Process and Data Flow Control in KLOE

E. Pasqualucci[1]
*for the KLOE collaboration**

[1] INFN, sezione di Roma, P.le A Moro 2, I-00185 Roma (Italy)

Abstract

The core of the KLOE distributed event building system is a switched network. The online processes are distributed over a large set of processors in this network. All processes have to change coherently their state of activity as a consequence of local or remote commands. A fast and reliable message system based on the SNMP protocol has been developed. A command server has been implemented as a non privileged daemon able to respond to "set" and "get" queries on private SNMP variables. This process is able to convert remote set operations into local commands and to map authomatically an SNMP subtree on a user-defined set of process variables. Process activity can be continuosly monitored by remotely accessing their variables by means of the command server. Only the command server is involved in these operations, without disturbing the process flow. Subevents coming from subdetectors are sent to different nodes of a computing farm for the last stage of event building. Based on features of the SNMP protocol and of the KLOE message system, the Data Flow Control System (DFC) is able to rapidly redirect network traffic, keeping in accout the dynamics of the whole DAQ system in order to assure coherent subevent addressing in an asynchronous "push" architecture, without introducing dead time. The KLOE DFC is currently working in the KLOE DAQ system. Its main characteristics and performance are discussed.

Keywords: message system, SNMP, data flow control, process control

1 Introduction

The KLOE experiment[1] is taking data at the DAΦNE Φ-factory in Frascati. The KLOE DAQ system[2] has been designed to sustain a rate of 10^4 events/s and a data throughput of 50 MB/s. Its architecture is shown in fig. 1.

The front-end electronic crates have been distributed among 10 chains, each of them capable

---

* M. Adinolfi, A. Aloisio, F. Ambrosino, A. Andryakov, A. Antonelli, M. Antonelli, F. Anulli, C. Bacci, A. Bankamp, G. Barbiellini, G. Bencivenni, S. Bertolucci, C. Bini, C. Bloise, V. Bocci, F. Bossi, P. Branchini, S. A. Bulychjov, G. Cabibbo, A. Calcaterra, R. Caloi, P. Campana, G. Capon, G. Carboni, A. Cardini, M. Casarsa, G. Cataldi, F. Ceradini, F. Cervelli, F. Cevenini, G. Chiefari, P. Ciambrone, S. Conetti, S. Conticelli, E. De Lucia, G. De Robertis, R. De Sangro, P. De Simone, G. De Zorzi, S. Dell'Agnello, A. Denig, A. Di Domenico, S. Di Falco, A. Doria, E. Drago, V. Elia, O. Erriquez, A. Farilla, G. Felici, A. Ferrari, M. L. Ferrer, G. Finocchiaro, C. Forti, A. Franceschi, P. Franzini, M. L. Gao, C. Gatti, P. Gauzzi, S. Giovannella, V. Golovatyuk, E. Gorini, F. Grancagnolo, W. Grandegger, E. Graziani, P. Guarnaccia, U. V. Hagel, H. G. Han, S. W. Han, X. Huang, M. Incagli, L. Ingrosso, Y. Y. Jiang, W. Kim, W. Kluge, V. Kulikov, F. Lacava, G. Lanfranchi, J. Lee-Franzini, T. Lomtadze, C. Luisi, C. S. Mao, M. Martemianov, M. Matsyuk, W. Mei, L. Merola, R. Messi, S. Miscetti, A. Moalem, S. Moccia, M. Moulson, S. Mueller, F. Murtas, M. Napolitano, A. Nedosekin, M. Panareo, L. Pacciani, P. Pages, M. Palutan, L. Paoluzi, E. Pasqualucci, L. Passalacqua, M. Passaseo, A. Passeri, V. Patera, E. Petrolo, G. Petrucci, D. Picca, M. Piccolo, G. Pirozzi, C. Pistillo, M. Pollack, L. Pontecorvo, M. Primavera, F. Ruggieri, P. Santangelo, E. Santovetti, G. Saracino, R. D. Schamberger, C. Schwick, B. Sciascia, A. Sciubba, F. Scuri, I. Sfiligoi, J. Shan, T. Spadaro, S. Spagnolo, E. Spiriti, C. Stanescu, G. L. Tong, L. Tortora, E. Valente, P. Valente, B. Valeriani, G. Venanzoni, S. Veneziano, Y. Wu, Y. G. Xie, P. P. Zhao, Y. Zhou

