In Proc. of the 16th IEEE International Conference on Emerging Technologies And Factory Automation (ETFA), Toulouse, France, September 2011.
Revised version.

Resolving State Inconsistency in Distributed Fault-Tolerant Real-Time Dynamic TDMA Architectures

Akramul Azim and Sebastian Fischmeister
Department of Electrical and Computer Engineering
University of Waterloo, Canada
{aazim,sfischme}@uwaterloo.ca

Abstract

State consistency in safety-critical distributed systems is mandatory for synchronizing distributed decisions as found in dynamic time division multiple access (TDMA) schedules in the presence of faults. A TDMA schedule that supports networked systems making decisions at run time is sensitive to transient faults, because stations can make incorrect local decisions at run time and cause state inconsistency and collisions. We refer to this type of TDMA schedule as a dynamic TDMA schedule. Faulty decisions are especially undesirable for safety-critical systems with hard real-time constraints. Hence, real-time communication schedules must have the capability of detecting state inconsistency within a fixed amount of time. In this paper, we show through experimentation that state inconsistency is a real problem, and we propose a solution for resolving state inconsistency in TDMA schedules.

I. Introduction

The application of real-time systems ranges from daily activities (e.g., automotive brake-by-wire systems) to nuclear power plants. In the last thirty years, safety-critical real-time applications such as aircraft, ground transportation, ships, and medical equipments have become essential to people. Most such systems require a reliable network connection to transmit sensitive data. However, these systems are subject to faults and the consequences of faults can be dangerous depending on the criticality of the applications. Thus handling faults in networked real-time systems is an important problem when building state-of-the-art safety-critical systems.

Faults on the communication line can cause a networked system with multiple stations to enter into an *inconsistent* state. For example, stations may disagree on the state of a schedule. Such an inconsistent state can lead to permanent or transient service failures. Transient faults can occur for a variety of reasons such as weak communication links, and hardware or software glitches [4]. In our work, we mainly consider transient faults, because they occur more frequently than permanent faults [13]. TDMA schedules usually employ fault detection schemes such as checksums and CRC [9], however, schedules that facilitate dynamic execution at run time might encounter a special type of *state consistency* problem which is the result of *faulty decisions*. In our work, faulty decision is an incorrect decision at any branching points of the schedule.

State inconsistency can occur in a number of ways in real-time systems. States refer to a certain configuration of the system having well-defined behaviours. In a distributed system, states may differ, for instance, due to communication channel faults, measurement faults, clock synchronization faults, and sampling rate mismatch among devices. An interesting survey on CRC [14] states that it may fail to detect some combinations of errors in the communication of industrial embedded networks. For example, CCITT-16 0x8810 fails to detect 84 of all possible 4-bit errors and 2 430 of all possible 6-bit errors, CAN 0x62CC fails to detect 4 314 of all possible 6-bit errors. Therefore, state inconsistency may occur even in systems with CRC due to CRC's inability of detecting specific communication faults. In this paper, we assume sensor measurements are correct, because we are only concerned with the communication faults.

Different stations may disagree on making decisions in a dynamic schedule. This type of decisions are defined as *faulty decisions*. For example, consider that station 1 makes a correct decision after sending a message to station 2. Assume station 2 never receives this message—or receives a corrupted message—and makes the wrong decision. Three important problems can occur because of such a faulty decision: (1) an incorrect output, (2) desynchronization, and (3) the domino effect. An incorrect output is an execution output that differs from the desired output. Desynchronization occurs when stations disagree after evaluation of conditions locally. The consequence for which a fault or faulty decision in any slot that may affect the subsequent slots is defined as the domino effect. All such problems can occur, for instance, in clocked graphs and tree schedules. The clocked graph [11] is a type of condition-based synchronous schedule. In clocked graphs, stations and the shared communication medium have individual condition-based schedules. Faulty decisions in the clocked graphs may

lead to a different output following the disagreement and affect subsequent behaviour. The tree schedule [8] describes a dynamic TDMA schedule that can make decisions at run time by executing guard expressions during every communication cycle. Unfortunately, this scheduling mechanism is unfit for systems with transient faults, because the scheme assumes that all participants always make the correct decisions at run time.

Dynamic TDMA [8], [20] is a state-of-the-art TDMA scheduling concept in the area of real-time systems, however, a number of challenges remain concerning its efficiency and efficacy. Dynamic TDMA schemes, are capable of making flexible decisions at run time. However, these schemes may suffer from faults more frequently than the conventional static TDMA because of faulty decisions. Transient faults that occur due to making on-the-fly decisions may degrade the system performance. Therefore, examining performance factors such as resolving state inconsistency due to faulty decisions can significantly improve the applicability of dynamic TDMA for different types of real-time applications.

Fault tolerance is required to provide safety guarantees in the presence of faults and several types of tolerance techniques are available. The basic principle of fault-tolerance is to enable computing systems, which can perform in a satisfactory fashion in the presence of faults. A widely used fault tolerance technique is the triple modular redundancy (TMR) technique [17] which has been used for its fault masking and quick response characteristics, consequently increasing the availability of systems. TMR is applicable to various safety-critical systems because it enables them to tolerate a certain number of faults but at the same time keeps fast-responses to continue operations. In this paper, we aim to use the dynamic TDMA scheduling scheme by demonstrating an application with TMR to show the advantages, challenges, and possible solutions for making the dynamic TDMA scheme applicable for real-time systems.

