Experimental Evaluation of the Unavailability Induced by a Group Membership Protocol

Kaustubh R. Joshi[1], Michel Cukier[2], and William H. Sanders[1]

[1] Coordinated Science Lab., Dept. of Electrical and Computer Engineering, and Dept. of Computer Science, University of Illinois, Urbana IL 61820, USA.
{joshi1,whs}@crhc.uiuc.edu
[2] Dept. of Materials and Nuclear Engineering, University of Maryland, College Park MD 20742, USA. mcukier@eng.umd.edu

Abstract. Group communication is an important paradigm for building highly available distributed systems. However, group membership operations often require the system to block message traffic, causing system services to become unavailable. This makes it important to quantify the unavailability induced by membership operations. This paper experimentally evaluates the blocking behavior of the group membership protocol of the Ensemble group communication system using a novel global-state-based fault injection technique. In doing so, we demonstrate how a layered distributed protocol such as the Ensemble group membership protocol can be modeled in terms of a state machine abstraction, and show how the resulting global state space can be used to specify fault triggers and define important measures on the system. Using this approach, we evaluate the cost associated with important states of the protocol under varying workload and group size. We also evaluate the sensitivity of the protocol to the occurrence of a second correlated crash failure during its operation.

1 Introduction

Group communication is an important paradigm for building dependable distributed systems. Reliable system designers often use the dependability-related properties of group communication systems (GCS) to make claims about the dependability of their own applications. This makes the verification and assessment of these properties an important endeavor. Consequently, formal specification and verification of GCS properties is a very active area of research [28, 12, 22, 17]. Work has also been done on automated construction of such specifications from source code [19]. These techniques have generally lacked the ability to assess the performance of GCS implementations. However, the performance of GCS components can have a significant impact on a system's dependability properties. For example, applications that require a virtual synchrony model [4] force the GCS to block communication when membership changes occur. This blockage can cause a loss of availability of some or all components in the system, thus reducing its overall dependability. Thus, the performance of the group membership component has a direct impact on dependability. The degree of this impact depends not only on the protocols and algorithms used, but also on the actual implementation.

That observation leads to the necessity of experimental evaluation of the dependability-related performance characteristics of group membership protocols to complement formal verification. Although there has been some work on evaluation of group communication protocols [21, 26], the primary focus has been on performance of message-ordering and delivery protocols. In this paper, we present an experimental evaluation of the blocking behavior of a widely used group membership protocol. Our evaluation sheds light on how the availability of the system is affected under varying operating conditions. We use the Ensemble GCS as the basis for our experiments.

Ensemble [27] is a popular group communication system developed at Cornell University. It was written in the OCAML dialect of the ML language so that it would be amenable to automated proof checking [16]. Ensemble communication stacks have a modular structure and are built up of several layers of micro-protocols stacked on each other. Ensemble allows applications to specify which micro-protocol layers to use in their stacks, thus allowing applications to choose group properties. Layers implement several message-ordering properties, group membership properties, and virtual synchrony. The Ensemble group membership protocol supports member addition and removal as well as partition merging, failure detection, and suspicion sharing. We use a stack that provides membership properties, virtual synchrony, and a sequencer-based total-ordering protocol. We treat events that cause membership changes in the system as faults, because they cause the system to go into an undesired (blocking) state, and we treat the group membership protocol as the fault tolerance mechanism. Consequently, we use the technique of fault injection [1] to exercise the Ensemble membership protocol and subsequently perform distributed measurements on the system to evaluate blocking time.

A large body of work exists on both fault injectors for networked systems [24, 9, 11, 13] and fault injection, including some work on fault injection of group communication systems [2, 10]. However, past injection efforts have concentrated either on assessment of statistical dependability metrics such as coverage [2, 25, 8], or on fault removal [10]. The problem of assessing blocking behavior for a distributed system such as Ensemble poses significant new problems. First, since this evaluation aims to get an insight on system availability under specific conditions, statistically driven fault triggers (such as those in [23]) cannot be used. Due to the distributed nature of the membership protocol, fault triggers based on local state of nodes [9] are also unable to represent all the configurations in which faults may be injected. Second, due to the need to compute a global measure such as the blocking time for the entire group, a facility for distributed measurement and measure specification is needed. The topic of distributed system measurement and performance evaluation has been considered in the literature [3, 20]. However, we are not aware of any past work that has combined the use of fault injection with distributed measurements to assess dependability characteristics of a real system.

To solve both problems, we use a novel global-state-based fault injection technique. The technique involves abstracting the operation of each node in the group by a state machine. A global state for the entire system is defined in terms of the local states of the nodes. Fault triggers are expressed in terms of this global state. Measurements of local state changes are taken, and combined to form a timeline of global state changes on which measures can be defined. We use Loki [6], a fault injector that supports global-state-based fault triggers and measure estimation, to conduct our experiments. This approach enables the injection of faults precisely in interesting states and the subsequent collection of fine-grained measures of the system.

As a case study, this paper presents a detailed state machine specification of the operation of the Ensemble group membership protocol, provides insight into its operation, and quantifies, via experimental results, the cost of group membership operations. In addition, this work serves as a real-life demonstration of how global-state-based fault injection coupled with distributed measurement can be used to assess the performance of complex distributed systems. In doing that, we also demonstrate how highly layered event-driven protocol structures may be modeled using state machines and how the resulting global state-space can be used to define scenarios for fault injection and to specify system measures. Finally, the results of experimentation provide insight on how the performance of the group membership service of a GCS can affect its availability.

2 Methodology and System Model

This section describes the fault injection methodology and fault models we use for this study. The target of the fault injection is the Ensemble group membership protocol, which can only exist in, and be exercised by, an application that instantiates the Ensemble stack. Hence, this section also describes the application we use to drive the Ensemble stack.

