CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Fault-tolerant Distributed Library for Embedded Real-time Systems

Master's thesis in Computer science and engineering

Johanna Gudmandsen

Hashem Hashem

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

A Fault-tolerant Distributed Library for Embedded Real-time Systems

Johanna Gudmandsen
Hashem Hashem

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

A Fault-tolerant Distributed Library for Embedded Real-time Systems

Johanna Gudmandsen, Hashem Hashem

Supervisor: Roger Johansson, Department of Computer Science and Engineering
Advisor: Lars-Berno Fredriksson, Kvaser AB
Examiner: Johan Karlsson, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

A Fault-tolerant Distributed Library for Embedded Real-time Systems

Johanna Gudmandsen
Hashem Hashem
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

A distributed embedded control system (DECS) may have functionality that is safety-critical and time-sensitive, meaning if these systems malfunction the consequences could be devastating. In order to meet these requirements, a system must fulfill real-time constraints and guarantee correct functionality even in the presence of faults.

In this thesis we present a software library providing clock synchronization, real-time scheduling and fault-tolerant decision making. It is intended for use with DECS communicating via controller area network (CAN). To achieve fault-tolerant decision making, we propose an early-stopping fault-tolerance algorithm solving up to t faults in a system of $2t + 1$ nodes. We further propose an adaptation of this algorithm to real-world applications where there may be an interval of correct values instead of one correct value, as assumed in the base solution.

The result is a lightweight and efficient library. The clock synchronization requires one message and has a precision comparable to other known solutions, but is not fault-tolerant. The scheduler runs in $\mathcal{O}(n^2)$ time and uses a non-preemptive rate-monotonic policy. It can handle up to 63 user-defined tasks, and has a worst-case task delay of 2.5 ms for the lowest-priority task in a system with 60 tasks, assuming a task execution time of 0. The drawback is its inability to handle mixed-criticality task sets. Our proposed algorithm utilizes the properties inherent in CAN to provide an efficient way to rectify faults in the value domain. Due to the early-stopping property of the algorithm, the bus utilization increases linearly with the number of faults.

We conclude that while the library is practical and efficient, fault-tolerant clock synchronization and fault handling in the time domain are necessary improvements before the library can be used in production systems.

Keywords: Byzantine fault tolerance, Real-time scheduling, CAN, Distributed systems, Embedded control systems

Acknowledgements

We would like to thank Kvaser AB for the opportunity to realize our thesis idea, and for providing us with the necessary tools to complete our thesis. A special thanks to our supervisor at Kvaser, Lars-Berno Fredriksson, for his help with scoping the project and his valuable feedback throughout.

Further, we would like to thank our supervisor at the Computer Science and Engineering department, Roger Johansson, for helping us reach the required scientific level, and for the guidance whenever we were uncertain how to proceed. Thanks to our examiner Johan Karlsson as well, for his input on the thesis.

Lastly, thanks to our opponents and our peer reviewer for the valuable feedback on our thesis, and thanks to our colleagues at Kvaser for their great tips and enjoyable fika breaks.

Johanna Gudmandsen, Gothenburg, June 2020
Hashem Hashem, Gothenburg, June 2020

Contents

List of Figures

1

Introduction

In distributed embedded control systems (DECS) several individual subsystems cooperate in order to perform certain functions in an embedded system. Some of these functions may be safety-critical and time-sensitive, meaning if these systems malfunction the consequences could be physically and economically devastating. An example of a DECS is a car with four brakes, where braking force has to be applied at the same time across all brakes in order to brake without changing the movement direction of the car. As these systems are dependent on the communication between the subsystems, there has to be guarantees on the communication. Guarantees in a time-sensitive system include real-time demands for all subsystems' communication and the executions locally, such as applying brake force. Communication guarantees could also relate to the safety aspect, i.e. the system must have fault tolerance on both the communication level (the network) and the system level [1], [2].

Fault tolerance of a system is what makes its action secure even when facing malfunctioning computer nodes. Even though a computer in the vast majority of its operating time runs without faults there is always a risk of it producing the wrong results [3]. Faults can affect the service of a computer system in both the value and time domains and be caused by attacks as well as environmental differences, hardware and software faults [4]. Because of this, it is important to think about possible faults that can occur in a system, and putting effort into mitigating these in the development phase.

A common problem in distributed computer systems is the Byzantine generals problem, where nodes in the system have to reach agreement in the presence of faults [5]. A Byzantine fault occurs when nodes have different perceptions about the values of one or more variables. Nevertheless, the system must be able to operate correctly in the presence of such faults, namely by producing the correct result. A system being resistant to these faults is said to be Byzantine fault tolerant and will be the subject of this thesis.

1.1 Aim

The goal of this thesis is to build a software library for a uniprocessor real-time operating system (RTOS) for use in DECS communicating via controller area network (CAN). The software library should provide clock synchronization, real-time scheduling with hard real-time constraints and fault-tolerant decision making achieved through a novel Byzantine fault tolerance algorithm for DECS which utilizes the strengths of CAN.

1.2 Limitations

We limit the scope of this project to achieve greater depth in the areas of focus. The limitations are the following:

- We consider only faults in the value domain for our fault-tolerant algorithm.
- We assume that the system runs on a wired network where the nodes are at most a few meters from each other, meaning we do not consider communication delays incurred from distances between nodes.
- We assume the nodes are homogeneous when it comes to hardware. As such, we can disregard architectural differences between nodes which allows us to design all components of the library for one node's hardware and assume all replicas act identically.
- We assume all nodes in the OS can execute all tasks in the system. This allows us to assume that fault detection and recovery times are not be affected differently by different nodes failing.
- We assume that the system needs to handle applications that require response times in the order of 10 to 20 ms.

1.3 Method

The thesis work consisted of three major parts. These were a literature study, an implementation phase and a testing phase. In the literature study, we studied our underlying building blocks RTOS and CAN, to understand the system's capabilities and limitations. We also looked at real-time scheduling and clock synchronization approaches, and decided upon an approach serving as a base for the library. Lastly, we looked at Byzantine fault tolerance methods, in order to gain inspiration for proposing and proving our algorithm.

After studying the literature, we started by proposing our algorithm and proving it, so we could begin implementing the library. First, we set up a suitable software structure in order to ease testing later. Then, we implemented both a scheduler and a clock synchronization method based on the findings of the literature study. Finally, we implemented the proposed algorithm.

The testing phase consisted of three tests. We tested the clock synchronization, the scheduler and the proposed algorithm separately. In the clock synchronization test, we measured the drifts of two different nodes' clocks compared to a reference node's clock. Then, we tested the scheduler by measuring the worst-case task delay for task sets of different sizes. The tasks' execution times were 0, in order to show the scheduler's running time. For the implementation of our algorithm, we measured the bus utilization for different message lengths and different amount of faults.

1.4 Structure

The remainder of the thesis is structured as follows. Chapter 2 details the findings of the literature study and explains the theory needed to understand the reasoning behind design choices. In Chapter 3 we propose and prove the correctness of the fault tolerance algorithm. This is followed by an explanation of the system implementation in Chapter 4. In Chapter 5 the test results are presented followed by a discussion in Chapter 6. We conclude by reflecting upon our work as a whole in Chapter 7.

2

Theory

This chapter provides all building blocks and background theory necessary to follow the rest of the paper. First, we describe the OS running at each node of the distributed system, as well as the network with which they communicate, CAN. Second, we briefly introduce the area of real-time scheduling and different scheduling techniques. Third, the concept of clock synchronization and its purpose in a distributed system is explained. Last, we describe how Byzantine faults can occur in these types of systems and more importantly, the significance of protecting the system against them.

2.1 Rt-kernel

The system is built upon *rt-kernel*, an embedded real-time operating system [6]. Rt-kernel provides primitives necessary for real-time scheduling as well as device drivers and interrupt handlers. In the case rt-kernel detects an error, it enters an error handling routine, simplifying error handling for the RTOS user. The rt-kernel scheduler is a preemptive fixed-priority scheduler.

An rt-kernel task can be in one of four states: *ready*, *waiting*, *running* or *dead* [6]. A task in the *ready* state is ready to run, and will be scheduled according to its user-defined priority. When it is time to run the task, it is put in the *running* state, and runs until it is either finished or a higher priority task enters the ready-state, in which case its execution is interrupted. A task waiting for a specific event to occur is put in the *waiting* state and remains blocked until the event occurs, at which time it enters the ready-state. A *dead* task has finished its execution and its allocated resources will automatically be freed by rt-kernel.

The task priorities of rt-kernel are in the interval of 0 to 31, with 0 being the lowest-priority. The RTOS itself keeps two tasks ready at the lowest priorities [6], an idle task which runs whenever no other task occupies the processor (thereby having priority 0), and a reaper task which reclaims allocated data from finished tasks. The reaper task is called upon whenever a task enters the dead state, and has priority 1.

Tasks originating from the user application are either within the priority interval 2 to 31 or time-triggered. The time-triggered tasks must be defined statically in a file by means of a schedule, as more tasks may not be defined once the application

starts running. However, it is possible to define several schedules and switch between them at runtime. All time-triggered tasks have priority 31, with only time-triggered interrupts having higher priority. The priority hierarchy can be viewed in Figure 2.1.

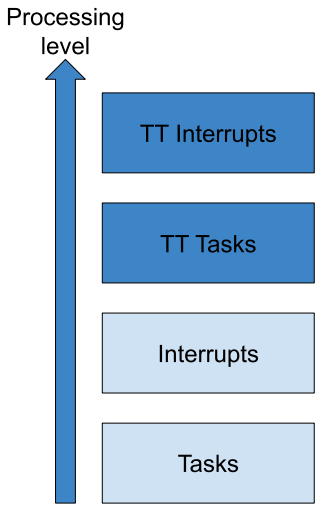


Figure 2.1: Priorities in rt-kernel. Figure inspired by [6].

Rt-kernel uses *ticks* to count time. A tick occurs when the *SysTick* timer overflows, which is once every millisecond by default. For every tick the `tick()` method is invoked, which increments the rt-kernel clock, handles hardware and software interrupts and performs necessary context switches in case a task should be preempted. The interval between each tick can be shortened, however this increases the rt-kernel overhead in the system since then `tick()` runs more frequently.

2.2 Controller area network

CAN has many qualities that make it suitable for use in distributed real-time systems [7]. It is a broadcast bus, which simplifies addressing when working with systems containing many nodes, as every node receives all messages and can either choose to read them all or use a mask to discard unwanted ones. The reliability of CAN's broadcast has been questioned by Rufino [8], however we will not tend to this problem in this thesis. Instead, we assume that the system communicates using a network that guarantees reliable broadcast. CAN messages are broadcast in a non-preemptive manner, meaning once a message begins transmission it cannot be preempted by another message of higher priority.

Moreover, CAN has several error detection mechanisms such as bit monitoring, bit stuffing, frame check, acknowledgment check and Cyclic Redundancy Check (CRC). It also has fault-confinement mechanisms that prevent faulty nodes from blocking the system and allow their exclusion from the communication if they do not recover [9]. All of these properties contribute to the reliability of the CAN protocol.

Bit stuffing happens when five consecutive bits are transmitted that have the same logical value [9]. In order to make sure that the five bits are intentional and not an error, a bit of the opposite value is inserted. This introduces a small amount of overhead and is important to account for when calculating transmission times over a CAN bus.

Every CAN frame has, among other fields, an arbitration field, see Figure 2.2. This is used in the arbitration mechanism, which allows messages with a higher priority to have precedence in case of a collision on the bus [9]. This field consists of an identifier and an RTR-bit. The identifier can be 11-bits or 29-bits large, where a larger field allows for more IDs on the bus if necessary.

The arbitration mechanism works as follows. When a node wants to send a message on the bus, it attempts transmission. If the bus is idle, it succeeds. However, multiple nodes can start transmission at the same time, and as such the arbitration field is used to decide which node gets its message across. In the arbitration field, bits with value 0 win over bits with value 1, which means that CAN-frames with a lower-valued arbitration field always win. By default, nodes that lose the arbitration process reattempt transmission right after the winner has finished transmitting its frame, but this can be disabled in the CAN controller if needed.

The other fields of the CAN-frame are the control field, the data field and the CRC field. The control field includes some information about the frame, the data field contains the data to be transmitted and can be 0 to 8 bytes, and finally the CRC field is used to check for bit errors.

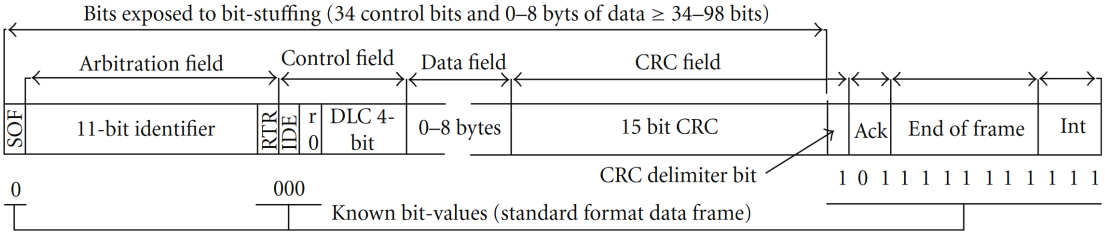


Figure 2.2: CAN-frame with 11-bit identifier [10].

2.3 Real-time scheduling

In hard real-time systems it is critical that all tasks finish executing before their deadline. This problem is typically solved using real-time scheduling algorithms. There are two approaches to scheduling, offline and online scheduling. Offline scheduling concerns the design of predefined schedules by hand or through an algorithm. Online scheduling is the process of deciding the order of execution of tasks during runtime according to a priority assignment policy. Examples of priority assignment policies are *earliest-deadline-first* [11], *rate-monotonic* [11] and *deadline-monotonic* [12]. According to Davis et al. [13], bad priority assignment can have a negative effect on bus utilization. They bring up CAN as an example, where bus utilization could be

improved from 35% to 80% solely by using better priority assignment schemes, see Figure 2.3.