A common mechanism used to detect faults relies on standard fault detection schemes such as CRC and checksum, but state inconsistency may arise even in the presence of these schemes due to limitations in detecting transient faults [9]. In addition, the schemes will incur more overhead than the conventional procedures due to their strong but expensive error detection capabilities. Increased computational complexity for better fault detection will cause longer delays at the source and destination for encoding and decoding. This results in poor goodput [23]. Our work addresses several problems that can occur at the branching points in the dynamic TDMA scheme:

- Scheduler states may differ in different stations when they try to send or receive samples (i.e., data) in a distributed system over an unreliable communication medium because of packet drops or CRC failures.
- Clock synchronization error may lead to state inconsistencies.
- Sampling frequencies of different system architectures may differ due to their difference of internal clocks which may create state inconsistencies.
- Communication latency may cause states of the participating stations to become inconsistent.
- Stations experience jitter which varies among systems and may create state inconsistencies.

In the paper we contribute the following to the state-of-the-art for dynamic TDMA for hard real-time systems:

- We analyze the prevalence of state inconsistency for dynamic TDMA schemes with an TMR-enabled application.
- We propose a clock synchronization method which significantly decreases the occurrence of state inconsistencies.
- We propose a sampling rate drift management scheme to decrease the number of state inconsistencies.
- We propose a fault recovery method for resolving state inconsistencies.

The rest of the paper is structured as follows: We address the related work in Section II. The fault model and the system model are in Section III and Section IV. We describe decreasing the number of state inconsistency and resolving state inconsistency in Section V and Section VI. Experimental results in a real-life domain are presented in Section VII. The paper ends with a conclusion in Section VIII.

II. Related Work

Fischmeister et al. [8] proposed a static TDMA scheduling framework, called tree scheduling framework that can make dynamic decisions at run time. We can call this as *dynamism* in static environments having high-confidence real-time software characteristics such as deterministic behaviour, meeting deadlines, and formal verification. This scheduling scheme has been successfully implemented in a networked medical device case study [3]. *Potop-Butucaru et al.* [20] extend the work of *Girault et al.* [11] for time triggered systems by introducing clock schedules. This work has been merged with the results of [8] to build a time triggered system [19] from specification to the hardware level. However, the system lacks state consistency checking required to operate correctly and timely over unreliable communication channels.

Kopetz et al. [15] propose the C-State based CRC method that checks for state consistency due to faults between senders and receivers using C-State–based CRC along with temporal redundancy to handle station failures. The C-State based CRC [15] method can detect state inconsistencies by using the controller state's information such as TDMA slot information, current mode, global time, and membership information. If the possibility of detecting faults using CRC is high, the performance of the C-State based approach will improve. CRC bits can detect faults, however, it needs a high number of bits and thus has significant communication

overhead. Their application area is static TDMA and can be extended to dynamic TDMA to detect state inconsistencies. The authors provide only simulation results to show the inconsistencies based on the number of arrival of faults. A number of redundant fault tolerant units are running in parallel which come into effect when state inconsistencies occur. Paxos [6] is a consensus protocol for checking state consistency using four types of message transmissions. The messages are: prepare, promise, accept, and accepted. This scheme, like the two phase commit (2PC) scheme or the three phase commit (3PC) scheme, has more communication overhead than the C-State based method due to a significant number of message transmissions required by each station to check state consistency.

Sorel et al. [11] proposed a *fault-tolerant schedule*. This fault-tolerant schedule contains fault handling syntax within the schedule. This type of synchronous software schedule is deterministic and verifiable, and it can be applied to real-life domains such as avionics software [5]. *Sorel et al.* propose backup stations for system recovery. However, frequently-occurring transient faults might lead to a significant amount of switching overhead between primary and backup stations. Therefore, a reliable method for state consistency checking and resolving strategies can improve the level of fault tolerance of this scheduling technique.

III. Fault Model

Due to the higher frequency of transient faults than permanent faults, our focus in this paper is on transient faults. We assume to have a redundant system which can reliably continue the system operation upon encountering permanent faults. Transient faults can occur for a number of reasons and exhibit random behaviour in the system. For example, sending wrong and contradictory information from sensors to the controller may corrupt the system. On the other hand, correct data may be altered or dropped during transmission. In this work, we assume sensors are not faulty and always send correct information to the controller, but we assume that the samples (i.e., data) may get corrupted during transmission.

In this paper, we avoid simulating the fault model or replicating a previously described fault model. We use a real-life communication environment for a heterogeneous architecture to observe the number of communication faults that can occur in the system. The characteristics of faults include communication CRC faults, packet drops, clock synchronization faults, and sampling frequency drifts.

IV. System Model & Terminology

Our model assumes a distributed hard real-time system that executes a set of periodic tasks. Tasks communicate via messages which are transmitted through a shared medium in predefined timed slots. Our model permits dynamic decisions over the conventional TDMA and allows the system to make decisions about which message to sent next. These schedules for the communication bus are generated offline. We assume that input variables for decisions at run time remain static within a communication cycle. All tasks of a communication cycle in our system are known *a priori* so that we can predict future communications. Since we know the schedule beforehand, we can plan the fault-tolerant measures in the schedule following a high-level specification of criticality and importance.

