Calibration of the ATLAS Data Collection component models.

Authors: Piotr Golonka, Krzysztof Korcyl
Keywords: DataCollection, modeling, calibration

Abstract:

The parameterized models of the Data Collection components are being constructed. The models aim to reproduce functionality and performance of these software packages. The models needs to be calibrated with data collected during measurements focused on performance of the components. The calibration and high-level approach of the models will be verified in models of larger size setups. Ultimately, the models will be used in the simulation of the full size ATLAS HLT system to predict latency, throughput and identify places with excessive queues build-up.
In this note we present briefly ideas behind the Event-Driven modeling. We followed the DC organization and built different models for the DC Message-Passing subsystem and for the DC applications. For each of the DC components we present high-level description in terms of received and produced messages. The description is followed by a list of times that need to be measured to calibrate the models.

NoteNumber: 57
Version: 0.9
Date: January-2003
Reference: http://cern.ch/Piotr.Golonka/modeling/at2sim

Document History:

1.Document Title: Calibration of the ATLAS Data Collection component models.			
2.Document reference number:			Atlas DC Note 57
3.Issue	***4.Revision***	***5.Date***	***6.Reason for change***
0	01	Oct 17	Initial version by Piotr Golonka
0	02	Oct 20	Kris: Introduction,DFM and SFI
0	03	Oct 22	Piotr and Kris: First draft for comments
0	04	Oct 23	P&K: Abstract, MsgPas, GenericApp, HwRobEmu updated
0	05	Nov 8	Corrections by Kris: SFI,DFM: PULL and PUSH
0	06	Nov 11	new SV and L2PU parameters
0	07	Nov 27	Message Passing chapters
0	08	Dec 6	Message Passing: removed send, changed L2PU and SV
0	09	Dec 11	Message Passing: removed send, changed ROSe, pROS, DFM, SFI

Contents

1 Introduction.

1.1 Discrete Event Simulation.

We plan to model the DC software functionality and reproduce its performance using the Discrete Event Simulation (DES) approach. The DES is very suitable for high level modeling of communication networks, queuing systems and computer architectures.

The DES is used to simulate components which normally operate at a higher level of abstraction than components simulated by continuous simulators. Within the context of discrete-event simulation, an event is defined as an incident which causes the system to change its state in some way. For example, a new event is created whenever a simulation component generates output. A succession of these events provide an effective dynamic model of the system being simulated. What separates discrete-event simulation from continuous simulation is the fact that the events in a discrete-event simulator can occur only during a distinct unit of time during the simulation – events are not permitted to occur in between time units. Discrete event simulation is generally more popular than continuous simulation because it is usually faster while also providing a reasonably accurate approximation of a system's behavior.

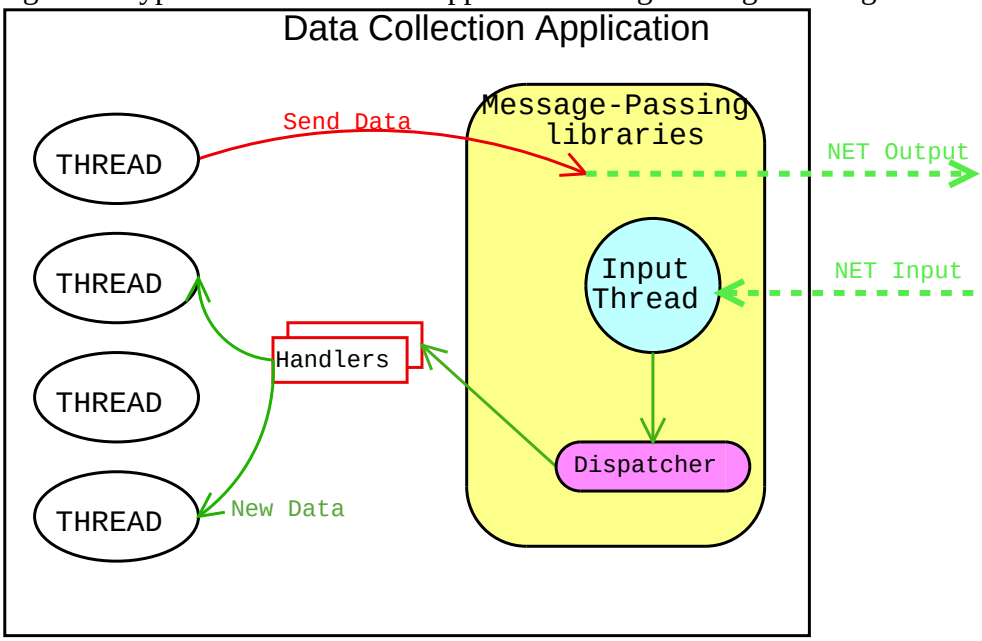
The principle restriction placed on DES is that an event cannot affect the outcome of a prior event, that is, logical time cannot run backward.

As a tool we use Ptolemy [1] which provides environment to create models in object oriented techniques and simulates interactions between them in the DES fashion. In our effort to model the ATLAS HLT system we plan to answer very basic questions related to operation of the complete system: what latency we may expect, what throughput we may reach, how big queues and where we may expect. To answer these questions we plan to use an high-level approach to model functionality of the HLT components: software or hardware emulators, PC-based nodes running the DC software and the interconnecting Ethernet network (mainly switches).

Therefore we plan to treat the DC software nodes as encapsulated objects being driven by incoming messages and producing messages directed to other components. The object representing model of the DC node changes state when a new message arrives. Ideally, in the high-level approach, it would be sufficient to assume that the next change of the status of that object will be caused by generation of message in response to the received one. Thus only one time would be necessary to represent such behavior: time between reception of a message and the moment when a response is created. The time needs not to be a constant - it may be a function of parameters defining the state of the object. However, it would be desirable that the value of the time can be calculated at the moment the incoming message arrives.

In the rest of this document we will describe the very basic times necessary to produce the high-level models of the DC software nodes. In most of the cases these times can be measured on nodes, running the DC software and using the time-stamp library [5]. In some cases the maximal rate measurements may also be useful. Measuring these time is called calibration. Using calibrated models, the models using calibration data, we will try to reproduce behavior in terms of latency, throughput and packet loss of various sizes of the testbed systems with the DC nodes. Should these results differ significantly from the testbed measurements, another iteration with more in depth look into the structure of the DC software will be necessary.

Figure 1: Typical Data Collection application using Message Passing libraries

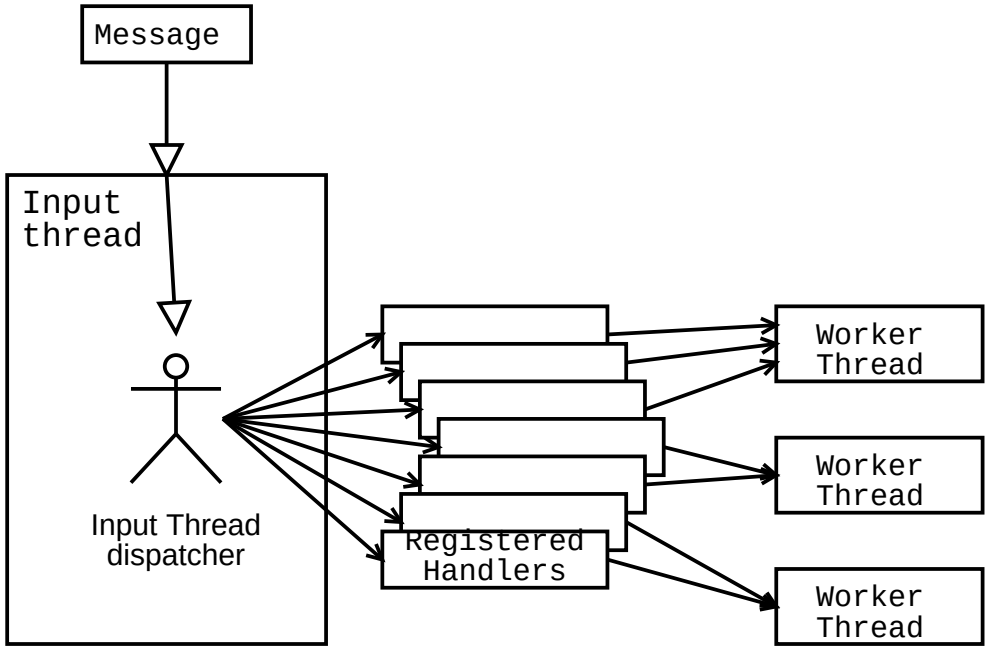


1.2 Data Collection framework and applications.

The Data Collection applications are built on top of a custom, OS-independent framework, which provides an access to the operating system services (i.e. network communication) through an abstract, high level, OO interface. Particularly, the network communication between the DC nodes is realized by the Message Passing subsystem (see fig. 1). This component is used in all Data Collection applications.

