PROPOSAL OF THE DAQ SYSTEM FOR THE BM@N PROJECT

A.Yu.Isupov[a,1], V.P.Ladygin[a], S.G.Reznikov[a]

a) JINR, 141980, Dubna, Moscow region, Russia

Abstract

The proposal of the DAQ architecture for the BM@N setup is described. The requirements to the DAQ, Trigger, and Slow Control hardware suitable for this architecture are issued. The DAQ software will be responsible for the managing of the experimental data transportation and processing. This software implementation is proposed on the base of the *ngdp* (for NetGraph Data Processing) framework, which allows us to:

— use the already implemented OS kernel modules for the data streams handling and control (***netgraph(4)*** package);

— implement the DAQ specific kernel modules in the uniform object-oriented style, which provides the high scalability and the easy remote distribution;

— execute the ***netgraph(4)***-style code out of the regular scheduling scope (with the very high, practically realtime performance);

— reduce the data copying over the context boundaries;

— eliminate the intermediate storages on the media slower than memory.

1. Introduction

The Baryonic Matter at the Nuclotron (BM@N) project is approved in 2012 for the comprehensive technical design report preparation. The present proposal is written in the frame of these efforts. According to the BM@N Letter of Intent the BM@N setup will contain approximately 370000 channels of the front-end electronics (see Table 1). The Au beam intensity of 10^7 Hz and the Au target with 1% nuclear interaction length leads to the mean interactions rate 10^5 Hz. The detectors occupancy from this Table accounts (as the most severe case) the average event multiplicity 164 from the minimal bias collision into the 82.5 msr around the forward direction, excluding 1.4 msr around the beam direction (for T0 — 10^7 Hz Au). This assumption allows us to estimate the average single event size as 44100 bytes and leads to the up to 40 Gbit/s data flow. The transportation of this data bandwidth through the 10 Gbit/s Ethernet requires 4 or more (for 1 Gbit/s Ethernet — more than 40) such Ethernet channels. On the other hand, the whole dataset belonging to some physical event should at some processing stage to appear on the single computer for the full event building. This requirement is true for the each event. So the system should contain more than one Event Builder (EvB) in principle. This fact requires to solve the nontrivial questions related with the data streams organization and management.

The proposed technical design has potential to be used by the larger experiments like MPD, SPD. It seems reasonable to implement and debug both hardware and software components of this design for the BM@N setup to obtain experience with the corresponding technologies. Anyway, the large scale DAQ system design is inevitable step for the upcoming NICA experiments.

[1]E-mail: isupov@moonhe.jinr.ru

Table 1: Data flows estimations

Digitization types legend: T — time, Q — charge, A — amplitude, c — coordinate (=channel number, embedded if we use hptdc). Per each Ethernet frame 1500 bytes we have overhead: 40 bytes of *ngdp* packet header + 38 bytes Ethernet overhead = 78 bytes (~6%).

subdet	number of channels / number of stations / number of planes	digitization type	digitization time required, ns	required for precision / really acquired value, bit/channel (assuming hptdc)	dynamic range	detector occupancy (per event per plane)	synchronous trigger producer	synchronous trigger consumer	full data stream, bit per event / Mbit per second (w/o 6% overhead)	channels per board × boards per subdet
T0	64/1/2	T+Q	≤?30	/64×?3 + 32, 16(?8)		100/32	yes	yes	15584+1024 (512) / 1661 [a]	32 × 2
ZDC	300/1/1	A(?Q)	$<$120..150	/64(32)		1.0	?yes	yes	19200(9600) / 1920(960)	128 × 3
DTE	64/1/1	A(?Q)	$<$120..150	/64(32)		1.0	?yes	yes	4096(2048) / 410(205)	64 × 1..2
RPC	1200/1/1	T+Q(+c)	≤100	12+?(+0) / ?64(32)	0..64 ns	0.16	yes	yes	12288(6144) / 1229(615)	256 × 5..6
DC	4000/2/8	(c+)T	≤111	(11+)?/?32	0..128 ns	0.66	no	yes	84480 / 8448	256 × 16..20
ST	7000/1/3	(c+)T		(13+)?/?32		≤0.16	no	yes	35904 / 3590	256 × 28..30
Sum:	~16000								172580/17258	
IT (STS)	350000/4/2 (450000/5/2)	c+A		/ 24+8		0.016	no	could	180000(225280) / 18000(22528)	16384 × (3×8(10))
Sum:	~370000								352580/35258	

[a] 6% overhead included.

2. Overview

Logically the whole proposed DAQ system for the BM@N setup could be subdivided by the three big subsystems:

- the Data AcQuisition subsystem itself (Fig. 1);
- the Fast Control and Trigger (**FC+T**) subsystem, which intended to produce and handle all the synchronous signals in the setup (Fig. 2);
- the Slow Control (**SC**) subsystem (Fig. 3).

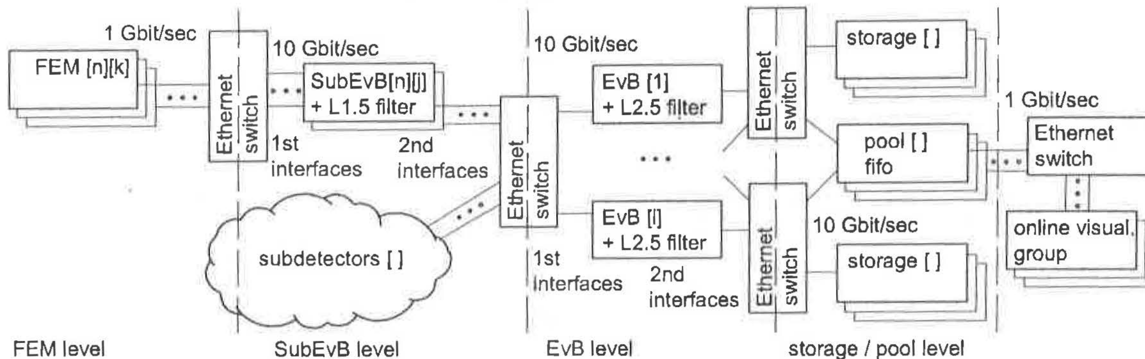


Figure 1: The levels of the DAQ system.
- `FEM[n][k]` — `k`-th entity of the `n`-th subdetector;
- `SubEvB[n][j]` — `j`-th computer belonging to the `n`-th subdetector;
- `EvB[l]` — `l`-th computer of the Event Builders level;
- `L1.5` — software filter on the SubEvB level;
- `L2.5` — software filter on the EvB level;
- `pool[]` — computers of the pool level;
- `storage[]` — computers of the storage level.

Of course, the same hardware card or board physically could contain parts of more than one subsystem due to the high electronics integration.

There are many reasons to functionally subdivide the hierarchy of the DAQ hardware units by the logical levels along the data stream, e.g.:

- to formalize and simplify the interactions between the DAQ elements,
- to provide for each subdetector the standalone working mode,
- to allow easy reconfiguration of the active subdetectors' set,
- to allow easy hardware replacement, etc.

