PAPER *Special Issue on Dependable Computing*

Formal Verification of an Intrusion-Tolerant Group Membership Protocol*

HARIGOVIND V. RAMASAMY†, **MICHEL CUKIER**††, *and* **WILLIAM H. SANDERS**†, ***Nonmembers***

SUMMARY The traditional approach for establishing the correctness of group communication protocols is through rigorous arguments. While this is a valid approach, the likelihood of subtle errors in the design and implementation of such complex distributed protocols is not negligible. The use of formal verification methods has been widely advocated to instill confidence in the correctness of protocols. In this paper, we describe how we used the SPIN model checker to formally verify a group membership protocol that is part of an intrusion-tolerant group communication system. We describe how we successfully tackled the state-space explosion problem by determining the right abstraction level for formally specifying the protocol. The verification 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.
key words: *intrusion tolerance, group communication systems, validation, formal methods*

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 consist of protocols that provide reliable ordered message delivery and consistent group membership, despite the malicious behavior of a subset of the group. In [9], we described an intrusion-tolerant group membership protocol (GMP) that is part of the ITUA Group Communication System ([3], [11]), and presented argument-based informal proofs about the correctness of the protocol. However, we are aware that the likelihood of design and implementation faults in a complex protocol such as ours cannot be totally ruled out. The use of formal methods has been widely advocated to reduce the likelihood of such faults and to validate the correctness of systems. Given the complexity of our protocol and the many possible ways in which its execution can be interleaved, we felt that automated analysis and exhaustive verification through formal methods are necessary to instill confidence in the correctness of the protocol. For this purpose, we used the PROMELA/SPIN [5] model-checking tool. PROMELA (Process Meta Language) is a specification language for modeling finite-state systems. SPIN (Simple Promela Interpreter) is a state-of-the-art model checker for analyzing the logical consistency of distributed systems and proving their correctness properties. We define a formal model of the GMP in PROMELA and employ the SPIN model checker to validate the correctness claims of the protocol specified in standard Linear Temporal Logic [7].

In [10], we presented a first effort at formal verification of the GMP. In this paper, we present an overview of the GMP, and describe the formal verification of the protocol in greater detail. This paper is intended to give a view of the issues involved in formally modeling and verifying group communication protocols using current state-of-the-art model-checking technology.

The paper is organized as follows. In Sect.2, we present an overview of the intrusion-tolerant GMP, and state the properties it is expected to provide. In Sect.3, we describe the formal specification of the protocol in PROMELA, explain how the properties were formally specified and verified using SPIN, and present detailed experimental results. We summarize the analysis and conclude in Sect.4.

2. The Group Membership Protocol

2.1 System Model and Assumptions

We consider a *timed asynchronous* [2] distributed system consisting of several processes on multiple hosts communicating over an unreliable network. By "unreliable," we mean that the messages can be dropped or delivered late. 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

Manuscript received March 31, 2003.
Manuscript revised July 7, 2003.
Final manuscript received August 14, 2003.
†The authors are with the University of Illinois, Urbana, USA. E-mail: {ramasamy, whs}@crhc.uiuc.edu
††The author is with the University of Maryland, College Park, USA. E-mail: mcukier@eng.umd.edu
*This research has been supported by DARPA contract F30602-00-C-0172.

helps circumvent the impossibility of consensus in an asynchronous environment [4] 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. The protocol does so by installing a series of views $V_0, V_1, \ldots$ at each group member. A view V_x is a set of process identifiers from 0 to $N-1$. Each group member has a unique identifier called its *rank*, which is a number from 0 to $N-1$, where N is the cardinality of the group ($N = |V|$). We use p_i to denote the group member with rank i. The member with the lowest rank is called p_0 and is the leader of the view.

Each group member is either *correct* or *corrupt*. A *correct* process conforms to the protocol specifications. A *corrupt* process can behave arbitrarily. 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. Hence, f gives an upper bound on the number of group members that can be simultaneously corrupt and still allow the GMP to satisfy its properties.

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 Sect.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.

Since the underlying network connecting the processes can be unreliable, the GMP depends on a reliable multicast protocol (such as the one described in [11]) to deliver the messages it sends to maintain group membership. The reliable multicast protocol must guarantee that all correct processes deliver the same set of multicast messages. It must also guarantee that messages from a given sender are delivered in FIFO order.