The requirements for fault-tolerant networked systems vary with the application performance requirements and tolerance on state consistencies. In this paper, we assume that the number of state inconsistencies that may lead to failure never exceeds the specified limit of maximum tolerance. To give an idea of how demanding safety critical systems are, we discuss some critical systems and their tolerance of failure rates. A commercial transport fly-by-wire Airbus A-320 can tolerate a maximum probability of 10^{-10} failures per hour [16] of the flight. Military aircraft, unmanned launch vehicles, and autonomous underwater vehicles can tolerate 10^{-6} to 10^{-7} failures per hour. Online transaction processors for banks and stock exchanges demand high availability rather than focusing exclusively on reliability. Another requirement of such real-time applications is the response time which has to be met otherwise the system will fail. In this regard, we design the system in such a way that the actual execution time is less than the allocated time. Therefore, the system never misses the deadline.

V. Overview of Dynamic Schedules

An important fault-tolerant mechanism for real-time systems is TMR. Depending on the type of application, one of two approaches of voting mechanism is commonly used: approximate consensus and exact consensus. In the approximate consensus approach, it is not mandatory to have the values of all the samples equal. Instead, the output of redundant samples needs to be less than or equal to a threshold. The threshold varies among applications and domains. The system may either use a mathematical definition of a threshold, which is dependent on the applications or may employ an empirically derived heuristic. However, the validation of such empirically derived heuristics is tedious. In the exact consensus, all stations in a system must agree bit-by-bit with no fault conditions. Though exact consensus is favorable to approximate consensus, it is difficult to achieve because of the assumed unreliable communication medium. In the exact consensus approach, two components of a system will produce identical results, if they have: (1) identical initial states, (2) identical input, and (3) identical operation. Two components will be in identical states, if the redundant copies of the hardware are initialized to the same state. Identical input means that each hardware copy must be provided with the identical sequence of data from sensors and events. Identical operation means that each channel must execute the same sequence of

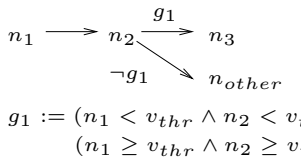


Fig. 1. An example of the dynamic schedule

operations on the same input.

Our approach focuses on checking and resolving state consistency in dynamic schedules. We base our notion of dynamic schedules on the work of *Fischmeister et al.* [7]. Their original work applies to media with a reliable atomic broadcast mechanism. In our work, we apply our state consistency checking mechanism to the system and thus make the original approach applicable for unreliable communication media such as wireless communication.

Stations that use a dynamic TDMA schedule may suffer from faulty decisions which will make the states inconsistent. Stations decide state transitions through satisfying guard expressions. However, stations can make wrong decisions because of undetected faults. Some stations may also make faulty decisions, because received data may be altered before making decisions at run time. This can happen, because of hardware or software glitches such as memory bit flipping [24].

Definition 1 (Dynamic Schedule): A dynamic schedule is a DAG defined by the tuple (V, v_0, V_F, sl, κ, E) where

- V is a set of states,
- $v_0 \in V$ denotes the initial state,
- $V_F \subseteq V$ denotes the set of final states,
- sl labels states V with broadcast communications,
- κ is a set of clocks with $|\kappa| \geq 1$, and
- E is a set of tuples (v_s, g_x, λ, v_d) representing transitions from state v_s to state v_d. The guard g_x is an enabling condition and λ is a set of updates on clock values. The set of transitions must be free of cycles.

Example 1: This example illustrates a dynamic TDMA schedule. We assume a distributed fire alarm using triple modular redundancy. If two of the three temperature sensor stations (n_1 to n_3) report a temperature beyond the set threshold (v_{thr}), the system will sound an alarm. In every cycle, the controller receives the temperature readings and makes the decision by voting on the results. In the dynamic TDMA implementation, if the first two readings are both below or beyond the threshold, then the third station will not report its readings, because the voting is already decisive. Figure 1 shows the resulting dynamic schedule for this example.

Suppose, each station spends t time units to send data to the controller. Therefore, station n_1 sends its data in $[0, t)$, station n_2 sends data in $[t, 2t)$, and station n_3 sends data in $[2t, 3t)$. If n_1 and n_2 send more or less than the threshold temperature value, then n_3 will not transmit data. Instead of sending data, the system can use this slot for best-effort traffic.

If we consider the above example in a noisy environment, then data may be changed during transmission. Therefore, faulty decisions may occur at branching points which will create state inconsistency. If the controller can detect the corrupted message or the state inconsistency, then the system will initiate a fault recovery routine and prevent a service failure by resolving state inconsistency.

Example 2: Continuing from Example 1. Suppose, n_1 reports a value below the threshold and n_2's reported value below the threshold is corrupted by noise during transmission to the controller. If the controller fails to detect the fault, station n_3 will try to transmit data. However, station n_2 assumes that it has sent the correct data, and therefore the system's state will be inconsistent.

VI. Resolving State Inconsistency

Dynamic TDMA is vulnerable to state mismatches at branching points and this necessitates reducing the number of inconsistencies for correct and safe operations. Our approach for preventing state inconsistency consists of two parts: (1) reducing the occurrence of state inconsistency and (2) recovery upon detecting state inconsistency. To reduce the occurrence of state inconsistency in dynamic TDMA, we propose a method of clock synchronization and a method of adjusting the drifts of sampling frequency among devices. To detect and resolve state inconsistency we propose distributed agreement-based schemes that have been adapted from well-known algorithms such as C-State based CRC in hard real-time systems.

