Event Building in a 4-Floor NESTOR Detector

G. Bourlis , A. Leisos, S. E. Tzamarias and A. Tsirigotis

Particle and Astroparticle Physics Group
School of Science and Technology
Hellenic Open University

Abstract

This note describes the Event-Building algorithm as well as the program that includes the previous algorithm and simulates the DAQ System of a 4-floor NESTOR detector. The purpose of this simulation is to examine if the Data Acquisition functions are fast enough and accurate in the extreme case where the Acquisition rate is maximum i.e. 3 kHz per floor.

1. Introduction

The NESTOR detector [1,2], which was successfully deployed in 2003, consisted of one floor with 12 photomultipliers (PMT) – 6 up-looking and 6 down-looking. Each of the PMTs detects optical photons emitted as a muon passes by the detector. The possibility to detect a photon that has been produced by the $^{40}$K or from tiny organisms (bioluminescence) also exists. The detector has been designed to collect the data from the PMTs and transmit them to the shore only if it detects a certain number (configurable) of PMT pulses within a certain time-window (also configurable). In such a case the detector is said to have triggered and an event has been detected [3,4]. The electronics (that perform the data collection, triggering, digitization, monitoring, data packaging, data transmission etc.) of the detector lie within a Titanium sphere at the centre of the floor and will be referred to hereafter as the Floor Board. The data are transmitted to shore (through optical fibers) where an ISA board plugged on the DAQ System receives the data and “passes” it to the DAQ System. This board (the Shore Board) is actually the interface between the optical link and the DAQ System and the control signals from the shore station to the detector are transmitted through it as well.

A 4-floor NESTOR detector will consist of 4 floors, each one functioning independently. The data will be transmitted to the shore and received by the DAQ System. The DAQ System should decide whether these data correspond (or might
correspond) to a muon passing by the detector. The same approach as for one single
floor will be adopted. The timestamps (corresponding to the time of detection) of the
(floor) events will be examined and a global trigger will occur if a certain
(configurable) number of them fall within a certain time-window (configurable
again). In that case a global event has been detected.

2. Architecture

The Floor Board [5] of the NESTOR detector, deployed in 2003, is responsible for
collecting the data from each PMT, performing the trigger and, in the case of a trigger
formation, digitizing the PMTs’ signals. Then, it transmits them to the shore after the
appropriate packaging. The Floor Board is also assigned the task of reading the
measurements of the environmental sensors of the detector (temperatures, humidity,
pressure, etc.) and embedding this information within the data packets transmitted to
the shore. The various control signals (e.g. PMT thresholds and high voltage, majority
trigger level, coincidence time window, controls signals to the calibration system)
transmitted from the shore station to the detector are also handled by the Floor Board.

On the other edge of the optical link lies the Shore Board which is serving the
communication requirements between the DAQ System and the detector and which
does not perform any operations on the data.

It is obvious that for the 4-floor NESTOR detector there will be 4 Floor Boards and
4 Shore Boards corresponding to each floor (Figure 1). The DAQ System will
communicate with each Floor Board in both directions by transmitting control signals
and receiving data through the respective Shore Board. The software running on the
DAQ System will support:

- controlling of the acquisition,
- controlling and configuration of the detector,
- controlling of the electronics,
- controlling of the Calibration System,
- monitoring of the detector performance and current status and
- logging of the run history,

just as the DAQ System used to for the 2003 deployment.
It should be noted that the old DAQ System receives data from the Floor Board and saves it without performing any operations on it. Therefore, the Floor Board used to be the only component to decide which data should be saved. Within the new architecture each floor runs independently (just like the 2003 floor) and the floor’s Floor Board transmits the data to the respective Shore Board. The data from the 4 floors should be combined by the DAQ System and those that correspond to global triggers should be saved. Therefore, the new DAQ System should also:

- combine the data transmitted from each of the Floor Boards to detect the global triggers (Event Building),
- store the data corresponding to global triggers and discard the rest (Data Packing),
- allow the effective monitoring of each of all the 4 floors performance and status (Fast Monitor).
In parallel, the new Shore Boards should allow (just like the old one did):

- downloading the configuration files of the FPGAs and the PLD of the Floor Boards,
- broadcasting the 40MHz common clock to the 4 Floor Boards,
- transmitting data to the Run Control System of each Floor Board and receiving data from each Floor Board and
- production of the event samples.

