *MBS* n $\times$ m event building (using the Bnet package)
- Simple *Active Buffer Board* readout (this example also contains a separate ransport package implementing the ***dabc::Device*** class for *Active Buffer Board*)
- A Bnet set up for *Active Buffer Board* readout (with *Active Buffer Board* transport package)

2.5 Implementation status

2.5.1 Packages

2.5.1.1 Core system

Plain C++ code; independent of any external framework.

2.5.1.2 Control system

- χ*DAQ* provides the application environment and the state machine. There may be a second non-χ*DAQ* state machine environment in the future.
- DIM is used as main transport layer for commands and parameter monitoring. On top of DIM, a generic record format for parameters is defined.
- Each registered command exports a self describing command descriptor parameter as DIM service.
- A generic controls GUI using the record and command descriptors is implemented with JAVA.

2.5.1.3 Configuration system

Configuration parameters are set from setup files (XML) and available as DIM services to the control system.

2.5.1.4 Transport packages

- Network transport for tcp/ip socket and InfiniBand verbs.
- *Active Buffer Board* **(ABB) readout**
- *Read Out Controller Board* **readout using Knut library**

2.5.1.5 Bnet package

Ready

2.5.1.6 Application packages

- Simple *MBS* event building
- Bnet with switched *MBS* event building
- Bnet with pseudo event generators and optional *Active Buffer Board* readout

2.5.2 Classes

2.5.2.1 Core system

The most important classes of the dabc core system

dabc::Basic : The base class for all objects to be kept in *DABC* collection(e.g. ***dabc::Folder***).

dabc::Command : Represents a command object. A command is identified by its name which it keeps as text string. Additionally, a command object may contain any number of arguments (integer, double, text). These can be set and requested at the command by their names. The available arguments of a special command may be exported to the control system as ***dabc::CommandDefinition*** objects. A command is sent from a ***dabc::CommandClient*** object to a ***dabc::CommandReceiver*** object that executes it in his scope. The result of the command execution may be returned as a reply event to the command client. The manager is the standard command client that distributes the commands to the command receivers (i.e. module , manager, or device).

dabc::Parameter : Parameter object that may be monitored or changed from control system. Any ***dabc::WorkingProcessor*** implementation may register its own parameters. The use case of a parameter depends on the lifetime of its parent object: The configuration parameters should be created from the application plug-in which is persistent during the process lifetime. Transient monitoring parameters may be created from a device (optionally also persistent, but not yet existing at process startup when configuration database is read!), from a module, or from a transport implementation (if this also inherits from ***dabc::WorkingProcessor***). The latter parameters do only exist if the application is at least in Configured state, and they will be destroyed with their parents when switching to Halted state. Currently supported parameter types are:

- ***dabc::IntParameter*** - simple integer value
- ***dabc::DoubleParameter*** - simple double value
- ***dabc::StrParameter*** - simple test string

- ***dabc::StateParameter*** - contains state record, e.g. current state of the finite stat machine and associated colour for gui representation
- ***dabc::InfoParameter*** - contains info record, e.g. system message and associated properties for gui representation
- ***dabc::RateParameter*** - contains data rate record and associated properties for gui representation. May be updated in predefined time intervals.
- ***dabc::HistogramParameter*** - contains histogram record and associated properties for gui representation.

dabc::WorkingThread : An object of this class represents a system thread. The working thread may execute one or several jobs; each job is defined by an instance of ***dabc::WorkingProcessor***. The working thread waits on an event queue (by means of pthread condition) until an event for any associated working processor is received; then the corresponding event action is executed by calling *ProcessEvent()* of the corresponding working processor.

dabc::WorkingProcessor : Represents a runnable job. Each working processor is assigned to one working thread instance; this thread may run either one working processor, or serve several working processors in parallel. ***dabc::WorkingProcessor*** is a subclass of ***dabc::CommandReceiver***, i.e. a working processor may receive and execute commands in his scope.

dabc::Module : A processing unit for one "step" of the dataflow. Is subclass of ***dabc::WorkingProcessor***, i.e. the module may be run by an own dedicated thread, or a working thread may execute several modules that are assigned to it. A module has ports as connectors for the incoming and outgoing data flow.