A. Resistance from State Inconsistency

1) Clock Synchronization: Salyers et al. [21] stated that clock drift can quickly cause clocks to become unsynchronized. Their clock drift model maintains synchronization among nodes in a distributed system by selecting a reliable host to transmit packets to each of the clients at a slow rate to obtain a training set for linear regression. The linear regression estimates clock drift between systems and synchronizes the clocks. In [2], [10], the authors proposed a clock synchronization scheme to influence clock values using fault-free nodes for time triggered and FlexRay systems. The time triggered architecture (TTP/C) generally uses offset values for correcting clocks whereas FlexRay uses both offset and rate-based mechanisms. Offset correction implies that the current clock state of nodes agrees on a global time base. The rate correction is the process of modifying the frequency of nodes' logical clock.

Clock synchronization methods [18], [21], [22] that ignore synchronization message transmission delay are unsuitable for networked systems. The authors of [25] proposed a simple clock synchronization algorithm that calculates *mean* network delay to measure clock differences for distributed real-time systems. In contrast, we use a variation of this algorithm [25] by considering *worst-case* network delay in our system when

the medium is idle for long periods so that synchronization messages do not interfere with the communication messages. The clock synchronization algorithm that works with our clock drift model will have two types of synchronization messages: a synchronization coordination message *SC* that notifies a node to prepare for synchronization, and a *Sync* message that is sent by a node having the global clock to all other nodes in the network containing the precise sending time of the *Sync* message. The node having the global clock also sends test *Sync* messages to each node before the clock synchronization process starts. This is done to calculate the worst-case network delay for each link from a global clocked node to other nodes. In the clock synchronization process, each node calculates the time difference (dev) by deducting the worst-case network delay and the *Sync* message sending time from the arrival time of the *Sync* message. The steps of the proposed clock synchronization algorithm are in Function 1.

Function 1 ClockSync

Input: $delay^i_{worst}$ for each station P_i, where $0 \leq i \leq num\_stations$
Output: Synchronized clocks G_c
1: **for** each station P_i **do**
2: wait for SC message from the global clocked node
3: record the arrival time $arrival_{time}$ of the $Sync$ message
4: extract the precise sending time $send_{time}$ of $Sync$ message from $Sync$ packet
5: calculate, $dev_i \leftarrow (arrival_{time} - delay^i_{worst} - send_{time})$
6: synchronize clocks using dev_i
7: **end for**

2) Adjusting Sampling Frequency Drifts: Sampling rates of devices, which we refer to as sampling frequency drifts, may vary because of the diverse nature of their internal architecture and crystal sizes of internal clocks. The sampling rate denotes the rate of producing a new sample, i.e., data. Sampling frequency drifts are common across devices of different architectures. Devices based on the same architecture may experience this type of problem less frequently than devices with different architectures because of the homogeneous nature of interactions. Heterogeneous interactions, communications across different architectures or capabilities of systems, often experience sampling frequency drifts.

Adjusting the drifts of the sampling frequencies of a heterogeneous system is challenging, and can be controlled through both hardware and software. In this work, we propose a software-based sampling frequency drift management scheme to reduce the occurrence of state inconsistencies. Our scheme dynamically adjusts the offset for converging the possible diversified decisions into a common distributed decision.

The sampling frequency drift management system assumes to have an independent global sampling rate manager which contacts each of the participating stations to know the architecture and the sampling rates with reference to the global sampling rate manager. Rather than comparing the difference of the sampling rates of the participating stations in the network, each of them synchronizes the sampling rate with the sampling rate of the global sampling rate manager. Each of the stations performs this action only once at the beginning of the dynamic TDMA schedule. If the system permits the addition of new nodes to the network, then the adjustment should be performed after each admission of a new node. To increase reliability by sacrificing resources, this adjustment may be performed after every communication cycle. Therefore, we let the developer decide when to adjust sampling rates for different types of applications. Function 2 describes the operation performed by the stations to synchronize the sampling rates. Lines 2-3 describe the basic operations performed to gather the required synchronization information and Lines 4-5 describe the rate adjustment operations.

Function 2 SamplingSync

Input: global sampling rate r_g , set of sampling rates $R_s = \{r_1, r_2, ..., r_{num\_stations}\}$
Output: set of synchronized sampling rates $R_{syn} = \{r_1, r_2, ..., r_{num\_stations}\}$
1: **for** each station P_i where $0 \leq i \leq num\_stations$ **do**
2: sampling rate manager calculates the sampling rate drift r_{drift} using r_g and r_i
3: receives a rate synchronization message r_{sync} containing r_{drift}
4: **if** $r_{drift} \geq tol$ **then**
5: adjusts the sampling rate r_i using r_{drift}
6: **end if**
7: **end for**

B. Recovery from State Inconsistency

To resolve state inconsistencies for a TMR system with dynamic TDMA, we aim to use a state inconsistency masking method once every communication cycle which may be required to run on the average case. However, to design the system for the worst case, we have to allocate time for using the state inconsistency masking method once every communication cycle. The average case for the TMR system is to receive two samples rather than receiving all three samples in the worst case. Since dynamic TDMA permits making decisions after the second transmission about whether to transmit the third sensor information upon comparing the first two samples, we check at the beginning of the third transmission whether all the other stations will make similar decisions. If all the stations correctly receive the first two samples, then stations will not need to check for state inconsistency. If any of the stations want to receive the third sample, the station will have to check consistency with other stations because of the possibility of having incorrect decisions. If one of the stations disagrees and wants to receive the third sample, then all the stations in the network will need to be synchronized each other by instructing them

either to collect the third sample or cancel the decision of receiving the third sample. The workflow for resolving state consistency is as follows:

Step 1. All stations receive the first two samples which are similar to static TDMA schemes since there are no branches up to this point of the scheme. For example, we have three stations in the network: a, b, and c. Figure 2 shows a dynamic schedule for all the stations. Stations a, b, and c either execute $c_1 \rightarrow 1 \rightarrow 2$ or $c_2 \rightarrow 1 \rightarrow 2$ for a particular control (c_1 or c_2).

