Distrib. Syst. Engng 3 (1996) 39–52. Printed in the UK

Numerical performability evaluation of a group multicast protocol*

Luai M Malhis†§, William H Sanders†‖ and Richard D Schlichting‡¶

† Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, IL 61801, USA
and
‡ Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA

Abstract. Multicast protocols that provide message ordering and delivery guarantees are becoming increasingly important in distributed system design. However, despite the large number of such protocols, little analytical work has been done concerning their performance, especially in the presence of message loss. This paper illustrates a method for determining the performability of group multicast protocols using stochastic activity networks, a stochastic extension to Petri nets, and reduced base model construction. In particular, we study the performability of one such protocol, called Psync, under a wide variety of workload and message loss probabilities. The specific focus is on measuring two quantities, the stabilization time—that is, the time required for messages to arrive at all hosts—and channel utilization. The analysis shows that Psync works well when message transmissions are frequent, but it exhibits extremely long message stabilization times when transmissions are infrequent and message losses occur. We use this information to suggest a modification to Psync that greatly reduces stabilization time in this situation. The results provide useful insights into the behaviour of Psync, as well as serving as a guide for evaluating the performability of other group multicast protocols.

1. Introduction

Group multicast protocols are becoming increasingly important in distributed system design for a number of reasons. One is that they often provide strong guarantees that can serve as an important foundation for building highly dependable distributed applications. For example, such protocols often preserve a consistent ordering among messages, so that each process in the multicast group is guaranteed to receive messages in the same order. Another common property is atomicity, which guarantees that a given message is delivered either to all processes or no processes. These properties make group multicast a useful abstraction for implementing, among other things, the *state machine approach* to building software that can continue executing despite failures in the underlying computing platform [1]. In this approach, a service is implemented as a state machine that is replicated on multiple independent hosts. Service requests are then disseminated to the replicas using group multicast. The ordering and atomicity properties implemented by the protocol mean that requests are processed in the same order by all replicas, thereby ensuring that states remain consistent despite failures.

* This work was supported in part by the Office of Naval Research under grants N00014-91-J-1015 and N00014-94-1-0015.
§ E-mail address: malhis@crhc.uiuc.edu
‖ E-mail address: whs@crhc.uiuc.edu
¶ E-mail address: rick@cs.arizona.edu

Many group multicast protocols that exhibit these properties, or variants thereof, have been developed and used in realistic settings. The Isis toolkit [2] includes ABCAST and CBCAST, which are group multicast primitives that implement different variants of message ordering. Researchers associated with the European Delta-4 project [3] constructed several versions of group multicast, including one called xAMP that provides real-time guarantees [4]. The Mars system, which is based on a custom hardware platform with redundant hardware elements, also includes a real-time group multicast [5]. Other systems with group multicast include Amoeba [6], Consul [7], Totem [8], and Transis [9].

Despite the large number of such protocols, however, little analytical work has been done concerning their performance, especially in the presence of message loss. While the guarantees made ensure that processes in a group receive the same sequence of messages, they often say nothing concerning the timeliness of those deliveries or the network bandwidth required to achieve delivery. Studies of the performance of such protocols have often been limited to their fault-free (non-message loss) behaviour or, if message losses are considered, to experimental results for a small number of test scenarios. While these results provide useful information, they are, by their nature, very time consuming to obtain and limited in scope to the range of test scenarios considered.

0967-1846/96/010039+14$19.50

Modelling is an attractive option for predicting the performability [10] of group multicast protocols under a wide variety of workload and fault scenarios. It has the virtue of abstracting away details that are unimportant with respect to measures of interest, while retaining important information about system behaviour. Simulation models are useful in this context, but they fail when the measures of interest are very small or when important events in the model are rare (such as message losses). Importance sampling simulation [11] has the potential of dealing with models with rare events but has been applied primarily to relatively simple systems to date. Analytic models do not suffer from these difficulties but suffer from rapid state-space growth, leading to difficulties in both construction and solution.

Stochastic activity networks (SANs) [12] and reduced base model construction methods [13] avoid, to some extent, both of these problems with analytic modelling. First, SANs allow the model to be constructed at the network rather than state level, and they permit specification of the behaviour of complex systems, whose behaviour would be extremely difficult, if not impossible, to specify at the state level. Second, reduced base model construction methods detect symmetries in a SAN model. To use this approach, a complete (or 'composed') model is built from one or more SAN submodels using 'replicate' and 'join' operations. Formally, the resulting model is known as a *composed SAN-based reward model* (SBRM). The *replicate* operation duplicates a SAN and associated *reward* structure a certain number of times, holding some subset of its places, called its 'distinguished places' in [13], common to all resulting submodels. The combination of several different submodels is accomplished using the *join* operation. Informally, the effect of this operation is to produce a composed model which is a combination of the individual submodels. This approach permits the construction of a stochastic process representation with far fewer states than traditional stochastic Petri net state generation methods, when such symmetries exist.