dabc::ModuleSync : Is subclass of ***dabc::Module***; defines interface for a synchronous module that is allowed to block. User must implement virtual method *MainLoop()* that runs in a dedicated working thread. Method *TakeBuffer()* provides blocking access to a memory pool. The optionally blocking methods *Send(port), buffer)* and *Receive(port), buffer)* are used from the *MainLoop()* code to send (or receive) buffers over (or from) a port of the ***dabc::ModuleSync***.

dabc::ModuleAsync : Subclass of ***dabc::Module***; defines interface for an asynchronous module that must never block the execution. Several ***dabc::ModuleAsync*** objects may be assigned to one working thread. User must either re-implement virtual method *ProcessUserEvent()* wich is called whenever **any** event for this module (i.e. this working processor) is processed by the working thread. Or the user may implement callbacks for special events (e.g. *ProcessInputEvent()*, *ProcessOutputEvent()*, *ProcessPoolEvent()*,..) that are invoked when the corresponding event is processed by the working thread. The events are dispatched to these callbacks by the *ProcessUserEvent()* default implementation then. To avoid blocking the shared working thread, the user must always check if a resource (e.g. a port, a memory pool) would block before any calls (e.g. *Send()*, *TakeBuffer()*) are invoked on it. All callbacks must **return** in the "I would block" case; on the next time the callback is executed by the framework the user must check the situation again and react appropriately (might require own bookkeeping of available resources).

dabc::Port : A connection interface between module and transport. From inside the module scope, only the ports are visible to send or receive buffers by reference. Data connections between modules (i.e. transports between the ports of the modules) are set up by the application plug-in by methods of ***dabc::Manager*** specifying the full module/port names. For ports on different nodes, commands to establish a connection may be send remotely (via controls layer, e.g. DIM) and handled by the manager of each node.

dabc::Transport : Moves buffers by reference between two ports, or between a port and the device which owns the transport, respectively. Implementation may be subclass of ***dabc::WorkingProcessor***.

dabc::Device : Is subclass of ***dabc::WorkingProcessor***. Represents generally an module-external data producing or -consuming entity, e.g. a network connection. Each device may create and manage several transport objects. The ***dabc::Transport*** and ***dabc::Device*** base classes have various implementations:

- ***dabc::LocalTransport*** and ***dabc::LocalDevice*** for memory transport within one process
- ***dabc::SocketTransport*** and ***dabc::SocketDevice*** for tcp/ip sockets
- ***dabc::VerbsTransport*** and ***dabc::VerbsDevice*** for InfiniBand **verbs** connection
- ***dabc::PCITransport*** and ***dabc::PCIBoardDevice*** for DMA I/O from PCI or PCIe boards

dabc::Manager : Subclass of ***dabc::WorkingProcessor*** (and by that also of ***dabc::CommandReceiver***) and ***dabc::CommandClient***. The manager is one single instance per process; it combines different roles:

1. It is a manager of all ***dabc::Basic*** objects in the process scope. Objects (e.g. modules, devices, parameters) are kept in a folder structure and identified by full path name.
2. It defines the interface to the controls system (state machine, remote command communication, parameter export); this is to be implemented in a subclass. The manager handles the command and parameter flow locally and remotely: commands submitted to the local manager are directed to the command receiver where they shall be executed. If any parameter is changed, this is recognized by the manager and optionally forwarded to the associated controls system. Current implementations of manager are:
   - ***dabc::StandaloneManager*** provides simple socket controls connection to send remote commands within a multi node cluster. This is used for the standalone Bnet examples without higher level control system.
   - ***dabc::xd::Manager*** for use in the XDAQ framework. Provides DIM as transport layer for inter-module commands. Additionally, parameters may be registered and updated automatically as DIM services. This manager is used in the XDAQ Bnet example from the dabc::xd::Node. [[DabcXdaq][More details of this implementation are described here.]]
3. It provides interfaces for user specific plug-ins that define the actual set-up: several ***dabc::Factory*** objects to create objects, and one ***dabc::ApplicationPlugin*** object to define the state machine transition actions.

dabc::Factory : Factory plug-in for creation of modules and devices.

dabc::ApplicationPlugin : Subclass of ***dabc::WorkingProcessor*** (and therefore ***dabc::CommandReceiver***). Interface for the user specific code. Defines the actions on transitions of the finite state machine of the manager. May export parameters for configuration, and may define additional commands.

