An Efficient Timer Implementation For HPR

Subir Varma
Bala Rajaraman

IBM Corporation
PO Box 12195
Research Triangle Park, NC 27709

Contents

Figure List

Abstract

High Performance Routing (HPR) is a new transport/network level protocol to be be used in IBM SNA networks. One of its most significant features is end-to-end rate based flow control, which replaces window based flow control used in the present SNA architecture. It also has end-to-end error recovery and route switch mechanisms. All these new features require the use of a number of different timers. Since previous SNA architectures did not require such an extensive use of timers, timer design is a critical aspect for a successful HPR implementation. In this report we discuss this problem and suggest some ways in which timers can be efficiently implemented.

Section 1. Introduction

High Performance Routing (HPR)(4), is a new transport/network level protocol to be be used in IBM SNA networks. One of its most significant features is end-to-end rate based flow control, which replaces window based flow control used in the present SNA architecture. It also has end-to-end error recovery and route switch mechanisms. All these new features require the use of a number of different timers. Since previous SNA architectures did not require such an extensive use of timers, timer design is a critical aspect for a successful HPR implementation.

The main focus on this report will be on the efficient implementation of the burst timer (B_T), that is used to control the rate R_s at which PIUs are being transmitted into the network (3). The component of HPR that controls the rate at which PIUs are being sent into the network is called ARB, and it works by varying the rate R_s in reponse to changing network conditions. For example if the network is congested, then R_s is reduced, while if the source requires more bandwidth, and the network has some available, then R_s is increased. ARB controls the rate at which data is being sent into the network, by sending no more than B_s bits during an interval B_T, such that the following equation is satisfied

$$R_S = \frac{B_S}{B_T}$$

Hence R_s can be controlled either by varying B_s or by varying B_T.

The rest of this report is organized as follows: In Section 2 we describe some typical scenarios for HPR, and the resulting timer values that they lead to. This has a bearing on the values that the burst timer is set to. In Sections 3 and 4 we discuss some simple implementations of the timer, while in Sections 5 and 6 we present new improved schemes.

Section 2. Scenarios

The value of B_T in a system can vary widely, depending upon the type of application that is using the services of the HPR pipe. Assuming that the packet size is fixed at 1 KB, consider the following two scenarios:

1. OLTP application, with a data rate of 10 KB/s. This leads to $B_T = 100ms$.
2. High speed file transfer application, with a data rate of 8 mbps. This leads to $B_T = 1ms$.

Any real system will have a mixture of these two types of applications, so that the values of B_T are expected to vary widely. Also the number of RTP pipes supported by the host can be quite large, as the following calculation shows: Assume that it takes 6000 instructions to send a PIU, then for the OLTP application, this translates into 60,000 instructions/s per application (since each application sends 10 PIUs/s). Assuming that 10% of the MIPS of a 50 MIPS processor are devoted to sending data, this translates into a total of 83 RTP pipes that can be supported by the host. If 50% of the MIPS are devoted to sending data, then it can support 416 RTP pipes. The corresponding number of pipes for the high speed file transfer applications are 0 and 4. These calculations show that a good timer implementation for HPR should account for the fact that the timer values can vary widely as well as the number of connections using timers, can be very large.

There are two main costs involved in timer implementations:

- A timer implementation is basically a sorting system, since its main function is to receive as input timer requests for different intervals and produce as ouput all these requests sorted according to their interval values. Hence one of the costs of a timer implementation is due to sorting of the timer control blocks (TCBs).
- The other cost is due to the dispatching overhead of the timer process itself. This overhead can be large either when the time intervals requested are small, or

if the number of timers is large. For example for the case when there are 16 RTP pipes with $B_T = 1ms$, there is a timer expiry every 62.5 microseconds, while for the case when there are 416 RTP pipes with $B_T = 100ms$, there is a timer expiry every 240 microseconds.

Hence a good timer implementation should be able to reduce the expense of TCB sorting as well as the expense of the timer process dispatches.

Section 3. The Naive Implementation

The simplest implementation of the timer structure is to have a simple linked list of TCBs, arranged according to their time of expiry. The sorting cost of this structure of $O(n)$, and is incurred every time a TCB is inserted into the linked list. The timer is set according to the value of the first TCB that is to expire, so that there is a timer dispatch with every TCB expiry. This can lead to an excessive amount of MIPS being devoted to the timer process dispatch. The sorting cost can be reduced to $O(\log(n))$ by using data structures such as heaps and left-leaning trees, however the cost of timer dispatches still remains high. For the case of the OLTP application there is one time dispatch on the average every 240 microseconds, which translates into a requirement of 2.5 MIPS (assuming 600 instructions are required to dispatch the timer process). The corresponding number for the file transfer application is 2.4 MIPS.

Section 4. The Clock-Tick Approach

The clock-tick approach tries to reduce the cost of excessive number of timer dispatches by periodically dispatching it once every clock-tick interval, which is usually fixed. At these times, all the TCBs are decremented by the clock-tick value, and those that have expired are then dispatched. The cost of doing the update every clock tick is now $O(n)$, however the cost of inserting a new TCB is $O(1)$. There is also a certain amount of im-precision introduced due to the fact that certain timers may now expire at a time slightly later than their set value. For example, for the OLTP case if we set the clock-tick interval $T = 5ms$, then there can be an error of at most 5 ms between the set and the actual value of timer expiry. The timer dispatch overhead for this case would be 0.12 MIPS. In order to get a comparable 5% error margin for the file transfer application, one would require that $T = 50$ microseconds. However this makes the clock-tick solution for high speed applications even more expensive than the linked-list solution (which led to a timer dispatch every 250 microseconds).