These features suggest that SANs and reduced base model construction methods can be profitably used to determine the performability of group multicast protocols. We illustrate this by studying the performability of Psync [14], the group multicast protocol found in the Consul system. We represent message, retransmission request, and retransmission losses in the model, and we faithfully represent the behaviour of the protocol when these events occur. The expected message stabilizing time—that is, the time until all processes in the group have received a multicast—and fraction of time various messages are on the communication channel are determined for a wide variety of workload and message loss probabilities. The analysis shows that Psync works well when message transmissions are frequent, but it exhibits extremely long message stabilizing times when transmissions are infrequent and message losses occur. We then use this information to suggest a modification to Psync that greatly reduces stabilizing time when message transmissions are infrequent.

The results are important for two reasons. First, they provide useful information concerning the performability of Psync. While the general relationship between message transmission rate and stabilizing time is perhaps obvious from the mechanism's design, the precise nature of the trend and magnitude in stabilizing time variation only became clear during the modelling process. Furthermore, the results obtained suggested a modification to Psync to improve message stabilizing time, which was then evaluated to show its usefulness. This modification greatly improved message stabilizing time at low new-message transmission rates. The good performance of the protocol at high new-message transmission rates was not affected by the proposed change. Second, the results illustrate the usefulness and practicality of stochastic activity networks and reduced base model construction in predicting the performability of realistic applications. The stochastic processes that were constructed automatically from the SAN representation ranged from approximately 10K to 120K states (much smaller than necessary if reduced base model construction was not employed). Note that the generated stochastic processes are not Markov, since we consider models that have a mix of exponential and deterministic delays, and are solved using recently developed solution methods for deterministic and stochastic Petri nets.

The remainder of the paper is organized as follows. First, in section 2, we provide a brief overview of the Psync protocol, describe the workload, fault environment, and protocol assumptions that were made in constructing the model, and present a high-level description of the model itself. The third section then describes the translation of the model, described informally in section 2, into a composed stochastic activity network representation. Section 4 describes the performability measures that can be determined using the model, and section 5 gives the results obtained by solving the model for a wide range of parameter values. Section 6 suggests a modification to Psync to improve its performance and illustrates the usefulness of the change by modifications to the model. Finally, section 7 offers some conclusions regarding the work.

2. The Psync protocol model

2.1. Overview of Psync

Psync [14] is a group communication protocol that supports multicast message exchange among a collection of processes. Messages are transmitted *atomically* and are presented to receiving processes in a *consistent partial order*. The first property guarantees that messages are delivered either to all processes or to no process, despite communication or processor failures. The second guarantees that each process receives messages in the same (partial) order and that the order is consistent with execution causality; this type of ordering has also been called *causal ordering* [2]. More information on how Psync is used within Consul and its relationship to other protocols in the system can be found in [7] and [15].

To realize these properties, Psync explicitly maintains on each host a copy of a directed acyclic graph called the *context graph*. The nodes in this graph represent the

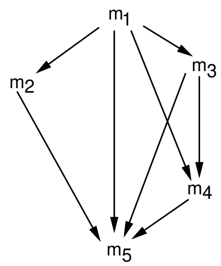

Figure 1. Example of context graph.

multicast messages, while the edges represent the causality relation between the receipt of one message by a process and the subsequent sending of another message. Actual transmission is done over the communications channel using either a broadcast facility or point-to-point message passing. Each message transmitted is identified by a *message id* and the *id* of the sender.

The following sections give a brief description of the context graph, the basic operations for sending and receiving messages, and how Psync operates in the presence of transient network failure. For a more detailed description of Psync, consult [14].

Context graph. Formally, the context graph at a given host defines the $\prec$ (*precedes*) relation on the set of messages that are multicast within the process group. For two messages, m and m', $m \prec m'$ if and only if the process that sent m' had already received (or sent) m prior to sending m'. Figure 1 gives an example of a context graph. In this example, m_1 was the initial multicast message, while both m_2 and m_3 were sent by processes that had received m_1. However, the lack of an edge between m_2 and m_3 implies that their respective senders had not yet received the other message prior to sending theirs. From the point of view of the computation, then, m_2 and m_3 are *concurrent messages*. Similarly, m_4 is concurrent with m_2, but not m_1 or m_3, since it was sent by a process that had received m_1 and m_3, but not m_2. The process sending m_5 received all prior messages before initiating its transmission. The actual graph kept by Psync differs from figure 1 in that redundant edges such as those from m_1 to m_4 and from m_3 to m_5 are not maintained by the implementation.