Global-State-Based Fault Injection Methodology Our methodology involves abstracting the parts of the distributed system that are to be measured by a collection of state machines; one for each node. The global state of the system is then defined as the collection of the states of these state machines. The fault triggers consist of the name of the node on which to inject the fault, and a boolean expression on the global state which, when true, triggers the fault. During experiment execution, the Loki fault injector is used to track the state machine states, and record state-transition-event occurrence times using the local clocks of the nodes. This requires that the application be instrumented to notify Loki of local events that cause state transitions. Since the Loki runtime only exports a C interface, we wrote a C-ML wrapper for this interface, so that notifications could be done natively from the Ensemble stack. After the experiments are conducted, the local transition times are combined into a global execution trace called the *global timeline*, which contains occurrence times of all state transition events that occurred in the system as measured by a single "global clock". Loki uses an offline clock synchronization algorithm to generate the global timeline [7], and hence the physical clocks of the hosts used for the experiments need not be synchronized. Measures are defined in terms of the global state [5], and are computed using the global timeline.

Fault Model In this case study, we treat the group membership protocol as a fault tolerance mechanism and seek to evaluate it as such. Hence, we do not consider faults, such as message corruption, delays, and packet loss, that have been traditionally used for fault-injecting communication systems [10], and restrict ourselves to the "faults" that the membership protocol is designed to handle.

The events that trigger the membership protocol are node crashes, network partitions, node joins and departures, and the merging together of different groups. However, a network partition is handled as a series of node crashes. Nodes leaving the group are removed through the same protocol used for crash failures, except that there is no crash detection timeout involved. Also, node joins are implemented as the merging of two groups: the primary group, and a singleton group consisting of just the new member. Hence, the crash failure and node join events are representative of all membership change events, and the fault model for this study is restricted to these events. Due to space limitations, we present results only for crash failure injections. The experimental results for node joins can be found in [18].

Application Model One of the most important characteristics of an application for a group communication system is the message workload it generates, and this is the only characteristic we consider. The message workload is specified in terms of the length, type (point-to-point or broadcast), and generation times of the messages generated. In our experiments, we consider 100-byte-long fixed-sized broadcast messages. We model the application message generation process at a single node (the workload) by a Poisson process with parameter λ, which is varied in different studies. Hence, to drive the experiments, we wrote a simple ML application that instantiates an Ensemble stack, and generates the Poisson workload once a group of a specified sized has formed. The application uses the compatibility application interface provided by the Ensemble distribution [15] and runs on a standard virtual synchrony stack with the sequencer-based total ordering protocol included.

3 State Machine Abstraction

This section presents the methodology used to construct a state machine abstraction of the group membership protocol, and then presents the resulting Loki state machine. Since we are not aware of a complete description of the Ensemble group membership protocol in the literature, we also briefly describe the protocol itself, to help the reader better understand the results.

3.1 Methodology for State Machine Construction

Construction of a state machine to represent the temporal behavior of a highly layered, event-driven stack like Ensemble is difficult due to a lack of order between layer executions [16] and the simultaneous existence of multiple event flows in the stack. Existing specifications have either not considered the interaction between layers [14], or not captured the temporal behavior of the protocol [17]. One approach would be to model each layer in isolation, and compose the resulting state machines. However, such a composition must be guided by a functional model of the protocol behavior (such as [17]) to constrain the state-space explosion that results due to the large number of layers. Since only non-functional measures (such as blocking time) are of interest for this study, we avoid the need for a functional model by using a novel approach based on "event chains."

An external event to incorporate into the model (e.g., a crash failure) is first chosen, and an event chain is constructed for it. An *event chain* is a directed graph in which the vertices represent inter-layer or external events and the edges represent direct causal relationships between the events. An incoming event at a layer may generate several outgoing events, and, conversely, an outgoing event at a layer may be generated as a result of several incoming events. Event chains represent the temporal behavior of the inter-layer protocol that is composed of events bouncing back and forth within the stack. Each edge in the event chain is assigned a weight, which is the time between the generation of the parent message and the generation of the child. Since the actual durations of these times are not known beforehand, the weights represent relative timing constraints only. The event chains for the group membership protocol were generated by reverse-engineering the source code.

A state machine description of the blocking behavior of the protocol is obtained by computing the critical path between the initial event in which we are interested (crash failure, node join) and the event that finally causes the group to unblock. The *critical path* between two vertices in an event chain is the maximum weight path between them. The edges on this critical path represent possible states in the state machine model. Since the number of edges in our critical path were large (more than 50), several adjacent edges of the same type were merged to form the states in the state model. The type of an edge (and thus its corresponding state) depends on the reason for the time delay associated with it. These reasons can include processing time within layers (represented by unshaded states in the state machine in Figure 1), time spent in application callbacks (the doubly circled states), and time spent waiting to receive a message from another node (the shaded states).

In order to incorporate an additional external event into a state machine, e.g., a second crash failure during the operation of the protocol, we examined the event chain for the event to be added, and determined, for each state of the original state machine, whether the events in the new event chain would cause additional blocking. If they did, and the reason for additional blocking was represented by a state other than the one currently under consideration, an edge was introduced between the two states.

