Azim *EURASIP Journal on Embedded Systems* (2017) 2017:34
DOI 10.1186/s13639-017-0082-x

EURASIP Journal on
Embedded Systems

RESEARCH Open Access

Generation of fault-tolerant state-based communication schedules for real-time systems

Akramul Azim

Abstract

State-based schedules use a time division multiple access (TDMA) mechanism that supports executing conditional semantics and making on-the-fly decisions at runtime in each communication cycle. Until now, state-based schedules are unable to tolerate transient faults due to the assumption that stations make the on-the-fly decision on which message to execute next. Stations may make a faulty decision at run time in an unreliable communication environment such as wireless medium due to the presence of transient faults. This faulty decision causes state inconsistency among the stations in the system.

In this work, we extend state-based schedules to tolerate faulty decisions in environments where transient faults can occur at the communication layer. Our proposed approach generates fault-tolerant state-based schedules using an integer linear programming optimization model after reducing the possibility of state inconsistency through using a clock and a sampling rate synchronization mechanism. The optimization model maximizes the use of time slots to place checkpoints for fault tolerance and resolving state inconsistency.

1 Introduction

The popularity of wireless networks is increasing every day because of their easy and affordable deployment characteristics. Due to the management issues, wired networks such as Ethernet-based networks often impede rapid deployment. However, wired networks in general are more reliable than wireless networks due to the transmission characteristics such as low channel interference and high bandwidth.

Several communication barriers such as channel interference and environmental challenges are the reasons for occurring faults in wireless networks. Moreover, faults can occur due to hardware and software glitches. For example, device memory can flip bits and routers may drop packets. In our context, a fault is a defect or flaw that occurs in a hardware or software component of the system. An error is a consequence of such a fault. As described in [1], a fault remains inactive until it produces an error. A failure occurs when an error results in the cancelation of the requested service of a system. The failures can have catastrophic affects in the system. For example, Therac-25 had catastrophic consequences due to software failures.

Fault recovery can be effectively carried out by either restoring a previously correct state [2] or using redundancy [3]. Faults like floating point arithmetic may occur but not be apparent at the same time [4]. Fault-tolerant systems attempt to detect and correct errors before they become effective.

Safety-critical real-time applications must function correctly and meet their timing constraints even in the presence of faults. Such faults can be permanent such as broken communication links and damaged stations, or transient such as temporary faults caused by interference. Transient faults occur temporarily in the system but occur more frequently (100 times more than permanent faults) than permanent faults [5, 6]. This paper discusses transient fault tolerance, leaving the extension to tolerate permanent faults in future work.

State-based schedules [7, 8] are effective in saving system resources for hard real-time systems because of scheduling messages for the average-case rather than the worst-case, and several case studies across different application areas already demonstrate the advantages of this

Correspondence: akramul.azim@uoit.ca
Department of Electrical, Computer and Software Engineering, University of Ontario Institute of Technology (UOIT), Oshawa, Canada

approach including control theory [9], hybrid systems [10], video-on-demand, hierarchical scheduling frameworks [11], and bursty demand models [12, 13]. It is possible to avoid executing the worst-case due to the ability of making on-the-fly decisions at run time. On the other hand, messages are always scheduled for the worst-case in the traditional static scheme that is TDMA-based and does not allow to make a decision at run time.

In safety-critical systems, the triple modular redundancy (TMR) technique [14] is widely used for fault tolerance. Although TMR is not a robust mechanism for fault tolerance, the scheme can mask faults quickly and runs efficiently. A state-based schedule can become fault-tolerant by the use of TMR, but it might not remain effective in unreliable environments due to the possibility of occurring faulty decisions. A faulty decision is an incorrect or inconsistent decision taken by any of the participating stations in the network. This results in state inconsistency and a potential deadline loss, which is unacceptable in real-time systems.

To ensure making the correct decision in a timely manner for safety-critical applications, architectures using a state-based schedule require state inconsistency detection and resolution. A system can use state-based schedules instead of static TDMA for lower bandwidth usage, but the possibility of occurring state inconsistency may become challenging for using such schedules in practice. Therefore, this paper discusses the state inconsistency problem and its resolving strategies to ensure correct operation while using the state-based schedules in an unreliable communication environment.

State inconsistency due to the occurrence of faulty decisions requires reduction, detection, and resolution. It is possible to reduce the number of faulty decisions when using clock and sampling rate synchronization. Existing approaches such as C-State-based approach or history of recent transmissions can be used to detect state inconsistency. To resolve state inconsistency, systems can either use approaches like majority voting for faster resolution or generated fault-tolerant state-based schedules with checkpoints for guaranteed recovery.

To demonstrate the advantage and challenges of using state-based schedules in the presence of communication and measurement faults, we use an unreliable wireless communication medium that connects a drive-by-wire automotive architecture. The faults occur for different reasons such as communication CRC failures, packet drops, clock synchronization issues, and sampling frequency drifts. We also observe the effect of measurement faults while using a state-based schedule as a communication mechanism in a position control system. These communication and measurement faults cause state inconsistency which is reduced using a clock and sampling rate frequency drift algorithm and a generated fault-tolerant schedule afterwards.

This work mainly contributes the following in using state-based schedules reliably for safety-critical applications:

- With an industrial testbed, we have shown how a state inconsistency can occur when using a state-based schedule in an unreliable environment
- We have shown that TMR, a commonly used fault-tolerance technique, can be efficiently implemented using state-based schedules, however, may still suffer from state inconsistency due to making incorrect decisions at run time
- We generate state-based schedules with checkpoints to recover from state inconsistency. Prior generating schedules, we also discuss how to reduce the number of state inconsistency and detect them if occurred
- We demonstrate the existence and recovery of state inconsistency through experimental analysis using a drive-by-wire (DVW) application and a position control system (PCS) application running state-based schedules

In this paper, we show how to tolerate state inconsistency in state-based schedules. In Sections 2 and 3, we present the fault and system model. Sections 4 and 5 discuss the state inconsistency problem that can occur when using state-based schedules. Sections 6, 7, and 9 present the strategies to reduce the occurrence of state inconsistencies and resolve the remaining of them. Section 10 explains the experimental design, setup environment, and results. We discuss the related work in Section 12. Finally, Section 13 concludes the paper.