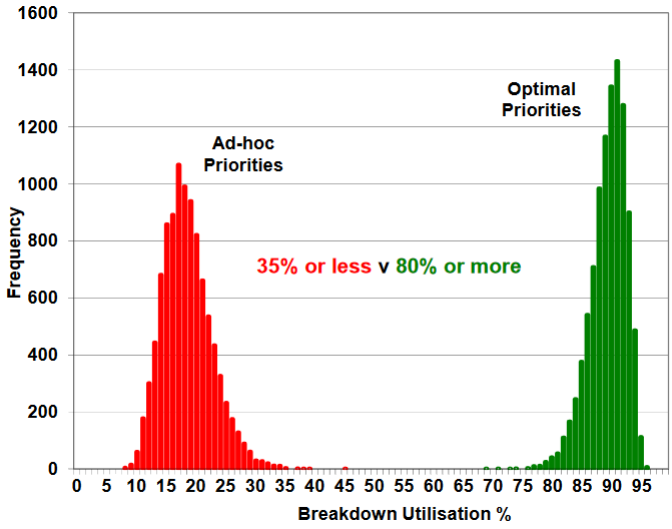


Figure 2.3: An illustration of CAN bus utilization using ad-hoc or random priority assignment compared to an optimal priority assignment [13].

A method that always finds a feasible priority assignment if it exists is said to be *optimal*. One example of an algorithm achieving this is the *optimal priority assignment* (OPA) algorithm [14]. In short, it works by testing schedulability for different priority assignments and chooses the first feasible assignment. OPA is a general solution to fixed-priority scheduling, dealing with arbitrary task sets. When introducing constraints to task sets, there are faster ways to find an optimal priority assignment. For example, for non-preemptive tasks whose deadline is equal to their period, it has been shown that rate-monotonic priority assignment is optimal [15].

2.4 Clock synchronization

In a distributed system, each node uses its own clock to perceive time. Different clocks' oscillators have small variations in frequency due to physical variations that over time causes them to drift in relation to each other [16]. Thus, over time, the nodes in a distributed system will become unsynchronized. For many distributed applications, clock synchronization is a key requirement [17], [18]. For example, consider two nodes in a time-triggered system that must execute some task simultaneously. If their clocks are unsynchronized, the same task will be scheduled at different times at the two nodes which will lead to faulty behavior.

There are two types of clocks synchronization, internal and external. An internal synchronization is when a set of nodes agree upon a global time, and external synchronization is where the global time is approximated with an external physical time, e.g GMT [19].

There are various methods of synchronizing clocks in CAN-based real-time systems.

Some proposed algorithms use a master node which initiates and handles the synchronization functionality of the network's clocks, e.g [20], [21]. In general this is done by a master node transmitting its timestamp and the slave nodes comparing it to their own clock value, or by a master collecting all the nodes' clocks, computing the average time and returning a global time value the nodes correct their clock to. This method is however prone to single-point failures if the master node fails. Lee and Allan [21] try to mitigate this problem by having multiple master candidates.

Another clock synchronization procedure is to use a time translator as described in [22]. Nodes then use a translator to keep track of the time difference between itself and another node in regards to a predefined reference event. This forces the nodes to keep some kind of data structure containing the time translator, and to include functionality that updates the time translator function according to each new reference event. Also, the schedule must include these reference events and one node must be responsible for sending them.

A decentralized technique of clock synchronization is proposed in [23]. When the nodes' clocks are to be synchronized they send messages on the bus until all nodes know the value of all clocks in the system and use this information to agree upon a global time. This method allows for f faulty clocks in a system with at least $f + 1$ correct clocks, but requires many messages for the agreement.

2.5 Fault tolerance

To protect a distributed system from faults there must exist mechanisms to detect and recover from them. The fault tolerance of a system is its capabilities to do exactly this. A fault is defined as a processor's deviation from the program's execution and failure models are used to describe such deviations. We consider three failure models:

- Crash failure: a process stops its execution prematurely.
- Omission failure: a process fails to send one or more messages it was supposed to send.
- Byzantine failure: a process behaves arbitrarily, for example by sending conflicting messages to different nodes.

The three models are listed in increasing severity, i.e. if a system tolerates Byzantine failures, then it also tolerates omission failures and crash failures [24]. Thus, Byzantine failures are the hardest to rectify.

Faults can be classified as symmetric or asymmetric [25]. A symmetric fault implies a fault observed identically by all observers, while an asymmetric fault implies that a fault is perceived differently depending on the observer. A Byzantine fault is an asymmetric fault by definition [25]. For example, in a fully connected point-to-point network there is a communication channel between every two nodes, meaning a faulty node might give conflicting information to different nodes in the network.

CAN on the other hand mitigates this asymmetry since all nodes communicate on one common channel, meaning all nodes observe the same channel and therefore observe the same information.

2.5.1 Byzantine fault tolerance

The Byzantine Generals problem was first introduced by Lamport et. al [5] in 1982. They named the problem of handling conflicting communication among multiple processors after generals of the Byzantine army, where the generals need to agree on a decision. It is also referred to as the consensus problem, and we will refer to both interchangeably.

Driscoll et al. [3] argue that Byzantine fault tolerance (BFT) is an important fault model to include in any system design. They state that it should even be a requirement for control systems, given all of the elements in such a system must work by consensus. Byzantine faults occur in safety-critical systems much more often than the maximum tolerated failure probability of avionics systems, 10^{-9}/h, and as such must not be overlooked [3]. An example of a Byzantine fault is that a value is stuck between a logical 0 and 1, viewed in Figure 2.4. If a value is stuck it could be interpreted as either 0 or 1 which may introduce a Byzantine fault. It could also be caused by slow drivers or time skews in real-time scheduling, among other things.

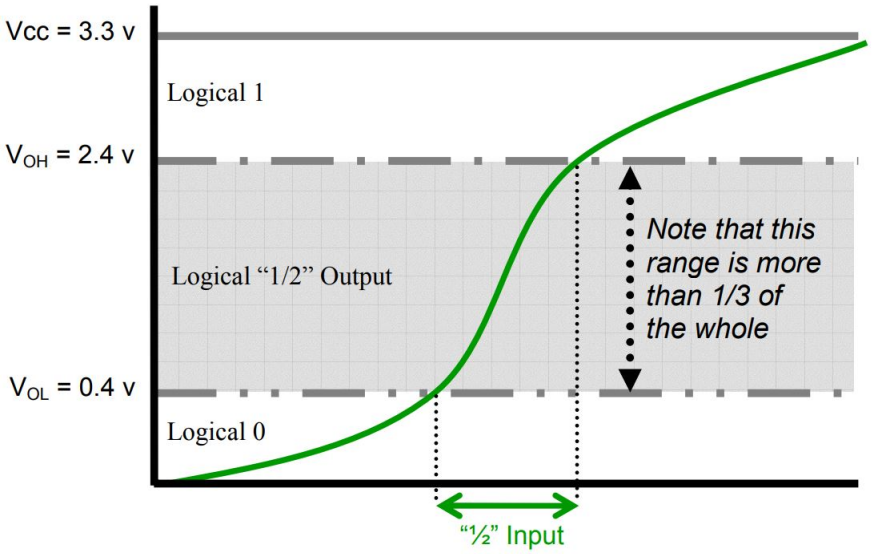


Figure 2.4: A typical Byzantine fault as described by Driscoll et al. [3].

Lamport et al. [5] discuss the problem using *oral messages* and *written messages*, with different properties. The content of an oral message can be tampered with by a relaying node, and its legitimacy cannot be verified by a receiving node. For such messages, $3t + 1$ processors can tolerate at most t simultaneous faults. On the other hand, written messages have unforgeable signatures that can be verified by any processor, allowing the detection of alterations to the content of the message. Using written messages is also referred to as using authenticated processors. In

systems with authentication, it is possible to tolerate up to t faults using only $2t+1$ processors.

The basic idea of reaching consensus in authenticated systems is to assign the same task to $2t + 1$ replicas, where every replica returns its result [5]. The final result is the value proposed by a majority of the replicas. Another well-discussed algorithm is the Practical Byzantine fault tolerance (PBFT) algorithm [26] which was originally made for asynchronous systems and needs $3t + 1$ nodes to operate correctly with at most t faults. An improvement was published later and shows that using synchronous rounds the algorithm requires $2t + 1$ nodes [27]. Relating this to networks assuming reliable broadcast, agreement can be reached by having every node proposing its value and taking the majority as the resulting value [28], which requires $2t + 1$ synchronous rounds.

Consensus algorithms must fulfill three criteria. These are *validity, agreement* and *termination* [29]:

- Validity: all correct processes that have proposed a value v also decide upon v.
- Agreement: no two correct processes decide on different values.
- Termination: every correct process eventually reaches a decision.

The validity constraint is harder to achieve in real-time systems, since information in such systems has two aspects, correctness and timeliness [4]. For real-time information to be valid, it must both have the correct value v and arrive during its designated time of arrival. For real-time consensus algorithms, this means that they must either run on a system where timeliness of messages is guaranteed, or that timeliness is incorporated into the algorithm itself.

Dolev and Strong [30] show that using authenticated processors, there is a need for at least $t + 1$ rounds of information exchange to reach agreement, where t is the maximum number of faults within a system. To clarify, only information exchange between correct processors is included in the count of rounds and the faults considered are crash faults. This holds no matter what type of message or protocol is used.

There has been some work on algorithms tolerating crash faults, that do not need $t + 1$ rounds to reach consensus every time [24], [31], [32]. These are called early-stopping algorithms. In general, when no faults are present in a network, there is no need to reach agreement as every processor already agrees. For such algorithms, it has been proven that the optimal number of rounds is $\min(t + 1, f + 2)$, where f is the actual number of faults and t it the maximum tolerated number of faults in the system [33]. This was proven to hold for crash and omission faults. The main advantage of this approach is the reduced communication overhead incurred when implementing BFT. In systems where BFT is required but errors are uncommon, early-stopping algorithms can provide extra bandwidth to execute non-critical tasks when no errors are present.

3

Last-Proposal-Wins algorithm

In this chapter, we propose an early-stopping fault tolerance algorithm, Last-Proposal-Wins (LPW). It is designed for decentralized distributed embedded systems communicating over a network guaranteeing reliable broadcast, where the nodes are authenticated. This means that the problem of asymmetry is solved in a lower layer protocol, e.g CAN. LPW aims to tolerate faults in the value domain by reaching a consensus among the nodes in the network. As such, LPW combined with CAN reduces the likelihood of Byzantine faults. It requires $n = 2t + 1$ nodes and runs in $\min(2t + 1, 2f + 2)$ communication rounds where f is the amount of actual faults and t is the maximum amount of tolerated faults. To the best of our knowledge, this is a novel algorithm.

3.1 Computation model

In the context of algorithms reaching consensus, the notion of *synchronous rounds* is often used. In such a round, it is assumed that a node first broadcasts a message, then receives a message from all other nodes, then performs a computation using its message and the received messages [31], [33]–[35]. It is also assumed that the network over which the nodes communicate is a full-duplex point-to-point network, where it makes sense to send and receive in the same round. However, the nodes in our system communicate over CAN which is a broadcast-only network, and we alter the definition of synchronous rounds thereafter. We define the term *synchronous broadcast round*: in one round, a node may broadcast one message, or it may receive one message from one node. Additionally, we define a *silent synchronous broadcast round*: a round in which no node broadcasts a message. Henceforth, we will refer to synchronous broadcast rounds simply as *rounds* and to silent synchronous broadcast rounds as *silent rounds*.

The fault model is the following. We limit the algorithm to only handle faults in the value domain, meaning we ignore faults in the time domain. We do not handle inconsistent views of messages, as we will implement the algorithm on CAN, which solves the problem of asymmetry. Hence, LPW handles faults where nodes either fail to respond or they respond with the wrong value. We assume that faulty nodes responding with a wrong value will not violate the protocol specification, e.g. by sending more messages than specified. As such, all non-crashed nodes agree on the

final value.

3.2 Functional description

The purpose of LPW is to make all nodes in the system reach a consensus about the result of a local computation. A flowchart describing its operation is seen in Figure 3.1. LPW is based on two rules:

1. A node receiving a message containing a value that matches its own agrees with that value, otherwise it disagrees.

2. A node that has successfully proposed its value may not propose again.

In the initializing round, a predetermined sender node proposes its value by broadcasting it. We denote the latest proposed message l. Now for each node in the system, it either agrees or disagrees with l. In the next round, every disagreeing node tries to propose its own value. Only one disagreeing node succeeds with its proposal in a given round. The algorithm terminates either once a silent round is reached, i.e. no node disagrees with l, or when $2t + 1$ nodes have proposed. In both cases, l is chosen as the final value. Two example executions of LPW can be viewed in Figure 3.2.

If the predetermined sender node crashes or omits the initial message, the first round becomes a silent round. The algorithm should never end before a value has been proposed, a fault which is rectified by not terminating LPW if the first round is silent. In the next round, all nodes will attempt proposal since no node can agree upon an absent message.