The task of combining the data from each floor, identifying the global triggers, packing the data that correspond to each global trigger and discarding the rest, and generating the sample files for the Fast Monitor, that would have to be assigned to the new DAQ System, turns out to be really tedious and requires a great deal of computing power and memory. The only alternative is to add functionality to the Shore Boards to “assist” the DAQ System and to allow it to function more efficiently. Therefore, since the only information that the DAQ System needs in order to decide whether a global trigger has occurred is each data packet’s timestamp (corresponding
to the time that the floor trigger occurred), each Shore Board on reception of a data packet extracts the timestamp from the data packet, passes it to the DAQ System and temporarily stores the data packet. The DAQ System combines the timestamps from each floor and identifies data packets that correspond to global triggers (Event Building). Then, it asks from each Shore Board the corresponding data packets. The Shore Boards pass the requested data to the DAQ System and delete the rest. In addition, the new Shore Boards are designed so as to extract the environmental information from each received data packet, average them over a certain time period or number of received data packets (since there is no point in having the environmental information at time intervals less than one or a few seconds) and hand the averaged values to the DAQ System when asked to. Therefore, the new Shore Boards should also accomplish:

- data stripping and
- production of event samples.

**Figure 2**

*The DAQ system interface to the optical link*

It is obvious that the new Shore Board (currently under development) should feature computational capabilities, sufficient memory and fast communication with their hosting computing machine. Figure 2 is a schematic representation of the new Shore Board configuration. The Shore Board consists of the Receiver – Transmitter Board (interface to the optical fibers) and a PCI card, which has a built-in FPGA (for the implementation of the required functionality that will be described in detail shortly) and a RAM for the temporary storing of data.
Figure 1 does also present the basic data flow between the DAQ System and each Shore Board. In brief, each Shore Board receives the data packets, extracts the timestamp, hands it to the DAQ System and stores the data packet. The DAQ System combines the floors’ timestamps and on detection of a global trigger requests the data packets that participate in the trigger from the corresponding Shore Board. The Shore Boards transmit the requested data packets and delete them along with the data packets that don’t participate in any trigger. On regular intervals the DAQ System does also request sample files for the monitoring of the detector status. Finally and on regular intervals as well, the DAQ System requests the extracted and averaged environmental information from each Shore Board.

3. Simulation

In order to identify the critical parameters of the whole DAQ System and configure it for the most efficient and bug free performance, extensive simulation and testing of the operation of the whole system is required. The first approach turns out to be a fully software one. This means that no Shore Boards are used but instead programs that simulate the functionality of the Shore Board and communicate with the DAQ System. Due to the fact, that the performance of the DAQ System as far as speed and memory usage is concerned need to be defined, the DAQ program should left run alone on a server computer. The program (or programs as will be argued shortly) that will be used to simulate the Shore Board functionality will run on separate server computers. The communication of the programs will be achieved through a LAN.

The simulation package consists of three major programs, namely the server program, the warehouse and the ram program. The server simulates the functions of the DAQ System relevant to data receiving (from the Shore Board), event building and data storing, as well as to producing files for the Monitor System. The warehouse and the ram programs simulate the Shore Board, excluding the role of the Shore Board as an interface to the optical links of course. The ram program simulates those functions of the Shore Board that can be summed up to the extraction of the timestamp from the data packets, their (the data packets) storing in the Shore Board memory (RAM) and the handing of the timestamp to the DAQ System. The warehouse program communicates with the server program receiving requests to transmit and transmitting those data packets that participate in a global trigger. The
server program also requests from the warehouse program, at regular intervals, the environmental data that have been extracted from the data packets and averaged for a certain number of events or a certain time period.

A straightforward question would doubt the choice to use two programs (ram and warehouse) for the simulation of the Shore Board instead of a single one. The avoidance of the implications of using multiple data sockets for the communication of the programs over the LAN provides the reasoning. A schematic representation of the basic data flow between the three programs is, therefore necessary and that is what Figure 3 hopefully achieves. It should be noted that although for a 4-floor detector there are 4 Shore Boards, only one ram and warehouse program are used that do transmit to the server the data that all 4 Shore Boards would.