Since all messages sent using Psync are multicast, the copies of the context graph on all hosts are identical except for transient differences. Psync employs a garbage collection routine that removes from the context graph messages that have been received by all processes. Through this context graph, an application using Psync is able to determine the context in which each message has been sent or received, and which messages have been received by each host. These properties can be exploited, for example, to sequence the messages consistently on all hosts as required by the state machine approach.

Sending and receiving messages. When a process sends a message m, the message is transmitted by Psync to those processors hosting other processes in the group. In addition, m is inserted into the local copy of the graph, with incoming edges from those nodes representing messages already seen by the sending process. These messages are called *m's predecessor messages*. To indicate the appropriate graph location to remote hosts, the message ids for these predecessor messages are included with m when it is sent over the network. When m subsequently arrives at a host, then it is inserted into the copy of the graph on that host based on these included ids. It is possible, however, that one or more of m's predecessor messages may not have arrived. In this case, m is placed temporarily into a *holding queue*. Once the appropriate messages arrive, m is moved from the holding queue to the context graph.

Sending and receiving retransmission requests. The key reason a message may be placed into the holding queue is that a predecessor message can be lost due to transient network failures. To handle this, Psync implements a *retransmission protocol*. Suppose m is a message in the holding queue. When it is placed there, Psync starts a timer. Should this expire without m's predecessors arriving, the missing messages are considered lost. When this occurs, a request to retransmit the missing messages is sent to the host that sent m. That host is guaranteed to have the missing messages in its copy of the graph since their ids were included with m as its predecessor messages. Actually, since it is possible that the predecessors' predecessors are also missing, the retransmission request identifies the subgraph of the context graph that needs to be retransmitted, not just the message(s) known to be missing.

Sending and receiving retransmissions. As discussed in the previous section, retransmission requests identify a subgraph of the context graph to be retransmitted. When a host receives a retransmission request, it responds by resending all messages in the subgraph. When a retransmitted message arrives at a host, the message is ignored if the host has previously received this message. Otherwise, the message is placed in the context graph or the holding queue as discussed previously.

2.2. Model assumptions

The first step in building an accurate model of Psync, and the fault environment considered, is to state assumptions about the protocol itself and the environment in which it will operate. These assumptions are needed to simplify the modelling and, as argued below, do not compromise the basic characteristics of the protocol.

(i) There is a limit, equal to MAX, on the total number of lost messages at any given time. A message is considered lost if it is missing from the context graph of at least one host. A transmitted message will not be lost when the total number of lost messages is MAX. MAX is chosen so that the fraction of time during which the total number of lost messages equals MAX is very small, thus making the model an accurate approximation of the real situation where there is no maximum value.

(ii) The number of processes in the group is three, and there is one process per host. This is a realistic number for many fault-tolerant applications where data or processing activity are often triplicated.

(iii) There are no outstanding messages at a host. If a message arrives at a host and all of its preceding messages are present in the context graph at that host, the message is immediately received by the process residing on that host.

(iv) There can be at most one message on the communication channel at a time. Thus, when a message arrives at a host, that host can determine whether any of its predecessors are lost and issue a retransmission request for the lost message(s).

(v) Retransmissions of lost messages and retransmission requests are given higher priority than new message transmissions, and retransmissions of lost messages are given higher priority than retransmission requests. Though Psync does not assign priorities to message transmission, this assumption is important from a modelling point of view (as will be shown later) and reasonable, given that it would reduce the variance of message stabilizing time and many communication networks allow assignment of priorities to different types of messages.

(vi) Message transmission, retransmission, and NAK transmission times are deterministic and depend on the length of the message sent.

(vii) Because Psync executes independently from application processing, application processing in the model is separated from the execution of the basic operations of Psync. A process executes application code for a period of time and then generates a message to be transmitted. The process does not start application processing again until the message has been sent on the channel. Hosts can receive and request retransmission for messages while processes are doing application processing.

(viii) Since the focus of this model is Psync, rather than the underlying network implementation, a simple scheme to model the MAC layer is devised. If more than one host wants to send a message on the channel, a host is selected uniformly to transmit. All remaining hosts wait until the communication channel again becomes idle to attempt to transmit their message.

(ix) Message loss in the network is modelled probabilistically, with a fixed loss probability assigned to each message transmission. This loss probability is varied to determine its effect on Psync's performability. Though the model permits assignments of different message loss probabilities for different message types, this paper will only consider the case where the same loss probability is assigned to all transmitted messages.