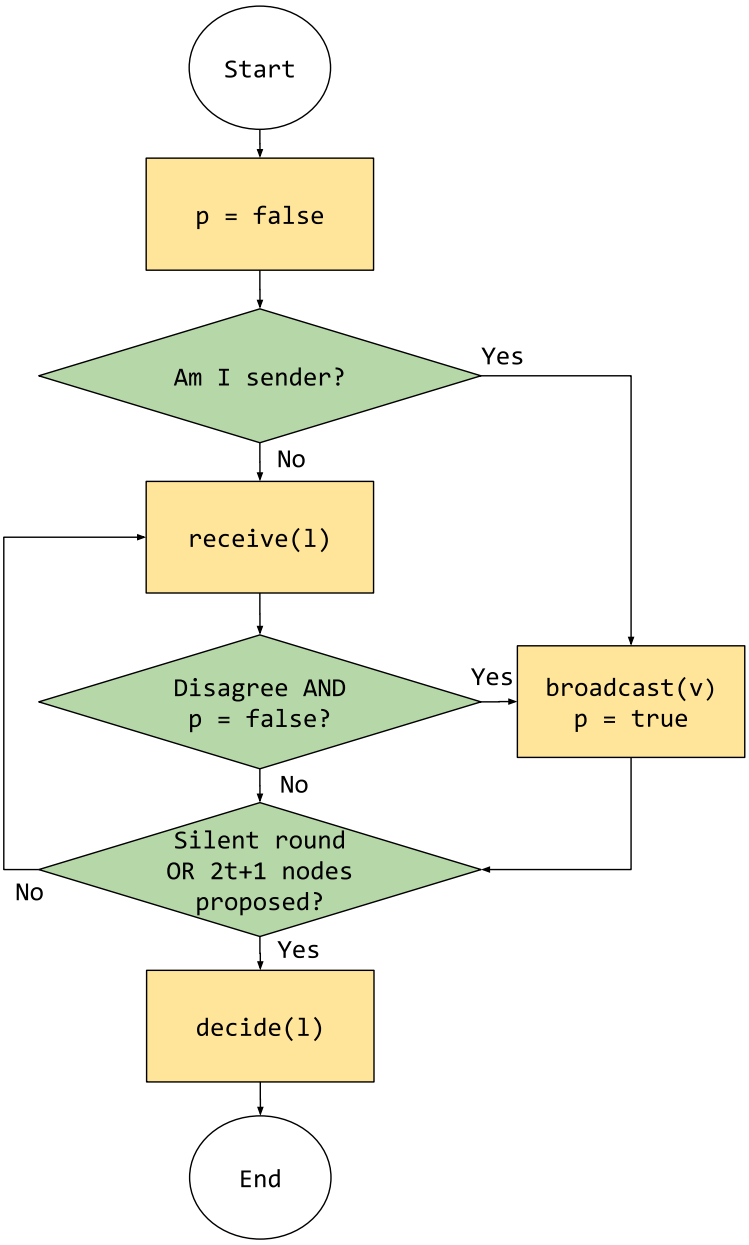
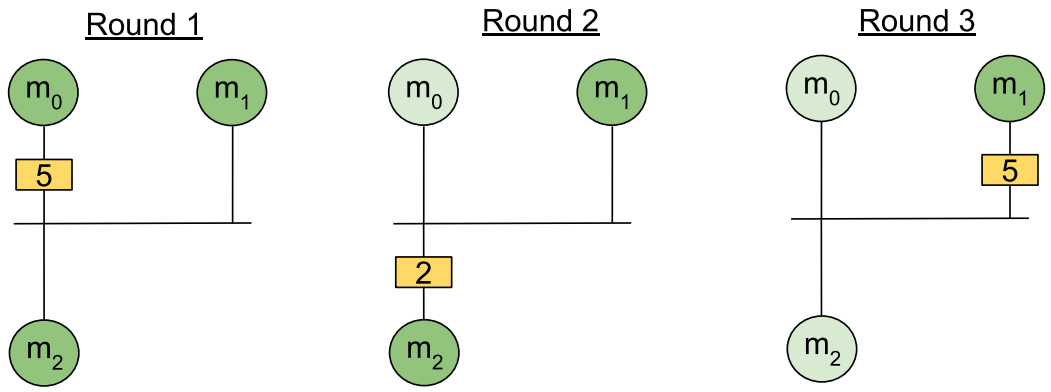


Figure 3.1: Flowchart of LPW running at all nodes. `p` is a flag denoting whether or not the node has proposed a value. `v` is the value computed at the node. `broadcast(v)` is a function broadcasting `v` and `receive(l)` receives the latest value on the bus and stores it in `l`, or stores the node's own value if it is the broadcaster. When `decide(l)` is called, a node decides on `l` as the correct value.

(a)



(b)

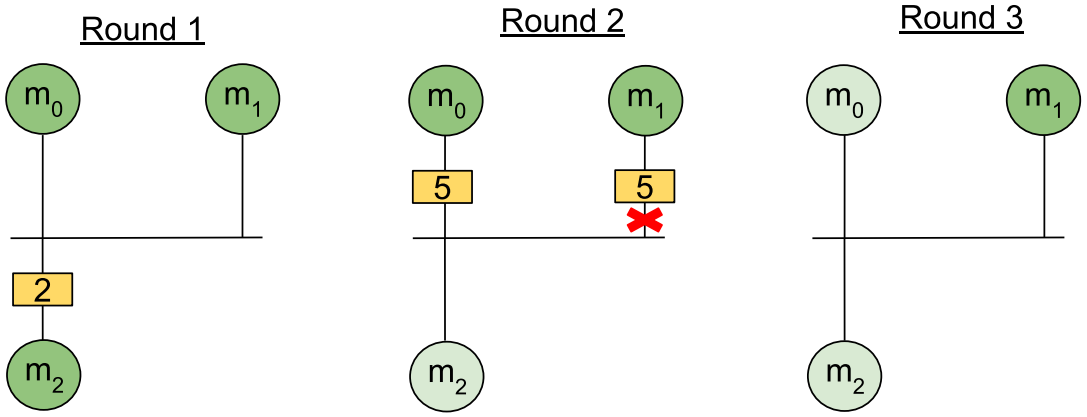


Figure 3.2: Two example instances of LPW. These examples consist of systems with three nodes m_0, m_1, m_2, connected to a CAN bus where m_0, m_1 are correct nodes and m_2 is faulty. Execution 3.2a begins by m_0 proposing its value 5 onto the bus. This is followed by m_2 disagreeing and proposing the faulty value 2. m_1 disagrees with the value of m_2 and proposes 5. This is round $2t + 1 = 3$ and the algorithm terminates with the correct value. Execution 3.2b begins by m_2 proposing its faulty value 2. Both m_0 and m_1 disagree and try to propose, and m_0 succeeds while m_1 does not. Round 3 is silent since the only node left to propose is m_1 which does not disagree with m_0. LPW terminates after this silent round.

3.3 Correctness proof

We claim that the execution of the algorithm will terminate at the latest in $\min(2t+1, 2f+2)$ rounds using a message complexity of $\mathcal{O}(n)$. We also claim that it satisfies the criteria for consensus algorithms: *validity*, *agreement* and *termination*.

We define two sets A and B, where A is the set of correct nodes, and B the set of faulty nodes, where nodes in A or B have not yet proposed. A node is either in A or B but not in both and the total number of nodes is n. Initially, this means that

$$\begin{aligned} &|A \cup B| = n, \\ &A \cap B = \emptyset, \\ &|A| \geq t + 1, \\ &|B| = f,\ f \leq t, \end{aligned} \tag{3.1}$$

where t is the maximum amount of tolerated faults and f is the actual number of faults. Once a node proposes its value it is removed from its corresponding set and cannot propose again. The node that proposed most recently is denoted p. There can only be one correct value and correct nodes always propose the same correct value whereas faulty nodes never propose the correct value.

A node is denoted m_i, $i = \{0, ..., n-1\}$ and its value is denoted v_i. The latest proposed value in the network is denoted l. A node m_i can either agree or disagree with l and in order to perform a proposal a node must disagree with l. Agreeing means $v_i = l$, and disagreeing means $v_i \neq l$. If a node m_i disagrees and proposes v_i, then $l := v_i$. From this, we derive the following relations between the two sets A and B:

A node $m_i \in A$ disagrees with l only if p was in B. (3.2)

A node $m_i \in B$ disagrees with l only if p was in A or p was in B and $v_i \neq l$. (3.3)

For two successive proposals l_k, l_{k+1}, a node agreeing with l_k will always disagree with l_{k+1}, since l_{k+1} originates from a node disagreeing with l_k. (3.4)

Theorem 1. *Validity: All correct nodes that have computed a value v also decide upon v.*

Proof. In the case where $B = \emptyset$ initially, validity is trivial since all nodes' proposals will be correct. When $B \neq \emptyset$ initially, then by 3.1 $|A| > |B|$. As such, the algorithm

will generate at least two more proposals because of 3.2, 3.3 and 3.4, thereby running for at least two more rounds. Whenever a node proposes, the node is removed from the set it was originally in and the set's size is reduced by 1.

Because of 3.3, every p that was in A is followed by a proposal from a node in B in the next round, implying that after these two rounds l is incorrect and both sets have been reduced in size by 1. On the other hand, due to 3.2 and 3.3 every p that was in B is either followed by a proposal from a node in A or B in the next round. This implies that after these two rounds l is either incorrect and $|B|$ has been reduced by 2, or l is correct and both sets have been reduced in size by 1.

Regardless of the order, B is reduced by at least 1 and at most 2 for every two rounds that pass, while A is at most reduced by 1. Since $|A| > |B|$ initially, it must hold that B is emptied first and once $B = \emptyset$ it must hold that $A \geq 1$. Thus, there will always be a correct node that can propose when $B = \emptyset$ without any node being able to disagree. Because all correct nodes have the same value and the last proposal originates from a node in A, it holds that all correct nodes decide upon their own value. Thus, the validity criteria is proven.

□

Theorem 2. *Agreement: No two correct nodes decide on different values.*

Proof. All correct nodes will decide upon the last l and since we have a reliable broadcast, this l is the same at all nodes. This proves that the agreement criteria holds. □

Theorem 3. *Termination: Every correct process reaches a decision.*

Proof. LPW terminates at the round when no node disagrees or at round $2t + 1$. If it ends at round $2t + 1$ it is trivial that all nodes reach a decision at exactly round $2t + 1$. In the case it ends before this, at a silent round, at some point one of the sets A or B must become empty. This must be the case because one node is removed from one of the sets in every proposal. None of the sets are infinitely large, meaning eventually there will be no nodes left disagreeing. At this point a silent round is reached for all nodes and the LPW instance ends simultaneously with the last proposed value as the decision. This proves that the termination criteria holds.

□

Theorem 4. *Early stopping: Every process reaches a decision at the latest in* $\min(2t + 1, 2f + 2)$ *communication rounds.*

Proof. In the case we do not have the maximum amount of faults, i.e. $f < t$, we have claimed the algorithm ends in at most $2f + 2$ rounds. In every two rounds we know $|B|$ is reduced by at least 1, meaning after $2f$ rounds $B = \emptyset$. At this round it must hold that $|A| > 1$ due to 3.1, and if l was incorrect, A will propose in the

next round. This is followed by a silent round since all nodes in A agree with each other, meaning LPW terminates at the latest in round $2f + 2$ in this case.

Similarly, if $f = t$ we need at most $2t$ rounds before B is empty. At this point, l could be incorrect and we must use one more round for the correct value to be proposed. However, in round $2t + 1$ all nodes know this proposed value must be correct and no silent round is needed, meaning the algorithm can terminate in round $2t + 1$. This proves the claim about early stopping. □

Since the algorithm requires $n = 2t + 1$ rounds in the worst case, and one message is sent in each round, the message complexity of LPW is $\mathcal{O}(n)$.

3.4 Adaptation to real-world applications

In real-world applications, there may not always be one correct value, as assumed by LPW. Consider a distributed embedded system where multiple nodes are connected to independent identical sensors measuring the same physical phenomenon. The sensor readings are within a known margin of error σ, meaning two readings from different nodes may differ while still being within the margin of error and both should be interpreted as correct. In its present state, LPW will interpret this as a disagreement between these two nodes. As such, LPW must be adapted to rectify this problem.

To do this, we start by redefining what a node's perception of a correct value is and how it agrees/disagrees. We previously assumed that one node m_i will have one value v_i as its perception of the correct value. Now we relax this assumption so that m_i considers all values in $[v_i - \sigma, v_i + \sigma]$ as correct. We must also redefine what it means to agree with l: m_i agrees with l when $[v_i - \sigma, v_i + \sigma] \cap [l - \sigma, l + \sigma] \neq \emptyset$. Thus, when $|l - v_i| \leq 2\sigma$, m_i agrees with l, otherwise it disagrees.

Now, let us examine the relationships between different nodes given the new definition of agreement. First, we denote the unknown x as the reading an ideal sensor with $\sigma = 0$ would produce. For all correct nodes, it holds that $|x - v| \leq \sigma$. All correct nodes must agree with all other correct nodes, since the maximum value difference between two correct nodes is 2σ. For all faulty nodes, it holds that $|x - v| > \sigma$.

Let us assume the simple case when all incorrect nodes are more than 2σ from all correct nodes. We reuse the definition of A and B from the correctness proof. All nodes in B disagree with all nodes in A and vice versa. Further, all nodes in A agree with each other. When this is the case 3.2–3.4 as well as the correctness proof still hold, therefore LPW keeps operating correctly.

In the other case, where at least one node in A agrees with a node in B, 3.2–3.4 no longer hold. The algorithm as previously defined can agree upon a value that any correct node proposes, but in this situation there are cases where a correct node agrees with a faulty node and LPW can terminate with a faulty value, see Figure 3.3. Consequently, it is possible that all nodes decide on an l for which it holds that

$|x - l| > \sigma$. For a correct node m_i that agrees with l it holds that $\max(|x - v_i|) = \sigma$ and $\max(|v_i - l|) = 2\sigma$. Therefore the value on which all nodes decide is bounded by $x \pm 3\sigma$.

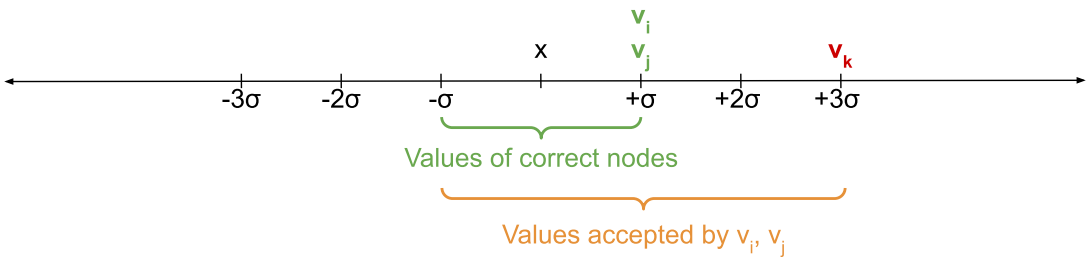


Figure 3.3: A situation where the three nodes agree with each other, even though v_k is a faulty value. If v_k is proposed first, the other two nodes will agree, and the algorithm will terminate with the incorrect value v_k.

3.4.1 Likelihood of faulty termination

For the aforementioned case to occur, a faulty node must have a value v such that $\sigma < |x - v| \leq 3\sigma$. Assuming we know the range of possible values, and that faulty nodes can take on any possible value with equal probability, the probability of this happening can be calculated as

$$\begin{aligned} P(\sigma < |x - v| \leq 3\sigma) &= P(\sigma < |x - v|) - P(|x - v| \leq 3\sigma) \\ &= 1 - \frac{2\sigma}{V} - (1 - \frac{6\sigma}{V}) = \frac{4\sigma}{V}, \end{aligned} \tag{3.5}$$

where V is the amount of possible values. Given a probability p_f that a node fails, the probability that a node both fails and this case occurs is

$$\frac{4\sigma}{V} p_f. \tag{3.6}$$

For example, assume we have a 12-bit sensor with a $\pm 0.05\%$ margin of error, which is sampled once every 200 ms and has a transient fault rate of once per hour. Then for each node, the probability that both the node fails and the situation in Figure 3.3 occurs is

$$\frac{4\sigma}{V}p_f = \frac{4 \cdot 2^{12} \cdot 0.0005}{2^{12}} \cdot \frac{1}{5 \cdot 60 \cdot 60} \approx 1,11 \cdot 10^{-7}. \tag{3.7}$$

Note that this is purely the probability that a faulty node acquires a value that *could lead* to an incorrect decision. The decision itself depends on other factors as well, such as the values of the correct nodes and the order of proposals when a node acquires this value.