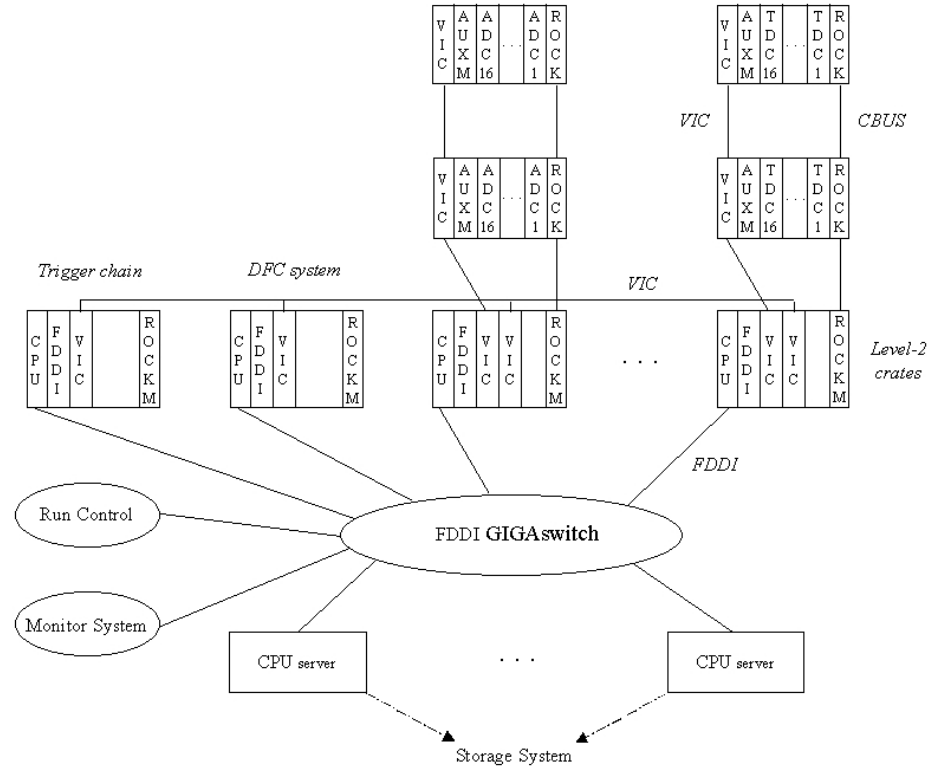

Figure 1: The KLOE DAQ system architecture

of more than 5 MB/s throughput. The first stages of event building are asynchronously performed in the chains:

- at the crate level, zero-suppressed data are collected through a KLOE designed bus and bufferized in a read-out controller (ROCK)[3];
- at the second level, the ROCKs' buffers are read through a KLOE designed fast crate interconnection and frames of subevents are stored in a ROCK manager (ROCKM) memory.

In each level-2 crate, a CPU on a VME board runs a process that asynchronously reads data from the ROCK manager and packs sub-events into packets, each of them containing a fixed number of sub-events[4]. Packets are built in a shared circular buffer. Another process gets packets from the circular buffer and sends them through a switched network to a farm of computers for final event building and storage[5]. The KLOE Data Flow Control (see section 4) optimizes the load of the farm processors, dinamically managing packet addressing.

Data quality and detector response have to be continuously monitored, as well as online event selection has to provide data for detector calibration. Online monitoring and event filtering in KLOE are described in [6].

2 The KLOE message system

The KLOE DAQ software architecture is shown in fig 2. The online processes are distributed over a large set of processors in the network. All of them have to change coherently their state of activity as a consequence of local or remote commands. A fast and reliable inter-process communication