Function 3 Detecting State Inconsistency

Input: Transitions E, Guards G_x
Output: State inconsistency CF_i

```
1: E_a ← active transition at run time ⊂ E
2: v_s ← source state at run time: v_s ∈ E_a
3: v_d ← destination state at run time: v_d ∈ E_a
4: g_a ← active guard at run time: (g_a ⊂ G_x) ∧ (g_a ∈ E_a)
5: N_var ← number of variables
6: buf ← data buffer
7: for (g_a ⊢ var_i: var_i ⊂ buf ≠ "not available" where 0 ≤ i ≤ N_var) do
8:    for all entries of var_i ∈ buf do
9:       if ((g_a ⊢ (value of var_i)) ≠ (buf ⊢ (value of var_i))) then
10:         ∃CF_i
11:      end if
12:   end for
13: end for
14: for (g_a ⊢ var_i: var_i ⊂ buf ← "na" where 0 ≤ i ≤ N_var) do
15:   if (C-StateBasedCRC(v_s) ≠ C-StateBasedCRC(v_d)) then
16:      ∃CF_i
17:   else
18:      add var_i into buf
19:   end if
20: end for
```

Step 2. There are several scenarios that can occur at this stage: (I) First all the stations receive correct samples whose values are within the tolerance. The tolerance value is used to decide whether to accept a received sensor value, and the tolerance value depends on the type of plant-controller communication. In every communication cycle, stations check the difference between the previous sample and current sample values against the tolerance for redundant transmissions to decide either to accept or reject samples. If all of the stations accept the first two sensor values which are within the tolerance, the states of all stations will remain consistent. (II) A second scenario is when the state of one station differs from the other stations. If such a situation occurs, then all stations will have to be notified immediately. A distributed agreement algorithm is required for provisioning this facility. We use the technique used in the C-State based approach [15] to detect such inconsistencies. C-State is sent explicitly using a *X frame* [15] or implicitly using an *I frame* [15] but

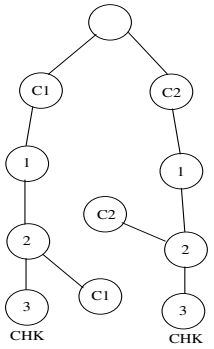


Fig. 2. Dynamic TDMA schedule for TMR

sent always for all frames during transmission. Instead of sending a C-State-based CRC for all transmissions, we may use a history buffer to store recent data and C-State for avoiding sending C-State for redundant transmissions. Having the same or better goodput than the C-State based approach [15], the buffer-based approach can improve the efficiency of transmissions significantly by increasing the CRC length for the first transmission. (III) In a third scenario, stations do not receive any expected data within a fixed amount of time. If there is a timeout, stations will inform other stations about possible future violations. This problem does not occur in a TMR system because of repeating the schedule after the third transmission but it can occur in a complex dynamic TDMA schedule that has data dependencies.

Step 3. Upon detecting state inconsistency, all the stations will have to take appropriate actions to prevent the system from failing. The recovery process can be either pessimistic or optimistic. Pessimistic recovery is to rollback of all the decisions made at the branching points. Optimistic recovery involves a majority voting on the decisions made by stations. The decision with the highest number of agreements by the stations wins and the stations that differ are forced to take that decision. Majority voting can be implemented using the arbitration method in CAN or any distributed agreement algorithms on top of Ethernet.

Function 3 describes the operations performed in detecting state inconsistency. If some of the information which is about to be received is known *a priori*, the stations will check for data mismatches, timeouts in receiving data, or receiving no data. Otherwise, the stations use the C-State information to detect state inconsistencies.

VII. Experimental Analysis

The advantages of using dynamic TDMA [7] have already been demonstrated in different research [7], [8]. Dynamic TDMA performs better in practice than traditional communication schemes such as conventional TDMA and round-robin in terms of scheduling flexibility and efficiency in both hardware and software implementations in real-time time-triggered networks. Apart from dynamic TDMA-based systems, several industrial commercial systems are available such as Powerlink

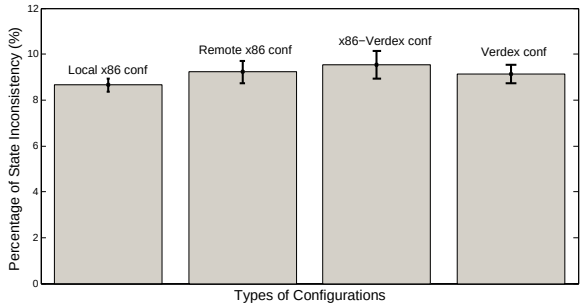


Fig. 3. State inconsistencies for different configurations

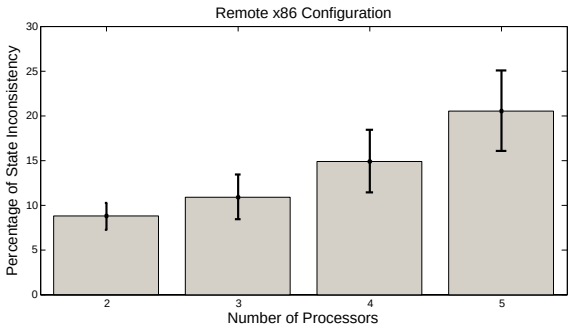


Fig. 4. State inconsistencies for different number of processors

Ethernet, PROFINET, SERCOS III, VARAN, Modbus, TTP/C, FlexRay, and EtherCAT, each of which has a different set of goals. Dynamic TDMA is different from what these schemes provide because the application layer facilitates throughput optimization by providing a flexible communication framework, which can be dynamically changed for different types of application requirements. Examples of applications include both hard and soft real-time systems such as safety critical systems and control systems.

