Reliable Communication in Distributed Computer-Controlled Systems

Luís Miguel Pinho[1], Francisco Vasques[2]

[1] Department of Computer Engineering, ISEP, Polytechnic Institute of Porto,
Rua Dr. António Bernardino Almeida, 431, 4200-072 Porto, Portugal
lpinho@dei.isep.ipp.pt
[2] Department of Mechanical Engineering, FEUP, University of Porto,
Rua Dr. Roberto Frias, 4200-465 Porto, Portugal
vasques@fe.up.pt

Abstract. Controller Area Network (CAN) is a fieldbus network suitable for small-scale Distributed Computer Controlled Systems, being appropriate for transferring short real-time messages. However, CAN networks are also known to present some reliability problems, which can lead to an inconsistent message delivery, thus to an unreliable behaviour of the supported applications. In this paper, a set of atomic multicast protocols for CAN networks is presented, preventing the occurrence of such unreliable behaviours. The proposed protocols explore the CAN synchronous properties to minimise its run-time overhead, and to provide a timely service to the supported applications. The paper also presents conclusions drawn from the implementation of the protocols in the Ada version of Real-Time Linux.

1 Introduction

Currently, there is a trend to incorporate Commercial Off-The-Shelf (COTS) components in the development of Distributed Computer-Controlled Systems (DCCS). Using COTS components as the systems' building blocks provides a cost-effective solution, and at the same time allows for an easy upgrade and maintenance of the system. However, the use of COTS implies that specialised hardware can not be used to guarantee the reliability requirements. As COTS hardware and software do not usually provide the confidence level required by reliable real-time applications, reliability requirements must be guaranteed by a software-based fault-tolerance approach. It is obvious that the reliability of a DCCS lies, in a great extent, in its communication infrastructure, hence, the use of COTS networks poses new problems to the reliability of DCCS.

Controller Area Network (CAN) [1] is a fieldbus network suitable for small-scale DCCS, being appropriate for sending and receiving short real-time messages at speeds up to 1Mbit/sec. Several studies on how to guarantee the real-time requirements of messages in CAN networks are available (e.g. [2] [3]), providing the necessary pre-run-time schedulability conditions for the timing analysis of the supported traffic, even for the case of a network disturbed by temporary errors.

CAN networks have extensive error detection/signalling mechanisms. The node that firstly detects an error sends an Error Frame, which leads to an automatic message retransmission. However, it is known that these mechanisms may fail when an error is detected in the last but one bit of the frame [4]. This problem occurs since the point of time at which a message is taken to be valid is different for the transmitter and the receivers. The message is valid for the transmitter if there is no error until the end of the transmitted frame. If the message is corrupted, a retransmission is triggered according to its priority. For the receiver side, the message is valid if there is no error until the last but one bit of the frame, being the value of the last bit treated as 'do not care'. Thus, a dominant value in the last bit does not lead to an error, in spite of violating the CAN rule stating that the last 7 bits of a frame are all recessive.

Receivers detecting a bit error in the last but one bit of the frame reject the frame and send an Error Frame starting in the following bit (last bit of the frame). As for receivers the last bit of a frame is a ‘do not care’ bit, other receivers may not detect the error and will accept the frame. However, the transmitter re-transmits the frame, as there was an error. As a consequence, some receivers will have an inconsistent message duplicate. The use of sequence numbers in messages can easily solve this problem, but it does not prevent messages from being received in different orders, thus not guaranteeing total order of atomic multicasts. However, if the transmitter fails before being able to successfully retransmit the frame, then some receivers will never receive the frame, which causes an inconsistent message omission. This is a more difficult problem to solve, than in the case of inconsistent message duplicates.

In [4], the probability of message omission and/or duplicates is evaluated, in a reference period of one hour, for a 32 node CAN network, with a network load of approximately 90%. Bit error rates were used ranging from 10^{-4} to 10^{-6}, and node failures per hour of 10^{-3} and 10^{-4}. For inconsistent message duplicates the results obtained were from 2.87×10^{1} to 2.84×10^{3} duplicate messages per hour, while for inconsistent message omissions the results ranged from 3.98×10^{-9} to 2.94×10^{-6}. These values demonstrate that for reliable real-time communications, CAN built-in mechanisms for error recovery and detection are not sufficient.

