A Flexible Slotting Scheme for TDMA-Based Protocols

Jens Chr. Lisner

ICB / University of Duisburg-Essen
lisner@informatik.uni-essen.de

Abstract: Popular fault-tolerant[1] TDMA-based protocols like TT-CAN ([Fu00] and [Mu02]), TTP ([KG93]) and FlexRay ([Fl02]) are using a static slotting scheme for bus arbitration. This paper describes a possible solution for a more flexible handling of the slotting, by providing extra time of variable length in the cycle for additional slots. These extra time is only provided if it is requested by the controllers in the network. Another possibility is to configure extra slots statically (to come to a fixed cycle length) and assign the slots dynamically to different controllers. A new node architecture is introduced for improving protection of the bus in case of a faulty controller. An agreement algorithm for determining a new schedule every cycle in the distributed network is presented. The system is capable of tolerating double-faults (one controller and one channel).

1 Motivation

In common fault tolerant TDMA-based protocols a time slot is assigned to each communication controller (processing the communication of a node) within a communication cycle (see figure 1). During fault-tolerant operation, messages are filling a whole slot. The schedule of the slots, the slots' lengths and the length of the communication cycles are statically preconfigured. This assures that all fault-free controllers in a distributed network can communicate, without using special arbitration techniques like CSMA/CD (eg. CAN, ...). To assure collision-free communication, a guardian prevents faulty controllers from sending in a slot it is not scheduled for. For redundancy it is common to use a two channel architecture. In this case, there is one guardian for each channel.

The drawback of this method lies in the fact that controllers are restricted to the order given by the preconfigured schedule. It is not possible for a controller to send in additional slots in the current communication cycle, if it needs temporarily higher throughput or redundant communication. To solve this problem with a TDMA-like protocol it is required to configure additional individual slots for each controller. This makes the communication cycle longer. All changes in the static configuration are permanent throughout the runtime of the network. If a controller needs additional slots rarely, bandwidth is wasted because the sum of the maximum slot numbers for each controller has to be taken into account. The

[1]This excludes TDMA-based protocols in wireless and satellite communication, which are not designed for use in safety critical environments.

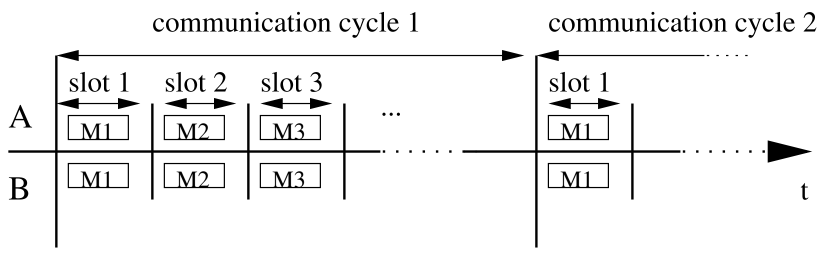

Figure 1: Controllers C1, C2, ... sending messages M1, M2, ... respectively in subsequent time slots on channels A and B synchronously.

proposed way to solve this problem is to change the length of the communication cycle dynamically.

This is not feasible in TDMA-based networks with static scheduling for the following reasons: First, there must be an agreement on a new schedule for all controllers. Second, not only the controllers, but all the bus guardians have to be informed about the schedule change independently from the controllers, because a faulty controller may mislead the bus guardians. The bus guardian is a cheap component, which must know about the current schedule. To be able to decide when the controller is allowed to send or not, the bus guardian must agree with all other fault-free nodes. This requires message decoding by the bus guardians. This makes the bus guardians more complex.

The FlexRay ([Fl02]) protocol introduces a possible solution for the problem. The protocol works with two different arbitration methods, a statically configured TDMA scheme with equal-length slots, and a minislotting solution.[2] In the minislotting part the schedule is not fixed, but the length of this part is always the same. Moreover, this part cannot be protected in an efficient way by guardians, so fault-tolerance is not guaranteed.

