Open source consortium of Instrumentation

DAQ-Middleware 1.2.1 Development Manual

Hiroshi Sendai
Institute of Particle and Nuclear Studies, High Energy Accelerator Research Organization

$Date: 2012/09/28 05:27:41 $

Outline

This document is the manual for the development of DAQ-Middleware. It describes the followings.

1. How to prepare the development environment for DAQ-Middleware 1.2.1.
2. How to use the development environment for DAQ-Middleware 1.2.1.
3. How to create and start the sample components.

For the items implemented in DAQ-Middleware 1.2.1, see "DAQ-Middleware 1.1.0 Technical Manual" [2].

The sample components created in this text are; the 'SampleReader' component to read out data from the read-out module, and the 'SampleMonitor' component to receive data from the 'SampleReader' component and then display their histogram on the screen. The 'SampleReader' has been designed to read out data from a software emulator so that no particular hardware is required.

The following programming skills should be premised in this manual.

1. Ability to write programs in C and C++ languages.
2. Ability to use development tools such as gcc and make.
3. Ability to program with sockets to read out data from network instruments.

Table of Contents

1 About this manual

This manual consists of the followings.

- How to prepare DAQ-Middleware 1.2.1 development environment (Section 2)
- A brief outline for DAQ-Middleware (Section 3)
- How to use the DAQ-Middleware 1.2.1 development environment (Section 4)
- Preparation of the development directory (Section 5)
- Confirmation of state transition using the 'Skeleton' components (Section 6)
- Data transfer between components using simple components (Section 7)
- How to create and start sample components (Section 8 and after)
- Parameter modification using the 'Condition' database (Section 13)

This manual describes how to develop components. It does not mention how to deploy them to any (actual) data-collection systems.

For the design and implementation of DAQ-Middleware 1.2.1, as well as available classes and methods, see the "DAQ-Middleware 1.1.0 Technical Manual" [2].

About DAQ components' source code described in this manual

The source codes described in this manual will be placed under the directory '/usr/share/daqmw/examples', once DAQ-Middleware 1.2.1 is installed.
Please see the source codes under this directory if you wish to avoid typing in the codes manually, or to find any missing part of the source codes of this manual.

The version of this manual

The version of this manual can be identified with 'Date:' on its cover page.

Set-up of DAQ-Middleware to 'Scientific Linux 5.x or 6.x'

The set-up (of this middleware) to the 'Scientific Linux 5 or 6' can be done easily with the 'yum' command described in Section 2.2.

About DAQ-Middleware source files

You can download the DAQ-Middleware source files from the URL in Section 2.3 if their binaries are not provided for in the OS, the versions don't match, you want to modify the libraries attached to the DAQ-Middleware, or for any other reason.

2 Preparation of development environment

The following three ways are currently available to prepare the DAQ-Middleware 1.2.1 development environment.

1. To use the 'VMware Player' image provided from the DAQ-Middleware development group.
2. To install RPM binaries to the 'Scientific Linux 5.x or 6.x'. (The binaries forboth 'i686 (32 bits)' and 'x86_64 (64 bits)' are provided).
3. To set up and install the dependents from their source files.

Below, these are described in the above order.

2.1 When using the 'VMware Player'

The 'Scientific Linux 5.8' disk image is shown, which is playable at VMware which has already been installed and set up the necessary software in this manual. With this image, you can readily start to create DAQ-Middleware components.

Please download the 'VMware Player' from the 'VMware' site, 'http://www.vmware.com/jp/products/player/'. There may be installation problems for the latest 'VMware Player 3.x' depending on which CPU you use. To use the 'VMware Player 3.x', your CPU needs to support CMOV, PAE, TSC, FXSAVE commands. Most of the recent CPUs should support these commands. The '3.x' was confirmed uninstallable for computers with Pentium M for its CPU, as they do not have PAE. Please download the 'VMware Player 2.5.4' from http://www.vmware.com/download/player/download.html, when '3.x' is not installable.

The 'Scientific Linux 5.8' disk image operatable on 'VMware Player' can be downloaded from http://daqmw.kek.jp/vmplayer/sl-55-daqmw.zip. The user name 'daq' and password 'abcd1234' have been registered for general use. The 'root' password is also 'abcd1234'.

This image is assigned with 1024MB for its memory capacity. Should more memory be needed, select this disk image on the screen that appears after clicking the 'VMware Player' icon, and then adjust the memory settings by choosing "Edit the virtual machine set-up (仮想マシン設定の編集)" at the right, and select "Hardware" -> "Memory", although there should not be any memory problems with the default component systems created in this manual.

'ROOT' (http://root.cern.ch/), used in creating the histograms with the monitor component developed in this manual, is placed in '/usr/local/root'. The value for the environment variable 'ROOTSYS' is set to '/usr/local/root' when logging in as a 'daq' user.

This 'VMware Player' image is created by generating 'Scientific Linux 5.8' image, and then the binaries are installed using the RPM described in the next Section.

2.2 How to install 'RPM' binaries to 'Scientific Linux 5.x or 6.x'

The 'Scientific Linux' is one of the Linux distributions based on 'RedHat Enterprise Linux'. For details, see http://www.scientificlinux.org/ . For how to set up the 'Scientific Linux 5.x or 6.x', see Appendix F.

2.2.1 Notice for those who have been using the DAQ-Middleware 2008.10 to 2009.10 Editions.

The distribution URL for 'rpm' has been changed since DAQ-Middleware 1.0.0. For updates, please delete your old environment using the commands described in the next Section before any new set-up. To delete, download the files from 'http://daqmw.kek.jp/src/daqmw-rpm', and then type in the followings as a 'root' user.

```
# chmod +x daqmw-rpm
# ./daqmw-rpm distclean
```

You may use 'sh daqmw-rpm distclean' instead of 'chmod'.
(Notice) This file is a shell script to execute the following commands sequentially by 'daqmw-rpm distclean'.

```
rpm -e kek-daqmiddleware-repo
rpm -e OpenRTM-aist
rm -fr /var/cache/yum/kek-daqmiddleware
```

2.2.2 How to set up

To install under the 'Scientific Linux 5.x or 6.x' environment the 'RPM' binaries provided by the DAQ-Middleware development group, download the files from 'http://daqmw.kek.jp/src/daqmw-rpm', and then execute the following commands as 'root' user.

```
root# chmod +x daqmw-rpm
root# ./daqmw-rpm install
```

You may use 'sh daqmw-rpm install' instead of 'chmod'.
(Notice) This file is a shell script to execute the following commands sequentially by 'daqmw-rpm install'.

```
root# rpm -ihv http://daqmw.kek.jp/rpm/el5/noarch/kek-daqmiddleware-repo-2-0.noarch.rpm
root# yum --disablerepo='*' --enablerepo=kek-daqmiddleware install DAQ-Middleware
```

The log generated on the execution of these commands is listed in Appendix C.

The 'rpm' packages to be installed with this 'yum' command are as follows. The download file size of the 'rpm' package for 32 bits is 22MB, and that for 64 bits is 23MB. The total file size after the installation is 78MB for 32 bits and 95MB for 64 bits.

- DAQ-Middleware-1.2.1
- OpenRTM-aist-1.0.0 (+ the patches not yet released)
- OmniORB server, library, development environment
  - omniORB-doc-4.1.6
  - omniORB-servers-4.1.6
  - omniORB-utils-4.1.6
  - omniORB-devel-4.1.6
  - omniORB-4.1.6
- In ‘SL 5.x’, ‘xerces-c-2.7.0’ and ‘xerces-c-devel-2.7.0’. Also, in ‘SL 6.x’, ‘xerces-c-3.0.1’ and ‘xerces-c-devel-3.0.1’.
- xalan-c-1.10.0 and xalan-c-devel-1.10.0

Any software (such as ‘ROOT’) necessary for the histogram in creating the monitor component, needs to be separately installed. Also, the following package is required when using the ‘Condition’ database described in Section 13.

- boost

Please install, using the ‘yum’ command etc. This package is included in the distributed items of ‘Scientific Linux 5.x or 6.x’.

2.2.3 How to update using the ‘yum’ command

When a newer version of DAQ-Middleware is released due to any bug-fixing, additions of examples, etc., they can be updated using the following commands (the same commands as in a new installation).

```
wget   http://daqmw.kek.jp/src/daqmw-rpm
chmod +x daqmw-rpm
./daqmw-rpm  install
```

2.2.4 How to uninstall

For uninstallation, execute the following commands.

```
#   ./daqmw-rpm   distclean
```

To delete the files one by one, execute as per the following example.

```
# rpm -qa –last|less
```

Then, the names of the installed 'rpm' packages are displayed in chronological order (the most recent one is displayed first). You can use 'rpm -e' command to start deleting files from the dependents sequentially.

2.3 How to install from source files

The DAQ-Middleware source files are available from 'http://daqmw.kek.jp/src/'. Their file names are in the form of 'DAQ- Middleware-M.m.p.tar.gz' where M, m and p are numbers.

To install from the source files, you need to download and expand, then 'make; make install' them. To compile them, 'OpenRTM-aist' is required. Also, 'OmniORB' is needed to run them. It is easier and recommended to download and install the binary files using 'RPM' or 'yum' when these commands are available for set-up, since preparing all the dependents manually can be time consuming.

The compile tips for OS other than the 'Scientific Linux', 'CentOS', 'RedHat Enterprise Linux (5.x, 6.x)', are summarized in Appendix D.

2.4 Confirmation of installation

When installing and setting up manually, you can confirm in the following ways, if the development environment has been correctly set up.

Confirm if everything for the development environment is in place, using the 'Skeleton' component. The 'Skeleton' component source files should have been installed under the directory '/usr/share/daqmw/examples/ Skeleton/', once the set-up of the DAQ-Middleware 1.2.1 was completed.

Confirm if everything for the development environment is in place as follows.

```
% cp -r /usr/share/daqmw/examples/Skeleton . (a dot (".") needs to be added to the end of line)
%  cd  Skeleton
%  ls
Makefile      Skeleton.cpp       Skeleton.h       SkeletonComp.cpp
%   make
```

If no errors occur, the executable file 'SkeletonComp' should be created. When the 'make' command causes an error and the process is abnormally terminated, find the cause and solve the problem. The solutions vary for different error contents[*1].

[*1] For example, when the compile fails in relation to 'libuuid' on 'Scientific Linux 5.x', 'e2fsprogs-devel' package needs to be installed from the distribution package of 'Scientific Linux 5.x'.

2.5 Directory structure after installation

You can check, as follows, what files have been installed after the installation of the DAQ-Middleware in case you installed from ‘RPM’ or used the ‘VMware Player’.

```
rpm -ql DAQ-Middleware
```

If you installed from the source files, there is no easy method to do this.

Regardless of using the ‘VMware Player’ or setting-up with ‘rpm’ on the native Linux environment, the directory structure of DAQ-Middleware becomes as follows.

/usr/bin

The following commands manually launchable by the user, are placed here. Details on how to use each of these commands will be described later when needed.

run.py, run.pyc, run.pyo

These are the commands to read out the system configuration file and start the components scripted therein, and then launch ‘DaqOperator’ at the end. It is written in ‘Python’. ‘run.pyc’ and ‘run.pyo’ are the byte compile file of ‘run.py’ and its optimized byte compile file, respectively.

daqmw-rpm

This is the utility to set up the DAQ-Middleware with ‘RPM’. For directions on how to use it, see Section 2.2.1 and 2.2.2. This command will be deleted when executing the uninstallation of DAQ-Middleware by ‘rpm -e DAQ-Middleware’. You can download this command from ‘http://daqmw.kek.jp/src/daqmw-rpm’ when you need it for re-setup, etc.

condition_xml2json and xml2json-with-attribute.xslt

In the DAQ-Middleware, the component parameters varying for each run are described in the ‘Condition’ database file. This description is done in ‘XML’. Each of the components reads this database and acquires the parameters. Practically, the XML description is converted to the ‘JSON’ format, then, to be read out by each component, since parsing the raw XML code is time-consuming. For the conversion from ‘XML’ to ‘JSON’ format, ‘condition_xml2json’ is used. ‘xml2json-with-attribute.xslt’ is the ‘XSL’ stylesheet needed for the conversion.

newcomp

This is the command to generate the templates for the files needed to create a new component. Its details are described in Section 4.1.

/usr/include/daqmw

Basic class files etc. for DAQ-Middleware are placed in this directory. Also, the API include files for the library mentioned below, are in here.

/usr/include/daqmw/idl

The directory for IDL files.

/usr/lib/daqmw (32bits) or /usr/lib64/daqmw (64bits)

The files to be used in creating the DAQ components, are in this directory. Here is the socket library and the 'Condition' related library ('json_spirit'). In 'Makefile' to compile the components, 'comp.mk' placed in the below-mentioned '/usr/share/daqmw/mk/' can designate this directory as 'DAQMW_LIB_DIR'. With this function, the component source files can be shared between 32 and 64 bits. For example, it can be used as follows.

```
LDLIBS += -L$(DAQMW_LIB_DIR) -lSock
```

For 'comp.mk', see Section 4.2.

A summary of the libraries installed in this directory as well as some notices for their use, are in Appendix A for your reference.

/usr/libexec/daqmw/DaqOperator

Here are the executable files of the DAQ Operator components. Their source files are under '/usr/share/daqmw/ DaqOperator/'.

/usr/share/daqmw/conf

The templates for the DAQ-Middleware settings-files (configuration files, and condition files) are in this directory. The configuration files used for running the 'SampleReader' and 'SampleMonitor' in this document, are named 'sample.xml' and placed under this directory. The 'sample.xml' cannot be used as is, dependent upon the directory of the executable files. For directions on how to change this, see Section 12.

/usr/share/daqmw/DaqOperator

A set of source codes for the DAQ Operator components are here. When you want to place all the component source files necessary for the DAQ system in one directory, copy the files from this directory concerning the DAQ operator component.

/usr/share/daqmw/docs

The documents for the DAQ-Middleware are in this directory.

/usr/share/daqmw/etc

> The remote boot function is needed to compose the DAQ system using multiple computers. The template files for realizing the remote boot with ‘xinetd’ are in this directory. For the set-up of the remote boot, see Appendix E.

/usr/share/daqmw/examples

> Example components are grouped in this directory. The source files for the components (‘SampleReader’ and ‘SampleMonitor’) to be developed in this document, have such components’ names and are placed under this directory.

/usr/share/daqmw/mk

> The commands always fixed for simplifying ‘Makefile’ description, are grouped in ‘comp.mk’ under this directory. For directions on how to script ‘Makefile’, see Section 4.2.

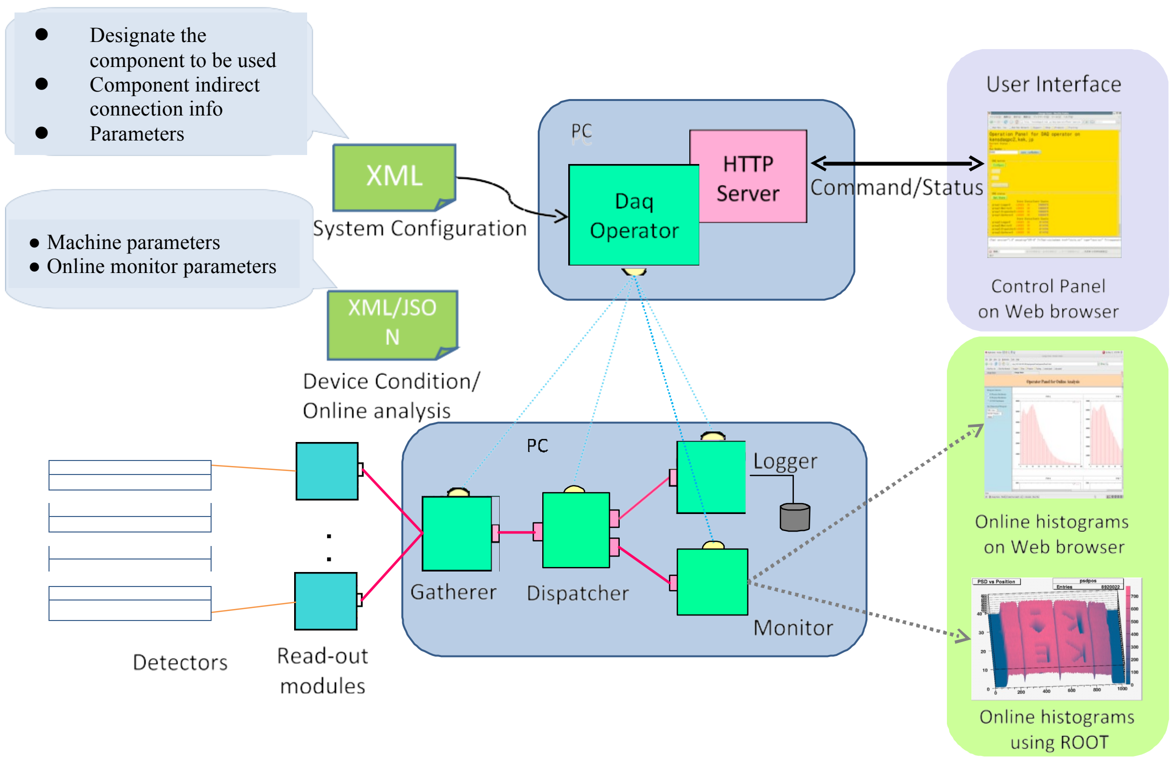


Fig.1 DAQ-Middleware overview

3 Outline of DAQ-Middleware

The architecture of the DAQ-Middleware, the specification of the components, the format of the data transferred between the components, etc. are described in the "DAQ-Middleware 1.1.0 Technical Manual" [2]. Please read through these descriptions before starting your component development. This Section summarizes the necessary knowledge in coding the components.

Fig. 1 shows an overview of the DAQ-Middleware structure. In the DAQ-Middleware, multiple DAQ components (below, simply referred to as 'components') carry out data collection. The 'DaqOperator' controls those components. The 'DaqOperator' handles the run-control by sending to each component the command of connection, data collection start and finish. The 'DaqOperator' reads the system configuration file scripted in 'XML', and then understands the system information, such as which components there are, which components are to be connected to which, etc. The upper system (such as Framework) commands 'DaqOperator'. The 'DaqOperator' has two modes. One is the web mode where the 'DaqOperator' receives the commands through HTTP, and the other is the console mode to receive the keyboard-input commands. In this manual, we command 'DaqOperator' through the console mode. The parameters varying for each run, are scripted in 'XML' as a condition file.

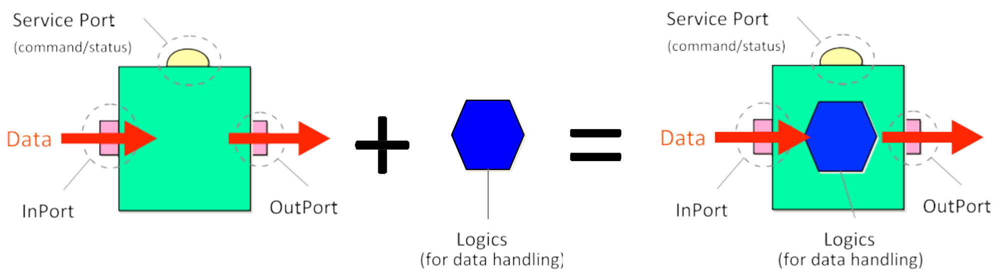


Fig. 2 Logic implementation

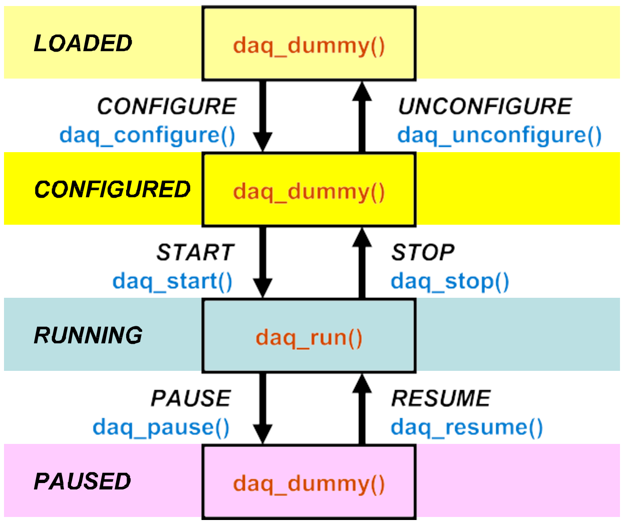


Fig.3 State chart

Since ‘XML’ parsing is a high load process, the file scripted in ‘XML’ is pre-converted to ‘JSON’ format, and then each of the components reads the re-formatted file to acquire the parameters.

3.1 Structure of components

Fig. 2 shows the structure of the components. In the DAQ-Middleware, multiple components communicate with each other to collect data. The first item on the left in fig.2 represents the parts already provided in the DAQ-Middleware. In the DAQ-Middleware, the ‘InPort’ is used for data reception and ‘OutPort’ is used for data transmission. The data sent from the upstream component come in through the ‘InPort’. The data to be sent to the downstream components is written in the ‘OutPort’. The ‘OutPort’ – ‘InPort’ communication function between different components is provided by the DAQ-Middleware. Also, the components have the ‘ServicePort’, used for run-control command reception from the ‘DaqOperator’, as well as for status information transmissions.

Function name	State transition / State	Programming Examples
daq_configure()	LOADED → CONFIGURED	Receive parameter (Read-out module's IP address, port no.) list from 'DaqOperator' to set the values
daq_start()	CONFIGURED → RUNNING	Create socket to 'connect' read-out module
daq_run()	RUNNING	Read data from read-out module and send it to a subsequent component
daq_stop()	RUNNING → CONFIGURED	'disconnect' from read-out module

Table 1 Programming example for 'Gatherer'

Function name	State transition / State	Programming Examples
daq_configure()	LOADED → CONFIGURED	Receive parameter list from 'DaqOperator' to set the values
daq_start()	CONFIGURED → RUNNING	Prepare histogram data
daq_run()	RUNNING	Read data from upstream component and codes it. Then, files the data into histogram data to regularly draw a histogram.
daq_stop()	RUNNING → CONFIGURED	Draw a histogram with the final histogram data

Table 2 Programming example for 'Monitor'

The component developer will program the logic to be realized. For example, the logic of the component to read data from the read-out module and send it to the subsequent component, is reading out data from read-out module via socket and writing it in the outport. Once the data is written in the outport, it is automatically sent to the subsequent component. So, the component developer does not need to implement this part bymanually. There is another logic example which reads data that came in 'InPort' and then draws a histogram.

3.2 Component states and its transition

Each component is in any of the states shown in Fig. 3 State Chart, during its operation. As such the states, 'LOADED', 'CONFIGURED', 'RUNNING', 'PAUSED' are defined. For example, the state immediately after the executable file is loaded to the computer and the component gets started to run as a process, is 'LOADED'. It keeps the 'LOADED' state until it receives the 'CONFIGURE' command from the 'DaqOperator'. When the 'CONFIGURE' command comes in, the state is transitioned to 'CONFIGURED'. On such transitions, the function defined for it will be executed only once. For example, on the transition from 'LOADED' to 'CONFIGURED', 'daq_configure()' is executed.

