Fault-tolerant Transmission of Messages of Differing Criticalities Across a Shared Communication Media

Kunal Agrawal
Washington University in St. Louis
kunal@wustl.edu

Sanjoy Baruah
Washington University in St. Louis
baruah@wustl.edu

Alan Burns
The University of York
alan.burns@york.ac.uk

ABSTRACT

We discuss the motivation behind, and the design and analysis of, an algorithm for synthesizing communication schedules for shared media networks in some safety-critical hard-real-time applications such as autonomous navigation and factory automation. Communication media may be inherently noisy in many such environments, and occasional transmission errors hence inevitable. Therefore it is essential that some degree of fault-tolerance be built into the communication protocol that is used — in some safety-critical application domains, fault-tolerance requirements may be mandated by statutory certification requirements. Since the severity of the consequences of failing to successfully transmit different messages may be different, we consider a mixed-criticality setting in which the fault-tolerance requirement specification for messages are dependent on their criticality: more critical messages are required to be able to tolerate a larger number of faults. We advocate that communication schedules be "as static as possible" in safety-critical applications in order to facilitate verification and validation, and discuss the synthesis of semi-static schedules – schedules that are driven by precomputed lookup tables – with the desired fault-tolerance properties for such applications.

KEYWORDS

Fault-tolerant message transmission; Mixed criticalities; Static scheduling.

ACM Reference Format:
Kunal Agrawal, Sanjoy Baruah, and Alan Burns. 2019. Fault-tolerant Transmission of Messages of Differing Criticalities Across a Shared Communication Media. In *Proceedings of X*. ACM, New York, NY, USA, 9 pages.

1 INTRODUCTION

The research described in this document is motivated by some questions that arose in attempting to provide some fault-tolerance in a proposed communication protocol designed for IoT applications [1]. Consider the following data transmission problem. We have a communication medium (such as a wireless network, a shared bus, CAN, etc.) which is shared amongst several communicating entities ("nodes") that seek to communicate messages amongst themselves. Each message is characterized by a source node and a destination nodes (i.e. point-to-point communication, not broadcast), and takes the same duration – a "slot". We assume that the local clocks in all the nodes are adequately synchronized such that each node knows when each slot begins and ends. The set of messages that is to be transmitted is known beforehand. We wish to pre-compute a static schedule[1] denoting which message is to be transmitted during which slot, and distribute this schedule to all the nodes beforehand. During run-time, each message will only be transmitted in a slot to which it has been assigned in this schedule.

Example 1. Suppose that n messages $H_1, H_2, \ldots, H_n$ are to be transmitted. We can represent a schedule by a sequence

$$\langle \{H_1\}, \{H_2\}, \ldots, \{H_n\} \rangle,$$

denoting that the message H_i is to be transmitted during the i'th slot. We assume that this schedule is precomputed and made known to all the nodes beforehand, so that the source node for message H_i transmits the message during the i'th slot and all nodes that may be the recipient of this message are aware that they should be listening in to receive the message during this slot. □

Transmission failures. If each transmission is successful, it is evident that a schedule such as the one in Example 1 above (i) suffices for transmitting all the messages, and (ii) is *optimal* from the perspective of schedule duration: assuming that at most one message can be transmitted successfully per slot, we cannot generate a shorter schedule for transmitting all n messages.

Now, let us consider what happens if transmissions may *fail*. We assume that a failure of a transmission may arise from two factors:

(1) A *collision* – multiple messages (presumably from different nodes) are transmitted during the slot. All the messages that are transmitted during the slot are lost when such a collision occurs.
(2) A *transmission error* – this is some (external) fault in the transmission medium. If such an error occurs during a slot then even a single (and hence non-colliding) message sent during the slot will not be successfully received, and must therefore be retransmitted.

Fault-tolerant scheduling. Collisions are caused by a communication protocol sending multiple messages during the same timeslot, and are therefore under the control of the algorithm responsible for scheduling the transmission of these messages. Transmission errors, however, are caused by external factors and hence not under the algorithm's (or protocol's) control. It is of course not possible to make any non-trivial guarantees for timely transmission if transmission errors may occur arbitrarily often; guarantees can only be given if bounds are placed on the occurrence of transmission errors. Such bounds are usually expressed as a fault model. In this paper, we consider fault models of the following form: Given n messages and a fault-tolerance parameter $f \in \mathbb{N}$, we desire to generate a schedule for the messages that guarantees the successful transmission of all the messages in the presence of up to f transmission errors.

[1] See [6, Section 2] for a discussion on the benefits and drawbacks of using static schedules versus dynamic ones for safety-critical communication.

(We emphasize that fault-tolerance requirements are specified with respect to the number of transmission *errors* and not the number of transmission *failures*: since collisions are a consequence of scheduling policy, transmission failures that are caused by collisions are not covered by fault-tolerance specifications.)

Example 2. Consider once again the n messages $H_1, H_2, \ldots, H_n$ of Example 1; suppose we desire to transmit them successfully in the presence of a single transmission error ($f = 1$). A schedule of duration $2n$, obtained by transmitting each message twice, would be able to tolerate any single error. □

Efficient fault-tolerant scheduling. Example 2 above illustrates an obvious upper bound on the number of transmissions needed in order to be able to tolerate up to f errors, for any constant f: in is evident that a schedule of duration $(f + 1) \times n$, obtained by transmitting each individual message $(f + 1)$ times, suffices for tolerating up to f transmission errors. (This is in fact the exact scheme that was used in [4] to incorporate fault-tolerance in the Time-Triggered Protocol [3, 5].) The question we wish to investigate in this paper is – can we in general do better than this upper bound? The answer, it turns out, is "*yes,*"provided the transmission medium and method satisfies certain additional conditions. Specifically, we require that at the end of each slot all the nodes can determine (by monitoring the communication medium during the slot interval) whether a successful transmission has occurred during that slot or not, and not retransmit any successfully-transmitted message. That is, although our schedule is indeed static as desired in the sense that the mapping of messages to slots is statically performed prior to run-time, during run-time each node monitors the transmissions so that a successfully-transmitted message is not transmitted again even if it assigned to another slot later on for reasons of fault tolerance. We illustrate in the following example.