2 The New Slotting Scheme

The solution presented in this paper, enables each controller to use one additional slot within a communication cycle, if needed. All controllers are able to announce that they need one more slot in the current cycle. These slot reservation requests are collected and the cycle is increased by the respective number of slots. This should be done in a fault-tolerant way. The algorithm presented in this paper will be proved to tolerate one faulty controller together with one faulty channel in a double-channel system. To achieve this, the communication cycle is splitted into two parts. The first one is the regular part, where all controllers send at least once. The schedule in this part is fixed as in a static TDMA-scheme. The second part is the extension part, which consists of all additional requested slots. If there are no slots requested, the extension part is left out, otherwise the length of

[2]A similar approach is used in the Byteflight protocol ([BPG00]).

the extension part is the sum of the lengths of the requested slots. After the extension part is completed, the next communication cycle starts. Figure 2 shows an example.

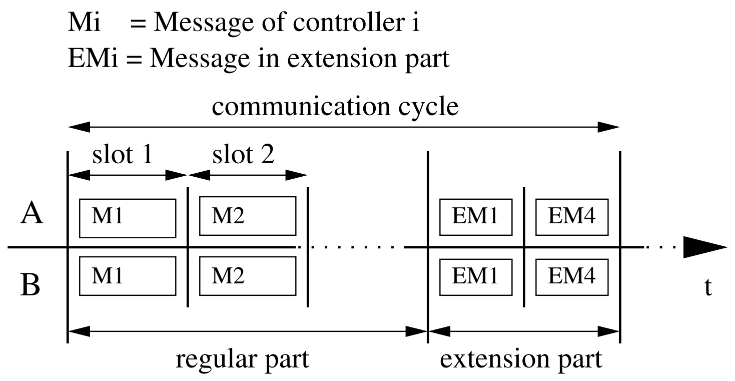

Figure 2: A communication cycle with regular and extension part. Two controllers requested a slot in the extension part.

As replacement for the inflexible bus guardian a new architecture is introduced, where two controllers are grouped together in a single node, so that a controller can disallow the other one to send. If a controller fault occurs in such a node, both channels are perfectly protected by the independent second controller. There is a drawback: A faulty controller may erroneously prevent a fault-free controller from sending. In this case the node gets completely disconnected from the network. If the requirements do not allow this, more redundant controllers or even nodes could be used. In any case the fault tolerance of the solution proposed here is not worse than single controller solutions. Figure 3 shows the new architecture. Instead of bus guardians, switches are used to protect the channels. These switches are not more than driver components which can be turned on or off under the neighbours control.

3 Discussion of Possible Fault Scenarios

All messages a node sends are assumed to be protected by a checkword (e.g. CRC), so that message corruption can always be detected. A message can be corrupted, if:

1. it was send corrupted (usually wrong physical coding) by a faulty controller,
2. it was assembled with a invalid checkword by a faulty controller,
3. the transmission was interrupted within a node by the neighbour controller, or
4. the message was corrupted on a faulty channel.

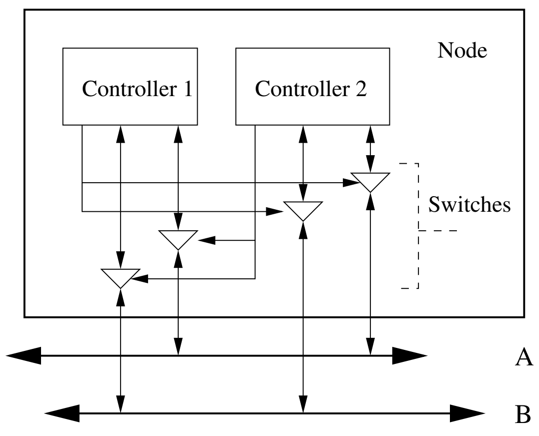

Figure 3: Two nodes controlling each other. A controller is able to deny its neighbour to send.

If an expected message is missing, this is considered as message corruption, too. Note that empty slots are not allowed. All slots must be filled with messages or pseudo-messages (null messages).

A faulty channel can lead to Byzantine faults, if messages with correct checkwords are delivered only to a subset of all controllers, while other controllers receive corrupted messages. It is not possible for a faulty channel to change parts of the message without corrupting it or to spontaneously generate messages with correct checkword.

To make all controllers able to guard their neighbours correctly in the extension part, the schedule in the extension must be agreed by all controllers within the regular part.

