Time survey on embedded software using remote IP log system: a GPON study case

Afonso Augusto Romão V. Alvim
afonso.alvim@hotmail.com

Fernanda Yumi Matsuda
Aline Cristine Fadel
André Berti Sassi
Rodrigo De Almeida Moreira
CPqD Foundation
Rod Campinas - Mogi-Mirim (SP-340) - km 118,5
13086-902 - Campinas, SP - Brazil
{ fmatsuda, afadel, asassi, rmoreira }@cpqd.com.br

Marcos Perez Mokarzel
Instituto Nacional de Telecomunicações - Inatel
P.O. Box 05 - 37540-000
Santa Rita do Sapucaí - MG - Brazil
mpmoka@inatel.br

Abstract—**Many telecommunications equipment must be able to deal with different terminals simultaneously, therefore they must have some sort of CPU time sharing. Converting tasks in messages to send to a fair scheduler is widely employed for that, however understanding CPU times is still a challenge. In this paper we propose a new scheme to remotely recover CPU resources information from embedded software using logger over Ethernet UDP/IP. We propose three log tickets, one when a message enters a message queue, other when a message exits the queue and the last when the software finishes message processing. With these three new tickets, we can calculate the time interval that a message waits in the queue and the time that the message takes to be processed. We implemented this log scheme in a real GPON OLT successfully, obtaining precision of 1 ms. With this new system, we were able to quickly identify time issues such as thread invasion and messages that must have higher priority.**

Index Terms— **Remote log, time survey, inter-process queue, processing time, CPU occupation.**

I. Introduction

Telecommunication systems, mainly in access networks, are composed by one aggregator and a set of terminals. Besides, all equipment must provide means to operators to configure the system and, nowadays, most of equipment can be configured by different ways at the same time. Therefore, aggregator's CPU (central processing unit) must share time among many different processes. These cases characterize a multitask client/server system, where the operators are the clients and the aggregator's CPU is the server that must fulfill the requests from different terminals and operators.

Regardless computational method employed in the time sharing, it is important that all the resources (time, memory, etc.) fit some values for proper functioning of the whole system [1], [2]. Particularly, some of these resources are especially important:

- *CPU utilization:* This is the percentage of the CPU time in use per time unit. The main goal is to maintain the CPU busy as long as possible [3].
- *Throughput:* Number of processes (or messages) executed per time unit [3].
- *Turnaround time:* Time spent by the CPU for each processing [3].
- *Waiting time:* Time that each process waits in the queue to start being processed by the CPU.
- *Response time:* Time to the system to produce the task result.
- *Memory utilization:* Amount of memory allocated in the process.

When it comes to embedded software, getting this information is a very difficult task. Most of the time, these equipment have no operating system (OS) or the OS and/or the CPU do not provide the required information. On the other hand, if the system provides it, sometimes the system may not correctly identify who is the owner of the process that is causing the problem, since OS doesn't know the software's internal division.

Providing a real-time and non-intrusive (RTNI) [4] way to get this information might be very helpful. In this paper, we propose a new scheme to remotely recover CPU timing information using a remote log system that runs over UDP/IP. This scheme is not totally RTNI, but the results are promising. Tests were made in an OLT (optical line terminal) of GPON (Gigabit-capability passive optical networks) [5] with success. Issues such as thread invasion and prioritization were quickly identified and fixed.

In the next section, we describe how time sharing must be to work out with our method. In Section III, we describe the log scheme, how to create and how to process tickets. In Section IV, we present our real implementation in an OLT and show the collected results. Results are evaluated in Section V. Finally, in Section VI, conclusions are presented and some future work is proposed.

II. Scheduler

One may find in technical literature many different ways to share time between tasks using the same CPU. Our OLT uses tasks in a queue, which means that all new tasks are converted to a message in a queue of the scheduler as shown in Fig. 1. This method is widely employed in telecommunication equipment.

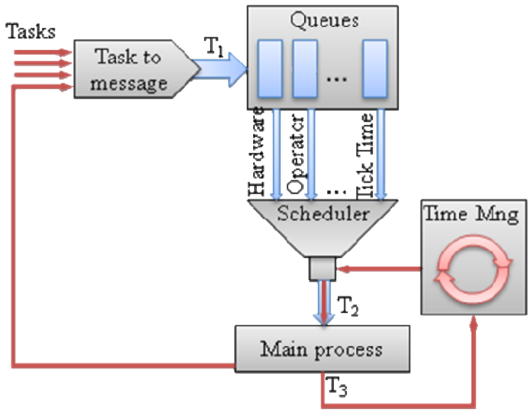


Fig. 1. Example of scheduler block diagram.

All events received by the CPU through hardware interruption are replaced by a new message stored in the message queue, represented in Fig. 1 by the arrow T_1. The system should have more than one priority queue. When a new message is created, it should be assigned to a specific priority (or queue number). Continuously, the CPU gets the next message from one of the queues and processes it. The processing may also generate other messages to be stored in the queues.