We have used real-life devices to setup the test bed for running experiments. We have used QUARC environment [1] which has been extensively used in the research industry to run and test several algorithms for different safety-critical system prototypes such as unmanned aerial vehicles, unmanned ground vehicles, and mobile robots [12]. We have used the Gumstix Verdex Pro XL6P series board which has the 400 MHz processor Marvell PXA270 with XScale to run experiments and observed state inconsistencies that may occur in the system when using dynamic TDMA. We have also tested with a dual-core x86 and a quad-core x86 machine to run experiments and observed state inconsistencies for four different configurations: local x86, remote x86, x86-verdex, and verdex. Our experiments discover a number of real-life communication issues while using dynamic TDMA or static TDMA.

The application area that is our focus of interest is the drive-by-wire system. The prototype we have tested uses the dynamic TDMA communication scheme. We assume a drive-by-wire system that uses control systems with electronic control systems using electromechanical actuators and human-machine interfaces. The command signals that are generated from the human-machine interface are sent to the controllers via a communication network. In a drive-by-wire system, faults of some safety-critical components may disrupt system functions which are sensitive to the applications. For example, wheel speed data are important in a drive-by-wire system to avoid skidding. The design of a drive-by-wire system should tolerate the loss of some of the data samples, i.e., packet drops which are caused by the safety-critical sensors due to a temporary problem with the sensor itself or with the communication network faults, or a sudden increase in noise. In such cases, the system has to be prepared to tolerate such scenarios for correct and safe operations. The send buffer and receiver buffer size are 1460 bytes and the sampling rate of the host and target system is set to 50 Hz in all experiments.

State Inconsistency Analysis. We have run all the experiments for more than an hour and observed the number of state inconsistencies that may occur when using a simple dynamic TDMA schedule with TMR. Figure 3 shows the percentage of state inconsistencies for four configurations. The percentage of state inconsistency is less than 9.5% for all configurations. We also observe (Figure 4) that state inconsistencies increase as the number of stations increases.

State-Based Scheduling Average Execution Analysis. We have observed by running experiments that the number of the stations that receive the first two samples correctly and do not need receive the third sample is higher than receiving all the three samples. Therefore, on average, stations do not need to receive all three samples. Figure 5 depicts the statistics of the average execution versus worst-case execution in the experiment. We see that the worst case occurs less frequently than the average case. This analysis demonstrates the advantages of using the dynamic TDMA scheme instead of the static TDMA scheme and justify the necessity for resolving state inconsistencies.

Reducing the Inconsistency Frequency. Using the proposed scheme described in Section VI-A, we are able to decrease the number of state inconsistencies or the rate of state inconsistencies occurring over time. The rest of the state inconsistencies are resolved using either our pessimistic or optimistic recovery schemes discussed in Section VI-B. Note that the use of these schemes is more expensive in terms of the time complexity than our efficient scheme given in Section VI-A for reducing the number of state inconsistencies Therefore, the use of all our schemes is application specific. That is, for applications that may tolerate a low threshold of state inconsistencies, our scheme of Section VI-A is suitable for quickly reducing the number of inconsistencies below this threshold. In other, more constrained cases, it may be necessary to resolve all inconsistencies by

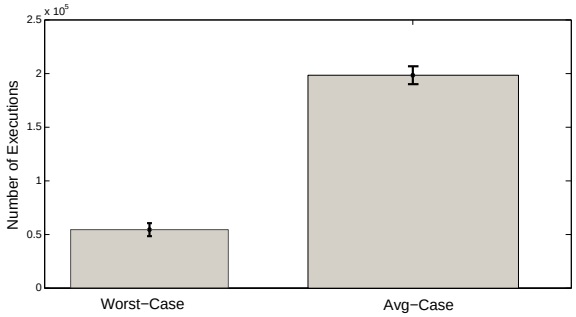


Fig. 5. Worst case versus average case execution analysis for the verdex configuration

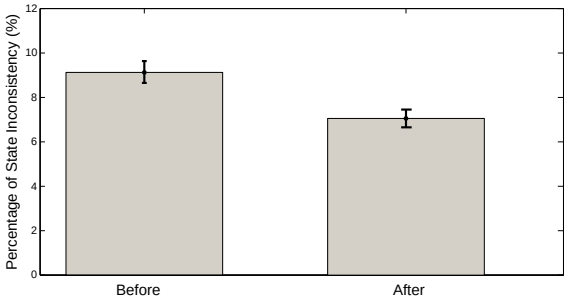


Fig. 6. State inconsistency decrease for sampling rate and clock synchronization for the verdex configuration

using our pessimistic or optimistic recovery schemes; however, note that first using our efficient scheme and then applying the more expensive pessimistic/optimistic schemes still reduces the cost in such cases. Figure 6 shows that the number of state inconsistency decreases using the clock synchronization and the sampling rate synchronization scheme. The response time for resolving a particular state inconsistency is bounded by a fixed amount of time which is provisioned within the schedule offline. In our experiments, the time reserved for each slot is large enough compared to the computation and communication time used in a slot. Therefore, recovery from a state inconsistency does not increase the worst case transmission time. Figure 7 shows the time spent for computation and communication over time in the experiments is less than 0.002 seconds, approximately one tenth of the sampling period of 0.020 seconds.