We propose, that at least the four logical levels of the data processing (see Fig. 1) are needed:

- FEM level implements at least the queues of ready data fragments produced by the `Dig[][]` cards;
- SubEvB level — data preprocessing computers grouped by the subdetectors. SubEvB level requests the `L1[]` trigger packets and the corresponding ready data fragments from the FEM level, merges the sub-events, implements the queues of ready sub-events, and possibly the `L1.5[]` software filters for the sub-events rejection;
- EvB level – full events building computers. EvB level requests the `L1<t>` / `L2<t>` trigger packets and the corresponding ready sub-events from the SubEvB level, merges the full events, implements the queues of ready full events, and possibly the `L2.5` software filters for the full events rejection;
- pool level — data postprocessing computers. Pool level requests the ready events subset from the EvB level, converts their from a native binary format to some ROOT

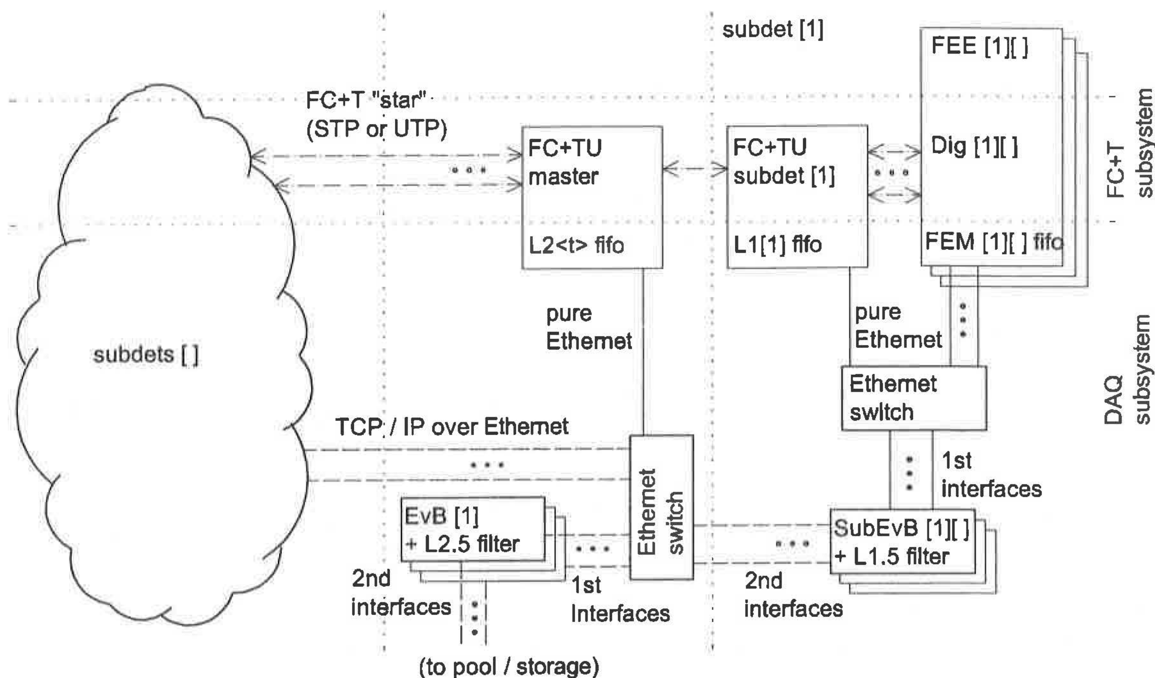


Figure 2: The Fast Control and Trigger (FC+T) subsystem.
• FC+TU — Fast Control and Trigger Unit card, the basic hardware unit of the FC system, which are master (1 per setup) and subdet[n] (1 per n-th subdetector);
• Dig[n][m] — m-th Digitizer card of the n-th subdetector;
• For other elements see legend in Fig. 1.
These elements are interconnected by the:
• dedicated lines of the trigger production ("ascending") path;
• dedicated lines of the trigger distribution ("descending") path;
• "pure" Ethernet lines;
• TCP/IP network over Ethernet lines.

[1] class representation, implements the queues of ROOT events, and provides their to clients for the online analysis and visualization. Also the ROOT events are histogrammed and these histograms are provided to clients for the online analysis and visualization;
• storage level (parallel to the pool level) — computers, which request the ready events from the EvB level and write their into the intermediate (HDD) storage. The storage level consists of some identical computer groups switchable while the data taking, so one group obtains the events from the EvB level while the other groups transfer the obtained data from the intermediate into the final storage, possibly slower than HDD.

Each intermediate level behaves as a server for the downstream level and as a client for the upstream level. This approach simplifies algorithms of inter–level interactions, which will be reduced to the ones only between neighbour levels.

In addition, some computer groups can be outside of the data stream:
• Fast Control and Trigger group — computers, which implement the FC+TU subsystem: trigger queues (see also chapter 4) and the corresponding GUI;
• Slow Control group — computers, which implement the SC subsystem (see also chapter 3) and the corresponding GUI;
• DAQ Operator group — computers, which perform the control over the DAQ software components and provide the user interface for them;

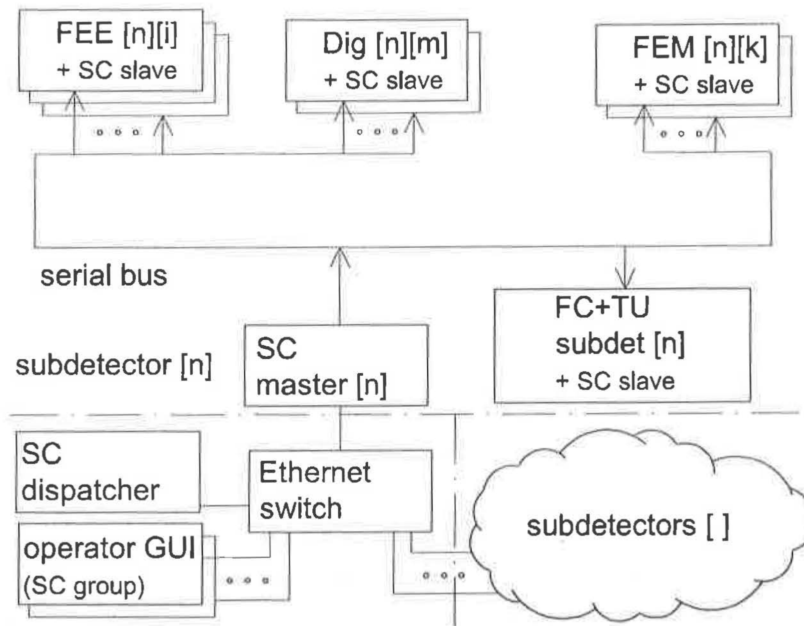


Figure 3: The Slow Control subsystem.

• FEE[n][i] — i-th Front-End Electronic card of the n-th subdetector, which prepares signals for the Dig[n][m] card;

• SC master[n] — Slow Control master card (1 per n-th subdetector), the basic hardware unit of the SC system;

• SC slave — peer part for the SC master (integrated into FEE, Dig, HV/LV, FC+TU cards);

• SC dispatcher — the computer (1 per setup), which obtains the operator GUI requests and forwards ones to the specific SC master[] according to their destination.

• online visualization group — clients of the pool level.