2 System model and terminology

We assume a distributed real-time communication framework that consists of periodic messages. Messages execute on stations that are connected wirelessly. The number of stations is fixed and known throughout the entire communication phase. Messages communicate with other messages through messages on channels. All channels are mapped onto one shared unreliable communication medium. All message transmissions are atomic broadcast, and therefore, potentially all stations receive messages reliably. Stations have mutually exclusive access to channels at any point in time defined a priori. The state-based schedules are based on TDMA for deterministic access to the medium.

In state-based scheduling, participating stations in the communication use dedicated slots to send their messages. We assume that the communication link has sufficient bandwidth to carry out messages. A slot has a known start and end time. A communication round or cycle is the time duration after which the state-based schedule is repeated.

We use state-based scheduling to tailor communication behavior to application demands. In contrast to traditional static TDMA schedules, the state-based schedules permit making decisions during the communication cycle. We use the abstraction of Network Code [15] to implement these state-based schedules.

We assume that the system is well designed and static. This refers that the communication behavior is known a priori including the message sizes, variables, bus bandwidth, and the timing requirements. We can thus assume that it is possible to generate an optimal state-based schedule, the programs, and all other data structures offline.

3 Fault model

The fault model comprises transient faults that are unable to cause permanent failure of a system. Transient faults can occur for a number of reasons such as sending wrong and contradictory information, or receiving altered data during transmission. Transmitted packets may also get dropped for a variety of reasons such as link failure and channel interference.

We assume that faults are independent, which avoids the domino effect that can occur when a small change in a component causes a subsequent change in the connected component. We also assume that the system can reliably continue the system operation upon encountering permanent faults because of running backup stations in parallel.

Due to a number of reasons, stations may make wrong decisions at run time. One reason is that faults may be undetected because of CRC failures or limitations on using CRC [16]. Some faults do not appear at the same time they occur. Moreover, faults may occur because hardware or software glitches such as memory bit flipping [17] that may alter the data before making decisions at run time.

The fault model consists of the following faults: (1) clock and sampling rate drifts, (2) corrupt messages, (3) corrupt memories, (4) measurement faults due to truncation or floating-point arithmetic, and (5) data races between computation and communication. Figure 1 shows the occurrence of these faults in different entities in a network architecture. Faults due to clock and sampling rate drifts are handled using synchronization mechanisms. The C-State mechanism [18] can detect corrupt messages, memories, or measurement faults, which can also be detected using history information of previous successfully sent messages. Data races between computation and communication will not occur, because state-based schedules allow only time-triggered communication. Therefore, in this work, we have considered all the above described faults except the data races.

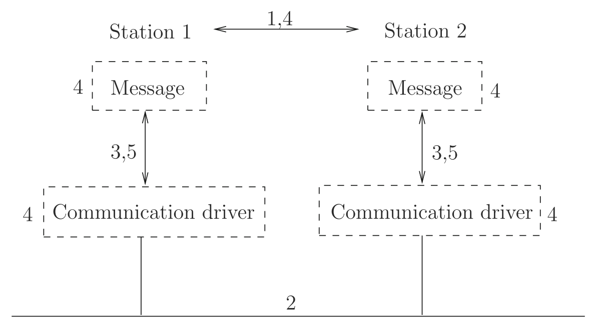

Fig. 1 Fault model

4 Overview of state-based schedules

Our approach towards building a fault-tolerant state-based schedule is based on the notion of state-based schedules as proposed in [15]. A state-based schedule can be represented as a graph where vertices denote the states, and arcs refer the transitions. A transition from one vertex to another vertex represents the condition associated with them evaluated to true. A state-based schedule is TDMA-based and facilitates making on-the-fly decisions at run time.

Definition 1 (State-based schedule) *A state-based schedule is a graph 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 $\langle v_s, g_x, \lambda, v_d \rangle$ 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 state-based schedule for an industrial shutdown system using TMR. We assume multiple controllers periodically receive the sensor samples. In each communication cycle, the controllers receive the temperature readings from either two sensors (best and average-case) or three sensors (worst-case) and make the decision by voting on the results. If two of the three temperature sensors report a temperature beyond a set threshold, the system will shutdown the system, because the voting will already be decisive. Otherwise, the controllers will receive the third sensor value and will include it in the voting process.*

Suppose each sensor spends T time units to send data to the controller. Therefore, the sensor n_1 sends a temperature sample in $[0, T)$, the sensor n_2 in $[T, 2T)$, and the sensor n_3 in $[2T, 3T)$. If n_1 and n_2 send more or less than the threshold temperature value, then the sensor n_3 will not transmit its sample. Instead of sending the temperature sample in the third time slot, the stations can transmit best-effort traffic. The controllers may also use this slot for running a background message displaying state information. Figure 2 *shows the resulting tree schedule for this example.*

5 The state inconsistency problem

Definition 2 (State Inconsistency) *A state inconsistency occurs if at least two stations at any state $v_i \in V$ execute different transitions to reach from state v_s to v_d, although each station executes the same state-based schedule.*

Faulty decisions occur at points that lead to stations executing different branches in state-based schedules. For example, in the state-based schedule of TMR, a faulty decision may occur after the transmission of the first two samples to decide whether to transmit the third sample in the next time slot. Therefore, stations must have a consensus on making the same decision after the first two sample transmissions. If the values of the first two samples are within a tolerance and correctly received by all stations, then the states will remain consistent. If undetected faults occur in any of the previous two transmissions, then state inconsistency will occur.

Example 2 (Continuing from Example 1) *An occurrence of a faulty decision in Example* 1 *is to make a wrong and different decision by any of the stations. This may occur in an unreliable environment (e.g., a wireless medium) on sending the third sensor sample to the controller when sensors making different decisions are likely. This can lead to one of the sensors or controllers to have a different view of the system and result in making a faulty decision.*

Suppose n_1 reports a value above the threshold, and n_2's reported value above the threshold is corrupted by noise during the transmission to the controllers c_1, c_2, and c_3. If c_2 fails to detect the data corruption or has a timeout in receiving the sample, then the controller c_2 may make not wait to receive the sample in the next time slot from n_3. As a result, state inconsistency will occur, because the controller c_2 will have a different view of the system than controller c_1 or c_3.

$$\begin{tikzcd}
n_1 \arrow[r] & n_2 \arrow[r, "g_1"] \arrow[dr, "\neg g_1"'] & n_3 \\
 & & n_{other}
\end{tikzcd}$$

$$g_1 := (n_1 < v_{thr} \wedge n_2 < v_{thr}) \vee (n_1 \geq v_{thr} \wedge n_2 \geq v_{thr})$$

Fig. 2 An example of a state-based schedule

6 Reducing the occurrence of state inconsistency

We use a two-step procedure to reduce the occurrence of state inconsistency in state-based schedules. First, we use a method of clock synchronization to adjust the clock drifts. Thereafter, we use a method to adjust the sampling frequency drifts of the stations.

