Formal Specification and Verification of a Group Membership Protocol for an Intrusion-Tolerant Group Communication System*

HariGovind V. Ramasamy†, Michel Cukier‡, and William H. Sanders†

University of Illinois†
Urbana, IL 61801, USA
{ramasamy,whs}@crhc.uiuc.edu

University of Maryland‡
College Park, MD 20742, USA
mcukier@eng.umd.edu

Abstract

We describe a group membership protocol that is part of an intrusion-tolerant group communication system, and present an effort to use formal tools to model and validate our protocol. We describe in detail the most difficult part of the validation exercise, which was the determination of the right level of abstraction of the protocol for formally specifying the protocol. The validation exercise not only formally showed that the protocol satisfies its correctness claims, but also provided information that will help us make the protocol more efficient without violating correctness.

1 Introduction

Process groups have been widely used to provide fault tolerance in distributed systems. Group communication systems have been developed to ensure state consistency among the processes that constitute a group in the presence of failures. While a considerable amount of work has been done to make group communication systems tolerate benign failures, the problem of building group communication systems that can tolerate malicious faults resulting from intrusions has begun to be addressed only more recently.

Intrusion-tolerant group communication systems provide reliable ordered message delivery and consistent group membership, despite the malicious behavior of a subset of the group. In this paper, we describe an intrusion-tolerant group membership protocol (GMP) that is part of the ITUA Group Communication System (ITUA GCS) [1][2].

Distributed systems are notoriously prone to design faults. The use of formal methods has been widely advocated to reduce the likelihood of such faults in the early stages of system development and to validate the correctness of systems. Because distributed protocols are complex and their executions can be interleaved in many ways, automated analysis of such protocols is not only desirable, but necessary.

PROMELA/SPIN [3] is a powerful tool for modelling and proving the correctness properties of distributed systems. We define a formal model of the GMP in PROMELA (PROcess MEta LAnguage) and employ the SPIN model checker to validate the correctness claims of the protocol specified in standard Linear Temporal Logic (LTL) [4].

The paper is organized as follows. In the next section, we describe the intrusion-tolerant GMP, and state the properties it is expected to provide. In Section 3, we describe the formal specification of the protocol in PROMELA, explain how the properties were formally specified and verified using SPIN, and present the results of this validation exercise. We summarize the analysis and conclude in Section 4.

2 The Group Membership Protocol

2.1 System Model and Assumptions

We consider a *timed asynchronous* [5] distributed system consisting of several processes on multiple hosts communicating over an unreliable network. Processes have local hardware clocks (which need not be synchronized), but there are no upper bounds on message transmission and scheduling delays. The *timed* asynchronous system assumption helps circumvent the impossibility of consensus in an asynchronous environment [6] by defining time-outs for message transmission and scheduling delays. A *performance failure* occurs if an experienced delay is greater than the associated time-out delay.

The GMP operates over a process group by maintaining and updating the membership list (called the *view*) at each of the constituent processes. Each group member has a unique identifier called its *rank*, which is a number from

*This research has been supported by DARPA contract F30602-00-C-0172.

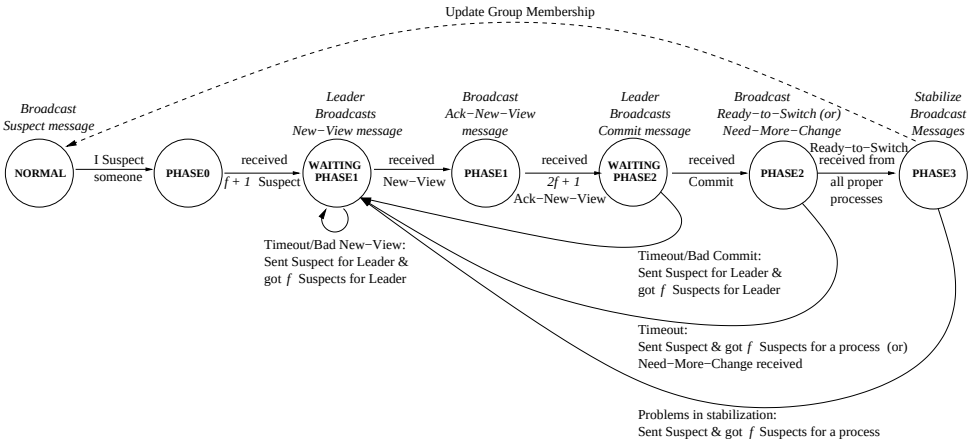

Figure 1. Finite state automaton for view installation

0 to $N-1$, where N is the cardinality of the group. A view is a set of process identifiers from 0 to $N-1$. The member with the lowest rank is called p_0 and is the leader of the view. Each member, from every other member's perspective, is either *correct*, *suspected*, or *corrupt*. A *correct* process conforms to the protocol specifications. If a process p_i does not conform to the specifications, p_i will eventually be *suspected* by other members. When a member decides that p_i is suspected by enough other members of the group, it labels p_i as *corrupt*, and the GMP at that group member starts a three-phase view-installation procedure, at the end of which p_i will be removed from the membership of the group. The GMP can provide consistent group membership to all the correct members of the group, as long as no more than $F = \lfloor (N-1)/3 \rfloor$ members of the group are corrupt.

If the leader is found to be corrupt, then the process p_1 with the second-lowest rank (called the *deputy*) will become the new leader. If both the leader and the deputy are found to be corrupt, the process with the third-lowest rank (called the *deputy's deputy*), p_2, will become the new leader, and so on. The leader wields no additional power to bring about any membership change single-handedly, but has additional responsibilities, which will be elaborated in Section 2.2.