The requirements to the proposed levels are described in chapter 4. Lets also introduce some terms absent in Figs. 1–3 legends:

Packet — the sequence of bytes, which contains the fixed length packet header (with at least the ID string, type, timestamp, number, length, flags, CRC32) followed by the packet body of known length (see [2] and ***packet(3,5,9)***).

Event merging — simple kind of event building, which produces the new output packet to contain sequence of all the input packets (headers are preserved, the ID string reflects the encapsulation level). This operation performed on SubEvB and EvB levels leads to the full event's encapsulation level ≥ 2.

Trigger packet — the packet is produced by the FC+TU master or FC+TU subdet[] card and contains the reliable (sub)event merging tag (number and/or timestamp) for the (Sub)EvB.

Fragment, sub–event, full event — the packets represent the data and are produced by the Dig[][], SubEvB[][], EvB[], correspondingly.

Queue — the software buffer of packets with the FIFO (First In — First Out) nature and some rule set (discipline) of the packets getting.

L1[n], L1<t> / L2<t> **fifo** — queues of the L1[n] or L1<t> / L2<t> trigger packets supplied by the FC+TU subdet[n] or master card, respectively.

3. Hardware requirements

Here we describe requirements to the hardware proposed for development.

Currently we assume for the whole BM@N setup the L1 central synchronous ("live") single-stage trigger to be the only trigger implemented by the hardware. This hardware belongs to the FC+T subsystem. This approach solves the deadtime account issues in the straightforward and clear manner, and allows us to establish the reliable triggers (and *ngdp* data packets) numbering. Such numbering will be simple and single tag (however see also chapter 4) for the (sub)events merging: the data fragments / sub-events will be selected to belong to the same N-th sub-event / full event, if and only if their reliable numbers are match with (i.e. corresponding integers are equal to) the N-th trigger number. The overall architecture of the FC+T subsystem we can see in Fig. 2. With the L1<t> hardware trigger (possibly of some type t) the higher level triggers are assumed to be the software filters on the SubEvB (L1.5[]) and EvB (L2.5) levels.

The whole setup's L1<t> will be assembled from the subdetector's L1[n], while L1[n] — from the Dig[n][] card's preL1[n][] by the hierarchy of analog and/or digital summators (n corresponds to the trigger-producer subdetectors). This hierarchy could be implemented as parts of the FEE[][]s, Dig[][]s and FC+TU subdet[]s (see below), or as independent modules. Former option is preferable in respect of the power solution, cabling arrangement, SC implementation reasons (entia non sunt multiplicanda sine necessitate). The presence of the FC+TU subdet[] instances also allows to operate subdetectors easily in the standalone mode for test, calibration, etc. purposes (setup scalability providing). Currently the trigger-producer subdetectors are T0 (existence of the beam particle), RPC (number of fired pads as the multiplicity characteristic), and ZDC (possibly with DTE) (total energy deposition as the centrality characteristic). The signals sum is discriminated according to the preset trigger conditions by some controlled discriminators, possibly on the FC+TU master card.

The nominal (mean over time) event rate on BM@N equals to the interactions rate 10^5 Hz, i.e. 10 μs per event, during which we should provide the L1 trigger decision. Of course, the beam inhomogeneity in time tightens this requirement to $< 1..2$ μs. This requirement is reachable, if the FPGA processing (the main time consuming part of the trigger production) will be no longer than 0.5 μs (50 cycles at the 100 MHz). Note if we can use the trigger queues on the each involved digitizer chip type (hptdc has one of 16 triggers deep) and their possible trigger latency is > 10 μs (hptdc — up to 25..50 μs), this requirement could be softened.

The proposed FC+T hardware architecture is expected to support also the two-stage L2<t> triggers production. Possibly this will require the Trigger Protocol Messages (TPMs, see below) content expansions (e.g., to carry the reliable timestamps) and assortment additions, (sub)events merging tag change from the reliable numbers to timestamps, etc.

The FC+T subsystem is intended to produce and handle all the synchronous signals in the BM@N setup (L0 clock, L1<t> / L2<t> and L1[n] triggers, etc.), while the FC+TU cards are basic hardware unit of it.

The 40 MHz L0 clock is the single synchronization source for all setup elements — hptdc, FPGA, serdes of the TPMs transfer system, etc.

The **FC+TU master card** (1 per setup) should contain at least:

- 8..16 trigger (TPM) input/output downlink connectors (UTP/STP/optic);

- (at least for the FC+TU subdet[]s) one trigger (TPM) input/output uplink connector (UTP/STP/optic);
- Begin of Burst (BoB) input (and configurable burst length) to be able to produce BoB and End of Burst (EoB) triggers;
- 1 Gbit/s Ethernet port (RJ-45/optic);
- the FPGA, which:

† (re)distributes the direct signals — L0 clock (40 MHz), Reset, etc.;

† analyses the L1[n] trigger (TPM) inputs from FC+TU subdet[]s according to programmable (not wired) rules (trigger "ascending" path);

† produces the L1<t> / L2<t> trigger signals (of some type t) in form of the TPM (see below), delivers ones, and obtains the TPM responses (ackL1<t> / ackL2<t>) synchronously (trigger "descending" path);

† for each L1<t> / L2<t> trigger occurrence produces the trigger packet (of some type t, each packet is the Ethernet frame, which encapsulates the *ngdp* packet), supplies ones to the L1<t> / L2<t> fifo server, and obtains ACK / NAK responses in the same form (through the Ethernet port);

† implements the uniform SC slave (conversation with SC master through the Ethernet port);

- (at least for the FC+TU master) the hardware timeserver (GPS, etc.) with the ability to synchronize the L0 with the absolute time of the accelerator cycle begin and produce the absolute timestamp (struct timespec {sec; nsec}) for each trigger packet;
- the uniform power solution: single input (some of 12, 24, 48 V) and on–board converters;
- solution for the hardware reset (by power switching or FPGA reset).

The trigger *ngdp* packets produced by the FC+TU master should have both the reliable packet number and the absolute timestamp, as well as valid type, length, flags, CRC32 (i.e. the full packet header of 40 bytes), and possibly carry some information in the packet body. The trigger *ngdp* packets are used by the EvB[]s.

The **TPMs assortment** should be like the following:

- FC+TU subdet[n] ← Dig[n][]: preL1[n][] pre-triggers, ackL1 acknowledgments;
- FC+TU master ← FC+TU subdet[]: L1[] trigger, ackL1 acknowledgments;
- FC+TU master → FC+TU subdet[]: L1<t> / L2<t> trigger (of some type: BoB, EoB, some kinds of event), ?ackL1[] acknowledgments;
- FC+TU subdet[n] → Dig[n][]: L1<t> / L2<t> triggers (with types mapped according to FC+TU subdet[n] configuration: localized kinds of event, possibly BoB, EoB, some out of burst tests), ?ackpreL1[n][] pre-trigger acknowledgments.

The ACK / NAK scheme should be used for both the TPM and *ngdp* packet trigger exchange control. This means the future trigger production will be suspended until the proper acknowledgment obtaining. The TPM could be implemented as the fixed length transfer, the front of start bit is the time moment of the trigger (for the trigger input of the hptdc, etc.), the 32-bit value is the reliable trigger number (used by the FPGA), and some (?16) bits for trigger type (BoB, EoB, some kinds of event). (For the timestamps tag scheme of the (sub)events merging the two 32-bit values for sec and nsec should be transmitted by each TPM, too.) All cable lengths should be aligned for the simultaneous TPMs arrival at each FC+TU subdet[] and Dig[][]. Of course, the TPM assortment could be expanded over these mentioned above if needed. It is very likely that the switch to/from the test/imitation/calibration mode, the downscaling factor, BoR and End of Run (EoR), etc. should be distributed as TPMs.

