MojaveComm: A Robust Group Communication Library for Grid Environments

Cristian Ţăpuş, David Noblet, and Jason Hickey
Computer Science Department
California Institute of Technology
{crt,dnoblet,jyh}@cs.caltech.edu

Abstract

This paper introduces a fault-tolerant group communication protocol that is aimed at grid and wide area environments. The protocol has two layers. The lower layer provides a total order of messages in one group, while the upper layer provides an ordering of messages accross groups. The protocol can be used to implement sequential consistency. To prove the correctness of our protocol we have used a combination of model checking and mathematical proofs. The paper also presents the behavior of our implementation of the protocol in a simulated environment.

1. Introduction

Distributed systems, historically, have proved to be both difficult to implement and difficult to reason about. In particular, it is not easy to develop systems that simultaneously have good usability, scalablility, reliability/fault-tolerance, and performance characteristics. Since these properties often tend to work in opposing directions, many current systems choose to sacrifice one in favor of another.

The focus of this paper is on the implementation of a distributed group communication protocol [13] for GRID environments. This protocol has two layers: the first layer provides atomic multicast within a group of nodes; the second layer, sitting on top of it, guarantees the causal ordering of messages sent in overlapping groups.

We designed this protocol as part of our effort to implement a distributed objects system for GRID environments. In order to maintain the consistency of objects in the presence of data replication and concurrent accesses we required an efficient protocol to enforce an order on the operations performed on the shared objects. Furthermore, we wanted to provide a natural semantics for these accesses in order to facilitate reasoning about the integrity of the data.

One compromise that designers of distributed systems consistently make in order to improve performance is the adoption of relaxed consistency models. Unfortunately, the use of such relaxed semantics often requires the user to have a deep understanding of the underlying system in order to write correct applications. The alternative is to adopt a strict consistency model, such as sequential consistency [8], which preserves the order of accesses specified by the programs. It is widely accepted that this is the simplest programming model for data consistency. In practice, the sequential consistency model has been reluctantly adopted due to concerns about its performance. However, recent work in compilers for parallel languages [7] has shown that sequential consistency is a feasible alternative. Our work introduces a communication infrastructure that enables the implementation of the sequential consistency model in a distributed environment.

Consensus protocols play an important role in maintaining consistency of replicated data and in solving the problem of concurrent accesses to shared resources in the GRID. Atomic multicast is a useful tool for implementing consensus protocols in such environments, as it is a natural model for providing fault-tolerant communication.

The issue of reliability is another critical factor in the development of distributed systems. While the main concern in designing such systems is coping with external failures it is often the case that programming bugs and unpredicted corner cases generate many of the problems. In order to combat this issue, we make use of formal verification mechanisms to guarantee the correctness of our protocol.

The major contributions of this paper are: the design and implementation of a totally distributed (no central point of failure) group communication protocol with guarantees for a total order of messages, and the use of formal methods to show the correctness of the implementation.

The paper is organized as follows. First, we introduce the protocol we designed. Next, we present the two model checkers we use and their contributions with respect to the correctness of our protocol. Finally, we present preliminary experimental results using our prototype implementation and conclude the paper by discusing differences between our approach and other group communication libraries.

0-7695-2622-5/06/$20.00 (c) 2006 IEEE

2. Protocol description

This section presents the two layers of our protocol. We have shown the correctness of the upper layer of our protocol and proved the guarantees that it makes in a previous work [13]. In this paper we focus on the correctness of the lower layer and present how its distributed approach makes it robust in the presence of failures.

Although our protocol was designed to be abstract and applicable in many situations, one of the main goals of the project was to develop a communication infrastructure for GRID services and to use it to build a distributed shared objects system.

2.1. Overview

The system is composed of a set of processes. Processes have a unique identifier that is persistent across failures. We define a *group* to be a set of processes. A *view* of a group is defined as a subset of the group membership.