We use standard cryptographic techniques, such as public-key signatures, to prevent spoofing and replays and to detect corrupted messages. Each process in the group possesses a private key, public key pair. A process uses its private key (known only to itself) to sign GMP messages. We assume the existence of a mechanism by which a process can obtain the public keys of all processes in the group to verify signed messages. We assume that the intruder is computationally bound, and hence unable to subvert the cryptographic techniques. We also assume that the intruder cannot steal the private keys from correct processes.

The GMP relies on a reliable multicast protocol (such as the one described in [1]) to reliably deliver the messages it sends to maintain group membership.

2.2 Detailed Description of the Protocol

The GMP is responsible for removing processes from the group, adding new processes to the group, and ensuring that all correct processes maintain consistent information about the membership in spite of intrusions. We summarized the protocol in [1]. We present it in detail here. Though our protocol was influenced by the work of Reiter [7], which tries to solve the group membership problem in an asynchronous environment in the presence of malicious faults, our protocol differs from Reiter's in the following aspects:

- While a view installation in [7] removes a *single* corrupt process or adds a *single* correct process to the group, each view installation that we describe removes *all* hitherto-known corrupt members from the group.
- The protocol in [7] becomes very complex if the manager (the equivalent of a leader in our protocol) is corrupt. Our protocol does not suffer from this drawback.

Figure 1 gives the finite state machine representation of the GMP. Upon initialization, all state machines are in the NORMAL state. Detections of deviation by a group member from the specifications are reported as *suspicions* to the GMP, which then multicasts these suspicions to the group. The GMP provides an interface $suspect(process\ rank\ i,\ reason\ R)$ for that purpose. Correct processes will invoke the $suspect$ function if they observe that another member has deviated from its specified behavior; corrupt processes may invoke this function at any time. When the $suspect$ function is invoked, the GMP of the process that suspects another process will broadcast a signed *Suspect* message to the group and change its state to PHASE0. The *Suspect* message has two parts:

- The current view and the rank of the suspect.
- A signed version of the above data structure, called *suspect_notice*. The *suspect_notice* sent by process p_k for a process it suspects, p_i, is denoted by $suspect\_notice_k(i)$.

$suspect\_message\_handler(Suspect\_message\ m,$
$process\_rank\ k)\ in\ view\ V_x$

1: $i \leftarrow get\_suspect\_rank(m)$
2: $y \leftarrow get\_view(m)$
3: $notice \leftarrow get\_suspect\_notice(m)$
4: **if** $(x \neq y)$ or $(i = my\_rank)$ **then**
5: Stop
6: **end if**
7: **if** $verify\_suspect(m)$ fails **then**
8: Invoke the function $suspect(k,\ Bad\ Suspect)$
9: Stop
10: **end if**
11: **if** $notice\_matrix_{k,i}$ is empty **then**
12: Store $notice$ in $notice\_matrix_{k,i}$
13: **if** this is the $f + 1^{th}$ distinct $suspect\_notice$ received for process p_i **then**
14: $faulty(i) \leftarrow$ TRUE
15: $my\_leader \leftarrow$ smallest j ($j \geq 0$) such that $faulty(j)$ is still false
16: $found\_new\_fault(i)$
17: **end if**
18: **end if**

Figure 2. The $suspect\_message\_handler$ function

The GMP maintains a $notice\_matrix$ data structure. The entry $notice\_matrix_{k,i}$ in the matrix contains $suspect\_notice_k(i)$. When the GMP receives a *Suspect* message m broadcast from process p_k, it invokes the $suspect\_message\_handler(m,\ k)$ function (Figure 2). The $verify\_suspect$ function verifies whether the signature in the $suspect\_notice$ is valid. It also checks whether the rank of the suspected process, i, is between 0 and $|V_x|$, where V_x is the current view. If the $suspect\_notice$ is the $f + 1^{th}$ distinct $suspect\_notice$ received for p_i, then the predicate *faulty(i)* becomes true, and the first phase of the view installation is triggered. When *faulty(i)* is true, that indicates that at least one correct process in the group observed p_i deviating from its specified behavior.

The $found\_new\_fault$ function (Figure 3) is invoked whenever the *faulty(i)* predicate becomes true for some process p_i. If the GMP state is NORMAL or PHASE0, the $found\_new\_fault$ function switches the GMP state to WAITING-PHASE1. At the leader, the $found\_new\_fault$ function creates a *New-View* message containing:

- The list of processes, p_i, such that *faulty(i)* holds.
- For each such p_i, the proof that *faulty(i)* is true. This proof is the i^{th} column in the $notice\_matrix$, containing at least $f + 1$ valid $suspect\_notices$ for p_i.
- The proposed new view to be installed next, which excludes all such p_i.

At a non-leader process, the $found\_new\_fault$ function starts a timer, T_1. The process expects a *New-View* message from p_{my_leader} before T_1 expires. (p_{my_leader} at any process indicates which process this process thinks is the leader.) If the process receives the *New-View* message m from p_{my_leader} before T_1 expires, it invokes the $received\_newview(m)$ function; otherwise, the process invokes $suspect(my\_leader,\ New\text{-}View\ time\text{-}out)$.