![Figure 3](https://example.com/figure3.png)

*Figure 3*

*Data flow between the server, the ram and the warehouse program*

During the development of this package it has been taken into account that:
- each floor transmits data packets with increasing timestamps, but
- there is no guarantee that data packets from different floors will arrive according to their timestamps,
- the time-window used for the global trigger formation (which is configurable as already mentioned) is selected according to the detector dimensions.

The data packets used during the testing of the package contain timestamps and some sample data of the correct size, since only the timestamp is necessary for the needs of this simulation. The timestamps have been produced using the random generator of the CERN Lib. They follow an exponential distribution with the additional restriction that they can't be less than what the design limitations of the
Floor Board impose. As for the environmental information, a small number of real data packets are used for their extraction and averaging.

All programs have been developed with the Linux g++ 3.2.0-5 compiler. The programs have been installed on 3 different dual-CPU servers, because the server program’s performance needs to be tested and its limitations defined. The servers are connected on a 1Gbps Local Area Network.

4. The Server Program

The server program’s flow diagram is presented in Figure 4. The program starts by opening 2 log files; the EventLog.dat file where all pairs of floor numbers and timestamps are stored and the EventSave.dat where information about the global events detected is stored (actually a global event count and the pairs of floor numbers and timestamps that participate in the global event). Then, it establishes the connections to the ram and the warehouse programs. There are two connections to the warehouse program used by the Event-Building algorithm to send requests to and receive data from. The connection to the ram program is the one that the server program listens to and that determines the subsequent actions. In particular, if the server program gets 8 bytes of data (which means it received a timestamp), it goes on to read 1 byte (which is the respective floor number) and then executes the Event-Building algorithm. If the server program gets 4 bytes of data it transmits a ‘q’ standing for quit to the warehouse and then stops execution (it received an ‘exit’ command). Of course, this is used for the simulation needs only to end the simulation, since the “real” DAQ System never receives an ‘exit’ command from the Floor Boards. It would just be turned off by the operator.

---

1 The server program will be integrated in the NESTOR DAQ System and will have to efficiently handle the data received from 4 Shore Boards and store a quite large amount of data, so it has to be tested having available all the CPU power, as it has already been mentioned.

2 The Floor Board encodes the timestamps in 60 bits. 8 bytes equal to 64 bits and therefore are sufficient to hold the timestamp.
After executing the Event-Building algorithm, the program checks whether to request environmental data. If a certain period of time has elapsed (configurable, typically 1 second) since the last acquisition of environmental data, the program runs a subroutine (*GetEnvironmentals()*) that requests from the warehouse program the environmental data, gets it and saves it to 4 files (corresponding to the 4 Floor Boards) that are used by the Monitor System. It should be noted that the data packets that each Shore Board receives from the respective Floor Board of the NESTOR detector contains environmental data (temperatures, humidity etc.). Due to the fact that data packets are transmitted in high rates and the environmental variables change slowly, there is no point in monitoring all the environmental data extracted from each the data packets. Getting the values of the environmental variables every one second or so seems reasonable. Therefore, each Shore Board on reception of the data packets, extracts the environmental data, averages their values over a number of received data packets (or over a time period) and transmits the averaged values to the DAQ System which stores the averaged values in files (one for each Shore Board). Subsequently, the Monitor System reads these files and monitors the environmental data.

*Figure 4*

*Server program flow diagram*
5. The Event Building Algorithm

The Event-Building algorithm takes as input timestamps, the corresponding floor numbers and the floor majority (2, 3 or 4) and decides which of the data packets transmitted from the Floor Boards, form a global trigger. It is also possible to run the algorithm with floor majority equal to 1, in which case the timestamps are just checked as to whether at least 2 fall within the time-window, irrespectively of the floor numbers (even timestamps from the save floor can form a trigger). This option can be traced back to the early stages of development of the algorithm and has survived all revisions because it supports testing needs.

