Reconstruction and verification of group membership protocols

Muhammad Atif, Sjoerd Cranen, MohammadReza Mousavi

Eindhoven University of Technology

Abstract. In this paper, we present a process-algebraic specification of group membership protocols specified in [Y. Amir, D. Dolev, S. Kramer and D. Malki, Membership Algorithms for Multicast Communication Groups, Springer-Verlag, 1992]. In order to formalise the protocol and its properties we disambiguate the informal specification provided by the paper. This requires trying different possible interpretations in the formal model and checking the consistency of the assumption and formally verifying the correctness properties. We thus present a formal reconstruction of the membership algorithms and model-check our reconstruction.

1 Introduction

Group membership protocols [CKV01] form an important foundation for distributed systems, allowing the different nodes in such a system to maintain a consistent view on which nodes are currently active. Naturally, a group membership protocol must be able to deal with nodes deliberately entering and leaving the current configuration set. However, it is also important for a group membership protocol to be fault-tolerant; due to faulty behaviour of nodes or of the network, nodes might also be considered inactive.

Pioneering algorithms among group membership protocols are the ones presented by Amir et al. in [ADKM92a]; these were the first group membership protocols with support for partitioning [CKV01]. Our initial intention was to formally specify and verify the algorithms presented in [ADKM92a]. However, we soon found numerous ambiguities making a straightforward formal specification of these algorithms virtually impossible. Since our attempts to communicate with the authors of [ADKM92a] were not successful, we were forced to reconstruct these algorithms by systematically enumerating, trying and verifying different possible interpretation of these algorithms. Hence, in this paper, we present two group membership algorithms inspired by [ADKM92a], formally specify them in the process algebra mCRL2 [GMA+09] and prove them correct with respect to the requirements formalized in terms of monitor processes. In our presentation, we show how we have come up with them and why some other (seemingly simpler and more faithful) interpretations of the algorithms in [ADKM92a] violate the basic properties required for a group membership algorithm.

The algorithms presented in [ADKM92a] are part of a larger distributed system framework called Transis [ADKM92c]. They essentially rely on Transis in

that they use some services provided by Transis and also augment its functionality by providing facilities for group membership. Hence, in our formalization, we not only build a formal model of these algorithms, but also develop a formal layer of abstraction presenting the behavior of Transis.

The rest of this paper is organised as follows. In Section 2 the architecture of the distributed system and protocols studied in this paper is presented. In Section 3 an overview of the process algebra mCRL2 used for our formalization is given. An informal description of Transis and an excerpt of its formal specification is provided in Section 4. Section 5 is dedicated to the membership algorithms and their formalisation. In Section 6 the requirements on the algorithms are defined, formalised, and verified on the formal models. A brief overview of the related work is given in Section 7. The paper is concluded in Section 8.

2 Architecture of the Distributed System

In short, the goal of the membership protocol is to keep a consistent view of the group among the member nodes by handling *faults*, i.e., failing nodes or communication channels (to be excluded from the group's view), and *joins*, nodes joining the network (to be added to the group's view).

The original paper [ADKM92a] specifies the membership protocol as a combination of a fault-handling mechanism and a join-handling mechanism, running on top of a communication subsystem called *Transis*. Both mechanisms are first explained separately, and then combined (only requiring modification of the join mechanism) to form the full membership protocol.

Essentially, the feature of Transis used in the specification of membership algorithms is a service called *causal multicast*. This service broadcasts messages to a group of recipients, and guarantees that the delivery of messages at their destination preserves a certain ordering, the *causal order* (a formal description of this order is given in the remainder of this paper). Preserving this order involves a directed acyclic graph, or DAG, of which the nodes are messages and the edges are direct causal orderings. This DAG is constructed locally at every node. As each node is supposed to construct the exact same graph, the local copies are often collectively referred to as 'the DAG'.

The first thing we would like to establish is what status the membership protocol has. Is it part of Transis, does it operate on top of Transis, or is it an independently operating application? The following quotes from [ADKM92a] seem to indicate that none of the above are exactly true.

> The membership protocol operates above the Transis communication layer, such that message arrival order within the protocol preserves causality. [p. 295]

From [ADKM92b,ADKM92a,ADKM92c] we deduce that messages are inserted into the DAG by Transis on arrival. Messages are delivered to the upper level by Transis when they become deliverable in the DAG. The first quote above suggests that the membership protocol therefore has to wait until a message

becomes deliverable in the DAG. The following quote however implies that the protocol must be able to handle messages *before* they are delivered.