Clock synchronization [19] mechanisms are necessary to ensure the timeliness occurrence of timing events in using state-based schedules. Due to clock drifts, stations may be de-synchronized in making decisions, and therefore, faulty decisions will occur. We use a clock synchronization algorithm [20] that considers the *worst-case* network delay instead of the average network delay. The synchronization takes place when the medium is idle for long periods so that synchronization messages do not interfere with the communication messages. If stations make intensive use of the medium and it almost never gets idle, then the clock synchronization happens locally based on previous updates.

The mismatch of sampling rates among stations can be controlled using hardware or software. However, it is difficult to eliminate the problem completely because of the temperature effects and the level of accuracy of clock skew correction. We use a software-based sampling frequency drift management scheme [20] to adjust the sampling frequency drifts with reference to an independent global sampling rate manager.

7 Recovery from state inconsistency

Safety-critical systems not only use mechanisms to reduce the occurrence of state inconsistency, but also require methods to recover from state inconsistency for safety. Safety-critical applications such as pacemakers and nuclear shutdown systems emphasize safety over performance. Therefore, state inconsistency is a safety issue that must be resolved completely when using state-based schedules in *safety-stringent* applications.

Despite the occurrence of state inconsistency, a recovery scheme keeps the system safe from failure. A detection scheme first identifies the occurrence of a state inconsistency. A recovery scheme ensures the correct functional operation in the system hereafter. A recovery scheme resolves the state inconsistency upon detecting it by running recover algorithms.

To recover from state inconsistency, stations first need to detect faulty decisions. All stations must check for faulty decisions that follow different branches in state-based schedules. This refers to a distributed consensus problem [21]. To detect such inconsistencies, a system can use the C-State-based approach [18]. In addition to using the C-State-based approach, this paper also discusses using history of recent transmissions to detect such inconsistencies.

After detecting state inconsistency, we use a recovery scheme that is either time-efficient such as the majority voting scheme or reliable such as the rollback. The majority voting scheme is less reliable than the rollback approach to recover from state inconsistency. However, rolling back to an earlier point in time may cause a system failure or destabilize a control system because of having any limitations on re-execution.

7.1 Detecting state inconsistency

To detect state inconsistency when using state-based schedules, the C-State-based approach [18] can be used. The C-State of the sender includes with the message contents to calculate CRC at the sender, and the stations use the C-State of the receiver to check CRC over the message contents at the receiver. A C-State contains state's information such as TDMA slot information, information on current mode, information on global time, and membership information. The C-State of the sender and receiver will differ if faults or faulty decisions occur.

The C-State-based approach requires to transmit the C-State information even though sending it may not necessary. For example, C-State information in redundant transmissions will become unnecessary to send or receive if they are the same. To avoid unnecessary transmission of C-State information, we propose an approach that uses the history of recent transmissions.

To use history of recent transmissions in detecting state inconsistency, it is required to store a number of variables of interest in a buffer. The system stores variables in a tabular format that are common in subsequent transmissions (see Table 1). Successful communications are recorded to use them in validating redundant transmissions. For example, in a TMR-based system, the system can store the C-State information after the first sample transmission and can avoid transmitting the C-State information for subsequent transmissions.

At design time, the developer specifies the buffer table that uses a tiny storage in the memory. In the buffer table, the columns represent the list of variables and the rows represent the slots. The first row stores the start values. Every subsequent row stores the values of variables for the slots over time. The number of rows is bounded by the number of slots in each communication cycle. The developer can specify initial values; otherwise, the variable is initialized with ϕ.

Example 3 *Assume that we have three stations s_1, s_2, and s_3 maintain a buffer history. Consider a state-based schedule that has three slots, and the schedule has five variables that are common in redundant transmissions. All variables are initially set to ϕ and updated over time. In the table, the value T_x denotes the value of a transmitted variable and R_x denotes the value of a received variable. For example, in slot 1, the station s_1 transmits message that contains the variables a and b which are common in subsequent transmissions. In slot 2, the station s_1 transmits message that contains the variables a, b, c, d, and e. In slot 3, the station s_1 may remain idle or transmit the message that contains the variable e.*

Upon receiving messages, the system will update the buffer table. At the beginning of a communication cycle, all variables reset to their initial value or ϕ. Upon receiving a message, if the variable's current history information contains ϕ in the buffer table, then the system will use the C-State-based CRC method to validate the transmitted message before updating the associated variable entry in the buffer table. On the other hand, if the variable in the buffer table has already been initialized or updated, then the system will use the variable's history to check state inconsistency. The history-based approach performs better than the C-State-based approach [22].

7.2 Resolving state inconsistency

7.2.1 Using majority voting for recovery

Majority voting [23] is a mechanism of reaching a consensus in distributed systems. The decision with the highest number of agreements among the stations is chosen and the stations which differ are forced to make that decision. In case of a tie, a random but the same decision will be taken. Majority voting can be implemented using the arbitration method in CAN [24] or any distributed agreement algorithms [25].

Example 4 *Continuing from Example 2, all controllers in turn check the sample values of the other controllers. For example, the controller c_2 checks the sample values of the other controllers (c_1 and c_3). Since controllers c_1 and c_3 decide to receive the third sample, controller c_2 will ask for the third sample, because two from three controllers want to receive the redundant sample.*

7.2.2 Using checkpoints for recovery

This buffer table can be used not only for detecting state inconsistency but also for recovery. A row in the buffer table can be treated as a checkpoint if the row contains all variables' information. Upon detecting a state inconsistency, stations can rollback to the closest checkpoint.

