Implementing Transactions in a Distributed Real-Time System without Global Time

A. Burns and Y. Chen
Department of Computer Science
University of York, UK
email: burns@cs.york.ac.uk

***Abstract*—A simple algorithm is presented for implementing and analysing real-time transactions executing on a distributed platform. The algorithm does not require global time, but does not suffer from excessive jitter.**

I. Introduction

In distributed real-time systems it is necessary to implement application code as *transactions* that incorporate processing elements on one or more processors and communications across one or more networks [3]. For distributed systems built upon a time-triggered architecture [2] the implementation and analysis of transactions is straightforward. With event-triggered architectures that do not require or support a global time base the implementation of transactions is not as simple if output jitter needs to be controlled. In this short paper we provide an implementation scheme and associated analysis for event-triggered systems. Note even when the basic architecture is event-triggered there will still be the need to support periodic tasks and therefore periodic transactions. We therefore assume the existence of local clocks on each node. Although there is no global time service, any two clocks will be assumed to have bounded drift.

To constraint the flow of work through the system some form of flow control is needed. But again this does not necessarily require global time. Also output jitter, at the end of the transaction, need to be bounded. Issues of composability require bounded behaviour at all nodes of the systems and it can be argues that end-to-end latency can be trade again composability [4] – this is not however discussed in this paper.

II. Standard Analysis with Global Time

Consider, as a means of illustrating the approach, a simple periodic transaction that has two processing parts τ_1 and τ_2 executing on different processors, and a communication link l. Task τ_1 inputs data from the environment, does some initial processing and then passes its 'result' to the link. Task τ_2 takes this results, undertakes further processing and produces an output for the environment. The transaction is simply: $\tau_1 \xrightarrow{l} \tau_2$. Task τ_1 is a pure periodic task that has a defined period T and has a simple structure such as the following:

```
Start_Time := clock
write Start_Time to link
next_release := Start_Time
loop
  input from environment
  undertake processing
  write result to link l
  next_release := next_release + T
  delay_until next_release
end loop
```

Using standard response time analysis it is possible to calculate R_1 the worst-case response time of this task. It will also be possible to estimate the worst-case transmission time for the link, R_l. The calculation of R_l will, of course, depend upon the network protocol. Note the values R_1 and R_l will be known prior to execution and will constitute common knowledge in the system.

In a time-triggered system any reading of a local clock is defined to give a global value. The code for τ_2 could therefore take the following form:

```
read Start_Time from link
next_release := Start_Time + R_1 + R_l
loop
  delay_until next_release
  read from link l
  undertake processing
  write result to the environment
  next_release := next_release + T
end loop
```

The advantage of this structure is that τ_2 is immediately executing with the right period and is guaranteed (within the bounds of the analysis) to have data available on the link when it executes the read operation. The read operation is non-blocking; if when a read is attempted no data is available then there is a fault that can, potentially, be dealt with.

The disadvantage comes from the need to support a global time base and the pessimism that arises from assuming that the communicated data can arrive as late $R_1 + R_l$ after the release of τ_1. Although both of these values are genuinely worst-case, it is not in general true that a transaction can suffer both worst-case situations at the same time. And hence there may be pessimism in the offset value used to separate the executions of τ_1 and τ_2.

When there is no global time base then all that is known at the second processor is the common knowledge of the period of the transactions. Any data arriving on the link may have been transmitted early in the cycle or towards the actual worst-case latency on transmission.

III. Analysis without Global Time

The simplest way to implement a transaction on a distributed platform without global time is to allow each task (apart from the first one) to execute the following simple loop.:

```
loop
  read from link
  undertake processing
  write to next link
end loop
```

Unfortunately this suffers from extreme output jitter. It is also difficult to derive an estimate of the worst-case behaviour as transactions can catch up with one another. Typically some form of *rate control* is applied to stop data been passed on 'too early'. In the following this idea is extended to produce a new protocol called NGT (No Global Time).

Assume that the read operation on the link has the following semantics. It blocks until data is available, and it returns, as well as the data, the earliest time the data was available to be read. So if the data is already available when the call of read is made, the operation succeeds immediately and the time returned is the time that the network interface placed the data in the appropriate buffer for the application code. If the data is not available the call is held until the data is communicated and the time returned is then the current clock value. The code for τ_2 is as follows (but note τ_1 no longer communicates its start time). The first task is assumes to start at time 0; in the following `t` is the time returned from the link (as defined above). *A global set of times is used to illustrate the behaviour of the protocol – but these values are not needed or the protocol to function.*

```
next_release := 0
read from link l returning t
loop
  undertake processing
  write result to the environment
                   (or next link)
  next_release := max(next_release,t) + T
  delay_until next_release
  read from link l returning t
end loop
```

So the arrival of the data sets up the period of the task. Initially, if the data arrives early in the cycle, the read operation will subsequently block and the effective 'period' of the task will be greater than T. But once the maximum latency for the data has been experienced τ_2 will behave as a purely periodic task will period T.