The following Section discusses the issue of atomic multicasts in CAN, and presents the considered failure assumptions. The proposed atomic multicast protocols are then presented in Section 3 (their specification is presented in Annex), where it is also shown how these protocols fulfil atomic multicast properties. Finally, Section 4 draws some considerations on the implementation of the protocols using the Ada version of Real-Time Linux [5].

2 Atomic Multicasts in CAN

In the considered system model, a hard real-time application is constituted by several tasks, which combined together perform the desired service. These processing tasks are distributed over the nodes of the system. To guarantee the reliability requirements of applications, some of its components must be replicated to tolerate individual faults. In order to have the same consistent state in the replicas, there must be a guarantee that they have the same input messages, and in the same order. That is,

communication mechanisms must be used which ensure the atomic multicast properties. Multicast messages must be delivered by all (or none) of the replicas of a component, and they must be delivered only once. Also, they must be delivered in the same order in all replicas.

Therefore, in order to support hard real-time applications, the communication infrastructure must provide atomic multicast protocols. Based on [6], an atomic multicast has the following properties:

- Validity: If a correct node multicasts a message *m*, then all correct nodes deliver *m*.
- Agreement: If a correct node delivers message *m*, then all correct nodes deliver *m*.
- Integrity: For any message *m*, every correct node delivers *m* at most once, and only if *m* was previously multicast by *sender(m)*.
- Total Order: If correct nodes *p* and *q* both deliver message *m* and *m'*, then *p* delivers *m* before *m'* if and only if *q* delivers *m* before *m'*.

CAN error detection and recovery mechanisms ensure the validity property, since when the sender is correct, all nodes will receive the message. Note that the network can be referred as a fail-consistent bus [7], since there is no possibility for different nodes to receive the message with different values. CAN error detection and recovery mechanisms are not, however, sufficient to guarantee the agreement and integrity properties [4]. In fact, it is possible for a correct node to receive a message not received by some other correct node (inconsistent message omission), and it is also possible that some node receives the same message more than once (inconsistent message duplicate). Total order is also not guaranteed, since new messages can be interleaved with retransmissions of failed messages, inducing nodes to receive messages in different orders.

Thus, the use of CAN to support reliable real-time communications must be carefully evaluated and appropriate mechanisms must be devised. In [4], a set of fault-tolerant broadcast protocols is proposed, which solve the message omission and duplicate problems. However, such protocols do not take full advantage of the CAN synchronous properties, therefore producing a greater run-time overhead under normal operation. For instance, in the best-case (data message with 8 bytes), the overhead of the total order protocol (TOTCAN) is approximately 150%. The problem is that, in order to achieve ordered multicasts, each receiver must re-transmit an ACCEPT message, even if there is no error. Other protocols in the set do not guarantee total order.

Another approach would be to use hardware-based solutions, such as the one described in [8]. This approach is based in a hardware error detector, which automatically retransmits messages that could potentially be omitted in some nodes. Although this approach solves the inconsistent message omission problem of CAN it does not provide solution to total order, as duplicates may occur. In order to achieve order, it is necessary to complement this mechanism with an off-line analysis approach. In this, messages must be separated in hard real-time and soft real-time. Only hard real-time messages have guaranteed worst-case response time inferior to the deadline, but it is necessary to use fixed time slots, off-line adjusting these messages to never compete for the bus.

2.1 Failure Assumptions

The proposed protocols only aim to tolerate network-related faults (including nodes' network interface faults). Application faults are masked by replication. Therefore, for the purpose of simplicity, from now on nodes will be used referring to their network interface. The protocols assumes that:

- System nodes either behave correctly or crash after a given number of failures. This behaviour is guaranteed by the CAN protocol, since in the case of multiple errors, the faulty node is disconnected from the network.
- During a time *T*, greater than the worst-case delivery time of any message, at most one single inconsistent message omission occurs. Considering the existence of 3.98 x 10^{-9} to 2.94 x 10^{-6} inconsistent message omissions per hour [4], the occurrence of a second omission error in a period *T* of, at most, several seconds has an extremely low probability.
- There are no permanent medium faults, such as the partitioning of the network. This type of faults must be masked by appropriate network redundancy schemes.

