The Babbling Idiot in Event-triggered Real-time Systems

Ian Broster and Alan Burns
Real-time Systems Research Group
Department of Computer Science
University of York, UK
{ianb, burns}@cs.york.ac.uk

Abstract

We present an approach to detecting 'babbling' faulty nodes on a bus by using a bus guardian to listen to previous messages and deduce a window during which future messages should appear. In general, one cannot detect all erroneous messages, but the messages that are incorrectly classified can be bounded, and therefore can be taken this into account when doing worst case response time analysis.

1. Introduction

It has long been a criticism of event triggered systems that they are unable to detect or tolerate *babbling idiot failures* or commission faults. A commission failure occurs if a process (or node) produces a result (or message, event *etc.*) when none should have been produced. In a real-time system, this extends to an event that occurs too *early*.

Informally, the babbling idiot failure is where a process/node repeatedly suffers commission faults and therefore consumes more resources than it would normally use. For example, consider a set of nodes communicating through a shared bus; if one node suffers a babbling idiot failure and begins to transmit extra messages onto the bus then it may starve the other correct nodes on the bus of network bandwidth.

Time triggered bus protocols such as TTP [5] dictate that a node is only allowed to write to the bus at predefined times. Therefore it is relatively easy to detect if a node becomes a babbling node by watching when it writes to the bus. Furthermore, it is possible to build a *bus guardian* [7] which only connects the node to the bus at these predefined times, hence ensuring that the node can never write to the bus when it should not.

The concept of a bus guardian to isolate a node in the event of failure is not new, it was used by the FTMP architecture in 1978 [4] to safely interface nodes to a multiple-redundant bus. Delta-4 [6] used a similar strategy, restricting node communication via 'network attachment controllers'.

With event triggered communication, is that it is not possible to exactly specify when a message will be sent. This flexibility is an advantage in many systems, but it makes babbling idiot detection (and hence design of a bus guardian) significantly more difficult.

In this paper, some architectures for detecting and preventing a babbling idiot are briefly discussed and an approach for detecting babbling nodes is introduced.

2. Bus Guardians

A bus guardian is a device designed to protect a bus from failure of some component attached to the bus. When designing a bus guardian, one of the most difficult problems is to ensure that there are no common failure modes between the bus guardian and the nodes that is it protecting. Independent hardware with no common components and design diversity can be used to help achieve this. Some potential sources of common failures are: protocol implementation, protocol errors, clocks, CPU/hardware, operating system/software, common data, etc.

However, some common elements cannot easily be removed. For example: it is apparent that the guardian and the node should have independent clocks, in order to prevent a fast/slow clock being a common failure mode. However, if the clocks are independent then we must assume that the guardian and the node may experience clock drift relative to each other. Therefore we must impose some clock synchronisation mechanism upon both the guardian and the node and this introduces either a common component or a dependency between the guardian and the node.

3. Architectures

One strategy (figure 1) to ensure independence is to have the guardian as a completely separate node connected *directly* to the bus. Using only information from the bus, it detects babbling nodes (a scheme for doing this appears later in this paper).

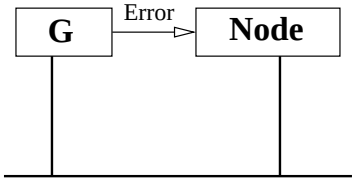

Figure 1. Node and Guardian: Not Coupled

The guardian is able to affect (*e.g.* shutdown) a node using a direct link to the node (this being the only connection between the guardian and the node). For a system where nodes are widely distributed, a guardian will only have a direct connection to nodes in the same physical location, therefore one guardian is required per 'cluster'.

This model requires no special features of the processor node itself, but the guardian is a fairly complex device because it must understand the network protocol. A major disadvantage of this approach is that the guardian is only able to detect a babbling node a *after* it has transmitted an incorrect message onto the bus.

Alternatively the more closely coupled strategy in figure 2 can be used. It is similar to the techniques used by TTP [7] and FTMP [4]. The node always listens to the bus, but transmission is only allowed under the control of the guardian.