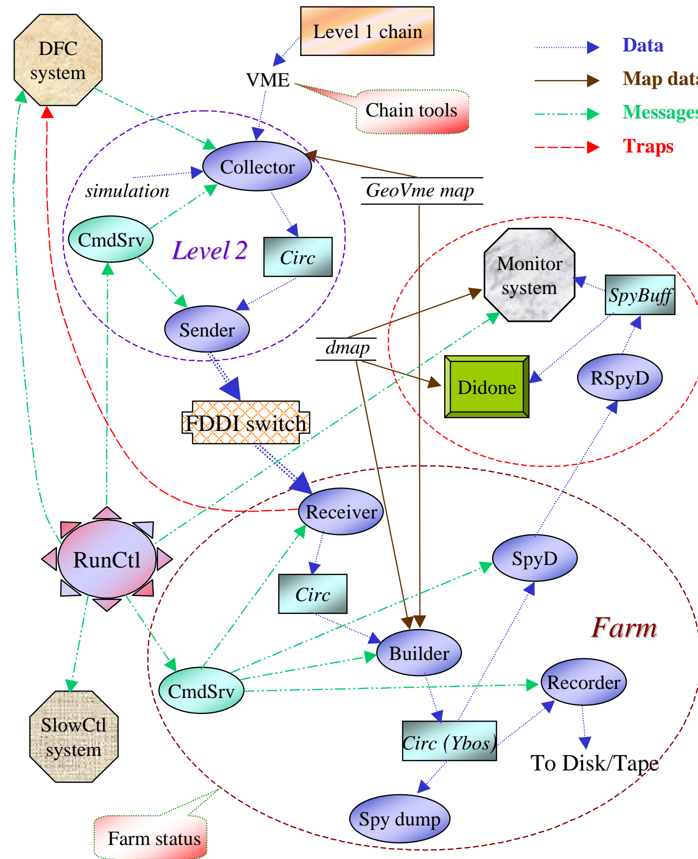

Figure 2: The KLOE DAQ software architecture

system is needed, as well as a method for real-time process monitoring.

2.1 Process structure and local communication

A DAQ node can be described in terms of a set of processes able to share their relevant parameters as well as to send and receive commands. In each DAQ node, the number and type of running processes may dynamically change, without affecting the management system.

The DAQ process structure has been inherited from the UNIDAQ model[7]. This model provides a common skeleton that allows any process to receive commands via standard UNIX mechanisms, to manage the run status and to declare itself to the DAQ system. When a process starts, it subscribes into a shared memory, adding a new item to a process table. A message queue is created to receive incoming messages. The reserved space contains both general information about the process itself (such as the process and message queue identifiers) and relevant local variables.

During the run, the process loop consists of three phases:

- it performs a slice of its task (for instance, it processes an event);
- it checks for command arrival and, on command, executes the required action;
- optionally, it stays idle for a while.

During each phase the process behaves according to its own run state. When a local command is

sent, the issuer locates the destination process in the process table, reads its identifiers, puts the command into the message queue and sends an interrupt to the destination process. The receiver's interrupt handler extracts the command from the message queue. At the end of the current task slice, it copies the command string to a variable in the process table containing the last received command, sets a command status variable to "executing" value and performs the required action. On command completion, the status variable is set to either "done" or "fault" value. The issuer process polls on this variable in order to get command acknowledgements.

2.2 Managing the DAQ network

The KLOE DAQ system looks like a complex network, including several computing nodes and network devices like switches and bridges. The standard protocol SNMP (Simple Network Management Protocol)[8], defined in 1987 by the Internet Engineering Task Force (IETF), was developed to provide a general purpose internetworking management protocol. It is commonly used to retrieve and set information about network configuration, traffic, faults, accounting and security. The SNMP is based on the User Datagram Protocol (UDP); reliability can be implemented by managing lost packets and retransmissions.

The information is made available as a tree of conceptual variables, defined in a Manager Information Base (MIB)[10] using elements of the ASN.1 notation. Managed objects are defined in standard MIBs by working groups within IETF, but both private extensions and proprietary MIBs can be developed as soon as new hardware or software products require them.

The SNMP protocol implements a client-server model: each device runs a daemon able to share information with remote managers, allowing them to perform operations on device variables. The daemon understands MIB requests and either obtains the required information or performs the required variable setting. The setting of a variable can be linked to an action to be performed by the daemon. A "trap" primitive is also defined for the daemon. It can be used in order to obtain manager's attention, usually signalling error conditions or status changes.

Well mantained, easy-to-use public domain software[8] is available. It is portable on many UNIX platforms and implements both dedicated daemons and utilities for remote access.

