Dependability Evaluation in Real-Time Network Interfaces

Dawid Trawczynski [1], Janusz Sosnowski [1], Janusz Zalewski [2]

[1] Institute of Computer Science, Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland
d.trawczynski@ii.pw.edu.pl, jss@ii.pw.edu.pl

[2] Computer Science Department, Florida Gulf Coast University
Fort Myers, FL 33965-6565, USA
zalewski@fgcu.edu

Abstract: This article presents a study on efficient dependability evaluation via fault injection using an extension of the True Time network simulator. The experimental results were obtained for checking and comparing fault tolerant aspects of CAN and TTCAN safety critical network interfaces.

1 Introduction

In many real time systems an important issue is high dependability of the communication infrastructure. This problem was analysed in some research studies (e.g. [Za05]), in which various fault tolerance schemes and dependability evaluation techniques have been described. Recently much interest is devoted to fault injection techniques [Aa05, Ar03, SG03]. In this paper we concentrate on a new approach to evaluate safety-critical network interfaces in the context of their performance and fault-tolerance capabilities, based on True Time simulator developed at Lund Institute of Technology [AHC05a, AHC05b] and extended in our Institute.

Our decision to use TrueTime was based on the simulator's ability to co-simulate a real-time operating system kernel and a network interface. Such co-simulation captures the dependencies of those two entities and makes the whole communication model more representative of the physical environment. Another reason why we decided to take advantage of this simulator comes from the fact that it was developed under Matlab/Simulink – a well established and powerful computation tool for digital/analog control, communication and signal processing systems. In cases where computation speed is important Simulink offers the ability to execute resource demanding and computation intensive tasks through MEX interface (C S-function programming). This interface feature is something we wanted to exploit in our newly proposed approach to fault injection in network interfaces.

To perform experiments that evaluate performance and fault tolerance capabilities of network interfaces we have developed algorithms, which generate state space models of considered protocols. These models have been verified and then checked for various fault injection scenarios. For these scenarios we have analyzed various parameters of these modeled systems and compared them with the case of no faults. The presented approach gives better insight into deeper analysis of fault effect propagation. This is useful in an identification of the most critical parts of the protocol and their coverage with effective fault handling mechanisms.

The next two sections of this paper present the methodology and experimental results that evaluate the performance and fault tolerance capabilities of network interfaces. The methodology explains the details of the fault injection model; particularly it specifies points in which protocols have been injected with disturbances. The discussion of fault injection methodology is presented from two perspectives – a high, application level interface and low, state machine level perspective. The experimental section presents network throughput, delay and scheduling results and their interpretation. All results in this section were obtained with the help of our new fault model described in section 2. Finally, in the last section of this paper, we present conclusion of this research.

2 Methodology

The TrueTime simulator models two distinct entities - the operating system kernel and the network interface. These models are therefore described respectively as M_{ki} and M_{ni} where i is the ith model instance in the simulation, n refers to the network model, and k refers to the kernel model. Details on the architecture of the kernel model are given in [HCA03]. Here we concentrate our attention on the network model.

This model can be further decomposed into its main subcomponents namely network nodes, network messages, and network interface faults. The simulator through the addition of new fault objects emulates network interface faults. The authors have implemented these objects for the purposes of disturbing CAN and TTCAN protocol state machines modeled in TrueTime.

At the highest abstract level, faults affect application program interface (API) functions in the True Time kernel. Since these experiments evaluate the network interfaces of TrueTime, fault affected API functions are the ones directly related to network interface operations. When an application task calls one of the network interface functions, it effectively passes control to the respective safety-critical communication protocol. Once this redirection of control is performed, the respective protocol state machine model is responsible for scheduling a databus access and message transmission. It is here, at the state machine level where our faults are injected. Below, we explain the logic of fault injection at this lower level of abstraction for some of the faults implemented in our model.