3.1 Single-Fault Scenarios

In a single-fault-scenario, there is either one faulty channel or faulty controller.

In case of a faulty channel, a message can be received on the respective other channel. Since there is no faulty controller, all messages contain a correct reservation request.

In case of a faulty controller without a faulty channel, faults may occur in the time or value domain. In the time domain, the controller can send out of the bounds of the slot it is assigned to. This is prevented by the non-faulty neighbour controller. In the value domain a message can be sent coded correctly or corrupted. This can happen on one channel or on both channels. The following combinations are possible:

1. Two correctly coded messages are sent on both channels. In this case all fault-free controllers agree on the information coded within the message.
2. One correctly coded message is sent on one channel, while one corrupted message is

sent on the other. In this case the fault-free controllers cannot distinguish between a channel and controller fault. Per default they trust in the information they received.

3. Two corrupted messages are received. In this case there must be a pre-defined default decision.

Note that, as a consequence of the chosen architecture, these types of faults can also be applied to the neighbour of a faulty controller. A non-faulty neighbour of a faulty controller can be handled the same way in case of faults.

Some behaviour can only occur in the case of a faulty controller. A faulty controller might compute a checkword on the basis of a request information which has no relation to the behaviour of the controller in the extension. Such misleading requests can only be sent by faulty controllers. Misleading requests are handled as follows: If the fault-free controllers can agree on the request the faulty controller has sent, they adjust their scheduling as requested. It is the task of the fault-free neighbour to prevent the faulty controller from sending without permission. If messages are corrupted, or the faulty controller sends contradicting requests on both channels, a pre-defined default rule has to be applied.

For all the scenarios, which have been described so far, the result shows that agreement can be reached without further activity in either case of one faulty channel or one faulty controller.

3.2 Double-Fault Scenarios

The double-fault scenario includes one faulty channel, one faulty controller and one fault-free neighbour controller, which can be blocked by the faulty one. The non-faulty controller, however, protects both channels.

If C is the set of controllers, f is the faulty one and n its neighbour, every controller c can receive a corrupted message, a misleading request (see section 3.1) or a fault-free message. On the faulty channel a subset $N_s \subseteq C$ of controllers receive a corrupted message in slot s. Note that N_s is not constant in different slots. Controllers from $C - N_s$ receive the message "as is" from a controller (faulty or non-faulty).

As seen before, a faulty controller can behave differently on each channel. This is important, because the different receivers might come to different conclusions, especially if one receiver has no usable input and must fall back to a default, and another needs not. As discussed in the last section, there are two different scenarios where this can happen. If $k \in N_s$ and $l \notin N_s$ are fault-free receivers, f can send a corrupted message on the fault-free channel, but a non-corrupted on the faulty one. In this case k must fall back to a default action (reserve or not reserve a slot), since it sees corrupted messages on both channels. But l sees the message on one channel, and accepts the request from f. If the request does not match the default action, k and l disagree. On the other side if f sends correct messages with different requests, k agrees on the request received but l sees a contradiction.Because l has to fall back to the default value both receivers disagree. Note that this situation is impossible in the case of n.

For an agreement on this benign Byzantine fault, at least three controllers must broadcast their decision to the other controllers. In a TDMA-based protocol, this could be the next three controllers following in the schedule. A problem arises at the end of the regular part. When the extension starts, the schedule must be fully agreed on.

4 Fault-Tolerant Operation

To solve this problem, two controllers are sending in each slot of the regular part. One controller sends via channel **A**, the other controller via channel **B**. The regular part is subdivided into two phases. These are called the "request" and "confirmation" phase. The request phase lasts from slot 1 to $\frac{m}{2}$, where m is the (even) number of slots (which is also the number of controllers in the network). Consequently, the confirmation phase ranges from slot $\frac{m}{2} + 1$ to m. A possible schedule for the regular part would be to schedule the odd numbered controllers in the request phase on channel **A** and for the confirmation phase on channel **B**, and vice versa for the even numbered controllers. This schedule is illustrated in figure 4.