While in one state, the function corresponding to the state is repeatedly called. For example, 'daq_run() ' is called on the 'RUNNING' state. After one 'daq_run()' finishes and the 'STOP' command has not been input, the 'daq_run()' will be called again. After that, 'daq_run()' will be called repeatedly until the 'STOP' command is received. The function called during each state, needs to be programmed such that the function never blocks the process permanently. This is because a permanent block will hamper the reception of the commands issued to make the transition to the next state. For example, when the 'daq_run()' reads data via socket, implementing a time-out function to the socket to avoid the permanent block will be necessary[*2].

The components are realized with the implementation of these functions. As examples, the items to be implemented in the component (gatherer) reading out data from the read-out module are shown in table 1. Also, an example of the component (monitor) to create a histogram, and then display it on the screen, is shown in Table 2. These are just examples, and do not always need to be implemented.

3.3 Format of data transferred between components

Fig. 4 shows the format of the data transferred between each component. The component header and footer in this Fig. have nothing to do with the header and footer that are (may be) in the data that the read-out module sends in. The formats of the component header and footer are shown in Fig. 5.

When an upstream component sends data to a downstream component, the byte size of the event data to be sent is recorded in 'EventByteSize' in the header. The downstream component that received the data, verifies if it has anything missing in data-reading, by comparing the received actual event data size and 'EventByteSize' in the header. Also, the upstream component sets the number of times it sends data to the downstream component, to 'sequence number' in the footer. The downstream component that received the data, records how many receptions there were from upstream, and then compares it with 'sequence number' in the footer. This enables the verification of any failure in data-receiving.

The setter and getter methods for 'EventByteSize' and 'sequence number', include 'set_event_byte_size()', 'inc_sequence_num()' and 'get_sequence_num()'. All such methods are described in "DAQ Middleware 1.1.0 Technical Manual" [2].

The event data format needs to be decided by the user.

[*2] 'read()' of the socket will be permanently blocked by default when there is no data to read.

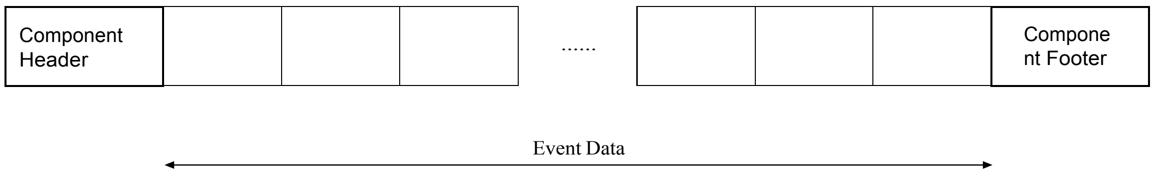


Fig. 4 Format of data transferred between components. For the format of component header and footer, see Fig. 5.

Component Header

Header Magic (0xe7)	Header Magic (0xe7)	Reserved	Reserved	Data ByteSize (24:31)	Data ByteSize (16:23)	Data ByteSize (8:15)	Data ByteSize (0:7)
0 7	8 15	16 23	24 31	32 39	40 47	48 55	56 63

Component Footer

Footer Magic (0xcc)	Footer Magic (0xcc)	Reserved	Reserved	sequence number (24:31)	sequence number (16:23)	sequence number (8:15)	sequence number (0:7)
0 7	8 15	16 23	24 31	32 39	40 47	48 55	56 63

Fig. 5 Component header and footer format

3.4 Read & write of data at ‘InPort’ and ‘OutPort’

Reading and writing of the data at ‘InPort’ and ‘OutPort’ are described with an example of a‘Sample’ component having only one ‘InPort’ and ‘OutPort’ each.

In ‘Sample.h’, the buffer used for each ‘InPort’ and ‘OutPort’ is defined as follows.

```
private:
    TimedOctetSeq               m_in_data;      // InPort
    InPort<TimedOctetSeq>       m_InPort;

    TimedOctetSeq               m_out_data; // OutPort
    OutPort<TimedOctetSeq>   m_OutPort;
```

‘m_in_data’ and ‘m_out_data’ are the constructors for ‘Sample.cpp’ and become the data buffer for ‘InPort’ and ‘OutPort’.

```
Sample::Sample(RTC::Manager*   manager)
    :  DAQMW::DaqComponentBase(manager),
       m_InPort("sample_in",            m_in_data),
       m_OutPort("sample_out",     m_out_data),
```

The data that has arrived at ‘InPort’ from the upstream component, is read by ‘m_InPort.read()’. The return value is either ‘true’ or ‘false’. When ‘false’, the state of ‘InPort’ is checked by ‘check_inPort_status(m_InPort)’. Normally, you should code the component to retry when the ‘check_inPort_status(m_InPort)’ is ‘BUF_TIMEOUT’. When ‘BUF_FATAL’ is given, the occurrence of a fatal error should be reported to the

'DaqOperator' using 'fatal_error_report()'. If data-reading was done normally, the data enter the 'm_in_data.data' array. To obtain the read length, 'm_in_data.data.length()' is used.

To send the data to the downstream component, write it in the 'OutPort'. First, the length of the data to be sent is specified by 'm_out_data. data.length(data length). The data length unit is byte. Next, the data to send is written in 'm_out_data.data' array. Then, 'm_OutPort.write()' is executed. The return value of 'm_OutPort.write()' is either 'true' or 'false'. 'true' indicates a normal data transmission. When 'false', the state of 'OutPort' is confirmed using 'check_outPort_status(m_OutPort)'. If 'check_outPort_status()' gives 'BUF_TIMEOUT', you normally need to code the component to retry. When 'BUF_FATAL' is given, the occurrence of a fatal error should be reported to the 'DaqOperator' using 'fatal_error_report()'.

For more details, see the "DAQ-Middleware 1.1.0 Technical Manual" [2].

3.5 Error handling

When a fatal error occurs with the component, it needs to be reported to the 'DaqOperator' using 'fatal_error_report()'. For details about 'fatal_error_report()', see the "DAQ-Middleware 1.1.0 Technical Manual" [2]. The condition for the fatal error is determined by the component developer. The fatal error will be handled by the upper system or human.

4 Component development environment

4.1 ‘newcomp’ command

When starting your component development, you may rewrite the ‘Skeleton’ components after copying them as in Section 2.4. However, when it comes to assigning a proper name to the component, you will encounter some file-naming such as include-guard name or component name, according to general rules. So, the ‘newcomp’ command is provided to do this automatically. It is in ‘/usr/bin/newcomp’. Specifying the name of the component to develop as an argument for this command, will create, under the current directory, a directory with the specified component name, and then the following files thereunder (e.g. an example of ‘newcomp MyMonitor’ is shown):

- Makefile
- MyMonitor.h
- MyMonitor.cpp
- MyMonitorComp.cpp

```
% newcomp MyMonitor
% ls
MyMonitor
% cd MyMonitor
% ls
Makefile    MyMonitorComp.cpp    MyMonitor.cpp    MyMonitor.h
```

The part ‘MyMonitor’ in the filename is replaced by the component name specified as the argument for ‘newcomp’.

The above logical content is the same as that of the ‘Skeleton’ component besides the include-guard name in ‘Makefile’ is ‘MYMONITOR’ and the component definition part is ‘mymonitor’. You can ‘make’ it as is. So, using the ‘make’ command confirms if the development environment has been correctly set[*3].

Among these files, ‘MyMonitorComp.cpp’ does not need any modification unless you want to put something in the ‘main()’ function for its program structure. For the‘SampleMonitor’ component created in this document, ‘ROOT’ is used to draw a histogram. To generate ‘TApplication’ object in the ‘main()’ function in this component, the ‘SampleMonitorComp.cpp’ is modified.

The methods such as ‘daq_start()’ and ‘daq_run()’ are implemented in ‘MyMonitor.cpp’ to create the component.

[*3] Executing ‘make’ command creates ‘autogen’ directory wherein the automatically generated files are placed. In component development, those files in ‘autogen’ directory do not need any modification.

In terms of data flow between components, the component that sends data to the other components but does not receive them from the others, is called the ‘Source’-type component. The component that receives data from other components but does not send them to the others is called the ‘Sink’-type component. The ‘newcomp’ command has options to delete or add the ‘InPort’ and ‘OutPort’ according to the types of components to develop. Available component types will be shown by displaying the help message of the ‘newcomp’ command with ‘newcomp –h’.

```
$ newcomp -h
Usage:  newcomp  [-f] [-t component_type]  NewCompName
(omitted…)
You may specify component type as -t option.                    Valid component types are:

null
sink
source
(omitted hereafter…)
```

The above ‘null’ type displayed by ‘newcomp –h’ is for the empty template files generated by most of the methods. When not specifying the type with ‘-t’, the files that are the same as the ‘null’ type will be generated. When creating the component which is not ‘Source’-type nor ‘Sink’-type, start your development with this ‘null’-type (or not specifying the type by ‘-t’) (The generated files do not contain anything other than all of the (empty) methods to be implemented.)

To create a ‘Source’-type component, execute the following.

```
% newcomp -tsource MySampleReader
```

Replace the above ‘MySampleReader’ with any component name you want to use. Also, to create ‘Sink’-type component, execute the following.

```
% newcomp -tsink MySampleMonitor
```

Replace the above ‘MySampleMonitor’ with any component name you want to use.

4.1.1 ‘Source’-type logic

‘newcomp -t source MyReader’ defines the followings as a template in ‘MyReader.h’.

```
private:
        TimedOctetSeq                m_out_data;
        OutPort<TimedOctetSeq>   m_OutPort;

private:
        int daq_dummy();
        int daq_configure();
        int  daq_unconfigure();
```

```
    int daq_start();
    int daq_run();
    int daq_stop();
    int daq_pause();
    int daq_resume();

    int parse_params(::NVList* list);
    int read_data_from_detectors();
    int set_data(unsigned int data_byte_size);
    int write_OutPort();

    static const int SEND_BUFFER_SIZE = 4096;
    unsigned char m_data[SEND_BUFFER_SIZE];
    unsigned int m_recv_byte_size;
```

The 3rd and 2nd Lines from the bottom are the buffer (template) used for data-reading from the read-out module. The following template part will be generated in 'MyReader.cpp' so that the data-reading logic can be written therein.

```
int MyReader::read_data_from_detectors()
{
    int received_data_size = 0;
    /// write your logic here
    return received_data_size;
}
```

The specification of 'read_data_from_detectors()' assumed here, is as follows.

- Return value is the number of bytes read out
- Data read out are put in 'm_data'.

Since this is just a template, components do not always need to be implemented as above.

4.1.2 'Sink'-type logic

'newcomp -t sink MyMonitor' defines the following template in 'MyMonitor.cpp'.

```
1  check_header_footer(m_in_data, recv_byte_size); // check header and footer
2  unsigned int event_byte_size = get_event_size(recv_byte_size);
3
4  /////////////          Write component main logic here. /////////////
5  // online_analyze();
6  /////////////////////////////////////////////////////////////////
7
8  inc_sequence_num();                          // increase sequence num.
9  inc_total_data_size(event_byte_size);        // increase total data byte size
```

In this template, 'online_analyze()' function on Line 5 assumes an arrangement wherein a process to draw histogram, etc. is written. Among the data sent from the upstream component, the user data length excluding component header and footer is put in 'event_byte_size', in byte size. The content of the user data starts from 'm_in_data.data [HEADER_BYTE_SIZE]' to 'm_in_data.data[HEADER_BYTE_SIZE + event_byte_size - 1]' (See fig. 4). The logics such as drawing a histogram from this data can be implemented. Since this is just a template, components do not always need to be implemented like this.

4.2 How to write 'Makefile'

The 'Makefile' generated with 'newcomp' command, is shown below (An example with 'newcomp MyMonitor'):

```
COMP_NAME = MyMonitor

all: $(COMP_NAME)Comp

SRCS += $(COMP_NAME).cpp
SRCS  +=  $(COMP_NAME)Comp.cpp

# sample install target
#
# MODE = 0755
# BINDIR = /tmp/mybinary
#
# install: $(COMP_NAME)Comp
#           mkdir -p $(BINDIR)
#           install -m $(MODE) $(COMP_NAME)Comp $(BINDIR)

include  /usr/share/daqmw/mk/comp.mk
```

Since the processes of 'MyMonitor.cpp' and 'MyMonitorComp.cpp' are written in '/usr/share/daqmw/mk/comp.mk', they do not need to be added to 'Makefile' manually (If added, an error occurs).

When implementing the component only with the files generate by the 'newcomp' command, 'Makefile' does not need any modification. With the addition of source files (*.cpp), additional SRCS variables need to be written in Makefile as below (An example for the addition of 'ModuleUtils.cpp' and 'FileUtils.cpp'):

```
COMP_NAME = MyMonitor

all: $(COMP_NAME)Comp

 SRCS += $(COMP_NAME).cpp
 SRCS  +=  $(COMP_NAME)Comp.cpp
 #
 # An example of adding ModuleUtils.cpp and FileUtils.cpp.
 #
 SRCS   +=   ModuleUtils.cpp
 SRCS  +=  FileUtils.cpp
```

```
12
13 # sample install target
14 #
15 # MODE = 0755
16 # BINDIR = /tmp/mybinary
17 #
18 # install: $(COMP_NAME)Comp
19 #           mkdir -p $(BINDIR)
20 #           install -m $(MODE) $(COMP_NAME)Comp $(BINDIR)
21
22 include /usr/share/daqmw/mk/comp.mk
```

Line 10 and 11 are for the additional files. Otherwise, you can specify the object file names put in OBJS variables.

```
OBJS += ModuleUtils.o
OBJS += FileUtils.o
```

You cannot specify a same file to SRCS and OBJS variables as follows (The compile will fail due to symbol overloading):

```
(This is a no good example.)
SRCS += FileUtils.cpp
OBJS += FileUtils.o
```

Also, notice that, with a source filename (*.cpp) wrongly specified to the OBJS variable, ‘make clean’ deletes the source file.

When the generation of ‘FileUtils.o’ requires ‘FileUtils.h’ and ‘FileUtils.cpp’, writing the dependency as follows, enables the compile only with modified source files but not compiling the entire source files, upon any modification in ‘FileUtils.h’ and ‘FileUtils.cpp’.

```
FileUtils.o: FileUtils.h FileUtils.cpp
```

Since the dependency of the cpp file generated by the ‘newcomp’ command (for example, the dependency of ‘MyMonitor.o’ and ‘MyMonitorComp.o’ upon ‘newcomp MyMonitor’) is already written in ‘comp.mk’, it does not need to be manually specified.

In ‘comp.mk’, ‘-I.’, ‘-I/usr/include/daqmw’ and ‘-I/usr/include/daqmw/idl’ are added as CPPFLAGS. You do not need to add ‘-I.’ for reading ‘*.h’ files in the directory where you execute the ‘make’ command. When reading include files in the directories other than these, and ‘/usr/include’, ‘CPPFLAGS +=’ is used like ‘CPPFLAGS += -I/path/to/myheader_dir’.

Also, when you want to use external libraries, add them to ‘LDLIBS’ variables. Some addition should be made to ‘Makefile’ as follows when, for example, using a library ‘mylibrary’ with its include file under ‘/usr/local/include’ and its library file ‘/usr/local/lib/libmylibrary.so’. Line 13 and 14 are the added ones.

```
COMP_NAME = MyMonitor

all: $(COMP_NAME)Comp

 SRCS += $(COMP_NAME).cpp
 SRCS   +=   $(COMP_NAME)Comp.cpp

  #
  # Include files are in '/usr/local/include'.
  # Library files are in '/usr/local/lib/libmylibrary.so'
 # To use the library, add them to'Makefile' as follows.
 #
 CPPFLAGS += -I/usr/local/include
 LDLIBS     +=   -L/usr/local/lib   -lmylibrary

 # sample install target
 #
 #  MODE  =  0755
 # BINDIR = /tmp/mybinary
 #
 # install: $(COMP_NAME)Comp
 #             mkdir -p $(BINDIR)
 #             install -m $(MODE) $(COMP_NAME)Comp $(BINDIR)

 include /usr/share/daqmw/mk/comp.mk
```

The directory ('/usr/lib/daqmw' for 32 bits SL, and '/usr/lib64/daqmw' for 64 bits SL) for the library (socket library, json library) provided by DAQ-Middleware, is referable as 'DAQMW_LIB_DIR'. Since the directory (/usr/include/daqmw) with the include files in it, is already added to CPPFLAGS as mentioned above, you do not need to make such addition manually.

An execution of the 'make' command creates the 'autogen' directory wherein automatically generated files are placed. Any modification to the files in the 'autogen' directory is not necessary in the component development.

Components can be developed in any directory although the 'makefile' subroutine utility (comp.mk) included in DAQ-Middleware is premised upon 1-directory-1component basis.

Use '.cpp' for the source file extension and '.h' for the include file extension when using 'comp.mk' subroutine attached to DAQ-Middleware. Compiling will not be done correctly using any other extensions (e.g. '.cc' or '.hh').

5 Preparation of development directory

This Section describes the premise that the development system is logged-in by the 'daq' user manually. Since multiple components are to be created here, a development directory '/home/daq/MyDaq' is created that can contain them all.

```
% cd
% mkdir MyDaq
% cd MyDaq
% pwd
/home/daq/MyDaq
```

6 Confirmation of state transition using 'Skeleton' component

Here, we confirm the state transition using the 'Skeleton' component. The 'Skeleton' component is the one wherein all the methods necessary for the component to run are implemented with their contents empty. Move to the development directory created in the previous Section, and then create the source files for the 'Skeleton' component using the 'newcomp' command (The generated source files are the same as those under '/usr/share/daqmw/examples/Skeleton'). Move to the newly created 'Skeleton' directory, and execute 'make'.

```
% cd
% cd MyDaq
% newcomp Skeleton
% ls Skeleton
Makefile     SkeletonComp.cpp     Skeleton.cpp     Skeleton.h
% make
(omitted…)
% ls -l SkeletonComp
-rwxrwxr-x 1 daq daq 281923 Apr   1 09:00 SkeletonComp
```

Subsequently, copy the configuration file for running this component.

```
% cd
% cd MyDaq
% cp /usr/share/daqmw/conf/skel.xml .
```

Then, open 'skel.xml' in your editor to check 'execPath'. The 'execPath' needs to designate the executable file the 'SkeletonComp' created above. When exactly following this example, there should be no modification needed. With the 'SkeletonComp' in a different directory, 'execPath' must be edited so that the path for the file is specified in its full path. The code part of '/usr/share/daqmw/conf/skel.xml' is shown below.

```
<configInfo>
    <daqOperator>
        <hostAddr>127.0.0.1</hostAddr>
    </daqOperator>
    <daqGroups>
        <daqGroup gid="group0">
            <components>
                <component  cid="SkeletonComp0">
                    <hostAddr>127.0.0.1</hostAddr>
                    <hostPort>50000</hostPort>
                    <instName>Skeleton0.rtc</instName>
                    <execPath>/home/daq/MyDaq/Skeleton/SkeletonComp</execPath>
                    <confFile>/tmp/daqmw/rtc.conf</confFile>
                    <startOrd>1</startOrd>
                    <inPorts>
                    </inPorts>
                    <outPorts>
                    </outPorts>
                    <params>
                    </params>
                </component>
            </components>
        </daqGroup>
    </daqGroups>
</configInfo>
```

For more details about the tags, see the “DAQ-Middleware 1.1.0 Technical Manual” [2]. The ‘execPath’ designates the full path of the component executable file. Since this component is not to be connected to any other components, the ‘InPorts’ and ‘OutPorts’ are empty.

As described in the “DAQ-Middleware 1.1.0 Technical Manual” [2], the ‘DaqOperator’ takes control of the DAQ system in the DAQ-Middleware. The ‘DaqOperator’ commands the component connections, as well as the start & finish of the data collection, while the commanded components must be launched in other ways beforehand (The ‘DaqOperator’ does not launch each component). There are some ways to boot each component, including a network boot with ‘xinetd’ launched from the command line of ‘shell’. Here, the local boot function of the ‘/usr/bin/run.py’ command included in the DAQ-Middleware, boots the system.

When option-l (l for lambda, not a number ‘one’) is specified with ‘run.py’, ‘run.py’ reads the configuration file specified in the last argument, and then acquires the path name of the component to be started. After it starts the component in that path on the local computer, it launches the ‘DaqOperator’ on the local computer. Also, ‘run.py’ with the ‘-c’ option specified, launches the ‘DaqOperator’ in the console mode. The ‘DaqOperator’ launched in the console mode, reads the commands from the user’s keyboard. Also, the number of data bytes handled by each component is regularly displayed (Each component regularly reports to the ‘DaqOperator’ the number of data bytes it has processed). Since the ‘Skeleton’ component has no data flow, the number of data bytes remains 0.

```
% cd
% cd  MyDaq
% run.py  -c  -l  skel.xml
(Otherwise, the option can be integrally specified as ‘run.py –cl skel.sml’.)
```

‘run.py’ displays the following for a while after its launch (This waiting time varies for different CPU performance although roughly 4 sec).

```
 Command:      0:configure      1:start      2:stop     3:unconfigure       4:pause      5:resume

 RUN  NO:  0
 start  at:               stop at:

  GROUP:COMP_NAME         EVENT  SIZE          STATE          COMP STATUS
group0:SkeletonComp0:                       0           LOADED          WORKING
```

As previously mentioned, the ‘DaqOperator’ outputs these characters, and it waits for your command key input. In this system, the component is only the ‘Skeleton’, and the ‘STATE’ column shows its current state ‘LOADED’. Available commands are displayed in the first line ‘Command:’. The Command input is carried out by pressing a corresponding number key. State transition needs to be done one by one sequentially. For example, pressing ‘start’ in this ‘LOADED’ state will be judged as an inadequate input. With the command input, the ‘DaqOperator’ sends transition commands to each component.

Confirm the ‘STATE’ column as ‘LOADED’ on the above screen. Pressing 0 to ‘configure’, changes the ‘STATE’ column to ‘CONFIGURED’. When subsequently pressing 1 to ‘start’, an input of the run-number is requested. So, input any proper run-number (like 1). Then, ‘STATE’ column changes to ‘RUNNING’. When subsequently pressing 2 to ‘stop’, the ‘STATE’ column changes to ‘CONFIGURED’.

Pressing ‘Ctrl-C’ after ‘stop’ping the component with a press of 2, sends ‘SIGINT’ to the ‘DaqOperator’ to terminate itself. The ‘SIGINT’ is sent also to the component started by ‘run.py’ at the same time as to the ‘DaqOperator’ (Because the ‘DaqOperator’ and each component launched by ‘run.py’ belong to the same process group). Since usually a component ‘exit’s earlier than the ‘DaqOperator’, it still attempts several connections to the component, the following line will be displayed on the screen for the same amount of times as the number of components after pressing ‘Ctrl-C’.

```
### ERROR:      : cannot  connect
```

‘DaqOperator’ will terminate after a while.

The standard output and standard error output of the component launched by ‘run.py’ will be saved under ‘/tmp/daqmw/log. **Component program name**’. In this case, it is saved in ‘/tmp/daqmw/log.SkeletonComp’.