A scheduler must decide from which queue it will take the next message. Different criteria can be employed to make this decision, such as "Shortest Job First" and "Round Robin" [6].

The main advantage of our method is that all heavy processing is confined in the same thread, which reduces problems with concurrent processes and also the time spent with context swapping. On the other hand, an infinite loop can cause a deadlock in the equipment.

All routines that process messages must be "on the fly", i.e., these routines must take one message, execute their work as quick as possible and free the CPU. If the time to process the message is too long, other processes will be delayed, possibly causing problems. In this case, processes longer than a predetermined interval must be separated in a set of shorter processes.

III. Log Scheme

A. Logger block diagram

The log system is composed by one component, called Logger, compiled with the embedded application (in our example, an OLT GPON), and a standalone application, called WinLogger, that collects and processes ticket information in a remote computer. Logger's block diagram is seen at Fig. 2.

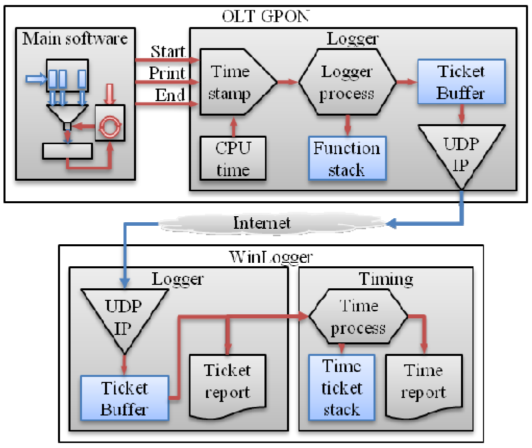


Fig. 2. GPON Logger block diagram.

Main embedded software may create tickets in the Logger using three methods:

- *Start:* This method must be called at the beginning of any function that intends to use Logger. It provides to Logger the function name e some log attributes.
- *Print:* This method is called for any new ticket. It receives the ticket type and a formatting string and its arguments.
- *End:* This method must be called at the end of any function that uses Logger. It informs to Logger that all information about the function is no longer necessary and it must be popped from the function stack inside Logger.

As soon as these methods are called, Logger retrieves current time from the OS and creates a new ticket with that. It is important to preserve time information as close as possible to the event time. Our OLT OS provides time information with 1ms of precision.

Logger process completes the ticket with the function name and a sequence number. Then, it sends the ticket to WinLogger through UDP/IP. A ticket buffer is necessary once tickets can be created faster than the Ethernet port data transfer rate.

WinLogger is responsible for remotely process logs. It has a ticket buffer to store all tickets sent by Logger. A Timing component compiles tickets to generate time reports.

B. Ticket structure

Ticket structure is shown in Fig. 3. Each ticket has only one event information and is wrapped in a UDP/IP packet.

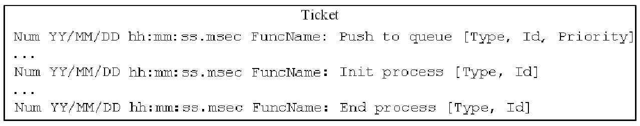

```
                              Ticket
Num YY/MM/DD hh:mm:ss.msec FuncName: Push to queue [Type, Id, Priority]
...
Num YY/MM/DD hh:mm:ss.msec FuncName: Init process [Type, Id]
...
Num YY/MM/DD hh:mm:ss.msec FuncName: End process [Type, Id]
```

Fig. 3. Ticket structure

Where:

`Num` → Ticket sequential number;

`YY` → Event year;

`MM` → Event month;
`DD` → Event day;
`hh` → Event hour;
`mm` → Event minute;
`ss` → Event second;
`msec` → Event millisecond (precision = 1ms);
`FuncName` → Name of the function that generates the ticket;
`Type` → Event type. It identifies who has generated that message, for instance, CLI (command-line interface) or OMCI (ONT management and control interface);
`Id` → Message identification, used to correlate tickets to the same message;
`Priority` → Queue or priority number.

There is no retransmission. Packet loss can be identified by the sequential number "`Num`", but no recovery system was implemented due to the high data volume. Therefore, Time process block in Fig. 2 must be prepared to resolve packet lost.

C. Time survey tickets

Using above structures, three new tickets were created to time survey (Fig. 3). These tickets are created using Logger's Print method:

- *Push to queue:* This ticket corresponds to the event of new message pushed into the queue. In Fig. 1, this event is represented by T_1.
- *Init process:* This ticket corresponds to the event of message popped from the queue and starting to be processed by the main process. In Fig. 1, this event is represented by T_2.
- *End process:* This ticket corresponds to the event of main process has finished to process message. In Fig. 1, this event is represented by T_3.

All tickets have an Id to identify a unique task that is used for synchronization. In the *Push to queue*, a Priority parameter identifies each priority queue that is used to store the task. It is important to follow priorities in scheduler and verify if it is correctly working.