Example 3. Consider an instance with $n = 2$ and $f = 1$; i.e., we have two messages H_1 and H_2 that are to be transmitted in a manner that is tolerant to one transmission error. The straightforward way of achieving this would be to transmit each message twice, resulting in a schedule length of four slots.

Consider now a schedule of length three that is denoted as follows:

$$\langle \{H_1\}, \{H_2\}, \{H_1, H_2\} \rangle.$$

The interpretation of this schedule for the third slot is that both messages H_1 and H_2 are assigned to this slot. However, let us assume that a source node transmits a message during a slot to which it has been assigned in the static schedule if and only if this message has not already been successfully transmitted prior to that slot. Let us examine how this schedule would play out during run-time:

- If the first two transmissions are both successful, we are done (and hence nothing is transmitted during the third slot).
- If the first transmission is unsuccessful but the second transmission succeeds, then only H_1 is transmitted during the third slot. This second transmission of H_1 is guaranteed to be successful under the fault-tolerance assumption that at most one transmission error may occur, since an error has already occurred (during the first slot).
  Analogously if the first transmission succeeds but the second one fails, then only H_2 is transmitted during the third slot. This second transmission of H_2 is guaranteed to be successful under the fault-tolerance assumption that at most one transmission error may occur.
- If both the first and the second transmissions fail both will be transmitted during the third slot and we will have a collision; hence no message is successfully transmitted. However, observe that in this scenario the fault-tolerance assumption that at most one transmission error may occur is violated, and hence we are not obliged to ensure correct transmission of either message.

We thus see that in this case we can achieve the desired degree of fault-tolerance using just three slots, which is strictly fewer than the $n \times (f + 1) = 2 \times (1 + 1) = \mathbf{4}$ of the upper bound stated in Example 2 above. □

Introducing mixed criticalities. Suppose now that we had messages of two different *criticalities* [7], that are required to be fault-tolerant to different degrees: high-criticality messages must be able to tolerate a larger number of faults than low-criticality ones. Specifically, suppose that we have n_H high-criticality messages that must be able to tolerate up to f_H faults and n_L low-criticality messages that must be able to tolerate up to f_L faults (with $f_H > f_L$). A naive fault-tolerant scheduling strategy, which directly generalizes the idea illustrated in Example 2 above to the mixed-criticality case, would replicate each high-criticality message $(f_H + 1)$ times and each low-criticality message $(f_L + 1)$ times, to yield a schedule of length equal to

$$\Big((f_H + 1) \times n_H + (f_L + 1) \times n_L \Big) \text{ slots.}$$

But one can often do better, as illustrated in the following example.

Example 4. Suppose we had one low-criticality message L_1 and one high-criticality message H_1 (i.e., $n_H = n_L = 1$). Suppose that the low-criticality message is required to be tolerant to one transmission error ($f_L = 1$), while the high-criticality message is required to be tolerant to up to three transmission errors ($f_H = 3$). The naive strategy would yield a schedule of length

$$\Big((f_H + 1) \times n_H + (f_L + 1) \times n_L \Big) = 4 \times 1 + 2 \times 1 = \mathbf{6} \text{ slots.}$$

Consider now the following schedule of length four:

$$\langle \{H_1\}, \{H_1\}, \{H_1, L_1\}, \{H_1, L_1\} \rangle,$$

and let us examine how it would play out during run-time:

- If either of the first two transmissions succeeds, then H_1 will not transmit during the third and fourth slots and hence L_1 gets to transmit twice if needed, thereby being able to tolerate one error.
- If both of the first two transmissions fail, then *the source node of message L_1, having determined that two faults have already occurred, does not transmit during the third or fourth slots*, thereby allowing message H_1 four transmissions if necessary (and hence guaranteeing H_1 the desired tolerance of up to three faults).

We point out that the fault-tolerance requirement of the high-criticality message H_1 determines a lower bound of four on the schedule length; hence in a sense the low-criticality message L_1

in this specific example is getting to "piggy-back" onto these slots that are needed by message H_1 for free. □

This research. In this work we seek to develop a systematic approach towards synthesizing fault-tolerant static schedules of the kind discussed above, for mixed-criticality collections of messages. Our optimization objective is to have schedules of short duration, as measured by the number of slots – the fewer the number of slots, the better the schedule.

Organization. The remainder of this paper is organized in the following manner. In Section 2, we formally define the problem that we are seeking to solve. In Section 3 we derive an algorithm that solves a simpler version of this problem – in essence, the simplification of considering all messages to have the same criticality. We use this algorithm in Section 4 to solve the mixed-criticality version under certain restrictions on the parameters specifying the instance. We conclude in Section 5 by providing some context to this work, and by discussing some ways in which our work can be extended to solve more general versions in which the restrictions on the parameters are no longer necessary,

2 WORKLOAD AND FAULT MODEL

We now formally define the problem that we have informally described above. We are concerned with synthesizing fault-tolerant static schedules for transmitting messages across a shared communication medium.

Fault and Communications Model. We make the following assumptions regarding the communication infrastructure being used for the communication.

(1) Each message is characterized by a source node and a single destination node (i.e. messages are not broadcast).
(2) All messages are equi-sized; without loss of generality, we assume that each message takes one time-slot to transmit.
(3) All communicating nodes have synchronized clocks and hence share a common notion of time.
(4) The communication medium is shared (fully connected) in that all nodes can send messages directly to all other nodes (but of course not at the same time).
(5) A transmission may succeed or fail. There are two possible causes of transmission failure during a slot:
    i. A *transmission error* may occur; such occurrences are due to external causes and the scheduling and run-time mechanism has no control over these occurrences.
    ii. Sending out multiple messages during the same time-slot will result in a *collision error*.