Fig. 1 presents our fault injection model and associated fault types. The state machines in this figure correspond to state machines of communication protocols (i.e. CAN, TTCAN). The queue is represented by a FIFO message array through which state machine and application messages are exchanged. The tabular list in the model represents implemented fault types. These faults disturb protocol state machines, namely their respective state variables. These variables control state transitions of the state machines as well as the contents of the message queues. Message queues themselves model the logical databus. The disturbance of the message queue is implicit because faults directly affect only the finite state machine variables. An explanation of state variables is presented next to better understand the logic behind the disturbance of CAN and TTCAN finite state machines.

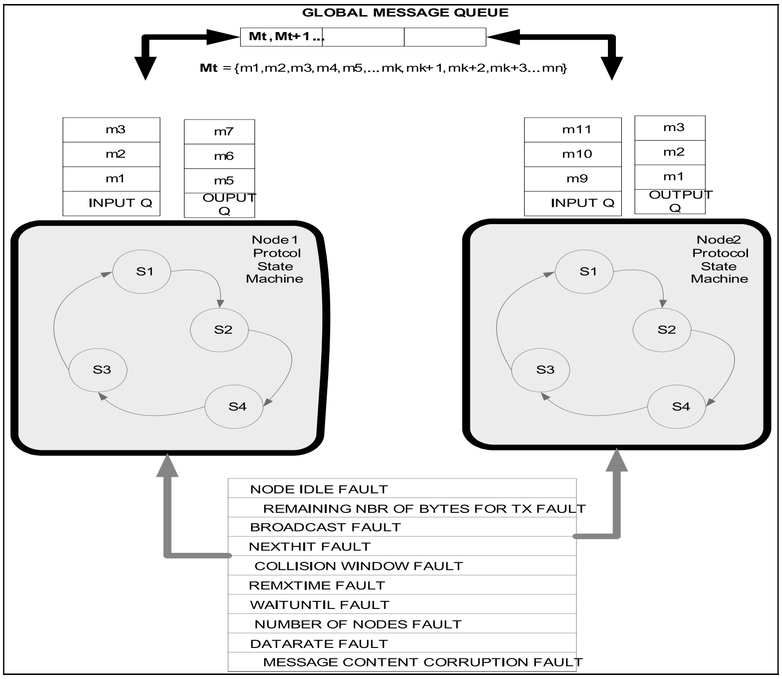

Fig. 1. TrueTime fault model

The first state variable that we decided to disturb is the *Remaining Number of Bytes for Transmission* conditional variable. Setting this variable to any value tells the state machine how many bytes still need to be transmitted for the message under consideration. For the given variable, decreasing its value will allow the state machine to cease the data bus access prematurely (or too late if increasing the value of this variable). This may cause incomplete message transmission (partial data transmission) or inefficient message transmission (allocated frame size is too large then required for efficient transmission).

Another important state variable that we decided to disturb is the *Broadcast* conditional variable. Setting this variable to a value equal to -1 means that protocol is dealing with a broadcasted message. Any other value for this variable ≥ 0 designates the node id of the receiving node. Therefore, changing the value of this variable will convert a broadcast packet to a unicast packet or vise versa. In cases where unicast packets are by fault converted to broadcast packets we expect to increase network interrupt processing delays since nodes now will have to process extra packets regardless of their actual significance. A significant packet is a packet that contributes to either protocol state machine transition or application state transition. Other packets simply occupy network bandwidth with redundant information. Redundant information may be useful (i.e. CRC coding) or useless (multiple copies of the same data packets).

The next state variable in our disturbance model is the *NextHit* conditional variable. This value determines the time when the operating system kernel should be executed. By kernel execution it is meant that the operating system kernel should process its inputs and outputs. Processing of inputs and outputs occurs by processing of tasks. The user in MATLAB development environment specifies these tasks. In case of network protocol state machines, the *NextHit* variable determines the time when network interfaces should be updated. The increasing of *NextHit* due to a fault will increase the time interval between two consecutive updates for network interfaces. Such fault will lead to increased network delays. In some instances these delays may then violate static transmission schedules. Eventually such fault may lead to transmissions errors where a message cannot be delivered to its destination because network interfaces are updated too late for a particular node.