2.2 Overview of the Protocol

The intrusion-tolerant GMP we evaluate ensures that all correct processes maintain consistent information about the current membership of the group in spite of intrusions that may occur. The GMP is responsible for maintaining group membership information, removing processes from the group, and joining new processes into the group. It has the properties described below, which are similar to those provided by [6] and [12].

Agreement *The view V_x at any two correct processes p and q will have the same membership.*

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 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 $\lfloor (|V_x| - 1)/3 \rfloor + 1$ correct members of V_x, then q is eventually removed.*

In this paper, we present only a high-level description of how the GMP acts to remove corrupt member(s) from the group. The interested reader is referred to [9] for details about how new members can be added to the group, a pseudo-code-level description of the entire GMP, and informal proofs that the protocol satisfies the abovementioned properties.

2.2.1 Removing a Single Corrupt Member

We represent the GMP as a set of communicating finite state machines, each as shown in Fig.1.

In this section, we assume that only one process will exhibit faulty behavior[†]. Upon initialization, all state machines are in the NORMAL state. The GMP provides an interface *suspect(process-rank i, reason R)* to the microprotocols of the group communication system. At a correct process, this function will be invoked if the process observes that another member has deviated from its specified behavior; a corrupt process may invoke this function at any time. When the function is invoked, the GMP at the process that suspects another process will broadcast a signed *Suspect* message to the group and change its state to PHASE0.

[†]However, we do tolerate the scenario in which multiple processes exhibit faulty behavior simultaneously. A description of how we do so is in Sect.2.2.2.

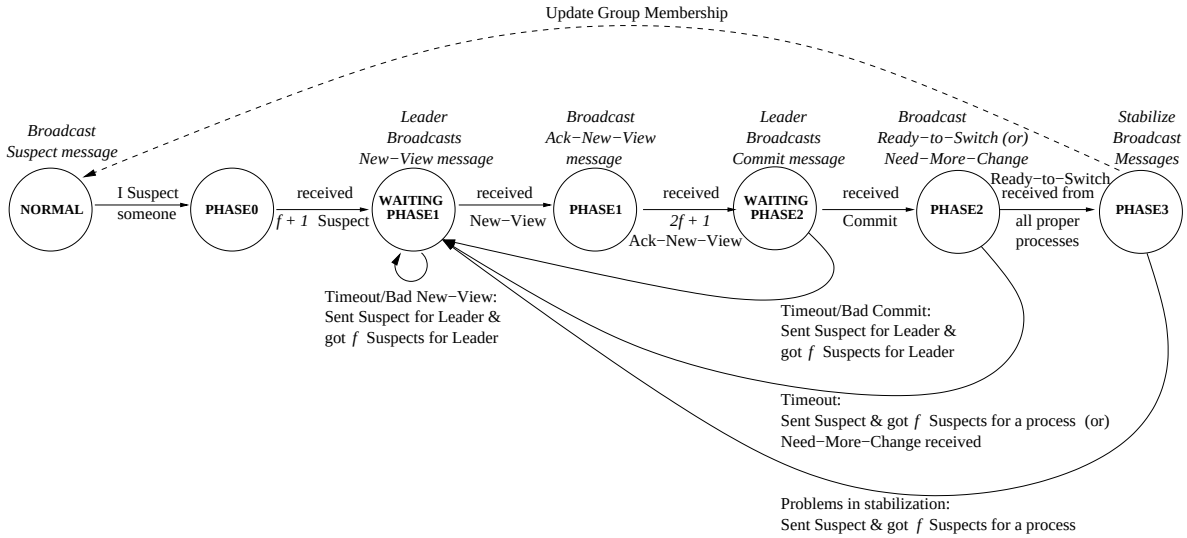


Fig. 1 Finite State Automaton for View Installation

When a non-leader process in the group has seen $f+1$ *Suspect* messages for a member, it changes its state to WAITING-PHASE1. That marks the initiation of a three-phased view installation procedure, which is a series of steps at the end of which the member suspected by $f+1$ other members to be corrupt will be removed from the group. The process also starts a timer, and expects the leader of the group to take action before the timer expires. If the $f+1$ *Suspect* messages were for the leader, then the leader is suspected to be corrupt; hence, the deputy is expected to become leader and take action. When the leader sees $f+1$ *Suspect*s, it broadcasts a signed *New-View* message, which contains (1) the list of endpoints for the next view that excludes the corrupt member, and (2) justification for this exclusion in the form of $f+1$ *Suspect* messages received from the group.