■Confirmation of state transition of ‘Skeleton’ component

Now, we try to confirm the component’s state transition with a modification to the ‘Skeleton’ component. In all the component source files under ‘/usr/share/daqmw/examples/’, ‘m_debug’ variable is defined. Using this enables, or stops the debug message outputs. The program is written as …

```
if(m_debug) {
      std::cerr << "debug message here" << std::endl;
}
```

First, rewrite the initialization part of the ‘Skeleton’ component constructor (in ‘Skeleton.cpp’) as follows.

```
Skeleton::Skeleton(RTC::Manager*    manager)
      :  DAQMW::DaqComponentBase(manager),
         m_InPort("skeleton_in",              m_in_data),
         m_OutPort("skeleton_out",      m_out_data),

         m_in_status(BUF_SUCCESS),
         m_out_status(BUF_SUCCESS),

         m_debug(true)        changed from //false to true
```

Also, the debug message of ‘onExecute’ regularly called is not important now and so commented out.

```
RTC::ReturnCode_t Skeleton::onExecute(RTC::UniqueId ec_id)
{
      // std::cerr << "*** onExecute¥n"; // Commented out for killing the output.
      daq_do();

      return RTC::RTC_OK;
}
```

Further, since ‘daq_dummy()’ in its original form has no debug message, it is added here.

```
int Skeleton::daq_dummy()
{
      std::cerr << "Skeleton::dummy" << std::endl; // Addition

      return 0;
}
```

Now, ‘make’ this, and then, as in the previous Section, start the ‘Skeleton’ component from ‘run.py’ by ‘run.py –cl skel.xml’ to launch the ‘DaqOperator’ in console mode. The component log will be created under ‘/tmp/daqmw/’. The ‘Skeleton’ component’s log is created in ‘/tmp/daqmw/log.SkeletonComp’. So, with a reference to this using ‘tail –f’, command ‘CONFIGURE’ etc. to ‘DaqOperator’ in order to confirm that the methods are about to transit, or the methods repeated during the state are actually called.

7 Example of creating simple components

To be able to transfer data between components, we create simple components.

The components to be created are the ‘Source’-type component (referred as ‘TinySource’ component, here) and the ‘Sink’-type component (referred as ‘TinySink’ component, here) created by ‘newcomp’. The ‘TinySource’ is the component that creates data by itself and sends it out. Also, the ‘TinySink’ is the component that outputs the received data to the standard error output in hexadecimal. Once the DAQ-Middleware 1.2.1 is installed, the source files for ‘TinySource’ and ‘TinySink’ will be placed in ‘/usr/share/daqmw/ examples/TinySource’ and ‘/usr/share/daqmw/examples/TinySink’, respectively. The configuration file to be used is ‘/usr/share/daqmw/conf/tiny.xml’. Execute the followings.

```
% cd
% cd MyDaq
% newcomp -t source TinySource
% newcomp -t sink        TinySink
% cp /usr/share/daqmw/conf/tiny.xml .
```

Then, modify ‘TinySource’ and ‘TinySink’ as below.

■Modification to TinySource.cpp

```
1  int TinySource::read_data_from_detectors()
2  {
3      int received_data_size = 0;
4      /// write your logic here
5      usleep(500000);                                    // Addition
6      for (int i = 0; i < SEND_BUFFER_SIZE; i++) {       // Addition
7          m_data[i] = (i % 256);                         // Addition
8      }                                                  // Addition
9      received_data_size = SEND_BUFFER_SIZE;             // Addition
10     /// end of my tiny logic
11
12     return received_data_size;
13 }
```

Numbers are simply embedded into the buffer secured in ‘TinySource.h’. Since looping this too frequently is troublesome, Line 5 ‘sleep’s for 0.5 sec.

■Modification to TinySink.h

```
private:
    int daq_dummy();
    int daq_configure();
    int daq_unconfigure();
    int daq_start();
    int daq_run();
    int daq_stop();
```

```
 8         int daq_pause();
 9         int daq_resume();
10
11         int parse_params(::NVList* list);
12         int reset_InPort();
13
14         unsigned int read_InPort();
15         //int online_analyze();
16         static const unsigned int RECV_BUFFER_SIZE = 4096; // Addition
17         unsigned char m_data[RECV_BUFFER_SIZE];            // Addition
18         BufferStatus m_in_status;
19         bool m_debug;
```

At Line 16 and 17, the buffer to copy the data at 'InPort' is added.

■Modification to TinySink.cpp

```
 1 check_header_footer(m_in_data, recv_byte_size); // check header and footer
 2 unsigned  int  event_byte_size  =  get_event_size(recv_byte_size);
 3
 4 ////////////          Write component main logic here. ////////////
 5 // online_analyze();
 6 if (event_byte_size > RECV_BUFFER_SIZE) {                                // Addition
 7         fatal_error_report(USER_DEFINED_ERROR1, "Length Too Large");     // Addition
 8 }                                                                        // Addition
 9 memcpy(m_data, &m_in_data.data[HEADER_BYTE_SIZE], event_byte_size);      // Addition
10 for (unsigned int i = 0; i < event_byte_size; i++) {                     // Addition
11         fprintf(stderr, "%02X ", m_data[i]);                             // Addition
12         if ((i + 1) %  16 == 0) {                                        // Addition
13                 fprintf(stderr, "\n");                                   // Addition
14         }                                                                // Addition
15 }                                                                        // Addition
16         /////////////////////////////////////////////////////////
17
18 inc_sequence_num();                          // increase sequence num.
19 inc_total_data_size(event_byte_size);        // increase total data byte size
```

'memcpy()' copies data from 'm_in_data.data[HEADER_BYTE_SIZE]' for the data size 'event_byte_size'. To avoid buffer overrun, the number of event data bytes is confirmed before executing 'memcpy()'. If the number is bigger than the buffer size (RECV_BUFFER_SIZE), it is judged as a fatal error occurrence and then 'fatal_error_report()' reports it to 'DaqOperator'. The 'for' loop starting from Line 10 extracts the data and outputs them to the standard error output.

After modifications, compile it.

```
% cd
% cd MyDaq
% cd  TinySource
%  make
% cd ..
%  cd  TinySink
%  make
% cd ..
```

■Operation test '/usr/share/daqmw/conf/tiny.xml' is the configuration file for the 'Tiny' component. It must be copied before use. Confirm if 'execPath' has a full path of the executable file for the component created above. If not, edit it using the editor. The code part of '/usr/share/daqmw/conf/tiny.xml' is shown below.

```
1  <configInfo>
2      <daqOperator>
3          <hostAddr>127.0.0.1</hostAddr>
4      </daqOperator>
5      <daqGroups>
6          <daqGroup gid="group0">
7              <components>
8                  <component cid="TinySource0">
9                      <hostAddr>127.0.0.1</hostAddr>
10                     <hostPort>50000</hostPort>
11                     <instName>TinySource0.rtc</instName>
12                     <execPath>/home/daq/MyDaq/TinySource/TinySourceComp</execPath>
13                     <confFile>/tmp/daqmw/rtc.conf</confFile>
14                     <startOrd>2</startOrd>
15                     <inPorts>
16                     </inPorts>
17                     <outPorts>
18                         <outPort>tinysource_out</outPort>
19                     </outPorts>
20                     <params>
21                     </params>
22                 </component>
23                 <component cid="TinySink0">
24                     <hostAddr>127.0.0.1</hostAddr>
25                     <hostPort>50000</hostPort>
26                     <instName>TinySink0.rtc</instName>
27                     <execPath>/home/daq/MyDaq/TinySink/TinySinkComp</execPath>
28                     <confFile>/tmp/daqmw/rtc.conf</confFile>
29                     <startOrd>1</startOrd>
30                     <inPorts>
31                         <inPort    from="TinySource0:tinysource_out">tinysink_in</inPort>
32                     </inPorts>
33                     <outPorts>
34                     </outPorts>
35                     <params>
36                     </params>
37                 </component>
38             </components>
39         </daqGroup>
40     </daqGroups>
41 </configInfo>
```

Since the 'TinySource' component has one 'OutPort', 'OutPorts' at Line 17 designates one 'OutPort'. Also, since the 'TinySink' component has one 'InPort', 'InPorts' at Line 30 designates one 'InPort'. For details of the other tags, see the "DAQ-Middleware 1.1.0 Technical Manual" [2].

Now, launch this.

```
%  cd
%  cd  MyDaq
% ls tiny.xml (Confirming the presence of 'tiny.xml'.  Executing the following commands if not yet copied.)
% cp /usr/share/daqmw/conf/tiny.xml .
%  run.py -c -l tiny.xml
(Since it gushes out into the log, stop it after 5 sec or so by pressing 2.)
(If CONFIGURED, press Ctrl-C to get out of 'run.py'.)
```

The 'run.py' starts the component to output component errors to '/tmp/daqmw/log. **component program name**'. Confirm that the data output by 'fprintf()' is recorded in '/tmp/daqmw/log.TinySinkComp'.

Now, the components can communicate with each other. In the actual 'DAQ' system, the 'Source'-type component will be socket-programmed to read data from the read-out module, and send the read data to the subsequent component. Also, the 'Sink'-type component will be designed so that it decodes the data sent from the upstream component using a histogram tool and then draws a histogram, instead of simply outputting it to the standard error output.

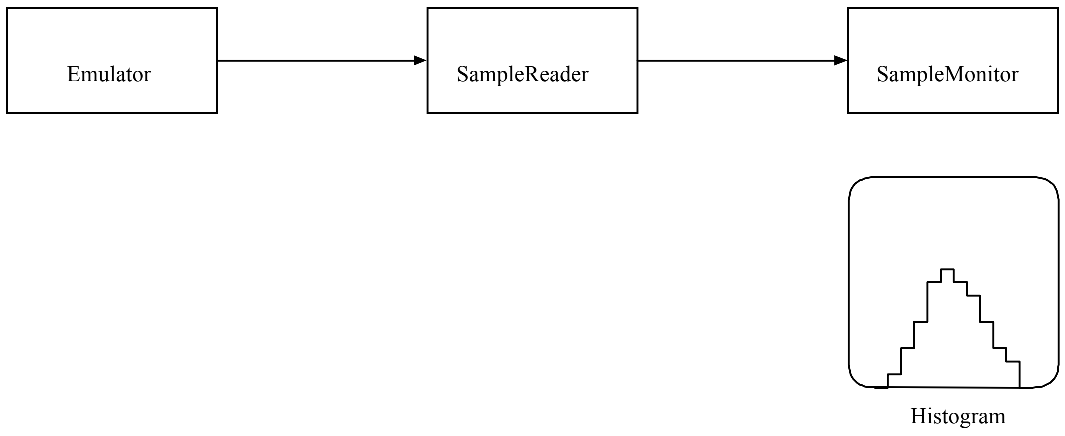


Fig. 6 The schematic outline of the data collection system developed in this document. The ‘SampleReader’ reads data from the ‘Emulator’ and then sends it to the subsequent ‘SampleMonitor’. The ‘SampleMonitor’ decodes the received data to draw a histogram displayed on the screen.

8 Outline of data collection system developed in this document

A schematic outline of the data collection system developed in this document is shown in Fig. 6. The ‘SampleReader’ reads the data from ‘Emulator’ and then send it to the subsequent ‘SampleMonitor’. The ‘SampleMonitor’ decodes the received data to draw a histogram displayed on the screen. This is a simple data collection system. For the ‘Emulator’, we use a software emulator.

9 Software emulator

9.1 Set-up

With DAQ-Middleware 1.2.0 or later, no particular installation process is needed for the software emulator, since it is already installed as ‘/usr/bin/daqmw-emulator’[*4]. With DAQ-Middleware earlier than 1.2.0, it can be downloaded from the URL shown below.

```
http://daqmw.kek.jp/src/daqmw-emulator.tar.gz
```

After the download, expand the file under the directory ‘/home/daq/MyDaq/’.

```
% cd
% cd MyDaq
% lftpget    http://daqmw.kek.jp/src/daqmw-emulator.tar.gz
% tar xf daqmw-emulator.tar.gz
% ls
daqmw-emulator
```

[*4] The source files are placed under ‘/usr/share/daqmw/daqmw-emulator/’.

```
% cd daqmw-emulator
% make
% cp emulator ~/bin/daqmw-emulator
```

9.2 Launch

When executing the following from the command line, the system waits for a connection at port 2222.

```
% daqmw-emulator
```

Right after the connection is established, it starts sending data. Without any options to the above command line, it sends the data at approx. 8kB/s. To change the transfer rate, specify it with '-t' option as follows.

```
% daqmw-emulator -t 128k
```

This sets the data transfer rate to 128kB/s. Specifying the option '-t 1M' will make it 1MB/s. Avoid too large a value for this, since it puts an extra computation load, particularly when using a VMware Player.

To stop, press 'Ctrl-C' as usual.

9.3 Data format of software emulator

The data format of the software emulator used in this document is shown in Fig. 7. It uses 8 bytes for sending 1 event data. First, it comes with a signature (magic) 0x5a. The system checks this byte to said value on decoding to confirm if it is reading the correct position. Next comes a data format version (0x01). A module number comes after the format version. In this software emulator, the value '0x0~0x7' is used therein. The subsequent 1 byte is reserved for a future expansion. The last 4 bytes have event data (integer value). This event data has the value of the number 0.000 to 1000.000 multiplied by 1000, then rounded off to the integer. When sending multiple bytes of any meaningful numerical value, a protocol for a byte order needs to be determined. This software emulator uses the network byte order to send data. To convert it to the host byte order on the reader side, the 'ntohl()' function is used.

9.4 Confirmation of data from emulator

Now, confirm what data is coming from the emulator. It is easier to use the 'nc' command to do so. Execute the following commands.

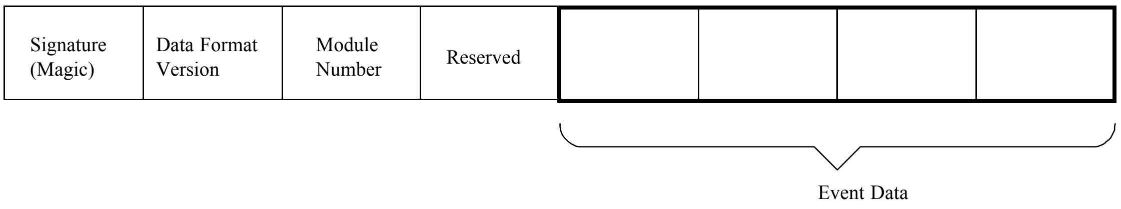


Fig. 7 The format of data coming in from the software emulator. It uses 8 bytes to send 1 event data. The event data takes the value of 0.000 to 1000.000 for its physical quantity and the emulator sends the value multiplied by 1000, then rounded off to 4 byte-integer. Its byte order is the network byte order. From bytes 0 to 3 are meta data. The magic is fixed to 0x5a and the data format version is fixed to 0x01. The module number comes in with 0x01 to 0x07 although this manual does not use it.

These commands need to be input on 1 line but not divided into multiple lines[*5].

```
(sleep 10; pkill -f /usr/bin/nc) & /usr/bin/nc 127.0.0.1 2222 > data.out
```

The ‘nc’ command then connects to port 2222 at 127.0.0.1. The read data is saved in the ‘data.out’ file. The reading time is seconds specified with ‘sleep’, which, in this case, is 10 seconds. For more details about the data format, see the previous Section. Decoding accordingly (for example, writing a program such that it reads 8 bytes and picks up 4th and 8th byte as ‘int’, then, converting it to host byte order with ‘ntohl()’ and divide it by 1000.0.) to draw a histogram, it will have peaks at 100, 200, 300,…,800 as in Fig. 8. The objective in this manual is to build a system to display such Fig.s, updated regularly.

Here, we explain the details of the emulator data. In fig. 8, the data around the peak at 100 have all their module numbers at 0. Those for 200 have all their module numbers at 1. Those for 800 have all their module numbers at 7. For Fig. 8, the module numbers have been ignored and the data from all the modules overlap. This document does not use the module number sent from the emulator.

10 ‘ SampleReader’ component development

The codes for the ‘SampleReader’ and ‘SampleMonitor’ described below are placed in the ‘SampleReader’ and ‘SampleMonitor’ directories under ‘/usr/share/daqmw/examples/’. The modifications from the template file generated by ‘newcomp’ can be checked using, for example, the ‘diff’ command as below.

[*5] Simply using ‘pkill nc’ sends a signal to all the processes having the string ‘nc’, not just the ‘nc’ process, and then those unrelated processes ‘exit’. So, here, a full path is specified with ‘nc’.

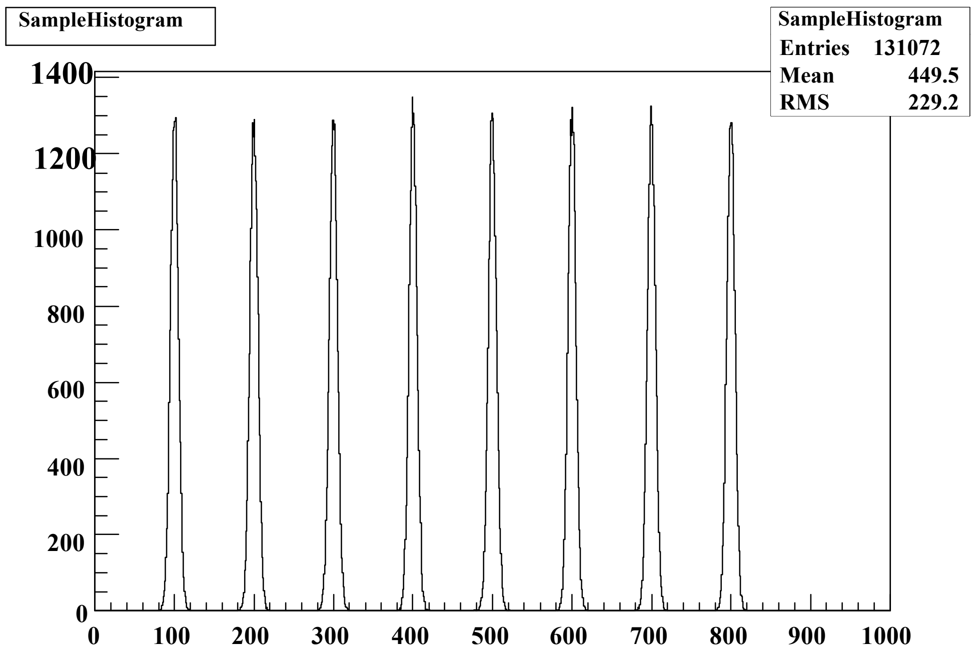


Fig. 8 Histogram of the data from software emulator used in this manual

```
% mkdir diff-test
% cd diff-test
% newcomp -t source SampleReader
% ls
SampleReader
% newcomp -t sink SampleMonitor
% ls
SampleMonitor    SampleReader
% mv SampleReader SK-SampleReader
% mv SampleMonitor SK-SampleMonitor
% cp -r /usr/share/daqmw/examples/SampleReader .
% cp -r /usr/share/daqmw/examples/SampleMonitor .
% diff -urNp SK-SampleReader SampleReader        | less
% diff -urNp SK-SampleMonitor SampleMonitor |less
```

With ‘-p’ option of ‘diff’ command, the name of function for which the modification has been done is displayed at the same line @@ to indicate that there was a modification as follows. This is useful in identifying the modified part.

```
@@ -85,6 +87,9 @@ int SampleReader::daq_configure()
 (Modifications come hereafter)
```

Now, we develop a component in reality. Here, we create a component that constitutes the DAQ system described in Section 8. In this Section, we create the ‘SampleReader’ component that reads data from the emulator and then sends it to subsequent components.

First, we consider the specification of the data-reading part of ‘SampleReader’ component. It is made it as follows:

- For the socket part, use the ‘Sock’ library attached to the DAQ-Middleware.
- Consider it as a fatal error occurrence with a connection failure.
- Prepare 1024 bytes for a buffer to read from the socket.
- Always read 1024 bytes at a time.
- Consider it as a fatal error occurrence when unable to read 1024 bytes within two seconds.
- Specify IP address and port number of the emulator, in the configuration file.
- When, in ‘daq_run()’, unable to send data to the subsequent component, the next ‘daq_run()’ will not newly read data but resend the data it could not send.

The include file of the ‘Sock’ library attached to the DAQ-Middleware is ‘/usr/include/daqmw/Sock.h’, and the library file is ‘/usr/lib/daqmw/libSock.so’.

Once the specification is determined, move on to the implementation work. First, create a template by executing the ‘newcomp -t source SampleReader’ to specify the ‘Source’-type component. Also, move to the generated directory (Here, ‘SampleReader’) and then ‘make’, to confirm if the development environment is working correctly.

```
% newcomp -t source SampleReader          (Generating template files)
% cd  SampleReader                        (Move to ‘SampleReader’ directory created)
% ls                                      (Check the generated files)
 Makefile    SampleReader.cpp     SampleReader.h      SampleReaderComp.cpp
 %  make                                  (Confirm the development environment)
rm -fr autogen                            (If it is normal, an executable file
mkdir  autogen                              ‘SampleReaderComp’ is created.)
(omitted…)
 %  ls                                    (Confirm the created files)
Makefile            SampleReader.h         SampleReaderComp*        SampleReaderComp.o
SampleReader.cpp    SampleReader.o         SampleReaderComp.cpp     autogen/
%  make clean                             (Delete the object file, executable file
                                            and automatically generated file (under ‘autogen’ directory
                                          ))
%  ls
Makefile     SampleReader.cpp     SampleReader.h      SampleReaderComp.cpp
```

10.1 Modification to ‘SampleReader.h’

We modify ‘SampleReader.h’ as follows.

10.1.1 Use of ‘Sock’ library

First, have ‘Sock’ library ready for use.

```
1  #include "DaqComponentBase.h"
2
3  #include <daqmw/Sock.h> // Addition
4
5   using namespace RTC;
```

The modification here is the addition of ‘#include’ on Line 4. This is to make ‘Sock’ library attached to DAQ-Middleware ready for use.

10.1.2 Addition of member variables etc.

Modify member variables and constants.

```
1          int set_data(unsigned int data_byte_size);
2          int write_OutPort();
3
4          DAQMW::Sock*  m_sock;                          ///  Addition of socket  for  data  server
5
6          static const int EVENT_BYTE_SIZE    =  8;               // Addition of event byte size
7          static const int SEND_BUFFER_SIZE = 1024;           // Change
8          unsigned char m_data[SEND_BUFFER_SIZE];
9          unsigned int         m_recv_byte_size;                      //Addition
10
11         BufferStatus  m_out_status;
12
13         int  m_srcPort;                                        // Addition of  Port No.  of data server
14         std::string  m_srcAddr;                                // Addition of  IP addr. of data server
```

The modifications are as written in the comments.

- (Line 4) Addition of the ‘Sock’ object
- (Line 6) It is defined that 1 event data from the emulator has 8bytes.
- (Line 7) It is defined that one data-reading handles 1024 bytes as mentioned above.
- (Line 13) The variable to designate IP port number of the emulator. The port number is obtained from the configuration file.
- (Line 14) The IP address variable of the emulator. The IP address is obtained from the configuration file.

Also, on Line 9, the variable ‘m_recv_byte_size’ is added to the member variables. This is the variable to implement the condition determined in the above specification that “in ‘daq_run()’, when unable to send data to the subsequent component, the next ‘daq_run()’ will not newly read data from the emulator but will resend data it could not send”. For data-resending, it needs to memorize how many bytes of data ‘daq_run()’ has read from the emulator. This variable is used for that purpose. This time, we defined it to always read 1024 bytes for future expansions.