The receiving process in the Data Collection software is much more complicated than the send process: the ideas of the input thread, message dispatcher and message handlers are used (fig.2). Upon the arrival of a packet from the network to the DC node, it is received at application level by the *Input Thread*, which is a part of the Message Passing system. The input thread decodes the message contained in the receiver packet and passes the message to the *Dispatcher* object. The dispatcher object is responsible for handling the message properly. The dispatcher have a dynamic list of *Handlers*: routines/objects that are registered by the tasks interested in certain types of messages. A task registers its own handler in the dispatcher in order to get any information from the network. Together with the routine, the type of information in which the task is interested is also passed at the stage of handler registration. Upon arrival of the new message, the dispatcher searches the list of registered handlers and passes the message to the one, which have its selecting criteria matching the message type. The handler ultimately processes the message: extracts the data and perform steering actions in the application.

Figure 2: Receive process in the DC Message Passing.



1.3 The generic model of application

All Data Collection applications are built upon the Data Collection framework: they share the same code for all communication services. We want to take advantage of that fact in the model: the behaviour and parameterization of the operating system and Message Passing subsystem will be modelled in the same way for all components.

We propose to create the common modeling framework, which could provide “generic services”: the model of the multi-tasking operating system with parallel execution of application threads based on CPU resource sharing and the interrupt-driven protocol stack.

Fot the at2sim model we have prepared the modeling framework that provides the set of base classes for a model of the generic, multi-threaded DC application executed on a computer with multi-tasking operating system and network connectivity. The model of application should be contain in a set of classes which inherit from the base classes we provide. The whole functionality of the modeled application should be contained in the model of its execution threads: each application should contain single or multiple execution threads, which are provided with the data incomming from the network, can request the CPU time for processing and send the data to the network. Because this functionality is contained in the base classes, one could parameterize the message-passing for all modelled Data Collection nodes and concentrate on models of distinct applications.

2 Parameterization of the Message Passing system

2.1 The generic model of DC Message Passing

The generic model of the Message Passing system used by the models of application provides the parameterization of the receive process in the Data Collection node. We propose a parameterization which models the behaviour of the multi-tasking operating system with interrupt-driven network communication. We have observed [4] that send routines are much simpler than receiving, that is why we are not going to model them at the level of generic Message-Passing model but rather leave the modeling of CPU usage and latency of send routines to the models of individual applications. The other reason for this is simplifying the calibration: the preparation of message, sending and cleanup may be parameterized and modelled together.

In the discussed generic model we want to reproduce the latency and CPU resources consumption due to the network communication. There are various ingredients for the latency and CPU use: interrupts from the network card, protocol specific processing, overheads of the message-passing processing. Moreover, the interrupts overhead may be dependent on interrupt mitigation techniques [4]. In our model we assume that there is no additional latency caused by the hardware (NIC). We are also *not* going to model the details of network communication protocols in the current version of the model.

We therefore propose to establish the set of times as parameters and measure their values under various conditions. That would mean that the times would not be the simple constant values, but should also specify the dependencies on other parameters, i.e. communication protocols, message size, CPU and bus speed.

2.2 Parameters

The values for the following parameters need to be specified:

recv_int_time: CPU time needed to serve interrupt for incoming data: models operating system's overhead; it is modeled as high-priority task, i.e. it preempts (delays) other CPU requests

recv_int_coalescence: interrupt coalescence timeout (in microseconds) for receiving a data: the interrupt is not raised immediately upon arrival of a message - it is delayed by the amount of time specified by this parameter, so more than one incomming message may be signaled to the OS using single interrupt

recv_protocol_time: CPU time to serve protocol-stack - related workload, i.e. network routines executed outside the interrupt handler; also executed at high priority; it should be accounted together with recv_int_time, however with interrupt coalescence behaviour properly modeled

recv_app_time: CPU time needed to serve "application-level" incoming data - the overhead of DC Input thread, buffer management, etc.

recv_delay: latency for receiving a packet (hardware-related) : as above - NIC latency for incoming packet.

2.3 Measurements

The values of parameters related the operating system may be taken directly from the "comms tests" results [4]. The other, specific to the overhead of Data Collection need to be measured using the setups and testing programs similar to the ones used in [4]: these programs need to use DC Message Passing libraries.

The DataCollection-specific measurements will be performed on setups similar to the ones of "comms" tests: request-response and streaming tests should stress-test the message-passing subsystem and provide values for maximum rates. Instrumentation of the Message Passing code will be one of possible ways to measure the value *of recv_app_time* . The other way (which would allow us to cross-check the parameter values) leads via comparison of results of message-passing stress-tests with the results of "comms" tests.

2.4 The messages.

The messages in the model are based on the "Message Format" document specification [6] . The messages names and type identifiers are taken directly from the document. However the further details may differ significantly: i.e. we do not model the whole information stored in the message, we'd rather concentrate on providing the same applicability and try to optimize the messages. Particularly, the addresses, port numbers, byte ordering, and generic header are not followed at all.

All messages exchanged by the models of DC applications inherit from the base class *Amessage*, which contain Ptolemy-specific infrastructure and some generic information: source and destination addresses, time stamps, event identifier, message length and network tags (VLANs). In order to take advantage of the message-passing model, the messages should be sent using the API specified in the *Task* class (**??**).

3 Supervisor model

The L2 Supervisor node is responsible for assigning the LVL1 results to L2PU nodes for further processing, then collecting the LVL2 results and sending them to the DFM. In the real system, the LVL1 Result will be obtained using hardware RoI builder cards. The average rate of LVL1 results will be 70-100 kHz.

The outline and messages used by the Supervisor are presented in fig. 3.

The Supervisor performs the following steps during its running state:

- fetches a single LVL1 result from the hardware RoI Builder cards and puts it on the queue
- tries to assign the LVL1 results from the queue to L2PU nodes using load balancing algorithms

- accepts and classifies received LVL2 results
- dispatches the LVL2 results to the DFM

Above activities are executed in a single execution thread, in a loop.

We propose the following sets of parameters to model the timing of the Supervisor (message-passing - related time excluded).

event_delay_time: this is the value of eventDelay parameter of the Supervisor; specifies the minimum delay (in microseconds) between two consecutive reads of LVL1 result from the RoIBuilder may occur.

get_LVL1_result_time: this is the time used to get the LVL1 result from RoIB cards; there exist implementations of various LVL1 result sources, therefore one needs to specify the value of this parameter for all possible implementations: i.e.: L1InternalSource, L1PreloadedSource, L1SLinkSource, L1TTCSource. The time needed for putting the result to the internal queue is also accounted here

choose_L2PU_time: this is the time spent in the load balancer routines: various implementations (e.g: LoadBalanceLeastQueued) needs to be parameterized

send_L2PU_request_time: this is the time needed to prepare and send the request to L2PU: the LVL1Result message; the time needed to get the message from the queue should be accounted;

process_LVL2_result_time: the time needed to process and classify the LVL2 result received in the L2PU_LVL2Decision message; the time spent for load-balancing services (i.e. in the L2Process::getResult() method) should be accounted for;

record_LVL2_result_time: time needed to add the LVL2 result to DFMReporter's list of LVL2Decision objects

send_DFM_message_time: time needed by DFMReporter to prepare LVL2DecisionGroupMsg message out of LVL2 results accumulated in LVL2Decision list, send it and clean up the data structures needed

check_timeout_time: the time needed by timeout-checking routines

The parameters are also presented in Figure 4.