D. Time processing

As discussed in section I, there are five relevant CPU times to be monitored. Considering T_1 the time in ticket *Push to queue*, T_2 the time in ticket *Init process* and T_3 the time in *End process*. For each task, *turnaround time* and *waiting time* can be calculated using Eqs. 1 and 2, respectively.

$$T_{Turnaround} = T_3 - T_2 \tag{1}$$

$$T_{Waiting} = T_2 - T_1 \tag{2}$$

CPU utilization and *throughput* need a time interval or time unit. We use a fixed interval of 1s. Therefore, *CPU utilization* for interval *k*, in percentage, may be calculated by Eq. 3. It is the ratio between all $T_{Turnaround}$ in a time interval by the time interval itself.

$$CPU = \frac{\sum_{n=N_k}^{N_{k+1}-1} T_{Turnaround}(n)}{\Delta T_{Interval}} \times 100 \tag{3}$$

Where:

N_k is the sequential number of the first *Init process* ticket in interval *k*;

N_{k+1} is the sequential number of the first *Init process* in interval $k+1$, which is 1 unit greater than the last ticket in interval *k*;

$\Delta T_{Interval}$ is the time interval;

$T_{Turnaround}(n)$ is the *turnaround time* for *Init process* ticket with sequential number *n*, calculated by Eq. 1.

Eq. 4 represents the *throughput* for any time interval as the number of *Init process* tickets in the interval. Note that some intervals may not process any message; in this case, their *throughput* will be zero.

$$throughput = N_{k+1} - N_k \tag{4}$$

Response time depends on task result. In this case, a table, such as Table I, must be provided to WinLogger associating the task with its expected result. This table will be used to measure the *response time* of the system.

TABLE I
EXAMPLE OF TASK AND RESULTS TABLE FOR *RESPONSE TIME* TRIGGER.

Command	Result
al	Link activated
aco	ONU activated
adf	Ethernet flow activated
dl	Link deactivated
cdf	Ethernet flow configured
rdf	Ethernet flow removed

This table uses two messages as shown in Fig. 4. A *Received* ticket identifies a command sent by the operator, which is considered the start of the task. An *Event sent* ticket identifies the answer sent by the application to operator, indicating the end of the task.

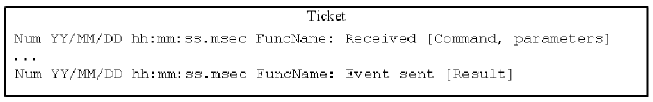

```
Ticket
Num YY/MM/DD hh:mm:ss.msec FuncName: Received [Command, parameters]
...
Num YY/MM/DD hh:mm:ss.msec FuncName: Event sent [Result]
```

Fig. 4. Trigger messages for *response time*

Thus, *Response time* is:

$$T_{ResponseTime} = T_{EventSent} - T_{Received} \tag{6}$$

Where:

$T_{EventSent}$ is the time of *Event sent* ticket;

$T_{Received}$ is the time of *Received* ticket.

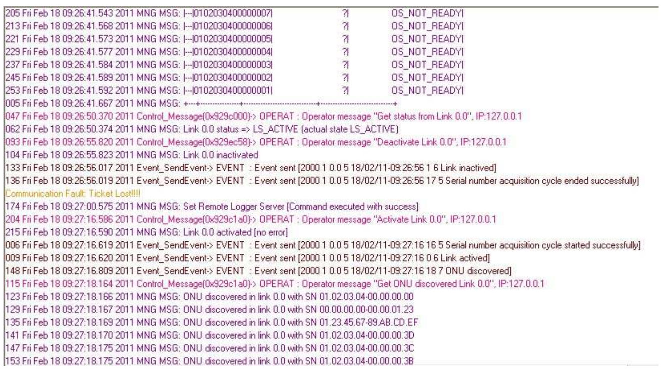
IV. Tests and results

A. Test infrastructure

In order to ensure the operation of the scheme proposed in this paper, we've implement it in a real GPON system. In this system, we monitor our OLT CPU usage. The CPU is responsible for 1024 ONUs (optical network units). Operators may use multiples CLI terminals and SNMP (Simple Network Management Protocol) to operate, administrate and maintain the system via Ethernet port or serial RS-232. All of these terminals send their tasks to the OLT CPU, which converts the tasks in messages as can be seen in Fig. 1.

OLT software was written in C language and Logger component was compiled together. Logger uses a single UDP/IP port to send its tickets to a remote PC.