The Event-Building algorithm utilizes a small array of records consisting of the timestamp, the respective floor number (0, 1, 2 or 3) and a flag to be used for the trigger formulation and during the process of saving the selected data packets. This array will be referred to as the buffer and its entries as events (pairs of timestamps and floor numbers). Events are stored in the buffer in ascending order of the timestamp. Three “pointers” (actually array position indicators) are used throughout the algorithm. The buffercurrent pointer is the first unoccupied position within the buffer. The firstcheck and the lastcheck pointers are used by the subroutine (checkevents()) that identifies the global triggers. firstcheck points to the first of the events in the buffer that hasn’t been completely checked for its participation in a global trigger. lastcheck is the last of the events in the buffer that can be checked for contributing to a global trigger.

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>12578</th>
<th>12602</th>
<th>12658</th>
<th>12845</th>
<th>12882</th>
<th>12990</th>
<th>13067</th>
<th>13256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floor no</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Flag</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5

An instance of the buffer and the three “pointers”

3 The required number of timestamps within the time-window.
4 If an event is checked its flag is changed accordingly but the possibility that an event participates in more than one global trigger exists.
5 Can be checked means that there is no chance of getting a new timestamp smaller than this event’s timestamp, in which case the check should be carried over again.
Another array (maxtimestamp[FLOORS] where FLOOR is the number of floors) that holds the maximum timestamp per floor is used, as well as an array (lookupmajority[16]) that is used by the subroutine checkevents() during the process of identification of the trigger. Additionally, a number of counters (eventcountperfloor[FLOORS], eventsavecount, bigeventcount) that mainly serve testing and statistics purposes are used.

Figure 6 depicts the Event-Building algorithm flow diagram. After the algorithm gets the timestamp, floor number and floor majority required, it updates the floor’s maximum timestamp (maxtimestamp array) and runs the inserttobuffer() subroutine that (as its name implies) inserts the timestamp – floor number pair in the buffer. The input data are also written to the event-log file (EventLog.dat). Then, a simple little routine (namely minmaxtimestamp()) is called that calculates the minimum of the 4 floor’s maximum timestamps. If the minimum did not change the algorithm waits for the next set of timestamp floor number and majority. If, on the other hand, the minimum of the 4 floor’s maximum timestamps has changed, the position of the new minimum within the buffer is located (lastcheck pointer). Then the program checks (see the checkevent() routine description that follows) whether there are events that form a global trigger and flags those events accordingly.

6 Of course, when running the algorithm the same floor majority is requested, the one defined by the DAQ System. However, the algorithm does work properly even if for successive sets of timestamps and floor numbers the floor majority is different, which is helpful during the testing of the program.

HOU-NS-TR-2004-12-EN
The checkevent() routine

The checkevent() routine takes as input a “pointer” (called last) to a position of the buffer between the firstcheck and the lastcheck and the floor majority required. It then checks, for all the events at positions from firstcheck to last, if their timestamps difference is smaller than the time-window. If this does not hold the firstcheck pointer is increased by 1. If it holds the number $2^i$, where $i$ is the floor number (0, 1, 2, 3), is added to a temporary variable (named tetraple). After scanning all the events in the above-mentioned range the number of 1’s of the tetraple variable (in binary form of course) equals to the number of floors which events contribute to the global trigger. Therefore, if this number is greater or equal to the requested floor majority, the respective events must be flagged accordingly. In fact, the program performs an OR operation with 2 (10 in binary) to the event’s flag if it is the first (in time) that contributes to the global trigger and with 1 (01 in binary) for the rest. The reason for this will be explained within the description of the way the Event-Building algorithm saves the data.

The Event-Building algorithm (every time that goes on to perform checking) calls the checkevent() routine successively, assigning to the last pointer (checkevent() routine input) all the values from firstcheck to lastcheck. This repeated search is done because different global triggers might intersect, in which case they have events in common and must be identified as distinctive ones.