4 L2PU model

The L2 Processing Unit verifies and refines the results of L1 trigger. It executes multi-step, algorithms to verify that the analyzed event passess one of criteria sets in the trigger menus. The data is queried from the Regions Of Interest (RoIs): the geometrical regions of the detector

Figure 3: Supervisor-related messages

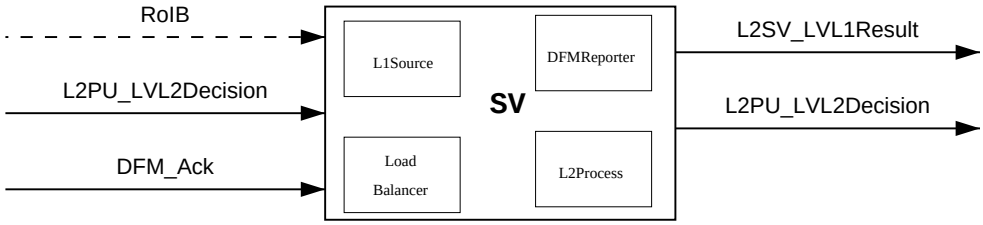
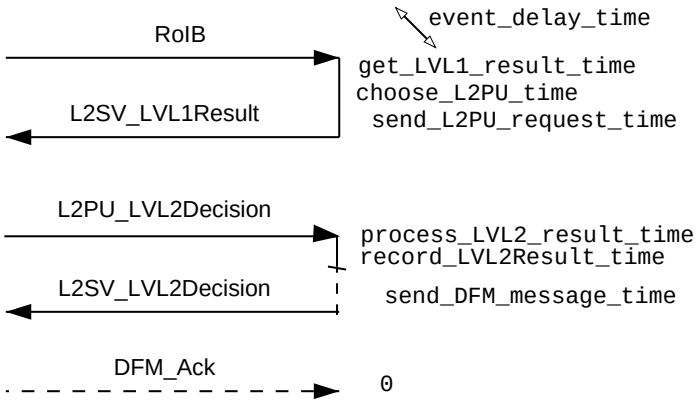


Figure 4: Supervisor parameters



indicated by the LVL1 trigger. Each fragment of detector data needs to be requested explicitly by means of a unicast message, then incomming frgaments need to be assembled to RoI data. The algorithms, and corresponding data requests, are executed sequentially, as indicated by the trigger menus ("sequential processing").

The outline and messages used by the L2PU are shown in Fig. 5.

In the running state, the L2PU performs the following activities:

- on reception of the processing request from the Supervisor (L2SV_LVL1Result message), it dedicates one of its worker tasks to process the event or puts the request on the internal queue if there is no idle thread available
- the worker tasks perform "sequential processing" of the assigned event according to trigger menus
- the worker task determines the addressess of RoBs to be asked for data and sends data request for current step/feature (L2PU_DataRequest messages)
- on arrival of data from the ROS, it gathers the data concerning the RoI and executes Feature Extraction (FEX) algorithms;
- FEX algorithms may produce additional RoIs to be analyzed;
- once all steps are executed, the final decision is taken and the result send to the pROS in L2PU_LVL2Result message
- the result is sent to the Supervisor in L2PU_LVL2Decision message
- if there are any events pending in the queue, the worker task starts processing the new event taken from the head of the queue, otherwise it becomes idle.

We propose the following set of parameters for the L2PU model:

receive_request_time: time spent in the LVL1ResultHandler to get LVL1ResultMessage and put it in the queue; time spent for buffer management is accounted here

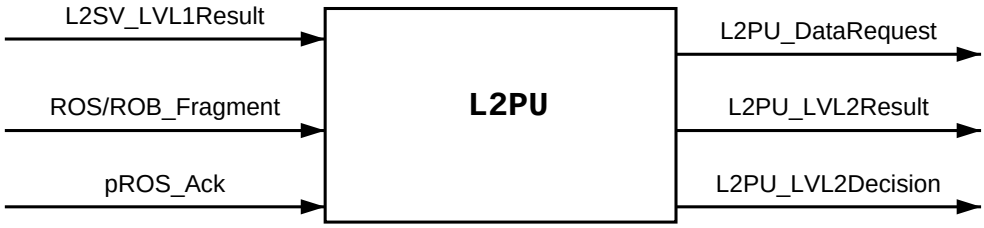
prepare_new_event_time: time needed by WorkerTask to get the LVL1ResultMessage from the queue and set up processing

prepare_collector_time: time needed to prepare the RoS requests infrastructure in the DataCollector::collect() - buffers allocation, cookie preparation, etc

send_ros_request: time needed to prepare and send single RoS request (RosRequestMessage), handler registration should be accounted here

get_ros_fragment_time: time needed to process the single incoming RoS fragment in DataCollector::MyInputHandler::message()

Figure 5: L2 Processing Unit related messages



unpack_ros_data_time: time needed to convert all received RoS fragments to RoB data, mainly the time spent in DataCollector::convert(); statistics updates, etc are accounted here as well

cpu_burn_time: the value of burnTime parameter for the step; this is the "dummy algorithm"; this parameter needs to be replaced by the set of FEX algorithms' times for the full-system simulation.

decision_time: time to calculate the result (e.g.: PESAResult()) + statistics updates

send_pros_result_time: time needed to prepare and send result message for the pROS: the LVL2ResultMessage; time required to set up the LVL2ResultReplyHandler should be accounted here

pros_ack_time: the time spent in the LVL2ResultReplyHandler after the Ack from pROS is received

send_sv_result_time: time needed to prepare and send result message to the Supervisor; Message::reply() workload is accounted here (i.e. creating header, serializing the payload, buffer allocation and deallocation, etc).

event_finalize_time: time needed by event cleanup routines

The parameters are also presented in Fig. 6.

5 ROSe model

The simplified diagram of the ROSe node model is presented in Figure 7. The ROSe model receives L2PU_DataRequests messages to supply data to the L2PU processors. Reception of the request triggers generation of the ROS/ROB_Fragment message with requested data.

If the DFM is set to run the PULL scenario the ROSe model receives messages SFI_DataRequest from SFIs to provide data for EventBuilding. Each request triggers reply with ROS/ROB_EventFragment message.