The last major state variable in our fault injection model is the *Remaining Tx Time* conditional variable. This variable determines how much time is needed for the entire transmission of a message. By varying this variable we may simulate value and slightly off specifications faults [Aa02]. For example, if the protocol state machine determined that at 1 Mbps it will take .512 ms to transmit a 64 byte packet then under ideal conditions a complete packet transmission occurs at $t = 0.512$ (where $t_o = 0$). Inserting a fault into *Remaining Tx Time* by changing this value to 0.500 ms decreases the amount of information transmitted in a packet to 500 bits. For example, if the last eight least significant bits of information are shortened from 10110010 to 1011(0000) then the application may interpret an incorrect sensor reading resulting in a slightly off specification error. Faults injected into this state variable are also expected to generate timing errors because in some instances data may not meet its assigned delivery deadline.

The above discussion only describes a sample of fault types implemented in our model. A more comprehensive description was not possible due to a lack of publication space. Having explained the basic fault injection methodology; the authors in section 3 present the simulation approach and results obtained with the help of our new network interface fault injection model.

3. Simulation Results

To illustrate the capability of the implemented model we give some examples of results related to CAN and TTCAN networks. Namely, we present the effects Remaining Tx Time faults have on the global network throughput, node message delay and node message scheduling.

All simulations were performed on a single Intel Pentium 4, 2.8 GHz microprocessor computer under Microsoft Windows XP operating system. The simulator TrueTime was executed in Matlab 7 and Simulink 6 environment. Fault injection was done by fault code insertion and recompilation of the TrueTime source code with Microsoft Visual C++ 6.0 compiler.

Fig. 2 presents network topology used in the simulation model. The network in all simulations contained 8 nodes communicating on a CAN or TTCAN logical data bus. This setup is a good representation of a physical network, and may model a distributed control network in a vehicle active suspension system. In order to enable simulation of network interfaces one must configure a list of specific parameters. When evaluating a simulation experiment the reader must refer to these parameters for correct interpretation of results. An explanation of network parameters can be found in [HCA03].

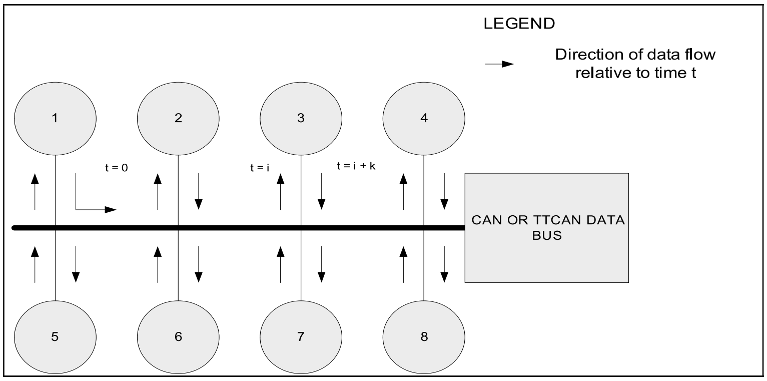

Fig. 2. Simulated network topology

In this network topology, data transmission is realized through network interface functions of the simulator. When a task is ready to send data it calls the respective network API function. When new data arrives at its destination, an interrupt is triggered in the receiving node and the real time kernel invokes an interrupt handler. The handling task then determines how the received data is processed. In our simulation model, we decided to forward the incoming data to the next node in the network. For example, if node i receives data from node i – 1, node i after the triggering of its interrupt handler function, sends the same data to node i + 1 in the network. This type of data flow proceeds in a circular fashion but still respects the databus scheduling policy of its respective protocol. This means that a message generated by a periodic task in node 1 eventually comes back to this node after all nodes in the network process it. This type of traffic flow can model sensor traffic for a shock absorber in an automotive suspension control system. All controller nodes that control the mechanical actuators process this sensor reading and compute control variables with their respective control laws. To more realistically model network traffic, we added three interference tasks into three network nodes. These tasks generate dummy data, once every 10 milliseconds with varying time phase shifts.