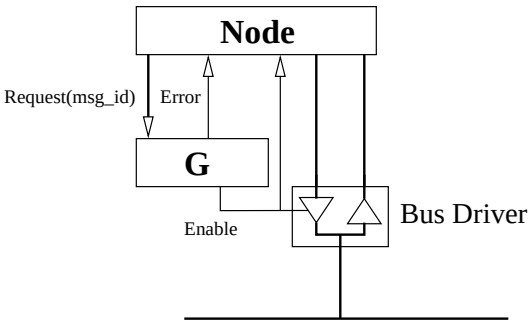

Figure 2. Node and Guardian: Loosely Coupled

Normally the bus drivers are switched 'off'; before attempting to transmit to the bus, the node must inform the guardian that it wishes to transmit. If the guardian agrees with the request then it enables the bus driver. Afterwards, the guardian disables the driver.

This scheme provides significantly more protection because a rogue message is stopped before it can reach the bus. It moves towards fail silence: in order for any message to appear on the bus, both the guardian and the node must concur.

A third architecture (figure 3) uses even closer coupling between the node and guardian. It is similar to the *network attachment controller* approach in Delta-4 [6] and the hardware supported approach for CAN [2] suggested by Tindell [9]. In this architecture, the guardian must be highly reliable: it is the sole interface to the network. It must understand the network protocol (which adds complexity to the guardian) but redundant hardware could be used to provide fail silent dependability.

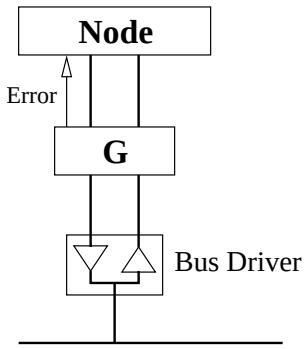

Figure 3. Node and Guardian: Close Coupled

4. Definitions and Model

We define a babbling node as one which incorrectly sends a message before such a message should have been released. It does not matter whether or not the message is functionally correct.

Justification for such a harsh definition is that even one early message is a violation of timing analysis because the early message may impose unanticipated interference on other messages. It is irrelevant whether the babbling node repeatedly transmits too early or not, although this may be taken this into account for any error recovery such as shutting down the node.

The system model in this paper is of a set of independent nodes connected to a single bus. The nodes broadcast periodic messages. The bus traffic may be analysed off-line to provide worst-case response times for each message stream in the absence of errors. The

worst case response time for any message is less than or equal to its period. An example network protocol is CAN [2] where the worst case response time for each message is easily calculated [8].

Babbling nodes are detected by a bus guardian, which is connected using a suitable architecture from section 3 such that it can monitor accesses to the bus of a particular message stream.

We initially assume that there are no faults in the system other than early messages.

5. Approach to Detection

One approach to detecting babbling nodes is to monitor the times that messages from a particular message stream appear on the network, and deduce a *window* during which future messages should appear.

For periodic messages, we can do this by considering the initially unknown *offset* for the stream. If we assume that all previous messages in the stream were timely (neither early nor late) we can deduce some approximate value for the offset of the stream.

For a given stream of messages, let S_i^* be the *actual* release time for the ith message in the stream (*i.e.* when the message is produced by the application). $S_i^* = O^* + iT$, where O^* is the initial offset of message stream. The value of O^* unknown.

For each message in the stream, the time that the message appears on the bus, N_i, is used to provide an estimate S_i for S_i^* using equation (1).

$$S_i = \begin{cases} N_i - Q & \text{if } i = 0 \\ \max(S_{i-1} + T, N_i - Q) & i > 0 \end{cases} \quad (1)$$

where Q is the maximum amount of time that the message could have been queued before appearing on the bus. For CAN in the absence of any faults, $Q = R - C$ (worst case response time - length of frame).

If all previous messages were timely then an approximate test for an early message is

$$N_i < S_{i-1} + T \Rightarrow \text{Message } i \text{ is early} \quad (2)$$

However, this test is not an 'if and only if' test, since the value of S_{i-1} may be up to Q too early. Therefore, in the worst case (no queueing ever occurred), an early message in the window $S_i^* - Q < N_i < S_i^*$ may be incorrectly accepted as on time, when in fact it is early.

5.1. Late Messages

The simple scheme above fails when there are late messages. For a particular real-time system, it may be appropriate to argue that late messages cannot ever occur (consider the quantity on research dedicated to guaranteeing deadlines), for example [3] where late messages are never sent (note that the paper assumes to know O). However, in general, for an event triggered system, late messages are much more likely than a babbling idiot, therefore late messages must be tolerated.

Some late messages can be detected using the variation in queueing to more accurately determine an upper limit to the window.