    If neither a transmission error nor a collision error occurs during a slot, then any transmission in that slot is successful (i.e., there are no other causes of transmission failure).
    There are no additional consequences of such failure (other than the failure to successfully transmit a message).
(6) At the end of each slot *all* the nodes can determine, by monitoring the communication medium during the slot interval whether a successful transmission has occurred during that slot or not.[2]

[2]In some of our algorithms we will require nodes that are scheduled to transmit messages in the future to monitor slots in this manner; however, nodes that only receive but do not send messages do not need to do any monitoring except during slots in which they are scheduled to receive messages.

We assume that all nodes share a common view as to whether a message has been successfully received. A number of protocols can provide this atomicity, for example the use of bit by bit parity checking (as in CAN) or the use of a short 'acknowledgement' frame (as in AirTight).

Structure of static schedules. A static schedule is defined as a *finite sequence of non-empty sets of messages*. The interpretation of such a schedule is as follows. The entire schedule is pre-computed in a centralized manner and is distributed to all the nodes prior to run-time. During the i'th time-slot, the messages that are in the i'th set in this sequence are considered for transmission. The decision on whether to actually transmit each message in this set is made by the source node of the message (the node that originates that message), based on two considerations:

(1) A message that has already been successfully transmitted is not transmitted again; and
(2) A message that no longer needs to be transmitted because its fault-tolerance requirements[3] have been violated, is not transmitted.

All remaining messages in the i'th set are transmitted during the i'th slot; if there is more than one such message, a collision occurs in this slot and the transmission is a failure.

Fault-tolerance requirements. We consider mixed-criticality systems in which there are messages of two criticality levels. (Our techniques and results are easily generalized to > 2 criticality levels; we choose to not do so here in order to simplify the presentation.) Two *fault-tolerance parameters* f_H and f_L are specified, one for each criticality level. These parameters should be interpreted in the following manner:

- All the low-criticality messages should be transmitted successfully in the presence of up to f_L transmission errors; and
- All the high-criticality messages should be transmitted successfully in the presence of up to f_H transmission errors.

We require that $f_H \geq f_L$; hence we of course require that all high-criticality messages also be transmitted correctly in the presence of up to f_L transmission errors.

A problem instance is characterized by the four-tuple (H, L, f_H, f_L), where $H = \{H_1, H_2, \ldots, H_{n_H}\}$ denotes a set of n_H high-criticality messages and $L = \{L_1, L_2, \ldots, L_{n_L}\}$ denotes a set of n_L low-criticality messages, and f_H and f_L are non-negative integers with $f_H \geq f_L$. These messages are all equi-sized, and are to be transmitted across a shared broadcast medium; each high-criticality message should be successfully transmitted in the presence of up to f_H transmission errors while each low-criticality message should be successfully transmitted in the presence of up to f_L transmission errors. (A note on notation: we will often use the notation n_H and n_L to denote the cardinalities of the sets of messages H and L respectively.)

Given an instance, our ***metric*** for evaluating the "goodness" of a correct schedule (i.e., one meeting the fault-tolerance requirements

[3]The manner in which such fault-tolerance requirements are specified is discussed below.

for all messages) for the instance is its schedule length: the shorter the length, the "better" we consider the schedule to be.

3 THE SINGLE-CRITICALITY CASE

In this section we derive a strategy for generating fault-tolerant schedules for instances in which all messages have the same criticality; we will use this strategy in Section 4 below to derive a strategy for generating fault-tolerant schedules for mixed-criticality instances. For the remainder of this section, we will therefore consider an instance of the form

$$(H, L \leftarrow \{\}, f_H \leftarrow f, f_L \leftarrow 0), \text{ with } |H| = n$$

indicating that there are n high-criticality messages and no low-criticality ones, and that each message should be successfully transmitted in the presence of up to f transmission errors.

The naive approach of replicating each message $(f+1)$ times would yield a schedule of duration equal to $(f+1) \times n$ slots; below we derive a strategy that generates schedules approximately half as long.

Let us assume for now that the number of messages n is considerably larger than the number f of faults that must be tolerated. We will partition the n messages into *groups* of size $(f+1)$ messages; there will be

$$\left\lceil \frac{n}{f+1} \right\rceil$$

such groups. Each group is considered separately in synthesizing the schedule; below we describe how the messages of an individual group are considered. In order to synthesize the sub-schedule responsible for achieving the fault-tolerant transmission of a particular group of $(f+1)$ messages,

(1) We first assign each message in the group separately, one to a slot — this consumes a total of $(f+1)$ slots.
Since at most f errors are to be tolerated, we may assume that at least one message will be successfully transmitted upon the end of transmitting these $(f+1)$ slots.
(2) Next, we assign *each possible pair* of these $(f+1)$ messages, one pair to one slot. There are $\binom{f+1}{2}$ possible pairs; hence $\binom{f+1}{2}$ such slots are needed.

The total number of slots needed to construct this schedule for this block of $(f+1)$ messages is therefore

$$(f+1) + \binom{f+1}{2} = (f+1) + \frac{f \cdot (f+1)}{2} = \left(\mathbf{(f+1)}\left(\mathbf{1} + \frac{\mathbf{f}}{\mathbf{2}}\right)\right)$$

We will now argue that all $(f+1)$ messages have been successfully transmitted by the end of these slots, assuming that at most f transmission errors have occurred during these slots.

- Let $\hat{f}$ denote the number of transmission errors during the first $(f+1)$ slots – those that had transmitted one message per slot. (Observe that we must have $\hat{f} \leq f$.)
- Hence, a total of $(f+1-\hat{f})$ messages were successfully transmitted during these first $(f+1)$ slots. Since
  - each message is individually paired with every other message in the subsequent $\binom{f+1}{2}$ slots, and
  - a message that has already been successfully transmitted is not retransmitted,

  it follows that each of the messages that was not successfully transmitted during the first $(f+1)$ slots is transmitted alone $(f+1-\hat{f})$ times in the $\binom{f+1}{2}$ subsequent slots.
- Since at most $(f-\hat{f})$ additional transmission errors may occur, it follows that each such message is therefore successfully transmitted during these $\binom{f+1}{2}$ subsequent slots.