The checkforsave() routine

The checkforsave() routine is called by the Event-Building algorithm each time it concludes the events checking. The routine starts from the zero position of the buffer and checks the flags of the events until the one at position (firstcheck – 1). If the flag is found to be equal to 2 (10 in binary), the event corresponds to the first event of a new global trigger. The bigeventcount is updated (increased by one) and a new file (named big_event_x where x is equal to bigeventcount) is created. Then, if the flag is not zero, the savetofile() routine is called with input the timestamp and the respective floor number. It should also be noted than in the case of overlapping global triggers (successive global triggers with common events) the data of both triggers are saved in the same big_event_x file. In such a case the first event of the first global trigger will be flagged with 2 (10 in binary), the following (if any) that belong to the first trigger.
with 1 (01 in binary), the first event of the second trigger which will also belong to the first with 3 (11 in binary) and the rest with 1 (10 in binary). Therefore, no new file is opened for the second of the global trigger, since this happens only in the case of coming across an event flagged with 2. Finally, the \texttt{checkforsave()} routine calls the \texttt{updatebuffer()} routine.

\textbf{The savetofile() routine}

The \texttt{savetofile()} routine writes the timestamp and floor number to the connection with the warehouse program and then reads the data that the warehouse returns (the corresponding data file). Afterwards, it writes the data to the \texttt{big\_event\_x} file. Before, returning the routine increases by one the \texttt{eventsavecount}.

\textbf{The updatebuffer() routine}

The \texttt{updatebuffer()} routine is called by the \texttt{checkforsave()} routine after it has checked which events to save and has saved them. The routine just moves all the entries in the buffer array to the left so that \texttt{firstcheck} (the first not fully checked event position) becomes 0 (the first position within the buffer).

\section{6. The Warehouse Program}

The warehouse program is quite simple, since it only gets commands to transmit the data packets or the environmental data and does so. The flow diagram of the program is presented in Figure 7. The program starts by reading from some files a number of data packets and environmental data. It establishes two connections with the server program and waits until the server program writes something to the connection. The server program might write to the connection a timestamp (8 bytes) followed by a floor number (4 bytes), requesting a specific data packet or a simple integer (4 bytes), requesting the environmental data. It might also right a ‘q’ (1 byte) forcing the warehouse program to exit. Therefore, the warehouse reads the connection and on reception of 8 bytes (meaning a timestamp) waits for 4 more bytes (floor number) and then transmits the data packet. Normally, it should transmit the data packet that corresponds to the particular timestamp – floor number pair, but for the needs of the simulation any transmission of a data packet of the correct size is sufficient. The same holds for the environmental data transmission, which is initiated in case the warehouse reads 4 bytes from the respective connection. It should be noted
that all the needed data are loaded to memory at the beginning of execution of the program, since the objective is to test the server program efficiency and performance and therefore, data need to be sent as quickly as possible.

Moreover, the data packets are sent into 40 blocks of 128 bytes each, while the environmental data are transmitted in one block of 64 bytes (2 bytes x 32 ADCs). This is so because the machines used to test the programs are not of the same architecture (native Linux RedHat 9 and VMWare V. 4.0.0 build-4460 running Linux RedHat 9 was used) and the speed of communication over the LAN was found to be higher for the above choice of size of the transmitted data. If the data are transmitted into smaller packets the speed decreases significantly, while in the case of transmission of all the data in one-packet results in server program’s failure to read the packet. Of course, this is no issue for the NESTOR DAQ System, since all the Shore Boards will be cards plugged on the PCI bus of the System.

7. The Ram Program

The ram program has been initially designated so that its functionality would be very similar to the functionality of the 4 Shore Boards. It includes 4 RAM arrays, the elements of which are sets of timestamps (8 bytes) and data packets (5120 bytes). There are also 4 FIFO implemented as double linked lists and corresponding to the 4 RAMs. Each element in the FIFOs is a set of a timestamp, the RAM address (RAM

Figure 7

Warehouse program flow diagram
array index) where the respective event has been stored and the pointers to the next and the previous entry in the list. The function that handles the RAM arrays and the FIFO lists are initializeRam(), insertToRam() and DeleteFromRam() with straightforward functionality. Adopting this approach the ram program would get the data packets (e.g. by reading a file), extract the timestamps, store the timestamp and data packet in the RAM array and insert the timestamp and RAM address in the FIFO. It would then send the timestamp and floor number to the server program and would go on. At some point the server program would communicate to request the data packet corresponding to some timestamp and floor number. The ram program would go to the respective FIFO and search for the timestamp, it would go to the, specified in the FIFO, RAM address, get the data packet and send it to the server program. It would then delete from the FIFO list and RAM array all entries with timestamp smaller (and equal) than the timestamp requested.