Let D_i be the relative difference between the "earliest" and the "latest" message in the stream so far (the messages which experienced the shortest and the longest queueing delay). Formally:

$$D_i = \max_{\forall j<i}(N_j - jT) - \min_{\forall j<i}(N_j - jT) \quad (3)$$

which can be efficiently computed for each message using:

$$D_i = S_i + Q - \min(N_i, S_{i-1} + T + Q - D_{i-1}) \quad (4)$$

A message (i) will be classified as early if

$$N_i < S_{i-1} + T - (Q - D_{i-1})$$

and late if

$$N_i > S_{i-1} + T + 2Q - D_{i-1}$$

The window (shown in figure 4) allows the detection of both early and late messages.

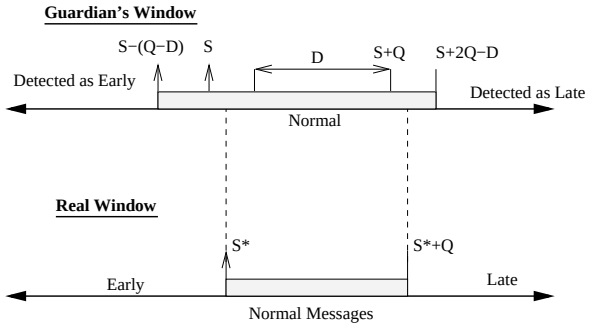

Figure 4. Bus guardian "window" for classifying messages

Notice that he guardian's window in figure 4 is wider than the real window: the accuracy of detection depends on the variation in queueing time for previous messages. The greater the variation, the closer the estimated window will match the 'real' window. Nevertheless, even in the worst case (no variation in queueing time), the inaccuracy of the window can be bounded:

The value of D is bounded by Q therefore in the worst case, an early message at $S_i^* - Q$ will not be detected.

As we can determine the worst case that will not be detected, we can feed this value back into the worst case response time analysis for the bus as a form of jitter (despite the pessimism in doing so). If assumptions are made about the number of correct messages that appear on the bus before a babbling idiot appears then analysis developed by Bernat [1] can be used to argue about the value of D and hence improve the pessimism. A system initialisation self-test (where the consequences of failure are nil) can be used to provide initial timely messages on the bus to fulfil the assumptions.

This means that we can still guarantee the response time of all messages even in the presence of a certain babbling idiot failures.

6. Summary and Future Direction

This ongoing research is investigating a run-time guard that detects certain types of babbling idiot failure and can shutdown offending nodes. Depending on the closeness of the coupling between the bus guardian and the nodes, the guardian may be able to detect different levels of failures.

In general, the guardian cannot detect *all* erroneous messages, but the messages that are incorrectly classified can be *bounded*, and therefore we can take this into account when doing worst case response time analysis.

There are many weak areas in the scheme so far, and these are being addressed in current work. Problem areas include: clock drift, the assumption that the first message is correct, common failure modes and failure of the guardian.

The next planned stage is to analyse *exactly* which errors the scheme outlined in this paper will detect, to further develop the analysis for a specific bus (*e.g.* CAN), and present a response time analysis which can take into account the insufficiencies in any babbling idiot detector.

References

[1] G. Bernat. *Specification and Analysis of Weakly Hard Real-time Systems*. PhD thesis, Universitat de les Illes Balears, January 1998.

[2] Bosch, Postfach 50, D-700 Stuttgart 1. *CAN Specification*, version 2.0 edition, 1991.

[3] I. Broster and A. Burns. Timely use of the CAN protocol in critical hard real-time systems with faults. In *Proceedings of the 13th Euromicro Conference on Real-time Systems*, Delt, The Netherlands, June 2001. IEEE.

[4] A. L. Hopkins, T. B. Smith, and J. H. Lala. FTMP - a highly reliable fault-tolerant multiprocessor for aircraft. *Proceedings of the IEEE*, 66(10):1221–39, October 1978.

[5] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic, 1997.

[6] D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing.*, volume 1 of *Research Reports, ESPRIT 818/2252*. Springer-Verlag, 1991. ISBN 3-540-54985-4.

[7] C. Temple. Avoiding the babbling-idiot failure in a time-triggered communication system. In *Proceedings 28th Annual International Fault Tolerant Computing Symposium, FTCS'98*, 1998.

[8] K. Tindell, A. Burns, and A. J. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3(8):1163–1169, 1995.

[9] K. Tindell and H. Hansson. Babbling idiots, the dual-priority protocol, and smart CAN controllers. In *Proceedings of the 2nd Internation CAN Conference*, pages 7.22–28, 1995.