If the DFM runs in the PUSH scenario, the ROSe model receives the DFM_Decision message. The message carries a list of event IDs and corresponding numbers of SFIs. This informs

Figure 6: L2PU parameters

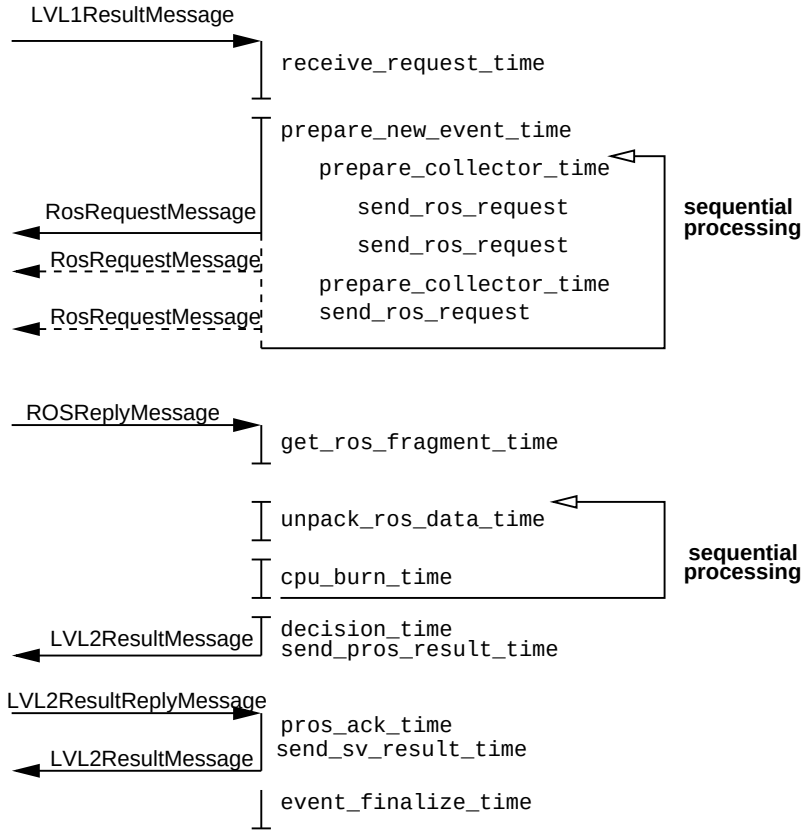
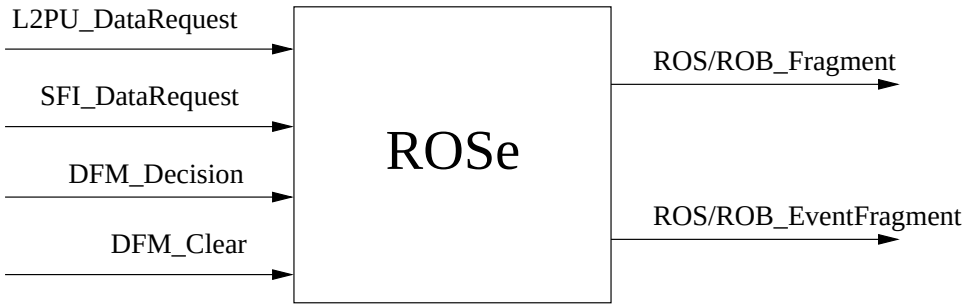


Figure 7: ROSe-related messages



the ROSe model which SFIs have been assigned events and triggers generation of one or more replies ROS/ROB_EventFragment directed to the assigned SFIs (one reply per SFI).

In both scenarios the ROSe receives the DFM_Clear messages with a list of events for which a place allocated in the data buffer may be freed.

The initial list of times is presented in Figure 8. The times need not to be a constant, they may be a function of the current ROS/ROB state (number of currently processed events, list of outstanding requests etc), as well as a function of computer's hardware (CPU speed, number of processors etc).

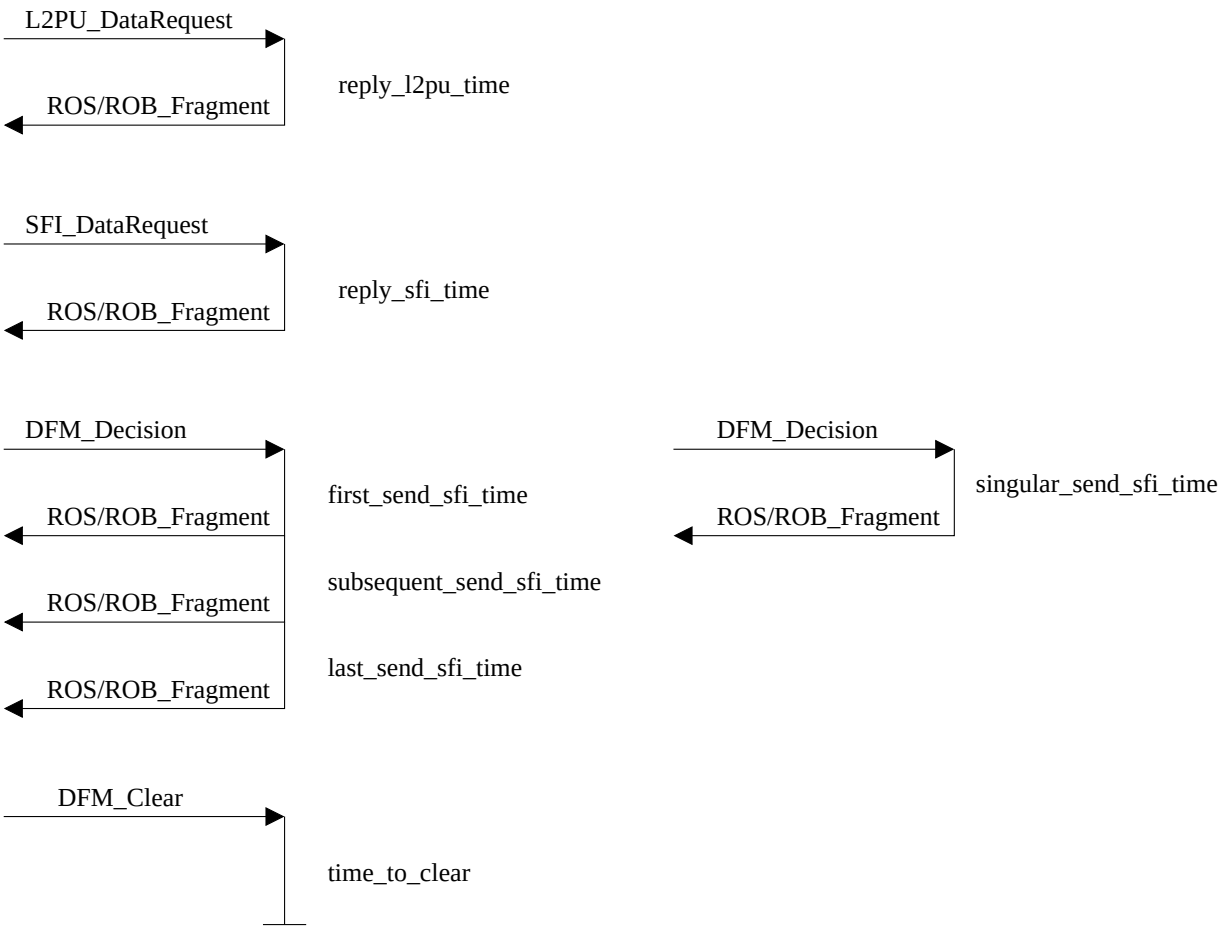
1. **reply_l2pu_time**: it is a time which elapses from the moment the L2PU_DataRequests is made known to the ROS/ROB application to the moment the ROS/ROB application completed sending the ROS/ROB_Fragment message (program control returned to application after the _send_).

2. **reply_sfi_time**: it is a time which elapses from the moment the SFI_DataRequest is made known to the ROS/ROB application to the moment the ROS/ROB application completed sending the ROS/ROB_EventFragment message (program control returned to application after _send_).

3. **first_send_sfi_time**: it is a time which elapses from the moment the DFM_Decision is made known to the ROS/ROB application to the moment the ROS/ROB application completed sending the first ROS/ROB_EventFragment message (program control returned to application after _send_). The **first_send_sfi_time** includes any additional processing the ROSe application performs related to the reception of the message DFM_Decision, upacking, preparing list of reply messages etc.

4. **subsequent_send_sfi_time**: it is a time which elapses from the moment the ROS/ROB application completed sending former ROS/ROB_EventFragment message, to the moment it completed sending the next ROS/ROB_EventFragment (program control returned to application after _send_).

Figure 8: ROSe parameters



5. **last_send_sfi_time**: it is a time which elapses from the moment the ROS/ROB application completed sending one but last ROS/ROB_EventFragment message to the moment it completed sending the last ROS/ROB_EventFragment message (program control returned to application after _send_). This time should include any additional processing related to the completion of the DFM_Decision message processing (for example: removing any additional data structures built after reception of the DFM_Decision message).