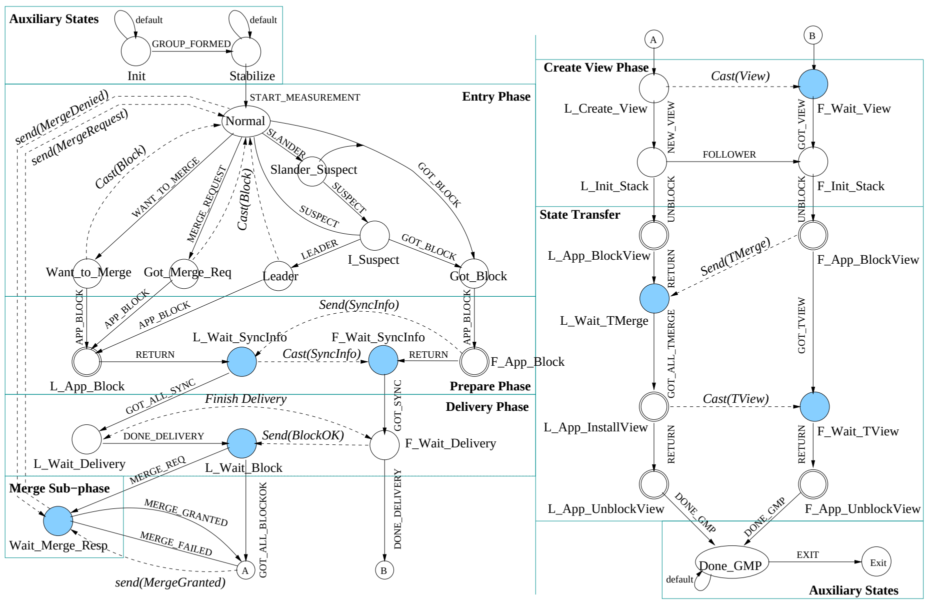

Fig. 1. State Machine for the Group Membership Protocol

3.2 State Machine Specification

The Ensemble group membership protocol can be decomposed into multiple phases. Figure 1 presents the state machine that describes the protocol operation in response to a single fault at a level of abstraction appropriate for this study. In the figure, solid transitions represent the local event notifications at each node that cause state changes. The dashed lines represent actual communication between the Ensemble stacks on different nodes (typically between the leader and followers). States with names prefixed by `L` (`F`) represent states that only the leader (followers) of a group may enter. To model the effects of multiple fault injections, additional transitions beyond those shown in the figure are needed. A description of the state machine for multiple crash failures can be found in [18]. Since node joins are not presented in this paper, the state machine description ignores the states and transitions related to the partition merge protocol.

Auxiliary states are not part of the protocol, but track the execution of the protocol stack until the conditions are right for fault injection and measurement. The `Init` state on the top left of Figure 1 is the initial state, and is held until the formation of a group of the required size. Once the group has formed, the node starts message transmission. However, before fault injection or measurement can be done, any startup transients need to be removed. This is done by causing the state machine to wait in the `Stabilize` state and ignore application notifications (via the `default` event) until a specified number of initial messages have been exchanged. Once this is done, the state machine transitions into the `Normal` state, and fault injection and measurement can be done from then on. After the group membership protocol has completed, the state machine transitions into the `Done_GMP` state shown in the bottom right-hand side of Figure 1. Any further application notifications are ignored by the state machine. In this manner, we ensure that for every run, exactly one instance of the group membership protocol is evaluated.

The *Entry Phase* is when Ensemble initiates the membership protocol due to a triggering event. The state machine models this for crash failures and partition merges. Crash failures are detected by a stack through either a timeout (suspect) mechanism or a slander message from another node. When a crash is detected, the state machine eventually transitions into the `I_Suspect` state, in which the node determines if it is the leader of the group. If it is, the state machine transitions to the `Leader` state, in which the node sends a *Block* message to the other nodes in the group. When a follower node receives a block message, it transitions to the `Got_Block` state regardless of its current state. At that point, all the nodes in the system execute the *Block* application callback (represented by the `L(F)_App_Block` states), informing the application of an impending group block. When this callback returns, no new messages can be sent until the completion of the group membership change. The time spent in an application callback is application-dependent. Our application does nothing in the callbacks, and hence represents the best-case blocking time.

The *Prepare Phase* begins when the application returns from the *Block* callback. It consists of an agreement protocol in which the nodes in the group agree upon a *consistent* set of messages that must be delivered in the current view to preserve virtual synchrony. The `L_Wait_SyncInfo` state on the leader represents the time spent by the leader waiting for each follower's version of the virtual synchrony information (sent when the follower returns from the *Block* callback). The `F_Wait_SyncInfo` state represents the time each follower then waits for the leader to send out a consistent version of the virtual synchrony information in a *SyncInfo* message. The dissemination of virtual synchrony information to the group marks the end of the prepare phase.

In the *Delivery Phase*, nodes in the system deliver outstanding messages from the set agreed upon in the Prepare Phase. The `L_Wait_Delivery` and `F_Wait_Delivery` states represent the time spent by the leader and followers, respectively, in delivering messages (represented by the *Finish Delivery* message exchange). Once a follower finishes its message delivery, it sends a *BlockOK* message to the leader. The `L_Wait_Block` state of the state machine represents the time spent by the leader waiting to receive *BlockOK* messages after it has finished its own message delivery. At the end of `L_Wait_Block`, no messages in transit exist anywhere in the system, and the group is said to be *totally blocked.*

The *Create View Phase* begins when the leader enters the `L_Create_View` state of the state machine. This state represents the time spent by the leader in creating a new view, which it broadcasts to the group. The time in the `F_Wait_View` state represents the time spent by each follower waiting for the new view. When a node has a copy of the new view, it creates a new stack. The

`L(F)_Init_Stack` states of the state machine represent the time required for this operation.

The *State Transfer* phase is present in some configurations of Ensemble, and is initiated after the creation of a new stack. It implements a protocol that enables applications to agree on a common state for the new view. Although state transfer is not part of the core group membership protocol, the group is blocked during its execution, and hence we model this protocol in our blocking behavior model. In this protocol, the stack at each node requests application state via the *BlockView* application callback (the `L(F)_App_BlockView` states), and the followers send their versions of this state to the leader via a *TMerge* message. The leader waits for *TMerge* messages (in the `L_Wait_TMerge` state), and on receiving them asks the application to compose them into a single state via the *InstallView* callback (the `L_App_InstallView` state). The single state is sent to all the followers (which wait for it in state `F_Wait_TView`) via a *TView* message. All nodes install this common state via the *Unblock* callback (represented by the `L(F)_App_UnblockView` states), and henceforth the group is unblocked.

4 Fault Triggers

Our evaluation of the group membership protocol as a fault tolerance mechanism seeks to quantify two properties of the protocol. The first is its performance under varying system conditions, while the second is its robustness to the occurrence of additional faults during its operation. Quantification of the first property can be achieved by injecting a single fault when the system is in the normal state, and parameterizing the experiments by workload and message size. Quantification of the second property requires injection of an additional fault into the system while it is recovering from the first fault. This second set of experiments is parameterized by the global state of the system at which the second "correlated" fault is to be injected. The performance metric in both cases is the blocking behavior of the group, which is computed by measuring overall group blocking time and per-state holding times in blocking states. In the trigger specifications, nodes are identified by their names, which are $\mathrm{NodeFail}, \mathrm{Node}_1, \mathrm{Node}_2, \ldots, \mathrm{Node}_n$. All nodes execute the same application and are tracked by the same state machine (the one shown in Figure 1 augmented with transitions to account for multiple crashes). NodeFail is the node used to inject the first crash failure.