We have thus shown that each group of $(f+1)$ messages is successfully transmitted in an f-fault tolerant schedule of length $\left((f+1)(1+\frac{f}{2})\right)$; the total number of slots in the schedule for all n messages is hence given by

$$\begin{aligned}
&\left\lceil \frac{n}{f+1} \right\rceil \times \left((f+1)(1+\frac{f}{2})\right) \\
&\quad < \quad \left(\frac{n}{f+1} + 1\right) \times \left((f+1)(1+\frac{f}{2})\right) \\
&\quad = \quad n \times \left(1+\frac{f}{2}\right) + \left((f+1)(1+\frac{f}{2})\right) \\
&\quad \approx n\left(1+\frac{f}{2}\right) \qquad (1)
\end{aligned}$$

for $n \gg f$ (or for n an integer multiple of $(f+1)$, in which case $\left\lceil \frac{n}{f+1} \right\rceil = \frac{n}{f+1}$ and the additional "+1" term introduced in the second line of the derivation above is not needed).

Now there is no particular reason why the $(f+1)$ single-message slots and the $\binom{f+1}{2}$ two-message slots corresponding to each group need to be transmitted immediately one after the other — the only requirement is that the single-message slots corresponding to a group must be transmitted before the two-message ones for that same group. Hence in synthesizing a schedule for the n messages we will place all n single-message slots first, followed by all the two-message slots corresponding to all the frames; the number of such two-message frames is

$$\begin{aligned}
&\left\lceil \frac{n}{f+1} \right\rceil \times \binom{f+1}{2} \\
&\quad = \quad \left\lceil \frac{n}{f+1} \right\rceil \times \frac{(f+1) \cdot f}{2} \\
&\quad \approx \quad \frac{nf}{2} \text{ (For } n \gg f\text{)}
\end{aligned}$$

We illustrate by an example:

Example 5. Consider an instance with $n = 6$ and $f = 2$. Since $(f+1) = 3$, we will partition the six messages H_1-H_6 into the two groups H_1-H_3 and H_4-H_6. The part of the static schedule that is constructed corresponding to the first group is

$$\langle \{H_1\}, \{H_2\}, \{H_3\}, \{H_1, H_2\}, \{H_1, H_3\}, \{H_2, H_3\} \rangle,$$

and the part corresponding to the second group is

$$\langle \{H_4\}, \{H_5\}, \{H_6\}, \{H_4, H_5\}, \{H_4, H_6\}, \{H_5, H_6\} \rangle.$$

Rearranging to have all single-message slots at the beginning, the final schedule that is distributed to the communication nodes looks

SCHED(M, f)

1 Let S_a denote a schedule of contiguous single-message slots, one per message in M
2 Partition M into groups of size f each
3 Let S_b denote a schedule of contiguous two-message slots, with each two-subset of each partition appearing exactly once in this schedule
4 **return** (S_a, S_b)

Figure 1: Pseudo-code for generating an f-tolerant schedule for the messages in M. This schedule is returned as a pair of sequences of slots. The first sequence comprises single-message slots; the second, two-message slots. Their concatenation yields the desired schedule.

like this:

$$\Big\langle \{H_1\}, \{H_2\}, \{H_3\}, \{H_4\}, \{H_5\}, \{H_6\}, \\ \{H_1, H_2\}, \{H_1, H_3\}, \{H_2, H_3\}, \{H_4, H_5\}, \{H_4, H_6\}, \{H_5, H_6\} \Big\rangle. \quad (2)$$

This schedule has length **12** slots.

Consider next an instance with the same messages (and hence $n = 6$) but a more severe fault-tolerance requirement: $f = 5$. All six messages are partitioned into a single group; the schedule generated will comprise the six single-message slots followed by $\binom{6}{2} = 15$ two-messages slots, one for each distinct pair of messages, for a total schedule length of $6 + 15 = \mathbf{21}$ slots[4]:

$$\Big\langle \{H_1\}, \{H_2\}, \{H_3\}, \{H_4\}, \{H_5\}, \{H_6\}, \\ \underline{\{H_1, H_2\}}, \underline{\{H_1, H_3\}}, \{H_1, H_4\}, \{H_1, H_5\}, \{H_1, H_6\}, \underline{\{H_2, H_3\}}, \\ \{H_2, H_4\}, \{H_2, H_5\}, \{H_2, H_6\}, \{H_3, H_4\}, \{H_3, H_5\}, \{H_3, H_6\}, \\ \underline{\{H_4, H_5\}}, \underline{\{H_4, H_6\}}, \underline{\{H_5, H_6\}} \Big\rangle. \quad (3)$$

□

The algorithm we have described above is also represented in high-level pseudo-code form in Figure 1. Observe that this pseudo-code returns a pair of sub-schedules that are to be concatenated in order to achieve the desired schedule; this is done in order to facilitate the presentation of the mixed-criticality schedule-generation algorithm that we will present in Section 4 below, that makes use of the procedure SCHED(M, f) of Figure 1.

An additional observation from Example 5: we point out that in this example all the two-message slots in the 2-fault-tolerant schedule are also present in this 5-fault-tolerant schedule (observe that the second row in Expression 2, listing all the two-messages in the 2-fault-tolerant schedule, are exactly the underlined two-message slots in Expression 3). One may wonder whether this observation generalizes: for a given set of messages, are all 2-message slots in an f-fault-tolerant schedule constructed by our algorithm also present in f'-fault-tolerant schedules generated by it, for any $f' > f$? The answer is "no": consider, for instance, the case where $f \leftarrow 2$ and $f' \leftarrow 3$:

- For $f = 2$, the messages are grouped in threes: the first two groups are $\{H_1, H_2, H_3\}$ and $\{H_4, H_5, H_6\}$. Since H_4 and H_5 are in the same group, there will be a 2-message slot with H_4 and H_5 assigned to it.
- However, for $f' = 3$ the messages are grouped in fours and hence H_4 and H_5 end up in different groups, meaning that they will not appear in the same two-message slot.