2.3 Command server and remote communication

In each DAQ node, a non privileged SNMP daemon runs, called a command server. A special private MIB tree has been defined that maps the structure of the process table. General and specific process variables have been implemented as one-indexed and two-indexed lists in this MIB subtree (see fig. 3).

In this way, the command server is able to map each variable in the local shared memory to the proper MIB one. Remote "get" or "set" queries on an SNMP variable allow remote clients to access information about processes running in a remote node, thus performing a sort of generalization of the shared memory. "Get-next" queries allow to scan the process table, identifying the location of each process and variable.

Getting remote variables allows us to monitor process activity by accessing their variables in the process table. Only the command server in involved in these operations, without disturbing the process flow. In general, SNMP daemons link "set" operations on MIB variables with variable-specific actions. As shown in subsection 2.1, a special variable in the process table is used to store the last command string received by each process. The command server converts remote "set" queries on this variable into a local command.

Our implementation of remote messaging is shown in fig. 4. When a client application (for instance the run control or a debugging utility) sends a command to a remote process:

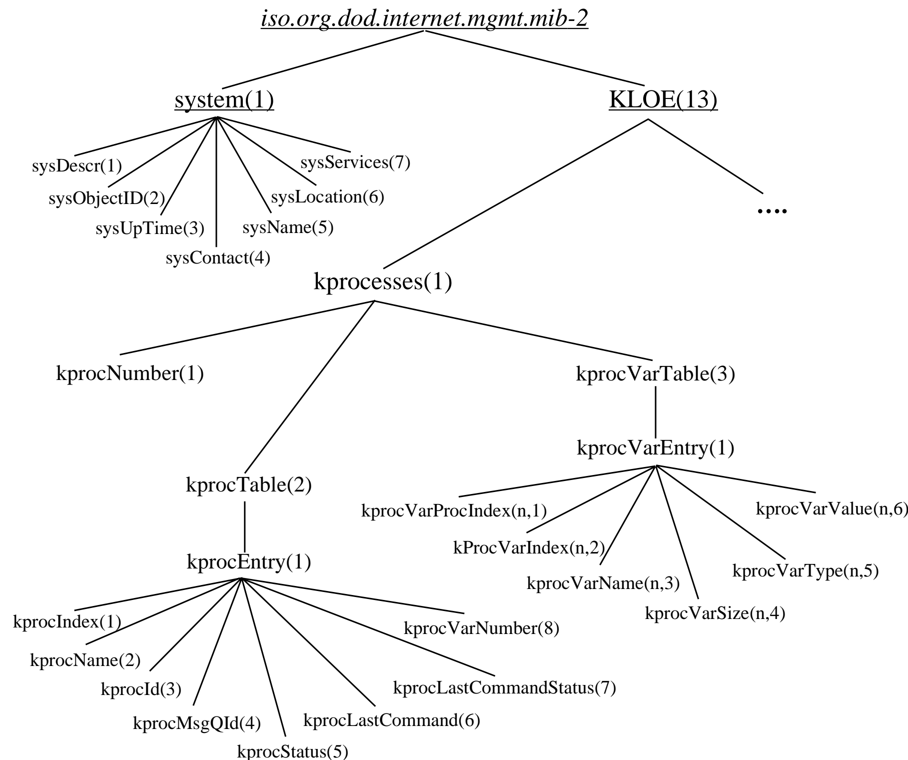

Figure 3: The KLOE MIB subtree

1. it locates the destination process (i.e., retrieves its index in the process list) scanning the MIB subtree;
2. it performs a "set" query on the last command variable. The acknowledgement of the "set" operation is provided by the SNMP protocol: the command server sets its internal variable to the required string and sends back a packet containing the new value. It does not modify the value in the process table. Then, it adds the command to the process' message queue and signals the process that a command has arrived;
3. the client polls for the last command variable being set in the process table ("first level acknowledgement"). When the destination process reads the command from the message queue, it writes the command string to the shared memory and sets the last command status to "executing". The acknowledgement is complete when the client gets a string equal to the issued command;
4. the client polls for the last command status being set to either "success" or "fault". The "second level acknowledgement" is completed when the destination process finishes to execute the command and sets the last command status.

The command server has been implemented as a DAQ process. It receives local and remote commands through the mechanism described above. In this way, global commands (for instance, run control commands like "start run" or "end run") can be defined for a node, delegating local command distribution to the command server.