6. **singular_send_sfi_time**: it is a time which elapses from the moment the DFM_Decision is made known to the ROS/ROB application to the moment the application completed sending a single ROS/ROB_Fragment message (program control returned to application after _send_). The **singular_send_sfi_time** includes both extra time related to the reception of the DFM_Decision message and related to the completion of the message processing.

7. **clear_time**: it is a time necessary for ROS/ROB application to model clearing event slots in the data buffers.

5.1 Hardware ROB emulator models

There exist two types of hardware ROB emulators (HWROB): the FPGA-based and the Alteon-based emulators. Each of the FPGA emulator contains 32 FE ports. The Alteon-based emulators are realised as a custom firmware for the Gigabit Ethernet Alteon AceNIC network card.

The models of hardware ROB emulators follow the hardware implementation and merge functionality of the DC ROSe application with the Message-Passing subsystem. The models accept and generate the same messages as the ROSe model (see section 5), because the hardware emulators fully comply to the DC set of messages. The only exeption here is lack of modeling the clear_time. As the hardware ROB emulators do not keep any track of event numbers kept untill a clear message arrives, also models will not need to model time spent for processing the DFM_Clear message.

The initial list of times is based on the DC software ROS/ROB emulators models - see section 5 and Figure 8.

1. **reply_l2pu_time**: it is a time which elapses from the moment the L2PU_DataRequests message arrived to the HWROB node to the moment when the ROS/ROB_Fragment reply starts to be sent off the node.

2. **reply_sfi_time**: it is a time which elapses from the moment the SFI_DataRequests message arrived to the HWROB node to the moment when the ROS/ROB_Fragment reply starts to be sent off the node.

3. **first_send_sfi_time**: it is a time which elapses from the moment the DFM_Decision message arrived to the HWROB node to the moment when the first ROS/ROB_EventFragment reply starts to be sent off the node. The **first_send_sfi_time** includes an additional processing (if any) by the HWROB related to the reception of the message.

4. **subsequent_send_sfi_time**: it is a time the HWROB node needs to produce subsequent ROS/ROB_EventFragment reply. The time to produce a subsequent reply may be irrelevant as the hardware emulators can produce replies faster than the time needed by the network to transfer the former message.

5. **last_send_sfi_time**: it is a time the HWROB node needs to produce last ROS/ROB_EventFragment reply. This time should include an additional bookeeping processing (if any) related to the completion of the message processing.

6. **singular_send_sfi_time**: it is a time which elapses from the moment the DFM_Decision messsage arrived to the HWROB node to the moment when the single ROS/ROB_Fragment reply starts to be sent off the node. The **singular_send_sfi_time** includes bookeeping time related to reception of the DFM_Decision message (if any), and also related to the completion of the message processing (if any).

6 pROS model

The simplified diagram of the pROS model is presented in Figure 9. The pROS receives L2PU_LVL2Result messages from the L2PUs with more detailed information on event being processed by the L2PUs. The pROS model acknowledges reception of the message by generation of the pROS_Ack message. The other operations performed by the pROS model are identical to operations performed by the ROS models related to the Event Builder activities.

The pROS model receives the DFM_Decision message from the DFM when the latter runs in the PUSH scenario. From the message the pROS retrieves an event IDs and the destination address of an SFIs where event data should be sent to. The pROS sends it's data in the ROS/ROB_EventFragment message(s).

In case the DFM runs in the PULL scenario, it sends event assignment to a SFI and the pROS model receives from that SFI the request message SFI_DataRequest to supply data. The pROS model replies with the ROS/ROB_EventFragment message.

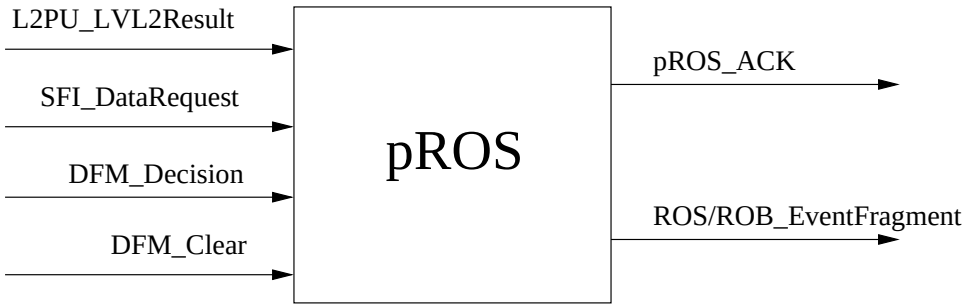
The event data stored in the pROS model can be cleared after reception of the DFM_Clear message.

The initial list of times is presented in Figure 10. The times need not to be a constant, they may be a function of the current pROS state (number of currently processed events etc), as well as a function of computer's hardware (CPU speed, number of processors etc).

1. **ack_l2pu_time**: it is a time which elapses from the moment the L2PU_LVL2Result message is made known to the pROS application to the moment the pROS sent the pROS_Ack message (program control returned to the application after _send_) and completed processing the L2PU_LVL2Result message.

2. **reply_sfi_time**: it is a time which elapses from the moment the SFI_DataRequest is made known to the pROS application to the moment the pROS application completed processing the SFI_DataRequest message (program control returns to application after _send).

Figure 9: pROS-related messages



3. **first_send_sfi_time**: it is a time which elapses from the moment the DFM_Decision is made known to the pROS application to the moment the pROS application completed sending the first ROS/ROB_EventFragment message (program control returns to application after _send_). The **first_send_sfi_time** includes any additional processing the pROS application performs related to the reception of the DFM_Decision message.

4. **subsequent_send_sfi_time**: it is a time which elapses from the moment the pROS application completed sending former ROS/ROB_EventFragment message, to the moment it completed sending a next ROS/ROB_EventFragment message (program control returns to application after _send_).

5. **last_send_sfi_time**: it is a time which elapses from the moment the pROS application completed sending one but last ROS/ROB_EventFragment message to the moment it completed sending the last ROS/ROB_EventFragment (program control returns to application after _send). This time should include any additional processing related to the completion of the DFM_Decision message processing (for example clearing any temporary data structures created at reception of the message).

6. **singular_send_sfi_time**: it is a time which elapses from the moment the DFM_Decision is made known to the pROS application to the moment the application is ready to pass a single ROS/ROB_Fragment message to the Message-Passing subsystem. The **singular_send_sfi_time** includes bookeeping time related to reception of the DFM_Decision message but also related to the completion of the message processing.

7. **clear_time**: it is a time necessary for pROS application to model clearing event slots in the data buffers.