> When a FA message is inserted into the DAG, the faults algorithm marks it *nondeliverable*. [p. 298]

The next quote reveals the cause of the confusion: the boundary between Transis and the membership protocol is not a strict one.

> The Transis communication sub-system [...] *delivers* the messages to the upper level. The services use different delivery criteria on the messages in the DAG. In some cases, the membership protocol interferes with the delivery of messages, as we shall see below. [p. 296]

We choose to resolve the above inconsistency by ignoring the first quote, which suggests a strict separation between Transis and the protocols and assumes causal ordering on the arrival of messages. As we will see in the rest of this document, it seems likely that the membership protocol uses Transis to keep track of the causal order of messages, but does in fact deal with messages that arrive out of causal order.

In order to make the formalisation of the protocol easier, we would like to define an interface between Transis and the membership protocol. As mentioned before, the membership protocol interferes with the normal operation of Transis. In an attempt to formalise this interference, we assume existence of the following mechanisms:

- Transis keeps a list of senders from which it will refuse to receive messages. The membership protocol may alter this list at any given time. We need this mechanism because the membership protocol needs to "instruct Transis to disallow any message from *f_set* to enter the DAG" (p. 299, fig. 3).
- Transis distinguishes between membership protocol messages and other messages. Since the protocol messages arrive at the membership protocol, but user messages don't, this must be the case.
- The membership protocol receives the IDs of the senders of received messages, so that it may recognise nodes that are not in the current configuration set. We need this to identify 'foreign messages' (p. 306, fig. 6).
- The membership protocol receives protocol-specific messages from Transis upon reception from the channel. As discussed above, the messages must be handled on arrival, not on delivery.
- The membership protocol may assign markings to messages in the DAG. The behaviour of Transis is defined in terms of these markings. This marking is done on in figures 3, 5 and 7 on pages 299, 305 and 307 respectively.

The above mechanisms provide all the functionality that the membership protocol needs to influence the behaviour of Transis. We have chosen these mechanisms such that Transis' behaviour can be defined independently of the membership protocol. The only exception to this rule is that Transis distinguishes membership protocol messages from other messages.

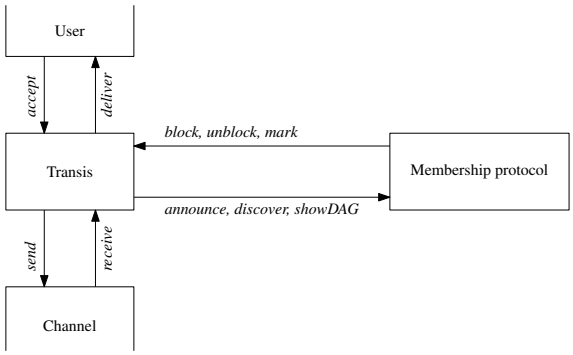

Fig. 1. Communication scheme for a single participant.

Figure 1 shows the context of the membership protocol on a single node. The membership protocol itself only communicates with Transis, which in turn may accept and deliver messages from and to the user (the application layer), and which may send and receive messages to and from the broadcast channel (the physical layer).

3 mCRL2

mCRL2 [GMA+09] (micro Common Representation Language 2) is a formal specification language for modeling, validation and verification of concurrent systems and protocols. It is based on process algebra. We apply it due to the available expertise and this toolset has already been applied successfully for the behavioural analysis of various protocols and distributed systems [FGP+04,vdPE03]. Its accompanying toolset supports different tools, which are used for linearisation (a simplified form of process suitable for analysis and state space generation), simulation, reduction and state-space generation (for visualization and analysis).

4 Transis

As mentioned before, we provide a model of Transis in order to be able to model the membership protocol. We only model those aspects of Transis that the protocol uses.

We define a set $\mathcal{P}$ of *participants* of the membership protocol, each of which may be identified by some unique id, and a set $\mathcal{M}$ of *messages* that may be broadcast by these participants. Transis provides reliable communication, so whenever a message is broadcast, it will eventually be received by every non-faulty member of the group. Messages are uniquely identified by their sender and a counter that indicates how many messages the sender has sent before the

current one. Messages consist of a header for identification, and a payload part. The payload denotes the type of the messages, e.g., protocol messages denoting failure or join attempts, or user messages, denoted by the payload USER.