The latest version of the message system library has been optimized by keeping track of

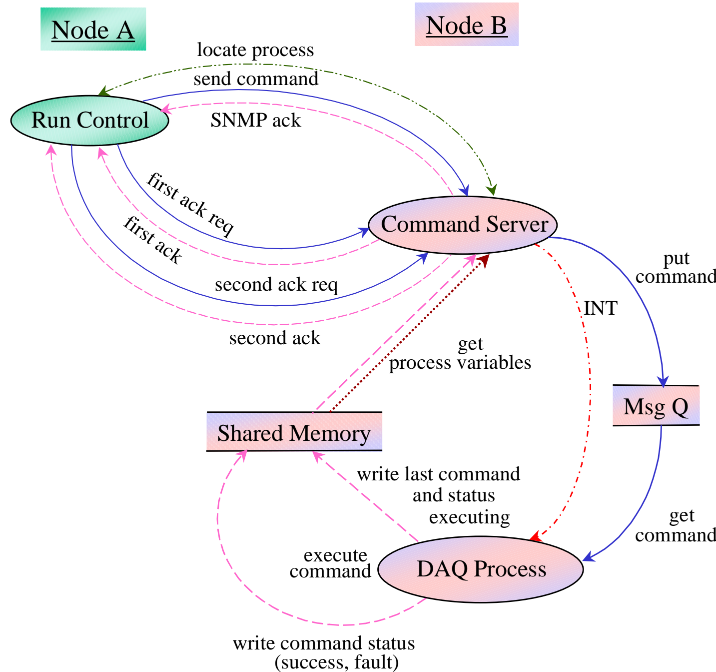

Figure 4: The KLOE message system implementation

process and variable locations. A client locates remote process and variable indexes only once, minimizing the number of queries for future requests. In this way, the average time required to get a remote variable in the KLOE DAQ network is ~ 1.2 ms and the completion of a remote command lasts ~ 4 ms, including the acknowledgements. Parallel command execution is supported. It has been implemented delaying acknowledgement requests after sending commands to a set of processes and has been very useful in implementing run control commands that take seconds to be executed by single nodes.

Graphic tools have been built using the tcl/tk package in order to help full process control. Only two new tcl commands have been implemented:

- a “get variable” command;
- a “send message” command.

Both the run control operator interface and some tools showing remote process variables have been developed using these tcl commands (see for instance fig. 5).

3 Production processes and remote control

The KLOE data analysis programs are developed in the Analysis Control (AC) environment[11] using the FORTRAN language. The message system described above is a very flexible and general tool. It has allowed us to implement full remote control of production processes. A new AC module and a C library have been developed in order to integrate Analysis Control programs in our message system.

The structure of our offline control system is described in fig. 6. Each production node runs a command server, a set of production processes and a local process checker. Each production process behaves like a DAQ process, sharing its relevant variables and accepting commands. The local process checker (locpc) continuously checks the correctness of process states. It signals

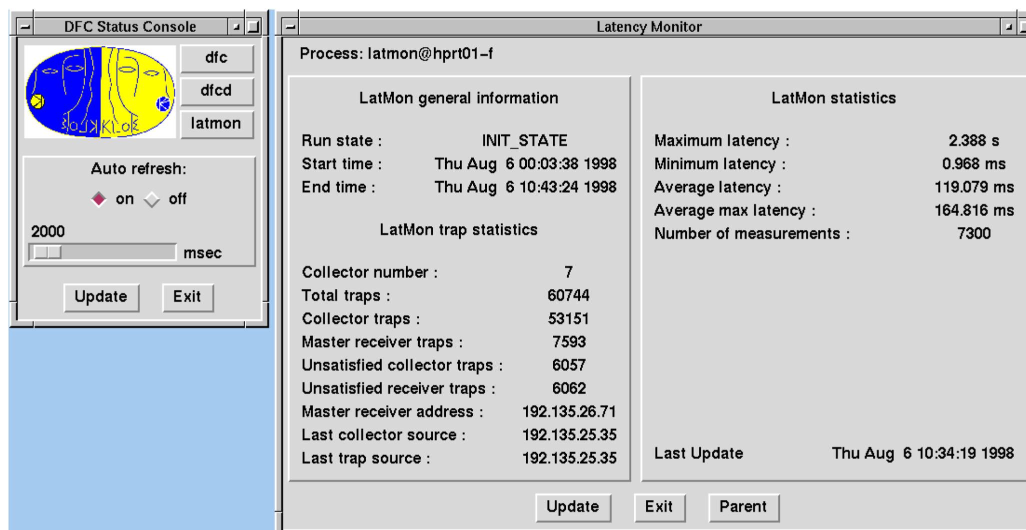

Figure 5: A tcl window showing the status of the latmon process (see subsection 4.2)