Hence we are led to conclude that the clock tick approach is more suitable than the linked list approach for OLTP applications, while neither of them seems to be suitable for high data rate applications. We now present a modification of the clock-tick approach, to make it more suitable for high speed applications, by taking advantage of the relation

$$R_s = \frac{B_s}{B_T}$$

The basic idea behind this approach is to suitably modify B_s in order to achieve a certain rate R_s. Since the burst-timer is always reset as soon as it expires, all the burst timers belonging to a particular set can be synchronized so that they are always set at the clock-tick instant. However depending upon time interval values, they expire at different times. For example if $T = 1ms$ and $0 < B_T < 1ms$, then the

effective value of B_T, let us call it B_T^{ef}, will always be greater than the set value. In order to compensate for this, we can increase B_s to B_s^{ef}, such that

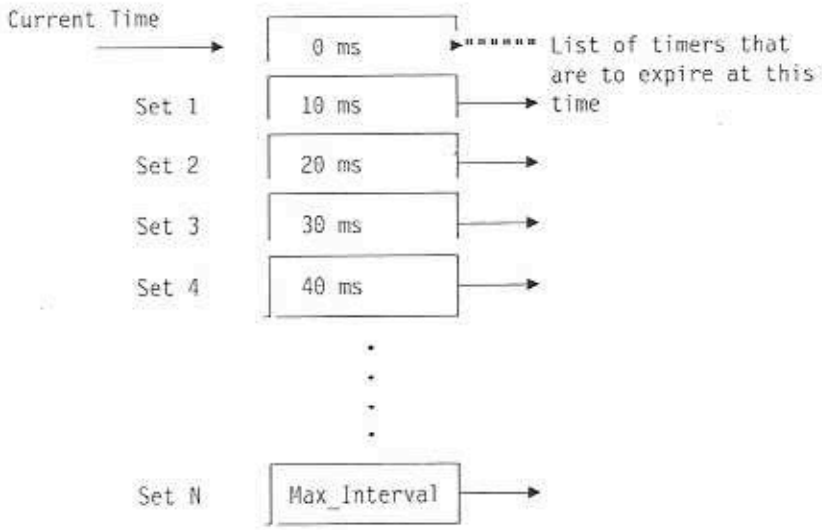
$$B_s^{ef} = \frac{B_T^{ef}}{B_T} B_s$$

Section 5. An Hierarchical Clock-Tick approach

In this section we present an approach that utilizes multiple sets of timers, with each set arranged according to the clock-tick design. Specifically, we propose that there be three timer sets, such that $T = 1ms$ for the first set, $T = 5ms$ for the second set and $T = 10ms$ for the third set. Sets 1 and 2 are organized in the conventional way as described in the last section, while set 3 is organized according to a timing wheel structure, which is described later in this section. The followings rules should be used in order to decide the set to which a timer should belong:

- If $0ms < B_T < 2.5ms$ then choose set 1.
- If $2.5ms \leq B_T < 7.5ms$ then choose set 2.
- If $B_T \geq 7.5ms$ then choose set 3.

We now describe the timing wheel which is the proposed data structure for set 3.



The basic idea behind the timing wheel technique is to sort the TCBs into different sets, depending upon the time of expiry. For example in Fig. 1, all TCBs for whom $7.5ms \leq B_T \leq 10ms$, are in Set 1, those for which $10ms \leq B_T \leq 20ms$ are in Set 2 etc.. Furthermore, there is a pointer that points to the current time, and at every clock-

tick it is incremented so that it points to the next set. At any instant of time, when a new TCB is inserted into this structure, the appropriate set is choosen according to the time of expiry for that TCB. If we know in advance the value of maximum burst time value, Max_Interval, then the number of sets required is given by $\frac{Max\_Interval}{10}$. The advantage of this scheme is that the insertion time and the per-tick book-keeping time are both $O(1)$ operations.

Some of the reasons for choosing the hierarchical clock-tick structure with timing wheels, are:

- The scan time per clock-tick interval is reduced as compared to the usual clock-tick approach. This is because of the timing wheel structure for normal values of B_T. For small values of B_T, a scan is still required, but since the number of applications which lie in this high speed range is limited, the scan overhead is small.

- The timer dispatch overhead is reduced, since the clock-tick value is not allowed to decrease below $1ms$, the difference being adjusted by changing B_c. Moreover, if there are no TCBs in the 1 ms or 5ms sets, then the clock-tick can be adjusted to $10ms$, thus reducing the timer dispatch overhead even further.

Section 6. Conclusions

The proposed burst timer design solves the central two problems for an efficient implementation: It optimizes the TCB insertion overhead as well as the overhead of the timer process dispatches.

6.1 References

[1] G. Varghese and T. Lauck, "Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility", *Proc. 11th Symp. Oper. Syst. Principles*, 25-38, (1987).

[2] D.E. Knuth, "The art of computer programming, Volume 3", *Addison Wesley, Reading, MA* (1973).

[3] L. Huynh and R. Bird, "High Performance Routing", *AWP-xxxx, IBM Corporation* (1993).

[4] E. Mumprecht, D. Gantenbein and R. Hauser, "Timers in OSI protocols: Specification vs implementation", *Undates Research Report, IBM Zurich Research Lab.*