The **FC+TU subdet[n]** (1 per n-th subdetector) should be similar or identical to the FC+TU master one. It:
• redistributes the L0, Reset, etc.;
• redistributes the trigger TPMs (and possibly produces local trigger TPMs: e.g., test/calibration between EoB and BoB) to the Dig[n][] cards (trigger "descending" path);
• analyses the preL1[n][] TPM inputs from Dig[n][] according to programmable (not wired) rules and produces the L1[n] trigger signal in the TPM form, delivers it, and obtains the TPM responses (?ackL1[]) synchronously (trigger "ascending" path);
• (in the reliable numbers tag scheme) for each L1<t> trigger TPM obtained produces the *ngdp* trigger packet with: the same reliable packet number as present in trigger TPM, the mapped type, valid length, flags, CRC32, the same timestamp as present in trigger TPM, or otherwise produced locally by the TPM arrival time;
• (in the timestamps tag scheme) for each L1<t> trigger TPM obtained / L1[n] trigger TPM issued produces the *ngdp* trigger packet with: the same timestamp as present in L1<t> TPM, or otherwise produced locally by the L1[n] TPM issuing time, the mapped type, valid locally produced number, length, flags, CRC32;
• supplies these packets to the L1[n] fifo server (or trigger input channel buffers of the SubEvB[n][]s) (through the Ethernet port).

The **Dig cards** are intended to the digital data preparation for the lower entities (FEM queues) of the DAQ subsystem. Such data (and possibly error) fragments are produced in form of the Ethernet frame, which encapsulates the *ngdp* packet with the full valid header of 40 bytes and the body, which contains the data fragment itself. Types for both data and error fragments (packets) are online configurable 16-bit constant integer offsets relative to the trigger TPM type values.

The Dig[][] hardware is responsible to produce the binary data in the reasonably compact format. Probably for the calorimeters (ZDC, DTE) we need the charge value only, so in the Q2T scheme the common FPGA of Dig cards should calculate the difference between trailing and leading signal edges, i.e. produce 32 bits per hit instead of 64 ones preserving the same precision 100 ps/bin.

The Dig cards location on setup is probably near the corresponding subdetector (1..3 m), while its shape should be suitable for both the Euromechanics crate (6U/9U) and standalone positioning.

The Dig cards use some ready digitization chips: hptdc [3] (TDC, QDC through Q2T conversion), n–XYTER or something other (coordinate), etc. under the common FPGA control, so it is possible to have a very few (2..3) slightly different Dig card types per BM@N setup. The common requirements for each of these types are to contain:
• connectors (with or without cables) or conductors (on the same printed circuit) to input data signals from the FEEs;
• the FC slave implementation in the common FPGA (see below for details);
• connector(s) (1 Gbit/s Ethernet) for the bidirectional conversations under the FPGA control:
† output — *ngdp*-packetized digital data fragments with the nanosecond precision timestamp (to FEM fifo), possibly SC responses (to SC master), etc.;
† input — backward ACK / NAK *ngdp* control packets to acknowledge the data fragments acception (from FEM fifo), possibly SC commands (from SC master), etc.;
• the uniform SC slave implemented by the common FPGA;
• the means to support the test/imitation/calibration mode (not yet understood in

details, however should not be forgotten);
• the uniform power input (some of 12, 24, 48 V) and on–board converters;
• solution for the hardware reset (by power switching or FPGA reset).

In respect of the FC+T subsystem the Dig cards should contain also:
• up to 8 signals sum inputs (UTP/STP/optic for digital) from the FEE[][]s;
• one trigger (TPM) input/output uplink connector (UTP/STP/optic);
• the common FPGA should in particular implement:
† the digital summator to produce the preL1[n][] TPMs (as part of the trigger production "ascending" path);
† the L1[] trigger TPM obtaining and the ackL1 acknowledge TPM sending (trigger "descending" path);
† the different behaviour at arrival the different trigger TPM types (BoB/EoB, some kinds of event or test / calibration);
† the data fragments production (see above), where the *ngdp* packet has: the body, which contains the data fragment itself, and the full packet header of 40 bytes, namely:
* (in the reliable numbers tag scheme) the reliable packet number (from the L1<t> trigger TPM), timestamp (from the trigger TPM, if present, or otherwise produced locally by TPM arrival time), type mapped according to Dig[n][] configuration, valid length, flags, CRC32;
* (in the timestamps tag scheme) the timestamp (from trigger L1<t> TPM, if present, or otherwise produced locally by theL1[n] TPM arrival time), type mapped according to Dig[n][] configuration, valid length, number (locally produced), flags, CRC32.

The Dig card with hptdc chips for the TDC and QDC is probably suitable for work with all but the STS subdetector and could be considered now in details.

The maximal capacity of the hptdc read–out token ring is 16 chips, so we can have no more than 512 channels per Dig card. However the reasonable size of the Dig card (6U) as well as the data flow issues (see below) restricts us by 8 hptdc chips (256/64 channels in 100/25 ps/bin modes). To reduce the data flow we should not produce the local (slave) header and trailer hptdc words. Some subdetector variants of the hptdc handling should be used:
• RPC mode: leading and trailing edges (100 or ?25 ps/bin), 64 bit/hit;
• T0 mode: leading and trailing edges (25 ps/bin), 64 bit/hit, without hptdc trigger matching ;
• Q2T mode: leading and trailing edges (100 ps/bin), 64 bit/hit, or with subtraction by the common FPGA, 32 bit/hit;
• DC/ST mode: leading edge (100 ps/bin) + width (more rough like 1.6 ns/bin), 32 bit/hit.

The common FPGA of the Dig card should for each obtained trigger TPM:
† read–out and buffer the data fragment according to the trigger TPM type for further Ethernet transfer,
† account the data fragment length;
† save the data fragment nanosecond timestamp (number of L0 clock pulses from start multiplied by 25), it is assumed the times for the same trigger in different hptdc are differ no more than by single L0 clock;
† fill in the 40 bytes of the *ngdp* packet header and store it in the 0th..9th words of the buffer;
† produce the summary error word using all encountered hptdc error words (if any, or fill by zeros otherwise) and store it in the 10th word of the buffer;

† produce the data fragment packet mentioned above;
† (as the online configurable option) produce separate packet of all hptdc error words, if their are encountered for current trigger.

We assume the data fragment should always be carried by single Ethernet frame to avoid the nontrivial reassembling algorithms on the obtaining side (FEM fifo). So, the data buffer size should be $\leq$ MTU = 1500 bytes, i.e. $\leq$ 187.5 bytes per event per chip. Note the maximal hits number per event could be enforced by the hptdc as power of 2, so we can guarantee that for each event the data fragment will fit into the single Ethernet frame. With up to $2^5(2^4$ for 64-bit hits) hits/event/chip the data of $(2^5 \times 8 + 2) \times 4 = 1032$ bytes ($2^4 \times 8 \times 8 + 2 \times 4 = 1032$ bytes) accompanied by the *ngdp* and Ethernet overheads of 40 and 38 bytes at the events rate of 10^5 Hz leads to the data flow of up to yet acceptable 888 Mbit/s.