Our fault injection model then disturbs the explained simulation network model. In the next three sub-sections we explain the effect our injection model has on the network throughput, delay and scheduling. Those three parameters are compared against the total network traffic or total number of transmissions completed by all nodes at any given simulation time t. The simulation results, due to limited space, preset the effects of only one fault type; namely the remaining transmission time fault effects.

3.1 Network Throughput vs. Number of Transmissions

This section presents experimental results with respect to network throughput. The experiments were performed under one set of tasks (3 interference tasks per simulation run). In each simulation run only one fault had been injected. By injection of only one fault at a time we avoided fault effect correlation and were able to directly observe the effects of the injection. All faults lasted for a time interval equal to simulation interval that in most experiments was set to 1 second. We set the simulation time to 1 second because this time was enough to judge the properties of the network protocol. This means that we were able to obtain enough state transitions and event occurrences within the network to judge its behavior.

Fig. 3 presents the network behavior under a fault, which causes the remaining transmission time to increase by 1 second. We can notice that the total network throughput decreased by 1000 times as compared to a fault-free case of 1 Mbps. TTCAN however performed slightly better then CAN under a presence of this fault. TTCAN achieved 3 times better throughput than CAN. Further research shall explain this phenomenon but currently we think that the reason behind this is that TTCAN schedule strictly defines the transmission slot size.

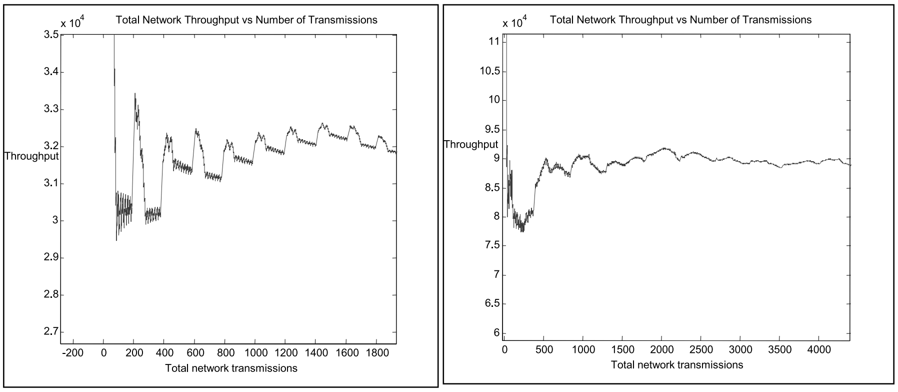

Fig. 3. CAN (a) and TTCAN (b) Throughput under presence of Remaining Tx Time fault

If a value of the *Remaining Tx Time* state variable is greater then the value of slot time then the transmission stops automatically according to schedule. Therefore, TTCAN's inherent ability to quickly stop an "empty" transmission is why the throughput remained higher as compared to CAN. Again, the reason why the throughput decreased in both cases was that the fault transmission time had been extended but no actual data was transmitted. This extra "empty" time decreased the total amount of information transmitted in the network per unit of time resulting in a lower throughput.

3.2 Packet Delay vs. Number of Transmissions

Fig. 4 compares CAN and TTCAN delay results under a presence of *Remaining Tx Time* fault. We can clearly notice here that for TTCAN the delay lost its discrete properties (as compared to fault free operation) and became randomized. The delay also increased at maximum by 4 times as compared to CAN delay. It is very interesting that CAN can take on the properties of TTCAN (discrete delays) under a presence of faults. TTCAN however completed more transmissions and delivered more information in the network by about 66% under a presence of this fault.