Processes can belong to several groups at the same time. The upper layer of the protocol relies on the existence of a total order of messages sent within each group that is provided by the lower layer of the protocol. Additionally, it requires that messages sent by one process to different groups become part of the total order in each group in the same sequence in which the messages were issued. For example, if a process were to send two messages, m_1 and m_2, to groups g_1 and g_2 respectively, then message m_1 must become part of the order in group g_1 before m_2 becomes part of the order in group g_2. It is important to notice that we do not require that m_1 is delivered before we can send m_2; rather, we simply require that m_1 obtains a sequence number before m_2 does. While there is a penalty for implementing this restriction we do minimize it by separating the tagging of messages with sequence numbers from the actual message delivery.

Each group can have several views and the views can overlap. Views are local to processes and represent their image of the membership of the group. In each group we implement the Virtual Synchrony [3] model. When a request to join a group is received by a process or when a process leaves or is detected to have failed a view change is triggered in the group. The view change is a synchronization point for the processes that survive it. It guarantees that membership changes within a process group are observed in the same order by all the group members that remain in the view. Moreover, view changes are totally ordered with respect to all regular messages that are sent in the system. This means that every two processes that observe the same two consecutive view changes, receive the same set of regular messages between the two view changes. Each message sent in a view is characterized by an epoch corresponding

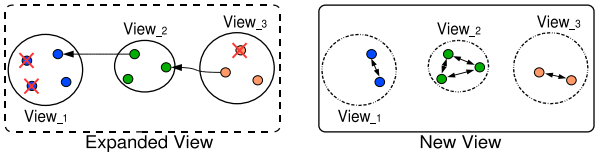

Figure 1. The view change event.

to the view in which the message was sent, and by a group sequence number. The epochs and the group sequence numbers are monotonically increasing during the life of a process, but they are not persistent across failures.

Each process that wants to join a group starts as a singleton view. It then contacts a directory service that provides hints on the current membership of the group. These hints are used by the process to join other processes that belong to the same group. At this layer of the protocol we do not discriminate between different views of a group. It is only at the upper layers that we treat separate views of the same group differently based on criteria that we deem appropriate. For example, we might want to allow only the members of a given view (like the one with heighest cardinality) of a group to send or receive messages.

2.2. The View Change

A view change is triggered by one of two events: a join request or a leave. A join request is sent by the member of one view to members of another view of the same group in an effort to merge the two views. We do not allow views to split at will. If a process wants to leave a group it sends a leave request to the members of its current view. If a process is detected to have failed, then one or more members of the view initiate a view change and try to exclude the process from the view.

Upon receiving a join or a leave request a process initiates the view change procedure. We employ a resolution mechanism for concurrent view changes in the same group involving an overlapping set of processes. This resolution mechanism allows only one of the initiators to succesfully complete the view change. However, we do permit concurrent view changes in disjoint views to evolve in parallel. One major advantage of our protocol is that it allows changes in view membership to include, at the same time, multiple processes joining and leaving the view. For example, Figure 1 illustrates how, in one step, three views merge and certain nodes from each view will be excluded as they fail during the view change process.

The view change is done in several stages: the *expanding stage*, followed by the *contracting stage*, the *consensus stage* and finally the *commit stage*. Figure 2 illustrates the states that a process can be in during the view change process, along with the events that trigger the transitions from