10.2 Modification to ‘SampleReader.cpp’

Next, we move on to the modification of ‘SampleReader.cpp’. Here, we describe for each function in the state chart (fig.3).

■Constructor

```
SampleReader::SampleReader(RTC::Manager*   manager)
      : DAQMW::DaqComponentBase(manager),
        m_OutPort("samplereader_out",  m_out_data),
        m_sock(0), // Addition
        m_recv_byte_size(0),
        m_out_status(BUF_SUCCESS),

        m_debug(false)
{
      // Registration: InPort/OutPort/Service

      // Set OutPort buffers
      registerOutPort("samplereader_out",  m_OutPort);

      init_command_port();
      init_state_table();
      set_comp_name("SAMPLEREADER");
}
```

This initializes the data-reading socket object (‘m_sock’) to 0. With this value 0, it can be judged if the socket object should be ‘delete’-d. See, ‘daq_stop()’ mentioned below.

■fatal_type

```
using  DAQMW::FatalType::DATAPATH_DISCONNECTED;
using DAQMW::FatalType::OUTPORT_ERROR;
using DAQMW::FatalType::USER_DEFINED_ERROR1;
using DAQMW::FatalType::USER_DEFINED_ERROR2;
```

The ‘using’ declaration omits the namespace names in the argument of ‘fatal_error_report()’. For the ‘fatal_error_report()’, see the “DAQ-Middleware 1.1.0 Technical Manual” [2].

■daq_configure()

```
int SampleReader::daq_configure()
{
      std::cerr << "*** SampleReader::configure" << std::endl;

      ::NVList* paramList;
      paramList  =  m_daq_service0.getCompParams();
      parse_params(paramList);
```

```

    return  0;
}
```

The above is the ‘daq_configure()’ generated by the ‘newcomp -t source SampleReader’. The implementation this time does not include any modifications. At Line 6, ‘getCompParams()’ obtains the parameters specified in the configuration file. Subsequently, we move on to the modification of ‘parse_params()’ which analyses the obtained parameters and sets up the variables.

The ‘parse_params()’ is modified so that the value acquired from the configuration file can be set to ‘m_srcAddr’ and ‘m_srcPort’ variables.

```
int SampleReader::parse_params(::NVList* list)
{
    bool srcAddrSpecified = false; // Addition
    bool srcPortSpecified = false; // Addition

    std::cerr << "param list length:" << (*list).length() << std::endl;

    int  len  =  (*list).length();
    for (int i = 0; i < len; i+=2) {
        std::string sname       =     (std::string)(*list)[i].value;
        std::string   svalue   =   (std::string)(*list)[i+1].value;

        std::cerr << "sname: " << sname << "                  ";
        std::cerr << "value: " << svalue << std::endl;

        // Addition (Setting m_srcAddr  and  m_srcPort)
        if ( sname == "srcAddr" ) {
            srcAddrSpecified  =  true;
            if (m_debug) {
                std::cerr << "source addr: " << svalue << std::endl;
            }
            m_srcAddr  =  svalue;
        }
        if ( sname == "srcPort" ) {
            srcPortSpecified  =  true;
            if (m_debug) {
                std::cerr << "source port: " << svalue << std::endl;
            }
            char* offset;
            m_srcPort  =  (int)strtol(svalue.c_str(),  &offset,  10);
        }

    }
    // Addition (A fatal error if ‘srcAddr’ and ‘ srcPort’ have not been obtained.)
    //
    if (!srcAddrSpecified) {
        std::cerr << "### ERROR:data source address not specified\n";
        fatal_error_report(USER_DEFINED_ERROR1,   "NO   SRC   ADDRESS");
    }
    if (!srcPortSpecified) {
        std::cerr << "### ERROR:data source port not specified\n";
        fatal_error_report(USER_DEFINED_ERROR2,   "NO   SRC    PORT");
    }

45         return 0;
46  }
```

Add ‘bool’ variables at Line 3 and 4 for recording if the acquisitions of ‘m_srcAddr’ and ‘m_srcPort’ variables to be added were a success or failure.

Since ‘list’ variables have the parameters specified in the configuration file in the order of parameter name, value, parameter name, value,……, the necessary parameter name is linear-searched. If found, set the value to the variable. The code at Line 17 does this. Since the value is string, ‘strtol()’ converts it to a numerical value. Also, the codes from Line 36 judge a fatal error occurrence in case the ‘m_srcAddr’ and ‘m_srcPort’ could not be acquired, and notifies the ‘DaqOperator’ of the error occurrence by ‘fatal_error_report()’. Now, the modification of the ‘daq_configure()’ is completed.

■daq_start()

```
1   int SampleReader::daq_start()
2   {
3       std::cerr << "*** SampleReader::start" << std::endl;
4   
5       m_out_status = BUF_SUCCESS;
6   
7       // The followings are added.
8       try {
9           // Create socket and connect to data server.
10          m_sock = new DAQMW::Sock();
11          m_sock->connect(m_srcAddr, m_srcPort);
12      } catch (DAQMW::SockException& e) {
13          std::cerr << "Sock Fatal Error : " << e.what() << std::endl;
14          fatal_error_report(USER_DEFINED_ERROR1, "SOCKET FATAL ERROR");
15      } catch (...) {
16          std::cerr << "Sock Fatal Error : Unknown" << std::endl;
17          fatal_error_report(USER_DEFINED_ERROR1, "SOCKET FATAL ERROR");
18      }
19  
20      // Check data port connections
21      bool outport_conn = check_dataPort_connections( m_OutPort );
22      if (!outport_conn) {
23          std::cerr << "### NO Connection" << std::endl;
24          fatal_error_report(DATAPATH_DISCONNECTED);
25      }
26  
27      return 0;
28  }
```

Here, the codes from Line 9 are added. The added contents are to generate the ‘Sock’ object (line 10) and then connect to the server (emulator this time) designated in ‘m_srcAddr’ and ‘m_srcPort’. We added the codes for the specification where a fatal failure is considered to occur with a connection failure (from Line 12). The ‘check_dataPort_connection()’ is the function to confirm the connection to the subsequent component. The connection failure generates a fatal error (from Line 21).

■daq_run()

```
 1 int SampleReader::daq_run()
 2 {
 3     if(m_debug) {
 4         std::cerr << "*** SampleReader::run" << std::endl;
 5     }
 6
 7     if (check_trans_lock()) {          // check if stop command has come
 8         set_trans_unlock();            // transit to CONFIGURED state
 9         return 0;
10     }
11
12     if(m_out_status == BUF_SUCCESS) {          // previous OutPort.write() successfully done
13         int ret = read_data_from_detectors();
14         if (ret > 0) {
15             m_recv_byte_size = ret;
16             unsigned long sequence_num = get_sequence_num();
17             set_data(m_recv_byte_size); // set data to OutPort Buffer
18         }
19     }
20
21     if (write_OutPort() < 0) {
22         ;       // Timeout. do nothing.
23     }
24     else {      // OutPort write successfully done
25         inc_sequence_num();                          // increase sequence num.
26         inc_total_data_size(m_recv_byte_size);       // increase total data byte size
27     }
28
29     return 0;
30 }
```

There is no modification in the ‘daq_run()’ from that generated by the ‘newcomp’. Below, the above codes are described.

(Line 7 to 10) This part checks if the ‘stop’ command has come in. The ‘check_trans_lock()’ returns the ‘true’ with the ‘stop’ command. In this case, the system transitions to the ‘CONFIGURED’ state by calling the ‘set_trans_lock()’ and then terminates the ‘daq_run()’. If the ‘stop’ command has not yet come in, it reads the data (however, we code that, when the component could not send the data to the subsequent component in the previous ‘daq_run()’, it will resend the previously read data without reading new data from the emulator). For the meanings and details of ‘check_trans_lock()’ and ‘set_trans_lock()’, see the “DAQ-Middleware 1.1.0 Technical Manual” [2].

(Line 12 to 19) The ‘if’ checks the correct transmission of the data to the subsequent component in the previous ‘daq_run()’ (When ‘m_out_status’ is ‘BUF_SUCCESS’, the data has been correctly transmitted). If the data has been sent to the subsequent component in the previous ‘daq_run()’, it reads the data from the emulator. Code ‘read_data_from_detector()’ so that the data read out enter the ‘m_data’ array defined in the ‘SampleReader.h’. We will mention this function later. Once data is successfully read, the ‘read_data_from_detector()’ returns the number of bytes that have been read (‘read_data_from_detector()’ will be coded to do so). So, Line 13 checks its return value. The ‘set_data()’ at Line 16 is the function to set the header, footer and data transferred between components to the ‘OutPort’ buffer. This implementation will be described later.

(Line 20) The ‘write_OutPort()’ is the function to send the data to the subsequent component and the implementation of which will be described later. When the data is successfully sent to the subsequent component, the ‘inc_sequence_num()’ increments the sequence number retained in this component, by 1. Also, the ‘inc_total_data_size()’ increments the number of the event data bytes handled previously that is retained in this component. For the data including the sequence number retained by the ‘inc_sequence_num()’, ‘inc_total_data_size()’ and the component, see the “DAQ-Middleware 1.1.0 Technical Manual” [2].

■read data from datectors()

```
int  SampleReader::read_data_from_detectors()
{
      int received_data_size = 0;

      /// write your logic here
      /// read 1024 byte data from data server
      int status = m_sock->readAll(m_data, SEND_BUFFER_SIZE);
      if(status == DAQMW::Sock::ERROR_FATAL) {
            std::cerr << "### ERROR: m_sock->readAll" << std::endl;
            fatal_error_report(USER_DEFINED_ERROR1,    "SOCKET    FATAL    ERROR");
      }
      else if(status == DAQMW::Sock::ERROR_TIMEOUT) {
            std::cerr << "### Timeout: m_sock->readAll" << std::endl;
            fatal_error_report(USER_DEFINED_ERROR2,   "SOCKET   TIMEOUT");
      }
      else {
            received_data_size = SEND_BUFFER_SIZE;
      }

      return received_data_size;
}
```

As mentioned above, the ‘read_data_from_detectors()’ reads the data from the emulator. Its specification is as follows.

- The return value is the number of bytes read out.
- The data read out enter the array represented by the ‘m_data’ member variable.
- Data-reading failure generates a fatal error.
- Time-out in reading-out (the default setting of ‘Sock’ library, 2 seconds is employed) generates a fatal error.

The socket-related data-reading cannot always read out the specified numbers of bytes (for example, it could be the case that 100 bytes were specified for reading, and only 50 bytes of the data has arrived.). If you want to ensure the reading of the exactly specified bytes, you need to code such functions manually. Since this type of function is commonly used, the DAQ-Middleware provides the ‘readAll()’ function in the ‘Sock’ library. The ‘readAll()’ has two arguments. The first one is specified with an array wherein the successfully read-out data are stored. The second one specifies the number of bytes to read out. The return value of the ‘readAll()’ is as follows.

- Returns ‘DAQMW::Sock::ERROR_FATAL’ on a read-out error.
- Returns ‘DAQMW::Sock::ERROR_TIMEOUT’ on read-out time-out (2 seconds by default).
- Returns the number of bytes read out on a correct reading.

Line 7 uses the ‘readAll()’ and Line 8 to 18 check a fatal error occurrence with reference to its return value. When reading correctly without a fatal error, Line 20 returns the number of bytes read out according to the specification of the above-mentioned ‘read_data_from_detectors()’.

■set data()

```
int SampleReader::set_data(unsigned int data_byte_size)
{
    unsigned char header[8];
    unsigned char footer[8];

    set_header(&header[0], data_byte_size);
    set_footer(&footer[0]);

    ///set OutPort buffer length
    m_out_data.data.length(data_byte_size + HEADER_BYTE_SIZE + FOOTER_BYTE_SIZE);
    memcpy(&(m_out_data.data[0]), &header[0], HEADER_BYTE_SIZE);
    memcpy(&(m_out_data.data[HEADER_BYTE_SIZE]), &m_data[0], data_byte_size);
    memcpy(&(m_out_data.data[HEADER_BYTE_SIZE + data_byte_size]), &footer[0],
            FOOTER_BYTE_SIZE);

    return 0;
}
```

For ‘set_data()’, there is no modification from the template generated by ‘newcomp’ command. ‘set_data()’ adds component header and footer to the data to be sent to the subsequent component[*6].

[*6] The format of the data transferred between components is as already shown in Fig. 4. Also, the header and footer formats for the figure are as shown in Fig. 5. ‘DataByteSize’ and ‘sequence number’ in the header and footer formats are used in ‘check_header_footer()’by the subsequent component to verify if there is no missings in the read-out data when receiving them from the upstream component.

First, secure ‘header’ and ‘footer’ arrays. Next, using ‘set_header()’ sets the ‘DataByteSize’ from 32nd to 63rd bit of the header. Also, the ‘set_footer()’ sets the ‘sequence number’. The ‘sequence number’ does not need to be explicitly specified because it uses the value of the private variable ‘m_loop’. So, the ‘set_footer()’ only has a pointer to the head of the ‘footer’ buffer as its argument. The ‘m_out_data.data.length()’ on Line 10 specifies the number of bytes to be written in the ‘OutPort’ to secure the ‘m_out_data.data’ buffer. Subsequently, it copies the header, footer and data created above to this buffer using ‘memcpy()’ (From Line 11 to 14).

■write OutPort()

```
1  int  SampleReader::write_OutPort()
2  {
3        ////////////////// send  data  from  OutPort    //////////////////
4        bool  ret  =  m_OutPort.write();
5
6        ////////////////////  check   write  status  /////////////////////
7        if  (ret  ==  false)  {            // TIMEOUT  or FATAL
8              m_out_status        =     check_outPort_status(m_OutPort);
9              if(m_out_status == BUF_FATAL) {              //  Fatal  error
10                   fatal_error_report(OUTPORT_ERROR);
11             }
12             if(m_out_status == BUF_TIMEOUT) { //Timeout
13                   return  -1;
14             }
15       }
16       m_out_status  =  BUF_SUCCESS;
17       return  0;  //  successfully  done
18 }
```

The ‘write()’ method of the ‘m_OutPort’ writes data in the out-port. The data to be written in is stored in the ‘m_out_data.data’ array in the ‘set_data()’. When ‘false’ returned, the ‘check_outPort_status(m_OutPort)’ checks the state of the out-port. With a fatal error, ‘fatal_error_report()’ notifies a fatal error occurrence to the ‘DaqOperator’. When there is a time-out, it returns -1 and notifies the time-out to the caller. If the ‘write()’ worked successfully, ‘BUF_SUCCESS’ is set to the ‘m_out_status’ and the ‘write_OutPort()’ terminates.

■daq stop()

```
int SampleReader::daq_stop()
{
      std::cerr <<  "***  SampleReader::stop"  <<  std::endl;

      if (m_sock) {                          // Addition
            m_sock->disconnect();            // Addition
            delete m_sock;                   // Addition
            m_sock = 0;                      // Addition
      }                                      // Addition

      return 0;
}
```

Here, the ‘disconnect()’ method of the ‘Sock’ library cuts off the connection from the emulator. Since, in some systems, you may need to program to move on to the ‘daq_stop()’ before creating the socket for data-reading, it refers to the value of the ‘m_sock’ to judge if the socket has been created[*7] (Although this works in the components other than the ‘SampleReader’, it is designed as above so that it can be modified from this example in the future).

■daq pause() and _daq resume() No modifications to the ‘daq_pause()’ and ‘daq_resume()’.

10.3 Modification to Makefile

Since the ‘SampleReader’ is designed to use the ‘Sock’ library, you need to specify the position of the library file in the ‘Makefile’. Modify the ‘Makefile’ as follows.

```
SRCS += $(COMP_NAME).cpp
SRCS  +=  $(COMP_NAME)Comp.cpp

# The following lines are added.
LDLIBS  +=  -L/usr/lib/daqmw  -lSock
```

The include file of the ‘Sock’ library is in ‘/usr/include/daqmw/Sock.h’. Since this is not the original standard directory, it requires additional measures including the addition of ‘-I/usr/include/daqmw’ to ‘CPPFLAGS’. However, this is not necessary because the addition has already been done in ‘/usr/share/ daqmw/comp.mk’ that is ‘include’-d at the last part in the ‘Makefile’. Also, when it needs to read the include files placed in an non-standard directory other than the ‘/usr/include/daqmw’, such as, for example, ‘ROOT’ referring to ‘/usr/local/root/include’, you should notice the need to script ‘Makefile’ accordingly (as with ‘SampleMonitor’ in the next Section).

Then, if ‘make’ still generates an error, take necessary the measures with reference to the error message.

11 SampleMonitor component development

Next, we move on to the development of the ‘SampleMonitor’ component to receive the data from the ‘SampleReader’ to display a histogram on the screen. Here, we use ‘ROOT’[*8] for the histogram drawing tool.

As in the ‘SampleReader’, the ‘newcomp’ command generates template files. Since this is a development of the ‘Sink’-type component, specify ‘-t sink’.

```
% cd                                (move on to home directory)
% mkdir  MyDaq                       (if there is no development directory, it must be created here.
                                         The directory name does not need to be this)
```

*7 We consider the case where the component needs to cast ‘SiTCP slow control’ packet before starting the data-read, for example. It is possible to program a code that is written before the creation of the data-reading socket, and when ‘slow control’ fails, ‘fatal_error_report()’ is executed before the socket creation.

*8 http://root.cern.ch/

```
% cd MyDaq
% newcomp -t sink SampleMonitor (With '-t sink' specified, execute ' newcomp')
%  cd  SampleMonitor
% ls                                  (Confirm the generated files)
Makefile    SampleMonitor.cpp      SampleMonitor.h      SampleMonitorComp.cpp
% make                                (try 'make' to confirm if the development environment is ok.)
rm -fr autogen
mkdir autogen
(omitted…)
% ls
Makefile              SampleMonitor.h     SampleMonitorComp*       SampleMonitorComp.o
SampleMonitor.cpp     SampleMonitor.o     SampleMonitorComp.cpp    autogen
% make clean
```

Here, the specification of a histogram where the minimum value is set to 0, its maximum is 1000, and the number of bins is set to 100.

This component works roughly as follows.

1. Read data from the upstream component.
2. Confirm that there is nothing missing in data-reading, with reference to the component header and footer.
3. Decode the multiple event data.
4. Increment the event data to the 'ROOT' histogram data.
5. Update the histogram if its update condition is met.
6. Repeat the above, hereunder.

This 'SampleMonitor' component is designed such that 'get_sequence_num()' acquires the number of times 'daq_run()' is repeated as the update condition for the histogram, and then updates at the determined repetitions.

11.1 Creation of 'SampleData.h'

Since the data format from this emulator is relatively simple, it may be possible to handle the data structure without defining it as 'structure'. However, for future expansions[*9], the 'structure' of the event data format is defined here. Here, a file named 'SampleData.h' is newly created and defined as follows. For the event data format from the emulator, see Fig. 7 in Section 9.3.

```
#ifndef SAMPLEDATA_H
#define SAMPLEDATA_H

const int ONE_EVENT_SIZE = 8;

struct sampleData {
   unsigned char magic;
   unsigned char format_ver;
```

[*9] And for other people (including yourself 2 months later)

```
    unsigned char module_num;
    unsigned char reserved;
    unsigned  int         data;
};

#endif
```

11.2 Modification to ‘SampleMonitor.h’

We modify ‘SampleMonitor.h’ as follows.

■Include file

```
1  #include <arpa/inet.h> // Addition for ntohl()
2
3  ////////// ROOT Include files //////////
4  #include "TH1.h"                      // Addition
5  #include "TCanvas.h"                  // Addition
6  #include "TStyle.h"                   // Addition
7  #include "TApplication.h"             // Addition
8
9   #include "SampleData.h"              // Addition
```

‘<arpa/inet.h>’ on Line 1 is for the ‘ntol()’ function. Line 3 to 7 are the include files for ‘ROOT’ used to create the histogram. In ‘SampleData.h’ on Line 9, the data format from the emulator is defined as ‘structure’ as seen in the previous Section.

■Variables and methods

```
1   int decode_data(const unsigned char* mydata);              // Addition
2   int fill_data(const unsigned char* mydata, const int size); // Addition
3
4   BufferStatus  m_in_status;
5
6    ////////// ROOT  Histogram  //////////
7   TCanvas  *m_canvas;                          // Addition
8   TH1F       *m_hist;                          //  Addition
9   int              m_bin;                      // Addition
10  double       m_min;                          // Addition
11  double       m_max;                          //  Addition
12  int              m_monitor_update_rate;      //Addition
13  unsigned  char  m_recv_data[4096];           // Addition
14  unsigned int         m_event_byte_size;      //Addition
15  struct sampleData m_sampleData;              // Addition
```

The ‘decode_data()’ on Line 1 is the method to decode the data. The ‘fill_data()’ on Line 2 is the method to fill data in the ‘ROOT’ histogram data. Line 7 to 11 are the variables for the histogram. The histogram is drawn on the ‘m_canvas’ defined on Line 7. In this monitor, we defined that the update timing of the histogram is based on how many times ‘daq_run()’ has run. Concretely, the histogram is updated when ‘daq_run()’ is repeated for the times specified with the ‘m_monitor_update_rate’ variable on Line 12. The ‘m_recv_data’ on Line 13 is the buffer to store the data sent from the upstream component having their component header and footer excluded. The 'm_event_byte_size' on Line 14 is the variable to retain the number of bytes read in one data-reading from the upstream component[*10]. The 'm_sampleData' on Line 15 is 'structure' for which the data format from the emulator is defined. The decoded data enter here, and the value of this variable is used to increment them to the histogram data.

11.3 Modification to 'SampleMonitor.cpp'

■Initialization of variables

```
SampleMonitor::SampleMonitor(RTC::Manager*     manager)
      :  DAQMW::DaqComponentBase(manager),
         m_InPort("samplemonitor_in",                m_in_data),
         m_in_status(BUF_SUCCESS),
         m_canvas(0),                                // Addition
         m_hist(0),                                  // Addition
         m_bin(0),                                   // Addition
         m_min(0),                                   // Addition
         m_max(0),                                   // Addition
         m_monitor_update_rate(30),                  //Addition
         m_event_byte_size(0),                       //Addition
         m_debug(false)
```

Here, initializing the variables used.

■daq_dummy()

```
int SampleMonitor::daq_dummy()
{
      if (m_canvas) {                       // Addition
            m_canvas->Update();             // Addition
            // daq_dummy() will be invoked again after 10 msec.
            // This sleep reduces X servers' load.
            sleep(1);                       // Addition
      }                                     // Addition

      return 0;
}
```

It is designed to draw a histogram regularly even in the 'CONFIGURED' state since, after a transition to the 'CONFIGURED' state (after the 'stop' command is issued), the histogram remains unseen when any 'window' other than 'ROOT' is moved over the 'canvas' of 'ROOT' and then moved back. Then redraw the histogram that had already been drawn, but not update it.

[*10] The number of bytes readable here is always 1024 bytes since the 'SampleReader' sends 1024 bytes. We prepared the variable for future expansions.