2.5.2.2 Bnet classes

The classes of the Bnet package, providing functionalities of the event builder network.

bnet::ClusterPlugin : Subclass of ***dabc::ApplicationPlugin*** to run on the cluster controller node of the builder network.

1. It implements the master state machine of the Bnet. The controlling GUI usually sends state machine commmands to the controller node only; the Bnet cluster plugin works as a command fan-out and state observer of all worker nodes.
2. It controls the traffic scheduling of the data packets between the worker nodes by means of a data flow controller (class ***bnet::GlobalDFCModule***). This controller module communicates with the Bnet sender modules on each worker to let them send their packets synchronized with all other workers.
3. It may handle failures on the worker nodes automatically, e.g. by reconfiguring the data scheduling paths between the workers.

bnet::WorkerPlugin : Subclass of ***dabc::ApplicationPlugin*** to run on the worker nodes of the builder network.

1. Implements the local state machine for each worker with respect to the Bnet functionality.
2. It registers parameters to configure the node in the Bnet, and methods to set and check these parameters.

3. Defines factory methods *CreateReadout()*, *CreateCombiner()*, *CreateBuilder()*, *CreateFilter()*, *CreateStorage()* to be implemented in user specific subclass. These methods are used in the worker state machine of the Bnet framework.

bnet::GeneratorModule : Subclass of ***dabc::ModuleSync***. Framework class to fill a buffer from the assigned memory pool with generated (i.e. simulated) data.

1. Method *GeneratePacket(buffer)* is to be implemented in application defined subclass (e.g. ***bnet::MbsGeneratorModule***) and is called frequently in module's *MainLoop()*.
2. Each filled buffer is forwarded to the single output port of the module.

bnet::FormaterModule : Subclass of ***dabc::ModuleSync***. Framework class to format inputs from several readouts to one data frame (e.g. combine an event from subevent readouts on that node).

1. It provides memory pools and one input port for each readout connection (either ***bnet::GeneratorModule*** or connection to a readout transport).
2. The formatting functionality is to be implemented in method *MainLoop()* of user defined subclass (e.g. ***bnet::MbsCombinerModule***).

bnet::SenderModule : Subclass of ***dabc::ModuleAsync***. Responsible for sending the event data frames to the receiver nodes, according to the network traffic schedule as set by the Bnet cluster plugin.

1. It has **one** input port that gets the event packets (or time sorted frames) from the preceding Bnet formatter module (or a special event combiner module, resp.). The input data frames are buffered in the Bnet sender module and analyzed which frame is to be sent to what receiver node at what time. This can be done in a non-synchronized "round-robin" fashion, or time-synchronized after a global traffic schedule as evaluated by the Bnet cluster plugin.
2. Each receiver node is represented by one output port of the Bnet sender module that is connected via a network transport (tcp socket, InfiniBand verbs) to an input port of the corresponding Bnet receiver node.

bnet::ReceiverModule : Subclass of ***dabc::ModuleAsync***. Receives the data frames from the Bnet sender modules and combines corresponding event packets (or time frames, resp.) of the different senders.

1. It has **one** input port for each sender node in the Bnet. The data frames are buffered in the Bnet receiver module until the corresponding frames of all senders have been received; then the combined total frame is send to the output port.
2. It has **exactly one** output port. This is connected to the ***bnet::BuilderModule*** implementation (or a user defined builder module, resp.) that performs the actual event tagging and "first level filtering" due to the experimental physics.

bnet::BuilderModule : Subclass of ***dabc::ModuleSync***. Framework class to select and build a physics event from the data frames of all Bnet senders as received by the receiver module.