When a valid *New-View* message is received, a correct process changes its state to PHASE1 and broadcasts a signed *Ack-New-View* message. If a process p_k acknowledges a *New-View* message from p_b, then it does not acknowledge any more *New-View* messages from processes of lower rank than p_b in that view. Upon receiving $2f+1$ *Ack-New-View* messages for a *New-View* message, a correct process changes its state to WAITING-PHASE2. If it is a non-leader, it starts a timer, and expects the leader to take action before the timer expires. If it is a correct leader, it broadcasts a signed *Commit* message. A valid *Commit* message contains the same view specified in the *New-View* message and includes the $2f+1$ *Ack-New-View*s as proof that the majority of the correct processes have acknowledged its *New-View* message.

When a valid *Commit* message is received, a correct process changes its state to PHASE2 and broadcasts a signed *Ready-to-Switch* message. It also starts a timer, and expects a *Ready-to-Switch* message from each member of the new view before the timer expires. When *Ready-to-Switch* messages have been received from all members of the new view, a correct process changes its state to PHASE3. The members of the current view that are also members of the next view then begin a consensus on which messages have been delivered by each of them up to this point. This is the *message stabilization phase*; it ensures that all correct processes deliver the same set of messages broadcast in the current view. After that phase, each correct process that is a member of the new view installs a new protocol stack initialized to the NORMAL state. Each of the three phases of the view installation has timers to ensure liveness. If a timer expires before the corresponding action expected from a process is observed, then a *Suspect* message is sent for the process.

2.2.2 View Installation when Multiple Faults Occur

An enhanced algorithm is used to remove the earlier assumption that no additional faults occur during a view installation. An additional fault has occurred during a view installation if $f+1$ *Suspect* messages have been received for a member that is not among those processes being removed by the current view installation.

Consider a case in which an additional fault occurred during the view installation when a correct process p_k had not yet reached the state PHASE2. When p_k receives a *Commit* message from the leader, it changes its state to PHASE2, as described in Sect.2.2.1. However, this time, it broadcasts a signed *Need-More-Change* message (instead of a *Ready-to-Switch* message, as described before) indicating that the proposed new 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 the $f+1$ *Suspect* messages

received for each of the other corrupt members. After sending this *Need-More-Change* message, p_k starts a timer, and expects a *Ready-to-Switch* message or a *Need-More-Change* message from each of the members of the view proposed by the last *New-View* message, except the corrupt ones, before timer expiry. After receiving those *Ready-to-Switch* or *Need-More-Change* messages, p_k changes its state to WAITING-PHASE1 (as shown in Fig.1). p_k then starts a timer and waits for another *New-View* message from the leader, excluding at least one more known corrupt member from the next view than it did in the last *New-View* message. The last *New-View* message received did not result in the installation of a new protocol stack. That *New-View* message and the corresponding *Commit* message (if it was broadcast) are part of what we call a *transitional view*. If the additional fault was at the leader, then a *Commit* message from the corrupt leader may never be received. If it is not received, then the deputy takes over as the new leader, changes its state to WAITING-PHASE1, and broadcasts a *New-View* message (as shown in Fig.1). Other correct processes change their states to WAITING-PHASE1, start timers, and wait for the *New-View* message from the new leader.

If the additional fault occurs when p_k is in state PHASE2 or PHASE3 (i.e., after it has responded with a *Ready-to-Switch* message), then it reverts back to the state WAITING-PHASE1. That is shown in Fig.1 by the reverse transitions from the states PHASE2 and PHASE3 to the state WAITING-PHASE1. Then, if p_k is a leader, it broadcasts a *New-View* message that excludes at least one more known corrupt member from the next view than the last *New-View* message did; if p_k is a non-leader, it starts a timer and expects to receive a *New-View* message from the leader before the expiration of the timer.