$found\_new\_fault(process\_rank\ k)\ in\ view\ V_x$

1: old-state $\leftarrow$ GMP-state
2: **if** old-state = NORMAL **or** old-state = PHASE0 **or** old-state = PHASE2 **or** old-state = PHASE3 **then**
3: GMP-state $\leftarrow$ WAITING-PHASE1
4: **if** $my\_leader = my\_rank$ **then**
5: Broadcast a *New-View* message
6: **else**
7: Start/Restart timer T_1 (timer for WAITING-PHASE1)
8: **end if**
9: **else if** old-state = WAITING-PHASE1 **then**
10: **if** this new fault has caused me to become the new leader **then**
11: Broadcast a *New-View* message
12: **else**
13: Restart the timer T_1
14: **end if**
15: **else if** old-state = PHASE1 **or** old-state = WAITING-PHASE2 **then**
16: **if** this new fault changed my_leader **then**
17: GMP-state $\leftarrow$ WAITING-PHASE1
18: **if** I am the new leader **then**
19: Broadcast a *New-View* message
20: **else**
21: Restart the timer T_1
22: **end if**
23: **else if** old-state = WAITING-PHASE2 **then**
24: Increase the timeout value for T_2 (timer for WAITING-PHASE2)
25: **end if**
26: **end if**

Figure 3. The $found\_new\_fault$ function

The $received\_newview$ function (Figure 4) checks the validity of the *New-View* message received. It updates the list of processes, p_i, for which the *faulty(i)* predicate holds. It also updates the entries in the $notice\_matrix$ for those processes. The GMP maintains a data structure called *View-List*, which is a list for storing the proposed next view indicated by the *New-View* messages received. For each *New-View* message, the *View-List* also contains an array of *Ack-New-View*s received (called *Ack-array*), an array of *Ready-to-Switch* messages received (called *RTS-array*), and an array of *Need-More-Change* messages received (called *NMC-array*). Each of the three arrays contains $|V_x|$ elements, which get updated when the corresponding messages are received. After the *New-View* message has been verified to be valid, the GMP switches its state to PHASE1 and adds the proposed next view to the *View-List* data structure. It then broadcasts an *Ack-New-View* message, which contains the signed version of the proposed next view.

$received\_newview(NewView\_message\ m)$ *in view* V_x

1: $being\_removed \leftarrow$ set of processes in V_x to be excluded from the V_{x+1}, as indicated by m
2: **for** each p_i in the set $being\_removed$ **do**
3: **if** $i = my\_rank$ **then**
4: Stop
5: **end if**
6: **if** m does not have the array of $suspect\_notices$ for p_i containing at least $f + 1$ valid $suspect\_notices$ **then**
7: Invoke the function $suspect(my\_leader,\ Bad\ New\text{-}View)$
8: Stop
9: **else**
10: **if** $faulty(i)$ is not already TRUE **then**
11: $faulty(i) \leftarrow$ TRUE
12: Update the i^{th} column in the $notice\_matrix$ with the $suspect\_notices$ in m for p_i
13: **end if**
14: **end if**
15: **end for**
16: $my\_leader \leftarrow$ smallest j ($j \geq 0$) such that $faulty(j)$ is still false
17: **if** $origin(m) = my\_leader$ **then**
18: GMP-state $\leftarrow$ PHASE1
19: Add the proposed next view indicated by m to the *View-List* data structure
20: Broadcast an *Ack-New-View* message containing the signed version of the proposed next view
21: **end if**

Figure 4. The $received\_newview$ function

When the GMP receives an *Ack-New-View* message, it verifies the validity of the message and then stores it in the *Ack-array* of the corresponding new view in the *View-List* data structure. When $2f + 1$ *Ack-New-View* messages for a proposed next view have been received, the second phase of the view installation begins.

At the leader, the GMP forms a *Commit* message containing the proposed next view and justification for installing the next view. The justification is the *Ack-array*, which contains at least $2f + 1$ *Ack-New-View*s. It shows that a majority of the correct members agree to exclude the faulty members indicated by the *New-View* message. The GMP now switches from PHASE1 to WAITING-PHASE2.

At processes other than the leader, if the GMP is in PHASE1 when it has received *Ack-New-View*s from $2f + 1$ processes, it switches to WAITING-PHASE2 and then starts a timer, T_2. It expects a *Commit* message from p_{my_leader} before T_2 expires. If the GMP receives a *Commit* message m from p_{my_leader} before timer expiry, it invokes the $received\_commit(m)$ function (Figure 5); otherwise, the GMP invokes $suspect(my\_leader,\ Commit\ time\text{-}out)$.

After checking the validity of the received *Commit* message, the GMP switches state to PHASE2 and broadcasts

$received\_commit\ (Commit\_message\ m)$ *in view* V_x

1: **if** $origin(m) \neq my\_leader$ **then**
2: Stop
3: **end if**
4: **if** the proposed next view doesn't include me **then**
5: Stop
6: **end if**
7: **if** the justification field of m doesn't contain $2f + 1$ *Ack-New-View*s **then**
8: Invoke the function $suspect(my\_leader,\ Bad\ Commit)$
9: Stop
10: **end if**
11: GMP-state $\leftarrow$ PHASE2
12: **if** the proposed next view excludes all p_i such that $faulty(i)$ is TRUE **then**
13: Broadcast a *Ready-to-Switch* message containing the proposed next view
14: Start a timer T_3
15: **else**
16: $cast\_NMC(m)$
17: **end if**

Figure 5. The $received\_Commit$ function

a *Ready-to-Switch* message containing the proposed next view, if the proposed next view excludes all p_i for which the $faulty(i)$ predicate holds. If the proposed next view contained in the *Commit* message does not exclude all known corrupt members, then the GMP invokes the $cast\_NMC()$ function (Figure 6). That may happen if new corruptions were detected during the view installation, i.e., $f + 1$ *Suspect* messages have been received for one or more members that are not among the processes being removed from the group by the last *New-View* message. The $cast\_NMC()$ function broadcasts a *Need-More-Change* message indicating that the proposed next view specified in the last *New-View* message does not exclude all known corrupt members. A valid *Need-More-Change* message points out the other corrupt members that need to be excluded, and provides justification in the form of $f + 1$ *Suspect* messages received for each of the other corrupt members.