So in general for $f' > f$ all 2-message slots in an f-fault-tolerant schedule need not be present in f'-fault-tolerant schedules. However, in the special case that $(f' + 1)$ is an integer multiple of $(f + 1)$, it is easy to show that this property does in fact hold:

LEMMA 1. For a given set of messages and two fault-tolerance requirement specifications f and f' such that $(f' + 1)$ is an integer multiple of $(f + 1)$, all the two-message slots in the f-fault-tolerant schedule are present in the f'-fault-tolerant schedule.

Proof Sketch: If $(f' + 1)$ is an integer multiple of $(f + 1)$, then every $(f + 1)$-sized group considered by our algorithm in synthesizing an f-fault-tolerant schedule is a subset of a single $(f' + 1)$-sized group considered by our algorithm in synthesizing an f'-fault-tolerant schedule. Hence each two-message pair that is assigned to a single slot while synthesizing the f-tolerant schedule is also assigned to a single slot while synthesizing the f'-tolerant schedule. □

A Discussion on Optimality. The schedule-generation algorithm we have derived in this section is not in general optimal from the perspective of minimizing schedule length. This is particularly easily seen for the special case $f = 1$ (i.e., only a single fault needs to be tolerated): the optimal schedule for such instances is obtained by scheduling each message in a separate slot, followed by one additional slot in which all the messages are scheduled together. However, for $f > 1$ we do not know of such "obvious" schedule-generation schemes that consistently beat the algorithm we are proposing, particularly when either $n \gg f$ holds, or n is an integer multiple of $(f + 1)$. Additionally, we believe it likely (but have not yet proved) that under the assumption that $n \gg f$ or n is an integer multiple of $(f + 1)$, optimal schedules exist that transmit either one or the same number $g > 1$ of messages per slot. If this belief holds, we are able to prove that our scheme is indeed optimal for $f > 1$; we omit a proof since we would like to first prove our belief that optimal schedules always exist in which each slot contains either one or the same number $g > 1$ of messages.

4 TWO CRITICALITY LEVELS

As stated in Section 2, a mixed-criticality problem instance is characterized by a four-tuple $\langle H, L, f_H, f_L \rangle$, denoting that there are $|H| = n_H$ high-criticality messages and $|L| = n_L$ low-criticality messages that are to be transmitted, and that the high-criticality messages (low-criticality messages, respectively) should be tolerant to up to f_H (f_L, resp.) transmission errors. We start out making the simplifying assumption that Lemma 1 is applicable; i.e., *the parameters f_H and f_L satisfy the property that $(f_H + 1)$ is an integer multiple of $(f_L + 1)$*, and develop an algorithm for generating fault-tolerant schedules for instances satisfying this property; we will discuss the implications of making this assumption in Section 5.

Our schedule-generation algorithm can be thought of as comprising four steps, the first three of which use the algorithm SCHED(M, f)

[4]The reason for underlining some slots in this schedule depiction is explained below.

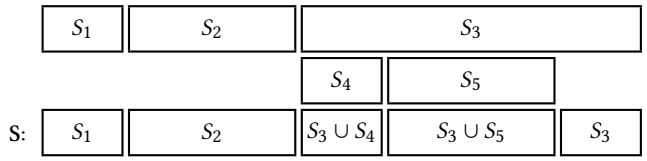

Figure 2: How the sub-schedules S_1–S_5 are merged to obtain the final schedule. The top row depicts the schedules generated for H, the middle row, those for L. The bottom row shows how the final schedule S is synthesized. First, add all the slots in S_1. Next, add all the slots in S_2. Finally, add the slot-by-slot union of the slots in S_3 and the schedule obtained by concatenating S_4 and S_5.

described in Figure 1 (Section 3) to generate schedules for some non-mixed-criticality instances that are derived from the mixed-criticality instance, and the fourth "merges" these generated schedules to obtain the desired schedule. We describe these four steps below; they are also represented in pseudo-code form in Figure 3 (and illustrated on an example instance in Example 6 — it may be helpful to follow along on the example).

§1. First, we construct an f_L-tolerant schedule of the n_H high-criticality messages, arranging this schedule to have all single-message slots appear before all two-message slots (as discussed in Section 3, and illustrated in Example 5). Let S_1 denote the single-message part of this schedule, and S_2 the two-message part: the schedule generated is the concatenation of S_1 and S_2.

This step is depicted in pseudo-code form as Line 1 of the pseudo-code in Figure 3.

§2. Next, we construct an f_H-tolerant schedule of these n_H high-criticality messages, again arranging all single-message slots before all two-message slots. Since we're assuming that $(f_H + 1)$ is an integer multiple of $(f_L + 1)$, it follows from Lemma 1, all the two-message slots in S_2 (the two-messages slots constructed in step §1 above) are also contained in this schedule. Let S_3 denote the remaining two-message slots in this schedule. We arrange our schedule to have all slots in S_2 appear before the slots in S_3, so that the schedule generated in this step is the concatenation of S_1, S_2, and S_3.

This step is depicted in pseudo-code form as Lines 2 and 3 of the pseudo-code in Figure 3.

§3. Next, we construct an f_L-tolerant schedule of the n_L low-criticality messages. Let S_4 denote the single-message-per-slot part of this schedule, and S_5 the two-messages-per-slot part. This step is depicted in pseudo-code form as Line 4 of the pseudo-code in Figure 3.

§4. Our final step is to "*merge*" the three schedules generated in the steps above into one integrated static schedule. This is done in the following manner (also see Figure 2).

a) The first part of the integrated schedule is S_1 (Lines 5–6 of the pseudo-code in Figure 3).
b) The second part is S_2, the two-message slots from the f_L-fault-tolerant schedule for the high-criticality messages (Lines 7–8 of the pseudo-code in Figure 3).
c) In order to better motivate the synthesis of the remainder of the schedule, let us now consider the possible scenarios during run-time after the execution of S_1 and S_2.
   - If no more than f_L faults have occurred thus far, then all high-criticality messages have been successfully transmitted by this point in time, and we should henceforth be transmitting low-criticality messages. Therefore, the schedules S_4 and S_5 should be transmitted next (in this order – i.e., S_4 followed by S_5).
   - However if more than f_L faults have already occurred, then the low-criticality fault-tolerance threshold has been crossed and we hence need no longer transmit the low-criticality messages. In this case, we should be transmitting the remainder of the f_H-fault-tolerant schedule for the high-criticality messages that we had generated in step §2 above. The parts S_1 and S_2 of this schedule have already been transmitted; it remains to transmit the part S_3.