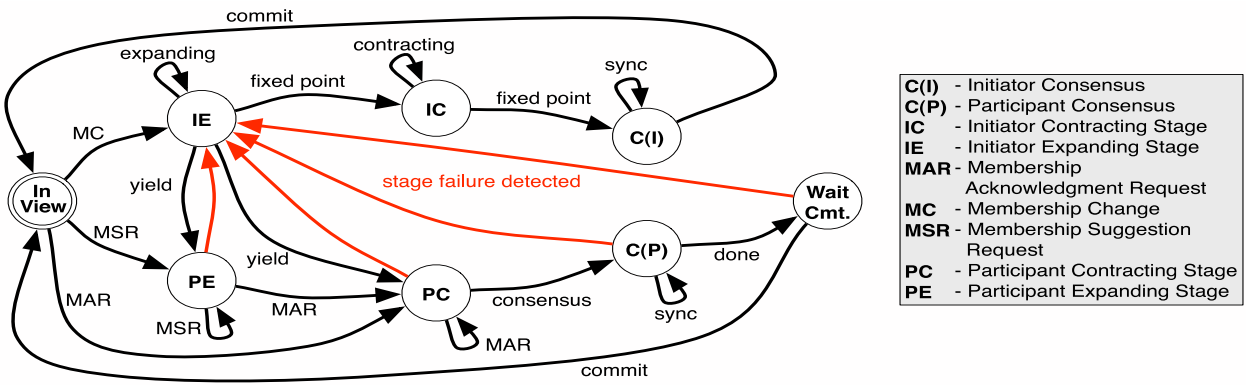

Figure 2. The state machine for our protocol.

one state to another. The initial state is the *In View* state. On detecting a membership change (MC) a process becomes an initiator of a view change and initiates the expanding stage (IE). The other nodes involved in the view change would be participants in the expanding stage (PE). If the stage ends succesfully the initiator becomes the initiator of a new view as part of the contracting stage (IC), while the other nodes become participants in the new view during the contracting stage (PC). The consensus stage is represented by two states, depending on the previous state of the process (C(I) and C(P)); the participants then wait for the final commit stage. Each stage is detailed below.

The purpose of the first stage of the view change process is to collect suggestions from the current members of the view and, from these suggestions, ascertain what the new membership of the view should be. This stage is repeated until a fixed point is reached (nodes are only added to the membership with each round of suggestions). In the example presented in Figure 1 this is shown in the top drawing. At the end of the expanding stage, the expanded view contains all the members of the initial three views.

During the contracting stage, processes that have failed or that want to leave the group are removed from the maximal fixed point view reached during the previous stage. The goal of this stage is to reduce the membership of the view to the current set of active processes. It is important to note that between a process's interest in joining a group and the commit of the view change that process could fail or there might be a network partition, in which case more than one process might need to be excluded from the view. In Figure 1 the *new view* illustrates the membership after the contracting stage, where failed nodes have been evicted from the view.

The consensus stage is critical for preserving the properties of the Virtual Synchrony model. During this stage processes that have survived so far agree on what messages they need to deliver to the application layer to guarantee that all members of the view that survive the view change have delivered the same set of messages in the previous view. The consensus stage is illustrated by the arrows in the second drawing of Figure 1, where each surviving process synchronizes with all the other processes in its previous view.

In the last stage, the new view to be installed is broadcast to all of its members and is locally installed by each member. The view change initiator sets the epoch of the new view to be larger than the largest epoch involved in the view change. Also, the sequence number is reset to 0 and a ViewChange message with the new epoch and the new sequence number is broadcast to all members of the new view. Upon receiving the ViewChange message each process delivers it to the application (the upper layer running on top of the group communication).

2.2.1 Surviving failures

To detect network or process failures we introduce a set of failure detection mechanisms that dynamically adapt to the environment. We expect acknowledgements for the messages sent in each group and we use heartbeat messages to detect failures during times of network inactivity. Our failure detectors can be dynamically changed to report on what is considered to be an acceptable latency or a tolerated loss rate before the processes composing the system are evicted from views.

On the failure of a process the appropriate action is taken by the process that detects the failure. This depends on the current state of the detector process. Figure 2 shows in red (light color) the actions triggered by our failure detection mechanism. When a critical node fails during the various stages of a view change it prompt a process to initiate a new view change. Generic node failures are reported during the contracting stage.

2.2.2 Providing sequential consistency

The existence of shared data and of concurrent processes that can access it prompts the need for using a data consis-

tency model. Using our two-layered group communication protocol, we can easily provide sequential access to shared data. We organize processes in groups based on the shared data they are interested in accessing. Thus, each group will be associated with a piece of shared data. For simplicity, we call this shared data an object. We map the "opening" and "closing" operations on objects to the join and the leave operations in the corresponding group. Thus, to access the information of a shared object, processes dispatch read and write messages to the appropriate group. One of the key features of our protocol is the presence of *explicit read messages*. The read messages act as synchronization points for the objects and guarantee that when they are processed all previous changes to the object have been seen. Mathematical proofs that show that our protocol guarantees sequential consistency can be found in [13].