After broadcasting the *Ready-to-Switch* message, the GMP starts a timer, T_3. All members of the proposed next view should send either a valid *Ready-to-Switch* or a *Need-More-Change* message before the timer T_3 expires. The function $suspect(j,\ Phase2\ time\text{-}out)$ would be invoked for any p_j that did not send a *Ready-to-Switch* or *Need-More-Change* message before the timer T_3 expired.

When the GMP at each member receives a *Ready-to-Switch* message from each member of the proposed next view, it is a confirmation that all of those members have agreed to be part of the next view. The GMP changes its state to PHASE3. PHASE3 is the message stabilization phase, which involves the collaboration of the GMP with the reliable delivery protocol. This phase is necessary if the

$cast\_NMC(Commit\_message\ m)$ *in view* V_x

1: $need\_to\_remove \leftarrow \{\}$
2: **for** each p_i such that $faulty(i)$ is TRUE but the next view proposed by m does not exclude p_i **do**
3: add i to the list $need\_to\_remove$
4: Include the i^{th} column of the $notice\_matrix$ containing $f+1$ $suspect\_notices$ for p_i (This serves as the proof that *faulty(i)* holds and hence p_i also needs to be excluded from the next view)
5: **end for**
6: Broadcast a *Need-More-Change* message containing the $need\_to\_remove$ list with the proofs attached

Figure 6. The $cast\_NMC$ function

group communication system has to ensure that all correct processes deliver the same set of messages broadcast in a view (for *virtual synchrony* [10]). In PHASE3, the GMP gives the reliable layer a list of processes that are being removed. The members of the current view that are also members of the next view begin a consensus round to decide which messages have been delivered by each of them at this point. To do so, each of those members reliably multicasts a signed message with an array that indicates the highest-sequence-numbered message from each member that has been reliably delivered at this member. After that, there is a second round in which every member multicasts a message containing all the signed copies from the previous round. At the end of the second round, each member is able to form a data structure containing three arrays, *max_num_cast*, *has_max_num_cast*, and *stability_vector*.

1. For $0 \leq j < |V_x|$, $max\_num\_cast[j]$ is the highest-sequence-numbered multicast from p_j received by some group member.
2. For $0 \leq j < |V_x|$, $has\_max\_num\_cast[j]$ indicates which group member has received that highest-sequence-numbered multicast from p_j.
3. For $0 \leq j < |V_x|$, $stability\_vector[j]$ indicates the highest-sequence-numbered multicast from p_j that has been received by all members of the next view.

Process $p_{has\_max\_num\_cast[j]}$ is responsible for re-broadcasting all messages from sequence numbers $stability\_vector[j]$ to $max\_num\_cast[j]$ received from process p_j. Once all such pending messages have been delivered, the stabilization phase is completed, and the GMP updates the group membership to that given by the last *New-View* message. That marks the end of the view installation, and the GMP changes its state to NORMAL.

Figure 1 has reverse transitions from WAITING-PHASE1, WAITING-PHASE2, PHASE2, and PHASE3 to WAITING-PHASE1. Reverse transitions take place if *additional faults* occur during view installation. An additional fault occurs if $f+1$ *Suspect* messages are received for a member that is not among those processes being removed by the current view installation. In such cases, the $found\_new\_fault$ function (see Figure 3) will be invoked. Table 1 explains the cause and effect of each reverse transition. Once in WAITING-PHASE1, the view installation proceeds as before so that the GMP changes state to PHASE1 upon receipt of the *New-View* message, and so on.

When a process wants to join the group, it sends a signed *Request-to-Join* message to the members of the group. A member that receives the message will check if the new process was previously removed from the group because of a malicious fault (time-out faults do not count as malicious behavior). If it was, the request is ignored. The member then checks if the process is in the list of processes authorized to join the group. This list would be given to the group member at the time of its creation by a higher-level entity that created the member. The creating entity itself could be replicated for fault tolerance. Updates to the list can be made by the creating entity based on increased intrusion-tolerance requirements and identification of new intrusions. An example of such a higher-level entity that spawns group members is the *manager group* in the ITUA architecture (see [2] and [9]). If the requesting process is in the list of processes authorized to join the group, the member that received the *Request-to-Join* multicasts a *Proposal* message to the group that contains the received *Request-to-Join* message. If *Proposal* messages are received from $f+1$ members, it triggers a three-phase view installation procedure similar to the view installation procedure for removing a member. Note that as before, new corruptions may be detected during the view installation. If they are, the actions indicated by Table 1 would take place. The difference between the view installation procedures for adding a new member and removing an existing member is that while the first *New-View* message broadcast during the removal carries $f+1$ *Suspect* messages as the proof, the *New-View* message in the joiner protocol carries the $f+1$ *Proposal* message as proof. Note that when a view installation is underway, no other requests for joining the group will be accepted. Any new process that wants to join the group will have to resend the *Request-to-Join* messages later, when the view installation is done.

2.3 Group Membership Protocol Properties

The GMP has the following properties. They are similar to those provided by SecureRing [8] and Rampart [7].