A state inconsistency handler contains the method for detecting state inconsistency and the method of rolling back to the immediate checkpoint. A number of state inconsistency handlers are associated to each checkpoint. The more the number of inconsistency handlers in between checkpoints, the more the system can tolerate state inconsistencies. However, increasing the number inconsistency handlers also increases the overhead. Inconsistency handlers can be placed based on weights, and developers can choose weights (w_i) based on the importance of the contents in the slots. If there are some trailing zero weighted slots, then these slots will not contain inconsistency handlers.

A system can tolerate a maximum number of occurrences of state inconsistencies at any slot, which we define as *replay bound*. When a system tolerates a maximum number of occurrences of state inconsistencies globally for each communication cycle, we define the *replay bound* as *global replay bound*. Figure 3 depicts an example of a schedule with the global replay bound which is set to ten.

Example 5 *The fault-tolerant schedule contains five checkpoints and ten data slots. We assume that the global replay bound is ten. Therefore, the system can handle ten repeated state inconsistencies at any slot in a cycle. Suppose, state inconsistencies occur three times between the slots s_1 and s_4, twice between s_5 and s_6, five between s_7 and s_{10}. Since the global replay bound is ten, the system can tolerate these ten state inconsistencies. If state inconsistencies occur six times in between s_7 and s_{10}, then the system will unable to tolerate an additional state inconsistency because of exceeding the limit on the global replay bound.*

The other way to handle repeated state inconsistencies is to set a replay bound at the data slots. The replay bound may vary based on weights or the importance of the contents of data slots. For example, Fig. 4 shows an example of the replay bound at checkpoints and Fig. 5 shows an example of the weighted reply bound. The maximum number of occurrences of state inconsistencies must be equal to the summation of all replay bounds at data slots.

8 Defining fault-tolerant dynamic schedules

When providing the definition of dynamic TDMA schedules, the authors of [15] assumed single-segmented bus networks. They assumed that the communication medium provides a reliable atomic broadcast service and either all stations receive a message or none of them do. This assumption is inapplicable for systems with unreliable channels such as found in wired or wireless communications. We now adapt dynamic TDMA or state-based TDMA schedules to accommodate the fault-tolerant functionality so that we can use the original abstraction for unreliable channels (see Fig. 6). We assume that communication slots are large enough to accommodate recovery activity. Since the system is well defined and static (see Section 2), we can check this at design time.

Definition 3 (Fault-tolerant dynamic schedule) *A fault-tolerant tree schedule is a tree defined by the tuple (V, v_0, V_F, sl, κ, E, E_{cp}) 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 is a mapping $sl : V \rightarrow B$ that maps a state to a broadcast communication associated with that state*
- *κ is a set of clocks with $|\kappa| \geq 1$*
- *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.*
- *A checkpointing transition or rollback edge E_{cp}:(v_s, λ, v_d) represents an transition from a state v_s to a state v_d. λ is a set of updates on clock values.*

E_{cp} is a special form of E. We can implement E_{cp} using E by assuming a specific fault tolerance guard for E. A location that detects a fault can use an edge in E_{cp} for rolling back to a checkpoint. We determine E_{cp} for a given schedule.

Given a checkpointing transition (v_s, λ, v_d), the state v_s refers the location of fault detection and v_d refers the location of a checkpoint.

9 Generating fault-tolerant schedules

A system may allow provisioning extra resources at the design time and perform schedulability analysis at run time before placing a checkpoint such that all messages in the system meet their deadlines. However, in this paper, we propose an optimization framework to generate fault-tolerant state-based schedules from the given

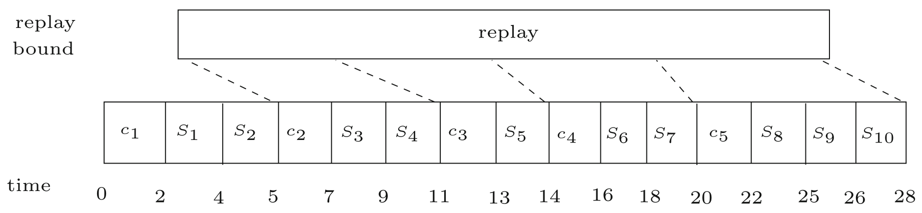

Fig. 3 Global replay bound for checkpoints

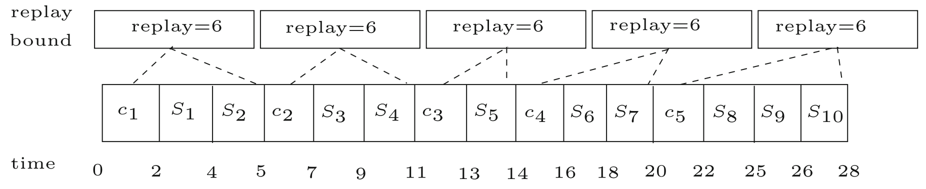

Fig. 4 Replay bound for checkpoints

timing requirements of a checkpoint and the messages that are schedulable. This framework optimizes fault tolerance and therefore places the maximum possible number of checkpoints after meeting the timing requirements of all messages. For simplicity, we assume that the timing requirement of each checkpoint is the same. The timing requirement of messages includes the faulty decision detection and rollback overhead. The schedulability analysis is performed on the messages in system (Eq. 1) before placing checkpoints. The optimization framework ensures that each message in the system gets at least the number of slots to meet the execution time requirements. If the schedulability analysis fails, the system designer may change the number of messages or the specification of checkpoints.

$$U(v_k) = \sum_{\tau_i \in v_k} \frac{e_i}{p_i} \leq \frac{B}{L} \tag{1}$$

where B is the bandwidth assigned to scheduled messages $\tau_i \in \{v_k\}$ and L is the link capacity, with $B \leq L$. This is because, we divide the communication cycle into time slots and each time slot can be occupied by a message according to our scheduling generation policy. Since each time slot can be assigned if needed, the worst-case usage of the communication link will be $\frac{B}{L}$.

We address the challenge of generating fault-tolerant state-based TDMA schedules using a number of constraints that are specific to messages requirements and to the characteristics (i.e., non-preemptive). We formulate a set of constraints using message requirements for each of the reachable and schedulable state in the system. The constraints specify that the messages at least get the required computational time units and no two stations get the same time slot in the same state.