4.1 Causal delivery order

Let, for some $p \in \mathcal{P}$ and $m \in \mathcal{M}$, $\mathsf{send}_p(m)$ denote the event that node p sends a message m. Likewise, $\mathsf{recv}_p(m)$ denotes the event that node p receives a message m. In [Lam78], Lamport defines a partial ordering on these events, assuming that events occurring at a single node are totally ordered using $\prec_p$. Inspired by Lamport's causal order, Amir et al. introduce the following *causal delivery order* on messages.

Definition 1 (Causal delivery order). *The* causal delivery order *is a partial ordering on the set $\mathcal{M}$ of messages, such that $m \in \mathcal{M}$ is said to* cause $m' \in \mathcal{M}$*, denoted $m \longrightarrow m'$, if and only if for some $p \in \mathcal{P}$ we have either*

- $\mathsf{recv}_p(m) \prec_p \mathsf{send}_p(m')$*, or*
- $\mathsf{send}_p(m) \prec_p \mathsf{send}_p(m')$*.*

If $\neg(m \longrightarrow m')$ and $\neg(m' \longrightarrow m)$, then m and m' are said to be concurrent*.*

Note that the causal delivery order is a transitive relation. The causal delivery order can be depicted as a DAG in which M is the set of nodes, and $\longrightarrow$ defines the set of edges such that there is an edge from m to m' if and only if $m \longrightarrow m'$ and there is no m'' such that $m \longrightarrow m''$ and $m'' \longrightarrow m'$.

In Transis, acknowledgements to the receipt of messages are sent along with new messages, thus ensuring that every node can derive the order at which events have taken place at a certain node. Therefore, each node can construct the above DAG independently, even though local views may be incomplete at any time due to delays in the network.

4.2 Pseudo code

Before we give a formal specification of the part of Transis that is needed to enable the membership protocol, we first describe it informally. Figure 2 gives the pseudo-code snippet of the Transis' response to certain events. In this pseudo-code, the following events may occur:

- *block*(q)*, unblock*(q) The membership protocol wishes to allow or disallow messages from $q \in \mathcal{P}$ to enter the DAG.
- *mark*(m, k) The membership protocol wants to mark $m \in \mathcal{M}$ with marking k.
- *accept*(p) The user wishes to broadcast payload p.
- *receive*(q, m) The communication channel has received message m from sender $q \in \mathcal{P}$.

Transis itself produces the 'discover', 'announce', 'send' and 'deliver' events, although for the sake of brevity the latter is not shown in the pseudo code; where the pseudo code says 'deliver messages', Transis uses criteria on its DAG to decide which messages may be delivered, if any. Out of these messages, it delivers one (i.e. it produces a 'deliver' event) and then repeats this checking and sending until no further message may be delivered. This specification guarantees that messages are delivered as soon as they become deliverable. In order to guarantee progress, we assume that deliverable messages are immediately delivered to the application layer.

Events are assumed to be processed in a queue-like manner: they do not get lost, and are processed in order of arrival.

TRANSIS(id, c, DAG, B)

```
on block(q)
    B = B ∪ {q}
on unblock(q)
    B = B \ {q}
on mark(m, k)
    update DAG
    deliver messages
on accept(p)
    send(q, ⟨⟨id, c⟩, p⟩)
    c = c + 1
on receive(q, m)
    if q ∉ B then
        if isUSER(m) then
            discover(q)
            insert m with marking deliverable into DAG
            deliver messages
        else
            announce(q, payload(m))
            receive marking k for m
            insert m with marking k into DAG
            deliver messages
```

Fig. 2. Pseudo-code for the Transis process.

5 The membership protocol

In this section we attempt to construct an accurate description of the membership protocol from [ADKM92a]. The goal of this protocol is to keep track of the *Current Configuration Set (CCS)* of the node it is run on, and therefore needs to deal with nodes leaving and entering the network. The paper [ADKM92a] starts with describing a protocol that deals with faults only, and then adapts it to also allow for participants joining a CCS. We follow their approach by describing the fault handling protocol in section 5.1 and the full membership protocol in section 5.2.

5.1 Faults protocol

The first protocol described in [ADKM92a] and re-constructed and verified here merely deals with faulty participants, i.e., participants that—for whatever

reason—do not communicate with other participants any longer. In [ADKM92a], the following assumptions have been made:

- Messages are not delayed indefinitely.
- Communication breaks can be detected.
- Initially, all participants in CCS know the contents of CCS.

We note that the CCS may differ per node, as the network may be partitioned.

Pseudo-code The idea behind the faults protocol is quite simple. Once a communication break with another participant is detected, this information is shared with the other participants by means of an FA (fault) message. Each node q has its own set F of participants that it thinks have failed, and a list $LastF$ that contains, for each participant q', the set of participants that q knows are in the set F of q'. When all participants have the same F set, then this will eventually cause $LastF$ to contain only sets equal to F, and participants can then locally conclude that consensus is reached.