For the single-fault experiments, a single crash failure is injected into the node NodeFail when a group of the proper size has formed and each state machine has transitioned to the `Normal` state of the state machine. The fault method for crash failures simulates a crash by generating a segmentation failure exception. The fault trigger for single crash failures is defined on node NodeFail simply as:

$$(\mathrm{Node}_1 : \mathrm{Normal}) \wedge \ldots \wedge (\mathrm{Node}_n : \mathrm{Normal}) \wedge (\mathrm{NodeFail} : \mathrm{Normal}) \mapsto \mathrm{CrashFault}$$

For the correlated fault experiments, exactly two faults are injected in every experiment. The first fault is injected when all nodes are in their `Normal` state, just as for the single-fault experiments, and hence uses the trigger shown above.

The second fault is then injected when the system is recovering from the first crash, and is in certain global states. Since faults can be either crash failures or node joins, four combinations of injections are possible. Of these, only Crash-Crash and Merge-Crash injections are valid, because Ensemble disallows group merges during membership changes. We have explored only the Crash-Crash scenario, but the techniques are equally applicable to the Crash-Merge scenario.

Although the trigger for the second crash failure depends on the exact global state in which the fault is to be injected, a recurrent theme is injecting a leader or follower when it is in one of a set S of local states, and no other injections have been done. If the injection is to be done in a leader, S will contain only leader states ($\mathcal{L}$) whose names have the prefix `L_`. Since any node in a system can be a leader, fault triggers must be installed on each node in the system. These triggers must be temporally synchronized; i.e., after a leader has crashed, the same trigger should not be invoked a second time in the new leader elected by the group. This is ensured by checking that when the fault is injected, no nodes except NodeFail have crashed. When injecting a follower, S will contain only follower states ($\mathcal{F}$) that have the prefix `F_`. We define the set function $L(S) : 2^{\mathcal{F}} \mapsto 2^{\mathcal{L}}$, which returns the subset of leader states ($\mathcal{L}$) that the leader of a group could be in when a follower is in one of the S states. When a follower is being injected, fault triggers need to be installed only on two nodes, since at-least one of them must be a follower. The triggers must be spatially synchronized (i.e., only one of the triggers injects the fault in a given experiment, even if both nodes are followers). This is ensured by having one of the two nodes inject a fault only if the other node is in one of the leader states ($L(S)$). Note that neither temporal nor spatial synchronization can be done without a knowledge of the global state. The fault triggers that satisfy the above conditions are the following:

Leader Crash Injections
$\forall i = \{1, \ldots, n\}$, At Node$_i$:
(Node$_i$: state $\in S$) $\wedge \neg$(Node$_1$: CRASH) $\wedge \ldots \wedge \neg$(Node$_n$: CRASH) $\mapsto$ CrashFault
Follower Crash Injections
At Node$_1$: (Node$_1$: state $\in S$) $\mapsto$ CrashFault
At Node$_2$: (Node$_1$: state $\in L(S)$) $\wedge$ (Node$_2$: state $\in S$) $\mapsto$ CrashFault

State Selection Using the general fault triggers described above, we now compute the smallest set of local states that need to be targeted (with a second crash failure) for completeness. A node crash impedes the progress of the protocol whenever the rest of the group waits for a protocol message from the crashed node. If the execution path of a node is split into intervals delimited by the times at which it transmits protocol messages, then it is necessary to target each interval at least once, since the blocking of different outgoing messages may each have a unique impact on the group. Also, each interval is representative of all the states within that interval, since a node can affect others only through outgoing messages. Hence, injecting *only one* state in the set of states (S) that represent each interval is enough to obtain *representative behavior* for all states. However, this does not mean that the response of the system to a crash at any instant is *identical* to the response to a crash elsewhere in the same interval, since other factors in the protocol (such as expiration of timeouts) are sensitive to the exact crash time. The dotted edges in the state machine in Figure 1 represent the protocol messages, and the states between successive outgoing messages represent the intervals. For example, the *Send(BlockOk)* and *Send(TMerge)* protocol messages delineate an interval that contains the `F_Wait_View`, `F_Init_Stack`, and `F_App_BlockView` states. The intervals used for the correlated fault injection experiments are shown below in terms of S and $L(S)$ (format is $S \mapsto L(S)$):

{Leader}, {L_Wait_SyncInfo}, {L_Create_View}, {L_Wait_TMerge, L_App_InstallView} $\mapsto \phi$
{Got_Block} $\mapsto$ {Leader, L_App_Block, L_Wait_SyncInfo}
{F_Wait_Delivery} $\mapsto$ {L_Wait_Delivery, L_Wait_Block}
{F_Init_Stack, F_App_BlockView} $\mapsto$ {L_Init_Stack, L_App_BlockView, L_Wait_TMerge}

5 Measures

For this study, group blocking time and contributions of individual local states to this time are the measures of interest. We define blocking times for two versions of the membership protocol. The version that includes the application state transfer function (Section 3.2) is called the *Base Protocol*, and the version without it is called the *Core Protocol*. The blocking times for the Core Protocol can be obtained by treating the state transfer phase as non-blocking and as a part of the application. The last states in which applications are allowed to transmit are the `L(F)_App_Block` states of the Entry Phase. The group unblocks in the `L(F)_App_BlockView` states for the core protocol and in the `L(F)_App_UnblockView` states for the base protocol. Hence, the sets B_{base} and B_{core} of blocking states in the base protocol and the core protocol, respectively, are defined as follows:

B_{core} = {L_Wait_SyncInfo, L_Wait_Delivery, L_Wait_Block, Wait_Merge_Resp, L_Create_View, L_Init_Stack, F_Wait_SyncInfo, F_Wait_Delivery, F_Wait_View, F_Init_Stack}
B_{base} = B_{core} $\cup$ {L_App_BlockView, L_Wait_TMerge, L_App_InstallView, F_App_BlockView, F_Wait_TView}

For a given set of blocking states B, we define the *Total Group Blocking Interval* as that interval of time during which all the nodes in the group are blocked. During that period, no messages can originate anywhere in the system. The length of this interval is $t_{total}(B)$, where $B \in \{B_{core}, B_{base}\}$. The *Partial Group Blocking Interval* is defined as that interval of time during which a non-empty subset of the nodes are blocked. The length of this interval is $t_{partial}(B)$. The intervals for the core protocol start at the same time as the corresponding intervals for the base protocol. However, a total interval is properly contained within its corresponding partial interval, because nodes enter and leave the protocol at different times due to variations in message delay, speed, load, and protocol characteristics. The difference between the start times of the partial and total interval of the base (or core) protocol is called the *preblock stagger*, and represents the maximum blocking time spent by a group member waiting for other nodes to start the protocol. It is upper-bounded by $t_{partial}(B_{core}) - t_{total}(B_{core})$.