After p_k moves to WAITING-PHASE1, the view installation follows the procedure outlined in 2.2.1, with a correct process changing state to PHASE1 upon receipt of the new *New-View* message, and so on. Should $f+1$ *Suspect*s for another process, p_j, be received after this latest *New-View* message has been received, then the *New-View* message will become part of another transitional view, and a *Need-More-Change* message will be broadcast as the response to the *Commit* message in this transitional view. That would trigger the broadcast of another *New-View* message, excluding p_j and the processes excluded by the last *New-View* message. This cycle of transitional views, in which a process keeps changing its state back to WAITING-PHASE1, would continue until a *New-View* message finally excludes all known corrupt members and all three phases of the view installation are complete.

3. Verification of the protocol

In this section, we describe how we formally verified the GMP. Before taking the formal approach, we argued the correctness of the protocol informally in [9]. Despite the informal proofs, we were aware that the likelihood of subtle errors in a complex 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 stable, well-documented and widely used 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 to debug the coordination problems that those interactions may create, and to rigorously prove the correctness of such systems. SPIN accepts design specifications written in the verification language PROMELA, which has a C-like syntax. It has a powerful yet simple notation for expressing general correctness requirements: standard Linear Temporal Logic (LTL). SPIN is an on-the-fly model-checker, which means that construction of states and error-checking happen at the same time.

3.1 Specification of the protocol in PROMELA

In this section we describe how we translated the protocol described in Sect.2.2 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 in 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 Sect.2.2 . 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*[†], the faulty members could be specified to be members with ranks 0 and 3, for example. Other processes could multicast *Suspect* messages for p_3 when they are in the NORMAL state, and could multicast *Suspect* messages 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.

We need to emphasize here that we are modeling *how the correct members respond to malicious faults that may occur in a subset of the group.* We are not concerned about the specific types of faults that may occur.

An alternative approach would be to model the faulty behavior of some group members by making them do *bad* things, such as sending a corrupt message. That, in effect, would be a fault injection. However, since an intruder can manipulate a compromised process to behave in any way he or she wants, the list of all possible *bad* behaviors is huge, and simulating *all* possible bad behaviors would be extremely difficult, if not impossible. Hence, we did not take that approach.

Our approach of modeling *effects* of faults (instead of the faults themselves) 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 digital signatures to ensure that any attempts to spoof or to send mutant messages[†] 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 timeouts In our protocol, if timeout 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* timeout feature, which means that timeout occurs only if *no* process in the SPIN execution environment is executable. However, it is impossible to simulate independent timers, or asynchronous timeouts 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.

3.1.2 Simplification of the stabilization phase

In our protocol, the third phase of view installation (stabilization phase) works in close collaboration with the reliable delivery protocol to ensure that all correct members deliver the same set of messages consistently in the current view before switching to the next view. This is done to guarantee Virtual Synchrony [1].

We simplified this phase in the PROMELA specification by making each correct member just send a message of type *Phase3* with a `true` value as its payload. The `true` value indicates that the sender has completed the message stabilization phase 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 the stabilization phase, we make each member send a `true` value in its *Phase3* message.

3.1.3 Data Abstraction

Taking advantage of the abstractions/assumptions described above, we retained only the critical parts of the data structures used for the GMP messages when specifying the protocol in SPIN. We modeled the member-

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

[†]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.

ship list or the View as just one `byte` (8 bits) or `short` (16 bits), depending on whether the initial number of members in the group is ≤ 8 or > 8. The i^{th} bit in the View gives the status of process p_i. If it is set to 1, then the View indicates the exclusion of p_i from the group.

```
typedef View byte;  (for N ≤ 8)
typedef View short;  (for 8 < N ≤ 16)
```

There are seven different types of GMP messages (*Suspect*, *New-View*, *Ack-New-View*, *Commit*, *Ready-to-Switch*, *Need-More-Change*, and *Phase3*):

```
mtype = {SUSPECT, NV, ACKNV, COMMIT, RTS, NMC,
         PHASE3};
```

We modeled all GMP messages as having the form {`mtype, byte, short`}, where the three parts indicate the message type, the rank of the sender, and the payload. For a message of type `SUSPECT`, the payload indicates the rank of the suspected process. For the `NV`, `ACKNV`, `COMMIT`, and `RTS` messages, the payload indicates the corresponding `View` information. For the `NMC` message the payload indicates a revised `View` that excludes more members than the `View` in the `COMMIT` message that was last received. The payload in a `PHASE3` message carries only a `true` value, indicating the successful completion of the message stabilization phase.