■daq unconfigure()

```
int SampleMonitor::daq_unconfigure()
{
      std::cerr << "*** SampleMonitor::unconfigure" << std::endl;
      if (m_canvas) {                 // Addition
            delete m_canvas;          // Addition
            m_canvas = 0;             // Addition
      }                               // Addition

      if (m_hist) {                   // Addition
            delete m_hist;            // Addition
            m_hist = 0;               // Addition
      }                               // Addition
      return 0;
}
```

Here, the 'canvas' and the data used to draw the histogram are 'delete'-d.

■daq start()

```
1  //////////////// CANVAS FOR HISTOS //////////////////
2  if (m_canvas) {                                                  // Addition
3        delete m_canvas;                                           // Addition
4        m_canvas = 0;                                              // Addition
5  }                                                                // Addition
6  m_canvas = new TCanvas("c1", "histos", 0, 0, 600, 400); // Addition
7
8  ///////////////                  HISTOS          //////////////////
9   if (m_hist) {                                                   // Addition
10       delete m_hist;                                             // Addition
11       m_hist = 0;                                                // Addition
12 }                                                                // Addition
13
14 int m_hist_bin = 100;                                            // Addition
15 double m_hist_min = 0.0;                                         // Addition
16 double m_hist_max = 1000.0;                                      // Addition
17
18 gStyle->SetStatW(0.4);                                           //Addition
19 gStyle->SetStatH(0.2);                                           //Addition
20 gStyle->SetOptStat("em");                                        //Addition
21
22 m_hist = new TH1F("hist", "hist", m_hist_bin, m_hist_min, m_hist_max); // Addition
23 m_hist->GetXaxis()->SetNdivisions(5);                            //Addition
24 m_hist->GetYaxis()->SetNdivisions(4);                            //Addition
25 m_hist->GetXaxis()->SetLabelSize(0.07);                          //Addition
26 m_hist->GetYaxis()->SetLabelSize(0.06);                          //Addition
```

Here, the histogram variables 'm_canvas' and 'm_hist' are set.

The value settings of both 'm_canvas' and 'm_hist' are ensured for 'start' executed once more after a 'stop'. If the values are set, they will be 'delete'-d and initialized with 0, and then 'new'-ed[*11]. Line 14 to 16 specify the parameter of the histogram. On Line 18 to 26, 'ROOT' command specifies the parameters of the histogram. For the details of this, see the manual of 'ROOT'.

■daq run()

```
1  int SampleMonitor::daq_run()
2  {
3          if (m_debug) {
4                  std::cerr << "*** SampleMonitor::run" << std::endl;
5          }
6
7          unsigned int recv_byte_size = read_InPort();
8          if (recv_byte_size == 0) { // Timeout
9                  return 0;
10         }
11
12         check_header_footer(m_in_data, recv_byte_size); // check header and footer
13         m_event_byte_size = get_event_size(recv_byte_size); // Change
14
15         /////////////         Write component main logic here. /////////////
16         memcpy(&m_recv_data[0], &m_in_data.data[HEADER_BYTE_SIZE], m_event_byte_size); // Addition
17
18         fill_data(&m_recv_data[0], m_event_byte_size);                  // Addition
19
20         if (m_monitor_update_rate == 0) {                               // Addition
21                 m_monitor_update_rate = 1000;                           // Addition
22         }                                                               // Addition
23
24         unsigned long sequence_num = get_sequence_num();                // Addition
25         if ((sequence_num % m_monitor_update_rate) == 0) {              // Addition
26                 m_hist->Draw();                                         // Addition
27                 m_canvas->Update();                                     // Addition
28         }                                                               // Addition
29         /////////////////////////////////////////////////////////
30         inc_sequence_num();                             // increase sequence num.
31         inc_total_data_size(m_event_byte_size);         // name of variable changed; increase total data byte size
32
33         return 0;
34 }
```

(Line 7 to 10) The 'read_InPort()' mentioned below attempts to read out the data in 'InPort' (When the data is successfully read out, they enter the 'm_in_data.data' array).

The return value of 'read_InPort()' is implemented as follows.

- Return 0 with 'timeout'
- Return the number of bytes read on a successful data reading.

The number of data bytes on a successful data reading includes those for the component header and footer. In the scope of this manual, the 'read_InPort()' needs no modification from the template file generated by the 'newcomp -t sink'.

[*11] Also, 'daq_stop()' initializes the variables with 0 after 'delete'-ing them just in case.

(Line 12) When the data-reading is completed successfully, ‘check_header_footer()’ confirms that there are no contradictions in the sequence number. If ‘check_header_footer()’ found any abnormality, ‘fatal_error_report()’ notifies an error to ‘DaqOperator’ and the component transitions to an idle state.

(Line 13) ‘get_event_size()’ function acquires the number of bytes for the event data.

(Line 16) When ‘read_InPort()’ read the data successfully, ‘memcpy()’ copies only the event data row to be decoded, from those in the ‘m_in_data.data’ array.

(Line 18) ‘fill_data()’ function to fill in the histogram data, makes an increment.

The implementation of ‘fill_data()’ will be described later.

(Line 20 to 28) This monitor determines the timing to update the histogram based on the number of repetitions of ‘daq_run()’ that received the data. On Line 24, ‘get_sequence_num()’ gets the process’s own sequence number, and Line 25 updates the histogram for every number of times specified by ‘m_monitor_update_rate’ to draw it on the screen. For making a division, the ‘if’ block on Line 20 confirms that the ‘m_monitor_update_rate’ is not 0 just in case.

(Line 29 to 30) The codes increment the sequence number as well as the number of event bytes handled. For the meaning of the sequence number, ‘get_sequence_num()’ and ‘inc_total_data_size()’, see the “DAQ-Middleware 1.1.0 Technical Manual” [2].

■read InPort

```
unsigned int SampleMonitor::read_InPort()
{
    ////////////// read data from InPort Buffer //////////////
    unsigned int recv_byte_size = 0;
    bool ret = m_InPort.read();

    /////////////////// check read status ///////////////////
    if (ret == false) { // false: TIMEOUT or FATAL
        m_in_status = check_inPort_status(m_InPort);
        if (m_in_status == BUF_TIMEOUT) { // Buffer empty.
            if (check_trans_lock()) {         // Check if stop command has come.
                set_trans_unlock();           // Transit to CONFIGURE state.
            }
        }
        else if (m_in_status == BUF_FATAL) { // Fatal error
            fatal_error_report(INPORT_ERROR);
        }
    }
    else {
        recv_byte_size = m_in_data.data.length();
    }

    if (m_debug) {
        std::cerr << "m_in_data.data.length():" << recv_byte_size
                  << std::endl;
    }

28        return  recv_byte_size;
29  }
```

This function attempts to read out the data from the ‘InPort’. There is no modification to this function that remains as generated by ‘newcomp -t sink’. Below, the codes are described.

(Line 5) The ‘read()’ method attempts to read out the data from ‘InPort’. The read data enter the ‘m_in_data.data’ array. Subsequently, the result read by ‘read()’ is checked.

(Line 8 to 14) When ‘read()’ returns ‘false’, ‘check_inPort_status()’ checks the state of the ‘InPort’. When ‘BUF_TIMEOUT’ is returned, that means no data to read. In this case, ‘check_trans_lock()’ checks if the ‘STOP’ command has come in. If it has, the system transits back to ‘CONFIGURE’ state.

(Line 15 to 17) The ‘check_inPort_status()’ judges a fatal error occurrence with ‘BUF_FATAL’ returned, and then ‘fatal_error_report()’ casts an error to the ‘DaqOperator’. As a result of reading ‘fatal_error_report()’, the component itself transits to an idle state.

(Line 19 to 21) The ‘read()’ method returning ‘true’ means a successful data-reading. Then, the ‘m_in_data.data.length()’ method acquires the length of the data that were read (its unit is byte).
(Line 28) When there is no length for the read data, or no data even on the component’s normal operation, the code returns 0 and this function terminates.

■fill data()

```
// Whole this function is added.
 int SampleMonitor::fill_data(const unsigned char* mydata, const int size)
 {
        for (int i = 0; i < size/(int)ONE_EVENT_SIZE; i++) {
               decode_data(mydata);
               float fdata = m_sampleData.data/1000.0; // 1000 times value is received
               m_hist->Fill(fdata);

               mydata+=ONE_EVENT_SIZE;
        }
        return 0;
 }
```

This is the routine to fill the decoded data in the histogram. Its arguments are the pointer to the data byte row and the data byte row length. It scans the data buffer from its head for every 1 event size (in case of emulator, ‘ONE_EVENT_SIZE = 8 bytes) to get the event data[*12]. The extracted data are filled in the histogram data using ‘Fill()’ of ‘ROOT’.

[*12] When you use data ‘structure’ defined to fit the byte row to scan, you need to recognize the alignment problem in ‘structure’, although this method is not used here.

■decode data()

```
int SampleMonitor::decode_data(const unsigned char* mydata) // Whole this function is added.
{
      m_sampleData.magic          =   mydata[0];
      m_sampleData.format_ver   =   mydata[1];
      m_sampleData.module_num  =  mydata[2];
      m_sampleData.reserved        =   mydata[3];
      unsigned  int  netdata           = *(unsigned int*)&mydata[4];
      m_sampleData.data              =    ntohl(netdata);

      if(m_debug) {
            std::cerr  <<  "magic:  "         << std::hex << (int)m_sampleData.magic                <<   std::endl;
            std::cerr  <<  "format_ver:  "  <<  std::hex  <<  (int)m_sampleData.format_ver         <<  std::endl;
            std::cerr << "module_num:  " << std::hex << (int)m_sampleData.module_num       << std::endl;
            std::cerr  <<  "reserved:  "       << std::hex << (int)m_sampleData.reserved             <<   std::endl;
            std::cerr  <<  "data:  "             << std::dec << (int)m_sampleData.data                  <<   std::endl;
      }

      return 0;
}
```

The functions to decode data are bundled in the ‘decode_data’. Here, the decoded data enter the member variable ‘m_sampleData’.

■daq stop()

```
1   int  SampleMonitor::daq_stop()
2   {
3         std::cerr  <<  "***  SampleMonitor::stop"  <<  std::endl;
4
5         m_hist->Draw();               //Addition
6         m_canvas->Update();         //Addition
7
8         reset_InPort();
9
10        return 0;
11  }
```

The Line 5 and 6 are newly added. Their purpose is to redraw the histogram based on the data that has been incremented by the time ‘daq_stop()’ is called. Now, the number of ‘Entries’ in the histogram, and the event bytes / 8 (1 event bytes) displayed on the terminal screen by ‘DaqOperator,’ match on the ‘stop’display.

11.4 Modification to ‘SampleMonitorComp.cpp’

Although the ‘SampleReader’ component did not require any modifications to the ‘SampleReaderComp.cpp’ with ‘main()’ function in it, the ‘SampleMonitor’ does require a creation of ‘TApplication’ object in the ‘main()’ function since it uses ‘ROOT’ to draw the histogram. The ‘SampleMonitorComp.cpp’ is modified as follows.

```
int main (int argc, char** argv)
{
      RTC::Manager*  manager;
      manager  =  RTC::Manager::init(argc,  argv);

      // for root application
      TApplication theApp("App", &argc, argv);                    // Addition

      // Initialize manager
      manager->init(argc, argv);
```

11.5 Modification to Makefile

Since this component uses ‘ROOT’ to make the histogram, the positions of its include files and libraries need to be notified to the compiler. Rewrite ‘Makefile’ as follows. First, add the followings to the head of ‘Makefile’

```
ifndef ROOTSYS
$(error This program requires ROOTSYS environment variable\
but does not defined.          Please define ROOTSYS as follows at\
shell prompt: "export ROOTSYS=/usr/local/root".               If you don’t install\
ROOT in /usr/local/root, please substitute your ROOT root directory)
endif
```

This is because the ‘ROOTSYS’ environment variable is used when the ‘ROOT’ utility program ‘root-config’ is called where ‘CPPFLAGS’ and ‘LDLIBS’ variables are set as below.

The following lines are added for ‘CPPFLAGS’ and ‘LDLIBS’.

```
CPPFLAGS += -I$(shell ${ROOTSYS}/bin/root-config --incdir)
LDLIBS     += $(shell ${ROOTSYS}/bin/root-config --glibs)
```

Since ‘root-config –glibs’ returns the value with ‘-L’ at the head for the use in ‘LDLIBS’, there is no need to put ‘-L’ to the head of the right side. On the other hand, you need to put ‘-I’ for ‘root-config –incdir’.

Now, ‘make’ and confirm if the ‘SampleMonitorComp’ executable file is generated.

12 Launch and operation check

Now, the component development is completed, and so we make it read the data from the emulator to display a histogram on the screen.

We describe how to launch the component, based on the following directory structure as mentioned in Section 5.

```
/home/daq/MyDaq
/home/daq/MyDaq/SampleReader
```

```
/home/daq/MyDaq/SampleMonitor
```

As described in the “DAQ-Middleware 1.1.0 Technical Manual” [2], ‘DaqOperator’ controls the DAQ system in DAQ-Middleware. The ‘DaqOperator’ commands the connection as well as the data collection start & finish of the components already launched. The ways to boot each component include a network boot using ‘xinetd’. Here, the boot is done with the local boot function of the ‘/usr/bin/run.py’ command included in the DAQ-Middleware.

In the DAQ-Middleware, the DAQ system is configurable using ‘XML’ documents. For the details, see the “DAQ-Middleware 1.1.0 Technical Manual” [2]. Here, we copy ‘/usr/share/daqmw/ conf/sample.xml’ for our use.

```
% cd
% pwd
/home/daq/MyDaq
% cp /usr/share/daqmw/conf/sample.xml .
```

If you have your directory structure as assumed in this document, there is no need to modify it. However, if it is not so, the following modifications are needed.

- Replace the two ‘execPath’s with the path name of the component files.

The code part of ‘/usr/share/daqmw/conf/sample.xml’ is shown below.

```
<configInfo>
    <daqOperator>
        <hostAddr>127.0.0.1</hostAddr>
    </daqOperator>
    <daqGroups>
        <daqGroup gid="group0">
            <components>
                <component  cid="SampleReader0">
                    <hostAddr>127.0.0.1</hostAddr>
                    <hostPort>50000</hostPort>
                    <instName>SampleReader0.rtc</instName>
                    <execPath>/home/daq/MyDaq/SampleReader/SampleReaderComp</execPath>
                    <confFile>/tmp/daqmw/rtc.conf</confFile>
                    <startOrd>2</startOrd>
                    <inPorts>
                    </inPorts>
                    <outPorts>
                        <outPort>samplereader_out</outPort>
                    </outPorts>
                    <params>
                        <param  pid="srcAddr">127.0.0.1</param>
                        <param  pid="srcPort">2222</param>
                    </params>
                </component>
                <component  cid="SampleMonitor0">
```

```
26                      <hostAddr>127.0.0.1</hostAddr>
27                      <hostPort>50000</hostPort>
28                      <instName>SampleMonitor0.rtc</instName>
29                      <execPath>/home/daq/MyDaq/SampleMonitor/SampleMonitorComp</execPath>
30                      <confFile>/tmp/daqmw/rtc.conf</confFile>
31                      <startOrd>1</startOrd>
32                      <inPorts>
33                          <inPort    from="SampleReader0:samplereader_out">samplemonitor_in</inPort>
34                      </inPorts>
35                      <outPorts>
36                      </outPorts>
37                      <params>
38                      </params>
39                  </component>
40              </components>
41          </daqGroup>
42      </daqGroups>
43  </configInfo>
```

The 'OutPorts' on Line 17 designates an 'OutPort' that the 'SampleReader' component has. Also, 'params' on Line 20 designates the emulator's IP address and port specified by the 'SampleReader' component as its parameters. In 'SampleReader' component source file ('SampleReader.cpp'), 'parse_params()' acquires the value specified here. 'InPorts' on Line 32 designates an 'InPort' that the 'SampleMonitor' component has. For the details about the other tags, see the "DAQ-Middleware 1.1.0 Technical Manual" [2].

Now, we launch it.

You need to have the emulator launched by opening the emulator-launch terminal.

```
%    daqmw-emulator
```

Start 'run.py' as follows.

```
%  cd   /home/daq/MyDaq
%  ls
SampleReader  SampleMonitor  emulator  sample.xml
%  run.py  -c  -l sample.xml
```

The option '-c' of 'run.py' is the one to launch 'DaqOperator' in console mode. With this option specified, 'DaqOperator' will display the number of data bytes handled by each component, regularly (each component regularly reports the number of data bytes it processed, to 'DaqOperator'). '-l' option of 'run.py' looks for the path of the component to be started from 'sample.xml', and then start it in the path on the local computer.

Wait for a while after the launch of 'run.py' (approx. 4 seconds depending on CPU performance of computer), and then the screen below will be displayed. The 'DaqOperator' outputs these characters, and it waits for command key input in this state.

Available commands are displayed on Line 1 titled 'Command:'. Pressing number key corresponds to command input. State transition needs to be done one by one sequentially. For example, pressing 'start' will be judged inadequate input in this state. With a command input, 'DaqOperator' sends a transition command to each component. In this state immediately after the launch of 'run.py', components are in 'UNCONFIGURED' in Fig. 3 state transition chart.

```
 Command:      0:configure       1:start       2:stop      3:unconfigure        4:pause      5:resume

 RUN  NO:  0
 start  at:                 stop at:

   GROUP:COMP_NAME          EVENT  SIZE            STATE           COMP STATUS
group0:SampleReader0:                         0           LOADED            WORKING
group0:SampleMonitor0:                        0           LOADED            WORKING
```

After pressing 0 in this state and waiting for a while, the components will transits to the 'CONFIGURED' state.

```
 Command:      0:configure       1:start       2:stop      3:unconfigure        4:pause      5:resume

 RUN  NO:  0
 start  at:               stop at:

   GROUP:COMP_NAME          EVEN    SIZE            STATE           COMP STATUS
group0:SampleReader0:                         0     CONFIGURED          WORKING
group0:SampleMonitor0:                        0     CONFIGURED          WORKING
```

Next, with pressing 1, it asks you the run number. So, input any proper number e.g. 1. Now, the 'DaqComponent' commands 'start' to 'SampleReader' and 'SampleMonitor'. 'DaqOperator' displays on the screen the number of processed data bytes reported by each component, and updates it every few seconds. Here, with the value (30) of the 'm_monitor_update_rate' given in this code, the histogram is updated every 4 seconds. To terminate the components, press 2 to transit each of them to the 'STOP' state. The screen state at this moment is shown in Fig. 9.

After pressing 2 to 'stop' the component, pressing 'Ctrl-C' sends 'SIGINT' to 'DaqOperator' to terminate. 'SIGINT' is sent also to the component launched by 'run.py' at the same time it is sent to 'DaqOperator' (because 'DaqOperator' and each component launched by 'run.py' belong to the same process group). Since normally the components 'exit's earlier and'DaqOperator' will attempt to connect to them several times, the following lines will be displayed on the screen for the number of components after pressing 'Ctrl-C'.

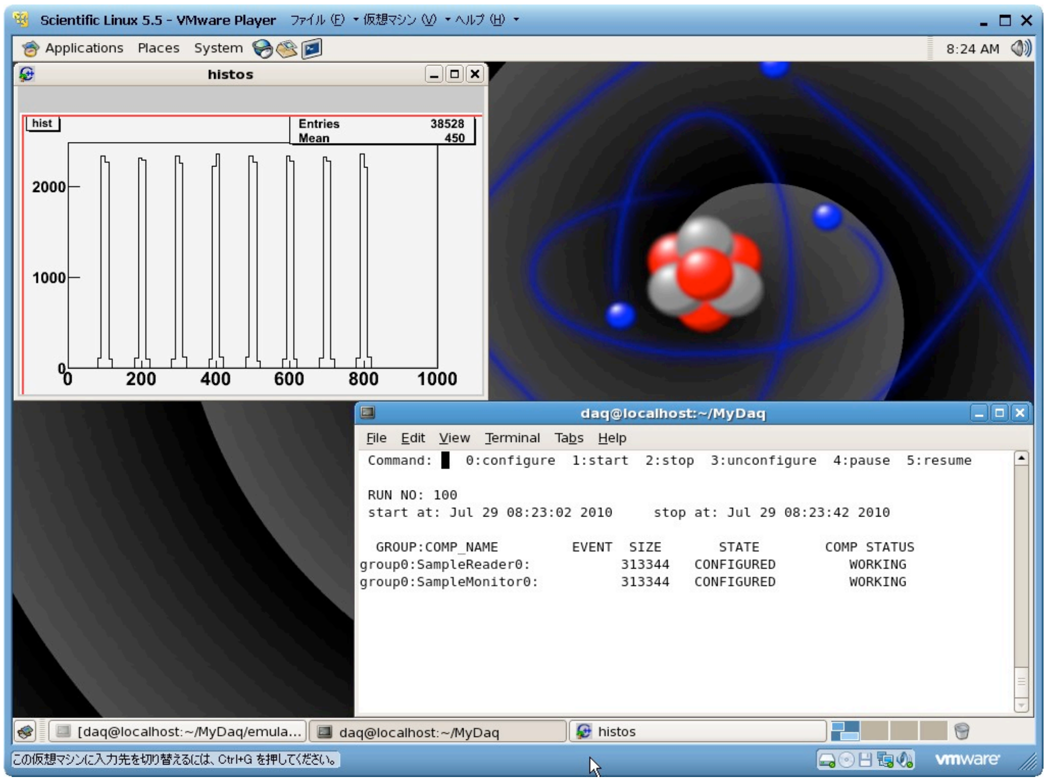
```
### ERROR:      : cannot  connect
### ERROR:      : cannot  connect
```

After a while, 'DaqOperator' terminates.

The update frequency of the histogram is specified with the value of the 'm_monitor_update_rate'.

Decreasing the value in the parentheses of the 'm_monitor_update_date()' at the initialization part in the 'SampleMonitor.cpp' below will increase the update frequency.

```
SampleMonitor::SampleMonitor(RTC::Manager*     manager)
       :  DAQMW::DaqComponentBase(manager),
           m_InPort("samplemonitor_in",                 m_in_data),
```


Scientific Linux 5.5 - VMware Player
Applications Places System
8:24 AM
histos
hist
Entries 38528
Mean 450
2000
1000
0
0 200 400 600 800 1000
daq@localhost:~/MyDaq
File Edit View Terminal Tabs Help
[daq@localhost:~/MyDaq/emula...
daq@localhost:~/MyDaq
histos
この仮想マシンに入力先を切り替えるには、Ctrl+G を押してください。
vmware

```
Command: █  0:configure  1:start  2:stop  3:unconfigure  4:pause  5:resume

RUN NO: 100
start at: Jul 29 08:23:02 2010     stop at: Jul 29 08:23:42 2010

 GROUP:COMP_NAME          EVENT  SIZE       STATE        COMP STATUS
group0:SampleReader0:           313344   CONFIGURED         WORKING
group0:SampleMonitor0:          313344   CONFIGURED         WORKING
```

Fig. 9 The screen of data collection terminated. The terminal that has launched the emulator is minimized and contained in the bottom panel.

```
        m_in_status(BUF_SUCCESS),
        m_canvas(0),                      // Addition
        m_hist(0),                        // Addition
        m_bin(0),                         // Addition
        m_min(0),                         // Addition
        m_max(0),                         // Addition
        m_monitor_update_rate(30),        // Addition
        m_event_byte_size(0),             // Addition
        m_debug(false)
```

This code requires some modifications and a recompile to change the update frequency (Try to change the value and see). To change the frequency without a recompile, you need to use the 'Condition' database in the DAQ-Middleware described in the next Section.