anomalous conditions to a centralized process checker daemon (pcd) using SNMP traps. A control system, made of a control process and an operator interface, is able to:

- distribute and start production processes according to processors' availability via remote command servers;
- monitor remote process status by reading their relevant variables (such as last event analized, current run number and so on);
- send Analysis Control commands to remote processes;
- send additional user-defined commands;
- receive error notifications from process checkers and recover error conditions due to process crashes.

This system gets the full control of the offline production.

4 The KLOE Data Flow Control system

The KLOE data transmission software is based on "push" architecture. When a subevent packet in a second level node is complete, it is made available in circular buffer for the sender process to be sent via a TCP/IP connection to a node in the event building farm (see fig. 2). The choice of each packet destination from a list of destination nodes is based on a round-robin algorithm. The receiver puts all the packets coming from different connections into a multiple circular buffer. When all the packets related to the same set of events are ready in the buffer, the event builder gets them, releasing the corresponding memory area. If a process in a farm node slows down for a while, the buffer fills up. If this condition holds till the buffer is full, soon the senders are unable to send any more data. The "buffer full" condition could propagate through the chains stopping the trigger and introducing additional dead time or even causing a blocking timeout.

The Data Flow Control system is able to change the packet distribution sequence, avoiding slow-down in data transmission and blocking timeouts. Its architecture is described in fig. 7. The

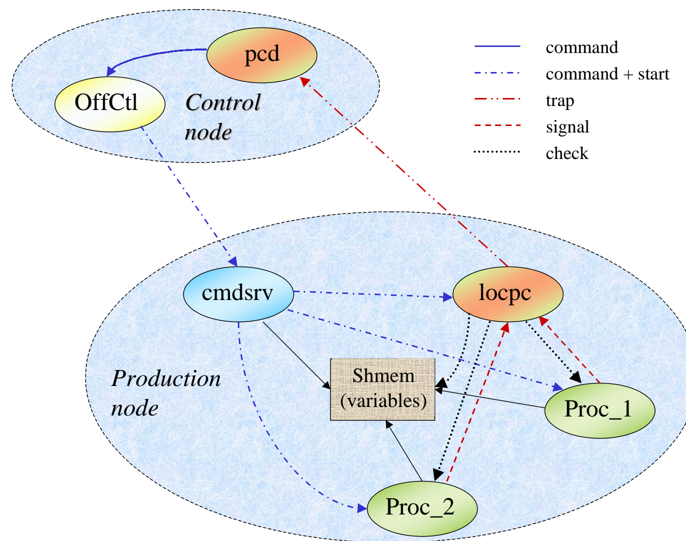

Figure 6: The KLOE offline control system

Data Flow Control machine runs a command server and three processes:

- the Data Flow Control daemon (DFCd);
- the Data Flow Control (DFC);
- the Latency Monitor (latmon).

All of them receive commands from the Run Control with the mechanism described in section 2 and have the same states of activity as all the other processes.

4.1 DFC and DFCd

At the beginning of run, the DFC builds a network map and a "DFC map", i.e. an ordered list of the IP addresses of the farm nodes. The DFC map is then written to a VME mirrored memory on a VIC module. This memory is shared, via a VIC bus, by all the L2 nodes (see fig. 1). A flow table is made of a set of flags indicating the status of activity of each node (active/inactive) and a last validity trigger. At the beginning of each run, the first flow table is generated with all the flags "active" and an infinite validity and is then written to the mirrored memory.