2.3. Model description

Given the preceding description of the protocol and assumptions, we can now describe a model of Psync that accounts for message, retransmission request, and retransmission losses. The model faithfully represents the behaviour of the protocol as described in section 2.1. We define the following terms to facilitate the discussion. The term *last-sender* identifies the process that transmitted the latest new message. The term *NAK-sender* refers to the process that requested retransmission for one or more messages from the last-sender. The term *third-process* refers to the process other than the last-sender and the NAK-sender. The term *recipients* refers to potential receivers of new or retransmitted messages. In the model, we keep track of both the last-sender and the NAK-sender.

Context graph representation. In the protocol, each newly transmitted message is identified by a message id and the id of the sender. Representing ids directly in the state space will generate a model with an intractable state space. Instead, the model keeps track of the number of lost messages and their type. Note that this is purely a modelling trick and does not in any way affect the results obtained. Lost messages are grouped into two types, depending on the number of processes that have lost them. Messages that are lost by one process are called type-1 messages, and messages that are lost by two processes are called type-2 messages. Locally, each process keeps track of the number and type of messages it has lost. Globally, the model keeps track of the number and type of all messages lost. Each process therefore knows the number and type of messages it has lost and the number and type of all lost messages. We need only keep track of lost messages, because these messages are important to the protocol operation and the evaluation of its performance. Once a message and its predecessors have been received by all processes, it can be deleted from the context graph, since it will not need to be retransmitted to another process.

The following variables are used to represent the context graph at each host and the global context graph. These variables replace identifying messages using message ids and the id of the sender. The variable *process-type1-messages* refers to the number of type-1 messages a process has not received. The variable *process-type2-messages* refers to the number of type-2 messages a process has not received. The variable *total-type1-messages* refers to the number of type-1 messages missing by all processes. The variable *total-type2-messages* refers to the number of type-2 messages missing by all processes. The variable *sender-type1-messages* refers to the number of type-1 messages that the last-sender has not received. The variable *sender-type2-messages* refers to the number of type-2 messages that the last-sender has not received. The variable *NAK-sender-type1-messages* refers to the number of type-1 messages the NAK-sender has not received. The variable *NAK-sender-type2-messages* refers to the number of type-2 messages the NAK-sender has not received. All of these variables except process-type1-messages and process-type2-messages are global (the values of global variables are known to all processes). The use of global variables in our model is equivalent to the 'piggy-backing' technique that the Psync protocol uses to pass context graph information between the processes.

Having described how lost messages are identified and how the context graph is represented, we can now discuss how the basic protocol operations are modelled.

Sending and receiving messages. Processes in the group periodically generate new messages to be transmitted on the communication channel. There are two alternate phases that an application process executes, a processing phase and a transmission phase, as per assumption (vii). At the end of the processing phase, a process generates a new message to be sent on the channel. During the transmission phase, processes that are contending for the channel continuously and independently sense the status of the channel. Once the channel becomes idle, one host, selected uniformly (per assumption (viii)), places a new message on the channel. When a new message is transmitted, it is duplicated twice (equal to the number of recipients, two in this model). Each copy either reaches the destination host or is lost (per assumption (ix)). The host that transmitted the message on the channel starts processing immediately after placing the duplicate copies on the channel.

If both copies reach their destination hosts, the recipients may independently request retransmission for previously lost messages, as described in the next section. If one copy reaches the destination host and the other copy is lost, the process that received the message may request retransmission for previously lost message(s). In this case, the process that did not receive the message increments its process-type1-messages variable. If both copies are lost, the recipients increment their process-type2-messages variable. At the end of transmission, if one or two copies are lost, total-type1-messages or total-type2-messages is incremented, respectively.

Sending and receiving retransmission requests. In the protocol, when a new message is transmitted, the message includes the ids of this message's immediate predecessors in the context graph. A process uses this information to determine if it has lost any messages. Since we do not keep track of message ids in the model, the message loss type distribution of the last-sender is made available to the recipients through sender-type1-messages and sender-type2-messages variables. Therefore, when a process receives a message, it may request retransmission for one or more messages from the last-sender by comparing the process's message-loss-type distributions with the total message-loss-type distributions and the last-sender message-loss-type distributions. The algorithm in figure 2 outlines how this is determined.

The algorithm is designed to determine if there is a correlated message loss between the last-sender and the receiver. A correlated message loss occurs when both the last-sender and the receiver are missing the same message(s). Hence, no NAK is sent for these messages. A retransmission request is sent for the remaining messages the receiver is missing.