7 DFM model

The simplified diagram of the DFM node model is presented in Figure 11. The DFM model receives two types of messages: L2SV_LVL2Decision and SFI_EoE. Reception of a message

Figure 10: pROS parameters

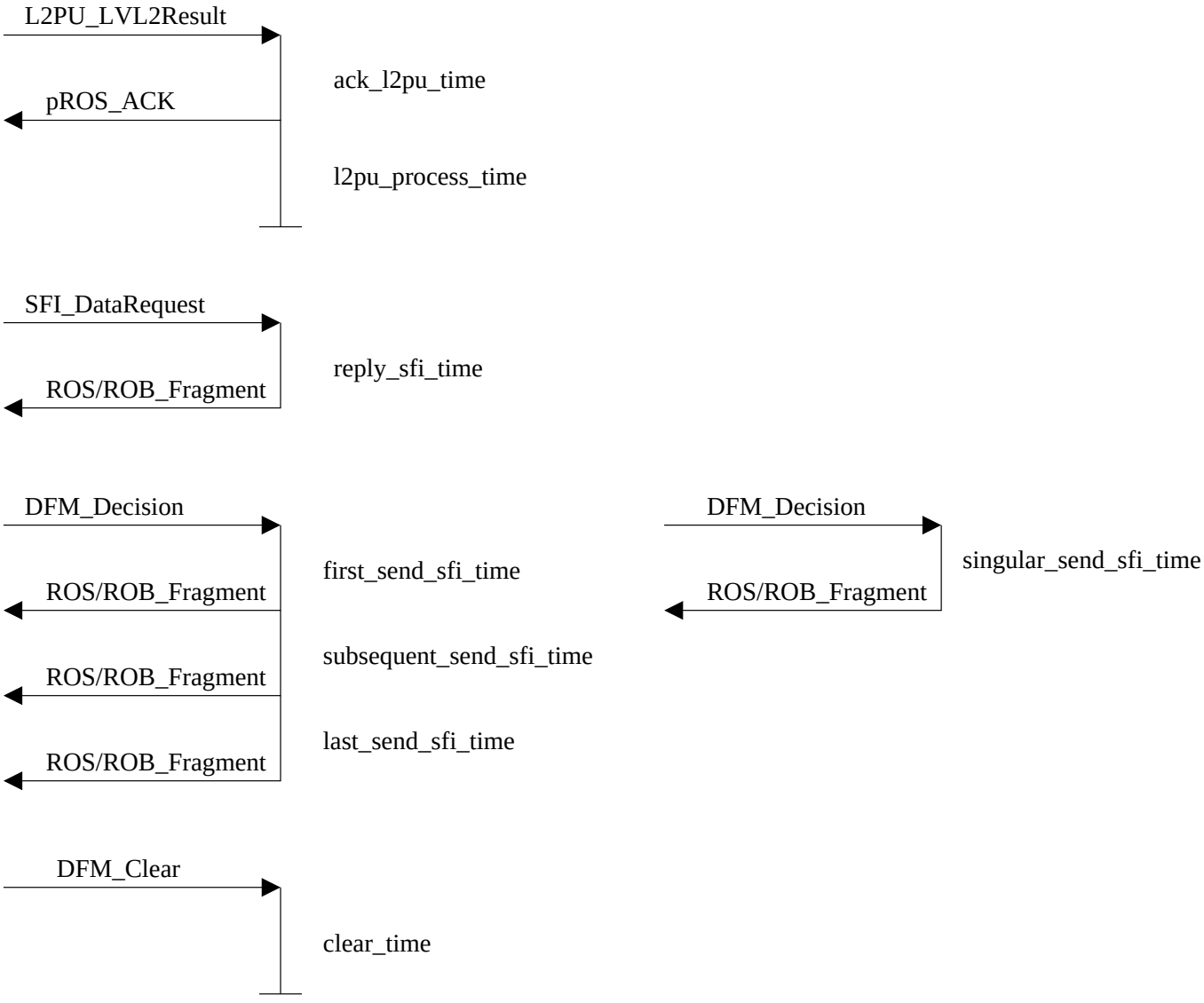
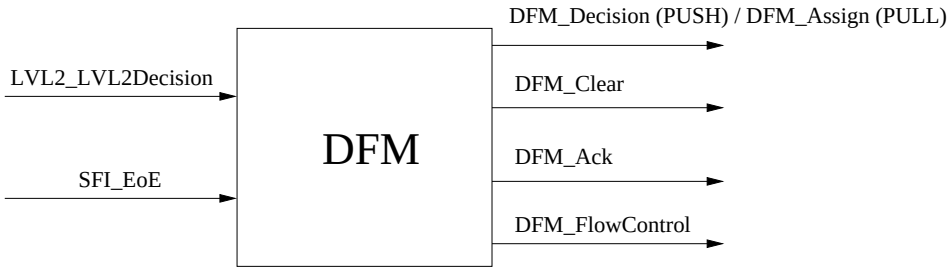


Figure 11: DFM-related messages



changes state of the DFM which may result in generation of outgoing message.

In the L2SV_LVL2Decision message, the LVL2 Supervisor sends lists of events with LVL2 decisions (either reject or continue processing). In the PUSH scenario, the DFM may or may not produce the DFM_Decision message(s). The decision whether to produce the DFM_Decision message and if so then how many, depends on a number of positively flagged events in the L2SV_LVL2Decision and also on an internal grouping performed inside the DFM. When the number of IDs of positively flagged events, accumulated inside the DFM, exceeds a group limit, the message is produced. The DFM_Decision message with ID of event(s) is directed to all ROS/ROBs (including pseudoROB) informing them on destination SFI(s) which have been assigned to run the Event Building for events listed in the message.

In the PULL scenario, the DFM may or may not produce the DFM_Assign messages. The decision whether to produce the DFM_Assign message(s) depends on a number of positively flagged events in the L2SV_LVL2Decision. For each positively flagged event from the list one DFM_Assign message is produced and directed to a SFI which has been assigned by the DFM to perform the Event Building.

The IDs of events with negative flag from the L2SV_LVL2Decision message will be added to an internal list and if the number of IDs in this list exceeds grouping, the DFM will generate the DFM_Clear message.

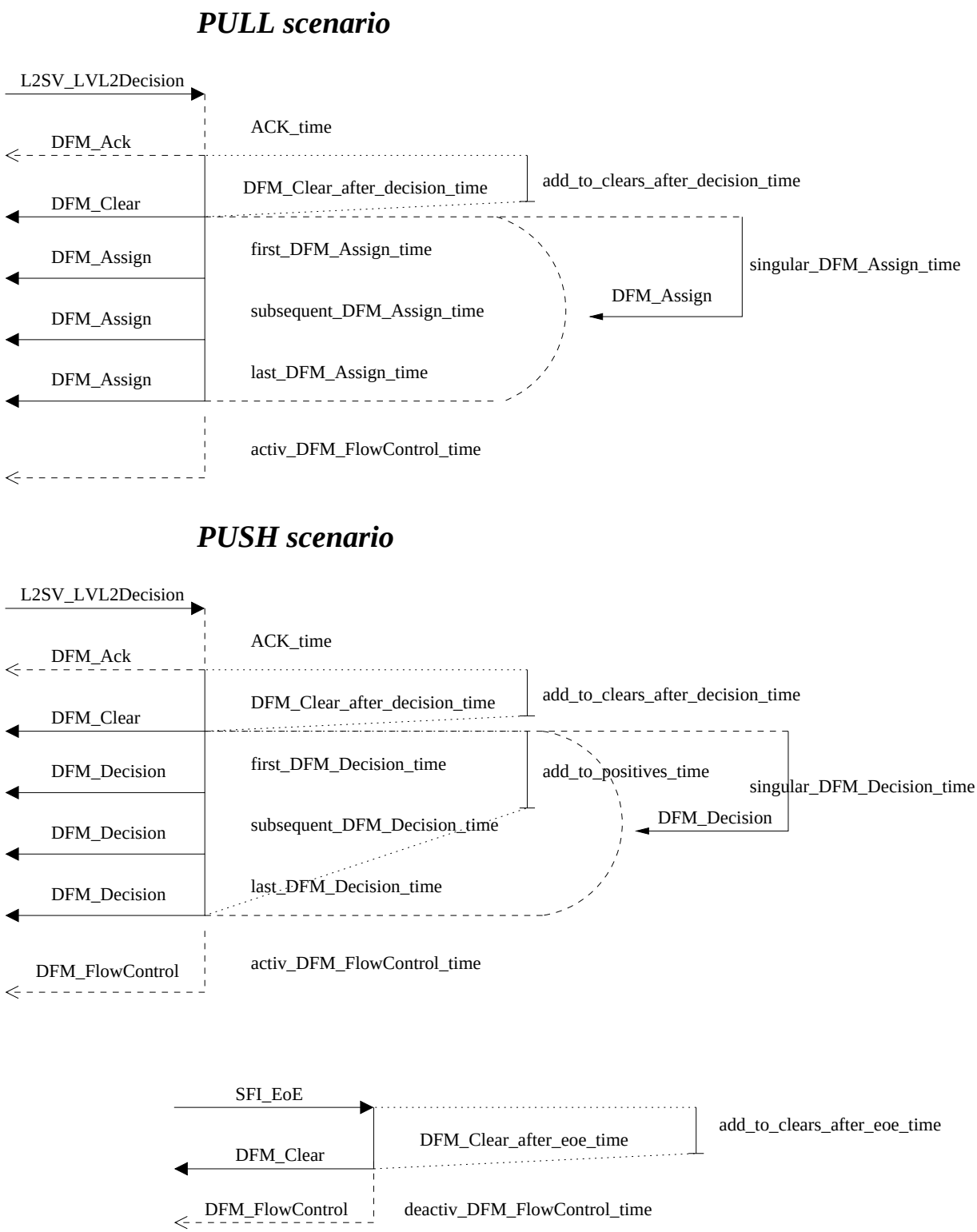
The SFI_EoE message informs the DFM that the event flagged positively for further processing has been completed by the SFI and sent to an Event Filter farm. On reception of this message the DFM adds the event ID to the list of events which should be removed from the ROS/ROB buffers. If the newly added ID exceeds grouping limit it triggers generation of the DFM_Clear message.