3.1.4 Modeling Reliable Multicast

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. As mentioned in Sect.2.1, the GMP relies on the services of a reliable multicast protocol that must guarantee that a multicast message is delivered properly to *all* correct processes, even in the presence of corrupt senders. However, the current version of SPIN does not provide support for multicasting messages to a group. In his thesis ([13] Sect.4.11), Ruys provides various strategies for modeling multicast protocols using SPIN. Our approach of modeling reliable multicast closely follows what Ruys calls the "Multicast Service" approach. We model reliable multicast by having a separate `Multicast` process; the group members simply issue a multicast request to the `Multicast` process that is responsible for delivering the message to all other group members. We define the following channels for communication between the group members and the `Multicast` process.

```
chan to_mcast = [CHAN_SIZE1] of { mtype, byte,
                short} ;
chan from_bcast[N] = [CHAN_SIZE2] of { mtype,
                     byte, short } ;
```

Here, `N` denotes the number of processes in the group, `CHAN_SIZE1` denotes the length of the channel from the group members to the Broadcast process, and `CHAN_SIZE2` denotes the length of the channel from the Broadcast process to the members. A value of 0 for the length of the channel indicates that the channel is rendezvous or non-buffered, which means that messages pass through handshakes between the sender and receiver. A non-zero value indicates that the channel is buffered. Buffered channels can add significantly to the verification complexity, so the value must be chosen appropriately.

The `Multicast` process is defined as follows:

```
proctype Multicast() {
   mtype msg ;
   byte src, i ;
   View contents ;
   do
   ::  atomic { to_bcast ?  msg,src,contents ->
       i=0 ;
       do
       ::  i<N ->
            from_bcast[i] !  msg, src, contents ;
            i++ ;
       ::  i>=N ->
            i=0; msg=0; src=0; contents=0; break
       od
   }
   od
}
```

To reduce the possible interleavings of the executions of the `Multicast` process with other processes, the computations of the `Multicast` process are made atomic. Any process p_i can issue a multicast request by writing a message to the `to_bcast` channel. The `Multicast` process receives the message from that channel and writes the message to the `from_bcast` array of channels. Process p_i receives messages only from the channel `from_bcast[i]`.

3.2 Specifying the properties in SPIN

We now describe how each of the properties given in Sect.2.2 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:

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

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, where $f = \lfloor (N-1)/3 \rfloor$. 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:

$$\begin{aligned}&\forall\ i : 0 \ldots N-1 \bullet\ (fault\_injected[i] = false \Rightarrow \\ &\quad (my\_current\_view[i].get\_bit[i] = false))\end{aligned}$$

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

Finally, p_k sets a global Boolean 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 `all_good_proc_done` must eventually become true. That is expressed as the LTL formula $\Diamond$ *all_good_proc_done*. SPIN negates this LTL formula to form a negative correctness claim represented by a Büchi automaton [5]. The SPIN model checker will prove the absence of acceptance cycles[†] 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 Additional Optimizations

In addition to developing the abstractions given in Sect.3.1, we further optimized the specification for efficient verification as follows. Many of these additional optimizations were motivated by the recommendations given in [13] for effective verification using SPIN.

PROMELA provides the constructs `atomic` and `d_step` (deterministic step), which can be used to combine sequences of statements into one indivisible unit whose execution is non-interleaved with other processes. We used the constructs in our specification at appropriate places to help reduce the complexity of the validation model and to improve the efficiency of verification. For example, a sequence of statements that modify values in variables that are local to a process can be combined in an `atomic` construct.

Initially, we used bit-arrays such as `fault_injected` and `done` (as described in Sect.3.2) to represent the state of the system. However, SPIN maps the bit-arrays into byte-arrays, allocating 8 times as much memory for the bit array in the system state vector as was intended. Following the approach in [13], we replaced the

[†]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.

bit-arrays with *bitvectors*, which in effect are unsigned 8-bit, 16-bit, or 32-bit integers. We defined operations to individually set, reset, and test any bit in the bitvector. For example, we represented the global variable `fault_injected` as a `byte` (for $N = 7$), whose i^{th} bit indicates whether process p_i is the faulty process in a particular verification run.

For variables whose range of possible values is known (for example, the rank of a process is between 0 and $N - 1$) we used the minimum number of bits necessary to encode the values. For $N = 7$, 3 bits would be sufficient; a byte is not needed. We also used SPIN's compile-time option `-DVAR_RANGES` to compute the ranges of all variables during verification. We then used those ranges to optimize the model's state vector by using the minimum bits necessary for the variables.