WinLogger (Fig. 5), the software in the remote PC, is responsible for dealing with the tickets, saving and processing them and displaying reports. This must be done in real time, when tickets arrive from Logger or in batch, using saved tickets. In order to generate graphs, WinLogger saves these data in tables that can be opened in many different spreadsheets.



```
205 Fri Feb 18 09:26:41.543 2011 MNG MSG: |---|0102030400000007|                ?|          OS_NOT_READY|
213 Fri Feb 18 09:26:41.568 2011 MNG MSG: |---|0102030400000006|                ?|          OS_NOT_READY|
221 Fri Feb 18 09:26:41.573 2011 MNG MSG: |---|0102030400000005|                ?|          OS_NOT_READY|
229 Fri Feb 18 09:26:41.577 2011 MNG MSG: |---|0102030400000004|                ?|          OS_NOT_READY|
237 Fri Feb 18 09:26:41.584 2011 MNG MSG: |---|0102030400000003|                ?|          OS_NOT_READY|
245 Fri Feb 18 09:26:41.589 2011 MNG MSG: |---|0102030400000002|                ?|          OS_NOT_READY|
253 Fri Feb 18 09:26:41.592 2011 MNG MSG: |---|0102030400000001|                ?|          OS_NOT_READY|
005 Fri Feb 18 09:26:41.667 2011 MNG MSG: +---+---------------+--------------------+--------------------+
047 Fri Feb 18 09:26:50.370 2011 Control_Message(0x929c000)-> OPERAT : Operator message "Get status from Link 0.0", IP:127.0.0.1
062 Fri Feb 18 09:26:50.374 2011 MNG MSG: Link 0.0 status => LS_ACTIVE (actual state LS_ACTIVE)
093 Fri Feb 18 09:26:55.820 2011 Control_Message(0x929ec58)-> OPERAT : Operator message "Deactivate Link 0.0", IP:127.0.0.1
104 Fri Feb 18 09:26:55.823 2011 MNG MSG: Link 0.0 inactivated
133 Fri Feb 18 09:26:56.017 2011 Event_SendEvent-> EVENT  : Event sent [2000 1 0.0 5 18/02/11-09:26:56 1 6 Link inactived]
136 Fri Feb 18 09:26:56.019 2011 Event_SendEvent-> EVENT  : Event sent [2000 1 0.0 5 18/02/11-09:26:56 17 5 Serial number acquisition cycle ended successfully]
Communication Fault: Ticket Lost!!!!
174 Fri Feb 18 09:27:00.575 2011 MNG MSG: Set Remote Logger Server [Command executed with success]
204 Fri Feb 18 09:27:16.586 2011 Control_Message(0x929c1a0)-> OPERAT : Operator message "Activate Link 0.0", IP:127.0.0.1
215 Fri Feb 18 09:27:16.590 2011 MNG MSG: Link 0.0 activated [no error]
006 Fri Feb 18 09:27:16.619 2011 Event_SendEvent-> EVENT  : Event sent [2000 1 0.0 5 18/02/11-09:27:16 16 5 Serial number acquisition cycle started successfully]
009 Fri Feb 18 09:27:16.620 2011 Event_SendEvent-> EVENT  : Event sent [2000 1 0.0 5 18/02/11-09:27:16 0 6 Link actived]
148 Fri Feb 18 09:27:16.809 2011 Event_SendEvent-> EVENT  : Event sent [2000 1 0.0 5 18/02/11-09:27:16 18 7 ONU discovered]
115 Fri Feb 18 09:27:18.164 2011 Control_Message(0x929c1a0)-> OPERAT : Operator message "Get ONU discovered Link 0.0", IP:127.0.0.1
123 Fri Feb 18 09:27:18.166 2011 MNG MSG: ONU discovered in link 0.0 with SN 01.02.03.04-00.00.00.00
129 Fri Feb 18 09:27:18.167 2011 MNG MSG: ONU discovered in link 0.0 with SN 00.00.00.00-00.00.01.23
135 Fri Feb 18 09:27:18.169 2011 MNG MSG: ONU discovered in link 0.0 with SN 01.23.45.67-89.AB.CD.EF
141 Fri Feb 18 09:27:18.170 2011 MNG MSG: ONU discovered in link 0.0 with SN 01.02.03.04-00.00.00.3D
147 Fri Feb 18 09:27:18.175 2011 MNG MSG: ONU discovered in link 0.0 with SN 01.02.03.04-00.00.00.3C
153 Fri Feb 18 09:27:18.175 2011 MNG MSG: ONU discovered in link 0.0 with SN 01.02.03.04-00.00.00.3B
```

Fig. 5. Tickets in WinLogger

Note in Fig. 5, one "Ticket Lost!" message that was purposely generated to show how WinLogger deals with this problem. Some events are shown in this image too, which are used as task results for *response time*.

B. Test methodology

The operator may configure GPON using OLT's CLI. In our tests we've employed scripts to help us to keep coherence between tests. Scripts' times are not precise, but they are precise enough for our tests once we have some messages as triggers such as those mentioned in Fig. 4.

Besides our embedded application, OLT's CPU has other software running in background (including OS) that may interfere in the tests results. To minimize it, we've run the same tests many times and have taken average values.

We've defined the scripts to include many different operations with different needs. Some GPON instructions spend more CPU time, while others need more network traffic. Our main script follows the sequence bellow.