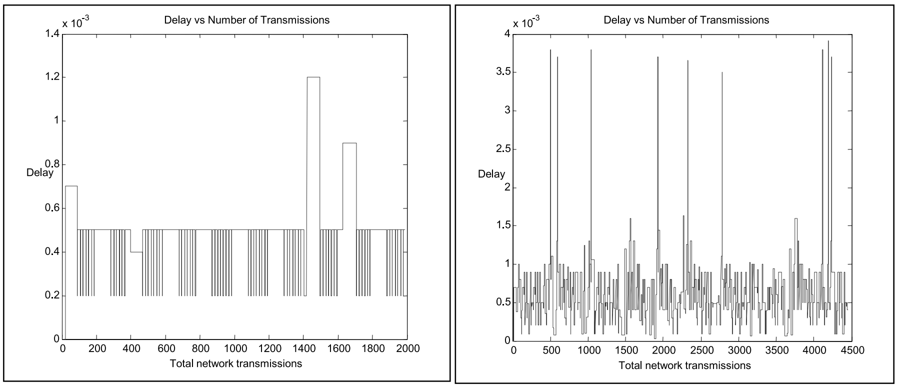

Fig. 4. CAN (a) and TTCAN (b) Delay under presence of Remaining Tx Time fault

3.3 CAN and TTCAN Scheduling Under Presence of Faults

Fig. 5 presents protocol-scheduling results under fault-present conditions. These experiments show the effect a *Remaining Tx Time* fault has on transmission schedules of both protocols. In this figure, a high signal or logical 1, corresponds to time interval in which a node is accessing a databus. A low signal or logical 0, corresponds to a time interval in which a node is not accessing a databus; it's either waiting for access or simply idle. Therefore, this experiment effectively measures bus utilization. In case of CAN, we notice much lower bus utilization within a simulation window of 50 ms. For CAN this utilization is less then 10%, where TTCAN achieves utilization over 40%. Therefore this experiment shows that although in fault-free conditions CAN achieves better bus utilization (under simulated traffic conditions), it performs poorly under a presence of state machine faults. TTCAN due to its scheduling mechanism performs better with respect to efficient bus access in the presence of this type of fault but its bus utilization also decreases considerably. This is an important result because it tells us that utilization can decrease drastically in the presence of fault where in fault free conditions, theoretically the utilization can be higher [AG03].

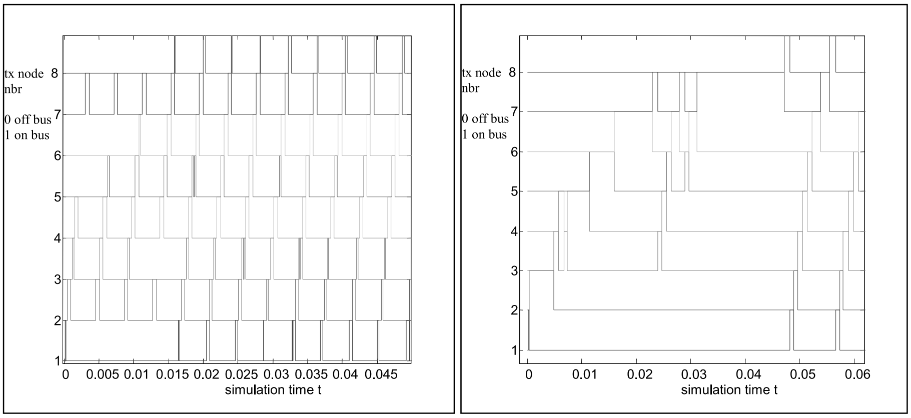

Figure 5. Schedule of (a) TTCAN and (b) CAN under the influence of Remaining TX Time fault