   We want to enumerate a static schedule that covers both scenarios described above: if at most f_L faults have occurred during the first $|S_1| + |S_2|$ slots, we should transmit according to schedule S_4 followed by schedule S_5; otherwise, we should transmit according to schedule S_3 Therefore, the messages to be assigned to each slot after the first $|S_1| + |S_2|$ slots is obtained by taking the union of the messages assigned to that particular slot in the schedule S_3 and the schedule obtained by concatenating S_5 to the end of S_4. (All this is represented in pseudo-code form in Lines 9–17 of the pseudo-code in Figure 3). The part of the schedule generated in this manner achieves what we desire, since
   - If no more than f_L faults have occurred prior to the execution of this part of the schedule, then all high-criticality messages have already been successfully transmitted. Since a successfully transmitted message is never retransmitted by its source node, then in each slot only the message[s] assigned to this slot in the concatenation of S_4 and S_5 are potentially transmitted.
   - If more than f_L faults have occurred, then all sources of low-criticality messages will cease transmitting these messages, and no messages from the concatenation of S_4 and S_5 will henceforth be transmitted. Therefore, in each slot only the message[s] assigned to this slot in S_3 are potentially transmitted.

This next example traces the operation of our schedule-generation algorithm on a simple example mixed-criticality instance.

Example 6. Let us consider an instance with six high-criticality messages ($H = \{H_1, H_2, \ldots, H_6\}$) and three low-criticality messages ($L = \{L_1, L_2, L_3\}$), with a high-criticality fault-tolerance requirement $F_H = 5$ and a low-criticality fault-tolerance requirement $F_L = 2$. We now describe the sub-schedules S_1–S_5 generated by our algorithm on this example (these sub-schedules are listed in Figure 4).

- S_1 is **6** slots long, and has one slot for each message $\in H$.

```
SCHEDMC(H, H, f_H, f_L)
1  (S_1, S_2) = SCHED(H, f_L) // See Figure 1 for the pseudo-code representation of SCHED(M, f)
2  (S_1, S') = SCHED(H, f_H)
3  S_3 = S' \ S_2 // Here S' \ S_2 denotes the slots in S' that are not in S_2
4  (S_4, S_5) = SCHED(L, f_L)
   // Now, construct schedule S from schedules S_1–S_4
5  for i = 1 to |S_1| // First, copy out schedule S_1
6      S[i] = S_1[i]
7  for i = 1 to |S_2| // Next, copy out schedule S_2
8      S[|S_1| + i] = S_2[i]
9  for i = 1 to |S_4| // Next, do a slot-by-slot union of schedules S_3 and S_4
10     if (|S_3| ≤ i) then tmp = S_3[i] else tmp = {}
11     S[|S_1| + |S_2| + i] = (S_4[i] ∪ tmp)
12 for i = 1 to |S_5| // Next, do a slot-by-slot union of schedules S_3 and S_5
13     if (|S_3| ≤ |S_4]| + i) then tmp = S_3[|S_4]| + i] else tmp = {}
14     S[|S_1| + |S_2| + |S_4| + i] = (S_5[i] ∪ tmp)
15 if (|S_3| > |S_4| + |S_5|) // Finally, copy out any remaining slots in schedule S_3
16     for i = |S_4| + |S_5| + 1 to |S_3|
17         S[i] = S_3[i]
18 return S
```

Figure 3: Pseudo-code representation of our algorithm for generating fault-tolerant static schedules for mixed-criticality instances

- S_2 is $2 \times \binom{3}{2}$ or **6** slots long: it has one slot for each pair of messages in $\{H_1, H_2, H_3\}$, and one slot for each pair of messages in $\{H_4, H_5, H_6\}$.
- S' is $\binom{6}{2}$ or 15 slots long, and has one slot for each pair of messages in H. Hence S_3, which contains one slot for each pair of messages in H that is not already contained in S_2, is $(15 - 6)$ or **9** slots long.
- S_4 is **3** slots long, and has one slot for each message $\in L$
- S_5 is $\binom{3}{2}$ or **3** slots long, and has one slot for each pair of messages $\in L$.

Merging these schedules together, we get the following schedule S (also listed in Figure 4):

- The first six slots are identical to S_1.
- The next six slots are identical to S_2.
- The next three slots are obtained by taking a slot-by-slot union of the first three slots ofS_3 with the three slots in S_4; they are therefore

$$\Big\langle \{H_1, H_4, L_1\}, \{H_1, H_5, L_2\}, \{H_1, H_6, L_3\} \Big\rangle$$

- The next three slots are obtained by taking a slot-by-slot union of the next three slots of S_3 with the three slots in S_5; they are therefore

$$\Big\langle \{H_2, H_4, L_1, L_2\}, \{H_2, H_5, L_1, L_3\}, \{H_2, H_6, L_2, L_3\} \Big\rangle$$

- The remaining three slots of S_3 constitute the final three slots of the schedule S:

$$\Big\langle \{H_3, H_4\}, \{H_3, H_5\}, \{H_3, H_6\} \Big\rangle$$

□

Evaluation. As stated in Sections 1 and 2, our metric for evaluating the "goodness" of the static schedules we generate is their length: the number of slots in the schedules. Let us now evaluate how our algorithm measures up with regards to this metric. We start out enumerating some fairly obvious facts about the schedule generated by our algorithm.

Fact 1. $(|S_1| + |S_2|)$ is the length of an f_L-tolerant schedule for the messages in H.

Fact 2. $(|S_4| + |S_5|)$ is the length of an f_L-tolerant schedule for the messages in L.

Fact 3. Since there are no common messages in H and in L, it follows from the facts above that $(|S_1| + |S_2| + |S_4| + |S_5|)$ is the length of an f_L-tolerant schedule for the messages in $(H \cup L)$

Fact 4. $(|S_1| + |S_2| + |S_3|)$ is the length of an f_H-tolerant schedule for the messages in H.