For example, assume time starts at 0, the period of the transaction is 20, the response times of τ_1 are initially 5, 7, 7, 6, 8, 5 and the transmission times of the resulting communication are 12, 13, 14, 14, 12, 12. Task τ_2 would behave as follows. Its first read operation would block until the data arrive at time 17 (5+12). It would then calculate its next release to be at time 37 (max(0,17)+20). Its second read would again block until time 40 (20+7+13). The delay time would now be 60 (max(37,40)+20).

The third data item arrives at time 61 (40+7+14) which will force the following loop to start at time 81. But now the worst case has been experienced. The 4th data message arrives at time 80 (60+8+12), so the read operation at time 81 now does not block and the task loops with a fixed period of 20 at times 101, 121 etc. (ie. an offset of 21). As long as the worst-case latency for the message has already been experienced the loop will now be purely periodic and all read operations will be non-blocking. Its start was however characterised by periods of 37, 23, 21, and 20 before this 20 value became fixed.

In terms of schedulability analysis, assuming a period of 20 is safe. The task will initially have a longer period, but this will not undermine any guarantee delivered by the schedulability test (as long as the test is sustainable[1] – which all standard tests are).

To complete an assessment of the example, note that the time triggered approach would require τ_2 to have an offset of 22. So it starts with a more regular execution but it has a longer latency in its normal phase.

A. Clock drift and infrequent worst-case behaviour

To cater for clock drift an occasional slightly shorter period can be added (i.e. a loop of 19). If this is too much the algorithm will force a 21 value on the subsequent iteration. Note if the clock drift is in the other direction (τ_2's processor clock running quicker) then the the algorithm will automatically extend one period by a small amount – a read operation will block.

If the worst-case message delay occurs very infrequently an application can decide to occasionally bring the period back from its maximum value (for example a one-off 19). This may result is a later period of 21 occurring. Overall jitter is increased but average (and normal) latency is reduced.

B. Fault recognition

One advantage of the time-triggered approach is that a fault (data not arriving) is immediately recognised. Without a global time service this is not as straightforward. There are however some bounds that can be derived. Once the algorithm has stabilised then all reads should be non-blocking so any delay can be interpreted as an error. However due to the reasons outlined above (e.g. clock drift) it would be necessary to give a tolerance on data arriving late.

A safe upper bound on a timeout value can be calculated as follows. With no other knowledge of actual execution and communication times the largest gap between two arrivals of the data is $2T$. Hence a timeout value of T is an upper bound (i.e. delay in the loop is T after the first arrival, and then wait up to T for the data to arrive). However an improvement on this can be obtained if one records how early data does arrive. If W is the maximum time that data has been in the input buffer waiting to be read then the timeout value can be reduced to $T - W$. The code would have the following form:

```
next_release := 0
read from link l returning t
W := 0
loop
  undertake processing
  write result to the environment
  next_release := max(next_release,t) + T
  delay_until next_release
  select
    read from link l returning t
  timeout T-W
    undertaken alternative action
  W := max(W, clock-t)
end loop
```

IV. Simulation Results

To evaluate the validity and performance of this NGT protocol a set of simulation experiments were undertaken. Here we report on one such experiment. The hardware platform was assumed to consist of ten nodes in a pipeline. A single repeating transaction runs through these nodes, with a single task per node. The period of the transaction (and hence the 'period' of each task) was 200ms. Each task had a maximum response time of 180ms, and each communication link had a maximum transmission time of 20ms. Actual response and communication times were obtained from a normal distribution constrained to have these maximum values.

Figure 1 illustrates the end-to-end latency values for the first 5,000,000ms of execution (ie. 25,000 executions of the transaction). Initially the latency is less than 1000ms, but this value grows until an interval of approximately 1800ms is obtained at the end of the simulation. For comparison it should be noted that the time-triggered protocol (TTP) would have a fixed latency of 1800ms plus the final task's execution. So an overall bound of 1980ms.

The NGT protocol produces a behaviour that approaches that of the time-trigger protocol (TTP). It corresponds to the worst-case actual behaviour of each task and communication link. If each worst-case behaviour can reach its theoretical limit then at that point NGT will give the same results as TTP. In the simulations the worst-case can be reached and so as the simulations continue the latency increases.

If the real upper bounds are below the 'worst-case' values used in the static analysis (as will often be the case), NGT will stabilise on a value below the theoretical worst-case. It reflects only the worst-case situations actually experienced by the system.