Agreement *If p and q are two correct processes, then view V_x will have the same membership at both processes.*

Self-inclusion *If a correct process p installs a view V_x, then V_x includes p.*

Validity *If a correct process p installs a view V_x, then all correct members of V_x will eventually install V_x.*

Integrity *If view V_x includes p but the next view V_{x+1} excludes p, then p was suspected by at least one correct*

Table 1. How the GMP handles additional faults that occur during the view installation

Reverse Transition	Cause	Effect
WAITING-PHASE1 to WAITING-PHASE1	Leader did not send a *New-View* message or sent an invalid *New-View* message	Deputy takes over as the new leader and broadcasts a *New-View* that excludes previously known corrupt members and the old leader from the next view; other members restart timer T_1
WAITING-PHASE2 to WAITING-PHASE1	Leader did not send a *Commit* message or sent an invalid *Commit* message	Deputy takes over as the new leader and broadcasts a *New-View* that excludes previously known corrupt members and the old leader from the next view; other members restart timer T_1
PHASE2 to WAITING-PHASE1	*Need-More-Change* message received, or a member of the proposed next view did not send a *Ready-to-Switch* or *Need-More-Change* message before timer T_3 expired	Leader broadcasts a *New-View* message that excludes previously known corrupt members and the new corrupt members; other members restart timer T_1
PHASE3 to WAITING-PHASE1	Some member(s) obstructed progress in the message stabilization phase	Leader broadcasts a *New-View* message that excludes previously known corrupt members and member(s) that obstruct progress in the stabilization phase; other members restart timer T_1

member of V_x.

Liveness *If there is a correct process p that is a member of view V_x and is not suspected by $\lceil(2|V_x|+1)/3\rceil$ correct members of V_x, and there is a process q that is suspected by $\lfloor(|V_x|-1)/3\rfloor$ correct members of V_x, then q is eventually removed.*

3 Verification of the protocol

In this section, we describe how we formally verified the GMP. Before taking the formal approach, we proved the correctness of the protocol informally in [9]. Despite the informal proofs, we were aware that the likelihood of subtle errors in a complex distributed protocol such as ours was not negligible. Although the use of formal methods does not *guarantee* correctness, we were convinced that because of the sound mathematical foundations upon which formal methods are based, our confidence in the correctness of the system would improve if we employed formal methods to validate our protocol.

The formal verification of our fairly complex protocol required a formal tool that could effectively model the asynchronous, distributed nature of our system, and perform verification on a large state space. We also considered ease of learning and use as an important criterion in choosing the verification tool.

SPIN is a general verification tool for proving correctness properties of distributed or concurrent systems. The processes in the system being modeled can interact through shared memory, through rendezvous operations, or through buffered message exchanges. SPIN provides an effective way for debugging the coordination problems that those interactions may create, and for rigorously proving the correctness of such systems. SPIN accepts design specifications written in the verification language PROMELA. It has a powerful yet simple notation for expressing general correctness requirements: standard Linear Temporal Logic (LTL).

3.1 Specification of the protocol in PROMELA

In this section we describe how we translated the protocol described in the previous section into PROMELA code, which can be verified by the SPIN model checker. The most important challenge in the translation was that of choosing the right level of abstraction of our protocol. Abstractions are necessary for verifying a complex protocol, such as ours, through model checking; otherwise, state-space explosion renders model checking useless. We use the term *protocol* to refer to our original intrusion-tolerant group membership protocol, detailed in the previous section. We use the term *specification* to refer to the formal specification of the protocol in PROMELA.

3.1.1 Modeling faulty behavior of processes

The protocol in a correct group member decides that another group member is faulty when the *suspect* function is invoked for the faulty member (resulting in the multicast of a *Suspect* message), or when a *Suspect* message for the faulty member is received. Either of these events could happen in any state of execution of the protocol. We modeled

that aspect of the protocol in our specification by stating a priori which group members are faulty, and by making the correct members multicast *Suspect* messages for the faulty members at various points of execution. Consider a verification of the correctness claims of the protocol for a group size of 7. In one *verification cycle*[1], the faulty members could be specified to be members with ranks 0 and 3, for example. Other processes could multicast a *Suspect* message for p_3 when they are in the NORMAL state, and could multicast a *Suspect* message for p_0 in WAITING-PHASE1 (so as to model non-receipt of a *New-View* message). In another verification cycle, p_4 and p_5 could be specified as faulty members, and *Suspect* messages for both of them could be multicast when the correct members are in the NORMAL state, for example. A complete verification will involve many such verification cycles covering all possible combinations of the members that are to be suspected, and all possible combinations of the states in which the *Suspect* messages for those members are to be sent.

Our chosen approach to modeling faulty behavior greatly simplified our specification, as described below:

Abstracting out justifications A *New-View* message gives a list of processes to be removed from the current view, and justification for removal, in the form of $f + 1$ signed *Suspect* messages received for each of those processes. In our PROMELA specification, we can model the receipt of an invalid *New-View* message at some state of execution of our protocol by multicasting a *Suspect* message for the sender at that state. We can do the same for other messages that carry justifications, namely the *Commit* message (carries $2f + 1$ signed *Ack-New-View*s as justification) and *Need-More-Change* message (carries $f + 1$ signed *Suspect*s for each member it proposes should be removed in the new membership, as justification). Hence, we eliminated justifications from those messages in the specification.

Abstracting out cryptography In our protocol, messages are signed to prevent spoofing. The underlying reliable delivery protocol uses cryptographic signatures to ensure that any attempts to spoof or to send mutant messages[2] are guaranteed to be detected through signature verification by enough group members to trigger the removal of the attempting member. In our specification, we eliminate signing and verifying of messages, and make processes send the same message contents to all other processes in the group. We model attempts to spoof or to send mutant messages from a corrupt process by making the other processes in the group multicast *Suspect* messages for the corrupt process.