To generate fault-tolerant state-based schedules, we formulate a number of constraints that are specific to message requirements and characteristics (i.e., non-preemptive). The constraints only refer to the messages for the reachable and schedulable states. The constraints specify that the messages at least get the required computational time units and no two stations get the same time slot in the same state. We also formulate constraints to represent the timing requirement of a checkpoint.

The computation time of a message in the system is to obtain at least the required number of slots for each message in its period. A boolean variable $x_{ij}^k = 1$ if a message i is allocated at a slot j in state k, and 0 if otherwise. Therefore, the summation of x_{ij}^k is greater than the computation time (c_i^k) required by a message i in state k. Transmitted messages are non-preemptive. Therefore, if a message is allocated to a slot in a state, the message will be allocated to the subsequent slots until the timing requirement of that message is met.

The formulation of the schedule generation problem is shown as an integer linear problem (ILP). Assume the following:

- V is the set of states in the system.
- N is the set of messages in each state $(\tau_1, \tau_2, \ldots, \tau_n)$.
- c_i^k is the computation time for each message τ_i in state k.
- p_i^k is the period for each message τ_i in state k and P^k is the set of all p_i in state k.
- α_i instances for every message τ_i such that $\alpha_i = \frac{LCM(P^k)}{p_i^k}$.

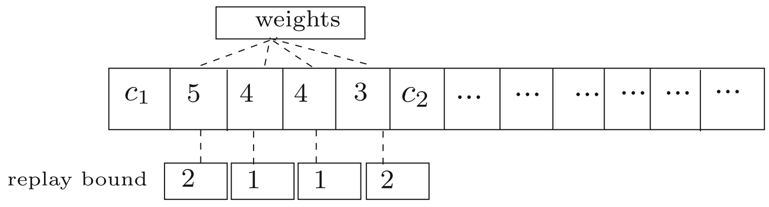

Fig. 5 Weighted replay bound for checkpoints

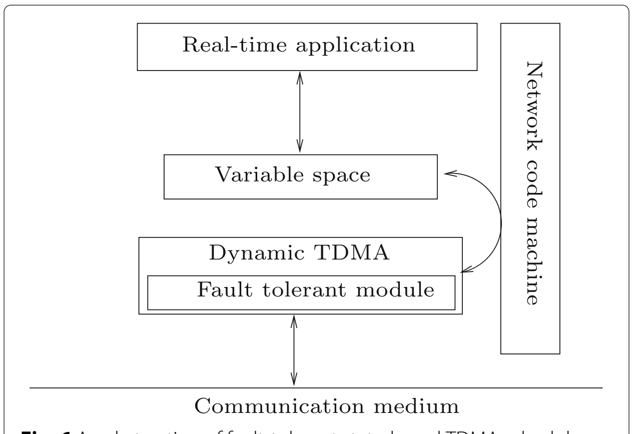

Fig. 6 An abstraction of fault-tolerant state-based TDMA schedule

- x_{ij}^k, the coefficients showing the usage of a time slot for $i \in N, j \in \{1, \ldots, LCM(P^k)\}$ and $k \in V$. These coefficients are defined as follows:

$$x_{ij}^k = \begin{cases} 1 & \text{if message } \tau_i \text{ uses slot } i \text{ in state } k \\ 0 & \text{otherwise.} \end{cases}$$

- s_{ij}^k, the coefficients showing the usage of a time slot by a checkpoint for $i \in N, j \in \{1, \ldots, LCM(P^k)\}$ and $k \in V$. These coefficients are defined as follows:

$$s_{ij}^k = \begin{cases} 1 & \text{if a checkpoint uses slot } j \text{ when } x_{ij}^k = 0 \\ 0 & \text{otherwise.} \end{cases}$$

Optimization model

$$\max \sum_{\forall i \in N, j \in \{1,\ldots,LCM(P^k)\}, k \in V} s_{ij}^k.$$

$$st. \quad C_1\{\forall i \in N, k \in V\}:$$

$$\sum_{j=gp_i^k+1}^{gp_i^k+p_i^k} x_{ij}^k \geq e_i^k \;, g = 0, \ldots \alpha_i - 1;$$

$$C_2 \left\{\forall j \in \left\{LCM(P^k)\right\}, k \in V\right\}:$$

$$\sum_i x_{ij}^k \leq 1,$$

$$C_3\{\forall u \in N, j \in \left\{LCM(P^k)\right\}, k \in V, q \in \{1 \ldots j-1\}:$$

$$\sum_i x_{iq}^k + x_{uj}^k \geq e_u^k,$$

$$C_4 \left\{\forall i \in N, j \in \left\{LCM(P^k)\right\}, k \in V\right\}:$$

$$s_{ij}^k \leq x_{ij}^k,$$

$$C_5\left\{\forall i \in N, j \in \left\{LCM(P^k)\right\}, k \in V\right\}:$$

$$s_{ij}^k \geq 0,$$

Table 1 An example of a buffer table

Time	*a*	*b*	*c*	*d*	*e*
$t_1 - t_0$	T_x	T_x	ϕ	ϕ	ϕ
$t_2 - t_1$	T_x	T_x	T_x	T_x	T_x
$t_3 - t_2$	T_x	T_x	T_x	T_x	T_x

The objective of the optimization model is to maximize the placement of checkpoints. Constraint C_1 specifies that all messages at least get the computation units in their periods. Constraint C_2 specifies that no two messages are assigned to the same slot at the same state. Constraint C_3 specifies a message is allocated to the consecutive slots until timing requirement is met. Constraint C_4 represents that a checkpoint is placed to a slot if no message exists to allocate to that slot. Constraint C_5 specifies the range on the number of checkpoints. Table 2 shows timing specifications of three messages in each of the four states. After running the AMPL/CPLEX optimizer with these constraints for maximizing the placement of checkpoints, we get the total number of checkpoints equal to 40 such that the execution time for a checkpoint is 1 time unit.

10 Experimental analysis

To demonstrate the advantage of using state-based TDMA and need for fixing the issues that arise in practice, we provide a number of experimental results. The experimental setup contains an industrial testbed that runs hard real-time applications. We have used Quanser rapid prototype environment QUARC [26] that has been used for running several real-time experiments such as double inverted pendulum control, unmanned aerial vehicles, unmanned ground vehicles, and mobile robots [27]. We implement state-based schedules to run applications using QUARC.