13 To make ‘Condition’ database of parameters

DAQ-Middleware has a framework called ‘Condition’ database to provide the system with the parameters varying for each run. In the ‘SampleMonitor’ component in the previous Section, it required some modification of the source code to change as shown below

- The number of bins in the histogram
- The minimum value of the histogram
- The maximum value of the histogram
- The update frequency of the histogram

that were unadjustable on the run-screen. Using ‘Condition’ database enables the parameter change without such modifications to the source code.

For the details about ‘Condition’ database, see the separate manual “’Condition’ database development manual” [3]. Here, we set the parameters of the above histogram according to the “Implementation using ‘class’” in this manual. The timing to set the parameters will be determined on ‘daq_start()’.

The source files modified as described below are under the ‘/usr/share/daqmw/examples/ConditionSampleMonitor’ directory. These have been modified based on the ‘/usr/share/daqmw/examples/SampleMonitor’, and for example, the modification will be displayed when the followings are executed.

```
%   cd   /usr/share/daqmw/examples
% diff -uprN SampleMonitor ConditionSampleMonitor
```

Also, the sample of ‘condition.xml’ is in ‘/usr/share/ daqmw/conf/condition.xml’. Below, we copy the source file of ‘SampleMonitor’ previously described, and modify it so that it can use the ‘Condition’ database.

```
%  cd
%  cd  MyDaq
%  cp  -r  SampleMonitor  ConditionSampleMonitor
%   cd   ConditionSampleMonitor
```

Newly generate ‘ConditionSampleMonitor.h’ and ‘ConditionSampleMonitor.cpp’ files, and, within these, create the ‘ConditionSampleMonitor’ class to acquire the variables that retains the parameters as well as the parameters themselves. Also, it needs to link the ‘JsonSpirit’ library and ‘boost-regex’ library so that the ‘DAQ’ component can read the ‘Condition’ database. Then, modify ‘Makefile’ together with the part for the additional source files. The modifications are the following two points.

- Link the ‘JsonSpirit’ library and the ‘boost_regex’ library.
- Add the additional source file ‘ConditionSampleMonitor.cpp’ to SRCS.

```
SRCS     +=     ConditionSampleMonitor.cpp
 ConditionSampleMonitor.o: ConditionSampleMonitor.h ConditionSampleMonitor.cpp
 LDLIBS  +=  -L/usr/lib/daqmw  -lJsonSpirit  -lboost_regex
```

The data structure of the parameters is defined with structure ‘monitorParam’ in the ‘ConditionSampleMonitor.h’.

```
#ifndef    _CONDITION_SAMPLEMONITOR_H
#define _CONDITION_SAMPLEMONITOR_H 1

#include  <string>
#include  "Condition.h"

struct monitorParam {
        unsigned  int  hist_bin;
        unsigned int hist_min;
        unsigned int hist_max;
        unsigned int monitor_update_rate;
};

typedef struct monitorParam  monitorParam;

class ConditionSampleMonitor  : public Condition  {
public:
        ConditionSampleMonitor();
        virtual ~ConditionSampleMonitor();
        bool   initialize(std::string   filename);
        bool   getParam(std::string   prefix,                   monitorParam*   monitorParam);
private:
        Json2ConList m_json2ConList;
        conList               m_conListSampleMonitor;
};

#endif
```

Next, the method to read ‘condition.json’ file and set the parameters to the ‘monitorParam’ structure variable, is added to ‘ConditionSampleMonitor.cpp’.

```
 #include  "ConditionSampleMonitor.h"

ConditionSampleMonitor::ConditionSampleMonitor()     {}
ConditionSampleMonitor::~ConditionSampleMonitor()     {}

bool
ConditionSampleMonitor::getParam(std::string prefix, monitorParam* monitorParam)
{
        setPrefix(prefix);
        unsigned  int  hist_bin;
        unsigned int hist_min;
        unsigned int hist_max;
        unsigned int monitor_update_rate;

        if (find("hist_bin", &hist_bin)) {
                monitorParam->hist_bin   =   hist_bin;
```

```
    }
    else {
        std::cerr << prefix + " hist_bin not fould" << std::endl;
        return false;
    }

    if(find("hist_min", &hist_min)) {
        monitorParam->hist_min  =  hist_min;
    }
    else {
        std::cerr << prefix + " hist_min not fould" << std::endl;
        return false;
    }

    if (find("hist_max", &hist_max)) {
        monitorParam->hist_max  =  hist_max;
    }
    else {
        std::cerr << prefix + " hist_max not fould" << std::endl;
        return false;
    }

    if (find("monitor_update_rate", &monitor_update_rate)) {
        monitorParam->monitor_update_rate  =  monitor_update_rate;
    }
    else {
        std::cerr << prefix + " monitor_update_rate not fould" << std::endl;
        return false;
    }

    return true;
}

bool ConditionSampleMonitor::initialize(std::string filename)
{
    if (m_json2ConList.makeConList(filename, &m_conListSampleMonitor) == false) {
        std::cerr << "### ERROR: Fail to read the Condition file "
                  << filename << std::endl;
    }
    init(&m_conListSampleMonitor);
    return true;
}
```

Now, the ‘ConditionSampleMonitor’ class has been prepared.

Subsequently, modify ‘SampleMonitor’ to acquire the histogram parameters using the ‘Condition’ database. First, the ‘Condition’ database filename ‘CONDITION_FILE’ as well as the structure ‘m_monitorParam’ that retains the parameters are added to ‘SampleMonitor.h’:

```
    ////////// ROOT Histogram //////////
    TCanvas *m_canvas;
    TH1F    *m_hist;
    unsigned    char    m_recv_data[4096];
    unsigned int m_event_byte_size;
    struct sampleData m_sampleData;
    ///////// Condition database ////////
    static const std::string CONDITION_FILE; // Addition
    monitorParam m_monitorParam;                    // Addition

    bool m_debug;
};
```

Further, substitute the values below for ‘CONDITION_FILE’ in ‘SampleMonitor.cpp’.

```
static const char* samplemonitor_spec[] =
{
    "implementation_id", "SampleMonitor",
    "type_name",         "SampleMonitor",
    "description",       "SampleMonitor component",
    "version",           "1.0",
    "vendor",            "Kazuo Nakayoshi, KEK",
    "category",          "example",
    "activity_type",     "DataFlowComponent",
    "max_instance",      "1",
    "language",          "C++",
    "lang_type",         "compile", ""
};

const std::string SampleMonitor::CONDITION_FILE = "./condition.json"; // Addition
```

Next, modify the ‘DAQ’ component source file so that it can use this class.
First, modify the ‘SampleMonitor.cpp’ to include ‘conditionSampleMonitor.h’.

```
#include "SampleMonitor.h"
#include "ConditionSampleMonitor.h" // Addition
```

Next, the ‘set_condition()’ function is added in the ‘SampleMonitor.cpp’ so that the ‘ConditionSampleMonitor’ class acquires the parameters. Also, ‘daq_start()’ is coded to call the ‘set_condition()’.

```
// Whole this function is added.
int set_condition(std::string condition_file, monitorParam *monitorParam)
{
    ConditionSampleMonitor  conditionSampleMonitor;
    conditionSampleMonitor.initialize(condition_file);
    if (conditionSampleMonitor.getParam("common_SampleMonitor_", monitorParam)) {
    std::cerr << "condition OK" << std::endl;
    }
    else {
        throw "SampleMonitor condition error";
    }

    return 0;
}

int SampleMonitor::daq_start()
{
    std::cerr << "*** SampleMonitor::start" << std::endl;
    m_in_status       = BUF_SUCCESS;

    try {                                                               // Addition
        set_condition(CONDITION_FILE, &m_monitorParam);                 // Addition
    }                                                                   // Addition
    catch (std::string error_message) {                                 // Addition
        std::cerr << error_message << std::endl;                        // Addition
        fatal_error_report(USER_DEFINED_ERROR1, "Condition error");     // Addition
    }                                                                   // Addition
    catch (...) {                                                       // Addition
        std::cerr << "unknown error" << std::endl;                      // Addition
        fatal_error_report(USER_DEFINED_ERROR1, "Unknown error");       // Addition
    }                                                                   // Addition
```

Further, modify the argument ‘TH1F()’ so that these acquired values are used as the number of bins, minimum and maximum values of the histogram. Also, modify the part determining the timing to update the histogram.

```
    m_hist = new TH1F("hist", "hist",
                      m_monitorParam.hist_bin,       // Change argument
                      m_monitorParam.hist_min,       // Change argument
                      m_monitorParam.hist_max);      // Change argument
```

```
    unsigned long sequence_num = get_sequence_num();
    if ((sequence_num % m_monitorParam.monitor_update_rate) == 0) { // Change
        m_hist->Draw();
        m_canvas->Update();
    }
```

Then, delete the part initializing ‘m_hist_bin’ etc. from the constructor of the ‘SampleMonitor’.

```
SampleMonitor::SampleMonitor(RTC::Manager*  manager)
    : DAQMW::DaqComponentBase(manager),
      m_InPort("samplemonitor_in",          m_in_data),
      m_in_status(BUF_SUCCESS),
      m_canvas(0),
      m_hist(0),
  // The initializations of ‘m_hist_bin’, ‘m_hist_min, m_hist_max’, ‘m_monitor_update_rate’ // have been deleted.
      m_event_byte_size(0),
      m_debug(false)
```

13.1 Histogram test using the ‘Condition’ database

The parameter values are given in ‘condition.xml’. Copy the samples in the ‘/usr/share/daqmw/conf/ condition.xml’ to the ‘/home/daq/MyDaq’ directory.

```
% cd   /home/daq/MyDaq
% cp /usr/share/daqmw/conf/condition.xml .
```

The component reads the file ‘condition.json’ which has been converted to JSON format but not this xml file directly. The conversion to JSON format is done by the command ‘condition_xml2json’.

```
%    condition_xml2json    condition.xml
```

This command creates ‘condition.json’ file[*13]. The ‘run.py -c -l sample.xml’ command launches it, similarly to the case without the ‘Condition’ database. The path for the component executable files has been changed (Since the directory has been changed from ‘SampleMonitor’ to ‘ConditionSampleMonitor’, the ‘execPath’ of ‘sample.xml’ needs to be changed to the full path for the component to be launched). Below, the modification to ‘sample.xml’ is shown.

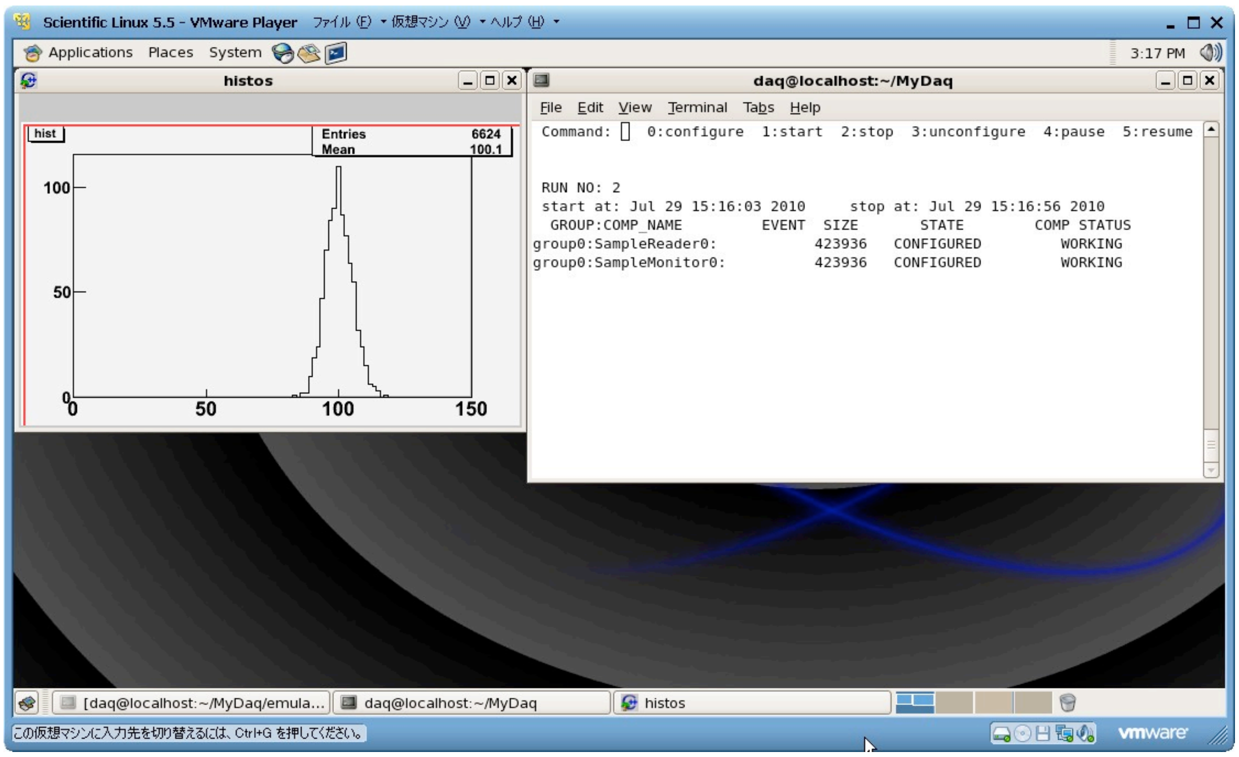
```
<component  cid="SampleMonitor0">
    <hostAddr>127.0.0.1</hostAddr>
    <hostPort>50000</hostPort>
    <instName>SampleMonitor0.rtc</instName>
    <!-- execPath changed -->
    <execPath>/home/daq/MyDaq/ConditionSampleMonitor/SampleMonitorComp</execPath>
    <!-- execPath changed          ^^^^^^^^^^^^^^^^^^^^^^^  -->
    <confFile>/tmp/daqmw/rtc.conf</confFile>
    <startOrd>1</startOrd>
    <inPorts>
        <inPort    from="SampleReader0:samplereader_out">samplemonitor_in</inPort>
    </inPorts>
    <outPorts>
    </outPorts>
    <params>
    </params>
</component>
```

The ‘run.py -c -l sample.xml’ launches it similarly to the previous case[*14]. Confirm if the number of bins, minimum and maximum values, etc. for the histogram are the same as those in ‘condition.xml’. After modifying the parameters of ‘condition.xml’, update ‘condition.json’ by ‘condition_xml2json condition.xml’ again and then start the component, to confirm that the histogram has the parameters specified in ‘Condition’ file.

Fig. 10 shows an example where the number of bins, minimum and maximum values for the histogram are set as 100, 0, 150, respectively, using the ‘Condition’ database. The 423936 bytes displayed on the terminal by ‘DaqOperator’ shows that the total of 52992 event data was collected. The data coming in from the emulator are centered on 100, 200, …, 800 evenly. ‘Entries’ item in the histogram on left side of the figure confirms that 6624 data, i.e. 1/8 of the total 52992 data has been incremented to the histogram.

[*13] Since this command is a shell script which uses the ‘Xalan’ command in it, the ‘xalan’ package is required.

[*14] Otherwise, you can leave ‘sample.xml’ as is and copy it to ‘conditionsample.xml’. Then, modify the above ‘execPath’ to launch it by ‘run.py -c -l conditionsample.xml’.


Scientific Linux 5.5 - VMware Player
Applications Places System
3:17 PM
histos
hist
Entries 6624
Mean 100.1
100
50
0
0
50
100
150
daq@localhost:~/MyDaq
File Edit View Terminal Tabs Help
[daq@localhost:~/MyDaq/emula...
daq@localhost:~/MyDaq
histos
この仮想マシンに入力先を切り替えるには、Ctrl+G を押してください。
vmware

```
 Command: [ ]  0:configure  1:start  2:stop  3:unconfigure  4:pause  5:resume


 RUN NO: 2
 start at: Jul 29 15:16:03 2010     stop at: Jul 29 15:16:56 2010
  GROUP:COMP_NAME          EVENT  SIZE       STATE        COMP STATUS
group0:SampleReader0:            423936   CONFIGURED         WORKING
group0:SampleMonitor0:           423936   CONFIGURED         WORKING
```

Fig. 10 An example where data is read with the histogram's minimum value = 0, maximum value = 150, the number of bins = 100 set by the 'Condition' database. See the text of this document for the numerical value of 'Entries' in the histogram.

14 How to use WebUI

Previously, the 'DaqOperator' was launched in console mode and then given commands through the keyboard. The 'WebUI' has been added since the DAQ-Middleware 1.1.0. In this Section, we describe how to use it.

Up to the DAQ-Middleware 1.1.1, 'firefox' was the only Web browser that could be used. Since the DAQ-Middleware 1.1.2, we can use 'IE', 'firefox', 'chrome' and 'safari'. However, in 'IE', "The XML document cannot be displayed (XML 文書が表示できません)" is displayed in "XML Response from DaqOperator" at the bottom of the Fig. 11, and the XML returned from 'DaqOperator' is not displayed (it can still be operated manually).

14.1 Check the software package

To use the 'WebUI', the 'mod_python' package is required for 'Scientific Linux 5.x', and the 'mod_wsgi' package is required for the other Linux distributions (including Scientific Linux 6.x). To check if the above package is already installed, execute the following.

```
% rpm -q mod_python  or  rpm -q mod_wsgi
```

If the terminal says "package mod_python is not installed", it needs to be installed using the following command.

```
%  su  (answer root password)
# yum install mod_python   or   yum install mod_wsgi
```

Right after the installation of 'mod_python' or 'mod_wsgi', 'httpd' cannot use them. So, restart 'httpd'.

```
# service  httpd  restart
```

If 'httpd' is not yet launched, the terminal says "Stopping httpd: [FAILED]". It is OK with " Starting httpd: [OK]".

To start 'httpd' automatically at every reboot, execute the following.

```
# chkconfig  httpd  on
```

Also, 'SELinux' needs to be disabled since 'WebUI' does not operate with 'SELinux' enabled. For how to check if the setting is enabled and to disable it, see Appendix F.5.

14.2 How to operate

The component starts by ‘run.py –l’ without ‘-c’.

```
[daq@localhost MyDaq]$ run.py -l sample.xml
Use config file sample.xml
Use /usr/share/daqmw/conf/config.xsd for XML schema
Use /usr/libexec/daqmw/DaqOperatorComp for DAQ-Operator
Conf file validated: sample.xml
start new naming service... done
Local Comps booting... done
Now booting the DAQ-Operator... done
[daq@localhost MyDaq]$
```

The ‘pgrep’ command confirms if all the necessary components have started.

```
[daq@localhost MyDaq]$ pgrep -fl Comp
6189 /home/daq/MyDaq/SampleReader/SampleReaderComp -f /tmp/daqmw/rtc.conf
6208 /home/daq/MyDaq/SampleMonitor/SampleMonitorComp -f /tmp/daqmw/rtc.conf
6217 /usr/libexec/daqmw/DaqOperatorComp -f ./rtc.conf -h 127.0.0.1 -p 9876 -x sample.xml
[daq@localhost MyDaq]$
```

The numbers displayed at the head of the above lines, are process IDs. Here, you see that all the components necessary in this example system (‘SampleReader’, ‘SampleMonitor’, ‘DaqOperator’) have started.

Now, launch your Web browser (Clicking the icon on the right side to ‘System’ on the upper panel will start it.) to access ‘http://localhost/daqmw/operatorPanel/operatorPanel0.html’. The screen state at this moment is shown in Fig. 11.

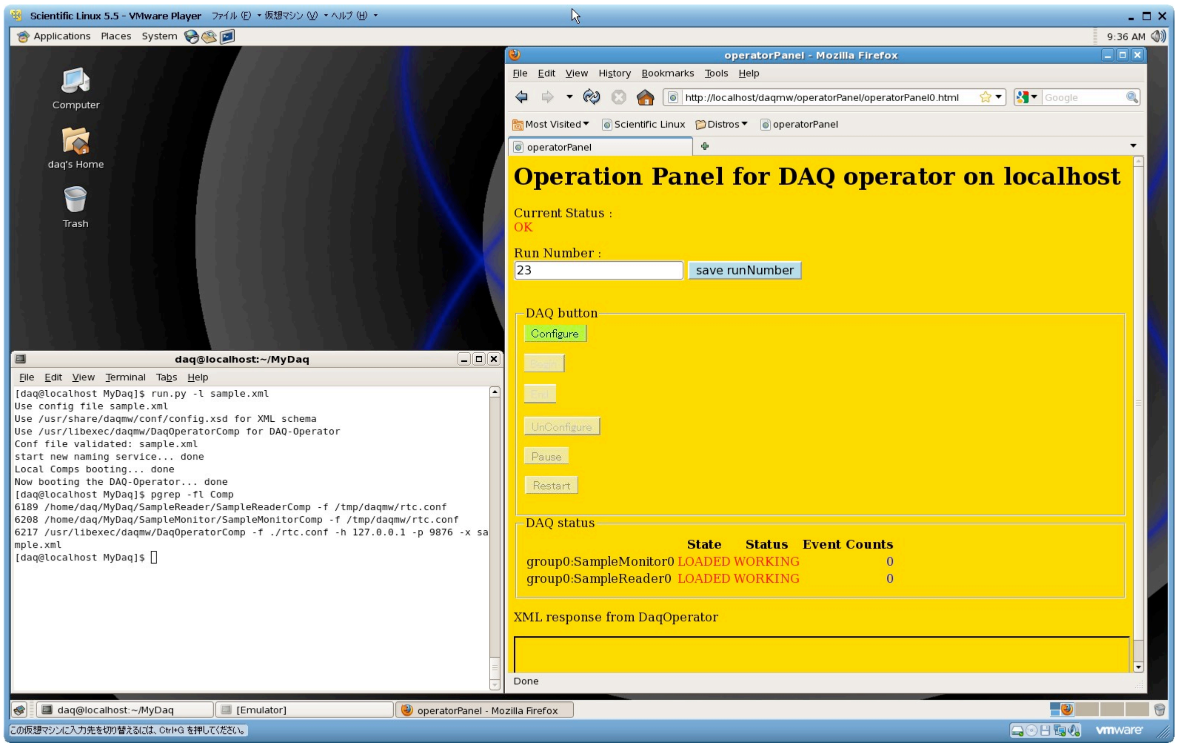
On the Web screen, from its top, ‘Current Status’, ‘Run Number’ entry field, DAQ button, ‘DAQ status’ and XML response from ‘DaqOperator’ are displayed.

The ‘Run Number’ is automatically incremented each time a run terminates. Also, you can specify the number manually. Inputting the number in the number field and pressing the “save runNumber” button will save it.

Pressing the DAQ button gives the command to ‘DaqOperator’. At that time, the button that can be pressed is determined by ‘state’. The green buttons are the currently usable. .

The component name, current ‘State’, ‘Status’ and the number of events are displayed in the ‘DAQ status’ field.