4

System model

In this chapter, we describe the implementation details of the software library. We provide an overview of both hardware and software building blocks and present the reasoning behind design choices based on Chapter 2. For complete documentation of the software library, see Appendix A.

4.1 System overview

This project uses a set of STM32 Arm Cortex microcontrollers which make up the nodes in the distributed system. The nodes in the network are identical in every way, i.e. they have the exact same hardware and they run the same task set according to the same schedule at all times. We use rt-kernel at all nodes so they are already equipped with various drivers as well as other utilities typically provided by an OS, such as interrupt handlers, semaphores and timers.

Our work on rt-kernel is an extension in the form of a software library that provides scheduling of CAN messages and other functionality necessary for distributed systems and fault tolerance. As rt-kernel already provides methods and structures for handling the internal workings of a node, our library provides the extra functionality that makes coordinating several nodes easier for the user. Functions provided are a non-preemptive real-time scheduler for CAN messages, a clock synchronization method and functionality for designing fault-tolerant distributed applications.

4.2 Communication protocol

CAN is chosen for communication due to its inherent fault tolerance to bit errors and bus collisions. The implementation of a CAN-based communication protocol is also very flexible, meaning the protocol can easily be designed for different types of systems, such as time-triggered, event-triggered or a combination of both [36]. Additionally, all messages on a CAN bus are reliably broadcast meaning that a message is either received by all nodes or by none of the nodes.

4.2.1 Passing information on the network

When a user wants the nodes in the network to communicate with each other, they must use the `communicate()` method. `communicate()` sends messages between nodes using bounded synchronous rounds and can be run either with or without fault tolerance. The time bound for a synchronous round depends on whether fault tolerance is used or not, and is larger for fault-tolerant communication since more computation time is required. `communicate()` designates a sender node which gets to transmit first. This sender changes in a round-robin manner for every invocation of the method. Swapping senders makes it possible to detect crash or omission faults in nodes when its their turn to send.

When `communicate()` runs without fault tolerance its behavior is straight-forward. The designated sender transmits its value and the other nodes act as receivers, and the returned result is always the sender node's value. If the sender's value is incorrect, then all nodes will also get an incorrect value. Thus, this option should only be used for non-critical transmissions. For transmissions where fault tolerance is a requirement, `communicate()` can be run with the LPW algorithm which is then invoked simultaneously at all nodes. If the user wishes to adapt a periodic system task for use with LPW, the user must keep in mind that LPW in the worst case requires n communication rounds for a system with n nodes. For implementation details of LPW, see Section 4.5.

4.2.2 Unique arbitration field assignment

To avoid arbitration errors, unique arbitration fields are a necessity for CAN messages. Additionally, we want to utilize the arbitration in CAN to prioritize certain messages over others. Our solution is to split the 11-bit CAN ID-field into a *message ID field* and a *node ID field*. The message ID field contains a 6-bit identifier for distinguishing different types of messages, while the remaining bits contain the node ID field, a unique node number which is randomly assigned at startup using an ID-generation procedure. Since CAN frames should be prioritized by message type during arbitration, we place the message ID first in the CAN ID-field. The node ID field is placed second, as the unique node IDs are mainly used to break ties during arbitration. This allows for 32 nodes on the system and 64 different types of messages, which is enough for our purposes. If more node or message IDs must be available then the 29-bit identifier field can be used instead, however this is not included in our implementation.

To guarantee node ID uniqueness, we run an ID generation method at startup. Each node starts by generating a random 11-bit number using a pseudo-random number generator (PRNG) [37], and stores this number in an array A. Then, every node transmits a CAN frame with its generated number in the CAN ID-field. Upon reception of a CAN frame, each node stores the frame's ID-field in A. Finally, A is sorted and every node sets its ID to A's index of its own random number, providing each node with an ID to be used in the 5-bit node ID field. As such, every node will know its unique ID and the total amount of nodes in the system.

In the unlikely event of duplicate IDs, the nodes are restarted and the ID generation begins again. The reason for duplicate IDs could be that the same seeds were used when generating these nodes' random number, which in our implementation changes only at startup, thus requiring a restart to be updated.

The seed used as input to the PRNG is created at startup. To generate an adequate seed for the PRNG there is a need for a source of entropy that differs from node to node. Since the used microcontroller lacks a hardware random number generator [38], we obtain the seed using the unpredictability of startup patterns in SRAM. The seed is created using the method described in Figure 4.1. Even though this method of seed generation is not guaranteed to be cryptographically secure, methods using other characteristics than the ones explained have proven to generate secure seeds from SRAM as discussed in [39], [40], implying that SRAM is a good enough source of entropy.

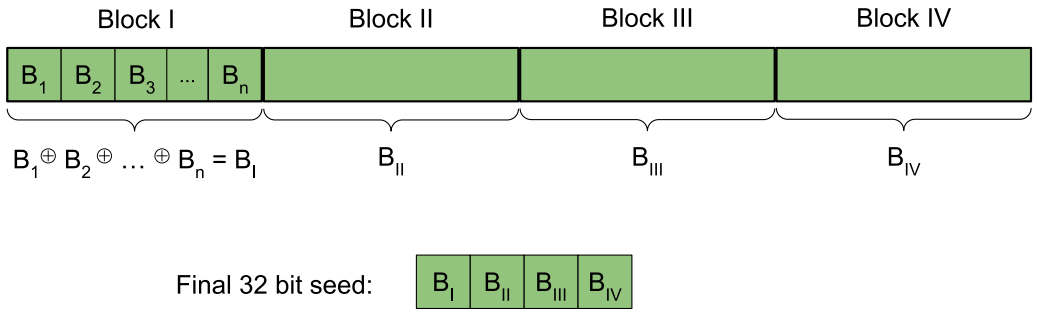


Figure 4.1: Illustration of the seed generation. The SRAM is divided into four blocks and each consecutive byte in these blocks is combined using the XOR operation to obtain one final byte. These four obtained bytes from each block are concatenated, producing a 32-bit seed.

4.3 Clock synchronization

The distributed nodes need to have the same recollection of time for the time-triggered schedule to execute simultaneously throughout the network. Overhead incurred from this synchronization must be bounded in order for its execution time to be predictable. To achieve this the system includes a master-slave clock synchronization task automatically used by the system's scheduler. This is run at a user-defined interval which specifies the amount of hyperperiods between each invocation of the clock synchronization task. The lowest priority is assigned to the synchronization task since it is desirable that it does not have precedence over the user-defined tasks. It is up to the user to leave room in the schedule for the clock synchronization task in order for it to meet its deadline.

The nodes' clocks are assumed to be fault-tolerant, meaning no clock can act arbitrarily, but will follow the protocol of the synchronization. Though, clocks may drift as time passes as they can oscillate at different speeds. The node with ID 0

is chosen to hold the master's clock, which all other nodes will adapt their clock to. Because one node is distinguished as the master node, the method is prone to single-point failures.

As mentioned in Section 2.1, rt-kernel uses ticks to count time. If ticks were to be used for synchronization, the clocks in the system would have to drift more than one millisecond (by default) before the drift could be noticed, which is too inaccurate for our real-time synchronous communication. Instead, we need a precision of microseconds and the synchronization method therefore synchronize the nodes' clocks within tens of microseconds to the master clock. This is achieved by designing a rough synchronization run at scheduler startup and a precise synchronization run at a user-defined interval as a part of the schedule. These methods have millisecond and microsecond precision respectively. Common for both synchronization methods is that none of them actively change the clock value. As a result, the nodes' clocks show different timestamps, however their actions are still coordinated and synchronized.

4.3.1 Rough synchronization

For each node, operations performed before initializing the scheduler may have different execution times for different nodes. As such, we have implemented a rough synchronization which is run before time-critical communication starts, which ensures that deviations within 100 ms from the master is fixed.

The rough synchronization is invoked in each node at scheduler startup, where all slave nodes await a CAN message from the master node. The master node delays for 100 ms before transmitting a clock synchronization message. When a slave receives the master's message, it leaves the rough synchronization. After synchronization, all slave clock drifts are within one tick from the master.

4.3.2 Precise synchronization

The precise synchronization runs as a periodic task. When it executes, the master node sends an empty CAN message as a reference message. The slave nodes wait for this message, using a hardware timer with microsecond precision to calculate the reception time. When the message arrives, they compare the value of the timer with the expected transmission time of the message. If the reception time is different from what was expected, a node is unsynchronized and will modify the length of its next tick to compensate for the drift.

To explain in more detail, let us denote the expected transmission time as e and the reception time as r. e is the combined time it takes for the master node to send the reference message to its hardware, the transmission time of the message on the bus and the time it takes for a slave to fetch the message from its hardware. This value is calculated to 428 µs assuming a bitrate of 125 Kbit/s. r instead is the actual reception time of the message.

If a slave node is in perfect synchronization with the master node then it holds

that $e = r$. Now, if $r < e$ then the slave node entered the synchronization method and started the timer after the master node sent the message, meaning its clock is behind the master's. The opposite holds when $r > e$, the node entered the function and started the timer before the master node sent the message, meaning its clock is ahead. All nodes will await their next tick and once it occurs, rt-kernel's tick timer is stopped and the next tick is manually executed after a modified interval, to compensate for the clock drift. In case the slave node's clock is behind, the manual tick is shorter than a normal tick, and if the slave is ahead the tick is longer. Finally, the slave's tick timer is restarted and it proceeds as normal. This procedure requires a few ticks and is bounded to always use four millisecond's worth of ticks, which could vary depending on the configured amount of ticks per second in rt-kernel. An example showing both a clock that is ahead and one that is behind can be viewed in Figure 4.2.

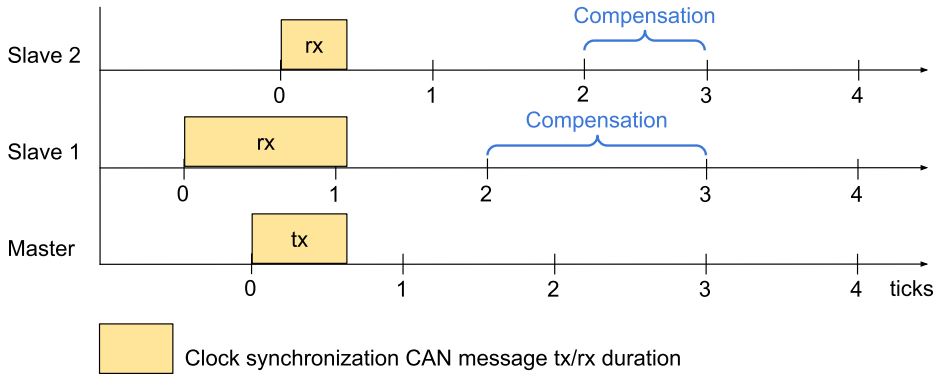


Figure 4.2: An example of how the clocks are synchronized with a master clock node *Master*, a slave node *Slave 1* whose clock is ahead of the master's and a slave node *Slave 2* whose clock is behind the master's. The slaves notice their clocks do not match the master's when their reception time is not equal to the expected transmission time, and at the beginning of the next tick, they compensate by shortening or lengthening the following tick. At tick 4 (assuming 1 ms ticks) the synchronization task is finished and the clocks are synchronized.

4.4 Scheduling

The system's inter-processor tasks (CAN tasks) have different priorities but share the same CAN bus. As such, a CAN scheduler is needed to guarantee the priority ordering of tasks. As our target is DECS with hard real-time requirements, the tasks have to be time-triggered. Event-triggered tasks can be difficult to predict as they can be triggered at any time, making the schedules complex to analyze and verify [36].

The scheduler operates under some assumptions about tasks that use CAN. First,

our scheduler handles only non-preemptive tasks, because messages cannot be interrupted during transmission on a CAN bus. Second, we impose the limitation that all tasks' relative deadlines are equal to their period, and that their first invocation can be offset, after which they run periodically. Third, we assume that CAN tasks are added before the scheduler is started, and no tasks are added or removed during its runtime. Finally, to be able to guarantee hard real-time demands, we assume that the user gives the scheduler the highest priority in rt-kernel and does not use rt-kernel's time-triggered tasks feature, as those tasks always get the highest priority and might preempt the scheduler.

The CAN scheduler schedules a set of user-defined CAN tasks. It is an online scheduler that runs a non-preemptive rate-monotonic (NPRM) scheduling policy which, as mentioned in Section 2.3, is optimal when the tasks' relative deadlines are equal to their period [15]. Additionally, it uses only the configured rt-kernel priority since it keeps its own priority assignment for the CAN tasks, where a smaller number indicates a higher priority. The scheduler is designed specifically for sending messages, and as such, rt-kernel's built-in scheduler should be used when running tasks that do not need to communicate on the bus, as rt-kernel is preemptive.

4.4.1 CAN tasks

The scheduler uses a new task type which is separate from rt-kernel's tasks. These tasks must first be created individually by the user by calling the `can_task_create()` method. This function takes a few parameters which are user-configurable and the scheduler sets some other hidden parameters as well, such as a task's state and its corresponding invocation timer. The four parameters that are user-defined are the CAN task's *period*, the *offset* of its first release, its *callback* function and the *argument* to be sent as input to the callback function.

When a CAN task is scheduled and its callback is run, it is allowed to send to or receive data from the CAN bus. This operation should only occur with the software library's own function `communicate()`. Since the scheduler is the single operator of the bus in a system, the overhead of a task callback which does not include communication should be minimized in order not to waste bus bandwidth.

All of this information allows the scheduler to know when to run a CAN task. A task's period is defined as the amount of ticks between two consecutive releases of the task, and the offset as the amount of ticks before the first release of the task. The state of a task is managed by the scheduler and can be either *ready*, *waiting* or *running*. When a task is released it becomes ready, when it is scheduled it is running and when it finished running and waits for its next release it is waiting.

All tasks' releases are handled by individual rt-kernel timers. A timer expires once the period of its task has been reached. An expired timer runs an interrupt handler which, among other things, sets the CAN task in the ready-state.