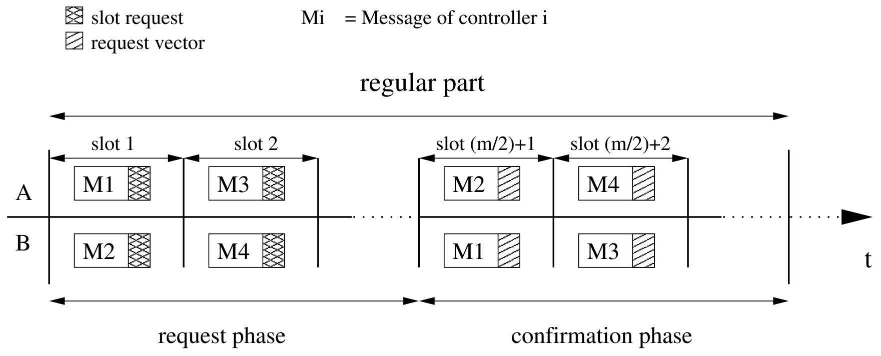

Figure 4: The regular part is split into two phases. In the request phase all controllers send a request for a slot. In the confirmation phase the requests which were received are sent as vector, by all controllers.

In the request phase, the controllers send their messages, with a possible request for a slot in the request field of the message (see figure 4). Each receiver builds up a request vector, containing the requests of the senders. The possible values appear in section 4.1 as `request_slot`, `request_no_slot` or `unknown`. A field is set to `unknown`, if the received message was corrupted. This is possible for senders scheduled on the faulty channel, for the faulty controller f, if it sends a corrupted message, or its neighbour n, if it is blocked by f during sending. For these senders, the entries can be different for different receivers. If a field is not set to `unknown` for each two receivers for a sender, then the content of the fields are equal (either `request_slot` or `request_no_slot`).

In the confirmation phase the controller sends the request vectors instead of a single request. Similar to the activities in the request phase, the receivers build a matrix from the set of received vectors. The resulting matrix is called `request_matrix` (see section 4.1). Each vector forms a row in this matrix, which is a square matrix with m rows and m columns. The entry `request_matrix[i][j]` expresses the request of node j received by node i in the request phase. If a vector could not be received, the whole row is marked as `corrupted`.

The matrix is the basis of the agreement algorithm described in detail in section 4.1. The outcome of this algorithm is a schedule for the extension part. The algorithm tolerates one faulty controller with arbitrary behaviour (including blocking of its neighbour), and one faulty channel in addition. This includes that the faulty controller can send a non-corrupted message carrying a random vector in the confirmation phase.

A row in the matrix is guaranteed not to be corrupted, if the sender is a non-faulty and non-blocked controller, and was scheduled to send on the fault-free channel. These rows are the same in every receiving controller. Since the vector can be different for different controllers, it is not guaranteed, that two rows match. On the other side it is guaranteed, that the entries in the columns of those controllers, which were scheduled for the fault-free channel, show either the original request, `corrupted` (see lemma 2 in section 4.2) or possibly `unknown` for controllers n and f . An exception of this is the entry of the row whose vector origins from f which can send random values in the vector. All in all, there are at least 4 values per column necessary to come to an agreement, two of them from non-faulty and non-blocked controllers. This is the reason why there must be at least 8 controllers available in the network.

4.1 An Algorithm for Agreement on the Schedule in the Extension

Each controller has to perform an algorithm to come to a schedule for the extension, using the information gathered in the regular part. The result must be the same for all fault-free controllers.

The matrix of received vectors with requests from controllers, the type of a request and the result schedule vector used in the following are defined below. The values defined in `request` are possible entries in the matrix. The first index of the matrix is the row (a received vector) the second the column. If the entry in the `result_schedule[c]` is set to `true` for a particular controller `c`, then the controller is scheduled in the extension part.

```
typedef enum { requested_slot, requested_no_slot,
               unknown, corrupted } request;

request request_matrix[m][m];
boolean result_schedule[m];
```

The function `majority(c, channel)` returns the majority of requests in the column for controller `c` of the matrix. It tolerates one misleading request from a faulty-controller.