3 Middleware for Atomic Multicasts in CAN

The provided middleware for atomic multicasts in CAN (Figure 1) presents several protocols, with different failure assumptions and different behaviours in the case of errors. The Filtering layer allows that only nodes registered to receive a particular message stream will process messages related to that stream. This layer also decreases the number of messages in the bus in error situations. The *Unreliable* protocol provides only a simple multicast mechanism, giving no guarantees whatsoever to the streams that use it. The *IMD* protocol provides an atomic multicast that just addresses the inconsistent message duplicate problem. The *2M* protocol provides an atomic multicast addressing both inconsistent message duplicates and omissions, where messages are not delivered in an error situation (a previous version of this protocol is presented in [9]). The *2M-GD* protocol is an improvement of the *2M* protocol, which guarantees the message delivery, if at least one node has correctly received it.

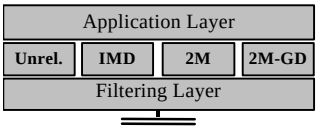
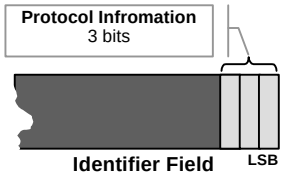


Fig. 1. Middleware for atomic multicasts in CAN

These atomic multicast protocols provide the system engineer with the possibility of trading efficiency by assumption coverage. The *IMD* protocol uses less bandwidth, but can lead to the violation of failure assumptions causing incorrect system behaviour. However, the use of protocols with higher assumption coverage may introduce unnecessary overheads in the system. Thus, it is possible to use different

protocols for different message streams. Hence, streams with higher criticality may use protocols with higher assumption coverage, while streams with smaller criticality may use lighter protocols.

Relying on CAN frames being simultaneously received in every node, the protocols are based in delaying the deliver of a received frame for a bounded time. This behaviour is exploited to achieve atomic multicasts with the minimum number of exchanged messages. The approach is similar to the Δ-protocols [10], where, to obtain order, delivery is delayed for a specific time (Δ). However, in the proposed approach, delays are evaluated on a stream by stream basis, increasing the throughput of the system, since messages are delayed accordingly to their worst-case response times.



Protocol Bits	Message Type	
0 0 0	Data Msg.	*2M-GD* Protocol
0 0 1	Confirmation Msg.	
0 1 0	Retrans. Msg.	
0 1 1	Data Msg.	*2M* Protocol
1 0 0	Confirmation Msg.	
1 0 1	Abort Msg.	
1 1 0	*IMD* Protocol	
1 1 1	*Unreliable* Protocol	

Fig. 2. Identifier field and protocol information

The protocols use the less significant bits of the frame identifier to carry protocol information (Figure 2), identifying the type of each particular message and allowing the simultaneous use of different protocols. As the protocol information uses the less significant bits of the frame identifier, then more critical messages can use any type of protocol (even the Unreliable one), without loosing their criticality.

3.1 *IMD* Protocol

The *IMD* protocol (Figure 3) provides an atomic multicast that just addresses the inconsistent message duplicate problem. In order to guarantee that the duplicates are correctly managed, every node, when receiving a message marks it as unstable, tagging it with a $t_{deliver}$ (current time plus a $d_{deliver}$). If a duplicate is received before $t_{deliver}$, the duplicate is discarded and $t_{deliver}$ is updated (since in a node not receiving the original message $t_{deliver}$ refers to the duplicate).

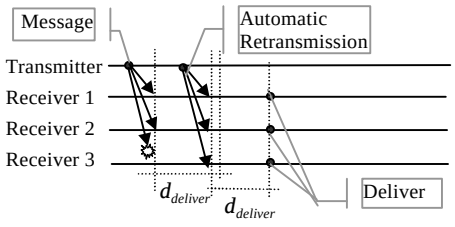


Fig. 3. *IMD* protocol behaviour: inconsistent message duplicate

For the transmitter (if it also delivers the message), since the CAN controller will only acknowledge the transmission when every node has received it correctly, there will be only one received message, even if there are duplicates. This message refers to the last duplicate sent, thus the transmitter can deliver the message after its $d_{deliver}$.