To compute the blocking intervals, a predicate is defined on the global state of the system such that the predicate is true when the system is blocked. Application of this predicate to the global timeline for an experiment yields a predicate timeline that is a Boolean-valued function of time. The interval for which this predicate timeline is true gives the blocking interval, and the time for which it is true gives the blocking time. The predicates for the partial group blocking interval $Block_{partial}(B, t)$ and the total group blocking interval $Block_{total}(B, t)$ are defined as follows:

$\text{Block}_{partial}(B, t) = \exists x \in \{\text{Node}_1, \ldots, \text{Node}_n\}\text{s.t.} \bigvee_{s \in B}(x : \text{state}(t) = s)$
$\text{Block}_{total}(B, t) = \forall x \in \{\text{Node}_1, \ldots, \text{Node}_n\}, \bigvee_{s \in B}(x : \text{state}(t) = s)$

The contributions of individual states to the blocking time are expressed as the ratio of the average individual-state blocking time to the overall blocking time. The group blocking time used is dependent on the state being considered. Computing this measure for a state s requires multiple predicates: one for each node in the system. Each predicate is defined as $\text{Block}_{partial|total}(B, t) \wedge (Node_i : state(t) = s)$. The measure is then the average ratio of the duration of time each predicate is true to the requisite group blocking time.

In the experiments of Section 6, it was observed that the crash detection timeout interval dominated the results. Using the Loki measures language, we were able to analytically factor this timeout interval out of a blocking time. We did so by first computing the time interval between the first `SUSPECT` event after the second crash failure injection, and the event just prior to that `SUSPECT` event. This is the interval when the protocol does nothing while waiting for a timeout, and its overlap with the group blocking time interval gives the contribution of the timeout to the group blocking time. That overlap is subtracted from the group blocking time to remove the effects of the timeout.

6 Experimental Results

This section presents the experimental results of our fault injection for both single and correlated crash failure injections. The node join results are not presented here. The testbed used consisted of 12 identical hosts (1GHz Pentium-III with 256MB RAM) running Redhat Linux 6.2. We used a bytecode version of Ensemble 1.20 compiled with the OCaml 3.02 compiler. Each application node was started on a separate host. The results for single fault injections are divided into studies with varying workload and varying group sizes. After we conducted experiments, we observed that message blocking depended on whether or not the first crash was a leader. For leader crashes, the group cannot deliver any messages during the crash detection timeout period. For follower crashes, the issue is more subtle. Hence, we have filtered out experiments in which the first crash was a leader, and present measures for follower crashes only.

Single Crash Failure with Varying Workload During these experiments, a crash failure was injected in a group of a fixed initial size of five nodes, and the workload rate parameter λ (Section 2) was varied from 1 message/sec to 10,000 msgs/sec. Figure 2(a) shows the resulting total and partial blocking times for the base protocol. Corresponding results for a node join in a group of four nodes are also shown for the sake of comparison. It is seen that for low message rates, the blocking time for crash failures is five orders of magnitude higher than that for

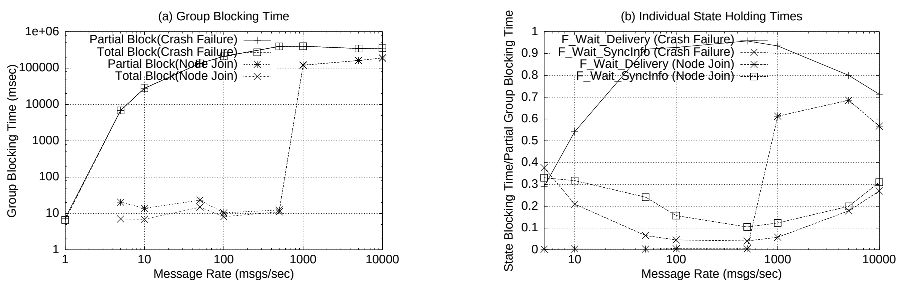

Fig. 2. Blocking Times for a Standard Stack with Varying Workload

node joins, even though the protocols are not very different. Node join blocking time also suddenly increases for high message rates; after an investigation that we will not detail here, we found that that was due to the lack of a negative acknowledgment mechanism in the merge protocol. Addition of such a mechanism would be a minor modification to the protocol.

To analyze the extraordinarily large blocking times for crash failures, we plotted the ratios of the time a node spent in each blocking state to the partial group blocking time. Figure 2(b) shows these ratios for the `F_Wait_Delivery` and `F_Wait_SyncInfo` states for crash failures. The ratios for node joins are also shown for the sake of comparison. It is seen that for message rates up to 500 msgs/sec, the `F_Wait_Delivery` state forms a significant fraction of the time spent blocking for crash failures, but not for node joins. This reflects a problem with message delivery in the presence of crash failures. An examination of the global timelines revealed that messages were being blocked and buffered by the flow control layer for most of the `Normal` state because of interactions with the failure detection mechanism. The Ensemble flow control layer *MFlow* employs a credit-based scheme for broadcast traffic that bounds the number of undelivered messages. While that does bound buffer sizes in the reliable delivery layer, nodes quickly run out of credit after a crash due to the lack of message acknowledgments from the crashed node. Subsequent transmissions are buffered by the flow control layer for almost the entire duration of the failure detection timeout period, causing a large backlog of messages that must be delivered in the delivery phase and resulting in large blocking times.