3. Formal Verification

Despite the complexity of the protocol, the approach we took in the design phase allowed us to formally verify the correctness of the protocol and reason about its overall behavior. As such, the protocol we have presented is the result of a systematic approach. We started by designing the protocol on paper. Then we moved to creating models with different levels of abstraction. And, at the end of the process, we translated the model into a real implementation. This section discusses the modeling and translation phases.

3.1. Model Checking

While it is common practice to study algorithms and protocols for a distributed environment by formally describing them in a mathematical framework and proving properties about the resulting models by hand, this approach is often not feasible to apply to systems of any real size and/or complexity (especially when failures are to be considered) as the process is highly labor-intensive.

A more automated alternative to writing formal proofs on paper is to use a model checking tool. In model checking, one develops a model of some algorithm of interest in a model checking language and the resulting code can be run through the associated model checking tool. This model checking tool then investigates all reachable states of a system and is able to verify safety and liveness properties according to some user-defined specification.

This approach provides a high level of confidence in the correctness of the algorithm if the right model is employed (i.e. a model where it is feasible for the tool to visit all or most of the reachable states). Furthermore, the automated nature of model checking makes this method of verification easy to use and, since model checking languages often resemble conventional programming languages, model checking provides a representation of the system that can closely mirror an actual implementation.

```
1:  inline initiate_view_change (id, view, joined, left) {
2:      become_member(inside_vc, id);
3:      become_member(initiated_vc, id);
4:      remove_member(ivc_failed, id);
5:      ivc_acks[id]=0;
6:      temp_view = diff_views(view,left);
7:      sent_view[id] = merge_views(temp_view, joined);
8:      increment_lseqn(id);
9:      /* Initiate a view change */
10:     bcast_msg(Initiate_vc, sent_view[id], lseqn[id]);
11: }
```

Figure 3. Illustrating the Spin model.

```
1:  InitiateViewChange(i, view, joined, left) ≜
2:    ∧ InsideViewChange′ =
          [InsideViewChange EXCEPT ![i] = TRUE]
3:    ∧ InitiateViewChangeAcks′ =
          [InitiateViewChangeAcks EXCEPT ![i] = {}]
4:    ∧ HasInitiateViewChangeFailed′ =
          [HasIVCFailed EXCEPT ![i] = FALSE]
5:    ∧ LET newView ≜ (view \ left) ∪ joined
6:          msg ≜ [PID ↦ i,
                   lseq ↦ LocalSequenceNumber[i],
                   type ↦ InitiateViewChangeMsg,
                   data ↦ newView]
7:      IN  ∧ BroadcastMessageTo(msg, newView)
8:          ∧ InitiatedViewChange′ =
                [InitiatedViewChange EXCEPT
                 ![i] = newView]
```

Figure 4. Process 'i' initates a view change where it wants to add nodes in 'joined' and exclude nodes in 'left'. TLA Model.

In order to verify our protocol we decided to use two different model-checkers: Spin and TLC. The reason for combining the two is that each has certain limitations that are better addressed by the other. Thus, by combining the use of the two model checkers we were able to get a more extensive verification of the models we developed.