3.2 *2M* Protocol

The *2M* protocol (Figure 4) addresses both the inconsistent message duplicates and inconsistent message omissions guaranteeing that either all or none of the receivers deliver the message. For the latter, not delivering a message is equivalent to a transmitting node crash before sending the message.

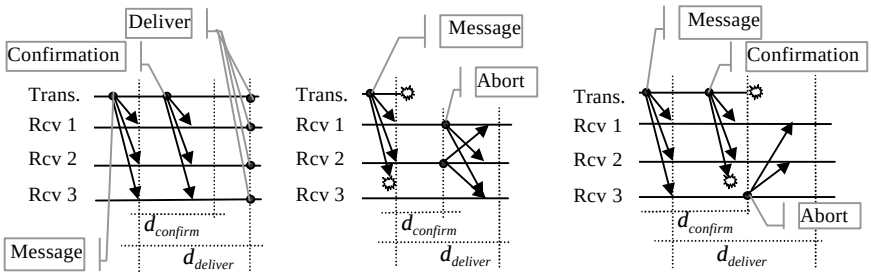


Fig. 4. *2M* protocol behaviour: error free situation (left), inconsistent message omission while sending the message (centre) and while sending the confirmation (right)

In the *2M* protocol, a node wanting to send an atomic multicast transmits the data message, followed by a confirmation message, which carries no data (*2M*: two messages). A receiving node before delivering the message, must receive both the message and the confirmation. If it does not receive the confirmation before $t_{confirm}$, it multicasts the related abort frame. This implies that several aborts can be simultaneously sent (at most one from each receiving node that is interested in that particular message stream). A message is only delivered if the node does not receive any related abort frame until after $t_{deliver}$ (a node receiving the message but not the confirmation does not know if the transmitter has failed while sending the message, or while sending the confirmation).

When a message is received, the node marks it as unstable, tagging it with $t_{confirm}$ and $t_{deliver}$. A node receiving a duplicate message discards it, but updates both $t_{confirm}$ and $t_{deliver}$. As the data message has higher priority than the related confirmation, then all duplicates will be received before the confirmation. Duplicate confirmation messages will always be sent before any abort (confirmation messages have higher priority than related abort messages), thus they will confirm an already confirmed message.

The advantage of the *2M* protocol is that in a fault-free execution there is only one extra frame (without data) per multicast. Only in the case of an error (low probability), there will be more protocol related messages in the bus, inducing a

higher bandwidth utilisation. Note that the transmission of an abort is only necessary if there is a previous failure of the transmitter. Therefore, from the failure assumptions (there is no second inconsistent in the same period *T*), this abort will be free of inconsistent message omissions.

The transmitter can automatically confirm the message, since if it does not fail, every node will correctly deliver the message and the confirmation. The situation is the same as for the *IMD* protocol, since if the transmitter remains correct and delivers the message, then it will re-transmit any failed message.

3.3 *2M-GD* Protocol

The *2M* protocol can be modified to guarantee the delivery of a transmitted message to all nodes, if it is correctly received at least in one node. In the *2M-GD* (guaranteed delivery) protocol (Figure 5), nodes receiving the message but not the confirmation, retransmit the message (instead of an abort). This protocol is however less efficient than the *2M* protocol (in error situations), since messages are retransmitted with the data field. To guarantee order of delivery, it is necessary to use a $t_{deliver\_after\_error}$ to solve inconsistent retransmission duplicates.

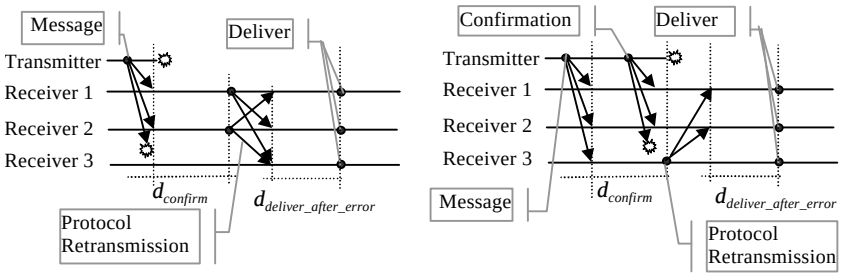


Fig. 5. *2M-GD* protocol behaviour: inconsistent message omission while sending the message (left) and while sending the confirmation (right)