To analyze state inconsistency, we have chosen two applications: a drive-by-wire (DVW) application and a position control system (PCS) application. The DVW application uses human-machine interfaces that send command signals to the electromechanical controllers via a communication network. Faults are common by nature in a DVW application, and therefore, it should tolerate them for reliable operation. One of the reasons for such faults is the unreliable communication. A triple modular redundant system reduces the impact of faults. Although both static TDMA and state-based

Table 2 Timing requirements for different states (in time units)

System state	Timing of messages (e,p)
1	$\tau_1(1,8), \tau_2(1,8), \tau_3(1,8)$
2	$\tau_1(1,8), \tau_2(1,4), \tau_3(1,8)$
3	$\tau_1(1,8), \tau_2(1,8), \tau_3(1,4)$
4	$\tau_1(1,8), \tau_2(1,4), \tau_3(1,4)$

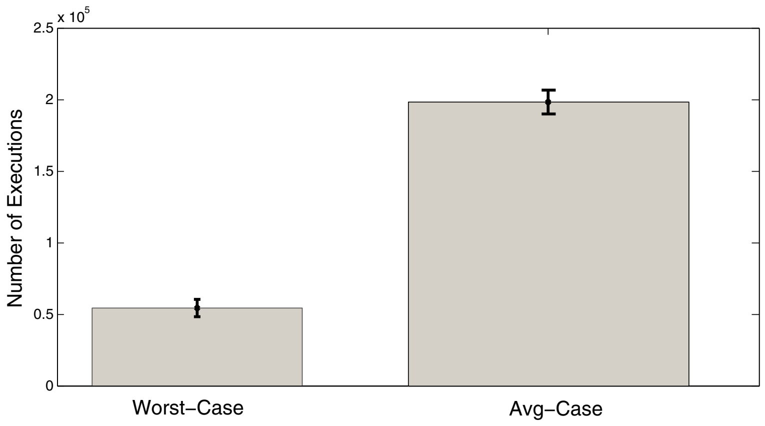

Fig. 7 Worst-case versus average case execution analysis for the DVW application

schedules can represent such system operation, a state-based TDMA schedule outperforms static TDMA when the first two transmissions are correctly received and sufficient enough. The PCS application tries to control a load shaft connected to a rotary motor in the presence of deliberately injected noise. The PCS is a feedback control loop-based system, where we use TMR to reduce the effect of the measurement faults.

The architectural setup for running both the DVW and PCS applications is different due to their operational requirements. For the DVW application, we have used the Gumstix Verdex Pro XL6P series board which has the 400 MHz processor Marvell PXA270 with XScale, a dual-core ×86, and a quad-core ×86 machine. We have used four different configurations for the DVW application: (1) local ×86, (2) remote ×86, (3) ×86-verdex, and (4) verdex. In the local ×86 configuration, both host and target system architecture are ×86 and they communicate through the wireless port. In the remote ×86 configuration, the host and the target system have the same architecture i.e., ×86 but are located distantly and they communicate via wireless. In the ×86-verdex configuration, the host architecture is ×86, and the target architecture is armv5te. In the DVW experiment, we set the buffer size 1460 bytes and the sampling rate 50 Hz. On the other hand, the PCS setup contains a hard real-time operating system QNX connected to a local ×86 machine to control the position of a load shaft.

10.1 State-based TDMA scheduling vs static TDMA scheduling

An advantage of using state-based TDMA for a TMR-based application is that TMR can be used for fault tolerance and state-based TDMA adds flexible behavior such that the system can discard the transmission of the third sample or can run a background message upon receiving first two successful sample transmissions. Therefore, the best case or average case is to receive two samples and the worst case is to receive all three samples. For the DVW application, Fig. 7 shows that the rate of receiving all three samples in each communication cycle is almost 5% less than the rate of receiving two samples to make a decision in the presence of only communication faults. However, this result uncovers the effect of measurement faults that can occur in the system. Using the PCS application, we have analyzed the effect of different rates of measurement faults as shown in Table 3. For the TMR-based PCS application, Table 3 shows the percentage of saved resources when the system uses state-based TDMA instead of static TDMA. We see from Table 3 that the rate of preserved resources increases with the decrease of measurement faults. Therefore, using state-based schedules provides efficient use of TMR in applications. The value μ indicates the mean value, and σ denotes the standard deviation. The confidence interval for the PCS experiments is 95%.

10.2 State inconsistency analysis

State inconsistency does not occur in static TDMA because of no points that can lead to multiple decisions. However, it is possible to occur in state-based schedules. We have found after running the applications for almost a day that the percentage of state inconsistency is less than 9.5%. The number of faulty decisions that lead to state inconsistency also varies with the number of stations (see Fig. 8).

Table 3 Statistical analysis for the TMR-based position control system application

Measurement faults (%)	Worst-case resource savings (%)	μ (%)	σ (%)
5	33.2822	33.3043	.0127
20	16.5915	16.6235	.0188
50	4.6414	4.6725	.0290

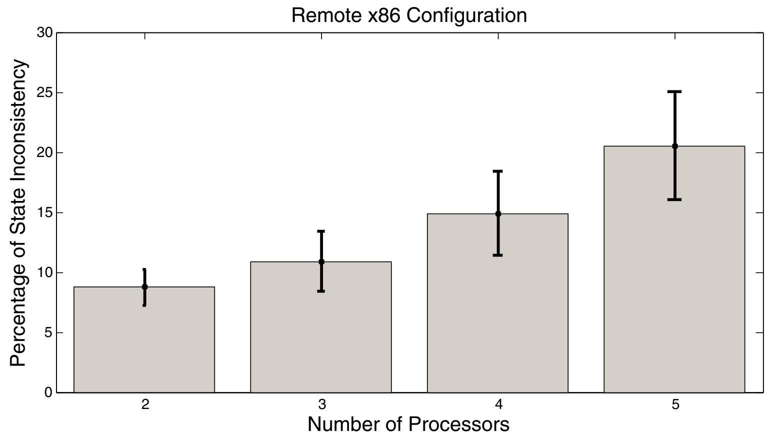

Fig. 8 Faulty decisions for different number of processors

10.3 Resolving state inconsistency

We use the methods described in this paper to resolve state inconsistency. The number of faulty decisions decreases (see Fig. 9) when we use the clock synchronization algorithm and the sampling rate drift management policy as described in [20]. The number of state inconsistencies decrease because of using clock synchronizaion, and Fig. 10 shows that the overhead of using clock synchronization methods are small enough to meet the requirements of execution for computation and communication. To resolve the remaining state inconsistency, we generate fault-tolerant schedules using the optimization solver. The schedules will contain checkpoints if placing them does not violate the timing requirements of the messages in the system.