1. Activate Link 1.
2. Create two ONUs in Link 1.
3. Activate ONU 1.
4. Activate ONU 2.
5. Create 10 Ethernet flows in ONU 1.
6. Create 10 Ethernet flows in ONU 2.
7. Activate all Ethernet flows in ONU 1.
8. Activate all Ethernet flows in ONU 2.
9. Save all configurations.
10. Get ONU 1 configuration.
11. Deactivate all ONUs in Link 1.
12. Deactivate Link 1.

The Table II shows commands at a log and their elapsed time. These commands can be easily synchronized with CPU resources information.

TABLE II
OPERATORS' COMMANDS IN LOG WITH ITS ELAPSED TIME.

Elapsed Time	Operator Command
16221	Sat Jan 1 00:02:19.189 2000 Control_Message()-> OPERAT : "Save OLT configurations", IP:XML File
31625	Sat Jan 1 00:02:34.593 2000 Control_Message()-> OPERAT : "Activate Link 0.0", IP:10.4.1.98
33029	Sat Jan 1 00:02:35.997 2000 Control_Message()-> OPERAT : "New ONU 0.0.1", IP:10.4.1.98
35065	Sat Jan 1 00:02:38.033 2000 Control_Message()-> OPERAT : "Activate ONU 0.0.1", IP:10.4.1.98
60077	Sat Jan 1 00:03:03.045 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.1.2", IP:10.4.1.98
61137	Sat Jan 1 00:03:04.105 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.1.3", IP:10.4.1.98
62213	Sat Jan 1 00:03:05.181 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.1.4", IP:10.4.1.98
63276	Sat Jan 1 00:03:06.244 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.1.5", IP:10.4.1.98
64328	Sat Jan 1 00:03:07.296 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.1.6", IP:10.4.1.98
65387	Sat Jan 1 00:03:08.355 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.1.7", IP:10.4.1.98
66472	Sat Jan 1 00:03:09.440 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.1.8", IP:10.4.1.98
67526	Sat Jan 1 00:03:10.494 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.1.9", IP:10.4.1.98
68578	Sat Jan 1 00:03:11.546 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.1.10", IP:10.4.1.98
69637	Sat Jan 1 00:03:12.605 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.1.11", IP:10.4.1.98
70707	Sat Jan 1 00:03:13.675 2000 Control_Message()-> OPERAT : "Activate Ethernet 0.0.1.0", IP:10.4.1.98
80735	Sat Jan 1 00:03:23.703 2000 Control_Message()-> OPERAT : "New ONU 0.0.2", IP:10.4.1.98
82741	Sat Jan 1 00:03:25.709 2000 Control_Message()-> OPERAT : "Activate ONU 0.0.2", IP:10.4.1.98
107822	Sat Jan 1 00:03:50.790 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.2.12", IP:10.4.1.98
108877	Sat Jan 1 00:03:51.845 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.2.13", IP:10.4.1.98
109936	Sat Jan 1 00:03:52.904 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.2.14", IP:10.4.1.98
111029	Sat Jan 1 00:03:53.997 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.2.15", IP:10.4.1.98
112077	Sat Jan 1 00:03:55.045 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.2.16", IP:10.4.1.98
113131	Sat Jan 1 00:03:56.099 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.2.17", IP:10.4.1.98
114189	Sat Jan 1 00:03:57.157 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.2.18", IP:10.4.1.98
115248	Sat Jan 1 00:03:58.216 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.2.19", IP:10.4.1.98
116327	Sat Jan 1 00:03:59.295 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.2.20", IP:10.4.1.98
117379	Sat Jan 1 00:04:00.347 2000 Control_Message()-> OPERAT : "New Ethernet 0.0.2.21", IP:10.4.1.98
118431	Sat Jan 1 00:04:01.399 2000 Control_Message()-> OPERAT : "Activate Ethernet 0.0.2.0", IP:10.4.1.98
128470	Sat Jan 1 00:04:11.438 2000 Control_Message()-> OPERAT : "Get Info Application ", IP:10.4.1.98
129521	Sat Jan 1 00:04:12.489 2000 Control_Message()-> OPERAT : "Deactivate ONU 0.0.1", IP:10.4.1.98
130601	Sat Jan 1 00:04:13.569 2000 Control_Message()-> OPERAT : "Deactivate ONU 0.0.2", IP:10.4.1.98
131736	Sat Jan 1 00:04:14.704 2000 Control_Message()-> OPERAT : "Deactivate Link 0.0", IP:10.4.1.98
132748	Sat Jan 1 00:04:15.716 2000 Control_Message()-> OPERAT : "Save OLT configurations ", IP:10.4.1.98

In order to test priority queues, two priority queues were employed. The highest priority is reserved to an internal periodic process. Messages that give command feedback to operator and hardware feedbacks were delivered using the lowest priority queue.