One solution is for the flow control layer to prevent the application from transmitting when it runs out of credits. After some investigation, we found that Ensemble does provide a new application interface (different from the one we used) that does that. However, this solution is not ideal from an availability standpoint, because it causes the group to be blocked for almost the entire crash detection timeout interval. Alternatively, the credits available to each node can be increased such that there are enough to sustain message delivery at the expected workload throughout the crash detection timeout period. However, doing so couples flow control settings with the timeout interval and expected workload; that is undesirable, since the flow control settings should depend only

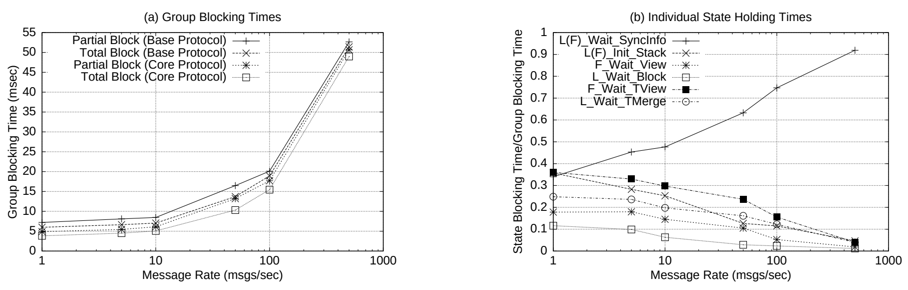

Fig. 3. Single Crash-failure Injection With Varying Workload (Without Flow Control)

on network and host capacities. A flow control scheme that takes into account the timeout interval when setting a bound on the maximum number of undelivered messages, but limits the instantaneous rate of traffic according to network and host characteristics, seems more suitable for high-availability applications.

To ensure that further results were not dominated by flow control layer effects, the remaining experiments were conducted without a flow control layer in the stack. If message traffic is kept low enough that the flow control layer would not have blocked any messages during failure-free operation, this configuration is equivalent to having a flow control scheme that does not block messages during a crash detection timeout period, and buffers unacknowledged messages. Hence, we restricted workload message rates to a maximum of 500 messages per second per node. Figure 3(a) shows the total group blocking times for the core and base protocols for follower crash failures under a varying workload. The difference between the total blocking times for the base and core protocols is the time taken for application state transfer. It is seen that the blocking times for crash failures increase linearly with increasing message rate and that all four curves are fairly close to each other. This indicates that the preblock stagger time (Section 5) is low. Figure 3(b) shows the contributions of individual states to the overall blocking time for those individual states that contribute significantly. The holding times for the state transfer protocol states (`F_Wait_TView` and `L_Wait_TMerge`) are expressed as fractions of the base protocol partial block time, while the other times are expressed as fractions of the core protocol partial block time. It is seen that at low rates, a node spends around 35% of its blocking time in the state transfer protocol. The `L(F)_Wait_SyncInfo` states are the largest consumer of the remaining time (the core protocol blocking time). Those states are also responsible for most of the increase in overall blocking time with increasing workload. At 500 msgs/sec, the state transfer protocol consumes only 5% of a node's blocking time, while the `L(F)_Wait_Sync_Info` states consume over 85%.

An investigation of this phenomenon revealed that most of the time spent in the `L(F)_Wait_SyncInfo` states was because virtual synchrony information sent by followers at the beginning of the `F_Wait_SyncInfo` state wasn't reaching the leader quickly enough. That, in turn, was due to processing going on

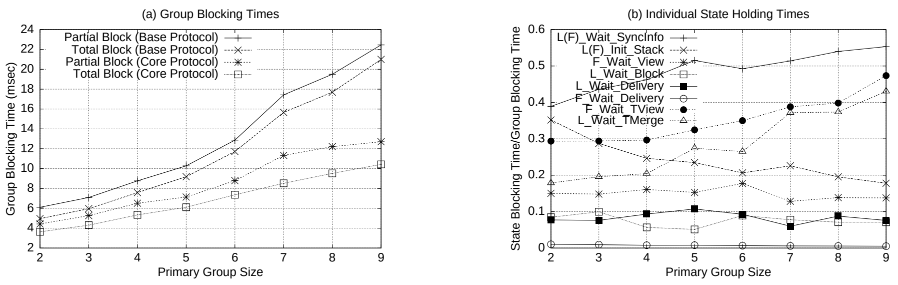

Fig. 4. Single Crash-failure Injection With Varying Group Size (Without Flow Control)

within one of the layers in the leader's stack; that processing was preventing the single-threaded leader stack from reading the network for new messages. The troublesome processing turned out to be the garbage collection of unacknowledged messages sent to the crashed node. This garbage collection was initiated when the crashed node was declared as Failed at the beginning of the `L_Wait_SyncInfo` state, and was a result of removing the flow control layer and thus allowing message delivery during the crash timeout period. However, note that the penalty of additional garbage collection is much smaller than the penalty of restricting message flow during the timeout interval, as demonstrated in the earlier experiments. This phenomenon points to a trade-off. When a crash failure occurs, a flow control layer may either block messages from being transmitted to any member in the group until the failure is resolved, or allow transmission to live members of the group but face the issue of maintaining and disposing of potentially large unacknowledged message buffers. The former approach guarantees bounded buffers and garbage collection times, but sacrifices availability of the entire group during the crash detection timeout period. Additionally, if the messages that have been blocked need to be buffered by the application anyway, that approach buys nothing. The latter approach increases the availability of the system, but can place on the communication stack the burden of maintaining and disposing of buffers that may be as large as the product of the maximum throughput of the system and the crash detection timeout interval.