The system experiences small amount of jitter while running the DVW application (Fig. 11). In the PCS application, jitter is low compared to that in the DVW application, because real-time system QNX has been used as the target with which PCS is connected.

11 Discussion

This work proposes the generation of fault-tolerant state-based schedules for real-time systems. The paper deals with state inconsistencies which can occur in schedules that have the capability of conditional executions. The conditional execution capability of communication scheduling makes the appearance of state inconsistency problem unique. The state consistency arises when there are faults in the system. Therefore, in this paper, the fault tolerance capability is embedded through generating the schedules according to the real-time constraints. This fault tolerance is required for the efficient operation of state-based schedules, specially in unreliable medium. Since state-based schedules have been already proven to perform well in situations [7, 8, 28], the fault tolerance is much needed to increase the applicability in different environments. Moreover, if the schedule is fault-tolerant, it will work well for both reliable and unreliable environments because we are designing schedules for the worst-case.

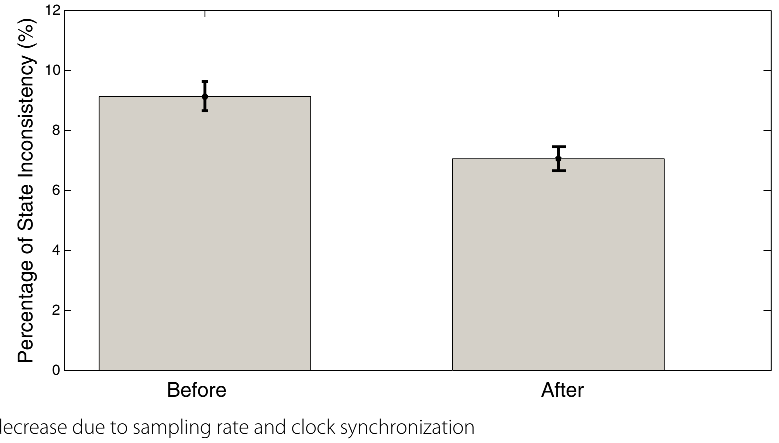

Fig. 9 State inconsistency decrease due to sampling rate and clock synchronization

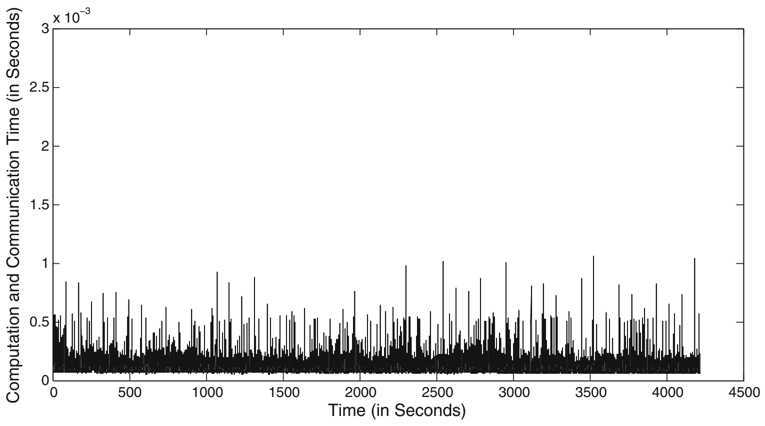

Fig. 10 Computation and communication time for the sampling period of 0.020 s

12 Related work

Kopetz et al. [18] propose the C-State-based CRC [18] method that can detect state inconsistencies using certain information such as TDMA slot information, current mode, global time, and membership information. The authors use the C-State-based CRC for static TDMA which can be extended to state-based TDMA for detecting state inconsistencies. Schemes like two-phase commit (2PC) scheme or the three-phase commit (3PC) scheme can also be used for detecting state inconsistencies; however, these schemes have more communication overhead than the C-State-based method due to a significant number of message transmissions.

State-based TDMA schedules [7, 8] demonstrate high-confidence real-time software characteristics such as deterministic behavior, meeting deadlines, verification, and separation of concerns in addition to making on-the-fly decisions at run time. A number of work on state-based schedules refer to build high confidence software for safety-critical systems such as networked medical devices [29]. However, state-based schedules assume to operate in the presence of high reliable communication channel which limits the applicability of the scheme in unreliable environments for operating correctly and timely because of the occurrence of state inconsistencies.

13 Conclusions

The conditional execution capability of the state-based schedule makes it advantageous in many systems such as TMR-based applications. However, this conditional execution capability creates challenges in distributed agreement because of faulty decisions, particularly in wireless architectures. Hence, in this paper, we have identified the reasons that make faulty decisions and proposed inconsistency reduction schemes and recovery schemes. This knowledge of reduction, detection, or recovery process can be encoded at the specification level when state-based schedules are generated. Experimental analysis demonstrates the necessity of generating fault-tolerant state-based schedules.