The DFM produces the DFM_Ack message to confirm reception of the L2SV_LVL2Decision.

The DFM produces the DFM_FlowControl message in case it has reached the processing capability (watermark in a queue of IDs of events waiting for processing) and needs to stop further events assignment. The parameter associated with the DFM_FlowControl message defines a time necessary to hold any new assignments. In case the DFM manages to process some of the queued events and can accept new assignments it produces the DFM_FlowControl message with a parameter enabling new assignments.

The initial list of times is presented in Figure 12. The times need not to be a constant, they

Figure 12: DFM parameters



may be a function of the current DFM state (number of currently processed events, number of events with positive/negative decisions in the L2SV_LVL2Decision message etc), as well as a function of computer's hardware (CPU speed, number of processors etc). To allow exploration of various traffic shaping ideas some times, especially the **subsequent_assign_time** and **subsequent_decision_time** may be set as a parameter when the original DC DFM code executes an initialization phase. For proper modeling we need to know the actual time to produce the message (an externally defined parameter together with an overhead added in the code to produce the message).

1. **ACK_time**: time which elapses from the moment the L2SV_LVL2Decision is made known to the DFM application to the moment the DFM completes sending the DFM_Ack message (program control returns to application after _send_).

2. **DFM_Clear_after_decision_time**: time which elapses from the moment the DFM has produced the DFM_Ack message to the moment the DFM compled sending the DFM_Clear message (program control returns to application after _send_).

3. **add_to_clears_after_decision_time**: time which elapses from the moment the DFM has produced the DFM_Ack message to the moment the DFM finished processing the L2SV_LVL2Decision for negatively flagged events and added them to internal list without producing the DFM_Clear message.

4. **first_DFM_Assign_time**: it is a time which elapses from the moment the DFM finished processing L2SV_LVL2Decision for the negatively flagged events (including sending the DFM_Clear message if decided to do so) to the moment the DFM completed sending the DFM_Assign message (program control returns to application after _send_) (for DFM in PULL scenario).

5. **subsequent_DFM_Assign_time**: time from the moment the DFM has produced former DFM_Assign message to the moment the DFM completed sending a subsequent DFM_Assign message (program control returns to application after _send_) (for DFM in PULL scenario). More than one DFM_Assign message will be sent if the L2SV_LVL2Decision message will contain more than one event with positive decision.

6. **last_DFM_Assign_time**: time from the moment the DFM has produced former DFM_Assign message to the moment the DFM completed sending the last DFM_Assign message (program control returns to application after _send_) (for DFM in PULL scenario). Time for generation of the last DFM_Assign message includes additional processing needed to complete processing of the L2SV_LVL2Decision message.

7. **singular_DFM_Assign_time**: time from the moment the DFM finished processing L2SV_LVL2Decisio for the negatively flagged events (including sending the DFM_Clear message if decided to do so), to the moment the DFM completed sending a single DFM_Assign message (program control returns to application after _send_) (for DFM in PULL scenario). The **singular_DFM_Assign_time** includes both the additional time related to reception of the

L2SV_LVL2Decision message and the time related to completion of the message procesing.

8. **activ_DFM_FlowControl_time**: time from the moment the DFM begins verification of it's status and realizes that is has reached processing capabilities (watermark in a queue of events waiting for processing) to the moment the DFM completed sending the DFM_FlowControl message (program control returns to application after _send_), requesting the LVL2_SV suspention of a new events assignment.

9. **first_DFM_Decision_time**: time from the moment the DFM completed processing negatively flagged events from the L2SV_LVL2Decision message to the moment the DFM completed sending the first DFM_Decision message (program control returns to application after _send_) (for DFM in PUSH scenario). The **first_DFM_Decision_time** includes any additional time the DFM needs to perform operations related to reception of the L2SV_LVL2Decision message.

10. **subsequent_DFM_Decision_time**: time from the moment the DFM has produced former DFM_Decision message to the moment the DFM completed sending the subsequent DFM_Decision message (program control returns to application after _send_) (for DFM in PUSH scenario). More than one DFM_Decision message will be sent if the L2SV_LVL2Decision message will contain more than one event with positive decision. The **last_DFM_Decision_time** should be used to model the last DFM_Decision.

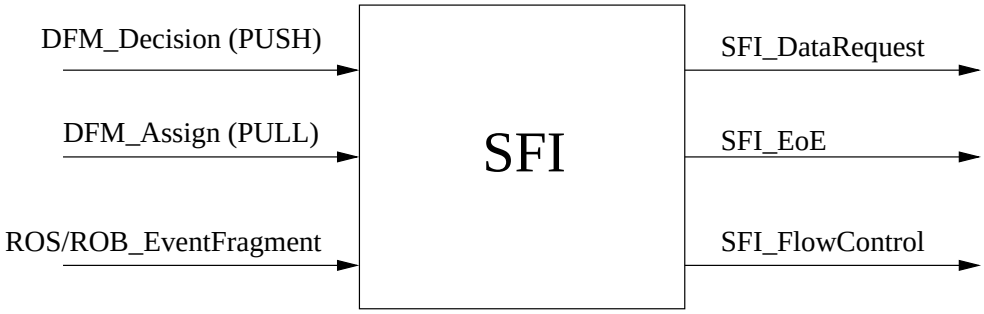
11. **last_DFM_Decision_time**: time from the moment the DFM has produced former DFM_Decision message to the moment the DFM completed sending the last DFM_Decision message (program control returns to application after _send_) (for DFM in PUSH scenario). The **last_DFM_Decision_time** should include any additional processing the DFM will perform to complete processing of the L2SV_LVL2Decision message.

12. **singular_DFM_Decision_time**: time from the moment the DFM completed processing negatively flagged events from the L2SV_LVL2Decision message to the moment the DFM completed sending a single DFM_Decision message (program control returns to application after _send_) (for DFM in PUSH scenario). The **singular_DFM_Decision_time** includes both the additional time needed to reception of the L2SV_LVL2Decision and the time needed to perform completion of the L2SV_LVL2Decision message.

13. **add_to_positives_time**: time spent by the DFM from the moment it completed procesing negatively flagged events from the L2SV_LVL2Decision message to the moment the positively flagged events are added to the internal queue without producing the DFM_Decision message (too few positively flagged events to reach grouping factor and produce a message).

14. **DFM_Clear_after_eoe_time**: time which elapses from the moment the SFI_EoE message is made known to the DFM to the moment the DFM completed sending the DFM_Clear message (program control returns to application after _send_).