The tests were executed using two different scheduler algorithms: a strict priority algorithm, where the messages in higher priority queues are delivered first, and a round-robin algorithm without priority. The period used for the periodic process, or tick time, was also changed during the tests. Our default tick time is 50ms, but we've tested 25ms and 100ms as well, resulting in six applications composing our test set.

C. Test result

Running the script above, we've captured log tickets using WinLogger. Each test produces graphs of *turnaround* and *waiting time*.

The graph in Fig. 6 shows *turnaround* and *waiting time* for 50ms of tick time. The marks for each operator command show when these commands were sent.

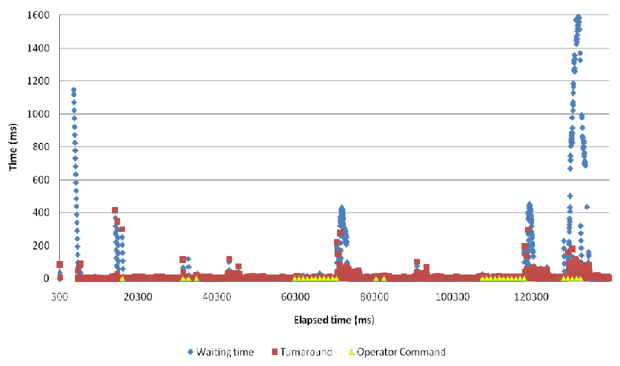


Fig. 6. *Turnaround* and *waiting time* for 50ms version.

Fig. 7 shows the last five commands from the graph in Fig. 6. And Fig. 8 shows *waiting time* histogram for high and low priority queues. In this histogram, tickets were grouped in portions of 10ms.

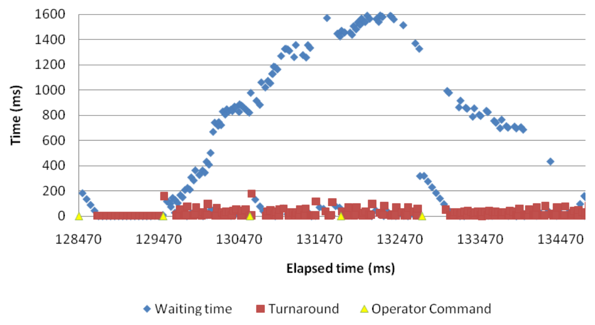


Fig. 7. *Turnaround* and *waiting time* for last five commands for 50ms version.

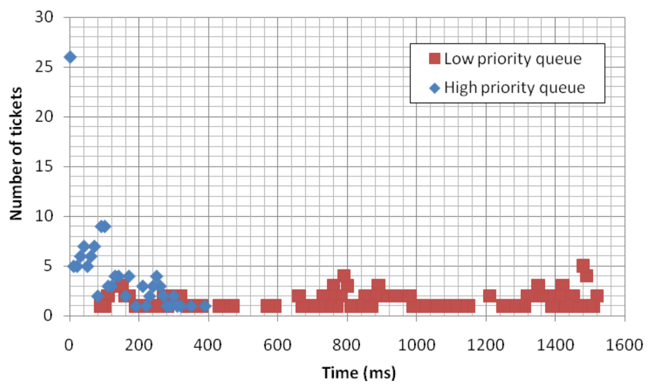


Fig. 8. *Waiting time* histogram for the last five commands in 50ms version. Curves for Q1 (high priority queue) and Q2 (low priority queue).

Comparing 25ms, 50ms and 100ms tick time versions, for the last five commands, Fig. 9 shows three *waiting times*, and Fig. 10 shows three *turnaround times*.

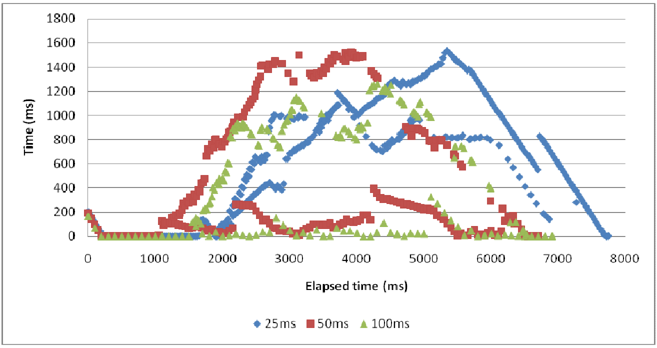


Fig. 9. *Waiting times* for 25ms, 50ms and 100ms tick time versions.

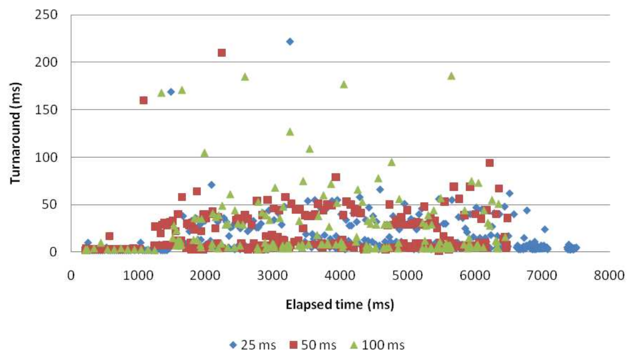


Fig. 10. *Turnaround times* for 25ms, 50ms and 100ms Tick Time versions.