4.4.2 CAN scheduler

The scheduler is initialized using the `init_scheduler()` method and requires an array of tasks to schedule and a network object to use for communication. The scheduler creates a task set using the user-defined tasks, checks that the tasks have harmonic periods, then adds a clock synchronization task to the task set. The synchronization task's release interval is specified by the user as the number of hyperperiods between each synchronization. All user-defined tasks are given a priority based on an rate-monotonic policy, while the synchronization task is always given the lowest priority. Then, the task set is sorted according to the task's priorities, all tasks' individual rt-kernel timers are created and task states are set to ready if their release offset is zero. Lastly, a counting semaphore is created which blocks the scheduler when no task is ready, allowing it to yield CPU time to other rt-kernel tasks when not needed.

To start the scheduler, the user has to spawn an rt-kernel task which runs the scheduler. The scheduler object as well as the task set and its tasks must persist in memory as long as the scheduler task is running. A user example of running a CAN scheduler with two CAN tasks is presented in Figure 4.3 and shows how CAN tasks and a CAN scheduler can be initialized and run.

In detail, the scheduler works as follows. At scheduler startup, a rough clock synchronization is run. Then, every node delays until its next tick occurs and executes a precise clock synchronization to fine-tune all nodes' clocks to the master node. This concludes the startup phase of the scheduler, which then starts all task timers and begins scheduling ready tasks. When no tasks are ready, the scheduler suspends until tasks are ready again.

When a task is released, its corresponding software timer triggers an interrupt. The associated interrupt handler, which is the same for all tasks, puts the task in the ready state and signals the scheduler semaphore, causing the scheduler to acquire it. It also asserts that the task is in the waiting state, since a task which is not in the waiting state during its next release has missed its deadline. These time-triggered interrupts are of a higher priority than the scheduler, meaning all tasks that should have been released are in a ready state when the scheduler task resumes execution.

When the scheduler is signaled, it iterates over the sorted task set to find the next ready task. It chooses the first ready task found, sets its state to running, and runs its callback function. When finished, the task's state is set to waiting until its next release, and the scheduler stops its iteration over all tasks and tries to acquire the semaphore again. As such, the worst case delay occurs when all tasks in the system are ready simultaneously. Assuming there are n tasks in total, the scheduler runs in $\mathcal{O}(n^2)$ in the worst case.

```
sched_t *sched_obj;
network_t *net_obj;

void callback_1(void *arg) { return; }
void callback_2(void *arg) { return; }

int main (void)
{
  /* Initialize a network object and generate unique node IDs. */
  net_obj = can_init_network();

  /* Create a task set with two tasks */
  size_t size = 2;
  can_task_t *task_set[size];

  /* Create two tasks with a period, offset, handler and a handler
   argument. */
  task_set[0] = can_task_create(tick_from_ms(20), 0, callback_1, NULL);
  task_set[1] = can_task_create(tick_from_ms(20), tick_from_ms(5),
   callback_2, &net_obj);

  /* Create a scheduler object using the initialized network and the
   created task set. */
  sched_obj = init_scheduler(net_obj, task_set, size, 1);

  /* Start the scheduler by spawning an rt-kernel task. */
  task_spawn("Scheduler task", scheduler, SCHED_PRIO, SCHED_STACK_SIZE,
    sched_obj);

  /* Suspend main indefinitely. */
  task_stop();
  return 0;
}
```

Figure 4.3: An example usage of the software library. A scheduler object is created containing two tasks, which is then spawned as an rt-kernel task.

4.5 LPW algorithm

To execute the algorithm in the system its functionality is implemented using the flowchart as given in Chapter 3. It is implemented using a single correct value and not intervals of correct values, and assumes that node IDs are unforgeable.

For LPW to work properly, no more than one message must be transmitted in any single round. As such, multiple nodes must be able to attempt proposal simultaneously with only one node succeeding in actually transmitting on the CAN bus. When nodes attempt to send messages the messages end up in CAN mailboxes, handled by hardware, which transmit a message as soon as the CAN bus is idle. For a waiting message to stop requesting transmission, an abort request can be sent to the hardware after a short period of time, less than the shortest possible transmission time. An abort request can only succeed if the message is not currently transmitting,

meaning all messages waiting for an idle bus will be aborted. A system flag indicates whether or not the abort request succeeded, thereby any node can detect if it is the transmitter, in which case its propose flag is set and its message transmitted.

LPW uses synchronized rounds implemented using a hardware timer, which every node uses to busy-wait until the next round. A round in the program consists of two stages, first a proposing stage followed by an executing stage. In the proposing stage a node can propose if it disagreed with the last proposed value and will attempt to transmit its value on the CAN bus, with its value in the data-field of the message. The exception is the first round, in which the designated sending node (see Section 4.2.1) always proposes. A node agreeing with the previously proposed value or attempting a proposal but aborts, listens for a CAN message until the proposing stage ends. This stage is bounded to 1382 µs, which accounts for the transmission time of the largest possible CAN frame, processing time of said frame, slight drift in the nodes' clocks as well as possible interrupts being triggered on rt-kernel ticks. The reason we always account for the largest frame size is the fact that even if the message is actually shorter, a faulty node may attempt to send an 8-byte frame, which could affect the synchronicity if the round is shorter.

In the executing stage a node has either transmitted a message, received a message, or nothing has happened. In the first case the node does nothing and waits before continuing on to the next round. In the second case the node saves the last seen value and then waits. In the last case nothing has happened, meaning no node disagreed in the previous stage and this round is silent. This always happens on all nodes in the same round. The timeout for this stage is set to 5 µs to make time for the calculations. If it is a silent round or all nodes have proposed, the algorithm terminates, otherwise it continues to the next round. In total, each round in the algorithm lasts for 1387 µs.

5

Results

This chapter presents the results obtained when testing the capabilities of the software library. First, we measured the precision of the clock synchronization for different synchronization intervals. Next, we computed the overhead incurred from the distributed CAN scheduler and show how it scales with the amount of tasks. Last, we tested the bus utilization when using LPW with different amounts of faults compared to when no fault tolerance algorithm is used.

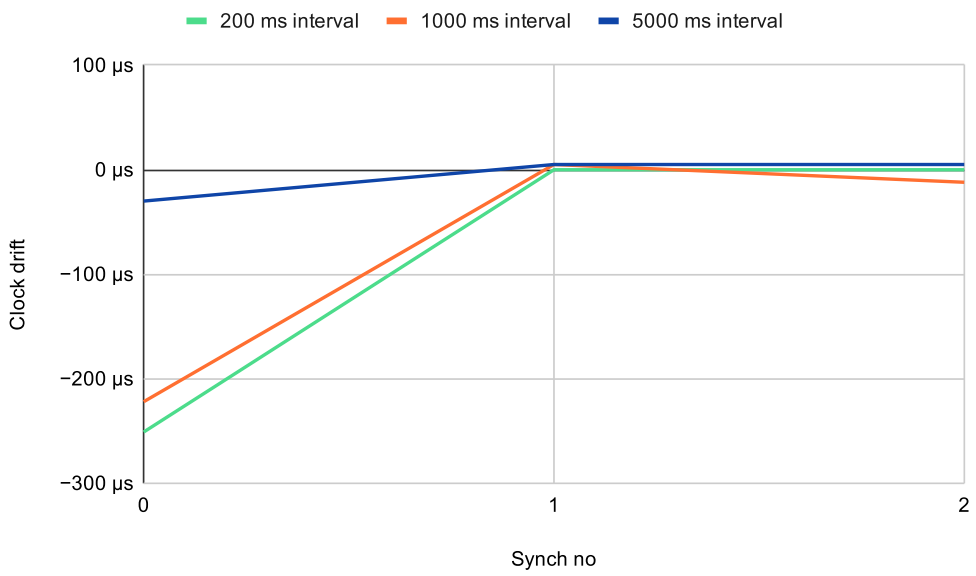
5.1 Precision of clock synchronization

The capabilities of the clock synchronization method are tested using a small system with one master node and two slave nodes. The reason we used two different slave nodes was to highlight how the drift can differ for different clocks. All nodes were assigned IDs beforehand which were not changed between different test cases in order to obtain consistent results.

The test data was produced by each slave node printing its drift on the serial port after every synchronization event. The tests were performed with three different intervals between the synchronizations, to measure how the time between synchronizations affects the precision. A drift of 0 indicates perfect synchronization with the master clock. The tested synchronization intervals were 200 ms, 1000 ms and 5000 ms. For each interval, 70 synchronizations were run.

The results are found in Figure 5.1 and Figure 5.2 and depict the measured clock drift in each synchronization. The drift during the first two synchronizations for both slaves is found in Figure 5.1 and shows the large drift the clocks have when they have not been synchronized before. The rest of the synchronization events are in Figure 5.2. Note that for the smallest synchronization interval the nodes' clocks are either perfectly synchronized, or they are 8 µs or -9 µs in comparison to the master clock. For larger synchronization intervals, these *spikes* are at most 14 µs or -12 µs in 5.2a, while in 5.2b the spikes are 5 µs or -13 µs respectively.

(a) Clock drift of slave 1.



(b) Clock drift of slave 2.

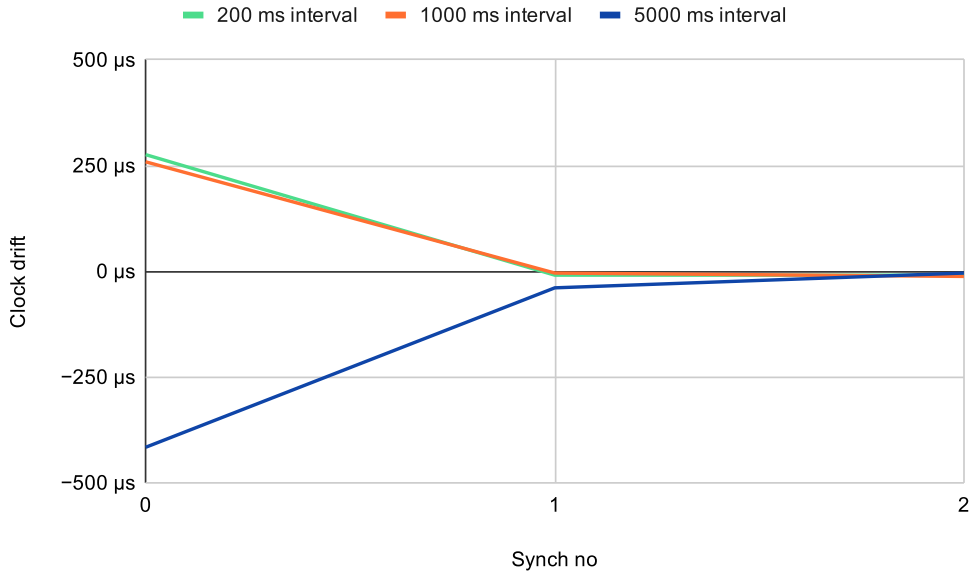
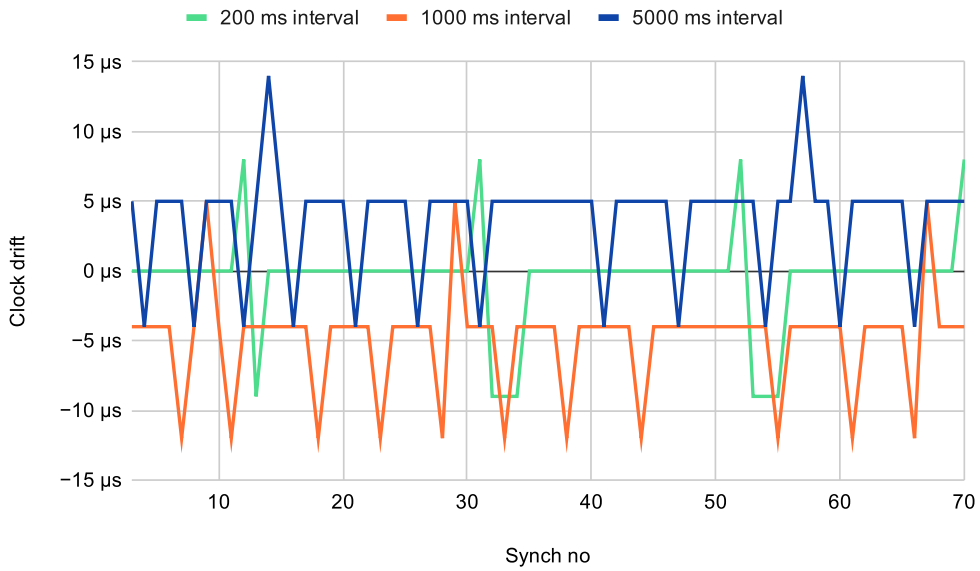


Figure 5.1: Clock drift of two different nodes in the first three synchronizations.

(a) Clock drift of slave 1.



(b) Clock drift of slave 2.

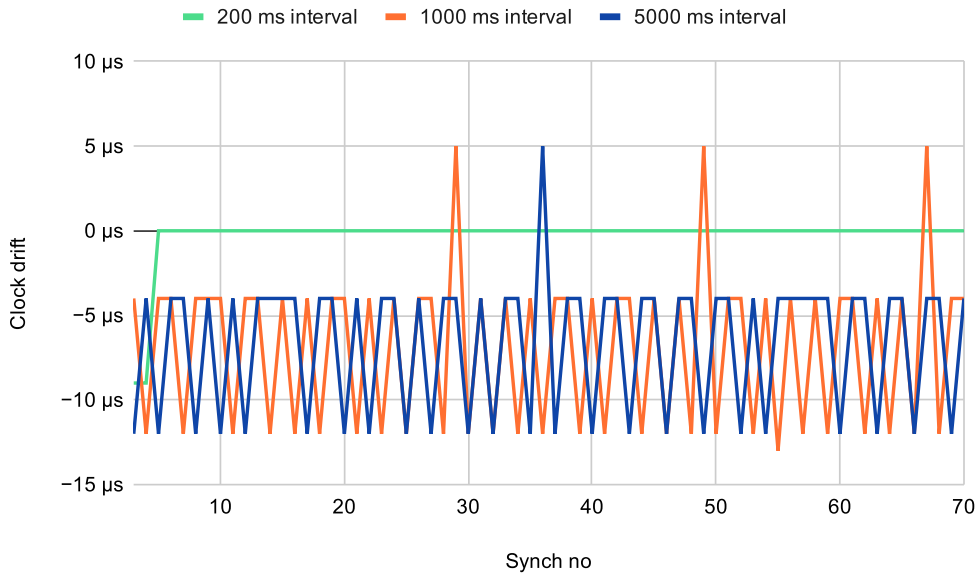


Figure 5.2: Clock drift of two different nodes excluding the first two synchronizations.