During the run, all the collectors in the L2 nodes use the DFC map to assign destinations to sub-event packets, according to activity flags and validity triggers. When a data receiver in the event building farm receives a packet, it puts data into a multiple circular buffer. Each section of this buffer corresponds to a connection. The data receiver continuously checks the status of its buffer. When the builder does not empty the buffer and its occupancy overcomes a threshold, it is declared "almost full" and an "almost full" SNMP trap is sent to the DFCd.

The DFCd is an SNMP trap daemon. It receives the receiver traps and translates them into local commands to the DFC. When an "almost full" trap is received, a "destination unavailable" command is sent to the DFC. The DFC creates a new flow table and writes it to the mirrored memory with infinite validity. Then, it asks the trigger supervisor (TS) to update its variables and reads the current trigger number and rate. A new validity trigger is calculated for the old flow table, and is written to the mirrored memory. When a collector processes a packet with a first event which has a trigger number greater than the validity trigger, it switches to the new table and assigns only enabled nodes as destinations.

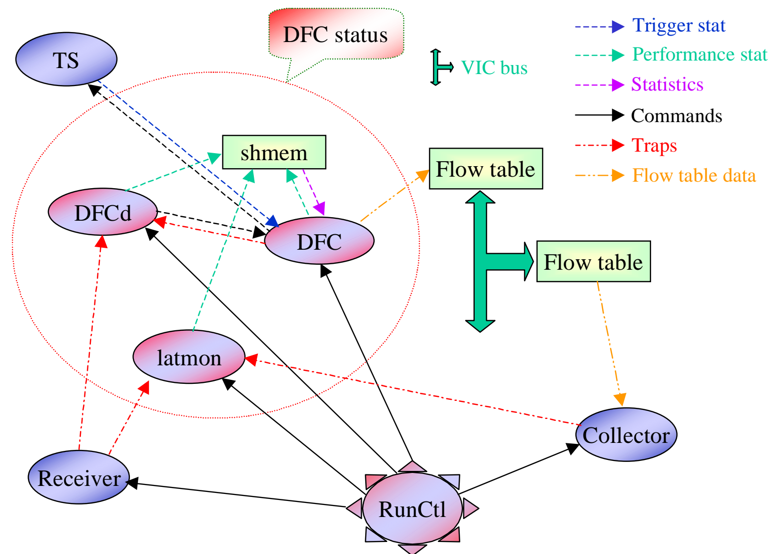

Figure 7: The KLOE DFC system architecture

After sending an "almost full" trap, the receiver continues to monitor the buffer occupancy. When the buffer occupancy becomes less than a low threshold, it sends an "almost empty" trap to the DFCd that becomes a "destination available" command to DFC. This mechanism allows us to avoid traffic congestion even if some operator intervention is required (for instance, if some cleaning or data storing procedure failed and an output disk is full). In this case, removing the error condition allows the node to build and write events again, emptying the buffer and re-enabling data transmission through its connections.

Due to the asynchronous architecture of the system, calculating the validity trigger is not straightforward. In general, corresponding packets are processed at different times and what we define as the "actual" trigger number and rate refer to the trigger supervisor at the moment of the execution of the variables update. The Data Flow Control must implement a smart algorithm in order to "guess" a reasonable validity trigger, and in this way redirects the traffic as quickly as possible. On the other hand, a wrong evaluation of the validity trigger would generate an event number that has already been processed by one or more collectors, causing a disalignement in packet addressing and generating a fatal error.

The Data Flow Control has not only to know the "actual" trigger number and rate, but also to maintain the statistics of these numbers in order to take into account trigger rate variations. Moreover, its own "reaction time" has to be considered, from the issue of the update command to the TS to the completion of the validity trigger writing operation on the VME mirrored memory. This time can be divided into a trigger interaction time (completion of the update command plus remote variable reading) and a DFC time (validity trigger evaluation and writing). The former can be measured before writing, the latter has to be evaluated statistically.