Single Crash Failure with Varying Group Size During these experiments, the message rate was fixed at a rate of 10 msgs/sec. Our intention was to keep it low to minimize the garbage collection effects explained earlier, while still ensuring enough load that we did not evaluate a trivial case. The initial group size was varied from 3 nodes to 10 nodes, and a single crash failure was injected. The resulting group blocking times are shown in Figure 4(a). It is seen that all the blocking times increase linearly with group size. The total blocking times for both the base and core protocols are close to their corresponding partial blocking times indicating that the preblock stagger is low. Hence, nodes are not blocked for too long waiting for other nodes to enter the protocol. Figure 4(b) shows the contributions of the individual states to the blocking times. As before, the contributions of the two state transfer protocol states are expressed as fractions of the base protocol partial blocking time, while the contributions of other states are expressed as fractions of the core protocol blocking time. The increasing ratios for the L_Wait_TMerge and F_Wait_TView states demonstrate that the state transfer protocol is more sensitive to increasing group size than the core protocol is; this is probably due to the fact that the cost of totally ordered message delivery (required in the state transfer protocol) is higher than the cost of unordered message delivery (which is used in the core protocol). The time the leader spends collecting state information from the followers (L_Wait_TMerge) increases more rapidly than the time followers wait to receive a reply from the leader (F_Wait_TView). The reason is that an increasing number of nodes implies increased synchronization costs, whereas the time required to compose a global state remains more or less constant.

Of the remaining part of the blocking time, the percentage contribution of the L(F)_Init_Stack states reduces with increasing group size, reflecting a near constant stack initialization time. The percentage contributions of all the other states, except the L(F)_Wait_SyncInfo states, remain fairly constant with group size, indicating that they are all equally sensitive to group size. The percentage contributions of L(F)_Wait_SyncInfo states, however, show a modest increase with increasing group size. The reason is that as group size increases, the total message traffic in the system increases, because of the constant workload at each node; hence, the increase in the L(F)_Wait_SyncInfo state holding time is due to the effects of garbage collection (as described before). It could be argued that this garbage collection time is an artifact of the flow control scheme (or the lack thereof), and hence should not be considered as part of the blocking time. However, the alternative would have been to conduct experiments with no workload, which, in addition to not being realistic, would also not exercise the virtual synchrony layers at all. If desired, the effects of the garbage collection time can be analytically removed using the same technique (see Section 5) that is used for removing the effects of timeout intervals in the correlated fault experiments.

Correlated Fault Experiments The correlated fault experiments evaluated the result of injecting two crash failures into the protocol, with the second one injected in the states described in Section 4. The initial group size was fixed at 5 nodes, and the message rate was set to 10 msgs/sec. The crash detection timeout was set to 5 seconds. Figure 5(a) presents the resulting group blocking times. The labels on the X axis denote the state in which the second fault was injected. It can be seen that blocking times were dominated by the crash detection timeout, with three exceptions. The blocking times for the core protocol were low when faults were injected into the F_App_BlockView and L_App_InstallView states because the definition of the core protocol does not include the state transfer protocol that these states are a part of. Using the same argument, the injection of a crash in the Leader state to prevent the leader from sending a *Block* message should have left the blocking times unchanged from the single-fault experiments. The only result should have been a doubling of the crash detection timeout period. However, this is not what happened. To investigate this phenomenon, we

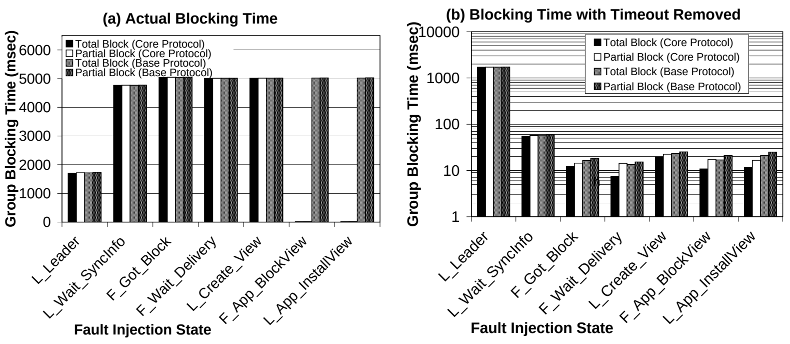

Fig. 5. Correlated Crash-failure Injections (Without Flow Control)

removed the effects of the timeout as described in Section 5, thus representing an idealized scenario of instant crash detection. The resulting group blocking times are shown in Figure 5(b). Comparing these times to the group blocking times for a single crash with a workload of 10 msgs/sec (Figure 3(a)), we see that a correlated fault injected in most states causes only a modest (10% to 100%) increase in blocking time over a single fault injection. The two exceptions are the `Leader` state and the `L_Wait_SyncInfo` state, for which blocking times are unusually high. An examination of the global timelines revealed that the reason is the sequencer-based total ordering protocol. The protocol works by having followers unicast broadcast messages to the leader, which then sequences and broadcasts these messages to the group. When the leader crashes before the group is blocked, any subsequent broadcast messages sent to the leader are not delivered and are buffered by their senders. When the members learn of the leader failure, they broadcast the buffered messages to the rest of the group. This burst transmission of accumulated messages from all members of the group causes the membership protocol to take longer than usual to complete. This phenomenon is the cause of the high blocking times for correlated injections in the `Leader` state and the `L_Wait_SyncInfo` state. This problem does not arise when the leader crashes after the group has blocked, because new message transmissions cannot be generated in a blocked group. However, it is easy to solve the problem by explicitly notifying the application to stop sending broadcast messages when the unicast flow control layer has run out of credits. Since the new Ensemble application interface has support for explicit flow control notification, this modification should be trivial to implement.

7 Conclusions

This paper has presented the use of a novel fault injection methodology based on global state to experimentally evaluate the blocking characteristics of the group membership protocol of the Ensemble GCS. Through this case study, we have shown how the notion of global state can be made precise through the construction of local state machines, and how fault triggers defined on this global state can be used to elegantly specify and coordinate the injection of faults into a distributed system. We have also shown how global measures on the system can be specified using a global-state timeline, and shown the importance of these measures in bringing out subtle timing characteristics of the distributed application. Thus, global-state-based fault injection is shown to have the potential to be an invaluable technique for dependability and performance assessment as well as testing of distributed systems.