To reduce even this improved end-to-end latency, one can apply the 'recovery' techniques described earlier. In Figure 2 the period of each task is reduced from 200ms to 199ms every 10 invocations if the data was found to have arrived within this bound on each of these 10 invocations. Jitter is controlled by only making a change occasionally, but as a result the end-to-end latency is rarely above 1600ms.

V. Conclusion

A simple algorithm has been presented that allows a periodic transaction to dynamically set its own parameters on a distributed platform without a global time service. The tasks of the transaction, once each has experiences its maximum latency for its input data, will execute as regular periodic tasks with a fixed period. An advantage of the proposed scheme is that the maximum latency through the transaction is minimised. There is no need to set a potentially pessimistic offset for later components of the transaction. Rather the protocol learn how long each task must wait to get a smooth flow of data through the system.

The only requirement on the hardware platform is that data, as it arrives on an input link, must be time-stamped with the local time of arrival. This is a straightforward operation for a network interface card.

The proposed scheme can deal with clock drift and is able to respond to omission failures (of the input data). It is also able to bring back the worst-case behaviour in a controlled way. So, for example, if the communications media experience a glitch that pushed the end-to-end latency out to an excessive level then the protocol would, over time, bring this value back. But would do so in a way that had a small effect on output jitter.

References

[1] S.K. Baruah and A. Burns. Sustainable schedulability analysis. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 159–168, 2006.

[2] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings 16th IEEE Real-Time Systems Symposium*, 1995.

[3] J.P. Gutierrez, J.G. Garcia, and M. González Harbour. On the schedulability analysis for distributed real-time systems. In *proceedings 9th Euromicro Workshop on Real-Time Systems*, pages 136–143, 1997.

[4] S. Matic and T.A. Henzinger. Trading end-to-end latency for composability. In *RTSS*, pages 99–110. IEEE, 2005.

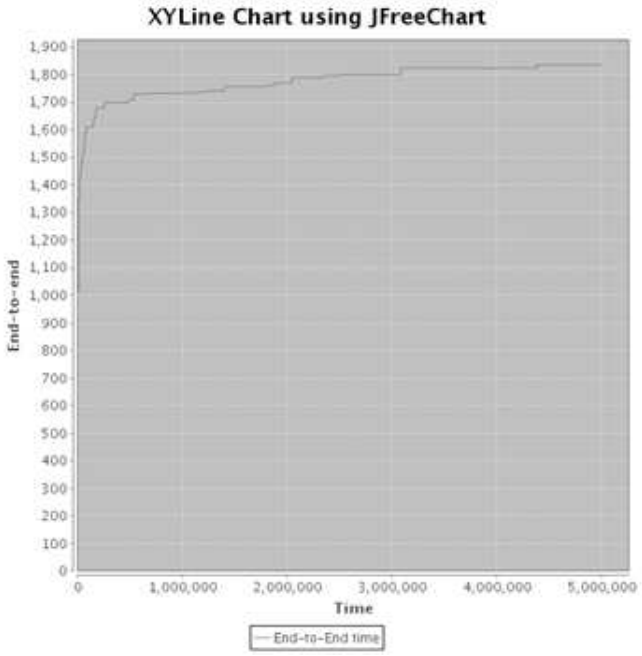

Fig. 1. No Recovery

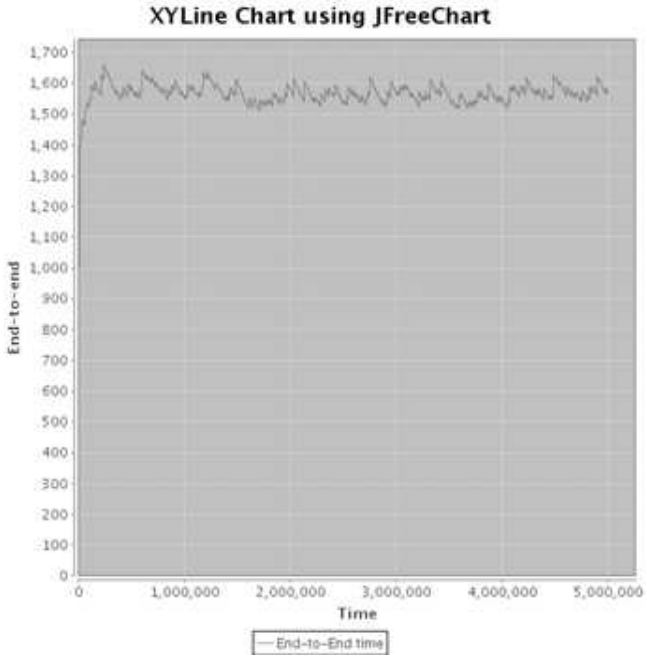

Fig. 2. With Recovery