Modeling time-outs In our protocol, if time-out occurs at a member p_i before the receipt of some message (like a *New-View* message) that was expected from another member p_j, then p_i multicasts a *Suspect* message for p_j. SPIN has a *global* time-out feature, which means that time-out occurs only if *no* process in the SPIN execution environment is executable. However, it is impossible to simulate independent timers, or asynchronous time-outs between different group members, as is required for our purpose. Hence, in our specification, we modeled timer expiry by making the correct group members multicast *Suspect* messages for the group member that failed to take the expected action before the timer expiry. We do not use real clocks for that purpose.

[1] We explain what a "verification cycle" is later in this section.

[2] A mutant message is a pair of messages that have the same identifier but different contents, and that are sent to two different processes in the group.

3.1.2 Abstraction of multicast and data abstraction

In Section 2 we described the data structures used for the GMP messages. Taking advantage of the abstractions/assumptions described above, we retained only the critical parts of those data structures when specifying the protocol in SPIN. Inter-process point-to-point communication is modeled in SPIN through use of a data type called *channel*. A process can send contents of some specified type to a channel, while other processes can receive those contents from that channel. However, the current version of SPIN does not provide support for multicasting messages to a group. Hence, to model multicast within the group, we created global arrays of channels, one for each GMP message type (*Suspect*, *New-View*, *Ack-New-View*, *Commit*, *Ready-to-Switch*, and *Need-More-Change*), as follows:

```
chan suspect_for[N]=[K] of {byte,byte};
/* rank of sender, rank of suspected process */
chan nv_for[N]=[K] of {byte,View};
/* rank of sender, membership list for next view */
chan acknv_for[N]=[K] of {byte,View,byte};
/* rank of sender, New-View that is being acknowledged,
    rank of process that multicast that New-View */
chan commit_for[N]=[K] of {byte,View};
/* rank of sender, New-View that is being committed */
chan rts_for[N]=[K] of {byte,View};
/* rank of sender, New-View to be installed */
chan nmc_for[N]=[K] of {byte,View};
/* rank of sender, membership list for next view */
```

where N is the size of the group, F is the number of simultaneous faults that can be tolerated ($F = \lfloor (N-1)/3 \rfloor$), and K is a sufficiently large constant.

For example, when a process wants to multicast a *Suspect* message to the group, it sends the message to the channel `suspect_for[i]`, for all i from 0 to $N-1$. Each process p_i then receives the contents from the channel `suspect_for[i]`. Since the underlying reliable multicast protocol guarantees that if a message gets delivered at one correct process, then all correct processes will deliver the

same message, we do not have to model the case in which a corrupt process sends a message (that it was supposed to multicast) only to a subset of the group.

The data structure for each GMP message is indicated by the specification for the channel used to send that message. For example, the *Suspect* message is modeled by the `suspect_for` channel, which can hold two 8-bit values for each message. The two 8-bit values indicate the sender ID (origin rank), and the rank of the suspected member.

A view is modeled as a Boolean array of N elements, where the i^{th} element indicates whether p_i has been convicted and excluded from the group.

```
typedef View {
        bool outGroup[N] };
```

All the elements in the array are initiailized to `false`. As, one by one, the faulty members are convicted, the corresponding elements in the Boolean array become true.

3.1.3 Simplification of the stabilization phase

In our protocol, the third phase of view installation (the stabilization phase) works in collaboration with the reliable delivery protocol to guarantee Virtual Synchrony [10]. In our specification, we defined a global array of channels:

```
chan phase3_ok[N] = [N] of {byte,bool};
/* rank of sender, whether sender performed phase 3 correctly */
```

To model the case in which some member does not cooperate in completing the stabilization phase correctly, we have other processes in the group send *Suspect* messages for the member in that phase. To model the case when all members behave correctly in this phase, we make each member send a `true` value to all `phase3_ok` channels.

3.2 Specifying the properties in SPIN

We now describe how each of the properties in Section 2.3 was specified in SPIN. We define three global arrays:

```
View my_current_view[N] ;
bool fault_injected[N] ;
bool done[N] ;
```

`my_current_view[i]` indicates the group membership from process p_i's perspective. p_i will update only the i^{th} element of the array `my_current_view`. All elements of the array are initialized to the same value, indicating that all N processes that are created are part of the group.

`fault_injected[i]` is used to simulate a deviation of process p_i from the protocol specification. For a correct process p_i, `fault_injected[i]` has the value `false`. For some j, if `fault_injected[j]` has the value `true`, that serves as the trigger for correct processes in the group to send *Suspect* messages for process p_j during some specified state of execution of the GMP.

A *verification cycle* for a group of N processes involves the following steps:

1. The `init` process, which is a special SPIN process used to initialize the system state, creates the processes that constitute the group. It labels a subset of the group as faulty by assigning a `true` value to the corresponding elements of the `fault_injected` array.
2. The suspect triggers for the faulty members are activated at the correct members at the specified stages of execution of the protocol.
3. The view installation procedure begins.
4. The correct members change their views to exclude the faulty members. Each correct member updates the corresponding element of the `my_current_view` array.