3.4 Guaranteeing Atomic Multicast Properties

Atomic multicast properties (Section 2) are guaranteed by the protocols, since:

- Validity: A correct node is defined as one that does not fail while a multicast is in progress, that is until the multicast is correctly received by every node. As the CAN built-in mechanisms guarantee that any message will be automatically retransmitted, in the case of either a network or a receiving node failure, then the Validity property is guaranteed.
- Agreement: For the *IMD* protocol, if the transmitter does not fail, the CAN built-in mechanisms guarantee that every correct node will receive the message. Thus, they will all deliver it. For the *2M* protocol, a correct node only delivers a data message

after receiving the related confirmation message and knowing that it will not receive any abort from other correct nodes. Therefore it knows that all correct nodes will also deliver the message. In the *2M-GD* protocol the behaviour is the same, except in error situations, where if a correct node receives the message it will retransmit it, thus every correct node will receive and deliver it.

- Integrity: As the delivery of the message is delayed, duplicates are discarded and the Integrity property is guaranteed. On the other side, the CAN built-in mechanisms guarantee that a message is from the actual sender, since an error in the identifier field is detected with a sufficiently high probability [1].
- Total Order: The CAN network guarantees that correct messages are received in the same order by all nodes. However, the existence of duplicates and omissions may preclude messages from being orderly delivered. The use of the $t_{deliver}$ parameter guaranteed the total order of message delivery.

3.5 Evaluation of the Proposed Protocols

It is clear that when using the proposed atomic multicast protocols the worst-case delivery time of messages will be significantly increased. Due to the lack of space, the reader is referred to [11] for a presentation of the model and assumptions for the evaluation of the message streams' delivery time considering the proposed protocols.

Also in [11] an example is presented, allowing to conclude that although worst-case delivery time of message streams is increased, the predictability of message transfers is still guaranteed. It is obvious that the *IMD* protocol is the one that introduces smaller delays, while the *2M-GD* protocol is the one with the higher delays. The system's engineer can use this reasoning to better balance reliability and efficiency in the system. Moreover, the protocols increase network utilisation by less than 50%, since protocols-related retransmissions only occur in inconsistent message omission situations. Although this load increase is still large, it is much smaller than in other approaches, and it is the strictly necessary to cope with inconsistent message omissions using a software-based approach.

4 Implementation Considerations

The concept of using a real-time version of Linux as a platform for Distributed Computer-Controlled Systems is gaining an increased attention. Real-Time Linux provides a solution in which applications with real-time requirements can execute, whilst allowing interconnection with non real-time applications, thus connecting (in a controlled manner) real-time applications to other levels of the system.

Predictability is still an open issue in the (several) Real-Time Linux variants, manly due to both the PC architecture and the support to background Linux applications, and the lack of consolidated studies is impairment for applications with safety requirements. However, Real-Time Linux presents an easy to use solution, which tied together to the current open source movement, makes it a strong contender

for a DCCS platform. Furthermore, small-scale DCCS can be easily supported due to the availability of support to CAN networks.

Therefore, it is pertinent to consider the viability of using Ada and Real-Time Linux together for the programming of Distributed Computer-Controlled Systems. Thus, the middleware has been implemented in a platform of PCs running the Ada version of Real-Time Linux [5], connected through a CAN network (for a more detailed description of implementation details the reader is referred to [9]). Currently, the Ada version of Real-Time Linux provides a tasking kernel underneath the Linux kernel, implementing the low-level tasking mechanisms that are used to support Ada concurrency constructs. However, there is still no compiler targeting this platform, thus these mechanisms must be directly used. Furthermore, the full set of mechanisms is not implemented, lacking, for instance, the capability for interrupt handling.

In the implementation, these two issues came up. First, the available implementation of the Ada executive does not provide the high level mechanisms of Ada for concurrency and to control shared resources (e. g. tasks, protected objects). It only provides the low-level primitives for task and lock managing, which are used to program such mechanisms. Thus, the implementation had to rely on such low-level mechanisms, instead of using the higher level Ada constructs.

Interrupt handling services are also not available. Therefore, an interface to the available Real-Time Linux kernel interrupt services was created. However, as the Real-Time Linux kernel patch (version 1.2) used by the Ada version does not allow handlers to receive the interrupt number, it was not possible to implement a generic mechanism for interrupt handling. Therefore, interrupt handlers are created in a one-by-one basis, and are only used to wakeup a task, which is the proper handler.