The alternative for the hptdc usage is the TDC implementation on base of the dedicated FPGAs.

Probably the T0 Dig card should have the separate version of the FPGA firmware to produce also both hit counts per each trigger[2] and per each time slice (of the configurable duration, 5..10000 μs) for each T0 channel. This will require to perform the trigger matching (in the hptdc manner with configurable trigger latency and time window) by the Dig FPGA itself, because each produced hit (i.e. detected beam particle) should be evaluated for these counts calculation. At least the following trigger TPMs should be recognized: BoB, EoB, some kinds of event.

The **FEE cards** are intended to the signals (possibly always logical: LVDS / LVTTL) preparation for the Dig card. The FEE cards are located on the corresponding subdetector and possibly do not require the crate-suitable shape.

The FEE cards are specific for the serviced subdetector (here we can not essentially reduce entities to be designed), however in general each of them should contain and implement:
• the uniform SC slave (or be controlled through Dig);
• the means to support the test/imitation/calibration mode;
• the means to feed output signals to the Dig card(s);
• power supply: the uniform power input (some of 12, 24, 48 V) and on–board converters (or obtains the ready voltages from the Dig card);
• solution for the hardware reset (by power switching or FPGA reset).

In respect of the FC+T subsystem the FEE cards should provide:
• the branching of the analog signals from the detector to feed both the main (digitization) dataway and the trigger production "ascending" path;
• the analog (?or digital) summator, ADC, and single digital output as part of the trigger production path.

The FEE cards in the TDC chains probably will use the NINO chips [4] to produce the logical output signals for the hptdc [3].

The FEE cards in the QDC chains probably will use the Q2T converters to produce the logical output signals for the hptdc, too.

All the **SC master units** (typically 1 per n-th subdetector, see Fig. 3) should be similar or identical and contain at least:
• the standalone or Single Board computer (SBC, see also below): 1 Gbit/s Ethernet,

[2]Really between BoB and the first trigger, between two consequent ones, and between the last one and EoB.

network bootable (PXE), optional local flash memory as the permanent storage, the proper heat sink by the low–profile passive cooler;
• the means to converse (connector of Ethernet and/or some standard serial bus (RS-485, etc.)) with SC slave parts;
• the uniform power input (some of 12, 24, 48 V) and on–board converters to ATX voltages.

All the hardware **SC slaves** embedded into the FEE, Dig, HV/LV, FC+TU cards should be uniform, i.e. contain the same connector(s) and intellect (FPGA, etc.), which implements the same protocol. The only SC function should be hardware implemented for the SubEvB, EvB level computers is a remote reset solution (by power switching or motherboard reset).

Also we should choose a **hardware architecture of the standalone computers** to be used in all but FEM levels of the DAQ system and for the DAQ software design and implementation purposes. On the one hand, we have no special requirements to these computers hardware — other than performance and reliability. On the other hand, the big DAQ system can require from tens to some hundreds of units of such hardware with corresponding maintenance, etc. So, we should choose the most standard and generic hardware reasonably cheap due to a great volume of production. This architecture is called `AMD64/EM64T`, previously known also as `x86-64` and `IA-32e`, and should be used currently and in the near future. For the embedded SBCs involved in the BM@N DAQ they will be very desirable to have the same architecture (or the 32–bit `x86`). Note the SBC is preferable over the System–on–Chip (SoC) due to flexibility, replaceability, low integration efforts (printed circuit design is not required).

4. Software requirements

The operating system (OS) used on the online computers determines the DAQ system design and organization, consequently the inadequate OS selection is sure to strongly complicate implementation, maintenance, and using of the DAQ system. The OS itself should have adequate technical abilities for easy multiple installations, remote maintenance and backup, read–only boot filesystem and diskless network boot, boot without input and output devices, easy and centralized startup customization, etc.

UNIX–like OSs are optimal for the above requirements. UNIX is a multiprocess and multiuser OS with powerful mechanisms for interprocess and inter–computer communications, a very advanced virtual memory subsystem, support of sophisticated networking and graphics interfaces, extended tools for the software design. Costs for UNIX working itself are practically negligible on the modern hardware. Availability of the OS sources is a mandatory requirement for the present developments, while the freely distributable nature of some UNIX–like OSs is highly desirable. After all, high portability of UNIX programming and approximately unlimited quantity of the existing software are also very attractive.

To achieve the reasonable performance, we should choose C programming language (or C++ — only in such cases, where we can neither avoid an object–oriented design nor implement it using C) and ultimately avoid interpreted languages like Perl or CINT.

Because the FPGA firmware responsibilities are described above, here we formulate our requirements to the DAQ software modules only, which should be executed by CPUs of the standalone computers and/or SBCs.

In the proposed design we could use two **(sub)events merging schemes** based on the reliable trigger numbers or the timestamps as the (sub)event merging tags. The

former is suitable for the central L1<t> trigger only, while the latter — for any trigger architecture, however requires more expensive TPMs for timestamps propagation from the FC+TU master in the central L1<t> case. In case of the two-stage L2<t> trigger the timestamp is the only possible merging tag, however should be produced locally by each Dig card instead of propagated from the FC+TU master. Anyway, the synchronization issues should be addressed by the FC+T subsystem. Note the packet queues implementation seems to be simpler for the timestamping scheme (see chapter 5 for details). Anyway we require the proper merging tag to be present on the each produced data fragment / sub-event and trigger packet. For the timestamps tag this means each data fragment packet should be timestamped by the Dig with the enough (practically nanosecond) precision: really ones could be the 64–bit numbers of the L0 clock pulses[3] from some start time moment (e.g., BoB), for which the absolute time ats with the nanosecond precision is saved (in the struct timespec form) by the FC+TU master.

The fragment searching in the queues by the corresponding merging tag will be the only method to obtain all the fragments belonging to each the full event. For the reliable numbers the correspondence means their equality. For the timestamping it is the falling of the data fragment timestamp tsd into the time window, defined by the reference timestamp ts, parameter twin, and twin's usage policy. For example, for BOTH policy the ts−twin $\le$ tsd $\le$ ts+twin means tsd is corresponded to the reference ts. The time window possibly should be: twin $\le 1/(2F_{\rm L0})$ s, namely ≤ 12.5 ns for the clock L0 with $F_{\rm L0} = 40$ MHz frequency. After merging the sub-events are stamped by the reference timestamp ts of the L1[n] trigger packet, while the full events — by the absolute event time ets = ats+ts, where ts is the reference timestamp of the L1<t> / L2<t> trigger packet and the described above ats is carried in the L1<t> / L2<t> packet body. Anyway the strictly no more than one L1<t> / L2<t> trigger instance should be produced for each L0 clock pulse, because otherwise we will have uncertainty in the full event merging. The same sub–event could be collected into ≥ 1 full event(s), if this required by the L1<t> / L2<t> triggers of the different types t. Alternatively (not so easy to implement) the trigger of type t, which represents the more complicated conditions (i.e., more sophisticated and rare physical event), should have precedence over the simpler ones in the L1<t> / L2<t> production to resolve EvB[]s competition for the same sub–events. On the other hand, the most simple L1<t> / L2<t> trigger type could appear as often as the L0 clock. This means, in principle (too huge data flow should be processed!) we can exploit the whole setup in the almost free–streaming mode to accept any hits in the inner tracker.