However, this approach was somehow abandoned because, while in hardware these functions could be implemented in parallel, it would be very difficult to maintain a steady more or less speed of transmitting time-stamps to the server program during the simulation. Thus, the ram program has been implemented to fill the 4 RAM arrays with a certain number of timestamps\textsuperscript{7} read from a file and to transmit those time-stamps and the respective floor numbers to the server program at regular intervals. But the ram program does not accept any request to transmit the data; instead these requests are directed to the warehouse program.

The ram program flow diagram is depicted in Figure 8. The program starts by initializing the RAM arrays and the FIFO lists. Subsequently, 4 timestamp files (floor1.dat, floor2.dat, floor3.dat and floor4.dat) are opened and read and the timestamps are entered to the respective RAM arrays and FIFO lists are updated accordingly. Following is the establishment of the connection to the server program.

Afterwards, the ram program enters a while loop transmitting timestamps to the server program. A number of variables need be explained here. Firstly, events stands for the number of timestamps that will be transmitted to the server for each floor (typical value 10000). If it is necessary to run the programs for longer time the variable iloop with typical value 1 is set to a higher value, in which case the same

\textsuperscript{7} The RAM arrays are filled with events of size 1 byte long, so that the previous functionality is maintained for the future but not speed down the program execution.
timestamps are re-sent as many times as \textit{iloop} indicates. In addition 4 variables that hold the number of timestamps transmitted per floor are used (\textit{ievent(x)}). Therefore, when entering the while loop the ram program produces a random integer \( x \) between 1 and 4 (corresponding to floor number). Then, if not all timestamps for the particular floor have been transmitted (the \textit{ievent(x)} variable is less than the \textit{events} variable), the next timestamp in the RAM array and the floor number are written to the connection and \textit{ievent(x)} is increased by 1.

When all the timestamps for each floor have been transmitted the program exits or in the case that \textit{iloop} is set to a value higher than 1, the whole set of timestamps is transmitted repeatedly as many times as \textit{iloop} indicates.

An empty for-loop is used just before the transmission of every timestamp. The number of loops is controlled by the variable \textit{total}. The delay caused by the for-loop is what actually determines the speed the server program receives timestamps and can be set in low values (high speed) so that the performance of the server program and of course the machine running the server program can be evaluated.

8. Conclusions

The programs have been tested mainly with respect to the Event-Building algorithm and the performance of the machine running the server program.
For the needs of testing the Event-Building algorithm, the produced random timestamps have been enhanced or changed, so that a certain number of 2, 3 or 4 timestamps can form a global trigger. In addition, timestamps have been changed so as to correspond to overlapping global triggers. In all cases the Event-Building algorithm recognised the global triggers and saved the data according to our wills.

As for the server program the test showed that approximately 90% of the total time is spent saving data files, which indicates that a reliable system to host the DAQ System should feature very short disk access time. As for the highest data packets transmission rate from the Floor Board, no conclusions can be drawn since the server program is only a part of the DAQ System. Changing the parameters in the ram program so that it sends data packets to the server program at a rate of 3kHz (per floor) resulted in absolutely no change in the server program performance. Attempts to find an upper limit are not feasible with the current testing configuration, since if transmitting data packets with a very high rate the server program would need more time to save the data, thus delaying to read the ram connection and decreasing the rate again.

Further development to the Event-Building algorithm should concentrate on the cases that wrong timestamps are received by the program, one floor stops transmitting for a short period of time and on the way the program gets the timestamps (one at a time, groups of certain size, etc.).

Acknowledgments

The authors of this note wish to thank the members of the NESTOR collaboration, the stuffs of the NESTOR Institute as well as the academic and technical personnel of the School of Science and Technology of the Hellenic Open University for their help, scientific, technical and financial support.

References