It is, undoubtedly, possible to build DCCS applications using the Ada version of Real-Time Linux. However, although the presented problems were solved in this specific implementation, the attained solutions make no use of Ada's advantages for real-time programming. In this case, programming is as error prone as with other languages (e. g. C), not taking advantage of Ada's full programming power. Furthermore, the solutions are specific to the considered problem, thus they may not be appropriate for other applications.

It is important for Ada to be widely considered suitable for this platform, since it is a platform that is increasingly being used for real-time applications, a domain where Ada (still) has some influence. However, the lack of proper tools is impairment for Ada, since it is difficult to justify its use in preference to other languages.

5 Conclusions

This paper proposes a set of protocols for the support of atomic multicasts in CAN networks, providing both a timely and reliable service to the supported applications. In the proposed approach, atomic multicasts are guaranteed through the transmission of just an extra message (without data) for each message that must tolerate inconsistent message omissions. Only in case of an inconsistent message omission (low probability) there will be more protocol-related retransmissions. Inconsistent message duplicates are solved with a protocol that does not require extra

transmissions, guaranteeing total order. Moreover, atomic multicast properties are achieved without more overheads than the strictly needed for a reliable multicast. These protocols explore the CAN synchronous properties to minimise its run-time overhead, and thus to provide a reliable and timely service to the supported applications.

The paper also draws some considerations on the platform used for implementation: PCs, running the Ada version of Real-Time Linux, connected through a CAN network. It is noted that, although this platform may be considered for Distributed Computer-Controlled Systems, currently the use of Ada does not present a significant advantage, and it is difficult to justify its use in preference to other languages.

Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments. This work was partially supported by FCT (project DEAR-COTS 14187/98).

References

1. ISO 11898. (1993). Road Vehicle - Interchange of Digital Information - Controller Area Network (CAN) for High-Speed Communication. ISO.
2. Tindell, K., Burns, A. and Wellings, A. (1995). Calculating Controller Area Network (CAN) Message Response Time. In *Control Engineering Practice*, Vol. 3, No. 8, pp. 1163-1169.
3. Pinho, L., Vasques, F. and Tovar, E. (2000). Integrating inaccessibility in response time analysis of CAN networks. In *Proceedings of the 3rd IEEE International Workshop on Factory Communication Systems*, pages 77–84, Porto, Portugal, September 2000.
4. Rufino, J., Veríssimo, P., Arroz, G., Almeida, C. and Rodrigues, L. (1998). Fault-Tolerant Broadcasts in CAN. In *Proc. of the 28th Symposium on Fault-Tolerant Computing*, Munich, Germany, June 1998.
5. Shen, H. and Baker, T. (1999). A Linux Kernel Module Implementation of Restricted Ada Tasking. In *Proc. 9th International Real-Time Ada Workshop*, Ada Letters, Vol. XIX, N. 2, June 1999.
6. Hadzilacos, V. and Toueg, S. (1993). Fault-Tolerant Broadcasts and Related Problems. In Mullender, S. (Ed.), *Distributed Systems*, 2nd Ed., Addison-Wesley, 1993.
7. Powell, D. (1992). Failure Mode Assumptions and Assumption Coverage. In *Proc. of the 22nd Symposium on Fault-Tolerant Computing*, Boston, USA, July 1992.
8. Kaiser, J. and Livani, M. Achieving Fault-Tolerant Ordered Broadcasts in CAN. In *Proc. of the 3rd European Dependable Computing Conference*, Prague, Czech Republic, September 1999, pp. 351-363
9. Pinho, L., Vasques, F. and Ferreira, L. (2000). Programming Atomic Multicasts in CAN. In *Proc. of the 10th International Real-Time Ada Workshop*, Avila, Spain, September 2000.
10. Cristian, F., Aghili, H., Strong, R. and Dolev, D. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *Information and Control*, 118:1, 1995.
11. Pinho, L. and Vasques, F. Timing Analysis of Reliable Real-Time Communication in CAN Networks. Technical Report HURRAY-TR-0026, December 2000.