1. It has **one** input port connected to the Bnet receiver module. The data frame buffers of all Bnet senders are transferred serially over this port and are then kept as an internal **std::vector** in the Bnet builder module.
2. Method *DoBuildEvent()* is to be implemented in user defined subclass (e.g. ***bnet::MbsBuilderModule***) and is called in module's *MainLoop()* when a set of corresponding buffers is complete.
3. It provides **one** output port that may connect to a Bnet filter module, or a user defined output or storage module, resp.
4. Currently the user has to implement the sending of the tagged events to the output port explicitly in his subclass. **Should be done automatically in** *MainLoop()* **later?**

bnet::FilterModule : Subclass of ***dabc::ModuleSync***. Framework class to filter out the incoming physics events according to the experiment's "trigger conditions".

1. Has **one** input port to get buffers with already tagged physics events from the preceding Bnet builder module.
2. Has **one** output port to connect a user defined output or storage module, resp.
3. Method *TestBuffer(buffer)* is to be implemented in user defined subclass (e.g. ***bnet::MbsFilterModule***)

and is called in module's *MainLoop()* for each incoming buffer. Method should return `true` if the event is "good" for further processing.

4. Forwards the tested "good" buffers to the output port and discards all other buffers.

3 *DABC* Programmer Manual: Plugins

[programmer/prog-plugin.tex]

3.1 *DABC* plug-in mechanism

4 *DABC* Programmer Manual: Parameters and Setup

[programmer/prog-setup.tex]

4.1 Setting up system

4.1.1 Parameter class

Configuration and status information of objects can be represented by ***Parameter*** class. Any objects, derived from ***WorkingProcessor*** class, can has a list of parameters, assigned to it - for instance, classApplication, ***Device***, ***Module***, ***Port*** classes.

There are number of class ***WorkingProcessor*** methods to create parameter objects of different kinds and access their values. Full list one can see in following table:

Type	Class	Create	Getter	Setter
string	StrParameter	CreateParStr	GetParStr	SetParStr
double	DoubleParameter	CreateParDouble	GetParDouble	SetParDouble
int	IntParameter	CreateParInt	GetParInt	SetParInt
bool	StrParameter	CreateParBool	GetParBool	SetParBool

As one can see, to store boolean parameter, string is used. If value of string is "true" (in lower case), boolean value of such parameter recognized as true, otherwise false.

For any type of parameter GetParStr/SetParStr methods can be used.

It is recommended to use class ***WorkingProcessor*** methods to create parameters and access its values, but one also can use FindPar() method to find parameter object and use its methods directly.

4.1.2 Use parameter for control

The main advantage to use parameter objects is that parameter values can be observed and changed using controlling system.

At any time, when parameter value changed by program via SetPar... methods, control system informed and represents such change in appropriate GUI element. At the same time, if user modifies parameter value in the GUI, value of parameter object will be changed and correspondent parent object (module, device) get callback via virtual method ParameterChanged(). Implementing correct reaction on this call, one can provide possibility to reconfigure/adjust running code on the fly.

Parameter value may be fixed via Parameter::SetFixed() method. This blocks possibility to change parameter value from both program and user side. Only when fixed flag set again to false, parameter value can be changed.

Not all parameters objects should be visible to control system. There is so-called visibility level of parameter, which is assigned to parameter when instance is created. Only if visibility level smaller than current level (configured in Manager::ParsVisibility()), parameter will be "seen" by control system. Such level is configured for complete object via WorkingProcessor::SetParDflts() function before parameter object is created.

4.1.3 Example of parameters usage

Lets consider example of module, which uses parameter:

```
class UserModule : public dabc::ModuleAsync {
   public:
      UserModule(const char* name, dabc::Command* cmd) :
         dabc::ModuleAsync(name, cmd)
      {
         CreateParBool("Output", true);
         CreateParInt("Counter", 0);
         CreateTimer("Timer", 1.0, false);
      }

      virtual void ProcessTimerEvent(dabc::Timer*)
      {
         SetParInt("Counter", GetParInt("Counter")+1);
         if (GetParBool("Output"))
            DOUT1(("Counter = %d", GetParInt("Counter")));
      }
};
```

In module constructor two parameters are created - boolean and integer and timer with period of 1 s. When module started, value of integer parameter will be changed every second. If boolean parameter is set to true, value of counter will be displayed on debug output. Using control system, value of boolean parameter can be changed. To detect and react on such change, one should implement following method:

```
      virtual void ParameterChanged(dabc::Parameter* par)
      {
         if (par->IsName("Output"))
            DOUT1(("Output flag changed to %s", DBOOL(GetParBool("Output"))));
      }
```

From the performance reasons one should avoid usage of parameter getter/setter methods (like GetParBool() or SetParInt()) inside loop, which executed many times. Main aim of parameter object is to provide connection to control system and in other situation normal class members should be used.