5.2 Scheduling delay

We tested the worst-case task delay incurred by the implemented scheduler in order to see how it might affect the user-defined task set. The tests were performed using task sets of sizes {10, 20, ..., 60}. All tasks in the task set had a period of 1 s and no release offset. We measured the delay using a hardware timer with microsecond precision, calculating the elapsed time between the release of the highest-priority task and the invocation of every task's callback. All tasks had the same callback which was an empty function.

We also performed a test where we added an offset of 0 ms for task 0, 1 ms for task 1 and so on, up to 59 ms for task number 59. The reason for this was to show how the task delay is affected when applying a small modification to the task set while keeping the same periodicity of all tasks. We slightly changed the method of measurement for this test to account for the change in the task set. Hence, we instead measure the delay by calculating the elapsed time between every task's release and its corresponding invocation.

The resulting task delay for all tests is found in Figure 5.3. We see that when multiple tasks are released simultaneously, the task delay drastically increases and grows for lower-priority tasks. In contrast, when delaying the first release of the tasks using offsets, the delay is small and increases only slightly for tasks with lower priority.

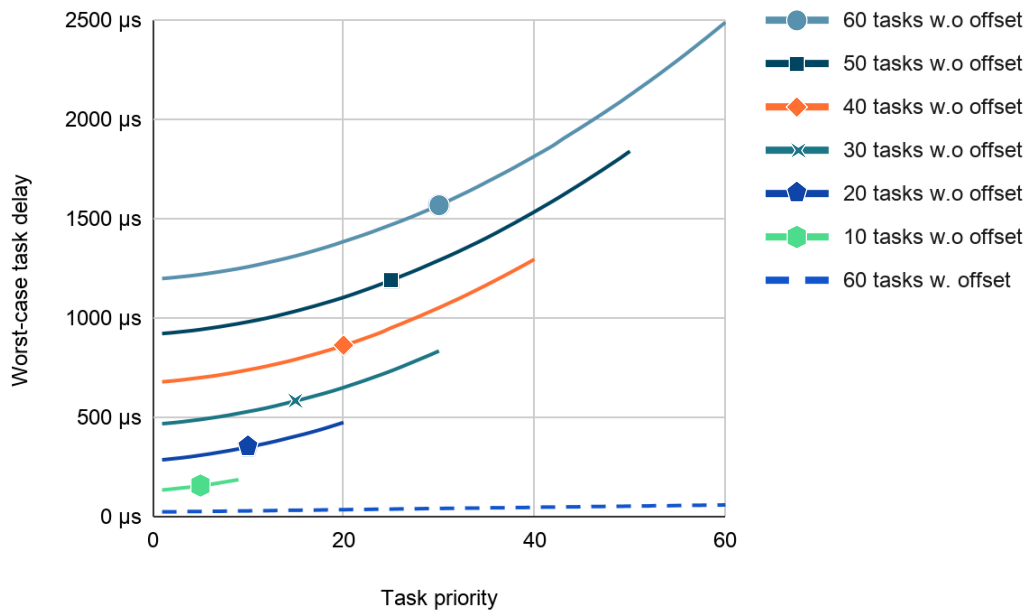


Figure 5.3: Worst-case task delay for different task set sizes.

5.3 Bus utilization of LPW

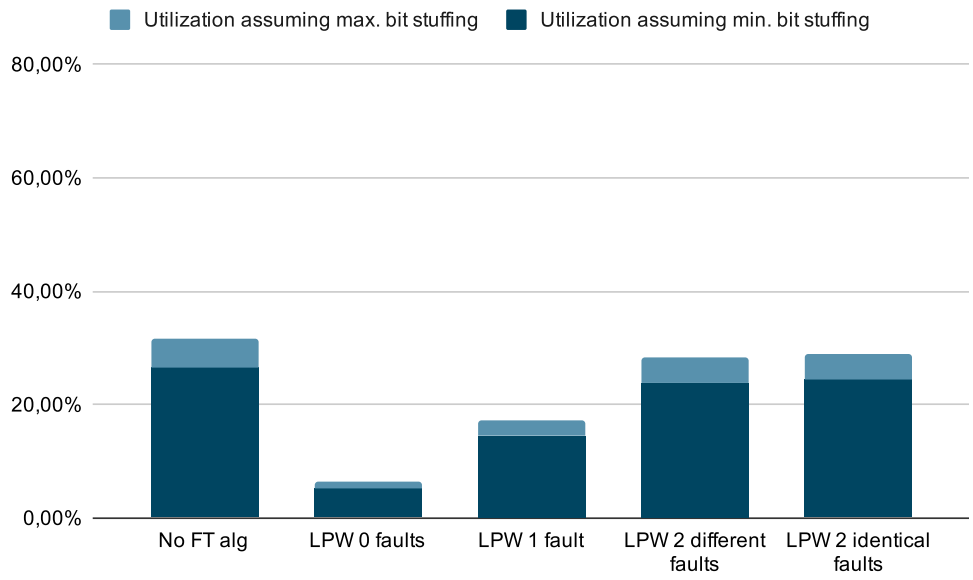
The following tests show the CAN bus utilization achieved when using LPW compared to when not using any fault tolerance algorithm. The test system consists of five nodes which run one task and clock synchronization once per hyperperiod. The task has a period of 1 s originally and 5 s when running LPW, since all five nodes need to be able to communicate their value in the worst case. The task's callback method runs `communicate()` 60 times before terminating. Since the bus utilization depends on how large the CAN messages are, we performed two tests, one with 1 byte data and one with 8 byte data. Due to the fact that the transmission time of a CAN frame differs depending on the amount of bit stuffing, we cannot determine the utilization exactly. Consequently, we calculated the utilization twice, once assuming minimum bit stuffing and once assuming maximum bit stuffing.

The task has a period of 1 s when not running any fault tolerance algorithm and allows for only one communication round. On the other hand, when running LPW all nodes in the network need to communicate on the bus, hence the task period is increased accordingly.

We derive the test results from data obtained during one hyperperiod, which is 1 s for the communication without LPW and 5 s with LPW. We retrieve the elapsed test time using the timestamp of the startup synchronization and the timestamp of the received message after the last synchronization. We calculate the cumulative transmission time of all messages and divide this with the total elapsed time to obtain the utilization.

The results can be viewed in Figure 5.4. The utilization is displayed with a margin of error for the unknown factor of the bit stuffing. The utilization is at its highest when running without any fault tolerance for both 1-byte and 8-byte messages. When using LPW, the utilization increases with the amount of faults. The maximum bus utilization of LPW is reached with maximum amounts of tolerated faults when the faulty values are identical, a slight increase in utilization compared to when running with different faulty values. The utilization is always higher for 8-byte messages than 1-byte messages.

(a) LPW bus utilization when sending 1-byte messages.



(b) LPW bus utilization when sending 8-byte messages.

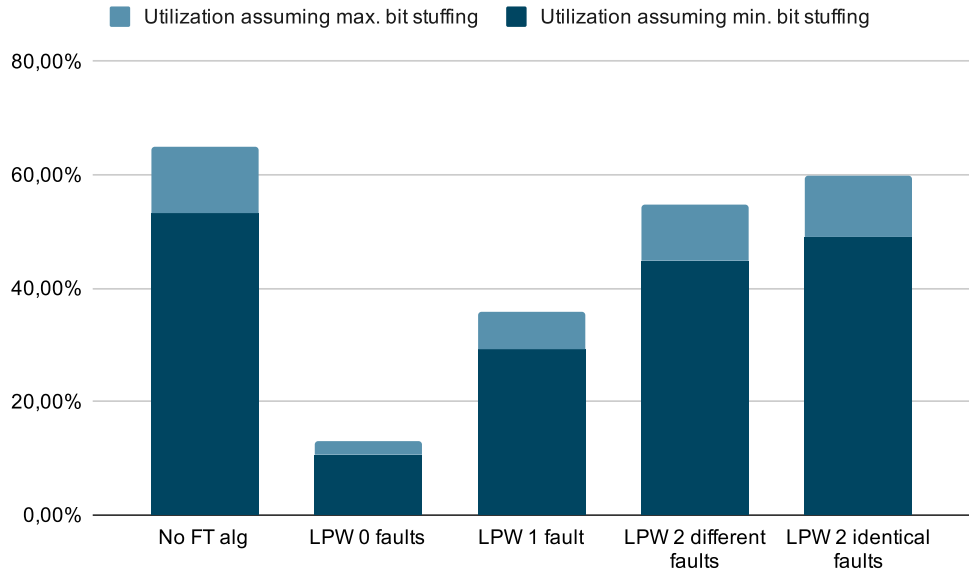


Figure 5.4: Bus utilization when running LPW with different data lengths.

6

Discussion

This chapter discusses the LPW algorithm presented in Chapter 3, the implementation details of the software library detailed in Chapter 4, as well as the results presented in Chapter 5.

6.1 Clock synchronization test results

The results in Figure 5.1 shows the drift in the first three synchronizations. These results highlight the fact that the clocks are unsynchronized at startup and that the rough synchronization is not enough to correct the drift, which is seen at synchronization 0 for both slaves.

Figure 5.2 shows the performance of the clock synchronization in terms of drift from the master clock for the remaining synchronizations. We see that for the shortest synchronization interval, both slaves are very synchronized with the master. With longer intervals, the synchronization worsens which is in line with our expectations. Note that the spikes in the graphs are similar in that the drift either changes by around 8 µs or 17 µs. This is due to the fact that the CAN bus runs at 125 Kbit/s resulting in a bit time of $1/125000 = 8$ µs. The nodes can only drift whole bits, which is reflected in the measurements that show a drift of one or two bit times between synchronizations. The actual value obtained from the graphs however are not necessarily divisible by the bit time as this factor depends on the drift in entering the clock synchronization method for each node.

6.2 Clock synchronization implementation

While the proposed clock synchronization method manages to synchronize the clocks of the system nodes, it has two major drawbacks. The first is that in this synchronization, if a node attempting to receive a message can do so immediately, it cannot be sure whether or not the message was just received or if it has been in the receive buffers for a long time. Its calculated drift will be the processing time it takes to fetch a message from its CAN buffers, and it will compensate only for the difference between its fetch time and the expected transmission time. As such, the clock could theoretically be infinitely behind in time compared to the master. However, a node can easily notice a longer receive time, hence this problem does not exist

when a slave's clock is largely ahead of the master clock. The rough synchronization at scheduler startup reduces the likelihood that this happens, and unless the configured synchronization interval is large, non-faulty clocks should never be able to drift more than can be corrected, as the system is homogeneous including the clocks' oscillators.

The other drawback is that the current implementation is system dependent. If the user changes the bitrate on the CAN bus or wants to use a different architecture for their homogeneous system, they would have to update the expected transmission time of the synchronization message since it varies for different bitrates and architectures. An improvement could be to provide a method in the software library that can calculate the expected transmission time given a bitrate, transmission overhead and reception overhead.

One strength of this clock synchronization method is its full compatibility with rt-kernel's ticks. As a consequence, tasks created within rt-kernel that are not part of the software library are synchronized between nodes, assuming they have the same periodicity and behavior.

Compared to other clock synchronization methods, our solution's main advantage is its message complexity. It requires only one CAN frame with no data field regardless of the system's size. It utilizes the bus less than the clock synchronization methods described in [20], [21], [23] while achieving similar precision. Common for [20], [21], [23] is that they all are usable in heterogeneous systems unlike our method, which assumes a homogeneous system. Our method also lacks fault tolerance, which both [21] and [23] provide.

The lack of fault tolerance combined with the limited drift correction makes our synchronization method unsuitable for safety-critical and highly reliable systems. For example, a node that experiences a fault which leads to its clock falling 4 ms behind the master's clock would need at least ten synchronization events to return to a synchronized state, which could be a long time depending on the configured synchronization interval. Consequently, if a user-defined task set has a hyperperiod of 200 ms and the user configures the scheduler to synchronize once every hyperperiod, it would take two whole seconds for the node to resynchronize, which could lead to system failure. Similarly, if the clock synchronization message is delayed or omitted because of a fault in the master node, the slaves could synchronize to an incorrect transmission time or enter a deadlock in case of omission. Therefore, for applications that require fault-tolerant synchronization, the current synchronization method must be modified or replaced.

6.3 Scheduler test results

The result in Figure 5.3 shows that the delay incurred at the critical instant increases with the amount of tasks in the system. The reason for this is that every task's timer triggers an interrupt at the critical instant. These preempt the scheduler as they have a higher priority in rt-kernel. We also see that the task delay grows quadratically

with the amount of tasks, since the scheduler runs in $\mathcal{O}(n^2)$ time.

In contrast, when modifying the tasks' release times using offsets, there is no critical instant as seen in Figure 5.3. The interrupts which preempt the scheduler do not occur simultaneously, but are instead spread out. The task delay is therefore kept very small even for the lowest priority task. Keep in mind that this also means that tasks are released later, and this modification is therefore not necessarily possible for all kinds of tasks. Even though the scheduler runs in $\mathcal{O}(n^2)$, the growth is only linear in this case. The reason is that since no two tasks are released simultaneously, the scheduler performs only one search for ready tasks, a $\mathcal{O}(n)$ operation. This gives a measure of the overhead incurred solely by the scheduler algorithm, as only one task is ready at a time.

Although offsets seem to reduce the task delay greatly, there are some aspects one has to consider. The execution order of ready tasks with the same periodicity is unpredictable since the scheduler does not differentiate between tasks with different offsets. Additionally, tasks with offsets will still release at some point which might interrupt the callback of another task. Using offsets is a design choice the user has to make.

To further minimize the task delay, the user can employ other tactics as well. For example, one can combine the callbacks of functions with the same periods and offsets, in order to avoid a situation similar to the critical instant where many tasks are released simultaneously. It is also possible to join tasks' callbacks if there are precedence constraints and they have the same periods, thereby enforcing the order of execution and getting around the fact that the scheduler has no support for precedence constraints.