Now, the length of the schedule we have generated is given by

$$\begin{aligned} & |S_1| + |S_2| + \max(|S_3|, |S_4| + |S_5|) \\ = \quad & \max\big(|S_1| + |S_2| + |S_3|, |S_1| + |S_2| + |S_4| + |S_5|\big) \end{aligned}$$

From Facts 4 and 3, it follows that this is *the maximum of the lengths needed for an f_H-tolerant schedule for the messages in H, and an f_L-tolerant schedule for the messages in $(H \cup L)$*. That is, our schedule length is no more than the schedule length we would use to transmit *all the messages* under the less conservative fault model, or to transmit *only the high-criticality messages* under the more conservative fault model. In a sense, we are paying the cost for only the more expensive of these fault-tolerance requirements: the other is being provided "for free" by piggy-backing on the already-required slots.

$$
\begin{aligned}
S_1 &= \Big\langle \{H_1\}, \{H_2\}, \{H_3\}, \{H_4\}, \{H_5\}, \{H_6\} \Big\rangle \\
S_2 &= \Big\langle \{H_1, H_2\}, \{H_1, H_3\}, \{H_2, H_3\}, \{H_4, H_5\}, \{H_4, H_6\}, \{H_5, H_6\} \Big\rangle \\
S_3 &= \Big\langle \{H_1, H_4\}, \{H_1, H_5\}, \{H_1, H_6\}, \{H_2, H_4\}, \{H_2, H_5\}, \{H_2, H_6\}, \{H_3, H_4\}, \{H_3, H_5\}, \{H_3, H_6\} \Big\rangle \\
S_4 &= \Big\langle \{L_1\}, \{L_2\}, \{L_3\} \Big\rangle \\
S_5 &= \Big\langle \{L_1, L_2\}, \{L_1, L_3\}, \{L_2, L_3\} \Big\rangle
\end{aligned}
$$

$$
\begin{aligned}
S = \Big\langle & \overbrace{\{H_1\}, \{H_2\}, \{H_3\}, \{H_4\}, \{H_5\}, \{H_6\}}^{S_1}, \overbrace{\{H_1, H_2\}, \{H_1, H_3\}, \{H_2, H_3\}, \{H_4, H_5\}, \{H_4, H_6\}, \{H_5, H_6\}}^{S_2}, \\
& \overbrace{\{H_1, H_4, L_1\}, \{H_1, H_5, L_2\}, \{H_1, H_6, L_3\}}^{\text{by per-slot } (S_3 \cup S_4)}, \overbrace{\{H_2, H_4, L_1, L_2\}, \{H_2, H_5, L_1, L_3\}, \{H_2, H_6, L_2, L_3\}}^{\text{by per-slot } (S_3 \cup S_5)} \\
& \overbrace{\{H_3, H_4\}, \{H_3, H_5\}, \{H_3, H_6\}}^{\text{remainder of } S_3} \Big\rangle
\end{aligned}
$$

Figure 4: Applying the schedule-generation algorithm – see Example 6.

A possible alternative to the algorithm we have proposed in this section would be to use the (single-criticality) algorithm of Section 3 to separately synthesize an f_L-tolerant schedule for the messages in L and an f_H-tolerant schedule for the messages in H, and then concatenate these two schedules together. Based on Equation 1, we conclude that the length of this schedule is approximately

$$
\begin{aligned}
& n_L\big(1 + \frac{f_L}{2}\big) + n_H\big(1 + \frac{f_H}{2}\big) \\
= \quad & (n_L + n_H)\big(1 + \frac{f_L}{2}\big) + n_H\big(\frac{f_H - f_L}{2}\big)
\end{aligned}
$$

Now the second term in the first line represents the length of a schedule to transmit only the high-criticality messages under the more conservative fault model, while the first term in the second line represents the length of a schedule to transmit all the messages under the less conservative fault model. Since both lines have an additional term $\big(n_L\big(1 + \frac{f_L}{2}\big)$ in the first line, $n_H\big(\frac{f_H - f_L}{2}\big)$ in the second), it follows that the new algorithm that we proposed in this section for scheduling mixed-criticality instances is strictly superior to the criticality-agnostic one of Section 3.

We conclude this section with some **numerical comparisons** of the schedule-lengths produced by the various schemes we have studied in this paper. Although these numerical comparisons are not meant to be exhaustive or definitive, we do believe that they are somewhat representative and provide a flavor of the kinds of savings in schedule length we are able to achieve.

On an instance (H, L, f_H, f_L) with $|H| = n_H$ and $|L| = n_L$, the naive strategy [4] of replicating each message individually yields a schedule with

$$n_L(1 + f_L) + n_H(1 + f_H) \tag{4}$$

slots. As discussed above, the criticality-agnostic scheme of Section 3 yields a schedule with length approximately

$$n_L\big(1 + \frac{f_L}{2}\big) + n_H\big(1 + \frac{f_H}{2}\big), \tag{5}$$

while the algorithm derived in this section results in a schedule of length approximately

$$\max\Big((n_L + n_H) \times \big(1 + \frac{f_L}{2}\big), n_H\big(1 + \frac{f_H}{2}\big)\Big) \tag{6}$$

The lengths of the schedules generated by these three different algorithms upon some example instances are presented in Table 1. The entries in the first are those of the example instance considered in Example 6 above. The next five rows are for instances with the same fault parameters as Example 6 and the number of high-criticality messages fixed (at 18), but with the number of low-criticality messages varied to equal $\times 1$, $\times 2$, $\times 3$, $\times 4$, and $\times 5$ the number of high-criticality messages. The last five rows have fault parameters $f_H \leftarrow 8$ and $f_L \leftarrow 2$; the number of high-criticality messages is fixed at 27 while the number of low-criticality messages varied to equal $\times 1$, $\times 2$, $\times 3$, $\times 4$, and $\times 5$ the number of high-criticality messages. It can be seen from the last two columns of the table that the criticality-cognizant algorithm generates schedules that are shorter than the naive ones by approximately a factor of two, and shorter than the ones generated by the criticality-agnostic algorithm by a factor that is, on average, greater than 1.25.