Whenever communication breaks with q or a FA message is received:

- if communication breaks with q:
  $f\_set = \{q\}$
- if receive message < FA, f_set > from r:
  LAST[r] = LAST[r] $\cup f\_set$
  mark the FA message non-deliverable
- if ($f\_set \not\subseteq F$)
  $F = F \cup f\_set$
  instruct Transis to disallow any message from f_set to enter[a] *the DAG.*
  broadcast < FA, F >
- if $\forall q \in (CCS \setminus F)$ $LAST[q] = F$
  assent to F
  mark all the FA messages of F deliverable
  deliver FA messages last in their concurrency sets
  $F = \emptyset$
  LAST = $\perp$

[a] unless it is already followed by another message in the DAG and required for recovery

Whenever delivering a FA message < FA, f_set > :

$CCS = CCS \setminus f\_set$
$\forall q \in CCS$: set $Expected[q]$ to the message index following the last message delivered from q.

Fig. 3. The faults protocol description, taken from [ADKM92a].

The original pseudo-code for the faults protocol is shown in Figure 3 and consists of four *if*-statements. However, the semantics of those four *if*-statements seems to be different; the first two are mutually exclusive and distinguish the type of event that occurred, while (for the reasons given below) the last two statements are to be executed sequentially. Our interpretation of the first and the last *if*-statements are respectively presented in the pseudo-code in Figure 4 and Figure 5. Note that the pseudo-code in Figure 4 invokes the pseudo-code of Figure 5.

FaultsProtocol

on commbreak(q)
    $Last, F =$ FPCheck
on announce(q, FA(f_set))
    $Last[q] = Last[q] \cup f\_set$
    mark(id,*undeliverable*)
    $Last, F =$ FPCheck

Fig. 4. The interpretation of the pseudo-code in Figure 3.

It is unlikely that the authors mean the first two statements to be executed sequentially, as both statements introduce a definition for *f_set*. The original paper states that the pseudo-code gets executed "whenever communication breaks with q or a [sic] FA message is received".

We assume that events are handled one by one, i.e., no two events are processed simultaneously.

FPCheck

if $f \not\subseteq F$ **then**
    $F = F \cup f$
    block(f)
    accept(FA(F))
if $\forall q \in (CCS \setminus F) : Last[q] = F$ **then**
    Mark all FA(*f_set*) messages *slow* that have $f\_set \subseteq F$
    return $\perp, \emptyset$
else
    return $Last, F$

Fig. 5. The interpretation of the pseudo-code in in Figure 3.

The interpretation of the last two *if*-statements in figure 5 makes sense because the first of these *if* statements may influence the truth value of the condition of the second.

5.2 Full membership protocol

The full membership protocol extends the faults protocol by adding a mechanism to deal with participants that should join the network. Its interesting features are the ability to deal with partitioning and joining up of partitions and its fully symmetric structure.

As explained in the remainder of this section, the full membership protocol is described using three 'stages'. The intuition behind the stages is that the response to protocol messages is different in every stage. Although in [ADKM92a]

it is nowhere made clear that the faults protocol is being run together with the so-called 'modified join protocol', we can safely assume that this is the case as the modified join protocol only deals with faults during a joining operation.

However, we cannot run the faults protocol without modification: the original pseudo-code states that the faults protocol is executed "Whenever communication breaks with q or a [sic] FA message is received". But when the join protocol is running, FA messages are being dealt with by the join protocol. We therefore assume that the faults protocol is also treated as a 'stage'; when no protocol is active, then the first message to arrive determines which stage the membership protocol moves to. The stages of the protocol are shown in figure 6.

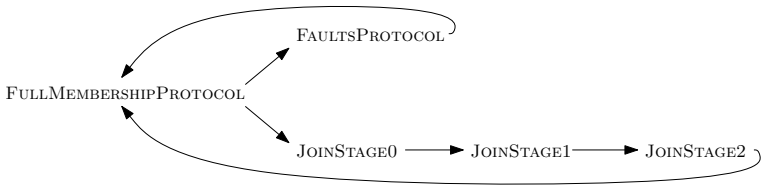

Fig. 6. Different stages of the membership protocol.

In our formal model, the FaultsProtocol stage is only defined implicitly; the FullMembershipProtocol handles faults, and JoinStage0 is never entered between detection of a fault and reaching consensus on the failed participants.