3.2. Spin

Spin [6] is a model checker that uses a C-like specification language, called Promela. Promela is a nondeterministic language based on Disjkstra's guarded command language notation. It supports primitives like buffered and un-buffered communication channels, thus making it a good choice for desigining models that are close to an actual implementation.

```
1:   let initiate_view_change gdesc view =
2:         gdesc.vc.inside_view_change ← true;
3:         gdesc.vc.acks ← ProcSet.empty;
4:         gdesc.vc.has_stage_failed ← false;
5:         (* calculate new view *)
6:         let new_view = ProcSet.union
7:               (ProcSet.diff view gdesc.vc.leave_nodes)
8:               gdesc.vc.join_nodes in
9:         let vc_msg = ControlMsg(Initiate_vc (new_view)) in
10:              send_msg_to_group vc_msg new_view gdesc
```

Figure 5. OCAML code for the actions taken to initiate a view change

3.3. TLA/TLC

TLC stands for "The TLA+ Model Checker". It is a model checker for specifications written in TLA+. TLA [9](Temporal Logic of Actions) is a logic for specifying and reasoning about concurrent systems that was developed by Leslie Lamport, and it is the basis for TLA+ [10].

The TLA+ specification language uses a mathematical syntax resembling that of logical formulae. The advantage to this is that, in many cases, an algorithm can be specified clearly and concisely without the need to worry unnecessarily about low-level implementation details that are not critical to the algorithm's operation.

3.4. Implementation

Using the two model checking tools we were able to verify some properties of our protocol, such as: in the absence of failures, a set of nodes belonging to views that are attempting to merge will eventually form a single view consisting of all of these nodes; and that, even in the presence of failures, a view change will not take infinite time.

In Figures 3, 4, and 5, we demonstrate the straightforward nature of the translation from the models to our OCaml implementation. The code snippets presented show the actions taken by a process in order to initiate a view change; the code for the models matches almost line-by-line with the implementation. This easy mapping of the formal models into a running implementation increases confidence in the correctness of our implementation as this close correspondence suggests that properties that hold for the models will also hold for the implementation.

4. Experimental results

This section presents a set of preliminary experimental results that focus on the lower layer of the group communication protocol. We have run our protocol in an emulated environment where we started 64 identical processes. We monitored the processes as they were trying to form one single view of the same group. Figure 6 shows the number of view changes that occured from the perspective of one of the processes and the membership of each of the view. This graph shows us that while this node was taking part in view changes there were parallel view changes involving other nodes that eventually merge into a single view. The second graph, presented in Figure 7, shows the number of view changes occuring in each one-second interval from when the first process was started until the final view change, comprised of all processes, was installed. It is important to keep in mind that our emulation environment had a few shortcomings. For example, it restricts the parallelism of the execution due to limited shared resources and it delays the start of a few of the processes until near the end of the experiment.

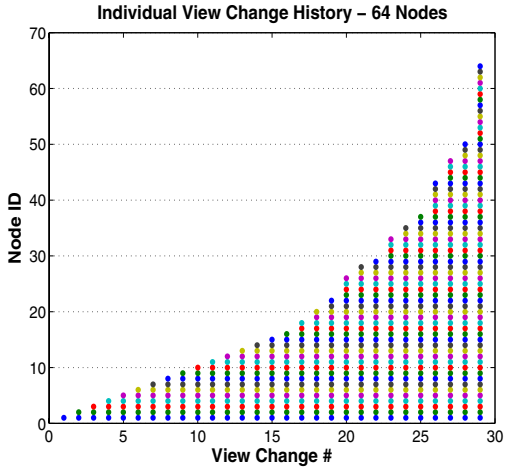

Figure 6. The membership of the view changes from the perspective of one node during the deployment of 64 nodes.

5. Related work

The idea of using formal methods to prove the correctness of distributed protocols is not new. Ensemble [5], and its predecessor Horus [11], used the PRL [12] theorem prover to show the correctness of the group communication protocols provided by it. The long and tedious process of formalizing the protocols also required a formalization of a significant subset of the language used to implement it, ML [4]. While this mechanism worked for Ensemble, where the high level of modularity in the system and the choice of the implementation language played a large role, this would not easily apply to protocols implemented in languages such as C or Java. Furthermore, more complex systems would be even harder to formalize. Also, our