4.1.4 Configuration parameters

Parameters may also be used for configuration of newly created objects.

There are another use of parameters

Configuration parameters for mostly all objects, created in dabc, can be placed in xml file.

4.1.5 Configuration file example

Lets consider simple example of configuration file:

```
<?xml version="1.0"?>
<dabc version="1">
  <Context name="Generator">
    <Run>
      <lib value="libDabcMbs.so"/>
      <func value="StartMbsGenerator"/>
    </Run>
    <Module name="Generator">
       <Port name="Output">
          <OutputQueueSize value="5"/>
          <MbsServerPort value="6000"/>
       </Port>
    </Module>
  </Context>
</dabc>
```

This is an example of xml file for mbs generator, which produces mbs events and provides them to mbs transport server. Other application (dabc or Go4) can connect to that server and read generated mbs events.

4.1.6 Basic syntax

DABC configuration file should always contain <dabc> as root node. Inside <dabc> node one or several <Context> nodes should exists. Each <Context> node corresponds to independent application context, which runs as independent executable. Optionally <dabc> node can has <Variables> and <Defaults> nodes, which are described further.

4.1.7 Context

<Context> node can has two optional attributes:

"host" host name, where executable should runs, default is localhost
"name" application (manager), default is host name.

Inside <Context> node configuration parameters for modules, devices, memory pools are stored. In example file one sees several parameters for output port of generator module.

4.1.8 Run parameters

Usually <Context> node has <Run> subnode, where user defines different parameters, relevant to start application:

lib library name, which should be loaded. Several libraries names can be specified.

func function name which should be called to create modules. It is alternative to writing subclass of ***Application*** and instantiating it.
port ssh port number of remote host
user account name to be used for ssh (login without password should be possible)
init init script, which should be called before dabc application starts
test test script, which is called when test sequence is run by run.sh script
timeout ssh timeout
debugger argument to run debugger. Value should be like "gdb -x run.txt –args", where file run.txt should contain commands "r bt q".
workdir directory where dabc application should start
debuglevel level of debug output on console, default 1
logfile filename for log output, default none
loglevel level of log output to file, default 2
DIM_DNS_NODE node name of dim dns server, used by dim control
DIM_DNS_PORT port number of dim dns server, used by dim control

4.1.9 Variables

In root <dabc> node one can insert <Variables> node, which may contain definition of one or several variables. Once defined, such variables can be used in any place of configuration file to set parameters values. In this case syntax to set parameter is:

```
<ParameterName value="${VariableName}"/>
```

It is allowed to combine variable with text or other variable, but neither arithmetic nor string operations are supported.

Using variables, one can modify example in following way:

```
<?xml version="1.0"?>
<dabc version="1">
  <Variables>
    <myname value="Generator"/>
    <myport value="6010"/>
  </Variables>
  <Context name="Mgr${myname}">
    <Run>
      <lib value="libDabcMbs.so"/>
      <func value="StartMbsGenerator"/>
    </Run>
    <Module name="${myname}">
       <Port name="Output">
          <OutputQueueSize value="5"/>
          <MbsServerPort value="${myport}"/>
       </Port>
    </Module>
  </Context>
</dabc>
```

Here context name and module name are set via myname variable and mbs server socket port is set via myport variable.

There are several variables, which are defined by configuration system:

- DABCSYS - top directory of dabc installation
- DABCUSERDIR - user-specified directory
- DABCWORKDIR - current working directory
- DABCNUMNODES - number of <Context> nodes in configuration files
- DABCNODEID - sequence number of current <Context> node in configuration file

Any environment variable can also be used as variable to set parameter values.

4.1.10 Default values

There are situations, when one need to set same value to several similar parameters, for instance same output queue length for all output ports in the module. One possible way is to use variables (as described before) and set parameter value via variable. Disadvantage of such approach that one should expand xml files and in case of big number of ports xml file will be very long and unreadable.