The algorithm is structured using the `if ... then ... else` construct. The first `if` statement checks to see whether the receiver has not lost any messages or the last-sender has lost all the messages that any host has lost. If either case is true, no retransmission request is sent for any messages because the just-received message is not in the context of a lost message. The second `else if` statement is true if the receiver has lost one or more messages, but the last-sender has not lost any messages. In this case, the just-received message is sent in the context of all messages the receiver has lost. A retransmission request is sent for these messages. The next `else if` statement is true if the last-sender has lost at least one message and the receiver has lost all messages considered lost thus far. In this case, a retransmission request is sent for all lost messages, except the messages the last-sender lost. The remaining statements check to see how many messages of those lost by the receiver are correlated with the messages the last-sender has lost. For example, if the statement `else if sender-type2-messages == 0 or process-type2-messages == 0` is true, all the messages the receiver has lost and the sender has lost are type-1 messages. Therefore, all the messages missing from the receiver's context graph are present at the last-sender's context graph. The just received message was thus sent in the context of all messages the receiver is missing, and a retransmission request should be sent for these messages (`num-retrans = process-lost`).

The algorithm continues to check all possible values of process-type1-messages, process-type2-messages, total-type1-messages, total-type2-messages, sender-type1-messages, and sender-type2-messages variables and to request retransmission for any messages present at the last-sender's context graph and missing from the receiver's context graph. As presented, the algorithm is applicable when MAX ≤ 3 (the situation considered in this paper), but it is straightforward to extend it to larger values of MAX.

Once the number of messages to be retransmitted has been determined, processes that are requesting retransmission contend for the channel to send a NAK to the last-sender. One process (as per assumption (viii)) is selected uniformly to transmit the NAK. If the last-sender receives the NAK, it will begin retransmitting messages as described in the next section. Otherwise, processes that are requesting retransmission contend for the channel again. No new transmission can begin (as per assumption (v)) until all NAKs have been received by the last-sender and all retransmitted messages have been received by the NAK-senders.

Sending and receiving retransmissions. In the protocol, the NAK messages identify, using message ids, a subsection of the context graph to be retransmitted. This identification is not needed in this model because the needed information can be determined from the global variables. The NAK message indicates the number of messages the last-sender must send. In addition, the NAK-sender message loss type distributions are made known to the last-sender through NAK-sender-type1-messages and NAK-sender-type2-messages variables.

When the last-sender receives a retransmission request for one or more messages, it must determine the message loss type (i.e., the number of processes for which

```
let num-retrans be the number of messages for which a retransmission is requested
let process-lost = process-type1-messages + process-type2-messages
let total-lost = total-type1-messages + total-type2-messages
let sender-lost = sender-type1-messages + sender-type2-messages

if process-lost == 0 or sender-lost == total-lost
   num-retrans = 0
else if sender-lost == 0
   num-retrans = process-lost
else if process-lost == total-lost
   num-retrans = total-lost - sender-lost
else if sender-type2-messages == 0 or process-type2-messages == 0
   num-retrans = process-lost
else if total-lost == 3 and sender-lost == 1 and process-lost == 2
   if sender-type2-messages + process-type2-messages == total-type2-messages
      num-retrans = 2
   else
      num-retrans = 1
else if total-lost == 3 sender-lost == 2 and process-lost == 1
   if sender-type2-messages + process-type2-messages == total-type2-messages
      num-retrans = 1
   else
      num-retrans = 0
else if sender-type2-messages == process-type2-messages == total-type2-messages
   and sender-lost == total-type2-messages
      num-retrans = 0
else
      num-retrans = 1
```

Figure 2. Algorithm to determine the number of messages for which retransmission is needed.

```
let type1-lost = total-type1-messages - process-type1-messages
let type2-lost = total-type2-messages - process-type2-messages
let total-lost = type1-loss + type2-loss

if type1-lost == 0 or NAK-sender-type1-messages == 0
   P{type-1 message} = 0
   P{type-2 message} = 1
else if type2-lost == 0 or NAK-sender-type2-messages == 0
   P{type-1 message} = 1
   P{type-2 message} = 0
else
   P{type-1 message} = type1-lost/total-lost
   P{type-2 message} = type2-lost/total-lost
```

Figure 3. Algorithm to determine message loss type of retransmitted messages.

the retransmission is intended) of each message to be retransmitted. If the message is of type 1, it is transmitted to the NAK-sender. If the message is of type 2, it is duplicated and transmitted to the NAK-sender and the third-process. The algorithm shown in figure 3 describes how the message loss type is determined for a retransmitted message. In this algorithm, the last-sender subtracts its local message loss type distribution from the total message loss type distribution to generate a new total message loss type distribution for the recipients. Using this distribution and the distribution of the NAK-sender, a type for the retransmitted message is probabilistically selected.

When the NAK-sender receives retransmission for a message of type 1, it decrements 1 from process-type1-messages. If the message is of type 2, the number of processes that receive this message is made known to both processes. If the message is delivered to both processes, each process decrements its process-type2-messages. If the message is delivered to one process only, the process that received the message decrements its process-type2-messages. The process that did not receive the message decrements its process-type2-messages and increments its process-type1-messages. If neither process receives the message, neither the local nor the global message loss type distributions are updated. At the end of retransmission, total-type1-messages, total-type2-

messages, NAK-sender-type1-messages and NAK-sender-type2-messages are updated to reflect the new total message loss type distributions and the new message loss type distribution for the NAK-sender.