Variables used only for bookkeeping operations, such as local variables in `atomic` sequences, are reset to 0 at the end of the sequences. This hides the scratch values of the variables outside those sequences.

Additionally, we also used SPIN optimizations such as partial order reduction (which is based on the observation that the property being verified may be insensitive to the order in which concurrently and independently executed events are interleaved) and state compression (in which a compressed version of the state is stored in the state space).

3.4 Experimental Setup

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 Fig.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.

3.5 Experimental Results

We carried out all experiments on an Athlon XP 2400 computer with 512MB DDR2700 SDRAM. Table 1 gives the results of the verification. All the presented results are averaged over the number of verification cycles needed for a full verification. For each verification cycle, we carried out separate experiments for verifying safety (checking assertions, deadlocks) and liveness (termination) properties. We used SPIN's `-DCOLLAPSE` compile-time directive to compress the state vector. Addition-

Table 1 Verification Results

N	Verification Mode	Property Verified	State Vector Size (byte)	Depth Reached	Number of States	Total Memory (MB)	Time (sec)
4	Exhaustive	Liveness	164	526	512	17.22	0.15
4	Exhaustive	Safety	160	444	588	17.22	0.09
7	Exhaustive	Liveness	266	2,336	17,445	26.12	1.02
7	Exhaustive	Safety	262	1,969	18,053	26.12	0.46
10	Exhaustive	Liveness	460	5,132	630,550	40.78	25.57
10	Exhaustive	Safety	456	4,155	637,976	39.75	9.57

ally, when checking for liveness, we used the `-DNOFAIR` directive to disable weak-fairness[†] (verification is much slower when weak-fairness is enabled). As was expected and as Table 1 shows, the state vector size (which gives the size of a single state), the depth reached (which gives the longest execution path), the memory required and verification time all increased dramatically with the increase in group size. All the verification cycles completed successfully without any errors. The results strengthen our claim that the GMP indeed provides the properties stated in Sect.2.2. However, the most interesting aspect of this effort was some observations we made that led to increased efficiency of the protocol.

In the original protocol specification, we had a data structure called *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* in a series of transitional views. During the SPIN validation effort, 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.

3.5.1 Services Required by the GMP

The GMP requires the services of a reliable multicast protocol for its correct operation. This dependency is fairly easy to deduce from the finite state machine of the protocol (Fig.1): the GMP requires *Ready-to-Switch* messages from *all* correct processes before it can complete. We easily checked the dependency in our SPIN verification by making the `Multicast` process skip some messages. During verification of safety properties, SPIN reported the existence of invalid end states; and when verifying liveness, SPIN reported the existence of an acceptance cycle. These in effect mean that the protocol did not terminate.

Our PROMELA model of the reliable multicast described in Sect.3.1.4 is very efficient, but it enforces a stronger guarantee than the GMP requires: it enforces a total order on all messages delivered at processes, in addition to guaranteeing reliable delivery. To verify that our GMP does *not* require total ordering, we repeated our experiments, but with two `Multicast` processes instead of one. For a particular {source, destination} combination, the routing `Multicast` process remains the same during the verification run. This ensures a FIFO ordering, but allows for messages from two processes to be delivered in different orders at various processes. Using this modified setup, we carried out our experiments, and verified that our GMP requires only reliable delivery, not total ordering.

4. Conclusion

In this paper, we presented a high-level description of 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 [14]

[†]A computation is weakly fair, if every process that is continuously enabled from any point in time is executed infinitely often.

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. To handle the state-space explosion problem, both the model and the property to be verified need to be coded as efficiently as possible. We described in detail how we abstracted our GMP to come up with a verification model of the protocol and the properties it claims to provide. The model had to closely resemble the specification and functionality of the actual protocol, and also be is efficient to verify. 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.

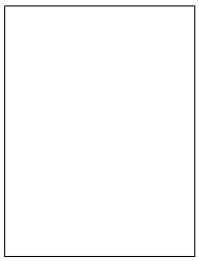
We must point out that what we verified with SPIN was the *abstracted* version of our actual protocol. Abstraction is essential in verifying a complex system such as ours. However, abstraction involves the hiding of details, which means that hidden details are not verified. For example, we are not verifying 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.