This is the reason why the function can return a valid request only, if at least two controllers vote for one particular value (see line 36 and lemma3 in section 4.2). To determine the majority, the entries of the column for controller c and a specific channel are used. Each channel is handled separately, because the results of the faulty and fault-free channel must be handled differently. For details see section 4.2.

```
   // Defined at configuration time:
   // scheduled_for_channel is true, if a controller
   // (first index) is scheduled for a specific
   // channel (second index) in the confirmation phase.
20  const boolean scheduled_for_channel[m][2];

21  request
22  majority(int c, int channel)
23  {
24    int request_slot_counter = 0;
25    int request_no_slot_counter = 0;

26    for (int i = 0; i < m; i++)
27    {
28      if (scheduled_for_channel[i][channel])
29      {
30        if (request_matrix[i][c] == request_slot)
31          request_slot_counter++;
32        else if (request_matrix[i][c]
                    == request_no_slot)
33          request_no_slot_counter++;
34      }
35    }
36    if (max (request_slot_counter,
               request_no_slot_counter) < 2)
37      return unknown;
38    else if (request_slot_counter
                             > request_no_slot_counter)
39      return request_slot;
40    else
41      return request_no_slot;
42  }
```

The following code builds the vector for the schedule in the extension. For all controllers allowed to send, the value is `true`. The default value, for a controller, when the scheduling request could not be determined, is `false`. The code will be analyzed in section 4.2. The function `count_unknown()` (see lines 54–58) returns the number of entries set to `unknown` for a channel within a column. If this value is greater than one (line 61) for the channel a controller is scheduled in the confirmation phase, the result is not trustworthy, and the default value must be applied. The reason is, that, if the number of `unknown` entries is too high, the controller has sent on the faulty channel in the request phase. A scenario is possible where two receivers get different majorities on the faulty channel, while the majority on the fault-free channel is `unknown`. On the other side, it is clear in such a situation, that the controller is sending on the fault-free channel in the confirmation phase, and slot reservation must be rejected if there are too little information available to guarantee agreement.

```
for (int i = 0; i < m; i++)
{
  request r_a = majority (i, CHANNEL_A);
  request r_b = majority (i, CHANNEL_B);
  int ch_unknown;
  if (scheduled_for_channel (i, CHANNEL_A))
    ch_unknown = count_unknown (i, CHANNEL_A);
  else
    ch_unknown = count_unknown (i, CHANNEL_B);
  if (r_a != unknown || r_b != unknown)
  {
    if (ch_unknown) > 1)
      result_schedule[i] = false;
    else if (r_a == request_slot
           && r_b == request_slot
           || r_a == request_slot && r_b == unknown
           || r_a == unknown && r_b == request_slot)
      result_schedule[i] = true;
    else
      result_schedule[i] = false;
  }
}
```

4.2 Proof of Correctness

This section shows, that all controllers can agree on a schedule for the extension part.

The set of controllers C is split into the sets D and $\overline{D} = C - D$. If $c \in D$, the controller sends on the fault-free channel in one of the slots in the request phase. The function $rq : C \longmapsto \{\times, 1\}$ becomes $\times$ if a controller has sent a corrupted message (possible for n and f), otherwise 1 (represented in the pseudo code as `request_slot` resp. `request_no_slot`).

The fault-free channel is defined as a function $ch_c(b) = b$ for message b (a request, or a vector in the confirmation phase) of controller c. The faulty channel is a function $cf_c(b) = \times$ if $c \in N_s$ (see section 3.2), or otherwise $cf_c(b) = b$.

At the end of the request phase, any controller c has collected the received messages from each controller in a vector $\mathbf{sg^c}$, where any $\times$ are replaced by 0 (in the pseudo code represented by the constant `unknown`).

Lemma 1 *For any receiver $r \in C$, $\mathbf{sg^r}_i = 1$ if $i \in D$ and i is neither n nor f. The number of correctly received request is at least $\frac{m}{2} - 2$.*

If $i \in D$ and i is fault-free, then $sg_i^r = ch_i(rc(i)) = 1$. This is true for at least $\frac{m}{2} - 2$ controllers, if both f and n are from D.

In the confirmation phase, $\mathbf{sg^c}$ is sent with the appropriate message. By definition the faulty controller f can send a random vector. If f sends another value than the one it has received i, then $\mathbf{sg^f}_i$ is set to -1. Note, that only f is able to send an "non-true" value.