Pressing the ‘Configure’ button in this state and subsequently the ‘Begin’ button will start a data collection to draw the histogram. The screen at this moment is shown in Fig. 12. To terminate, press the ‘End’ button. Pressing the ‘Begin’ button subsequently will start another data collection as a next run. After the ‘End’ button, pressing the ‘Unconfigure’ button will make a transition to the ‘LOADED’ state, and the ‘State’ field in the ‘DAQ Status’ will show ‘LOADED’.



```
[daq@localhost MyDaq]$ run.py -l sample.xml
Use config file sample.xml
Use /usr/share/daqmw/conf/config.xsd for XML schema
Use /usr/libexec/daqmw/DaqOperatorComp for DAQ-Operator
Conf file validated: sample.xml
start new naming service... done
Local Comps booting... done
Now booting the DAQ-Operator... done
[daq@localhost MyDaq]$ pgrep -fl Comp
6189 /home/daq/MyDaq/SampleReader/SampleReaderComp -f /tmp/daqmw/rtc.conf
6208 /home/daq/MyDaq/SampleMonitor/SampleMonitorComp -f /tmp/daqmw/rtc.conf
6217 /usr/libexec/daqmw/DaqOperatorComp -f ./rtc.conf -h 127.0.0.1 -p 9876 -x sa
mple.xml
[daq@localhost MyDaq]$ 
```

http://localhost/daqmw/operatorPanel/operatorPanel0.html

Operation Panel for DAQ operator on localhost

Current Status :
OK

Run Number :
23 save runNumber

DAQ button

Configure

UnConfigure

Pause

Restart

DAQ status

	State	Status	Event Counts
group0:SampleMonitor0	LOADED	WORKING	0
group0:SampleReader0	LOADED	WORKING	0

XML response from DaqOperator

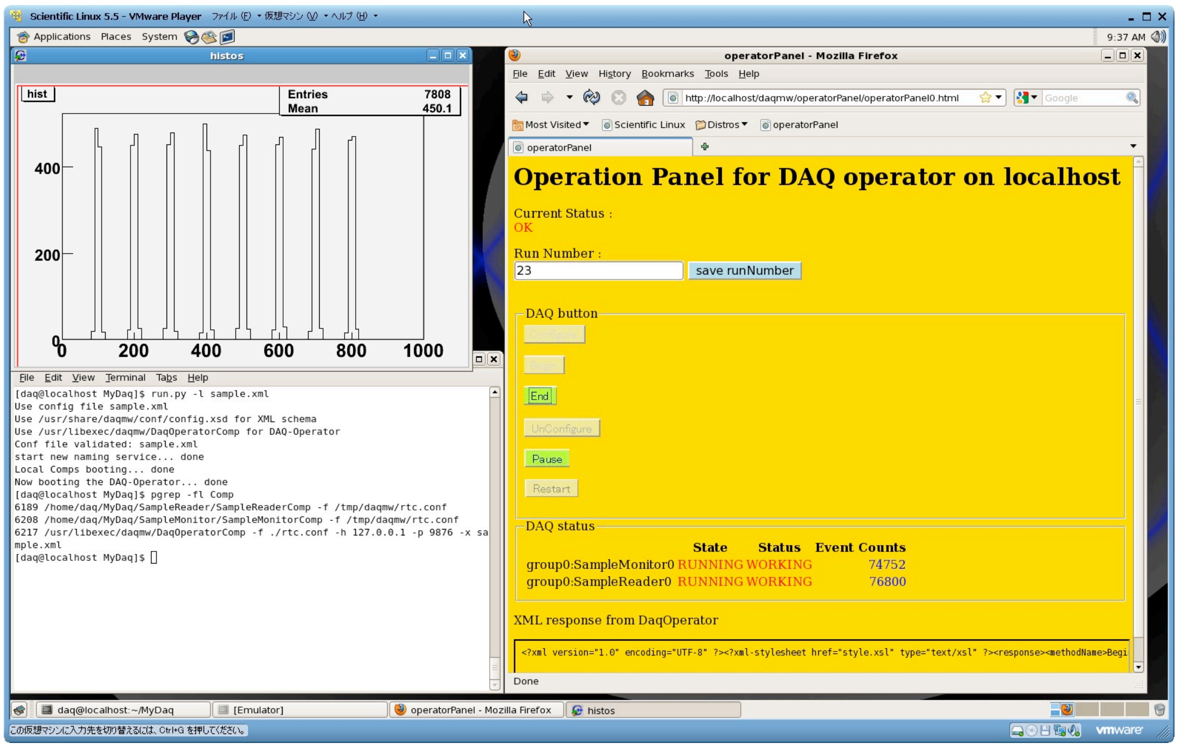
Done

Scientific Linux 5.5 - VMware Player
Applications Places System
9:36 AM
Computer
daq's Home
Trash
daq@localhost:~/MyDaq
File Edit View Terminal Tabs Help
operatorPanel - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Most Visited Scientific Linux Distros operatorPanel
[Emulator]
vmware

Fig. 11 WebUI, now running the emulator (it is running in the terminal titled 'Emulator' stored in the bottom panel) to start each component by 'run.py -l sample.xml', and access 'http://localhost/daqmw/operatorPanel/operatorPanel0.html' with Web browser.

To stop the component process, use the 'pkill' command from the command line. Since the 'pkill' command refers to the process name of no more than 15 characters by default, simply executing the "pkill Comp" may not send the 'SIGTERM' to the component. In order to make the 'pkill' command to refer to the full name of each process, execute it with '-f'.

```
% pkill -f Comp
```



```
[daq@localhost MyDaq]$ run.py -l sample.xml
Use config file sample.xml
Use /usr/share/daqmw/conf/config.xsd for XML schema
Use /usr/libexec/daqmw/DaqOperatorComp for DAQ-Operator
Conf file validated: sample.xml
start new naming service... done
Local Comps booting... done
Now booting the DAQ-Operator... done
[daq@localhost MyDaq]$ pgrep -fl Comp
6189 /home/daq/MyDaq/SampleReader/SampleReaderComp -f /tmp/daqmw/rtc.conf
6208 /home/daq/MyDaq/SampleMonitor/SampleMonitorComp -f /tmp/daqmw/rtc.conf
6217 /usr/libexec/daqmw/DaqOperatorComp -f ./rtc.conf -h 127.0.0.1 -p 9876 -x sa
mple.xml
[daq@localhost MyDaq]$
```

Operation Panel for DAQ operator on localhost

Current Status :
OK

Run Number :
23

DAQ status

	State	Status	Event Counts
group0:SampleMonitor0	RUNNING	WORKING	74752
group0:SampleReader0	RUNNING	WORKING	76800

XML response from DaqOperator

```
<?xml version="1.0" encoding="UTF-8" ?><?xml-stylesheet href="style.xsl" type="text/xsl" ?><response><methodName>Begi
```

Fig. 12 The screen state at ‘RUNNING’, the histogram on left side is regularly updated.

Appendix A About the libraries provided with DAQ-Middleware

The DAQ-Middleware provides the library possibly and commonly used in any of the DAQ systems. The follow are currently provided.

- The socket library written in C
- The library for 'SiTCP Bus Control Protocol (BCP)' written in C
- The socket library written in C++
- The 'JsonSpirit' library used in the 'Condition' database

The include files are installed in '/usr/include/daqmw'. Also, the shared library files and static library files are installed in '/usr/lib/daqmw' for 32 bit Scientific Linux (SL) and '/usr/lib64/daqmw' for 64 bit SL. These are installed in original directories due to their relatively large number of files and also for the purpose to separate them from the library files provided by OS and the other application packages.

Since '/usr/lib/daqmw' is not a standard directory like '/usr/lib' which 'ld.so' refers to, you need to command 'ld.so'to search '/usr/lib/daqmw' as well as the following, when you run the component using these libraries.

(1) To use environment variable 'LD_LIBRARY_PATH'

The '/usr/lib/daqmw' for 32bit SL or '/usr/lib64/daqmw' for 64bit SL is added to the environment variable 'LD_LIBRARY_PATH'. When you network-boot using 'xinetd', although this is not seen in this manual, 'LD_LIBRARY_PATH' is not set to the environment variable for the process started on a remote computer. So, if you use 'LD_LIBRARY_PATH', you need to avoid this in by including setting 'env' attribute in the settings file of 'xinetd' (the file under '/etc/xinetd.d/' directory).

(2) To use 'ld.so' settings file

In 'Scientific Linux', creating the files under '/etc/ld.so.conf.d/' and designating the directory having the shared library files for them, will enable 'ld.so' to search in the designated directory as well. To use this function, create the '/etc/ld.so.conf.d/daqmw.conf' file and then write the '/usr/lib/daqmw' for 32bit SL or '/usr/lib64/daqmw' for 64bit SL, therein. To enable this function immediately after creating the file, you need to execute the following once as the 'root' user.

```
root# ldconfig
```

After rebooting, the function is enabled automatically. So, you do not need to execute the 'ldconfig' command for every reboot.

The place to put the file to write in the settings is different for OSs. In most Linux, it is placed in '/etc/ld.so.conf' or the files under the '/etc/ld.so.conf.d/' directory read out by '/etc/ld.so.conf'.

The DAQ-Middleware has adopted the latter way to place the settings file in the '/etc/ld.so.conf.d/' directory. The advantages of this method are that each user does not need to consider the value of the environment variable 'LD_LIBRARY_PATH' for a run, and that the user does not need to elaborately pass 'LD_LIBRARY_PATH' to multiple computers at the component's start when any data collection is distributed to them (in the case of the above 'xinetd', using 'env' attribute, etc.).

When installing from the source files, the system checks the presence of the '/etc/ld.so.conf.d/' directory at 'make install' and, if there, generates the 'daqmw.conf' file. In the file, '/usr/lib64/daqmw' will be written if the '/usr/lib64/' directory is present, or otherwise, without the directory, '/usr/lib/daqmw/'.

Appendix B Update history of this manual

The update history of this manual is summarized below.

2010-08

- Initial revision
- For DAQ-Middleware 1.0.0
- Used in the DAQ-Middleware lecture held in August 2010.

2011-01

- Bundled in DAQ-Middleware 1.0.1(/usr/share/daqmw/docs/DAQ-Middleware-DevManual.pdf)
- Added the outline of DAQ-Middleware

2011-02

- Changed the title for DAQ-Middleware 1.0.2
- No updates to the contents since the manual for 1.0.1.

2011-06

- Updated for DAQ-Middleware 1.1.0 supporting 64bit.
- Added the description for WebUI added since DAQ-Middleware 1.1.0.
- Added tips for compiling from source files.
- Miscellaneous corrections of characters and phrases, etc.

2011-10

- Added reboot set-up method.
- Added installation of Scientific Linux 5.x.

2012-04

- Update for DAQ-Middleware 1.2.0.

Appendix C Log generated on the set-up using ‘rpm’ and ‘yum’ commands

```
[root@localhost ~]# rpm -ihv http://daqmw.kek.jp/rpm/el5/noarch/
(→) kek-daqmiddleware-repo-2-0.noarch.rpm (Returning this too long line with (→) )
Retrieving http://daqmw.kek.jp/rpm/el5/noarch/kek-daqmiddleware-repo-2-0.noarch.rpm
Preparing...                ########################################### [100%]
   1:kek-daqmiddleware-repo ########################################### [100%]
[root@localhost ~]# yum --enablerepo=kek-daqmiddleware install DAQ-Middleware
Loaded plugins: kernel-module
kek-daqmiddleware                                         |  951 B     00:00
kek-daqmiddleware/primary                                 | 5.1 kB     00:00
kek-daqmiddleware                                                        15/15
Setting up Install Process
Resolving Dependencies
--> Running transaction check
---> Package DAQ-Middleware.i386 0:1.0.0-0.el5 set to be updated
--> Processing Dependency: OpenRTM-aist >= 1.0.0 for package: DAQ-Middleware
--> Processing Dependency: xerces-c-devel for package: DAQ-Middleware
--> Processing Dependency: libomniDynamic4.so.0 for package: DAQ-Middleware
--> Processing Dependency: libcoil.so.0 for package: DAQ-Middleware
--> Processing Dependency: libRTC-1.0.0.so.0 for package: DAQ-Middleware
--> Processing Dependency: xalan-c-devel for package: DAQ-Middleware
--> Processing Dependency: libxerces-c.so.27 for package: DAQ-Middleware
--> Processing Dependency: libomniORB4.so.0 for package: DAQ-Middleware
--> Processing Dependency: libomnithread.so.3 for package: DAQ-Middleware
--> Running transaction check
---> Package OpenRTM-aist.i386 0:1.0.0-2.el5 set to be updated
--> Processing Dependency: omniORB-doc for package: OpenRTM-aist
--> Processing Dependency: omniORB-bootscripts for package: OpenRTM-aist
--> Processing Dependency: omniORB-utils for package: OpenRTM-aist
--> Processing Dependency: omniORB-servers for package: OpenRTM-aist
--> Processing Dependency: omniORB-devel for package: OpenRTM-aist
---> Package omniORB.i386 0:4.0.7-4.el5 set to be updated
---> Package xalan-c-devel.i386 0:1.10.0-2.el5 set to be updated
--> Processing Dependency: xalan-c = 1.10.0-2.el5 for package: xalan-c-devel
--> Processing Dependency: libxalanMsg.so.110 for package: xalan-c-devel
--> Processing Dependency: libxalan-c.so.110 for package: xalan-c-devel
---> Package xerces-c.i386 0:2.7.0-1.el5.rf set to be updated
---> Package xerces-c-devel.i386 0:2.7.0-1.el5.rf set to be updated
--> Running transaction check
---> Package omniORB-bootscripts.i386 0:4.0.7-4.el5 set to be updated
---> Package omniORB-devel.i386 0:4.0.7-4.el5 set to be updated
---> Package omniORB-doc.i386 0:4.0.7-4.el5 set to be updated
---> Package omniORB-servers.i386 0:4.0.7-4.el5 set to be updated
---> Package omniORB-utils.i386 0:4.0.7-4.el5 set to be updated
---> Package xalan-c.i386 0:1.10.0-2.el5 set to be updated
--> Finished Dependency Resolution
Beginning Kernel Module Plugin
Finished Kernel Module Plugin

Dependencies Resolved

================================================================================
 Package                Arch      Version                  Repository        Size
================================================================================
===========
Installing:
  DAQ-Middleware            i386     1.0.0-0.el5            kek-daqmiddleware     1.0 M
Installing for dependencies:
  OpenRTM-aist              i386     1.0.0-2.el5            kek-daqmiddleware     6.6  M
  omniORB                   i386     4.0.7-4.el5            kek-daqmiddleware     6.4  M
  omniORB-bootscripts       i386     4.0.7-4.el5            kek-daqmiddleware     6.1  k
  omniORB-devel             i386     4.0.7-4.el5            kek-daqmiddleware     2.9  M
  omniORB-doc               i386     4.0.7-4.el5            kek-daqmiddleware     986  k
  omniORB-servers           i386     4.0.7-4.el5            kek-daqmiddleware      59  k
  omniORB-utils             i386     4.0.7-4.el5            kek-daqmiddleware      37  k
  xalan-c                   i386     1.10.0-2.el5           kek-daqmiddleware     1.2  M
  xalan-c-devel             i386     1.10.0-2.el5           kek-daqmiddleware     443  k
  xerces-c                  i386     2.7.0-1.el5.rf         kek-daqmiddleware     1.6  M
  xerces-c-devel            i386     2.7.0-1.el5.rf         kek-daqmiddleware     649  k

Transaction Summary
================================================================================
Install          12 Package(s)
Upgrade           0  Package(s)

Total download size: 22 M
Is this ok [y/N]: y (Input y)
Downloading Packages:
(1/12):    omniORB-bootscripts-4.0.7-4.el5.i386.rpm                | 6.1  kB      00:00
(2/12):    omniORB-utils-4.0.7-4.el5.i386.rpm                      |  37  kB      00:00
(3/12):    omniORB-servers-4.0.7-4.el5.i386.rpm                    |  59  kB      00:00
(4/12):    xalan-c-devel-1.10.0-2.el5.i386.rpm                     | 443  kB      00:00
(5/12):    xerces-c-devel-2.7.0-1.el5.rf.i386.rpm                  | 649  kB      00:00
(6/12):    omniORB-doc-4.0.7-4.el5.i386.rpm                        | 986  kB      00:00
(7/12):    DAQ-Middleware-1.0.0-0.el5.i386.rpm                     | 1.0  MB      00:00
(8/12):    xalan-c-1.10.0-2.el5.i386.rpm                           | 1.2  MB      00:00
(9/12):    xerces-c-2.7.0-1.el5.rf.i386.rpm                        | 1.6  MB      00:00
(10/12): omniORB-devel-4.0.7-4.el5.i386.rpm                        | 2.9  MB      00:00
(11/12): omniORB-4.0.7-4.el5.i386.rpm                              | 6.4  MB      00:00
(12/12): OpenRTM-aist-1.0.0-2.el5.i386.rpm                         | 6.6  MB      00:00
               ------------------------------------------------------------------------------
Total                                                     10 MB/s |    22 MB       00:02
Running rpm_check_debug
Running Transaction Test
Finished Transaction Test
Transaction Test Succeeded
Running Transaction
   Installing         : omniORB                                                  1/12
   Installing         : xerces-c                                                 2/12
   Installing         : xerces-c-devel                                           3/12
   Installing         : omniORB-doc                                              4/12
   Installing         : omniORB-utils                                            5/12
   Installing         : omniORB-servers                                          6/12
   Installing         : xalan-c                                                  7/12
   Installing         : omniORB-devel                                            8/12
   Installing         : xalan-c-devel                                            9/12
   Installing         : omniORB-bootscripts                                     10/12
   Installing         : OpenRTM-aist                                            11/12
   Installing         : DAQ-Middleware                                          12/12

Installed:
   DAQ-Middleware.i386 0:1.0.0-0.el5

Dependency Installed:
   OpenRTM-aist.i386 0:1.0.0-2.el5                omniORB.i386  0:4.0.7-4.el5
  omniORB-bootscripts.i386 0:4.0.7-4.el5          omniORB-devel.i386   0:4.0.7-4.el5
  omniORB-doc.i386 0:4.0.7-4.el5                  omniORB-servers.i386 0:4.0.7-4.el5
  omniORB-utils.i386  0:4.0.7-4.el5               xalan-c.i386   0:1.10.0-2.el5
  xalan-c-devel.i386  0:1.10.0-2.el5              xerces-c.i386   0:2.7.0-1.el5.rf
  xerces-c-devel.i386  0:2.7.0-1.el5.rf

Complete!
[root@localhost  ~]#
```

Appendix D Tips for the installation from source files

It is easier to use the 'RPM' binaries to set up the middleware to the 'Scientific Linux', 'CentOS', 'RedHat Enterprise Linux (5.x, 6.x). Since the binaries for the other OSs have not yet been provided, you need to compile the DAQ-Middleware from the source files if you want to use it on those OSs. This Section summarizes the tips for the installation of the DAQ-Middleware from the source files.

The following software are required to run the DAQ-Middleware. If these softwares are included in your OS distribution, it should be easier to install them.

- A set of omniORB (4.1.x)
- OpenRTM-aist 1.0.0 + patch
- xerces-c 2.x or 3.x
- xalan-c 1.10
- boost

The DAQ-Middleware uses 'OpenRTM-aist' for its basis. The 'OpenRTM-aist' uses 'omniORB'. Also, DAQ-Middleware uses 'xerces-c' for its XML-related library. It uses 'xalan-c' and 'boost' to convert XML to the json file for the 'Condition' database. Some sample components use 'boost'.

The 'omniORB' requires a development environment, such as 'omniidl' and 'omniNames' to operate. The 'omniNames' may automatically start at reboot if you have installed the one provided from your OS distribution. In order to prevent any troubles concerning 'omniNames' in your development, the started 'run.py' stops 'omniNames' once and then restarts it. The automatic start of 'omniNames' needs to be set disabled since the developer user's authority often cannot stop it with its enabled automatic start on OS launch. In 'RHEL' system, execute "chkconfig omniNames off" as 'root' user.

The 'xerces-c' is compliant to both 2.x and 3.x. 'xalan-c' uses 'xerces-c'. When using the binaries provided by your Linux distribution, you need to install 'xalan-c' with its version matched to that of 'xerces-c'.

The patch for 'OpenRTM-aist' is included in 'OpenRTM-aist-1.0. 0-X.r1971.el5.src.rpm' located at 'http://daqmw.kek.jp/rpm/SRPMS/'. Since this 'SRPM' file also includes the files equivalent to 'OpenRTM-aist-1.0.0', extract the desired files by the 'rpm2cpio' command and work as follows. The 'Autotools' is required for this work.

```
tar xf OpenRTM-aist-r1971.tar.gz cd
OpenRTM-aist-r1971
```

```
Apply all the patches included in ‘SRPMS’ (concrete commands are omitted here)
sh       build/autogen
./configure --prefix=/usr (Change the value for ‘--prefix’ as required)
make
make  install
```

After these preparations are completed, obtain the source files for the DAQ-Middleware from ‘http://daqmw.kek.jp/src/’ to expand and then execute the following.

```
make
make  install
```

You can use the ‘uninstall’ target of ‘Makefile’ to uninstall the middleware.

```
cd DAQ-Middleware-1.x.y
make   uninstall
```

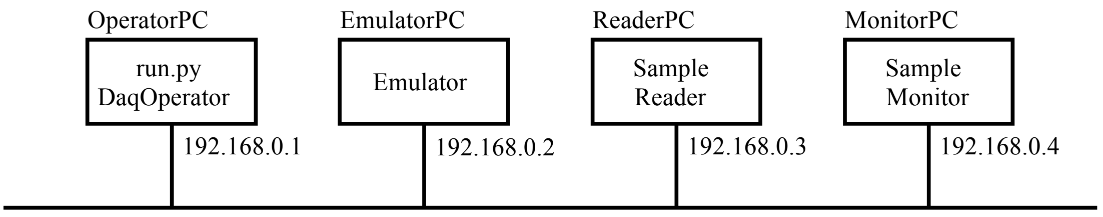


Fig. 13 The arrangement of the computers for explaining the remote boot. OperatorPC, EmulatorPC, etc. are host names. Their IP addresses are shown below the names. The DAQ component and emulator run on each computer is specified in the rectangles.

Appendix E Remote boot set-up

Here, we describe how to arrange the system distributing DAQ components to multiple computers. The computer arrangement exampled in this Section is shown in Fig. 13. In the arrangement of this example, 'SampleReader' is run on 'ReaderPC', and then the data is sent to 'SampleReader' running on 'MonitorPC' via network.

With the DAQ components distributed to multiple computers, it requires a mechanism to start the components over the network. Here, we use 'xinetd'.

The related files other than the 'xinet' program file are '/etc/services', '/etc/xinetd.d/bootComps' and '/usr/share/daqmw/etc/remote-boot/bootComps.py'.

E.1   Network communication check

Data transfer via network requires communications between each computer. You should notice that it is not always possible to have other communications even with a ping response. When packet filtering with 'iptables' etc., you can prevent problems by having it stopped[*15]. To stop the packet filtering using 'iptables', execute the following.

```
root# service iptables stop
```

To disable it in a stopped state at the start of a component, execute the following.

```
root# chkconfig iptables off
```

[*15] For the condition to stop to ensure the security packet filtering, check the security policy etc. for the network you get connected to.