Model summary. The execution of the model can be summarized as follows. Execution begins with each process in the application processing phase. When a process generates a new message, it places the message on the channel if the channel is idle. If the channel is busy, the process contends for the channel and eventually sends its message. When a process receives a message, it checks to see if this message was sent in the context of any message it is missing. If this is the case, the process waits for its turn to send a NAK to the last-sender. When the last-sender receives the NAK, it determines the message loss type for each requested message and transmits this message on the channel. Throughout the communication, messages can be lost. The type and number of lost messages are recorded using local and global variables. These global variables are used to identify the number and type of messages to be retransmitted.

3. Modelling Psync using SANs

Based on the model description above (assuming deterministic times for message transmissions, retransmissions, and NAK transmissions (assumption (vi)) and exponential processing times), a Markov regenerative stochastic process (MRSP) [17] can be constructed for the protocol. Note that the generated stochastic process is *not* Markov and hence cannot be solved using standard Markov solution methods. Rather than building the MRSP directly, which would consist of tens of thousands of states, the model is constructed as a composed stochastic activity network (SAN) [12]. To construct and solve the SAN model, the modelling package *UltraSAN* [20] is used. Once a SAN model is built, *UltraSAN* automatically converts the SAN model to a MRSP process and solves the resulting process for the performance, dependability, or performability variables of interest. The solution method employed is based on recently developed solution methods for deterministic and stochastic Petri nets [18, 19]. While space does not permit a review of SANs and *UltraSAN*, we will try to illustrate their use as the Psync SAN model is described. For more information, consult [12, 13, 16].

In the following, SAN models are built for each of the protocol operations described, and a complete (or 'composed') model of the multiple processes is built using replicate and join operations. Composed models consist of SANs that have been replicated and joined multiple times. The replicate operation reduces the state-space size of the constructed Markov process by detecting symmetries in the model [13]. Replicated and joined SAN submodels can interact with each other through a set of places which are common to multiple submodels. These places are known as *common* or *distinguished* places.

Figure 4 shows the composed model for Psync. The model consists of three SAN submodels: **transmit**, **receive**, and **retransmit**. These three submodels are joined to generate a complete model of the operations a process performs. The joined model is then replicated three times, representing the three processes in the group.

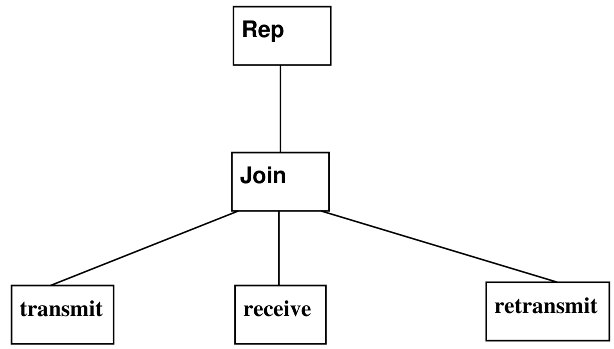

Figure 4. Composed model for Psync.

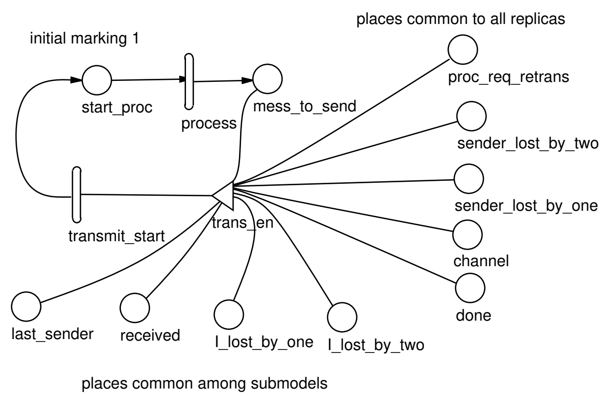

Figure 5. SAN submodel of **transmit**.

Figure 5 shows the SAN representation of the **transmit** submodel. This SAN models the application processing and transmission of new messages on the channel. The SAN consists of the timed activities *process* and *transmit_start* (timed activities, which are drawn as ovals, are used to represent delays in the model). Activity *process* represents the delay (time) in the model that a process spends in the application processing phase. When this activity completes, a new message is generated by adding a token to place *mess_to_send* (represented as a circle). When the activity *transmit_start* completes, a new message is placed on the channel.