The software on the L1[n] **fifo server** will behave as follows:

• some driver in the OS kernel:

† for each Ethernet frame arrival from the FC+TU subdet[n] card on the dedicated interface decodes this frame and obtains the trigger packet, which carries the merging tag (i.e. it is reliably numbered and/or timestamped with the nanosecond precision);

† puts this packet into the tail of L1[n] queue in the OS kernel;

• this queue:

† answers the SubEvB[n][] client(s) requests by sending the next packet of the corresponding type from the queue, or by sending the answer packet in the absence case;

† responds by ACK / NAK packets to the FC+TU subdet[n] card to propagate the "full" state on the lower DAQ levels and prevent the queue overfull;

[3]L0 pulse counts of 40 MHz possibly should be multiplied by 25 to express time in nanoseconds.

† supports the garbage collection to remove from the queue packets, which already obtained by all the connected clients;
† supports the queue cleaning — full or on the packet type basis.

The software of the **FEM fifo** will behave as follows:
• some driver in the OS kernel:
† for each Ethernet frame arrival from the Dig[][] card on the dedicated interface decodes this frame and obtains the data fragment packet, which is reliably numbered and/or timestamped with the nanosecond precision;
† puts this fragment into the tail of FEM queue in the OS kernel;
• this queue:
† answers the SubEvB[n][] client(s) requests by sending the data fragment(s) of the corresponding type and number (and/or timestamp) from the queue, or by sending the answer packet in the absence case;
† responds by ACK / NAK packets to the Dig[n][] card to propagate the "full" state on the lower DAQ levels and prevent the queue overfull;
† supports the queue cleaning — full or on the packet type basis.

The software on the L1<t> / L2<t> **fifo server** will behave as follows:
• some driver in the OS kernel:
† for each Ethernet frame arrival from the FC+TU master card on the dedicated interface decodes this frame and obtains the trigger packet, which is reliably numbered and/or timestamped with the nanosecond precision;
† puts this packet into the tail of L1<t> / L2<t> queue in the OS kernel;
• this queue:
† answers the EvB[] clients requests by sending the next packet of the corresponding type from the queue, or by sending the answer packet in the absence case;
† responds by ACK / NAK packets to the FC+TU master card to propagate the "full" state on the lower DAQ levels and prevent the queue overfull;
† supports the queue cleaning — full or on the packet type basis.

The software on the **SubEvB[n][] computers** will behave as follows:
• sub-event merger entity in the OS kernel:
† requests the FEM queues for data fragments and L1[n] queue for trigger packets according to own configuration, obtains their, matches merging tags, and merges packets to produce the sub-event packet;
† puts this packet into the tail of SubEvB[n][] queue in the OS kernel, possibly through some filter facility, which implements the L1.5[n] software trigger;
• this queue:
† answers the EvB[] clients requests by sending the sub–event of the corresponding type from the queue, or by sending the answer packet in the absence case;
† responds by ACK / NAK packets to the sub-event merger to propagate the "full" state on the lower DAQ levels and prevent the queue overfull;
† supports the queue cleaning — full or on the packet type basis.

The software on the **EvB[] computers** will behave as follows:
• full event merger entity in the OS kernel:
† requests the L1<t> / L2<t> queue for trigger packets and SubEvB[][] queues for sub-events according to own configuration, obtains their, matches merging tags, and merges packets to produce the full event packet;
† puts this packet into the tail of EvB[] queue in the OS kernel, possibly through some filter facility, which implements the L2.5 software trigger;

• this queue:
† answers the pool[] client(s) requests by sending each N–th full event of the corresponding type from the queue (this packet is kept in the queue for the storage client), or by sending the answer packet in the absence case;
† answers the storage[] client requests by sending the next full event of the corresponding type from the queue (this packet is removed from the queue), or by sending the answer packet in the absence case;
† responds by ACK / NAK packets to the event merger to propagate the "full" state on the lower DAQ levels and prevent the queue overfull;
† supports the queue cleaning — full or on the packet type basis.

The software on the **pool[] computers** will behave as follows:
• event pool entity in the OS kernel:
requests the EvB[] queues for the representative subset of full events according to own configuration, obtains these full events, and puts into the tail of pool[] queue in the OS kernel through the filter facility;
• this filter:
converts each of these full event packets into the representation by the corresponding ROOT class, which is serialized and packetized again;
• this queue:
† answers the visualization group client(s) requests by sending the next ROOT packet of the corresponding type from the queue (this packet is kept in the queue for other clients), or by sending the answer packet in the absence case;
† supports the garbage collection to remove from the queue packets, which are already obtained by all the connected clients;
† supports the queue cleaning — full or on the packet type basis;
† provides two policies of the full queue handling: the cleaning of the oldest possibly not yet read packet (default) and the newcomer packet dropping.

For the proposed histogramming server and client see chapter 5.

The **storage[] computers** possibly should be equipped by:
• the ***writer(1)*** utility [2];
• some network demultiplexer with the requests sending ability like the pool entity described above (possibly ported into the user context) to feed ***writer(1)***.

In Fig. 3 we can see main elements of the SC subsystem. The SC dispatcher and SC master[]s should contain CPUs and execute **SC software**. The SC dispatcher obtains operator commands from the operator GUI and forwards their to the specific subdetector's SC master[], which interacts with SC slaves to perform the corresponding actions. The correspondence between operator command and actions to be performed is established by the SC master[] local configuration. The remote network interactions between SC elements should be implemented by the standard client-server model (possibly using RPC) over TCP/IP or UDP/IP.

The SC slaves integrated into FEE, Dig, HV/LV, FC+TU cards are controlled by their firmware, while on SubEvB, EvB computers the SC slaves should be implemented by software (with exclusion of the remote reset solution, see chapter 3).

5. Software implementation

The key features of the proposed DAQ software, which are natural for the UNIX–like OS, are the following:
• splitting into the software modules interconnected by the experimental data streams;

• spreading of these modules over CPUs and networked computers easily;
• experimental data representation in the unified form of *ngdp* packets (see [2] and ***packet(3,5,9)***);
• experimental data transportation by the streams of these packets, which (without using the media slower than memory) can:
† be buffered, copied, filtered, demultiplexed in a different manner;
† to cross the context boundaries from the kernel space to the user one and vice versa;
† be transferred between software modules locally and/or remotely, etc.
• software modules implementation in form of the user context processes or, in the kernel context, — the so called loadable kernel modules (KLD);
• implementation of the packet streams between the processes:
† locally — by the unnamed pipes and
† remotely — by the TCP/IP socket pairs;
• using the kernel threads to implement the process-like activity in the kernel context allows us:
† to avoid the preemptive scheduling;
† to reduce the unnecessary data copying in the memory at the context boundary crossing;
• using the ***netgraph(4)*** package [5] to implement the packet streams between the ***netgraph(4)***-style kernel modules (nodes):
† locally — by the ***netgraph(4)*** data messages along the graph edges and
† remotely — by the TCP/IP ***ng_ksocket(4)*** pairs;
• high and easy scalability due to the object-oriented programming (OOP) style of the ***netgraph(4)*** package.