Another possibility to set several parameters at once - create <dabc/Defaults> node and specify cast rule, using "*" or "?" symbols like that:

```
<?xml version="1.0"?>
<dabc version="1">
  <Variables>
    <myname value="Generator"/>
    <myport value="6010"/>
  </Variables>

  <Context name="Mgr${myname}">
    <Run>
      <lib value="libDabcMbs.so"/>
      <func value="StartMbsGenerator"/>
    </Run>
    <Module name="${myname}">
       <Port name="Output">
          <MbsServerPort value="${myport}"/>
       </Port>
    </Module>
  </Context>
  <Defualts>
    <Module name="*">
       <Port name="Output*">
          <OutputQueueSize value="5"/>
       </Port>
    </Module>
  </Defualts>
</dabc>
```

In this case for all ports, which names are started with string "Output" from any module, output queue length will be 5.

In form, as it is specified in example, such multicast rule will be applied for all contexts from configuration file means by such rule we set output queue length for all modules on all nodes. This allow us to create compact xml files for big multi-nodes configuration.

4.1.11 Configuration priorities

Setting parameter values via xml file not the only possibility for system configuration. Usually parameter has default value, which is specified at the time when parameter is created. One also can supply some configuration parameters to the command, when module or device or transport is created.

Here possibilities to set parameter value listed with decreased priority:

- Command argument. It has the highest priority in all cases and allows programmer to overwrite any configuration parameters which user may define in configuration file
- Parameter value from appropriate <Context> node. In this case names of parameter and all its parents should exactly match to names which are specified in xml file. No any kind of cast is supported.
- Parameter value from <Defaults> node. Parameter name and names of at least one parent should match to multicast rules. If there is several matches exists match with maximal number of parent matches will be preferred. For instance node <Defualts/Context/Module/Port/OutputQueueSize> will be preferable rather than <Defualts/Port/OutputQueueSize>.
- Default parameter value.

5 *DABC* Programmer Manual: GUI

[programmer/prog-gui.tex]

5.1 GUI Guide lines

The *DABC*GUI is written in Java. It uses the DIM Java package to register the DIM services provided by the *DABC*DIM servers. It is generic in that it builds most of the panels from the services available.

5.2 GUI Panels

5.2.1 *DABC* launch panel

5.2.2 *MBS* launch panel

5.2.3 Combined *DABC* and *MBS* launch panel

5.2.4 Command panel

5.2.5 Parameter selection panel

5.2.6 Monitoring panels

5.2.7 Parameter table

5.2.8 Logging window

5.3 GUI save/restore setups

5.4 Application specific GUI plug-in

Besides the generic part of the GUI it might be useful to have specific user panels as well, integrated in the generic GUI. This is done by implementation of a xPanelPrompt derived class. The class name can be passed as argument to the java command starting the GUI. The class must implement some interfaces.

5.4.1 Application interfaces

A user may implement these interfaces in his own menues. He can connect his own call back functions to parameters, and a command function to be called when a command shall be executed. He may create his own panels for display using the graphical primitives like rate meters.

5.4.2 Java Interfaces to be implemented by application

5.4.2.1 Interface xiUserPanel

`public abstract void init(xiDesktop desktop, ActionListener actionlistener);` Called by xgui after instantiation. The desktop can be used to add frames (see below).
`public String getHeader();` Must return a header/name text after instantiation.
`public String getToolTip();` Must return a tooltip text after instantiation.
`public ImageIcon getIcon();` Must return an icon after instantiation.
`public xiUserCommand getUserCommand();` Must return an object implementing xiUserCommand, or null. See below.
`public void setDimServices(xiDimBrowser browser);` Called by xgui whenever the DIM services had been changed. The browser provides the command and parameter list (see below). One can select and store references to commands or parameters. A xiUserInfoHandler can be registered for each selected parameter. Then the infoHandler method is called for each parameter update.
`public void releaseDimServices();` All local references to commands or parameters must be cleared!

5.4.2.2 public interface xiUserCommand

`public boolean getArgumentStyleXml(String scope, String command);` Return true if command shall be composed as XML string, false if MBS style string. scope is specified in the XML command descriptor, command is the full command name.