Jitter Analysis. We have run the experiments on several platforms and Figure 8 shows the jitter. We observe different types of jitter in the system for different configurations and find that the jitter is small compared to the sampling period.

VIII. Conclusion

The domain of real-time applications has been expanding over the last few decades. With the advancement of distributed architectures such as dynamic TDMA-based architectures, real-time systems would benefit from

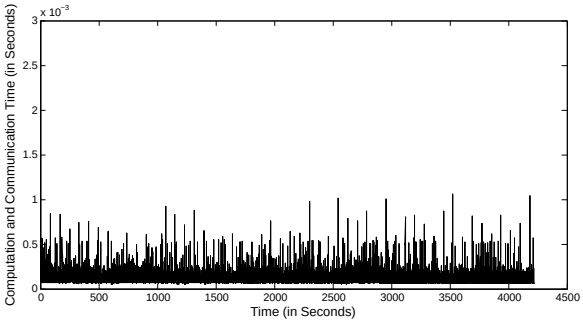


Fig. 7. Computation and communication time for the sampling period of 0.020 seconds

new features such as the on-the-fly decision making capability in dynamic TDMA. However, these new features introduce new challenges such as handling state inconsistency in dynamic TDMA.

In this paper, we identify and discuss the problem of state inconsistency which can occur in dynamic TDMA due to making on-the-fly decisions. We also present appropriate strategies to resolve state inconsistency. We propose a clock synchronization algorithm and a sampling rate drift correction algorithm for reducing the occurrence of state inconsistency. To resolve the remaining state inconsistency cases, we also propose appropriate recovery schemes. Experiments on several computing architectures support the validity of our approach. Future work includes analyzing measurement faults which may affect system performance and protective measures of such faults to guarantee safety standards.

Acknowledements.

We would like to thank Daniel Madill for in-depth discussions on the implementation using QUARC. This research was supported in part by NSERC DG 357121-2008, ORF RE03-045, ORF RE04-036, ORF RE04-039, CFI 20314 and CMC, and ISOP IS09-06-037.

References

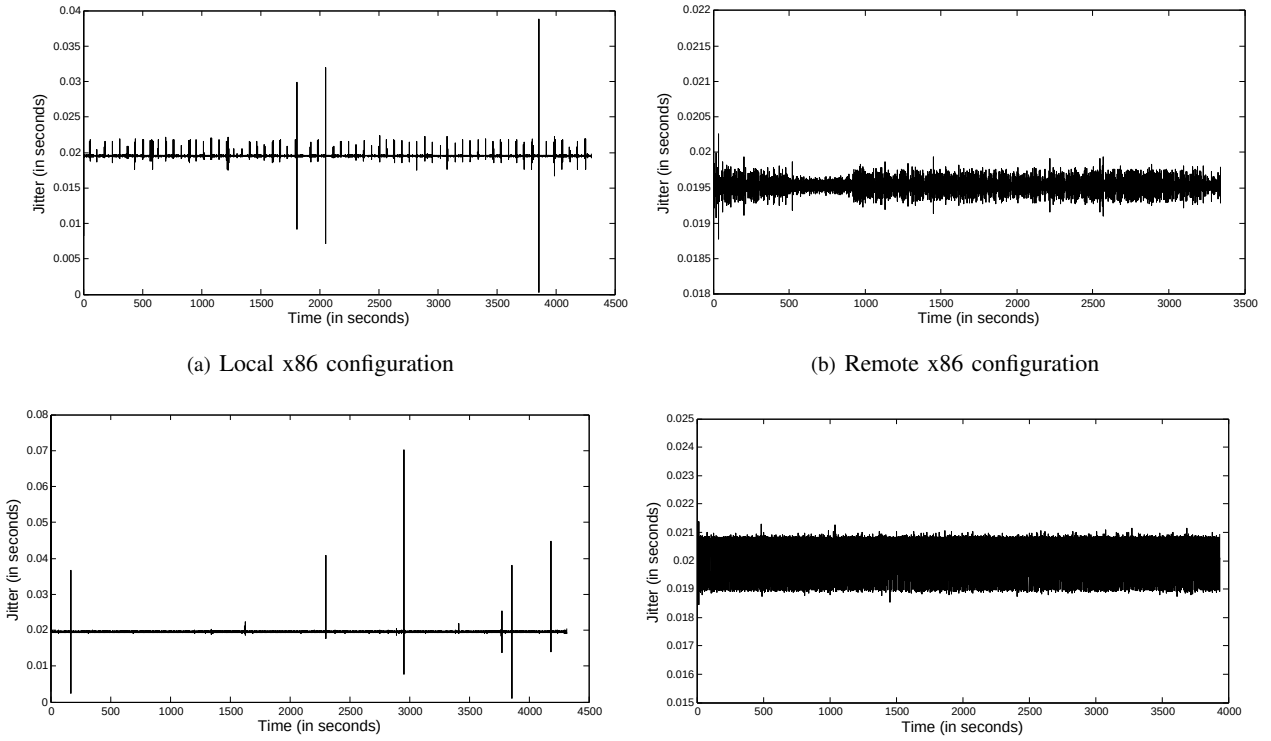
[1] QUARC Control Design Software. www.quanser.com. Visited July. 2011.