FullMembershipProtocol

on
commbreak(q), $q \in CCS$
  $Last, F =$
  FPCheck($Last, F, \{q\}$)
on discover(q), $q \notin CCS$
  if $F = \emptyset$ **then**
    Broadcast AJ(CCS)
    Set a timer
    $J = CCS$
    JoinStage1

on announce(q, FA(f_set)), $q \in CCS$
  $Last[q] = Last[q] \cup f\_set$
  mark(id,*undeliverable*)
  $Last, F =$
  FPCheck($Last, F, f\_set$)
on announce(q, AJ(j_set)), $q \in CCS$
  if $F = \emptyset$ **then**
    Set a timer
    $J = CCS$
    JoinStage1

Fig. 7. The full membership protocol changes its behaviour dynamically.

JoinStage1

on timer expiration or commbreak(q), $q \in CCS$ BroadcastJoin JoinStage2	**on** announce(q, AJ(j_set)) $J = J \cup j\_set$ **on** announce(q, JOIN(j_set, f_set)) Incorporate-Join(q, j_set, f_set)
on announce(q, FA(f_set)), $q \in CCS$ IncorporateFA(q, f_set, f_J, f_F)	

Fig. 8. The full membership protocol changes its behaviour dynamically.

JoinStage2

on announce(q, JOIN(j_set, f_set))
if $j\_set \not\subseteq J \vee f\_set \not\subseteq F_{\text{before}}$ **then**
Incorporate-Join(q, j_set, f_set)
BroadcastJoin
else
Incorporate-Join(q, j_set, f_set)
JPCheck
on commbreak(q), $q \in CCS$
$F_{\text{after}} = F_{\text{after}} \cup \{q\}$
BroadcastFA
JPCheck
	on announce(q, FA(f_set)), $q \in CCS$
	if $f_J = J \wedge f_F = F_{\text{before}}$ **then**
	Incorpo-rateFA(q, f_set, f_J, f_F, after)
	BroadcastFA
	else if $f\_set \not\subseteq F_{\text{before}}$ **then**
	Incorpo-rateFA(q, f_set, f_J, f_F, before)
	BroadcastJOIN
	else
	$LastF[q] = LastF[q] \cup f\_set$
	JPCheck

Fig. 9. The second stage of the join protocol.

Pseudo-code After a message from a participant outside the *CCS* is received, or after an AJ message from a participant in *CCS* is received, the join protocol proceeds in two phases. The first phase is entered directly after sending an AJ message communicating the current *CCS*, as shown in figure 7.

In the first phase, shown in figure 8, the protocol collects AJ messages from outside the *CCS*. After some time, a timer expires and the second phase is entered. If a communication break occurs before this occasion, then the second phase is entered early.

In the second phase, shown in figure 9, the participants communicate all the information from the received AJ messages until consensus is reached (by executing the code in figure 10) about which participants are connected. Note that it may occur that no AJ message was received by any of the participants before their timers expired, in which case the protocol terminates without having caused a configuration change.

The pseudo code uses some macros given in figure 11, which are a direct translation of the specification given in [ADKM92a].

The pseudo code given in [ADKM92a] for the join protocol shows ambiguities similar to those in the faults protocol. We restructure the pseudo code in a similar

fashion as before, in order to be able to convert this code to a formal model later on.

In figure 8 of that paper, we change $j = \langle f\_set, f_b, f_a \rangle$ to read $j = \langle f\_set, j\_set \rangle$ and $CCS = j\_set \setminus f_b$ to read $CCS = j\_set \setminus f\_set$, because there is no message that fits the given description.

In the original specification, a second DAG (the *completion DAG*) is employed to ensure that messages are not lost when they are discarded. In our model, we keep all blocked messages in a buffer, thus keeping the completion DAG empty (and therefore irrelevant).

We remark that a small change could make the protocol operate more efficiently. As it is now, the first AJ message that is sent only contains the own CCS, so all participants in CCS will go to the next stage without knowing who wishes to join. When the timer expires too soon, it might happen that the participants move to stage 2 without having received an AJ message from the participant that wishes to join, and consensus will be reached on the old CCS.

This behaviour can be avoided by sending an AJ($CCS + \{q\}$) and initializing $J = CCS + \{q\}$ when receiving a foreign message, and by updating $J = CCS + j\_set$ when receiving an AJ(j_set) message in figure 7.