At the end of the verification cycle, when a correct process p_i has completed switching over to the new view that excludes all faulty members, it sets `done[i]` to true. The last correct process to finish (let us call it p_k) will check to see whether the protocol properties of *agreement*, *validity*, *integrity*, and *self-inclusion* are satisfied. For that, we define four Boolean global variables, all initialized to `false`.

```
bool agreement, integrity,
     self_inclusion, all_good_proc_done;
```

For *agreement* and *validity*, p_k checks whether all correct processes have the same new view excluding all faulty members of the previous view. This check is expressed as the following proposition:

$$\begin{aligned}&\forall\, i,\, j : 0 \ldots N-1 \bullet\ (fault\_injected[i] = false \Rightarrow \\ &\quad (fault\_injected[j] = true \Rightarrow \\ &\qquad my\_current\_view[i].outGroup[j] = true) \\ &\wedge\ (fault\_injected[j] = false \Rightarrow \\ &\qquad my\_current\_view[i].outGroup[j] = false))\end{aligned}$$

If the above proposition is satisfied, then p_k sets the value of the global variable `agreement`.

For *integrity*, p_k checks whether all correct processes have received *Suspect* messages from $F+1$ group members for each faulty member. There is a global variable that tracks the number of *Suspect* messages received by each member of the group for each member of the group.

```
typedef SuspectArr {
        byte numSuspects[N] };
SuspectArr suspectArr[N];
```

Here, `suspectArr[i].numSuspects[j]` gives the number of *Suspect* messages received by p_i for p_j. The integrity property is expressed as the following proposition:

$$\begin{aligned}&\forall\, i,\, j : 0 \ldots N-1 \bullet\ (fault\_injected[i] = false \Rightarrow \\ &\quad (fault\_injected[j] = true \Rightarrow \\ &\qquad suspectArr[i].numSuspects[j] \geq F+1))\end{aligned}$$

If the above proposition is satisfied, then p_k sets the value of the global variable `integrity`.

The *self-inclusion* property is expressed as follows:

$$\forall\, i : 0 \ldots N-1 \bullet\ (fault\_injected[i] = false \Rightarrow$$

$(my\_current\_view[i].outGroup[i] = false))$

If the above proposition is satisfied, then p_k sets the value of the global variable `self_inclusion`.

Finally, p_k sets a global variable called `all_good_proc_done` and checks the following assertion to see whether all safety properties were satisfied:

```
assert(agreement && integrity &&
       self_inclusion)
```

The liveness property is expressed simply as the constraint that the global Boolean variable, `all_good_proc_done`, must eventually become true. That is expressed as the following LTL property:

$$\Diamond \ all\_good\_proc\_done$$

SPIN negates this LTL formula to form a negative correctness claim represented by a Büchi automaton [3]. The SPIN model checker will prove the absence of acceptance cycles[3] in the combined execution of the system (consisting of the processes that form the group) and the Büchi automaton representing the claim. Thus, during each verification cycle, SPIN will verify that no execution sequence of the system in any order matches the negated correctness claim. It will also verify that all possible event sequences in all possible orders satisfy the assertion given above.

3.3 Verification and Results

In a verification cycle, we fix the group members that are to be suspected, and also the state of the protocol in which the *Suspect* messages for those members are to be sent. A full verification for a process group of size N involves several verification cycles, which cover all possible combinations of the members that are to be suspected, and all possible combinations of the states in which the *Suspect* messages for those members are to be sent.

We carried out such a full verification for groups of sizes 4, 7, and 10 that can tolerate 1, 2, and 3 simultaneous faults, respectively. In essence, that covers all the cases for group sizes between 4 and 12. The reason is that full verification for group sizes 5 and 6 would be the same as the full verification for group size 4, because each of those groups can tolerate only one fault. Similarly, a full verification for group sizes 8 and 9 would be equivalent to the full verification for group size 7, because the number of simultaneous faults that can be tolerated in each of those group sizes is 2. In the same way, group sizes 10, 11, and 12 are equivalent.

We now explain the actual number of verification cycles involved in each of the above 3 cases.

Consider a process group of size 4. Only one fault can be tolerated, and the trigger for the fault must come when the GMP is in the NORMAL state, at the beginning of the view installation. A naive assessment would yield 4 verification cycles, depending on which of the 4 members is labelled faulty. However, we only need to consider two cases, depending on whether the faulty member is the leader. If the leader is faulty, the deputy takes over as the new leader. By symmetry of all the non-leader processes, the view installation procedure would be identical for the three other cases, in which any one of the non-leaders is faulty.

Now consider a group of size 7. Two simultaneous faults can be tolerated in that case. The trigger for the first fault must come when the GMP is in the NORMAL state, but the second fault can occur in any of the 7 states of the finite state machine shown in Figure 1. Again, a naive assessment would yield $7 \times {}^{7}_{2}C = 147$ verification cycles, ${}^{7}_{2}C$ being the number of possible choices of the two processes to be labelled faulty among the seven processes forming the group. However, only five out of the ${}^{7}_{2}C$ cases were unique in the way they affected the progress of the view installation. The five cases are the ordered pairs (leader, deputy), (deputy, leader), (leader, non-leader), (non-leader, leader), and (non-leader, non-leader), where each ordered pair gives the first and second faults to be triggered during the view installation. Therefore, we had to consider a total of $5 \times 7 = 35$ verification cycles for a full verification.

For a group size of 10 processes, three simultaneous faults can be tolerated. The trigger for the first fault must come when the GMP is in the NORMAL state, but the second and third faults may occur in any of the 7 states of the protocol. That means that there is a total of 49 possible combinations for the fault activation points alone. A naive assessment would yield $49 \times {}^{10}_{3}C = 5880$ verification cycles, ${}^{10}_{3}C$ being the number of possible choices of the three processes to be labelled faulty among the ten processes forming the group. However, only 16 out of the ${}^{10}_{3}C$ cases were unique in the way they affected the progress of the view installation. The sixteen cases are the six permutations in (leader, deputy, deputy's deputy), the six permutations in (leader, deputy, non-leader), the three permutations in (leader, non-leader, non-leader) and the single permutation in (non-leader, non-leader, non-leader), where each 3-tuple gives the first, second, and third faults to be triggered during the view installation. Thus, we considered a total of $49 \times 16 = 784$ verification cycles for a full verification.