The input gate *trans_en* (represented as a triangle with its point connected to the activity) has an enabling predicate and function. The predicate specifies the conditions under which the connected activity, *transmit_start*, is enabled. The function is executed when the activity completes. The predicate for gate *transmit_en* is true if the channel is idle, i.e., *MARK(channel) == 0* (*MARK(x)* is a macro that returns the number of tokens in place *x*), the previous transmission on the channel has completed (*MARK(done) == 0*), no process is requesting retransmission (*MARK(proc_req_retrans) == 0*), and there is a new message to transmit (*MARK(mess_to_send) == 1).* The function for this gate places the message on the channel, makes this process's message-loss-type

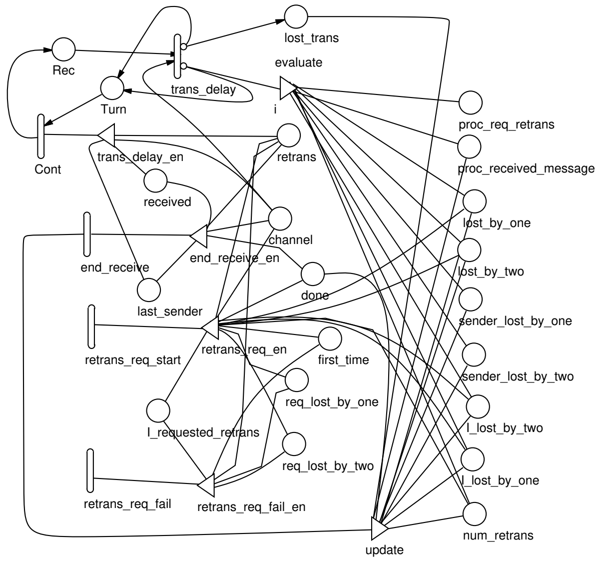

Figure 6. SAN submodel **receive**.

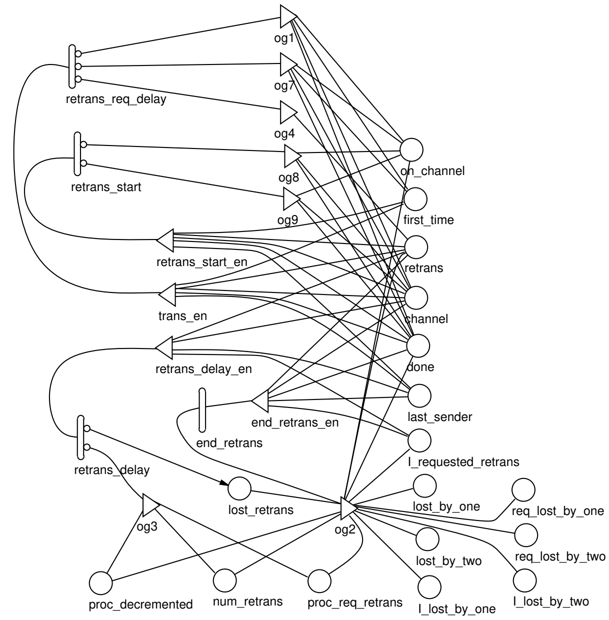

Figure 7. SAN submodel **retransmit**.

distribution known to all processes, and initializes the number of tokens in the places needed to make sure that each recipient receives, at most, one copy of the message. Places that are at the right of figure 5 are common to all processes. Places at the bottom of the figure are common to the submodels but local to a process. Places *start_proc* and *mess_to_send* are local to this submodel.

Figure 6 is the SAN representation of the **receive** submodel. This SAN models the reception of new messages and sending retransmission requests for lost messages. The place *lost_trans* is the only place local to this SAN. The places at the left of the figure are common with other submodels but local to a single process. All other places in the figure are common to all processes. The activity *trans_delay* represents the message transmission delay and has a deterministic time which depends on the length of the message and the speed of the media. There are two *cases* (small circles at the right of the activity) associated with the activity *trans_delay*. One case represents successful message delivery; the other case represents message loss. A single case is chosen probabilistically when an activity completes and the attached gates and arcs are executed. The completion of activity *end_receive* signals the end of a transmission. When this activity completes, processes will start contending for the channel to send a NAK to the last-sender, if a message loss has been detected. The activity *retrans_req_start* represents contention for the channel to send a NAK to the last-sender. The NAK is either delivered to the last-sender or lost. If the NAK is lost, activity *trans_req_fail* is enabled, signalling that processes can begin contending for the channel to send another NAK to the last-sender. The function of output gate *evaluate* (which is represented as a triangle with its back side connected to an activity) executes the algorithm given in figure 2. The function of output gate *update* updates the global and local message-loss-type distributions at the end of transmission as described in section 2.3.