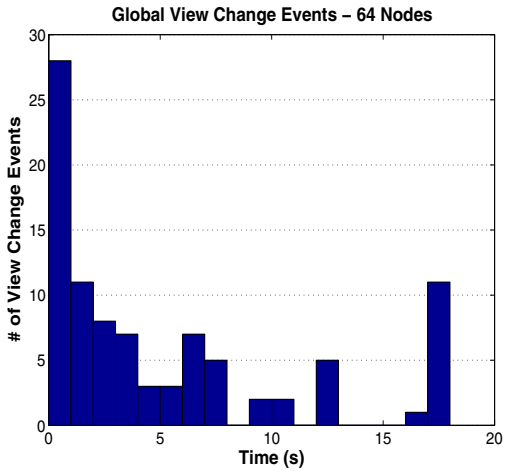

Figure 7. The number of view changes occuring in each 1s interval during the deployment of 64 nodes.

group communication protocol has taken on a few of the challenges neglected by Ensemble, like handling multiple join and leave events simultaneously and allowing merges of overlaping views.

Spread [1], another group communication system, tries to address the problem of communication in the context of wide area networks. While Spread is a popular toolkit, its implementation is not very close to the abstract formal model discussed in the design paper. Over time, this has led to some degree of confusion.

Newtop [2] provides a solid mathematical model for proving the correctness of the group communication protocol that it uses. However, implementations have been slow to be developed.

Finally, we want to mention the Message Passing Interface (MPI), the "de-facto" standard for communication APIs in GRID systems. One of the problems of MPI stands in that the specification of the behavior and API of the system is too loose, which has led to various interpretations that mapped into sometimes incompatible implementations.

6. Conclusion and future work

We introduced a new fault-tolerant group communication protocol that uses adaptive failure detection mechanisms to run over both LANs and WANs, making it a good match for applications in GRID environments. The protocol uses a two layer architecture. The lower layer guarantees the total order of messages in single groups, while the upper layer guarantees the inter-group ordering of messages. We used model checking and formal proofs in the design phase of our protocol to develop an algorithm that we were able to prove correct.

Based on our experience with desigining this protocol using model checkers we are interested in building collections of components that can be used in designing other protocols, without having to re-invent the wheel. Furthermore, we plan on implementing a distributed shared object system which guarantees sequential consistency of concurrent accesses to objects based on our group communication protocol. We also want to use the protocol to implement the communication layer of a distributed system that uses speculative execution to improve the performance of distributed applications.

References

[1] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton. The spread toolkit: Architecture and performance. Technical Report CNDS-2004-1, John Hopkins University, 2004.

[2] P. D. Ezhilchelvan, R. A. Macêdo, and S. K. Shrivastava. Newtop: a fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, page 296. IEEE Computer Society, 1995.

[3] R. Friedman. Using virtual synchrony to develop efficient fault tolerant distributed shared memories. Technical report, Ithaca, NY, USA, 1995.

[4] R. Harper, D. MacQueen, and R. Milner. Standard ML. Technical Report TR ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, University of Edinburgh, 1986.

[5] M. Hayden. The Ensemble system. Technical Report TR98-1662, Cornell University, 1998.

[6] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[7] A. A. Kamil, J. Z. Su, and K. A. Yelick. Making sequential consistency practical in titanium. In *The International Conference for High Performance Computing and Communications, SuperComputing 2005*, 2005.

[8] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Trans. Comput. C*, 28(9):690–691, 1979.

[9] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[10] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[11] R. V. Renesse, K. P. Birman, B. B. Glade, K. Guo, M. Hayden, T. Hickey, D. Malki, A. Vaysburd, and W. Vogels. Horus: A flexible group communications system. Technical report, Ithaca, NY, USA, 1995.

[12] T. N. Staff. PRL: Proof refinement logic programmer's manual (Lambda PRL, VAX version). Cornell University, Dept. of Computer Science, 1983.

[13] C. Ţăpuş, A. Nogin, J. Hickey, and J. White. A simple serializability mechanism for a distributed objects system. In D. A. Bader and A. A. Khokhar, editors, *Proceedings of the 17th International Conference on Parallel and Distributed Computing Systems (PDCS-2004)*. International Society for Computers and Their Applications (ISCA), 2004.