6.4 Scheduler implementation

One could design a $\mathcal{O}(n)$ scheduler using a priority queue (PQ) to potentially minimize the task delay further. However, such a PQ scheduler is not always better, since inserting a task in the priority queue is a $\mathcal{O}(n)$ operation which executes in every task's release interrupt. Compare this to our scheduler which only has $\mathcal{O}(1)$ interrupt execution time. As n insertions are required in the critical instant, the worst-case interrupt delay for a PQ scheduler is $\mathcal{O}(n^2)$. On the other hand, the delay incurred by a PQ scheduler would increase linearly with the amount of tasks as retrieving the highest-priority ready task is a $\mathcal{O}(1)$ operation. At some point, the two schedulers' worst-case delays would meet and after that point, the PQ scheduler would perform better than our scheduler. Since we have not implemented and tested a PQ scheduler, we do not know after how many tasks this turning point occurs. In any case, it might be beneficial to allow the user to choose between a PQ scheduler and our current scheduler, depending on what is best for their task set.

This scheduler assigns strictly rate-monotonic priorities which cannot be overridden, making it impossible to use in mixed-criticality systems. However, it can be easily modified for this purpose by allowing creation of scheduler objects containing

multiple task sets with user-defined priorities. The scheduler could then prioritize ready tasks from higher-priority task sets over those from lower-priority ones.

6.5 LPW test results

The results of the LPW tests in Figure 5.4 show a notable utilization difference between various fault scenarios. The LPW utilization increases with the amount of faults, an expected behavior as the round count increases. In the test with no faults, the correct value is always proposed first without further communication, thus only using 1/5 tolerated communication rounds on the bus. The test with one fault uses 3/5 slots when a correct node initializes LPW, but every fifth instance it uses only 2/5 slots, since when the designated sender is a faulty node only one correct node needs to propose. In the cases with two faults, more synchronous rounds are required which is reflected in the larger utilization. No instance of LPW reaches the bus utilization achieved when communicating without fault tolerance, due to the fact that LPW requires more time in each synchronous round.

The utilization is slightly higher with two identical faults in comparison to two different faults. When the faulty nodes have identical faults, a faulty proposal will always be followed by a correct one. On the other hand, nodes with different faults disagree with not only the correct nodes, but also each other. Therefore, they sometimes propose after each other, meaning fewer correct nodes need to propose in total. In Figures 5.4a and 5.4b, we see that the difference in utilization between two identical and two different faults is larger for Figure 5.4b. This is not due to the size of the message, but the unpredictability of proposal orders which affects the round count of each instance of LPW.

When comparing the different data length tests with each other, the utilization obtained from the messages with a longer data field is approximately doubled. This is an expected result as the message grows from a transmission time of at most 522 µs to at most 1082 µs, which is approximately a doubled transmission time.

6.6 LPW implementation

LPW is designed for systems where faults are not expected to happen in every iteration, but occur with a small probability. Because of this, it is designed to be most efficient in the fault-free case and has an early-stopping approach. As described in Section 2.5.1, a trivial solution for incorporating Byzantine fault tolerance in a broadcast system is by each node proposing its value and all nodes choosing the value transmitted by a majority of nodes. This algorithm will henceforward be called Trivial Byzantine (TB). Yet, TB has a round complexity of $\mathcal{O}(n)$ regardless of how many faults there are, whereas LPW only has a round complexity of 2 in the average fault-free case, and $\mathcal{O}(n)$ in the worst case. LPW is therefore much more efficient during normal operation.

A downside of LPW is that it does not store any information about faulty nodes.

This information is important since it would allow a user to identify faulty nodes and act accordingly. Consequently, a user could define a maximum fault rate allowed in a time interval and any node exceeding this could be indicated to the user. However, crashed nodes or nodes omitting messages cannot be noticed in every instance of LPW, but only when they are the initial senders. As such, the fault rate of such a node would not necessarily reflect reality. Compare this to TB, in which all nodes always send their value. Every faulty node would be known in every instance, providing better information of the fault rate in the system. Conclusively, it comes down to a trade-off between a low communication overhead and better fault-detection capabilities.

Even though LPW is efficient in the fault-free case, a full operation with n rounds must always be allowed to run. Since LPW in most cases will not use all rounds, the bus is left with low utilization and a scheduler with much idle time. We believe an improvement for this software library is to enable a user to create soft real-time tasks that are scheduled whenever a hard real-time task does not use all communication rounds. This would lead to a higher bus utilization when faults are few or non-existing.

6.7 LPW adaptation

There are cases where the real-world adaptation of LPW presented in Section 3.4 results in an incorrect decision. These faults can occur when faulty nodes are within $\pm 3\sigma$ from the ideal value. As the example calculation 3.7 shows, the probability that this occurs is small but not negligible. Additionally, it is not solely this factor which affects the final outcome, as it depends on the order of proposal and the values of correct nodes as well. For example, if the correct values follow a uniform distribution within σ from the ideal value, the probability that one node acquires a value which is exactly σ from the ideal value is greater than in the case where correct values follow a normal distribution.

Further, LPW has to terminate with this faulty value, which depends on the order of proposals. As in the example in Figure 3.3, the initializing node is always the last proposer as all nodes agree with each other, meaning only 1/3 of LPW instances end up faulty if this specific fault occurs. This in turn is affected by the network size where a larger network leads to fewer permutations terminating with a faulty value. The user can further reduce the probability that such a fault occurs, for example by sampling their value multiple times and calculating a mean before communicating the value.

The fact that the real-world adaptation of LPW can result in values outside of the tolerated range is important to keep in mind. Its main advantage is its speed and communication efficiency, but it sacrifices precision as a result. Luckily, the resulting error margin is bounded to 3σ and as such, LPW might be a viable choice in real-time systems where this precision is good enough, as it provides fault tolerance at a small overhead.

7

Conclusion

We have achieved our goal of implementing a software library for use in distributed embedded control systems (DECS). The presented solution provides a lightweight clock synchronization that while being more efficient than other solutions in the field, has a few major drawbacks including a lack of fault tolerance. The NPRM scheduler implementation supports hard real-time tasks but has no way of handling mixed-criticality task sets. Further, it incurs small delays in general, but delays the highest priority tasks significantly when many tasks are released simultaneously. Regarding the proposed algorithm LPW, it tolerates faults in the value domain efficiently in the average case and only uses all allocated time when the maximum amount of faults are present, making it a lightweight fault tolerance solution.

Future work could look into multiple possibilities. Fault tolerance could be implemented in both the scheduler and the clock synchronization, in order to make the library more robust. One could also examine different ways of guaranteeing fault tolerance in the time domain. Additionally, one could investigate how to extend this library to support tasks of mixed criticality and tasks where the restriction that deadlines are equal to periods is relaxed.

In conclusion, work remains to be done before the library can be used in production systems. The essential improvements are a fault-tolerant clock synchronization and fault handling in the time domain. The library provides a distribution layer designed with real-world systems in mind, more specifically DECS. It is both lightweight and efficient, making it especially useful in resource-constrained systems.

Bibliography

[1] H. Aysan, R. Dobrin, and S. Punnekkat, "Fault tolerant scheduling on controller area network (CAN)", in *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, May 2010, pp. 226–232. DOI: 10.1109/ISORCW.2010.32.

[2] R. Ramezani and Y. Sedaghat, "An overview of fault tolerance techniques for real-time operating systems", in *ICCKE 2013*, Oct. 2013, pp. 1–6. DOI: 10.1109/ICCKE.2013.6739552.

[3] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg, "Byzantine fault tolerance, from theory to reality", in *Computer Safety, Reliability, and Security*, S. Anderson, M. Felici, and B. Littlewood, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 235–248, ISBN: 978-3-540-39878-3.

[4] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed fault-tolerant real-time systems: The mars approach", *IEEE Micro*, vol. 9, no. 1, pp. 25–40, Feb. 1989. DOI: 10.1109/40.16792.

[5] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem", *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982, ISSN: 0164-0925. DOI: 10.1145/357172.357176. [Online]. Available: https://doi-org.proxy.lib.chalmers.se/10.1145/357172.357176.

[6] rt-labs. (2020). rt-kernel: User & Reference Manual, [Online]. Available: https://rt-labs.com/refman/rt-kernel/ (visited on 02/07/2020).

[7] M. Livani, J. Kaiser, and W. Jia, "Scheduling hard and soft real-time communication in the controller area network (CAN)", *IFAC Proceedings Volumes*, vol. 31, no. 14, pp. 13–18, 1998, 23rd IFAC/IFIP Workshop on Real Time Programming 1998 (WRTP '98)., Shantou, China, 23-25 June, ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)44865-8. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1474667017448658.

[8] J. Rufino, P. Verissimo, G. Arroz, C. Almeida, and L. Rodrigues, "Fault-tolerant broadcasts in CAN", in *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No.98CB36224)*, Jun. 1998, pp. 150–159. DOI: 10.1109/FTCS.1998.689464.

[9] "CAN Specification 2.0", Robert Bosch GmbH, Postfach 50, D-7000 Stuttgart 1, Standard, Sep. 1991. [Online]. Available: http://esd.cs.ucr.edu/webres/can20.pdf (visited on 02/07/2020).

[10] G. I. Mary, Z. C. Alex, and L. Jenkins, "Response time analysis of messages in controller area network: A review", *Journal of Computer Networks and Communications*, vol. 2013, 2013.

[11] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment", *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[12] N. C. Audsley, A. Burns, M. Richardson, and A. Wellings, "Deadline monotonic scheduling", 1990.

[13] R. I. Davis, L. Cucu-Grosjean, M. Bertogna, and A. Burns, "A review of priority assignment in real-time systems", *Journal of Systems Architecture*, vol. 65, pp. 64–82, 2016, ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2016.04.002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762116300200.

[14] N. C. Audsley, "Optimal priority assignment and feasibility of static priority tasks with arbitrary start times", Department of Computer Science, University of York, Tech. Rep. YCS-164, 1991.

[15] M. Park, "Non-preemptive fixed priority scheduling of hard real-time periodic tasks", May 2007, pp. 881–888. DOI: 10.1007/978-3-540-72590-9_134.

[16] G. F. Coulouris, *Distributed systems: concepts and design*. Ser. International computer science series. Addison-Wesley / Pearson Education Ltd., 2012, ISBN: 1447930177.

[17] M. Lévesque and D. Tipper, "A survey of clock synchronization over packet-switched networks", *IEEE Communications Surveys Tutorials*, vol. 18, no. 4, pp. 2926–2947, 2016.

[18] B. Simons, "An overview of clock synchronization", in *Fault-Tolerant Distributed Computing*, B. Simons and A. Spector, Eds., New York, NY: Springer New York, 1990, pp. 84–96, ISBN: 978-0-387-34812-4.

[19] H. Kopetz and W. Ochsenreiter, "Clock synchronization in distributed real-time systems", *IEEE Transactions on Computers*, vol. C-36, no. 8, pp. 933–940, Aug. 1987, ISSN: 2326-3814. DOI: 10.1109/TC.1987.5009516.

[20] M. Gergeleit and H. Streich, "Implementing a distributed high-resolution real-time clock using the CAN-bus", 1994.

[21] D. Lee and J. Allan, "Fault-tolerant clock synchronisation with microsecond-precision for CAN networked systems", 2003.

[22] L.-B. Fredriksson, "Method and arrangement for correlating time bases between interconnected units", United States Patent 8065052, Nov. 2011.

[23] L. Rodrigues, M. Guimaraes, and J. Rufino, "Fault-tolerant clock synchronization in CAN", in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, Dec. 1998, pp. 420–429. DOI: 10.1109/REAL.1998.739775.

[24] M. Raynal, "Consensus in synchronous systems: A concise guided tour", *2002 Pacific Rim International Symposium on Dependable Computing, 2002. Proceedings.*, pp. 221–228, 2002.

[25] T. Henzinger and C. Kirsch, *Embedded Software: First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October 8-10, 2001. Proceedings*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, ISBN: 9783540426738. [Online]. Available: https://books.google.se/books?id=lpuGNxjSCl4C.

[26] M. Castro and B. Liskov, "Practical byzantine fault tolerance", in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99, New Orleans, Louisiana, USA: USENIX Association, 1999, pp. 173–186, ISBN: 1880446391.

[27] L. Ren, K. Nayak, I. Abraham, and S. Devadas, "Practical synchronous byzantine consensus", *IACR Cryptology ePrint Archive*, vol. 2017, p. 307, 2017.

[28] A. Groce, J. Katz, A. Thiruvengadam, and V. Zikas, "Byzantine agreement with a rational adversary", in *Automata, Languages, and Programming*, A. Czumaj, K. Mehlhorn, A. Pitts, and R. Wattenhofer, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 561–572, ISBN: 978-3-642-31585-5.

[29] M. Correia, G. Veronese, N. Neves, and P. Veríssimo, "Byzantine consensus in asynchronous message-passing systems: A survey", *IJCCBS*, vol. 2, pp. 141–161, Jan. 2011. DOI: 10.1504/IJCCBS.2011.041257.

[30] D. Dolev and H. Strong, "Authenticated algorithms for byzantine agreement", *SIAM J. Comput.*, vol. 12, pp. 656–666, Nov. 1983. DOI: 10.1137/0212045.

[31] A. Castañeda, Y. Moses, M. Raynal, and M. Roy, "Early decision and stopping in synchronous consensus: A predicate-based guided tour", May 2017, pp. 206–221. DOI: 10.1007/978-3-319-59647-1_16.

[32] A. Castañeda, Y. Gonczarowski, and Y. Moses, "Unbeatable consensus", Oct. 2014. DOI: 10.1007/978-3-662-45174-8_7.

[33] D. Dolev, R. Reischuk, and H. R. Strong, "Early stopping in byzantine agreement", *J. ACM*, vol. 37, no. 4, pp. 720–741, Oct. 1990, ISSN: 0004-5411. DOI: 10.1145/96559.96565. [Online]. Available: https://doi-org.proxy.lib.chalmers.se/10.1145/96559.96565.

[34] P. R. Parvédy and M. Raynal, "Optimal early stopping uniform consensus in synchronous systems with process omission failures", in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '04, Barcelona, Spain: Association for Computing Machinery, 2004, pp. 302–310, ISBN: 1581138407. DOI: 10.1145/1007912.1007963. [Online]. Available: https://doi.org/10.1145/1007912.1007963.

[35] J. A. Garay and Y. Moses, "Fully polynomial byzantine agreement for n > 3t processors in t + 1 rounds", *SIAM Journal on Computing*, vol. 27, no. 1, pp. 247–290, 1998. DOI: 10.1137/S0097539794265232. eprint: https://doi.org/10.1137/S0097539794265232. [Online]. Available: https://doi.org/10.1137/S0097539794265232.