Another odd aspect of the specification is that the first *if*-statement in figure 10 may cause the protocol to assent to the same (and possibly even empty) faults set repeatedly. Even though it does not affect the behaviour of the protocol, it seems strange that this path in the pseudo code is executed unnecessarily often.

6 Verification

The purpose of the membership protocols in [ADKM92a] is to guarantee two properties, namely *consensus* and *virtual synchrony*. In this section we reconstruct what these notions mean, as they are not formally defined in the aforementioned paper. We then formalize these requirements in terms of monitor processes and then verify that these monitors do not detect errors in our formalisations of the membership protocols.

JPCheck

if $\forall_{q \in CCS \setminus F_{\text{before}}}\ LastF[q] \supseteq F_{\text{before}} \cap CCS$ **then**
  assent to $F_{\text{before}} \cap CCS$
  mark all FA(f_set) deliverable that have $f\_set \subseteq F_{\text{before}} \cap CCS$
if $\forall_{q \in J \setminus (F_{\text{before}} \cup F_{\text{after}})}\ LastJ[q] = \langle J, F_{\text{before}} \rangle \wedge LastF[q] \supseteq F_{\text{after}}$ **then**
  assent to $\langle J, F_{\text{before}}, F_{\text{after}} \rangle$
  mark all JOIN(J, F_{before}) deliverable
  mark all FA(f_set) deliverable that have $f\_set \subseteq F_{\text{after}}$

Fig. 10. The consensus check for the join protocol.

BroadcastJoin(j_set, f_set)
$F_{\text{before}} = F_{\text{before}} \cup F_{\text{after}}$ $F_{\text{after}} = \emptyset$ broadcast JOIN(J, F_{before}) mark all JOIN(j_set, f_set) rejected if $j\_set \neq J$ or $f\_set \neq F_{\text{before}}$

IncorporateJoin(q, j_set, f_set)
$J = J \cup j\_set$ $LastJ[q] = \langle j\_set, f\_set \rangle$ $F_{\text{before}} = F_{\text{before}} \cup f\_set$ $LastF[q] = LastF[q] \cup f\_set$ mark the message undeliverable

IncorporateFA(q, f_set, f_J, f_F, x)
$F_x = F_x \cup f\_set$ $LastF[q] = LastF[q] \cup f\_set$ block(F_x)

Fig. 11. Macros used in the join protocol.

As explained above, the full membership protocol can be seen as a composition of the faults protocol and the join protocol, which then operate mutually exclusively. In order to keep the verification problem tractable, we verify both protocols separately.

6.1 Consensus

The protocols presented in [ADKM92a] are claimed to be correct with respect to a property called *consensus*. The meaning of this property is formulated as follows:

> **P.1** Maintain the CCS in consensus among the set of machines that are connected throughout the activation of the membership protocol. [p. 300]

Regrettably, this property is not referred to again in the rest of the paper. It is, however, stated two pages earlier that in the faults protocol "different machines need not assent to the same F set". Because we know that assenting to an F set means that the local view on the current configuration set changes, this must mean that at certain points in time it is allowed for two participants in the same network to have different views on the CCS. Hence, we can deduce that some forms of incoherence are allowed.

Indeed we can think of a simple scenario in which two nodes assent to different F sets, as depicted in the message sequence chart in figure 6.1. Here we see that p_0 believes for a short period that p_1 is still alive, while p_2 assents to the failure of p_1 and p_3 directly. Thus, we need to find an interpretation of consensus that is consistent with these intermittent incoherencies. We therefore propose the following, more precise definition of consensus.

Definition 2 (Consensus). *A participant $p \in \mathcal{P}$ is said to be* unstable*, denoted* $\mathsf{unstable}(p)$*, if a membership protocol is executing (i.e. $F \cup J \neq \emptyset$). A membership protocol preserves the property of* consensus *if at all times we have that*

$$\forall_{p,q \in \mathcal{P}} \ \ (q \in CCS_p \wedge \neg\,(\mathsf{unstable}(p) \vee \mathsf{unstable}(q))) \Rightarrow CCS_p = CCS_q$$

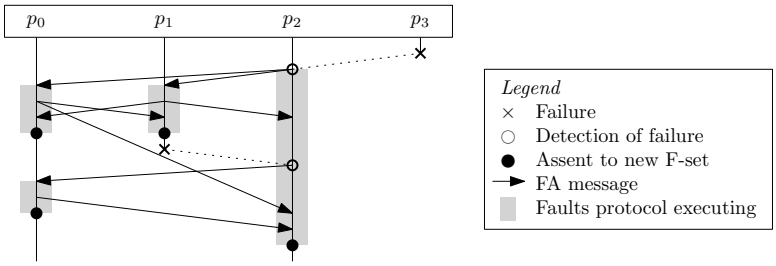

Fig. 12. A simple case in which two nodes assent to different sets of failed nodes.

where CCS_p denotes the local view of $p \in \mathcal{P}$ on the current configuration set.