Figure 13: SFI-related messages



15. **add_to_clears_after_eoe_time**: it is a time which elapses from the moment the SFI_EoE message is made known to the DFM to the moment the DFM finished processing the message and added the completed event to internal list without producing the DFM_Clear message.

16. **deactiv_DFM_FlowControl_time**: it is a time which elapses from the moment the DFM begins verification of it's status and realizes that is has regained some space to resume new events assignment to the moment the DFM completed sending the DFM_FlowControl message (program control returns to application after _send_).

8 SFI model

The simplified diagram of the SFI model in presented in Figure 13. The model of the SFI receives the DFM_Assign message (in the PULL scenario). The message informs the SFI that an event (ID listed in the message) has been assigned to it and it has to perform event building collecting fragments from all ROS/ROSs (including the pseudoROB). The completed event has to be sent to the Event Farm for further processing. The SFI starts sending one or more SFI_DataRequest messages to request data from the ROS/ROB buffers. After data from all buffers have been collected in the SFI it sends the SFI_EoE message to the DFM.

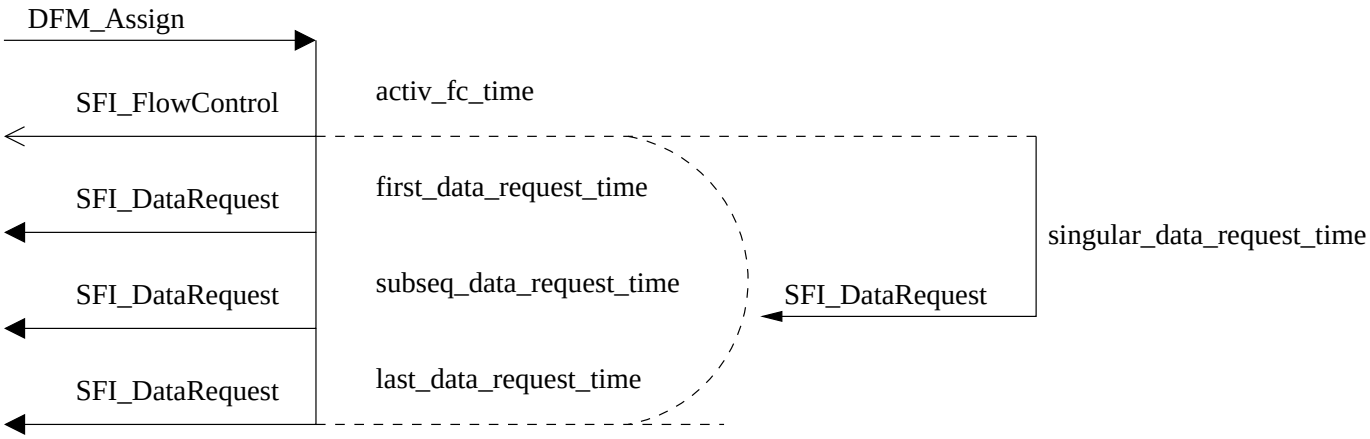
In the PUSH scenario the SFI waits for data from the ROB/ROS buffers (ROS/ROB_EventFragment messages) as they received a message from the DFM with a number of SFI which has been assigned to perform event building. After data from all buffers have been collected the SFI produces the SFI_EoE message to the DFM.

In case the SFI can not cope with the rate of assigning events, it can generate the SFI_FlowControl message to stop assigning any more events. The suspended events assignment may be released when the SFI completes an event(s) and regains ability to take another event. In such case it can send the SFI_FlowControl message to the DFM for a new assignment.

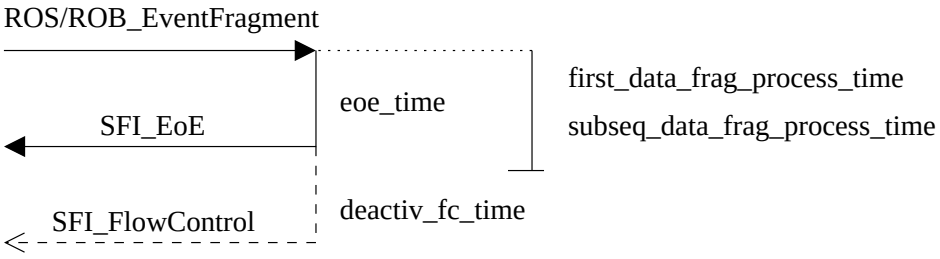
The initial list of times is presented in Figure 14. The times may be a function of the current SFI state (for example a number of currently processed events) as well as a function of computer's hardware (for example: CPU speed, number of processors etc.). To allow exploration of

Figure 14: SFI parameters

PULL scenario



PUSH and PULL (cont.) scenario



various traffic shaping ideas some times, especially **subseq_data_request_time** may be set as a parameter, when the original DC SFI code executes initialization phase. For proper modeling we need to know the actual time to produce a message (externally defined parameter together with an overhead added in the code to produce a message).

1. **activ_fc_time**: time from the moment the SFI_Assign message is made known to the SFI to the moment the SFI completed sending the SFI_FlowControl message (program control returned to the application after _send_). The SFI_FlowControl message is used to stop DFM sending more events to the SFI.

2. **first_data_request_time**: time the moment the SFI passed a check to rise flow control (which may result in generation of SFI_FlowControl message) to the moment the SFI completed sending the first SFI_DataRequest message (program control returned to the application after _send_) (for DFM in PULL model). The **first_data_request_time** includes time needed to setup event building for a new event. This time should be used in case the SFI_DataRequest message will be sent as a multicast.

3. **subseq_data_request_time**: time from the moment the SFI sent the former SFI_DataRequest message to the moment the SFI completed sending the subsequent SFI_DataRequest message (program control returned to the application after _send_).

4. **last_data_request_time:** time from the moment the SFI sent the former SFI_DataRequest message to the moment the SFI completed sending the last SFI_DataRequest message (program control returned to the application after _send_). This time should include any additional time (if any) needed by the SFI to complete process of data requests generation.

5. **singular_data_request_time**: time from the moment the SFI passed a check to rise flow control (which may result in generation of SFI_FlowControl message) to the moment the SFI completed sending a single SFI_DataRequest message (multicast?) (program control returned to the application after _send_) (for DFM in PULL model). The **singular_data_request_time** includes both the additional time related to reception of the SFI_Assign message and the additional time related to completion of the SFI_Assign message (for example clearing temporary data structures created at the reception of the message).

6. **first_data_frag_process_time**: time from the moment the first ROS/ROB_EventFrag message is made known to the SFI to the moment the SFI completed processing the message.

7. **subseq_data_frag_process_time**: it is time which elapses from the moment the subsequent ROS/ROB_EventFrag message is made known to the SFI to the moment the SFI completed processing the message.

8. **eoe_time**: it is time which elapses from the moment the last ROS/ROB_EventFrag message is made known to the SFI to the moment the SFI completed sending the SFI_EoE message (program control returned to the application after _send_).

9. **deactiv_fc_time**: time from the moment the SFI finished sending data to an Event Filter farm (processor) to the moment the SFI completed sending the SFI_FlowControl message (program control returned to the application after _send_). The SFI_FlowControl message is used to inform the DFM that the SFI is ready to accept more events.

References

[1] Ptolemy `http://ptolemy.eecs.berkeley.edu/`

[2] "Modeling large Ethernet networks for the ATLAS high level trigger system using parameterized models of switches and nodes.", P. Golonka, K. Korcyl, F. Saka; CERN-OPEN-2001-061; poster presented on RT 2001 Conference, Valencia

[3] "An ATLAS TDAQ Candidate architecture"; R. W. Dobinson, K. Korcyl, M. LeVine; DC note 49

[4] "Linux network performance measurements" - DC note in preparation

[5] "A high-resolution time-stamping library for the Trigger and Data Acquisition System"; V.Perez et al.; DC note 48

[6] "Message Format"; H. P. Beck, F. Wickens; DC Note 22