After the confirmation phase has passed, any controller has a matrix rm^c (appears as `request_matrix`in the pseudo-code) containing the vectors received with the messages on the respective channel. If the message of i has been corrupted on the faulty channel, then the row $\mathbf{rm^c}_i$ is set to $\times$. As a consequence, $\frac{m}{2} - 2$ entries $rm_i^c \neq \times$ for the same reason as in lemma 1.

Lemma 2 *For any $r \in C$ the entry $\mathbf{rm^r}_{i,d}$ is 1 or $\times$, if $d \in D$ and $i \neq f$.*

As shown in lemma 1, $\mathbf{sg^i}_d = 1$. If $i \in D$, then $\mathbf{rm^r}_i = cf_i(\mathbf{sg^i})$ is either $(\times, \ldots, \times)$ or $\mathbf{sg^i}$. If $i \notin D$, then $\mathbf{rm^c}_i = \mathbf{sg^i}$.

Lemma 3 *The function `majority(c, channel)` returns the original request for the fault-free, and the original request or `unknown` for the faulty channel, for any fault-free controller `c` $\in C$ and receiver $r \in C$. In the case of the fault-free channel the result is the same for all r.*

For `c`inD, the equation $\mathbf{rm^r}_{i,\mathtt{c}}$ is 1 for all fault-free $i \in D$ or either 1 or $\times$ for all $i \in \overline{D}$ according to lemma 1. If `c` $\in D$ and `channel` is the fault-free channel, then for all $i \in \overline{D}$ (scheduled for the fault-free channel in the confirmation phase), then the same request appears at least $\frac{m}{2} - 2$ times, which is greater or equal 2, if $m \geq 8$ (see line 36 in pseudo code). In the other case for $c \in D$ where `channel` is the faulty channel and the sum of the entries with $\mathbf{rm^r}_{i,\mathtt{c}} \neq \times$ is greater or equal three for all $i \in D$, then the worst case, arises if controller f is from D, which can send a value $\mathbf{sg^f}_{\mathtt{c}} \in -1$. So f is outvoted by the other two controllers, or the condition could not be fulfilled, in which case the return value is `unknown`. If $c \notin D$, and the `channel` is fault-free, then this is the similar situation as for a controller from D on a faulty channel, because $\mathbf{rm^r}_{i,\mathtt{c}}$ can become 0 instead $\times$, which is only possible for the senders f and n. In contrast to the case of a controller from D, it cannot be guaranteed that the function $majority()$ does not return `unknown`, even on the fault-free channel (there may be less than two fault-free controllers receiving a value from c in the request phase). If `channel` is the faulty one, then the values for controller c can be 1, 0 or $\times$ (or -1 from f). Since 0 and $\times$ are handled the same way, this is similar to the case with the fault-free channel. Since the function ch_c is the identity for all c, the results are the same for all controllers.

Theorem 1 (Agreement) *The result schedule is the same for every receiver $r \in C$. In addition, it can be guaranteed for all $c \in D$ that the slot is reserved if it was requested, and not reserved if not requested.*

According to lemma 3, the function `majority(c, channel)` returns the same value for all r if `channel` is fault-free. The value $\mathbf{rm^r}_{i,\mathbf{c}} = 0$ can only appear once (for $i = f$), for $c \in D$ in (lemma 1). As a result, if `majority(c, channel)` returns `unknown` for a $c \in D$, c passes the test in line 61. Since there is always a return value other than `unknown` on the fault-free channel (lemma 3), `result_schedule[c]` is set to the correct value in lines 63–67. If $c \in \overline{D}$, than it is possible, that `majority(c, channel)` returns `unknown`, because at least two entries in the matrix are set to `unknown` on the fault-free channel. This is always the channel c is sending on. Such a situation can become critical (section 4.1 gives an example). In this case, the test in line 61 leads to the default value for any r. Since this happens on the fault-free channel, this situation is visible to any r. In the other cases, there can be a majority on the fault-free channel or the result for both channels is `unknown`. For the first case the test in line 61 passes, because this is not the channel, c is sending on. In either case c is handled the same way as if it would be from D. Controller n is handled as a special case of $c \in \overline{D}$, because $sg_n^r = 0$ for any f, if it is blocked by f. Otherwise it behaves like any other fault-free controller. Controller f behaves either like a normal controller, like n or it sends a contradicting value in the confirmation phase, in which case f is outvoted in `majority()`, or the function returns `unknown`. It is up to controller n to prevent f from sending if necessary.