In order to take into account the whole dynamics of the system, the DFC performs continuously an autocheck procedure: a "test" trap is sent to the DFCd, that translates it into a "test" command to the DFC. All the operations described above are performed and a new flow table and validity trigger are written to a test area in the same mirrored memory. In this way, the DFC can measure both the total reaction time of the Data Flow Control system (from the trap issuing to the validity writing) and its own "reaction time" (defined as above), keeping both short term and long

term statistics. The validity trigger is computed as:

$$v = t_0 + (t_{tr} + (\overline{t_{dfc}} + k\sigma_{dfc})) \times (\overline{\nu} + k\sigma_\nu) + \tau,$$

where t_0 is the trigger number as read from the trigger supervisor, $\overline{\nu}$ is the trigger rate averaged over a short time, $\overline{t_{dfc}}$ is the average DFC time and:

$$t_{tr} \equiv max(t_{tr}^{measured}, \overline{t_{tr}} + k\sigma_{tr})$$

is the trigger interaction time. Actually, $k = 5$ and τ in an offset that makes v the last event number in the packet.

The overall DFC system reaction time is given by the sum of the following contributions:

- the DFCd reaction to a trap ($\overline{t_{dfcd}} \sim 1.2$ ms);
- the DFC reaction to a local command ($\overline{t_{local}} \sim 1.2$ ms);
- the trigger interaction time ($\overline{t_{tr}} \sim 6 - 7$ ms), including:
  - the update command;
  - the remote variables readout;
- the DFC time ($\overline{t_{dfc}} \sim O(10^{-2})$ ms).

The average total reaction time is $\sim$ 10 ms, negligibile with respect to the average packet latency (see subsection 4.2).

4.2 The Latency Monitor

The DFC system has to check that both the receiver's output buffer size and the trap upper threshold are adequate to provide, in an "almost full" condition, enough room for packets released by the collectors before traffic redirection. It has to monitor packet latency, defined as the time elapsed between the availability of a packet of subevents for the sender and the availability of all the packets related to the same set of events for the event builder.

Latency measurement is performed by latmon using traps. When a packet with an identifier which is a multiple of a given number is released by each collector, a "collector trap" is sent to the latency monitor. All the packets with the same identifier are received by the same receiver. As soon as the last one is made available for final event building, another trap ("receiver trap") is sent to the latency monitor. Latmon calculates minimum, maximum and average latency for each packet and their averages. The fastness of the trap mechanism (the average time needed to receive and react to a trap is ~1.2 ms) allows us to have very good time resolution with respect to the average packet latency (of the order of some hundreds of ms).

If the values set by default at the beginning of a run are not adequate, new values can be suggested for future runs. While the buffer size is a static property and can be changed only at the end of current run, thresholds can be changed dynamically by simply sending a "set" command to the receivers.

5 Conclusions

The KLOE message system, based on standard UNIX mechanisms and the SNMP protocol, has been successfully used within the KLOE data acquisition system. Its latest version has been optimized, providing a very fast and reliable way of sending both local and remote commands to DAQ processes and getting information about them without disturbing the process flow. Using the SNMP protocol has allowed us to have a unique set of facilities in order to monitor the behaviour of network devices and implement full process control.

The KLOE Data Flow Control system, based on features of the SNMP protocol and the KLOE message system, has been proved to be a very efficient and reliable tool to balance network traffic and dinamically recover local farm failures during the first period of data taking.

References

1 The KLOE Collaboration, "The KLOE Detector, Technical Proposal", LNF-93/002, 1993.

2 The KLOE Collaboration, "The KLOE Data Acquisition System, Addendum to the Technical Proposal", LNF-95/014, 1995.

3 A. Aloisio et al., "Level-1 DAQ system for the KLOE experiment", CHEP'95, Rio de Janeiro, 1995.

4 P. Branchini et al., "Front-end data acquisition for the KLOE experiment", CHEP'98, Chicago, 1998.

5 E. Pasqualucci et al., "The online farm of the KLOE experiment: event buiding and monitoring", CHEP'98, Chicago, 1998.

6 A. Doria et al., "Online monitoring system at KLOE", CHEP 2000, Padova, 2000.

7 M. Nomachi et al., "UNIDAQ - UNIX based data acquisition system", CHEP'94, s. Francisco, 1994.

8 AA. VV., "The SimpleWeb - The source for SNMP and Internet management info", `http://www.simpleweb.org`.

9 AA. VV., "The Simple Times, the Bi-montly Newsletter of SNMP Technology, Comments, and Events", `http://www.simple-times.org`.

10 D. Perkins, E. McGinness, "Understanding SNMP MIBs", Prentice Hall, 1996.

11 M. Shapiro et al., "A beginner's guide to analysis_control and build_job", CDF note 384, 1995.