E.2 Installation of ‘xinetd’

The ‘rpm -q xinetd’ can check if ‘xinetd’ has been installed. If it has not, install it from your OS distribution repository by executing the following.

```
root# yum install xinetd
```

It is required to make ‘xinetd’ available on all the computers that start DAQ components. In the example of fig. 13, ‘xinetd’ is needed for ‘ReaderPC’ and ‘MonitorPC’.

E.3 Set-up of ‘xinetd’

Add ‘bootComps’ to ‘/etc/services—‘using the following command (Notice that ‘>>’ of two ‘>’s is needed instead of one single ‘>’).

```
root#  cat  /usr/share/daqmw/etc/remote-boot/services.sample  >>  /etc/services
```

Create ‘/etc/xinetd.d/bootComps’ defining the program started by ‘xinetd’ as follows (Since one command line is too long, it is returned with (→). However, input the command just in one line when executing).

```
root# cp /usr/share/daqmw/etc/remote-boot/bootComps-xinetd.sample (→)
/etc/xinetd.d/bootComps
```

To start ‘xinetd’, execute the following.

```
root#  service  xinetd  restart
```

After the screen below appears, confirm if ‘xinetd’ has normally started by referring to ‘/var/log/messages’.

```
root#  service  xinetd  restart
Stopping  xinetd:  [FAILED]  (When ‘xinetd’ has not started, it shows FAILED.
Starting  xinetd:  [            OK   ]     If ‘Starting xinetd’ shows OK, it is successful.)
root#
```

If an error appears, make the necessary corrections with reference to the output messages. To check if it is indeed operable, execute the following from the other computer (Replace ‘remote-host’ with any host name or its IP address to which ‘xinetd’ was set up).

```
% echo hello |nc remote-host 50000
```

Now, if the following screen is output, it was successful.

```
-1 need more info.
```

When this screen is not displayed, check '/var/log/messages' on the computer on which you set the 'xinetd' startable, or check if any packet filtering is running but the packet has not been delivered, etc. The registration of user 'daq' is required on the computers that start DAQ components since 'bootComps-xinetd.sample' starts them under the 'daq' user authority (To change the user name, modify the right side of 'user =' in '/etc/xinetd.d/bootComps'). To start 'xinetd' automatically at every boot, execute the following.

```
root# chkconfig xinetd on
```

Even when you start the DAQ components on the computer that runs run.py, you still need to do it via 'xinetd'.

The DAQ component logs are generated in '/tmp/daqmw/' directory. The log is generated under the process authority of the DAQ component. If you have any '/tmp/daqmw/' directory owned by another user, delete the logs together with the directory.

E.4 Creating configuration file

The configuration file for the arrangement of Fig. 13 is shown in Fig. 14.
'execPath' specifies the location of the DAQ component program files.The information including the computer that operates a name server when starting the DAQ component, is written in 'confFile'. The 'run.py' automatically generates it and then sends it to the remote computer. Since the location of 'confFile' is fixed to '/tmp/daqmw/rtc.conf' in 'run.py', any settings for it specified in a configuration file will be ignored.

E.5 System launch

To start the system, execute the following assuming the configuration file as 'remote.xml', after starting 'Emulator'.

```
daq% run.py -c remote.xml
```

Although the histogram by 'SampleMonitor' is not drawn anywhere on the screen without the permission from X-Window by default, you can check the normal data-reading by referring to the number of events displayed by 'DaqOperator'.

```
<configInfo>
        <daqOperator>
                <hostAddr>192.168.0.1</hostAddr>
        </daqOperator>
        <daqGroups>
            <daqGroup gid="group0">
                <components>
                    <component   cid="SampleReader0">
                        <hostAddr>192.168.0.3</hostAddr>
                        <hostPort>50000</hostPort>
                        <instName>SampleReader0.rtc</instName>
                        <execPath>/home/daq/MyDaq/SampleReader/SampleReaderComp</execPath>
                        <confFile>/tmp/daqmw/rtc.conf</confFile>
                        <startOrd>2</startOrd>
                        <inPorts>
                        </inPorts>
                        <outPorts>
                            <outPort>samplereader_out</outPort>
                        </outPorts>
                        <params>
                            <param   pid="srcAddr">192.168.0.2</param>
                            <param  pid="srcPort">2222</param>
                        </params>
                    </component>
                    <component   cid="SampleMonitor0">
                        <hostAddr>192.168.0.4</hostAddr>
                        <hostPort>50000</hostPort>
                        <instName>SampleMonitor0.rtc</instName>
                        <execPath>/home/daq/MyDaq/SampleMonitor/SampleMonitorComp</execPath>
                        <confFile>/tmp/daqmw/rtc.conf</confFile>
                        <startOrd>1</startOrd>
                        <inPorts>
                            <inPort      from="SampleReader0:samplereader_out">samplemonitor_in</inPort>
                        </inPorts>
                        <outPorts>
                        </outPorts>
                        <params>
                        </params>
                    </component>
                </components>
            </daqGroup>
        </daqGroups>
</configInfo>
```

Fig. 14 An example of the configuration file in the case of fig.13.

To display the histogram on ‘OperatorPC’, execute the following on the said computer…

```
daq@OpeartorPC% xhost +MonitorPC
```

and then…

```
daq@OperatorPC% run.py -d OperatorPC:0 -c remote.xml
```

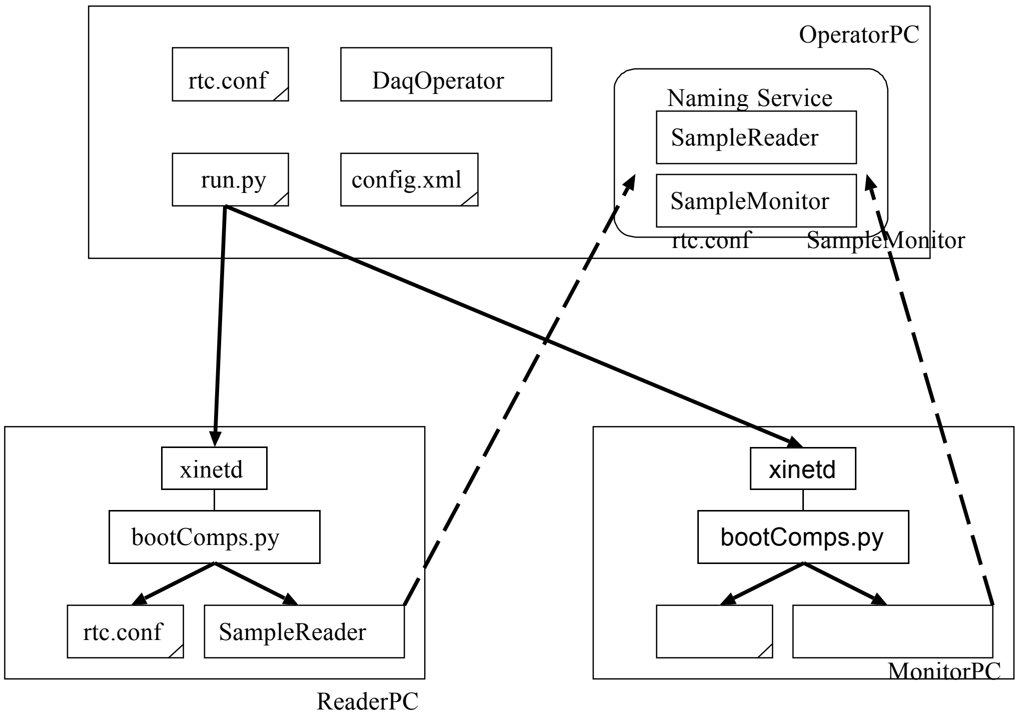


Fig. 15 The mechanism of remote boot

If the histogram is still not displayed, check if the X-Window communication has been permitted by executing, for example, the following…

```
daq@OperatorPC%  ssh  -x  MonitorPC  (small letter-x disables SSH  X  forwarding)
daq@MonitorPC%    DISPLAY=OperatorPC:0  xterm
```

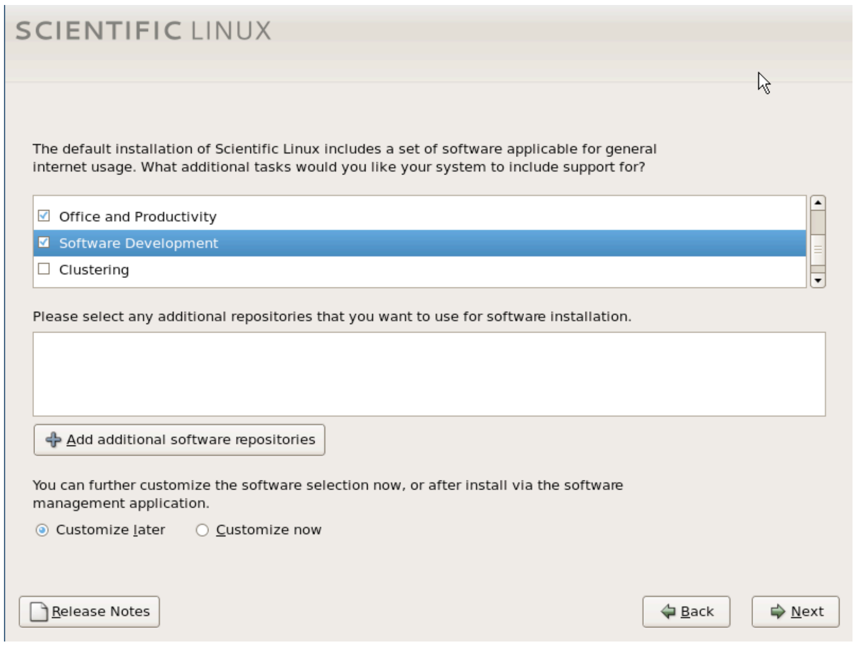
to confirm if the X-client of ‘MonitorPC’ can be displayed on the X-Window of ‘OperatorPC’, or try setting the argument of ‘-d’ to **IP address**:0 or etc.

E.6 Mechanism of remote boot

The mechanism of the remote boot is shown in Fig. 15. This figure shows the boot of the system where each of ‘DaqOperator’, ‘SampleReader’ and ‘SampleMonitor’ described in the previous Sections runs on different computers respectively.

1. The ‘run.py’ is started from the command line. The started ‘run.py’ parses the configuration file designated by its argument to know on which computer the DAQ component is to be started.
2. The ‘run.py’ accesses ‘Port 50000’ of the computer starting the DAQ component to issue a command to create ‘rtc.conf’. The ‘bootComps.py’ started by ‘xinetd’ receives this command.

3. The ‘bootComps.py’ generates ‘rtc.conf’ and notifies to ‘run.py’ if this was successful.
4. After ‘rtc.conf’ was successfully generated on each computer, ‘run.py’ subsequently issues a command to ‘bootComps.py’ via ‘xinetd’ to start the DAQ component. ‘config.xml’ designates the path for the component to start.
5. With reference to ‘rtc.conf’ generated on each computer, the started DAQ component finds the computer where ‘Naming Service’ is running and then accesses the service to register the information itself.
6. The ‘run.py’ subsequently starts ‘DaqOperator’. The ‘DaqOperator’, after its start, grabs the system by referring to ‘config.xml’ and asks the information of each DAQ component to the ‘Naming Service’. Then, it connects the DAQ components.


The default installation of Scientific Linux includes a set of software applicable for general internet usage. What additional tasks would you like your system to include support for?

- [✓] Office and Productivity
- [✓] Software Development
- [ ] Clustering

Please select any additional repositories that you want to use for software installation.

You can further customize the software selection now, or after install via the software management application.

[✓ Customize later] Customize now

Fig. 16 Selection of computer tasks

Appendix F How to install Scientific Linux

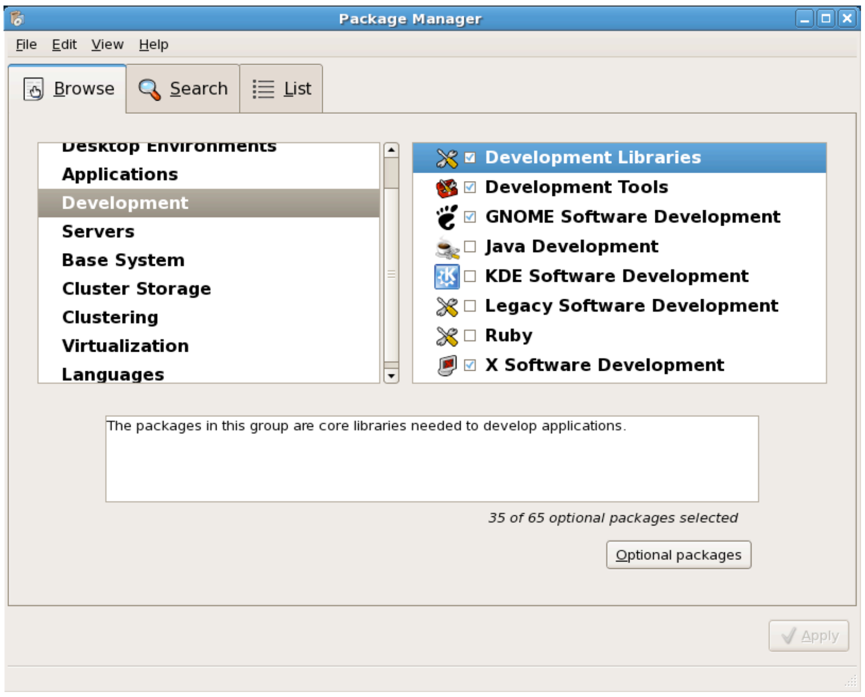
The web site of Scientific Linux is 'http://www.scientificlinux.org/'. You can obtain the installation CD or DVD image from this site, or the following Japanese mirror sites.

- http://ftp.riken.jp/Linux/scientific/ (Riken)
- http://ftp.ne.jp/Linux/distributions/scientificlinux/ (KDDI R&D Laboratories)
- http://ftp.jaist.ac.jp/pub/Linux/scientific/ (JAIST)
- http://reflx1.kek.jp/scientific/ (KEK (accessible only internally within KEK))

If you follow the GUI guidance, the installation should not be complicated. Just in case, we describe below, which packages need to be installed for the DAQ component development.

F.1 New installation (SL 5.x)

In the course of installation, the screen concerning the operations needed will appear on your machine. When the 'Software Development' is checked (Fig. 16), almost all the necessary packages will be installed. To use 'WebUI', 'mod_python' is required. So, after the installation of Scientific Linux, execute the following to install it.



Desktop Environments
Applications
Development
Servers
Base System
Cluster Storage
Clustering
Virtualization
Languages

- ✓ Development Libraries
- ✓ Development Tools
- ✓ GNOME Software Development
- ☐ Java Development
- ☐ KDE Software Development
- ☐ Legacy Software Development
- ☐ Ruby
- ✓ X Software Development

The packages in this group are core libraries needed to develop applications.

35 of 65 optional packages selected

Fig. 17 Addition of development packages on SL 5.x

```
root# yum install mod_python
```

Also, in Scientific Linux 5.x, the above package selection **will not install** ‘Emacs’ as your editor. If you need it, execute the following to install it, after the system installation.

```
root# yum install emacs
```

For the settings of SELinux and ‘iptables’, see Section F.5.

F.2 Adding the development environment later (SL 5.x)

If ‘Gnome Desktop’ is already running but any development environments (such as gcc, make) are not yet installed, click the icon at the left end of the menu bar and launch “Add/Remove Software” from the menu. Then, check the followings among “Development” and apply them (Fig. 17). For the addition of ‘mod_python’ and ‘Emacs’, see the previous Section.

- Development Libraries
- Development Tools
- GNOME Software Development

- X Software Development

For the settings of SELinux and 'iptables', see Section F.5.

F.3 New installation (SL 6.x)

During the course of installation, the screen of the main operations appears. When the 'Software Development' is checked (fig. 18), almost all the necessary packages will be installed. Since 'libuuid-devel' package necessary for the compile of the DAQ components has not yet been installed, execute the following to install it.

```
# yum install libuuid-devel
```

To use 'WebUI', 'mod_wsgi' is required. So, after the installation of Scientific Linux, execute the following to install it.

```
root# yum install mod_wsgi
```

Also, in Scientific Linux 6.x, the above package selection **will not install** 'Emacs' as your editor. If you need it, execute the following to install it after the system installation.

```
root# yum install emacs
```

For the settings of 'SELinux' and 'iptables', see Section F.5.

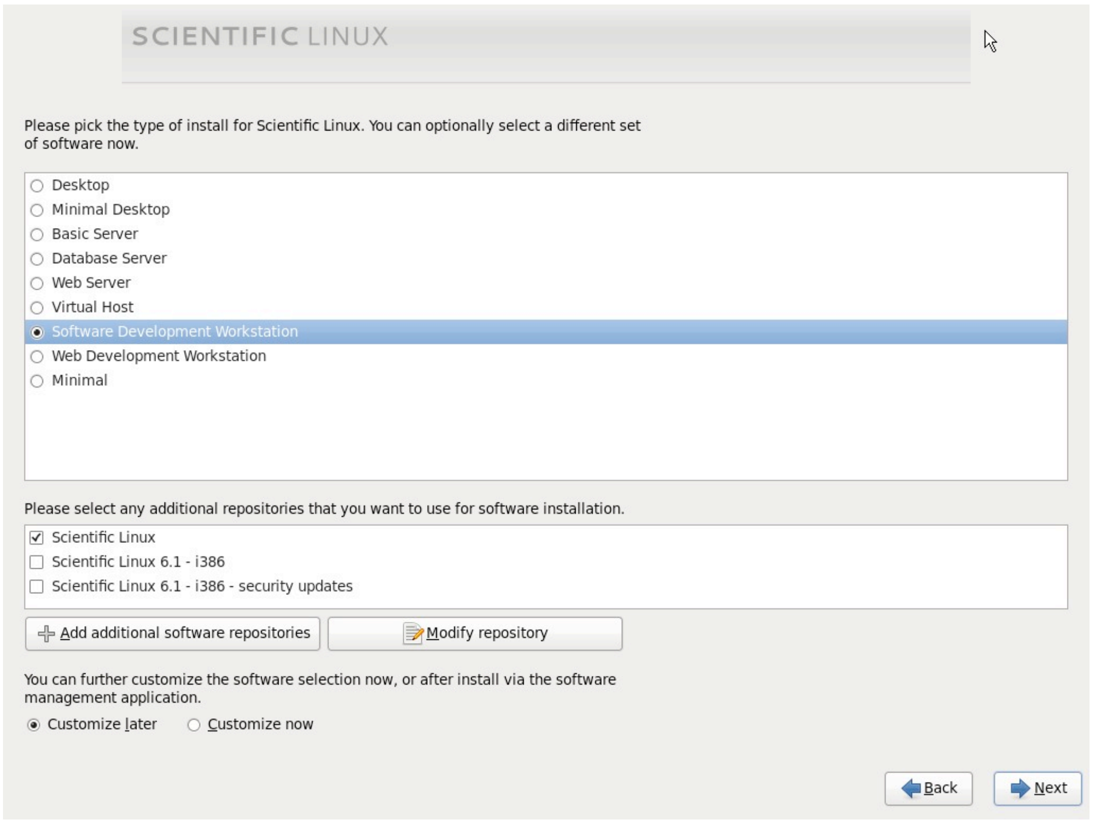
F.4 Adding the development environment later (SL 6.x)

If 'Gnome Desktop' is already running but the development environments (such as gcc, make) are not yet installed, use 'yum' command to install 'Development tools' and 'Additional development' group [*16]. Further, 'libuuid-devel' package is required.

```
root# yum groupinstall 'Development tools'
root# yum groupinstall 'Additional development'
root# yum install libuuid-devel
```

For the addition of 'libuuid-devel', 'mod _wsgi' and 'emacs', see the previous Section.
For the settings of 'SELinux' and 'iptables', see Section F.5.

[*16] Selecting these groups on GUI seems to be impossible.



Please pick the type of install for Scientific Linux. You can optionally select a different set of software now.

- ○ Desktop
- ○ Minimal Desktop
- ○ Basic Server
- ○ Database Server
- ○ Web Server
- ○ Virtual Host
- ● Software Development Workstation [✓]
- ○ Web Development Workstation
- ○ Minimal

Please select any additional repositories that you want to use for software installation.

- ☑ Scientific Linux
- ☐ Scientific Linux 6.1 - i386
- ☐ Scientific Linux 6.1 - i386 - security updates

You can further customize the software selection now, or after install via the software management application.

[✓ Customize later] Customize now

Fig. 18 Selection of computer tasks on SL 6.x

F.5 Setting up ‘SELinux’ and ‘iptables’

Since ‘WebUI’ will not operate with ‘SELinux’ enabled, it needs to be disabled.
To check if ‘SELinux’ is enabled, use ‘getenforce’.

```
%   getenforce
```

When enabled, a message shows ‘Enforcing’ or ‘Permissive’. When disabled, it shows ‘Disabled’.

To disable it, rewrite the line starting with ‘SELINUX=’ in ‘/etc/sysconfig/selinux’ as follows and then reboot the system.

```
SELINUX=disabled
```

With the ‘iptables’ packet-filtering, the transmission and reception of the packets required by the DAQ-Middleware may not be possible. To avoid this problem, it should be simpler to stop the ‘iptables’ packet filtering.

To confirm the ‘iptables’ currently packet-filtering, execute the following.

```
root# service iptables status
```

When ‘iptables’ is operating, its packet filtering rule will be displayed. When not operating, it displays as follows.

```
iptables: Firewall is not running.
```

When the packet filtering is enabled, execute the following to disable it.

```
root# service iptables stop
```

To make it always disabled after any reboots, execute the following.

```
root# chkconfig iptables off
```

Appendix G To command from command line to DAQ Operator

As the tool, other than the web browser, to issue a command to the DAQ Operator via ‘http’, a command line tool using ‘Python’ is provided. The ‘/usr/bin/daqcom’ is its executable file. Simply executing ‘daqcom’ shows its help message. In SL 5.x, you can obtain the state (of the http communication with DAQ Operator) by executing the following.

```
daq%   daqcom   http://localhost/daqmw/operatorPanel/   -g   state
```

For SL 6.x, the URL has been changed to the following.

```
daq%  daqcom  http://localhost/daqmw/scripts/  -g  state
```

For more details, see the reference [4].

References

[1] DAQ-Middleware Home page http://daqmw.kek.jp/

[2] DAQ-Middleware 1.1.0 Technical Manual, June 2011 (DAQ-Middleware 1.1.0 技術解説書、2011 年 6 月),
http://daqmw.kek.jp/docs/DAQ-Middleware-1.1.0-Tech.pdf

[3] Yoshiji Yasu, Hiroshi Sendai, Condition database development manual, 3rd August 2010 (安芳次、千代浩司、Condition データベースの開発マニュアル、2010 年 8 月 3 日),
http://daqmw.kek.jp/docs/ConditionDevManual-1.0.0.pdf

[4] Python command manual for DAQOperator operation, April 2012 (DAQOperator 操作用 Python コマンドマニュアル、2012 年 4 月),
http://daqmw.kek.jp/docs/PythonCommandForDAQMWOperation.pdf