Fig. 11 shows the *throughput* in the CPU. Tests were made with 25ms, 50ms and 100ms versions. To improve visualization, Fig. 12 and Fig. 13 show, respectively, *CPU utilization* and *throughput* for last five commands.

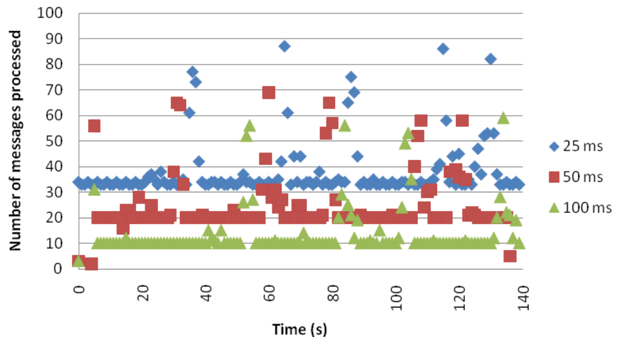


Fig. 11. *Throughput* for 25ms, 50ms and 100ms versions.

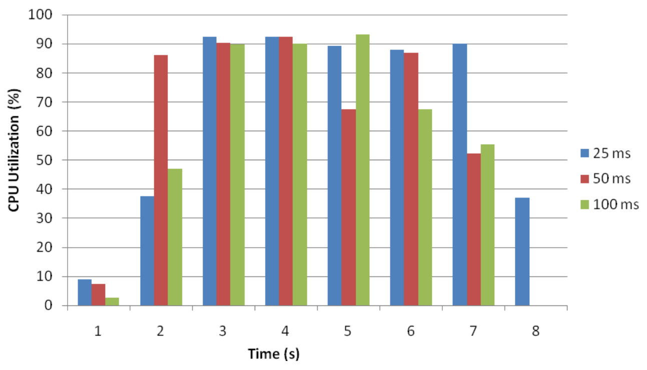


Fig. 12. *CPU utilization* for last five commands in 25ms, 50ms and 100ms versions.

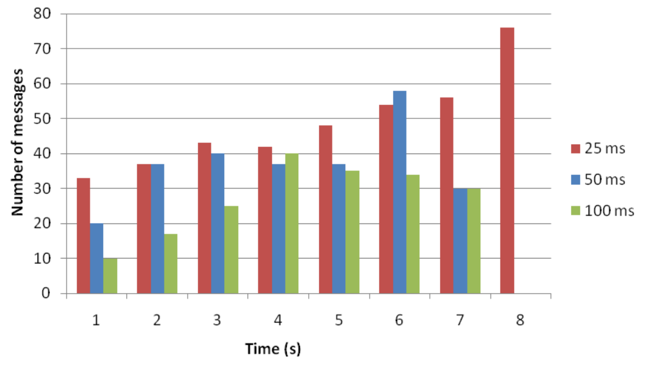


Fig. 13. *Throughput* for last five commands in 25ms, 50ms and 100ms versions.

The last measure obtained by our log scheme was CPU *response time*. For that, we've used the last five commands in our script as the start trigger and the event "*OLT configuration saved*" as the expected result. After running the script four times, we've obtained the following results: the average time was 6.688s; the minimum time was 6.519s; and the maximum time was 6.782s.

V. Results Evaluation

A. Turnaround and waiting time

Evaluating the results presented in the previous section is possible to verify that the proposed scheme is very useful and some important information can be inferred from the CPU time. Fig. 6 shows *Waiting* and *turnaround time* for the entire test, marks were plotted for each operator command associating cause and effect, which is important for debugging and troubleshooting.

In that experiment, a 50ms tick time was used, then any *turnaround time* greater than this value may represent a performance issue and must be checked by the developers. In order to solve this kind of problem, messages taking more than 50ms may be split in two or more messages that take less than 50ms to be processed.

Concerning *waiting time*, we've got values greater than 1.5s. It might be a problem, depending on how fast the system is expected to respond to a command and even on memory limitations, as the queue length tends to increase significantly in this situation. The last four commands, better visualized in Fig. 7, were sent to the OLT before the previous command had been completed, causing message accumulation in the queues and, consequently, increasing the *waiting time*.

Processing a message may create other messages and ONU deactivation command is one of the commands that triggers a lot of hardware callbacks, which are the reason why, after ONU deactivation, approximately at 130s in Fig. 7, the *waiting time* starts to increase.