Acknowledgements

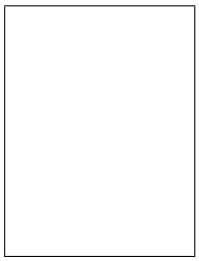
We thank Prof. Michael Loui for his useful discussions, and Jenny Applequist for helping us improve the readability of the paper. We also thank the anonymous reviewers for their help in improving the manuscript.

References

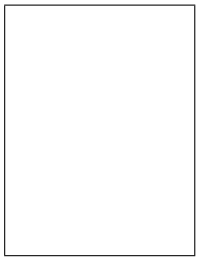
[1] K.P. Birman, Building Secure and Reliable Network Applications, Manning Publications, 1996
[2] F. Cristian and C. Fetzer, "The Timed Asynchronous Distributed System Model," IEEE Trans. on Parallel and Distributed Systems, Vol. 10, No. 6, pp. 642–657, 1999
[3] M. Cukier, J. Lyons, P. Pandey, H.V. 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 Conf. on Dependable Systems and Networks, pp. B64–B65, 2001
[4] 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, 1985
[5] G.J. Holzmann, "The Spin Model Checker," IEEE Trans. on Software Engineering, Vol. 23, No. 5, pp. 279–295, 1997
[6] K.P. Kihlstrom, L.E. Moser, and P.M.Melliar-Smith, "The SecureRing Protocols for Securing Group Communication," Proc. 31st IEEE Hawaii Intl. Conference on System Sciences, pp. 317–326, 1998
[7] A. Pnueli, "The Temporal Logic of Programs," Proc. 18th IEEE Symposium on Foundations of Computer Science (FOCS 1977), pp. 46–57, 1977
[8] P. Pandey, "Reliable Delivery and Ordering Mechanisms for an Intrusion-Tolerant Group Communication System," MS Thesis, University of Illinois at Urbana-Champaign, 2001
[9] H.V. Ramasamy, "Group Membership Protocol for an Intrusion-Tolerant Group Communication System," MS Thesis, Univ. of Illinois at Urbana-Champaign, 2002
[10] H.V. Ramasamy, M. Cukier, and W.H. Sanders, "Formal Specification and Verification of a Group Membership Protocol for an Intrusion-Tolerant Group Communication System," Proc. 2002 Pacific Rim International Symposium on Dependable Computing (PRDC-2002), pp. 9–18, 2002
[11] H.V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W.H. Sanders, "Quantifying the Cost of Providing Intrusion Tolerance in Group Communication Systems," Proc. 2002 International Conf. on Dependable Systems and Networks (DSN-2002), pp. 229–238, 2002
[12] M.K. Reiter, "A Secure Group Membership Protocol," Proc. IEEE Symposium on Research in Security and Privacy, pp. 176–189, 1994
[13] T.C. Ruys, "Towards Effective Model Checking," PhD Thesis, University of Twente, Enschede, The Netherlands, 2001
[14] F. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," ACM Computing Surveys, Vol. 22, No. 4, pp. 299–319, 1990



HariGovind V. Ramasamy received his B.E. degree in Computer Science and Engineering from Anna University, India in 1999, and his M.S. in Computer Science from the University of Illinois in 2002. He is currently a Ph.D. student in Computer Science at the University of Illinois. His research interests include intrusion-tolerant protocols, and validation of survivable systems.



Michel Cukier received a physics engineering degree from the Free University of Brussels, Belgium, in 1991, and the Doctor in engineering degree from the National Polytechnic Institute of Toulouse, France, in 1996. He is now an Assistant Professor in the Department of Mechanical Engineering at the University of Maryland, College Park. His research interests include intrusion tolerance, evaluation of fault-tolerant systems combining modeling and fault injection, and the estimation of fault tolerance coverage.



William H. Sanders is a Professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory at the University of Illinois. He has published more than 100 technical papers in the areas of performance/dependability evaluation and reliable distribution systems. He is Vice-Chair of IFIP Working Group 10.4 on Dependable Computing, and serves on the Board of Directors of ACM Sigmetrics and the editorial board of IEEE Transactions on Reliability. He is the Area Editor for Simulation and Modeling of Computer Systems for the ACM Transactions on Modeling and Computer Simulation. He is a Fellow of the IEEE.