The resulting evaluation has quantified group membership blocking characteristics for the Ensemble system. This data can be useful for designers of applications built on top of the Ensemble system to evaluate the dependability of their applications. Our work has also demonstrated that the choice of a flow control scheme in a GCS can be a very important factor in determining the availability of the entire system. This, along with other insights presented in this paper, can be useful to the designers of group communication systems. In conclusion, our work highlights the role of experimental evaluation as a technique that is complementary to formal verification in the design of highly dependable distributed systems.

Acknowledgments This material is based on work supported by the National Science Foundation under Grant No. 0086096. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. The authors would also like to thank Ryan Lefever for providing technical feedback and Jenny Applequist for helping improve the readability of this work.

References

1. J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martin, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Trans. on Software Eng.*, 16(2):166–182, Feb. 1990.
2. J. Arlat, M. Aguera, Y. Crouzet, J. Fabre, E. Martins, and D. Powell. Experimental evaluation of the fault tolerance of an atomic multicast protocol. *IEEE Trans. on Reliability*, 39:455–467, Oct. 1990.
3. D. Bhatt, R. Jha, T. Steeves, R. Bhatt, and D. Wills. SPI: An instrumentation development environment for parallel/distributed systems. In *Proc. of the 9th Int'l Parallel Processing Symp.*, pages 494–501, 1995.
4. K. P. Birman. *Building Secure and Reliable Network Applications.* Manning, 1996.
5. R. Chandra, M. Cukier, R. M. Lefever, and W. H. Sanders. Dynamic node management and measure estimation in a state-driven fault injector. In *Proc. of the 19th IEEE Symp. on Reliable Distrib. Systems*, pages 248–257, Oct. 2000.
6. R. Chandra, R. M. Lefever, M. Cukier, and W. H. Sanders. Loki: A state-driven fault injector for distributed systems. In *Proc. of the Int'l Conference on Dependable Systems and Networks (DSN-2000)*, pages 237–242, Jun. 2000.
7. M. Cukier, R. Chandra, D. Henke, J. Pistole, and W. H. Sanders. Fault injection based on the partial global state of a distributed system. In *Proc. of the 18th IEEE Symp. on Reliable Distrib. Systems*, pages 168–177, Oct. 1999.
8. M. Cukier, D. Powell, and J. Arlat. Coverage estimation methods for stratified fault-injection. *IEEE Trans. on Computers*, 48(7):707–723, Jul. 1999.

9. S. Dawson, F. Jahanian, T. Mitton, and T. L. Tung. Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *Proc. of the 26th Int'l Symp. on Fault-Tolerant Computing (FTCS-26)*, pages 404–414, Jun. 1996.
10. S. Dawson and F. Jahanian. Probing and fault injection of dependable distributed protocols. *The Computer Journal*, 38(4):286–300, 1995.
11. K. Echtle and M. Leu. The EFA fault injector for fault-tolerant distributed system testing. In *Proc. of the IEEE Workshop on Fault-Tolerant Parallel and Distrib. Systems*, pages 28–35, 1992.
12. Fekete, Lynch, and Shvartsman. Specifying and using a partitionable group communication service. In *PODC: 16th ACM SIGACT-SIGOPS Symp. on Principles of Distrib. Computing*, 1997.
13. S. Han, K. G. Shin, and H. A. Rosenberg. DOCTOR: An integrated software fault injection environment for distributed real-time systems. In *Proc. of the Int'l Computer Perf. and Dependability Symp.*, pages 204–213, 1995.
14. M. Hayden. *Ensemble Reference Manual.* Cornell Univ., 1997.
15. M. Hayden. *Ensemble Tutorial.* Cornell Univ., 1997.
16. M. Hayden. *The Ensemble System.* PhD thesis, Comp. Sci., Cornell Univ., 1997.
17. J. Hickey, N. Lynch, and R. van Renesse. Specifications and proofs for Ensemble layers. In *Proc. of the Fifth Int'l Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, Mar. 1999.
18. K. R. Joshi. *A Global-State Based Approach to Evaluation of Unavailability caused by Group Membership.* M.S. thesis, Univ. of Illinois, 2002. To be published.
19. C. Kreitz, M. Hayden, and J. Hickey. A proof environment for the development of group communication systems. In *Automated Deduction - 15th Int'l Conference on Automated Deduction. Proc.*, pages 317–332, Jul. 1998.
20. F. Lange, R. Kroeger, and M. Gergeleit. JEWEL: Design and implementation of a distributed measurement system. *IEEE Trans. on Parallel and Distrib. Systems*, 3:657–671, Nov. 1992.
21. S. Mishra and Wu. Lei. An evaluation of flow control in group communication. *IEEE/ACM Trans. on Networking*, 6(5):571–587, Oct. 1998.
22. G. Neiger. A new look at membership services. In *Proc. of the Fifteenth ACM Symp. on Principles of Distrib. Computing*, pages 331–340. May 1996.
23. D. Scott, N. Speirs, Z. Kalbarczyk, S. Bagchi, J. Xu, and R. K. Iyer. Comparing fail-silence provided by process duplication versus internal error detection for a DHCP server. In *Proc. of Int'l Parallel and Distrib. Processing Symp.*, Apr. 2001.
24. D. T. Scott, B. Floering, D. Burke, Z. Kalbarczyk, and R. K. Iyer. NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proc. of IEEE Int'l Computer Perf. and Dependability Symp.*, pages 91–100, 2000.
25. D. T. Scott, M. C. Hsueh, G. Ries, and R. K. Iyer. Dependability analysis of a commercial high-speed network. In *Symp. on Fault-Tolerant Computing*, pages 248–257, 1997.
26. P. W. Uminski, M. R. Matuszek, and H. Krawczyk. Experimental evaluation of PVM group communication. In *Recent Advances in PVM and MPI. 4th European PVM/MPI Users' Group Meeting. Proc.*, pages 57–63, 1997.
27. R. van Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. A. Karr. Building adaptive systems using Ensemble. *Software - Practice and Experience*, 28(9):963–979, 1998.
28. R. Vitenberg, I. Keidar, G. Chockler, and D. Dolev. Group communication specifications: A comprehensive study. Tech. report CS99-31, Comp. Sci. Inst., The Hebrew Univ. of Jerusalem and MIT Tech. Report MIT-LCS-TR-790, Sep. 1999.