[36] L.-B. Fredriksson, "CAN for critical embedded automotive networks", *IEEE Micro*, vol. 22, no. 4, pp. 28–35, Jul. 2002, ISSN: 1937-4143. DOI: 10.1109/MM.2002.1028473.

[37] E. Sultanik, *Mtwister*, 2014. [Online]. Available: https://github.com/ESultanik/mtwister (visited on 05/11/2020).

[38] *STM32F302xB/C/D/E and STM32F302x6/8 advanced ARM®-based 32-bit MCUs reference manual*, English, version 7.0, STMicroelectronics, Jan. 2017.

[39] V. van der Leest, E. van der Sluis, G.-J. Schrijen, P. Tuyls, and H. Handschuh, "Efficient implementation of true random number generator based on SRAM PUFs", in *Cryptography and Security: From Theory to Applications: Essays Dedicated to Jean-Jacques Quisquater on the Occasion of His 65th Birthday*, D. Naccache, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 300–318, ISBN: 978-3-642-28368-0. DOI: 10.1007/978-3-642-28368-0_20. [Online]. Available: https://doi.org/10.1007/978-3-642-28368-0_20.

[40] A. Van Herrewege, V. van der Leest, A. Schaller, S. Katzenbeisser, and I. Verbauwhede, "Secure PRNG seeding on commercial off-the-shelf microcontrollers", in *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*, ser. TrustED '13, Berlin, Germany: Association for Computing Machinery, 2013, pp. 55–64, ISBN: 9781450324861. DOI: 10.1145/2517300.2517306. [Online]. Available: https://doi.org/10.1145/2517300.2517306.

A

Software library documentation

1 Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

2 File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

3 Data Structure Documentation

3.1 can_data_t Struct Reference

A struct which can include both the data field of a CAN frame and its length.

`#include <schedule.h>`

3.1.1 Detailed Description

A struct which can include both the data field of a CAN frame and its length.

The documentation for this struct was generated from the following file:

- schedule.h

3.2 can_split_id_t Struct Reference

A struct which allows a user to access the node and message ID parts of a CAN ID field.

`#include <can_helper.h>`

3.2.1 Detailed Description

A struct which allows a user to access the node and message ID parts of a CAN ID field.

The documentation for this struct was generated from the following file:

- can_helper.h

4 File Documentation

4.1 can_helper.h File Reference

This file contains functionality for CAN communication.

`#include <can/can.h>`

Data Structures

- struct can_split_id_t

  A struct which allows a user to access the node and message ID parts of a CAN ID field.

Macros

- #define ID_SIZE 11

  CAN ID field size in bits.
- #define NODE_ID_SIZE 5

  Amount of bits in CAN ID field that is the node ID. Uses the least significant bits in a CAN ID.
- #define NODE_ID_BUILD_MASK 0xFFFFFFE0

  Mask used to verify that a node ID is within range.
- #define MSG_ID_BUILD_MASK 0xFFFFF81F

  Mask used to verify that a message ID is within range.
- #define NODE_ID_SPLIT_MASK 0x001F

  Mask used to acquire a node ID from a CAN ID.
- #define MSG_ID_SPLIT_MASK 0x07E0

  Mask used to acquire a message ID from a CAN ID.
- #define MAX_NODES 20

  The maximum amount of nodes allowed in the system.
- #define ID_GEN_TIMEOUT 1000

  The amount of time (in ms) that each node waits before concluding that there are no more nodes to detect at startup.
- #define CAN_BITRATE 125000

  Bitrate of CAN bus specified in bits/second.
- #define CAN_BIT_SEG_1 7

  Bit segment 1. See CAN specification for details.
- #define CAN_BIT_SEG_2 8

  Bit segment 2. See CAN specification for details.
- #define CAN_SJW 4

  Resynchronization jump width. See CAN specification for details.

Typedefs

- typedef struct network network_t

  A struct which keeps the network state.

Functions

- can_split_id_t can_split_id (can_id_t id)

  Splits an 11-bit CAN ID into a node ID and a message ID.
- can_id_t can_build_id (uint32_t msg_id, uint32_t node_id)

  Builds a can_id_t from a message ID and a node ID.
- uint8_t get_my_id (network_t *net)

  Returns a node's own ID in a network `net`.
- uint8_t get_can_fd (network_t *net)

  Returns a node's CAN file descriptor.
- uint8_t get_network_size (network_t *net)

  Returns the number of nodes in the network.

- void generate_seed (void)

  Generate a seed for use with the PRNG.
- network_t * can_init_network (void)

  Allocates and initializes a network struct by spawning a can_id_gen() task.
- int can_write_message (int can_fd, can_id_t id, const uint8_t *msg, uint8_t dlc)

  Writes a message to the CAN bus.
- int can_abrq (void)

  Tries to abort pending CAN transmit requests.
- uint32_t can_wcet_us (uint8_t dlc)

  Calculates the worst case transmission time of a CAN-frame with a data field of length `dlc`.

4.1.1 Detailed Description

This file contains functionality for CAN communication.

4.1.2 Function Documentation

4.1.2.1 can_abrq() `int can_abrq ( void )`

Tries to abort pending CAN transmit requests.

Returns

0 if abort request succeeds, else 1.

4.1.2.2 can_build_id() `can_id_t can_build_id ( uint32_t msg_id, uint32_t node_id )`

Builds a can_id_t from a message ID and a node ID.

Parameters

msg_id	The desired message ID.
node_id	The desired node ID.

Returns

A can_id_t to be used with can_write_message().

4.1.2.3 can_init_network()

```
network_t* can_init_network (
        void  )
```

Allocates and initializes a network struct by spawning a can_id_gen() task.

Returns

Pointer to the created network_t.

4.1.2.4 can_split_id()

```
can_split_id_t can_split_id (
        can_id_t id )
```

Splits an 11-bit CAN ID into a node ID and a message ID.

Parameters

id	ID to split.

Returns

A can_split_id_t with a node ID and a message ID.

4.1.2.5 can_wcet_us()

```
uint32_t can_wcet_us (
        uint8_t dlc )
```

Calculates the worst case transmission time of a CAN-frame with a data field of length `dlc`.

Parameters

dlc	Length of the CAN frame's data field.

Returns

Time (in microseconds) it takes to transmit the frame in the worst case.

4.1.2.6 can_write_message()

```
int can_write_message (
        int can_fd,
        can_id_t id,
        const uint8_t * msg,
        uint8_t dlc )
```

Writes a message to the CAN bus.

Parameters

can↩ _fd	File descriptor of the CAN interface.
id	ID field value.
msg	The data field of the transmitted CAN frame.
dlc	The length of the CAN frame's data field in bytes.

Returns

ETIMEDOUT if no transmit buffers were available, 0 otherwise.

4.1.2.7 generate_seed() `void generate_seed ( void )`

Generate a seed for use with the PRNG.

Reads the SRAM address space, and divides it into four blocks, XORs all bytes in a block to obtain a final byte. Finally, the four obtained bytes are concatenated into a 32-bit seed that is written to a global variable seed. Should be called as early as possible after boot.

4.1.2.8 get_can_fd() `uint8_t get_can_fd ( network_t * net )`

Returns a node's CAN file descriptor.

Parameters

net	Pointer to network struct.

Returns

File descriptor number.

4.1.2.9 get_my_id() `uint8_t get_my_id ( network_t * net )`

Returns a node's own ID in a network `net`.

Parameters

net	Pointer to network struct.

Returns

ID value.

4.1.2.10 get_network_size() `uint8_t get_network_size (`

```
network_t * net )
```

Returns the number of nodes in the network.

Parameters

net	Pointer to network struct.

Returns

Number of nodes.

4.2 lpw.h File Reference

This file contains the implementation of the Last-Proposal-Wins algorithm.

Functions

- can_data_t run_lpw (sched_t *sched, can_data_t my_data)
  Runs the LPW algorithm.

4.2.1 Detailed Description

This file contains the implementation of the Last-Proposal-Wins algorithm.

4.2.2 Function Documentation

4.2.2.1 run_lpw() `can_data_t run_lpw (`

```
sched_t * sched,
can_data_t my_data )
```

Runs the LPW algorithm.

LPW is initiated with a sender node transmitting its value onto the bus. All nodes listen to this and if the first transmission fails, all nodes disagree and try to propose. If the first transmission succeeds all nodes retrieve this value and either agree or disagree with it. It runs in synchronized rounds and terminates whenever a silent round is reached or all nodes have proposed.

Parameters

sched	The scheduler operating on the bus.
my_data	This specific node's value and dlc. A value can at most be 8 bytes (one CAN frame).

Returns

A can_data_t containing the value decided on by all nodes and its length.

4.3 schedule.h File Reference

This file contains functionality for the scheduler and tasks.

```
#include <kern.h>
#include "can_helper.h"
```

Data Structures

- struct can_data_t

  A struct which can include both the data field of a CAN frame and its length.

Macros

- #define SCHED_PRIO 25

  The task priority of the scheduler.
- #define SCHED_STACK_SIZE 1024

  The necessary stack size of the scheduler task.

Typedefs

- typedef struct can_task can_task_t

  A struct defining a CAN task.
- typedef struct sched sched_t

  A struct defining a scheduler object.

Enumerations

- enum ft_alg { NONE, LPW }

  Types of fault tolerance algorithms.

Functions

- uint8_t get_sender_id (sched_t *sched)

  Returns the current sender ID.
- uint8_t get_msg_id (sched_t *sched)

  Returns the current message ID.
- network_t * get_network (sched_t *sched)

  Returns the network struct of a scheduler.
- can_data_t communicate (sched_t *sched, can_data_t can_data, ft_alg alg)

  Communicates a message on the CAN-bus using the fault-tolerance algorithm `alg`.
- can_task_t * can_task_create (tick_t period, tick_t offset, void(*callback)(void *), void *callback_arg)

  Initializes and allocates a can_task_t.
- sched_t * init_scheduler (network_t *net, can_task_t **tasks, size_t ts_size, uint8_t synch_interval)

  Initializes and allocates a scheduler object. Assigns priorites to the tasks according to a rate-monotonic priority assignment policy. Also creates a clock synchronization task using `synch_interval`.
- void scheduler (void *arg)

  Continuously runs the schedule. Should be run using an rt-kernel task with SCHED_PRIO and SCHED_STACK_SIZE as its input arguments.

4.3.1 Detailed Description

This file contains functionality for the scheduler and tasks.

4.3.2 Macro Definition Documentation

4.3.2.1 SCHED_PRIO `#define SCHED_PRIO 25`

The task priority of the scheduler.

This is the task priority used when creating a scheduler task in rt-kernel. It should have higher priority than any other user-defined rt-kernel tasks for the scheduler guarantees to hold.

4.3.3 Enumeration Type Documentation

4.3.3.1 ft_alg `enum ft_alg`

Types of fault tolerance algorithms.

Enumerator

NONE	No fault-tolerance algorithm.
LPW	Last-proposal-wins algorithm. A Byzantine fault-tolerance algorithm.

4.3.4 Function Documentation

4.3.4.1 can_task_create()

```
can_task_t* can_task_create (
        tick_t period,
        tick_t offset,
        void(*)(void *) callback,
        void * callback_arg )
```

Initializes and allocates a can_task_t.

The callback function `callback` must call communicate() in order to communicate its result to other nodes. The scheduler allocates a time slot on the CAN bus for the duration of `callback`. Therefore, it is recommended that `callback` is short.

Parameters

period	The period of the task specified in ticks.
offset	The offset of the first invocation of this task relative to when the scheduler is started. If 0, this task becomes ready when the scheduler task is started. The value is specified in ticks.
callback	Pointer to a callback function which runs when the task is run by the scheduler.
callback_arg	Input argument to the callback function. Use NULL for no argument.

Returns

Pointer to the created can_task_t.

4.3.4.2 communicate()

```
can_data_t communicate (
        sched_t * sched,
        can_data_t can_data,
        ft_alg alg )
```

Communicates a message on the CAN-bus using the fault-tolerance algorithm `alg`.

This function transmits a message onto the bus if the calling node is currently the sending node, otherwise it awaits reception of a message. The algorithm LPW will begin with `can_data` as its value if `alg` == LPW.

Parameters

sched	Pointer to scheduler struct.
can_data	can_data_t to be transmitted.
alg	The type of fault-tolerance algorithm to communicate with.

Returns

If `alg` == NONE and the calling node is also the sending node, `can_data` is returned. If `alg` == NONE and the node is not the sender, the received message's data is returned. If a fault-tolerance algorithm is used, the function returns the value decided by the algorithm.

4.3.4.3 get_msg_id() uint8_t get_msg_id (sched_t * sched)

Returns the current message ID.

Parameters

sched	Pointer to scheduler struct.

Returns

Message ID.

4.3.4.4 get_network() network_t* get_network (sched_t * sched)

Returns the network struct of a scheduler.

Parameters

sched	Pointer to scheduler struct.

Returns

Pointer to network struct.

4.3.4.5 get_sender_id() uint8_t get_sender_id (sched_t * sched)

Returns the current sender ID.

Parameters

sched	Pointer to scheduler struct.

Returns

Sender ID.

4.3.4.6 init_scheduler()

```
sched_t* init_scheduler (
        network_t * net,
        can_task_t ** tasks,
        size_t ts_size,
        uint8_t synch_interval )
```

Initializes and allocates a scheduler object. Assigns priorites to the tasks according to a rate-monotonic priority assignment policy. Also creates a clock synchronization task using `synch_interval`.

Parameters

net	Pointer to the network_t the scheduler will use.
tasks	Pointer to an array of tasks the scheduler will schedule.
ts_size	The amount of tasks in the task array.
synch_interval	The amount of hyperperiods between each clock synchronization.

Returns

The scheduler object.

4.3.4.7 scheduler()

```
void scheduler (
        void * arg )
```

Continuously runs the schedule. Should be run using an rt-kernel task with SCHED_PRIO and SCHED_STACK_SIZE as its input arguments.

A scheduling function which wakes when tasks are ready and schedules them based on their priority. Tasks become ready when their time-triggered interrupts occur, which are implemented using rt-kernel timers. When a task is run it is put in the running state and its callback function is invoked. Once finished the task is put in the waiting state and awaits a new timer interrupt.

Parameters

arg	Pointer to scheduler struct.