We carried out all experiments on a 1GHz Pentium III computer with 256MB PC133 RAM. Despite our abstractions, the complexity of our protocol and the huge number of possible interleavings of process executions meant that we had to use SPIN optimizations, like partial order reduction and bit-state hashing [3], to obtain a good coverage of the state space, given our system memory constraints.

Our verification showed that the GMP indeed provides the properties given in Section 2.3. However, the most in-

[3] A Büchi automaton has an acceptance cycle for a system execution if and only if that execution forces it to pass through one or more of its accepting states infinitely often.

teresting aspect of this exercise was some observations we made that led to increased efficiency of the protocol.

In the original protocol specification, we had a data structure (*View-List*) to hold all the *New-View* messages received. The structure also stored the *Ack-New-View*, *Ready-to-Switch*, and *Need-More-Change* messages received for any particular *New-View*. During the SPIN validation exercise, we found out that it is not necessary to maintain copies of all the *New-View* messages received. It is sufficient to store only the last *New-View* message that excludes more members than the previous *New-View*. (Of course, in our actual protocol implementation, that latest *New-View* message must be valid and carry justification in the form of signed *Suspect* messages for each member to be excluded.) In our SPIN verification, we used the latter, more efficient model and verified that the properties still hold.

In our original protocol specification, when two or more simultaneous faults are involved, during the second phase (PHASE2) of the view installation, members wait for *Ready-to-Switch* or *Need-More-Change* messages from all hitherto non-faulty members before switching to the WAITING-PHASE1 state. During the validation exercise, we verified that it is correct to switch to WAITING-PHASE1 after receiving just one valid *Need-More-Change* message. Subsequent *Need-More-Change* messages or *Ready-to-Switch* messages corresponding to the old *New-View* message can be discarded without affecting the correctness of the protocol. As a generalization, we found that it is safe to discard GMP messages that correspond to old *New-View* messages. (Such messages include *Ack-New-View*, *Commit*, *Ready-to-Switch*, or *Need-More-Change* messages that correspond to old *New-View* messages.)

We also determined that once a member has been found to be faulty (through receipt of $f + 1$ Suspect messages for that member) all group-membership-protocol-related messages from that member can be discarded in the future.

We must point out that what we verified with SPIN was the abstracted version of our actual protocol. Abstraction is essential in verifying a complex system such as ours. However, abstraction involves the hiding of details, which means that hidden details are not verified. For example, we are not verifying the part of our system that reports suspicions to the GMP. However, our goal was to verify the core of our GMP given the current state-of-the-art model-checking technology, and we believe that we were successful in that.

4 Conclusion

In this paper, we described a group membership protocol that can provide consistent group membership while tolerating multiple, correlated intrusions, provided that no more than one-third of the group members are corrupt. Such a protocol would be at the core of an intrusion-tolerant group communication system that keeps replicated information [11] consistent even in the presence of faults.

Complex distributed protocols are notoriously prone to design faults. Automated verification tools can model and verify complicated scenarios in such protocols. We exploited the expressiveness of PROMELA and interpretative strength of SPIN to formally verify our protocol. By abstracting the GMP as described in Section 3.1, we successfully developed a PROMELA specification of the protocol and the properties it claims to provide. We used the SPIN model checker to verify that those claims hold. In doing so, we found optimizations that could be made to the protocol while still retaining correctness.

Acknowledgments We thank Prof. Michael Loui for his useful discussions and helpful comments, and Jenny Applequist for helping us improve the readability of the paper.

References

[1] H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders, "Quantifying the Cost of Providing Intrusion Tolerance in Group Communication Systems," *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN-2002)*, pp. 229-238, June 2002

[2] M. Cukier, J. Lyons, P. Pandey, H. V. Ramasamy, W. H. Sanders, P. Pal, F. Webber, R. Schantz, J. Loyall, R. Watro, M. Atighetchi, and J. Gossett, "Intrusion Tolerance Approaches in ITUA," FastAbstract in *Supplement of the 2001 International Conference on Dependable Systems and Networks*, pp. B64-B65, 2001

[3] Gerard J. Holzmann, "The Spin Model Checker," *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, pp. 279-295, May 1997

[4] A. Pnueli, "The Temporal Logic of Programs," *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS 1977)*, pp. 46-57, 1977

[5] F. Cristian and C. Fetzer, "The Timed Asynchronous Distributed System Model," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 10, No. 6, pp. 642-657, 1999

[6] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, Vol. 32, No. 2, pp. 374-382, April 1985

[7] Michael K. Reiter, "A Secure Group Membership Protocol," *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pp. 176-189, 1994

[8] Kim Potter Kihlstrom, Louise E. Moser, and P. Michael Melliar-Smith, "The SecureRing Protocols for Securing Group Communication," *Proc. of the 31st IEEE Hawaii Intl. Conference on System Sciences*, pp. 317-326, 1998

[9] HariGovind V. Ramasamy, "Group Membership Protocol for an Intrusion-Tolerant Group Communication System," MS Thesis, Univ. of Illinois at Urbana-Champaign, 2002

[10] Kenneth P. Birman, *Building Secure and Reliable Network Applications*, Manning Publications, 1996

[11] Fred Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, Vol. 22, No. 4, 1990