Annex: Protocol Specifications

IMD Protocol

Transmitter

```
1:    atomic_multicast (id, data):
2:          send (id, data)

3:    when sent_confirmed (id, data):  -- if it is registered for this message
4:          received_messages_set := received_messages_set ∪ msg(id,data)
5:          t_deliver(id) := clock + d_deliver(id)

6:    deliver:
7:          for all id in received_messages_set loop
8:                if t_deliver(id) < clock then
9:                      state(id) := delivered
10:               end if
11:         end loop
```

Receiver

```
1:    when receive (id, data):
2:          if id ∉ received_messages_set then
3:                received_messages_set := received_messages_set ∪ msg(id,data)
4:                state(id) := unstable
5:          end if
6:          t_deliver(id) := clock + d_deliver(id)

7:    deliver:
8:          for all id in received_messages_set loop
9:                if state(id) = unstable and t_deliver(id) < clock then
10:                     state(id) := delivered
11:               end if
12:         end loop
```

2M Protocol

Transmitter

```
1:    atomic_multicast (id, data):
2:          send (id, message, data)
3:          send (id, confirmation)

4:    when sent_confirmed (id, message, data):
5:          received_messages_set := received_messages_set ∪ msg(id,data)
6:          state(id) := confirmed
7:          t_deliver(id) := clock + d_deliver(id)

8:    deliver:
9:          for all id in received_messages_set loop
10:               if state(id) = confirmed and t_deliver(id) < clock then
11:                     state(id) := delivered
12:               end if
13:         end loop
```

Receiver

```
1:    when receive (id, type, data):
2:          if type = message then
3:                if id ∉ received_messages_set then
4:                      received_messages_set := received_messages_set ∪ msg(id,data)
5:                      state(id) := unstable
```

```
6:                end if
7:                t_deliver(id) := clock + d_deliver(id)        -- duplicate update
8:                t_confirm(id) := clock + d_confirm(id)
9:          elsif type = confirmation then
10:               state(id) := confirmed
11:         elsif type = abort then
12:               if id Î received_messages_set then
13:                     received_messages_set := received_messages_set – msg(id)
14:               end if
15:         end if

16:   deliver:
17:         for all id in received_messages_set loop
18:               if state(id) = confirmed and t_deliver(id) < clock then
19:                     state(id) := delivered
20:               elsif state(id) = unstable and t_confirm(id) < clock then
21:                     send (id, abort)
22:                     received_messages_set := received_messages_set – msg(id)
23:               end if
24:         end loop
```

2M-GD Protocol

Transmitter

```
1:    atomic_multicast (id, data):
2:          send (id, message, data)
3:          send (id, confirmation)

4:    when sent_confirmed (id, message, data):
5:          received_messages_set := received_messages_set ∪ msg(id,data)
6:          state(id) := confirmed
7:          t_deliver(id) := clock + d_deliver(id)

8:    deliver:
9:          for all id in received_messages_set loop
10:               if state(id) = confirmed and        t_deliver(id) < clock then
11:                     state(id) := delivered
12:               end if
13:         end loop
```

Receiver

```
1:    when receive (id, type, data):
2:          if type = message then
3:                if id ∉ received_messages_set then
4:                      received_messages_set := received_messages_set ∪ msg(id,data)
5:                      state(id) := unstable
6:                end if
7:                t_deliver(id) := clock + d_deliver(id)
8:                t_confirm(id) := clock + d_confirm(id)
9:          elsif type = confirmation then
10:               state(id) := confirmed
11:         elsif type = retransmission then
12:               if id ∉ received_messages_set then
13:                     received_messages_set := received_messages_set ∪ msg(id,data)
14:               end if
15:                     state(id) := confirmed
16:                     t_deliver(id) := clock + d_deliver_after_error(id)
17:         end if

18:   deliver:
19:         for all id in received_messages_set loop
20:               if state(id) = confirmed and t_deliver(id) < clock then
21:                     state(id) := delivered
22:               elsif state(id) = unstable and t_confirm(id) < clock then
23:                     send (id, retransmission, data)
24:               end if
25:         end loop

26:   when sent_confirmed (id, retransmission, data):  -- if message was retransmitted
27:         state(id) := confirmed
28:         t_deliver(id) := clock + d_deliver_after_error(id)
```