Note that this definition allows for the existence of cliques (due to partitioning) and for nodes not assenting to the same fault or join sets. In particular, the situation in Figure 6.1 is allowed.

6.2 Virtual synchrony

Another aspect informally described in [ADKM92a] is the notion of *virtual synchrony*. In the introduction, the following descriptions can be found.

> Virtual-synchrony. It guarantees that members of the same configuration receive the same set of messages between every pair of configuration changes. [p. 293]

The technical report [ADKM92b] describing the membership protocols refers to [SBS91] as the source of their notion of virtual synchrony. However, the latter paper only defines virtual synchrony in terms of processes 'observing consistent delivery orders', and not in terms of delivering messages in between configuration changes, like in the second quote above. This seems to indicate that all messages should be sent in Transis' causal multicast mode, but that conflicts with the following quote.

> The *Basic* service of Transis overcomes arbitrary communication delays and message losses and guarantees fast delivery of messages at all of the currently connected destinations. The membership protocol automatically maintains the set of currently connected machines inside the broadcast domain. [p. 293]

The description of causal multicast from the quote below suggests that in the *basic* multicast mode, delivery order does not preserve causality:

> 1. **Basic** multicast: guarantees delivery of the message at all the connected sites. This service delivers the message immediately from the DAG to the upper level.

2. **Causal** multicast: guarantees that delivery order preserves causality. [p. 296]

Assuming that messages are delivered as soon as they have been inserted into the DAG however immediately leads to problems: if a configuration change happens simultaneously at all nodes, but one of the nodes receives a regular message before that change while another receives that same message only later, then virtual synchrony (as described in the second quote) is violated.

In the proof of the faults protocol, we find the following:

> If a message follows any of the messages in $Electors_p(f)$, it is delivered only after the configuration change of f. [p. 300]
> Then p and q deliver the same set of causal messages [...] [p. 300]

These quotes seem to indicate that we are dealing with causal delivery order.

The above quote speaks of "causal messages", which seems to indicate that there are other types of messages being delivered too. We therefore assume that the basic multicast and causal multicast services can be used independently.

Remember that the faults and join protocol change the *CCS* on delivery of FA and JOIN messages. Call these messages configuration changing messages. Trying to formulate a property in the spirit of [SBS91] that matches [ADKM92a], we define virtual synchrony as follows:

Definition 3 (Virtual synchrony). *The network is virtually synchronous if and only if*

- *configuration changing messages are delivered in the same order at every node, and between every two delivered, and*
- *if m_1 and m_2 are configuration changing messages, and a node delivers a message m that was sent using the causal multicast service after m_1 but before m_2, then all other nodes in the current CCS deliver m after m_1 and before m_2.*

6.3 Requirements monitoring

In order to verify that the faults and join protocols have the desired properties, we construct monitor processes that produce an *error* action upon violation of the requirement they encode. We check reachability of this action on the system composed in parallel with the monitor processes.

The consensus monitor synchronises with the system whenever a process crashes and whenever a process starts or stops executing the faults (resp. join) protocol (i.e. whenever a process assents to a configuration change). When no processes are executing the faults (join) protocol, we require that the processes have assented to the same changes. If not, then an *error* action is possible.

The virtual synchrony monitor synchronises with the system whenever a process delivers a message. For every process, the monitor process keeps a list of delivered messages. Whenever these lists violate definition 3, an *error* action is possible.

To check the faults protocol, we model a network consisting of three nodes, of which one can crash. For the join protocol, we start with a network that is partitioned into a clique of two nodes, of which one may crash at any given time, and a clique consisting of a single node. Every node is allowed to send a user message (i.e. a message that has no meaning for the protocol, but that is sent using the causal multicast service) once.

Results The specification of the protocols in mCRL2 are transformed to linear process specification format [GMA+09] using the *mcrl22lps* tool. We then check reachability of the *error* action with the *lps2lts* tool. If an *error* action is reachable, then this tool will produce a trace that leads to this action.

Verification was done using a Q9400 Intel® Core™ 2 Quad CPU (of which one core is used) and 3GB of memory using the mCRL2 toolset development version (revision 7884).