One of the freely distributable open source UNIX-like OSs with the ***netgraph(4)*** support, enough stable, reliable, modern, and dynamically developed simultaneously due to its very weighted up design policy, is FreeBSD.

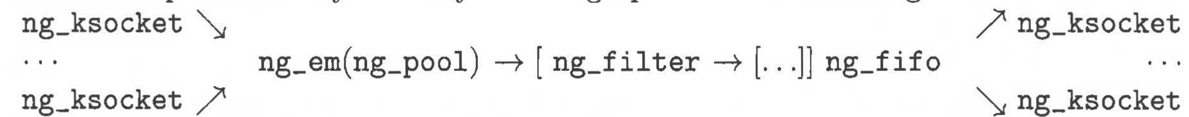
The *netgraph(4)* package [5] provides the following entities of our interest:
• ***ng_ksocket(4)*** socket for the remote data exchange by IP protocol (TCP, UDP, etc.);
• ***ng_ether(4)*** interface for the remote data exchange by Ethernet protocol;
• ***ng_socket(4)*** socket for data and control messages interchange between the kernel context graph and the user context process;
• means for building the graph itself: ***ngctl(8)***, ***nghook(8)*** utilities;
• service nodes for data flow managing: ***ng_tee(4)***, ***ng_one2many(4)***, ***ng_split(4)***;
• nodes for debugging: ***ng_source(4)***, ***ng_hole(4)***, ***ng_echo(4)***.

The ***netgraph(4)*** could be improved [6] in some aspects:
• the remote delivering of the ***netgraph(4)*** control messages is introduced;
• the function for the node insertion between the two already connected nodes is implemented (very useful for the software filters implementation);
• some additions for the ***ngctl(8)***'s scripting language are implemented;
• the maximum data message size could be tuned by changing some OS kernel variables (without the kernel recompile).

The *ngdp* framework [6] was implemented to be used for a big DAQ system software building and provides the following ***netgraph(4)***-style node types:
• ***ng_fifo(4)*** supports the packets buffer with some queuing disciplines;
• ***ng_em(4)*** implements the SubEvB and EvB algorithms of the event merging;
• ***ng_pool(4)*** implements the pool level functionality according to the above require-

ments (see chapter 4);
• ***ng_filter(4)*** supports (chain of) KLD plug–in(s) with the filter procedure(s) and/or the external filter implemented by the (pipe of) user context process(es) or the (chain of) netgraph node(s);
• ***ng_defrag(4)*** reassembles the big *ngdp* packets obtained from the ***ng_ksocket(4)*** after passing through network, which unavoidably fragments their to fit into the TCP/IP packets;
• ***ng_mm(4)*** (for Memory Mapping) provides the more efficient mechanism for the *ngdp* packets exchange between the kernel context graph and the user context process as the alternative to the standard ***ng_socket(4)***;
• ***ng_sv(4)*** (for SuperVisor) delivers the ***netgraph(4)*** control messages remotely, also could be used to automatize the remote ***netgraph(4)*** graph building;
• ***ng_mysource(4)*** and ***ng_kthsource(4)*** produce the packets stream for debugging.
The *ngdp* framework provides the following user context utilities:
• ***ngput(1)*** injects the ***pipe(2)***d packets into the graph;
• ***ngget(1)*** extracts the packets from the graph into the ***pipe(2)***;
• ***b2r(1)*** (for "binary–to–ROOT") converts the packetized binary data into the packetized ROOT class instances;
• ***r2h(1)*** (for "ROOT–to–histograms") is the runtime configurable histogramming server;
• ***histGUI(1)*** is the client for the ***r2h(1)***.

As we can see from the chapter 4 requirements, the SubEvB, EvB, and pool levels could be implemented by the very similar graphs like the following:



Of course, the queue and event merger nodes functionalities are slightly variated on the different levels. The FEM fifo as well as the `L1[n]` and `L1<t>` / `L2<t>` queues on the dedicated network nodes could be represented by the following scheme:

For SubEvB level there is also option to implement FEM and `L1[n]` queues as the input channels of the event merger and, subsequently, use more fast medium (memory instead of network) to request and obtain the data packets, which should provide some performance benefit. This will lead to the following scheme, which requires to implement some configurable multiplexor **ng_mux** and essentially revise the event merger to be **ng_eml**:

The further details about the *ngdp* provided entities can be found in [6, 7].

The **queue node type** (let it name as ***ng_fifo(4)***) according to the chapter 4 requirements could be implemented with two mutually exclusive assumptions about the (sub)event merging tag will be used, namely:
• the reliable packet numbers. This requires the following modes (i.e. buffer disciplines) of the packet extraction from the queue:
† "N": provides packet of the defined type `type`: `GETPACK(type)`. Needed by `L1<t>` /

L2<t>, EvB, and pool queues.
† "T": provides packet of the defined type type without removing it from the queue: COPYTRIG(type). Needed by L1[n] queue, if it is not embedded into SubEvB, otherwise replaced by "N".
† "G": provides packet of the defined type type by its number num: GETNTHPACK(num,type). Needed by FEM and SubEvB queues.
† "O": provides one of each Nth packets of the defined type type without removing it from the queue: COPY1OFN(N,type). Needed by EvB and pool queues.
• the timestamps. This requires the following buffer disciplines:
† "N". Needed by L1<t> / L2<t>, EvB, and pool queues.
† "T": provides packet of the defined type type without removing it from the queue: COPYTRIG(type). Needed by L1[n] queue if it is not embedded into SubEvB, otherwise replaced by "N".
† "S": provides packet(s) of the defined type type from the time window [ts−twin, ts+twin]: GETTSPACK(ts,twin,type). Needed by FEM and SubEvB queues.
† "O". Needed by EvB and pool queues.

Both "nNTGO" [6] and "NTSO" set of the buffer disciplines are implemented now by two separate versions of ***ng_fifo(4)*** node type. It seems the latter one is more simple.

The FEM level could be partially based on the CAMAC hardware. For the CAMAC hardware the number of DAQs are already built in this way using the *camac* package [8] and the ***ng_camacsrc(4)*** node [7] from the *ngdp* framework, e.g., SPILL DAQ, QUADRO DAQ [9], and IntTarg CDAQ [10], [11]. DAQ architectures like these are practically ready for the BM@N DAQ integration.