5.4.2.3 public interface xiUserInfoHandler

```
public void infoHandler(xiDimParameter p);
```

5.4.3 Java Interfaces provided by xgui

5.4.3.1 Interface xiDesktop

```
public void addDesktop(JInternalFrame frame, String name);
```

5.4.3.2 Interface xiDimBrowser

```
public xiDimParameter[] getParameters();
public xiDimCommand[] getCommands();
public void setInfoHandler(xiDimParameter parameter,
                           xiUserInfoHandler infohandler);
public void sleep(int s);
```

5.4.3.3 Interface xiDimCommand

```
public void exec(String command);
public xiParser getParserInfo();
```

5.4.3.4 Interface xiDimParameter

```
public xRecordMeter getMeter();
public xRecordState getState();
public xRecordInfo getInfo();
public xiParser getParserInfo();
```

5.4.3.5 Interface xiParser

```
public String getDns();
public String getNode();
public String getNodeName();
public String getNodeID();
public String getApplicationFull();
public String getApplication();
public String getApplicationName();
public String getApplicationID();
public String getName();
public String getNameSpace();
public String[] getItems();
public String getFull();
public String getFull(boolean build);
public String getCommand();
public String getCommand(boolean build);
public int getType();
public int getState();
public int getVisibility();
public int getMode();
public int getQuality();
public int getNofTypes();
public int[] getTypeSizes();
public String[] getTypeList();
public String getFormat();
public boolean isNotSpecified();
public boolean isSuccess();
public boolean isInformation();
public boolean isWarning();
public boolean isError();
public boolean isFatal();
public boolean isAtomic();
public boolean isGeneric();
public boolean isState();
public boolean isInfo();
public boolean isRate();
public boolean isHistogram();
public boolean isCommandDescriptor();
public boolean isHidden();
public boolean isVisible();
public boolean isMonitor();
```

```
public boolean isChangable();
public boolean isImportant();
public boolean isLogging();
public boolean isArray();
public boolean isFloat();
public boolean isDouble();
public boolean isInt();
public boolean isLong();
public boolean isChar();
public boolean isStruct();
```

References

[1] CBM collaboration, “CBM Experiment: Technical Status Report”, Januar 2005

[2] CMS collaboration, http://cmsinfo.cern.ch/outreach/, “CMS Outreach”, CERN 2006

[3] The Experimental Physics and Industrial Control System website, http://www.aps.anl.gov/epics/index.php, Argonne National Laboratory 2006

[4] C. Gaspar et al., "DIM - Distributed Information Management System" , http://dim.web.cern.ch/dim/, CERN May 2006

[5] Y. Liu and P. Sinha, “A Survey Of Generic Architectures For Dependable Systems”, IEEE Canadian Review, Spring 2003

[6] The National Instruments Labview web site, http://www.ni.com/labview/, National Instruments Corporation 2006

[7] L. Orsini and J. Gutleber, “The XDAQ Wiki Main Page” http://xdaqwiki.cern.ch/index.php, CERN 2006

[8] L. Orsini and J. Gutleber, “I2O Messaging” http://xdaqwiki.cern.ch/index.php/I2O_Messaging , CERN 2006

[9] L. Orsini and J. Gutleber, “XDAQ Monitor application” http://xdaqwiki.cern.ch/index.php/Monitor_CGI_interface , CERN 2005

[10] L. Orsini and J. Gutleber, http://xdaqwiki.cern.ch/index.php/Configuration_schema “XDAQ XML configuration schema”, CERN 2006

[11] D. Stenberg et al., The curl and libcurl web site, http://curl.haxx.se/, HAXX HB 2006

[12] SystemC website, http://www.systemc.org/

[13] The W3C Consortium, “SOAP Version 1.2 Part 1: Messaging Framework”, http://www.w3.org/TR/soap12-part1, W3C Recommendation, 24 June 2003

[14] K. Whisnant, R.K. Iyer, Z. Kalbarczyk, and P. Jones, “The Effects of an ARMOR-based SIFT Environment on the Performance and Dependability of User Applications”, University of Illinois, 2006

[15] The Wikipedia, “Finite State Machine”, http://en.wikipedia.org/wiki/State_machine, Wikipedia 2006