4. Conclusion

From the simulations performed we conclude that neither model, CAN or TTCAN, is completely immune to faults. We found that under certain fault types CAN protocol ensures better performance, while for another set of faults TTCAN protocol behaves better. TTCAN performed better in cases where additional time delays were introduced by fault injection. In those instances the scheduling mechanism of this protocol ensured that "useless" time delays were limited by the specified slot size. CAN however performed better in instances where the time when the kernel should wake up and process its input/outputs, was injected with a faulty value. By better performance we mean that more information had been transferred by CAN than TTCAN due to the event driven nature of the protocol. TTCAN transferred less information because of its strict scheduling policy. According to this policy if a message misses its slot it must wait at minimum one cycle before it can be transmitted over the network. For example, incrementing a value of the *NextHit* variable delays these messages and causes more slot misses. Effectively this causes decreased global network throughput in TTCAN.

The experiments also showed that certain faults could lead to completely unpredictable results. Such was the case when we injected a fault into the *Remaining Tx Time* state variable. This injection resulted in an increased and randomized delay values for TTCAN. The main benefit of TTCAN under protocol state machine fault-free operation is its predictability of delay. Under a presence of this fault, the observed delay randomization, to some extent, falsifies the notion of uniformity and predictability of delay in time-triggered systems. This effect shall be studied in further research.

The main benefit of our fault injection modeling as compared to other dependability evaluation methods is that we were able to simulate fault occurrence in the logical level of the network protocol state machine and at the same time were able to simulate a real time operating system kernel. Co-simulation of the network interface and real time kernel allowed us to create a more realistic and complicated traffic flows where control, interrupt, protocol and interference tasks all compete for bus access. Additionally, with our model we could inject faults into specific areas of the state machines, where other research projects often inject errors randomly. Such approach allowed use to trace the injection effects and map them onto performance issues.

In the future this research will strive to implement models of fault tolerant mechanisms that will prevent state machine faults that manifest themselves as errors and degradations of network performance. Finally, future research will also model other safety-critical network interfaces and compare them with the two discussed in this paper in the context of their fault handling capabilities and data transmission performance.

References

[Aa02] Ademaj A.: Slightly-of-specification failures in the time triggered architecture. Proc. of 7th IEEE Int. workshop on Hogh Level Design and Vakidation and Test, 2002, pp. 7-122.

[Aa05] Ademaj A.; Grillinger P.; Herout P.; Hlavicka J.: Fault Tolerance Evaluation Using Two Software Based Fault Injection Methods. http://vmars.tuwien.ac.at/projects/tta

[AG03] Albert A.; Gerth W.: Evaluation and Comparison of the Real-Time Performance of CAN and TTCAN. Proc. of 9th CAN Conference, Munich, iCC 2003.

[AHC05a] Lund Institute of Technology, Sweden. TrueTime 1.3 Simulink Simulator. **http://www.control.lth.se/**~dan/truetime/

[AHC05b] Anderrson M.; Henriksson D.; Cervin A.: TrueTime 1.3 Manual. Lund Institute of Technology, Sweden, June 2005.

[Ar03] Arlat J.; Crouzet Y.; Karlsson J.; Folkesson P.; Fuchs E.; Leber G.H.: Comparison of physical and software implemented fault injection techniques. IEEE Transactions on Computers, vol. 52, no.9, Sept. 2003, 1115-1133.

[HCA03] Henriksson D.; Cervin A.; Arzen K.: TrueTime: Real-Time Control System Simulation with MATLAB/Simulink. In Proceedings of the Nordic MATLAB Conference, Copenhagen, Denmark, October 2003.

[SG03] Sosnowski J.; Gawkowski P.; Lesiak A.: Software implemented fault inserters. Proc. of IFAC PDS2003 Workshop, Pergamon, 2003, pp.293-298.

[Za05] Zalewski J.; Trawczynski D.; Sosnowski J.; Kornecki A.; Sniezek M.: Safety Issues in Avionics and Automotive Databuses. IFAC World Congress 2005, Prague, Czech Republic.