The priority queue efficiency can be analyzed using the *waiting time* histogram in Fig. 8. While the highest priority queue histogram spreads from 100ms to 1500ms, the lowest priority histogram is concentrated below 400ms. The probability is 65% to stay under than 100ms, 81% under than 200ms and 97% under than 300ms, even in an overloaded situation.

B. Comparing tick time values

The tick time must be correctly dimensioned once it has high impact over equipment performance. Three tick time values were tested: 25ms, 50ms and 100ms. In the overloaded situation of the last five commands, looking at the Fig. 9, the software behavior differs on each version. The worst case, considering *waiting time*, occurred at the 25ms version, in which queues have peaks greater than 1s. Besides, OLT response time increases 15%.

At each tick, main process can drive more than one action. For instance, if two ONUs must be checked in the next tick time, these checks will happen in the same tick. For this reason, a small tick time reduces the number of actions to be processed in the same call. Consequently, *turnaround* time is reduced, as shown in Fig. 10. However, comparing with other versions, this reduction is not significant.

Both versions 50ms and 100ms have a good *waiting time* behavior, as shown in Fig. 9, nevertheless 100ms version is better, as high and low priority queues remain lower, compared to 50ms version, during almost the entire test. Considering *turnaround time* in Fig. 10, 100ms version has eight high amplitude peaks (greater than 100ms), instead of two at the 50ms version. It happens whereas 100ms version accumulates many actions to be processed in the tick time call. These peaks are dangerous and must be avoided.

When the OLT is idle, OS and tick time keep the CPU running, creating a minimum amount of messages in a 1s interval. This amount is directly related to the tick time, thus at least 10 messages are created for 100ms, at least 20 for 50ms, and at least 40 for 25ms. It is also remarkable that the 25ms version does not have the required throughput to process the message that arrives at all tick times, once its *throughput* is less than 40 messages, as shown in Fig. 11. The 50ms and 100ms versions have enough throughputs to handle the tick time messages.

C. Coherence in CPU utilization

As mentioned in Section I, the scheduler was supposed to keep the CPU as busy as possible during heavy load. In the test equipment, OS and other processes use approximately 10% of CPU time. In Fig. 12, it is possible to notice that in all versions, CPU utilization reaches approximately 90%, which shows that, for this parameter, the scheduler is properly fulfilling its role.

On the operator's viewpoint, the system continues accepting commands and answering even under heavy load, i.e., degradation occurs in a coherent way (the system does not collapse). It occurs even with the low throughput of 100ms version shown in Fig. 13.

Another way to analyze scheduler coherence using our method is to compare *waiting time* curves for multiple instances of an experiment. The resulting curves of all instances should have almost the same behavior, and commands should have almost the same *response time*. This curve may be used to compare different versions, as in Fig. 9.

VI. Conclusion

In this paper, we propose a new scheme to collect and analyze CPU resources in an embedded software using a log system. Tickets are sent to a remote computer through UDP/IP over Ethernet. When each ticket is created, a timestamp, with 1ms of precision, is attributed and the ticket is stored in a local buffer. When embedded software is idle, tickets in the buffer are sent to another computer where they will be stored and processed.

The queue input and output, and the end processing delimit the path of the messages in a multitask software scheduler. Using the time information embedded on these tickets, we can directly calculate *turnaround* and *waiting time*. With some indirect processing, the *CPU utilization*, the *throughput* and the *response time* may be calculated too.

An OLT GPON was used as a study case; data was collected and the analysis helped to identify the better tick time interval, 50ms. A histogram of *waiting time* shows high and low priority queues behaviors with all high priority messages processed with less than 400ms while low priority reaches 1.5s.

Analyzing *CPU utilization* and *response time* in heavy load, we've noted that the system kept running without any problem, even when *CPU utilization* reaches 90%.

For future work, more tests could be done with other equipment, employing the same log system. Other improvements in log processing may involve the creation of log tickets to report memory allocation. Memory utilization must be monitored to identify issues such as memory leak or to predict when equipment will run out of memory.

References

[1] K. Ramamritham and J.A. Stankovic, *Scheduling Algorithms and Operating Systems Support for Real-Time Systems.* Proc. IEEE, vol. 82, no. 1, pp. 55-67, Jan. 1994.

[2] T. Yen and W. Wolf, *Performance Estimation for Real-Time Distributed Embedded Systems*, IEEE Trans. on Parallel and Distrib. Syst., vol. 9, no. 11, pp. 1125-1136, Nov. 1998.

[3] F. Machado and L. Maia, "Arquitetura de Sistemas Operacionais," 2nd Edition, Livros Técnicos e Científicos, 1997.

[4] R. Freyer, *FPGA Based CPU Instrumentation for Hard Real-Time Embedded System Testing.* SIGBED Rev., vol. 2, no. 2, pp. 39-42, Apr. 2005.

[5] Gigabit-capable Passive Optical Networks (GPON), ITU-T Rec. G.984, 2003.

[6] H. M. Deitel, "Operating Systems," 2nd Edition, Addison Wesley, 1990.