5 CONTEXT & FUTURE DIRECTIONS

A tremendous body of work has been done on achieving fault-tolerant communication in shared networks, from both a pragmatic and theoretical perspective, see any of several surveys (e.g., [6] is a seminal and widely-cited survey, while [2] is a more recent one) for further detail. However, we did not find any schemes that are similar to the one we have derived – perhaps this is because there isn't much prior work on fault-tolerant algorithms for mixed-criticality messaging considering a notion of mixed criticalities along the fault dimension (as we are doing here).

n_H	n_L	f_H	f_L	Naive (Eqn 4)	Sec 3 (Eqn 5)	Sec 4 (Eqn 6)	Naive ÷ Sec 4	Sec 3÷Sec 4
6	3	5	2	45	27	21	2.14	1.29
18	18	5	2	162	99	72	2.25	1.375
18	36	5	2	216	135	108	2	1.25
18	54	5	2	270	171	144	1.875	1.1875
18	72	5	2	324	207	180	1.8	1.15
18	90	5	2	378	243	216	1.75	1.125
27	27	8	2	324	189	135	2.4	1.4
27	54	8	2	405	243	162	2.5	1.5
27	81	8	2	486	297	216	2.25	1.375
27	108	8	2	567	351	270	2.1	1.3
27	135	8	2	648	405	324	2	1.25

Table 1: Representative numerical comparison of schedule lengths generated by the naive approach (Eqn 4), the criticality-agnostic approach of Section 3 (Eqn 5), and the criticality-cognizant approach of Section 4 (Eqn 6).The last two columns depict the relative efficiency of the criticality-cognizant algorithm over the other two.

The algorithms we have presented here make two simplifying assumptions:

(1) The initial (non-mixed-criticality) algorithm assumes that the number of messages to be transmitted is an integer multiple of one plus the degree of fault-tolerance required. Since this algorithm is subsequently used in the mixed-criticality algorithm to synthesize sub-schedules, this assumption is required in the mixed-criticality case as well.
(2) For mixed-criticality instances, we require that $(f_H + 1)$ be an integer multiple of $(f_L + 1)$ (recall that f_H and f_L are the degrees of fault-tolerance required by high-criticality and low-criticality messages respectively), in order that the conditions of Lemma 1 hold.

Getting rid of the first assumption is straightforward, but rather tedious – one must in essence separately consider the $\big(n \bmod (f + 1)\big)$ messages that are left over after all the remaining ones have been grouped into $(f + 1)$-sized groups, and separately synthesize an optimal f-fault-tolerant schedule for these messages. Choosing to not do so, but rather simply assigning each two-subset of these $\big(n \bmod (f + 1)\big)$ messages a separate slot, appears an adequate hack that does not compromise optimality by too much provided n is reasonably large when compared to f.

Getting rid of the second assumption is somewhat more challenging: there appear to be several different approaches possible with different trade-offs between benefits and drawbacks. We are currently working on examining these different approaches in order to better understand whether one approach can be shown to strictly dominate the others; meanwhile, a safe and correct (although sub-optimal) work-around is to increase one or the other of the fault-tolerance requirements to achieve the desired divisibility property. For instance suppose that we had $(f_L = 3, f_H = 9)$. Although $(3 + 1 =) 4$ does not divide $(9 + 1 =) 10$ exactly, $(4 + 1 =) 5$ does, and so we could consider increasing f_L to 4, thereby providing more fault-tolerance to the low-criticality messages than required. Alternatively we could consider not changing f_L but instead increasing f_H to 11 (since $(3 + 1 =) 4$ divides $(11 + 1 =) 12$). Thus either $(f_L = 4, f_H = 9)$ or $(f_L = 3, f_H = 11)$ could be used; which is preferred depends on the relative values of n_L and n_H.

In this paper we have assumed a relatively simple fault model for specifying fault-tolerance requirements: messages should be tolerant to up to a specified number f of faults. A more sophisticated fault model would require tolerance for a specified number of faults *over a specified duration of time*. In a general form, we can think of such fault-tolerance specifications as a monotonically non-decreasing function $f : \mathbb{N} \to \mathbb{N}$, with the interpretation that $f(t)$ denotes the maximum number of faults that must be tolerated over any interval of duration t. Extending the algorithms we have presented here to such a more general fault model seems interesting and challenging, and provides further motivation for minimizing the duration of the schedule.

REFERENCES

[1] Alan Burns, James Harbin, Leandro Indrusiak, Iain Bate, Robert Davis, and David Griffin. 2018. AirTight: A Resilient Wireless Communication Protocol for Mixed-Criticality Systems. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '18)*. IEEE Computer Society.

[2] Beom-Su Kim, HoSung Park, Kyong Hoon Kim, Daniel Godfrey, and Ki-Il Kim. 2017. Review Article: A Survey on Real-Time Communications in Wireless Sensor Networks. *Wireless Communications and Mobile Computing* (2017).

[3] H. Kopetz and G. Bauer. 2003. The time-triggered architecture. *Proc. IEEE* 91, 1 (Jan 2003), 112–126.

[4] H. Kopetz and G. Grunsteidl. 1993. TTP - A time-triggered protocol for fault-tolerant real-time systems. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*. 524–533. https://doi.org/10.1109/FTCS.1993.627355

[5] Hermann Kopetz and Günter Grünsteidl. 1994. TTP – A Protocol for Fault-Tolerant Real-Time Systems. *Computer* 27, 1 (Jan. 1994), 14–23. https://doi.org/10.1109/2.248873

[6] John Rushby. 2001. Bus Architectures for Safety-Critical Embedded Systems. In *Proceedings of the ACM International Conference on Embedded Software (EMSOFT)*, Thomas A. Henzinger and Christoph M. Kirsch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 306–323.

[7] Steve Vestal. 2007. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Proceedings of the Real-Time Systems Symposium*. IEEE Computer Society Press, Tucson, AZ, 239–243.