For the faults protocol, it took 28 minutes and 55 seconds to exhaust the state space comprising about 1.2 million states (this state space is the result of the composition of the monitor process and the protocol specification). We could successfully verify both consensus and virtual synchrony for our formal model of the faults protocol.

For the join protocol, the state-space proved to be too large to be verified in a few days. Hence, we resorted to bounded model-checking by limiting the depth or the breadth of the search algorithm. Using the limit on the depth, we successfully verified over 4 million system states (22 levels, i.e. 22 'steps' in the protocol), and over 113 million for the consensus property alone. Using the highway search (breadth-constrained) algorithm [EGvWW09], we could verify 1000 randomly chosen states on every level of the state space.

The complete models of the protocols and the monitors are available online at `http://www.win.tue.nl/~atif/adkm_membership.zip`.

7 Related Work

An overview of different group membership protocols can be found in [CKV01], where [ADKM92a] is cited as the first group membership protocol dealing partitioning. We have gathered and used scattered information about the algorithms of [ADKM92a] and its underlying system from various sources about the protocols and the Transis system [ADKM92a,ADKM92b,ADKM92c].

In the literature several attempts have been made to formalise and verify group membership protocols. The first one, that we are aware of is [Ric93], where a group membership algorithm has been formalised and verified. Later in [ACbMT95], a number of flaws were identified in the formalisation of [Ric93]. In [BDM01], a formal specification of a group membership protocol is presented.

8 Conclusions

We provided a formal specification of the group membership protocols presented in [ADKM92a] and their correctness properties and model-checked the properties on the formal specifications. For the formalisation, we had to disambiguate the description provided in the original paper and in order to realise the properties, we had to try different interpretations of the informal text. The process of disambiguation has been laborious and rather difficult; it often appeared from the formal analysis that the most natural interpretation of the text led to inconsistencies or incorrect behaviour and thus, less faithful interpretations of the text had to be used in order to reconstruct correct group membership protocols. We could successfully model-check our final specification of the faults protocol. For the join protocol, we had to resort to bounded model-checking, which did not indicate any counter-example after both a highway search (breadth-constrained) as well as depth-constrained search.

References

[ACbMT95] Emmanuelle Anceaume, Bernadette Charron-bost, Pascale Minet, and Sam Toueg. On the formal specification of group membership services. Technical Report 95-1534, INRIA Rocquencourt, 1995.

[ADKM92a] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Membership algorithms for multicast communication groups. In *WDAG '92: Proceedings of the 6th International Workshop on Distributed Algorithms*, pages 292–312, London, UK, 1992. Springer-Verlag.

[ADKM92b] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Membership algorithms in broadcast domains. Technical Report 10, The Hebrew University of Jerusalem, 1992.

[ADKM92c] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication subsystem for high availability. In *FTCS '92: Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, 1992. IEEE Computer Society.

[BDM01] Özalp Babaoglu, Renzo Davoli, and Alberto Montresor. Group communication in partitionable systems: Specification and algorithms. *IEEE Trans. Software Eng.*, 27(4):308–336, 2001.

[CKV01] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[EGvWW09] Tom A.N. Engels, Jan Friso Groote, Muck J. van Weerdenburg, and Tim A.C. Willemse. Search algorithms for automated validation. *Journal of Logic and Algebraic Programming*, 78(4):274 – 287, 2009.

[FGP+04] Wan Fokkink, Jan Friso Groote, Jun Pang, Bahareh Badban, and Jaco van de Pol. Verifying a sliding window protocol in μCRL. In Charles Rattray, Savi Maharaj, and Carron Shankland, editors, *AMAST*, volume 3116 of *Lecture Notes in Computer Science*, pages 148–163. Springer, 2004.

[GMA+09] Jan Friso Groote, Aad Mathijssen, Michel A. Reniers, Yaroslav S. Usenko, and Muck van Weerdenburg. Analysis of distributed systems

with mCRL2. In *Handbook of Process Algebra for Parallel and Distributed Processing*, pages 99–128. CRC Press, 2009.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Ric93] Aleta Ricciardi. *The Group Membership Problem in Asynchronous Systems*. PhD thesis, Department of Computer Science, Cornell University, 1993.

[SBS91] A. Schiper, K. Birman, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)*, 9(3):314, 1991.

[vdPE03] Jaco van de Pol and Miguel Valero Espada. Verification of javaspaces$^{\text{tm}}$ parallel programs. In *ACSD '03: Proceedings of the 3rd International Conference on Application of Concurrency to System Design*, pages 196–205. 2003. IEEE Computer Society.