5 Slot Overloading

There is also a benefit for applications requiring that all cycles have equal length. For this type of application, the extension part can have a fixed length, and be filled with messages by controllers when needed. In contrast to TTP, where the schedule is always statically configured, the schedule in the extension can be chosen dynamically. In contrast to FlexRay, where a dynamically chosen schedule is possible in the dynamic segment, the communication on the channels is always fully protected, so that fault-tolerant operation is assured.

6 Conclusion

A new algorithm has been developed for dynamically reserving time slots in a TDMA-based network protocol. The correctness of the algorithm has been proven. The communication controllers are able to reserve time slots dynamically for spontaneous needs, while the network is fully protected against wrongly sending faulty nodes. With these properties, applications can be handled by the protocol in a more efficient way. Exception handling is an important example, to apply the new slotting scheme. The exchange of information for the exception handling can be done in the extension part in a complete fault-tolerant way.

This is possible without interrupting the normal stream of messages.

References

[BKS02] Bauer, G., Kopetz, H., Steiner, W., “Byzantine Fault Containment in TTP/C” Proceedings 2002 Intl. Workshop on Real-Time LANs in the Internet Age (RTLIA 2002), pages 13–16

[BPG00] M. Peller, J. Berwanger, and R. Griebach. “Byteflight – A New High-Performance Data Bus System for Safety-Related Applications.” BMW AG, EE-211 Development Safety Systems Electronics, 2000

[Fl02] International FlexRay Workshop München 2002, see www.flexray.com

[Fu00] Führer T., et al, “Time-triggered Communication on CAN (Time-triggered CAN – TTCAN)” Proceedings of ICC’2000, Amsterdam, The Netherlands, 2000.

[HT98] Heiner, G., Thurner, T., “Time-Triggered Architecture for Safety-Related Distributed Real-Time Systems in Transportation Systems”, In Digest of Papers, The 28th IEEE Int’l Symp. on Fault-Tolerant Computing Systems, Munich, Germany.

[KG93] Kopetz, H., Gruensteidl, G., “TTP – A time-triggered protocol for fault-tolerant real-time systems.” Proceedings 23rd International Symposium on Fault-Tolerant Computing, pages 524–532, 1993.

[Ko95] Kopetz, H., “The Time-Triggered Approach to Real-Time System Design” In Predictably Dependable Computing Systems (B. Randell, J.-C. Laprie, H. Kopetz and B. Littlewood, Eds.), Basic Research Series, pp.53-66, Springer, 1995.

[Ko97] Kopetz, H., “Real Time Systems – Design Principles for Distributed Embedded Applications”, Kluwer Academic Publishers, 1997

[Ko01] Kopetz, H., “A Comparison of TTP/C and FlexRay” Research Report 10/2001 Institut für Technische Informatik, Technische Universität Wien, Austria

[KBP01] Kopetz, H., Bauer, G., Poledna, S., “Tolerating Arbitrary Node Failures in the Time-Triggered Architecture” SAE 2001 World Congress, March 2001, Detroit, MI, USA

[Mu02] Müller, B., et al, Proceedings 8th International CAN Conference; 2002; Las Vegas, Nv

[PSL80] Pease, M., Shostak, R., Lamport L., “Reaching agreement in the presence of faults.” Journal of the ACM, 27(2):228-234, April 1980.

[Ru97] Rushby, J., “Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms” Proc. DCCA 6, Garmisch, Germany. IEEE Press. pp. (Preprints) 191-210.

[Ru01] Rushby, J., “A Comparison of Bus Architectures for Safety-Critical Embedded Systems” CSL Technical Report, SRI International, 2001

[Te98] Temple, C., “Avoiding the Babbling-Idiot Failure in a Time-Triggered Communication System.” Fault Tolerant Computing Symposium 28, Munich, Germany, June 1998, IEEE Computer Society, pp. 218-227

[Te99] Temple, C., “Enforcing Error Containment in Distributed Time-Triggered Systems: The Bus Guardian Approach” PhD Thesis, Inst. f. Techn. Informatik, Vienna University of Technology.