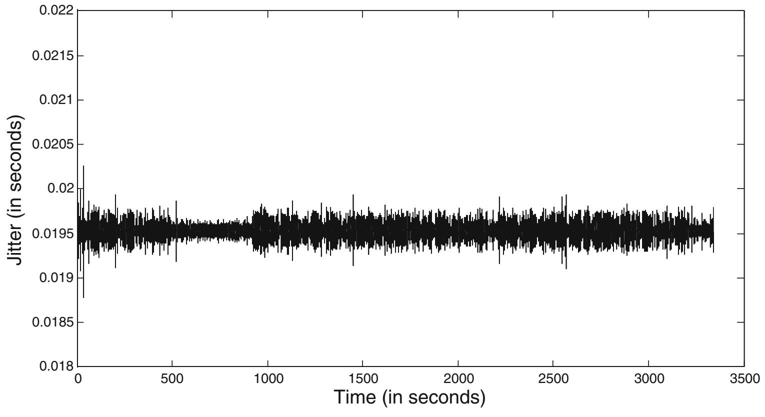

Fig. 11 Jitter analysis

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Received: 30 March 2017 Accepted: 6 October 2017
Published online: 26 October 2017

References

1. A Avizienis, J Laprie, B Randell, in *IARP/IEEE-RAS workshop on robot dependability: technological challenge of dependable robots in human environments*. Fundamental Concepts of Computer System Dependability, (South Korea, 2001), pp. 1–16
2. R Koo, S Toueg, Checkpointing and rollback-recovery for distributed systems. IEEE Trans. Softw. Eng. **13**(1), 23–31 (1987)
3. A Girault, C Lavarenne, M Sighireanu, Y Sorel, in *International Conference on Distributed Computing Systems*. Fault-tolerant static scheduling for real-time distributed embedded systems, (USA, 2001), pp. 695–698
4. D Boley, GH Golub, SN Saxena, EJ Mccluskey, Floating point fault tolerance with backward error assertions. IEEE Trans. Comput. **44**, 302–311 (1995)
5. V Izosimov, P Pop, P Eles, Z Peng, in *Conference on Design, Automation and Test in Europe*. Design optimization of time- and cost-constrained fault-tolerant embedded systems, (Germany, 2005), pp. 864–869
6. H Kopetz, R Obermaisser, P Peti, N Suri, in *Technische Univ. Wien, Vienna, Austria. Rep. 22*. From a federated to an integrated architecture for dependable embedded real-time systems, (Austria, 2004)
7. S Fischmeister, R Trausmuth, I Lee, Hardware acceleration for conditional state-based communication scheduling on real-time Ethernet. IEEE Trans. Ind. Informatic. **5**, 3 (2009)
8. A Azim, G Carvajal, R Pellizzoni, S Fischneister, in *Proceedings of Design, Automation and Test in Europe (DATE)*. "Generation of Communication Schedules for Multi-Mode Distributed Real-Time Applications", (Germany, 2014), pp. 1–6
9. G Weiss, S Fischmeister, M Anand, R Alur, in *Proceedings of the 12th International Conference on Hybrid Systems: Computation and Control (HSCC)*. Specification and analysis of network resource requirements of control systems, (San Fransisco, 2009), pp. 381–395
10. M Anand, S Fischmeister, Y Hur, J Kim, I Lee, Generating reliable code from hybrid systems models. IEEE Trans. Comput. **59**(9), 1281–1294 (2010)
11. A Easwaran, M Anand, I Lee, in *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS)*. Compositional analysis framework using EDP resource models (IEEE Computer Society, Washington, 2007), pp. 129–138
12. S Chakraborty, L Phan, PS Thiagarajan, in *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS)*. Event Count Automata: a state-based model for stream processing systems (IEEE Computer Society, Washington, DC, 2005), pp. 87–98
13. LTX Phan, S Chakraborty, PS Thiagarajan, in *Proc. of the 29th IEEE Real-Time Systems Symposium (RTSS)*. A multi-mode real-time calculus, (Barcelona, 2008)
14. RE Lyons, W Vanderkulk, The use of triple-modular redundancy to improve computer reliability. IBM J. Res. Dev. **6**, 200–209 (1962)
15. S Fischmeister, O Sokolsky, I Lee, A verifiable language for programming communication schedules. IEEE Trans. Comput. **56**(11), 1505–1519 (2007)
16. P Koopman, T Chakravarty, in *International Conference on Dependable Systems and Networks (DSN)*. Cyclic Redundancy Code (CRC) polynomial selection for embedded networks, (USA, 2004), pp. 145–155
17. R Velazco, A Corominas, P Ferreyra, in *Proceedings of the 17th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT)*. Injecting bit flip faults by means of a purely software approach: a case studied (IEEE Computer Society, Washington, DC, 2002), pp. 108–116
18. H Kopetz, G Bauer, S Poledna, Tolerating arbitrary node failures in the time-triggered architecture. SAE 2001 World Congress, March 2001, Detroit, MI, USA (2001)
19. M Fuegge, E Armengaud, A Steininger, Safely stimulating the clock synchronization algorithm in time-triggered systems—a combined formal & experimental approach. IEEE Trans. Indu. Informatics. **5**(2), 132–146 (2009)
20. A Azim, S Fischmeister, in *International Conference on Emerging Technologies and Factory Automation (ETFA)*. Resolving state inconsistency in distributed fault-tolerant real-time dynamic TDMA architectures, (2011)
21. L Xie, T Li, M Fu, J Zhang, Distributed consensus with limited communication data rate. IEEE Trans. Automatic Control. **56**(2), 279–292 (2011)
22. A Azim, in *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*. Analyzing consensus in multi-mode real-time communication using history information (IEEE, Germany, 2016), pp. 1–6
23. C Ryan, D Heffernan, G Leen, Interactive consistency on a time-triggered real-time control network. IEEE Trans. Ind. Inform. **2**(4), 242–254 (2006)
24. G Leen, D Heffernan, TTCAN: A new time-triggered controller area network. Microprocess. Microsyst. **26**(2), 77–94 (2002)
25. A Tanenbaum, M Steen, *Distributed Systems: Principles and Paradigms. US ed edition*. (Prentice Hall Publishers, 2002), pp. 1–803
26. QUARC Control Design Software (2011). www.quanser.com
27. R Huq, H Lacheray, C Fulford, D Wight, J Apkarian, in *Canadian Conference on Electrical and Computer Engineering (CCECE)*. QBOT: an educational mobile robot controlled in MATLAB Simulink Environment, (Canada, 2009), pp. 350–353
28. X Liu, S Fischmeister, J Ma, X Chen, A Azim, Dts: dynamic tdma scheduling for networked control systems. J. Syst. Archit. **60**(2), 194–205 (2014)
29. D Arney, S Fischmeister, JM Goldman, I Lee, R Trausmuth, Plug-and-play for medical devices: experiences from a case study. Biomed. Instrum. Technol. **43**, 313–317 (2009)