The **event merger** ***ng_em(4)*** for both the SubEvB and EvB levels should work very similar for the timestamps merging tag scheme. The "SubEvBT" mode should (with some simplifications) behave as follows. The ***ng_em(4)*** node makes one loop over the configured merging rules (and corresponding requests) array and launches the kernel thread (see ***kthread(9)***) for each configured index, so each thread serves only its "own" request. Each request has the so called trigger input channel and is handled in two phases:
• In the first (Trig) phase each thread emits the COPYTRIG(type) ("T") control packet to the L1[n] trigger fifo through the hook of the trigger input channel and waits for a positive or negative response up to obtaining one or corresponding (trigger) timeout expiration. If the answer packet is obtained, the thread analyses the error code and repeats the request after either the same or increased trigger timeout. If the trigger input channel does not respond at all before the trigger timeout expiration, the thread repeats the request and waits during the increased trigger timeout. The trigger timeout can be increased up to the limit only. If the data packet from the L1[n] trigger fifo is successfully obtained, the ***ng_em(4)*** node extracts the ts timestamp[4] and goes to the second phase, which for each request index is handled by the same thread as the first phase. Note the FEM and L1[n] queues implementation as the SubEvB input channels will simplify this behaviour and replace the "T" request by the "N" one, however leads to essentially revised node type ng_eml (not implemented yet) and very high requirements to the memory size available for SubEvB.
• In the second (afterTrig) phase the thread emits the GETTSPACK(ts, twin, type) ("S") control packets to the FEM queues through all the hooks of the involved input

[4] T type is also extracted and checked against the resulting type of the corresponding merging rule.

channels (other than trigger one) using the `ts` trigger timestamp obtained in first phase. After that each thread waits for positive and/or negative responses up to obtaining all the required packets or regular timeout expiration. If all the required data packets are obtained, the ***ng_em(4)*** node merges their (preserving the headers) into the sub-event packet of resulting type and with the number and timestamp of the trigger packet, and sends it to the `output` hook, if it exists, or drops packet otherwise.
After that the thread sets a regular timeout to the nominal value, sends the full request again, and so on.

The "EvBT" mode for the timestamps merging tag uses the `L1<t> / L2<t>` fifo server in the trigger input channel and requests it by the `GETPACK(type)` ("`N`") control packets, while the regular inputs — by the `GETTSPACK(ts,twin,type)` ("`S`") ones like the "SubEvBT" mode.

For the reliable numbers tag merging we have two-phase "EvBt" algorithm (like described in [6]) with "`N`" trigger and "`G`" regular requests. The SubEvB algorithm could be implemented as the single-phase "SubEvBt" one (see [6]), where `L1[n]` fifo has not dedicated meanings (it is simply one of the FEM queues, which requested by "`N`"), or like the two-phase "SubEvBT" algorithm described above, however uses the afterTrig phase "`G`" requests. The latter option seems preferable, because the two-phase algorithm should be more robust and less CPU-intensive. Anyway the existing ***ng_em(4)*** implementation [6] should be revised for BM@N purposes.

The *ngdp* framework implements the **pool level functionality** by the ***ng_pool(4)*** node separately from the very similar ***ng_em(4)***. The existing ***ng_pool(4)*** algorithm [6] seems suitable for us. The production of the events in a ROOT class representation from the full events in a native binary format will be done by the ***ng_filter(4)*** node type using the ***b2r(1)*** utility [7].

The ***r2h(1)*** **histogramming server** [7] could be executed to fill the ROOT histograms from the ROOT events. The ***r2h(1)*** could be fed by some network demultiplexer with the requests sending ability (e.g. the pool node ***ng_pool(4)***), as it proposed above for the ***writer(1)***.

The ***r2h(1)*** histograms assortment and parameters could be online configured using the ***histGUI(1)*** histogram visualization client [7] with the read–write access. This (single per server) client as well as many read–only ***histGUI(1)*** clients will be executed on the visualization group computers.

Note also some entities are inherited by the *ngdp* from the *qdpb* framework [2]:
• the application program interfaces (APIs):
† the ***packet(3,9)*** for the packet making, reading, writing, merging, etc.
† the ***pack_types(3)*** to deal with the packet types and corresponding attributes;
• the ***writer(1)*** utility to write the packet stream as the regular files on the HDD.
Some improvements for our needs are possible:
• the packet header could be improved by enlarging some existing fields and introducing the new ones (without keeping the backward compatibility);
• the packet body compression by the *zlib* or XZ Embedded package could be provided.

6. Conclusions

The basic principles and overall architecture of the DAQ system for the BM@N setup are described. The requirements to the DAQ, Trigger, and Slow Control hardware suitable for this architecture are issued. The *ngdp* framework [6, 7] seems to be suitable for needs of the BM@N DAQ due to the high scalability, easy remote distribution,

and the high performance of execution out of the scheduling scope. The *ngdp* nodes implementation should be adopted for the event merging scheme, which will be chosen. Possibly the present design can be used also by the upcoming NICA experiments, so the essential experience can be obtained during implementation and debugging of the proposed hardware and software.

Acknowledgments

The authors have a pleasure to thank T.A.Vasiliev for the very preliminary simulation results about the BM@N subdetectors occupancy and S.N.Basilev for the fruitful discussions about the hardware DAQ part.

References

[1] R. Brun and F. Rademakers. ROOT - An Object Oriented Data Analysis Framework. In *Proc. of the AIHENP'96 Workshop*, volume **A**(389) of Nucl.Instr.and Meth.in Phys.Res. (1997), pages 81–86, Lausanne, Switzerland, (1996). See also `http://root.cern.ch/`.

[2] K. I. Gritsaj and A. Yu. Isupov. A Trial of Distributed Portable Data Acquisition and Processing System Implementation: the *qdpb* – Data Processing with Branchpoints. JINR Communications, **E10–2001–116**, 1–19, (2001).

[3] J.Christiansen. *HPTDC. High Performance Time to Digital Converter. Ver.2.2 for HPTDC ver.1.3.* CERN/EP-MIC, March (2004).

[4] F.Anghinolfi et al. NINO, an ultra-fast, low-power, front-end amplifier discriminator for the Time-of-Flight detector in ALICE experiment. IEEE Trans. Nucl. Science, **51**(5), 1974–1978, (2004).

[5] `http://www.freebsd.org/cgi/man.cgi?query=netgraph&sektion=4` (2008).

[6] A. Yu. Isupov. The *ngdp* framework for data acquisition systems. JINR Communications, **E10–2010–34**, 1–20, (2010).

[7] A. Yu. Isupov. CAMAC subsystem and user context utilities in *ngdp* framework. JINR Communications, **E10–2010–35**, 1–20, (2010).

[8] K. I. Gritsaj and V. G. Olshevsky. Software package for work with CAMAC in Operating system FreeBSD (in Russian). JINR Communications, **P10–98–163**, 1–16, (1998).

[9] A. Yu. Isupov, V. E. Kovtun, and A. G. Foshchan. Implementation trial of the DAQ system for the compact physics setup on base of the *ngdp* framework (in Russian). In A. K. Vlasnikov, editor, *Fundamental problems and applications of nuclear physics: from space to nanotechnologies. Book of Abstracts*, 59 International Meeting on Nuclear Spectroscopy and Nuclear Structure (NUCLEUS-2009), page 346, Cheboksary, Russia, (2009). Saint-Petersburg, 2009.

[10] A. Yu. Isupov. New software of the control and data acquisition system for the Nuclotron internal target station. JINR Communications, **E10–2012–32**, 1–20, (2012).

[11] A. Yu. Isupov, V. A. Krasnov, V. P. Ladygin, S. M. Piyadin, and S. G. Reznikov. The Nuclotron Internal Target Control and Data Acquisition System. Nucl. Instr. and Meth. in Phys. Res., **A**(698), 127–134, (2013).