[2] E. Armengaud and A. Steininger. Remote Measurement of Local Oscillator Drifts in FlexRay Networks. In *Design, Automation, Test in Europe*, pages 1082–1087, 2009.

[3] D. Arney, S. Fischmeister, J. M. Goldman, I. Lee, and R. Trausmuth. Plug-and-play for medical devices: Experiences from a case study. *Biomedical Instrumentation & Technology*, 43:313–317, 2009.

[4] A. Avizienis, J. Laprie, and B. Randell. Fundamental Concepts of Computer System Dependability. In *IARP/IEEE-RAS Workshop on Robot Dependability: Technological Challenge of Dependable Robots in Human Environments*, pages 1–16, May 2001.

[5] A. Jean-Louis Camus, P. Vincent, O. Graff, and Sebastien Poussard. A Verifiable Architecture for Multi-task, Multi-rate Synchronous Software. In *International Conference on Embedded Real-time Software (ERTS)*, 2008.



(a) Local x86 configuration

(b) Remote x86 configuration

(c) x86-Verdex configuration

(d) Verdex configuration

Fig. 8. Jitter Analysis

[6] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *In Proc. of ACM Symposium on Principles of Distributed Computing*, pages 398–407. ACM Press, 2007.

[7] S. Fischmeister, O. Sokolsky, and I. Lee. A Verifiable Language for Programming Communication Schedules. *IEEE Transactions on Computers*, 56(11):1505–1519, nov 2007.

[8] S. Fischmeister, R. Trausmuth, and I. Lee. Hardware Acceleration for Conditional State-Based Communication Scheduling on Real-Time Ethernet. *IEEE Transactions on Industrial Informatics*, 5:3, 2009.

[9] B. Forouzan and S. Fegan. Data communication and networking. *Fourth Edition*, The McGraw-Hill publishers, 2007.

[10] M. Fugger, E. Armengaud, and A. Steininger. Safely Stimulating the Clock Synchronization Algorithm in Time-Triggered Systems A Combined Formal and Experimental Approach. *IEEE Transactions on Industrial Informatics*, 5(2):132–146, 2009.

[11] A. Girault, C. Lavarenne, M. Sighireanu, and Y. Sorel. Fault-tolerant static scheduling for real-time distributed embedded systems. In *International Conference on Distributed Computing Systems*, pages 695–698, apr. 2001.

[12] R. Huq, H. Lacheray, C. Fulford, D. Wight, and J. Apkarian. QBOT: An Educational Mobile Robot Controlled in MATLAB Simulink Environment. In *Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 350 –353, May 2009.

[13] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Design Optimization of Time- and Cost-Constrained Fault-Tolerant Embedded Systems. In *Conference on Design, Automation and Test in Europe*, pages 864–869, 2005.

[14] P. Koopman and T. Chakravarty. Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks. In *International Conference on Dependable Systems and Networks (DSN)*, pages 145–155, 2004.

[15] H. Kopetz, G. Bauer, and S. Poledna. Tolerating Arbitrary Node Failures in the Time-Triggered Architecture. *SAE 2001 World Congress, March 2001, Detroit, MI, USA*, March 2001.

[16] J.H. Lala and R.E. Harper. Architectural Principles for Safety-Critical Real-Time Applications. *Proceedings of the IEEE*, 82(1):25 –40, jan 1994.

[17] R. E. Lyons and W. Vanderkulk. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM J. Res. Dev.*, 6:200–209, April 1962.

[18] M. Mock, R. Frings, E. Nett, and S. Trikaliotis. Continuous Clock Synchronization in Wireless Real-Time Applications. *19th IEEE Symposium on Reliable Distributed Systems*, pages 125–132, 2000.

[19] D. Potop-Butucaru, A. Azim, and S. Fischmeister. Semantics-Preserving Implementation of Synchronous Specifications Over Dynamic TDMA Distributed Architectures. In *Proc. of the International Conference on Embedded Software (EMSOFT)*, Scottsdale, Arizona, USA, October 2010.

[20] D. Potop-Butucaru, R. de Simone, Y. Sorel, and J. Talpin. Clock-driven Distributed Real-time Implementation of Endochronous Synchronous Programs. In *Proceedings of the 7th ACM International Conference on Embedded Software (EMSOFT)*, pages 147–156, New York, NY, USA, 2009. ACM.

[21] D. Salyers, A. Striegel, and C. Poellabauer. A Light Weight Method for Maintaining Clock Synchronization for Networked Systems. In *International Conference on Computer Communications and Networks*, pages 522–526, 2008.

[22] K. Sun, P. Ning, and C. Wang. Secure and Resilient Clock Synchronization in Wireless Sensor Networks. *IEEE Journal on Selected Areas in Communications*, 24(2):395–408, 2006.

[23] A. Vasan and A. Shankar. An empirical characterization of instantaneous throughput in 802.11b WLANs. Technical Report, 2002.

[24] R. Velazco, A. Corominas, and P. Ferreyra. Injecting Bit Flip Faults by Means of a Purely Software Approach: A Case Studied. In *Proceedings of the 17th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT)*, pages 108–116, Washington, DC, USA, 2002. IEEE Computer Society.

[25] M. Zhang, J. Shi S. Shen, and T. Zhang. Simple Clock Synchronization for Distributed Real-Time Systems. *IEEE International Conference on Industrial Technology*, pages 1–5, 2008.