The SAN **retransmit** submodel is given in figure 7. This SAN models the sending and receiving of retransmissions. The activity *retrans_req_delay* represents the deterministic NAK transmission delay. The cases associated with the activity represent the probability of receiving the NAK and the loss type of the retransmitted message is type 1, the probability of receiving the NAK and the loss type of the retransmitted message is type 2, and the probability of not receiving the NAK, respectively. When this activity completes, depending on which case is chosen, zero, one, or two copies of a retransmitted message are placed on the channel. Since a single NAK message can request retransmission for more than one message, the activity *retrans_start* models transmission of the second and third possible retransmissions for the same NAK message.

Two activities are needed because we need to represent the NAK transmission delay only once. The cases associated with activity *retrans_start* determine what message loss type to place on the channel. The message loss type of a retransmitted message is determined using the algorithm in figure 3. Once the message loss type for a message is determined and the message is placed on the channel, activity *retrans_delay*, which represents the transmission delay for each retransmitted message, is enabled. This activity indicates that there is a message on the channel. The cases associated with activity *retrans_delay* represent the uncertainty in message delivery. When all messages have been removed from the channel, activity *end_retrans* is enabled, indicating the end of retransmission.

The functions of output gates *og2* and *og3* update the local and global variables (represented as places in the SAN) according to whether the retransmitted message is received or lost (as described in section 2.3). The place *lost_retrans* is the only local place to this SAN. The places *l_lost_by_one* and *l_lost_by_two* are common with other submodels and local to a process. The remaining places are common (global) to all processes.

4. Performability variables

Several performability variables of interest can be determined from the model. First, and probably the more interesting, is the expected steady-state time for a message to *stabilize*. Psync, formally, defines a message m sent by host h to be stable, if each process $q \neq h$ has sent a message m_q in the context of m ($m \prec m_q$) [14]. Thus, m_q serves as an acknowledgment to m. In a distributed application, such as replicated data, where messages are used to implement operations on the data, the shorter the message stabilizing time, the higher the system's throughput, especially if ordered execution of operations is required. Therefore, a short message stabilizing time is desirable, and the effect of message loss probability on stabilizing time is of interest. We consider a more refined notion of stabilizing, where a message is considered to be stable when all processes in the group have received it. Stabilizing, as defined in this paper, is thus a lower bound on stabilizing, as formally defined in [14]. This measure is useful to a protocol designer, who would like to minimize the time until all processes in a group receive a message. Since Psync supports a negative acknowledgment scheme, message stabilizing time is dependent on two factors: the reliability of the network in delivering messages and the frequency of sending messages.

The expected steady-state time for a message to stabilize can be computed from the composed SAN model using Little's result. To see this, let N be the number of processes in the group, λ be the application processing rate, v be the fraction of time a process is in the processing phase, and w be the expected steady-state number of unstable messages for all processes (this includes lost messages, new messages on the channel, and new messages ready to be transmitted on the channel). Then, the expected steady-state stabilizing time, S, is

$$S = \frac{w}{N \times \lambda \times v}.$$

N and λ are parameters of the model, and w and v can be determined through steady-state solution of the model (as discussed in the next section).

The fraction of the times messages of various types are on the communication channel is also interesting, since they provide insight into the proportion of time spent doing useful message transmission and the proportion of time spent in activities related to protocol operation. More precisely, we determine the fraction of time the channel is idle, the fraction of time a new message is on the channel, the fraction of time a retransmission is on the channel, and the fraction of time a NAK message is on the channel, for varying message loss probabilities. These variables also give an indication of the channel bandwidth which is needed to support an application for different workloads and fault environments.

5. Results

Once all the SAN models, the composed model, and the performability variables have been specified, the stochastic process representation of the model is automatically constructed. The model described results in a Markov regenerative stochastic process with 14 031 states. The resulting Markov process can then be solved by *UltraSAN* using methods developed for models with deterministic and exponential delays.

The results in this section were derived using the following network, environment, and protocol parameter values:

(i) a deterministic multicast message transmission delay of 15 ms,
(ii) a deterministic NAK transmission delay of 1 ms,
(iii) an average application processing rate which was varied (see below),
(iv) a message loss probability which was varied (see below),
(v) an equally likely message loss probability, whether the message is a new transmission, a retransmission, or a NAK (assumption (ix)), and
(vi) a value of MAX = 3.

The goal here was not to specify a set of parameter values that correspond to a particular, existing network, but to vary important parameters through reasonable ranges to see their effect on Psync's performability.

Expected message stabilizing time. Figure 8 shows the expected steady-state time for a message to stabilize, as a function of application processing rate λ and message loss probability p. For these measurements, we assumed the following values for p: 0.0, 0.000 001, 0.000 01, 0.0001, 0.001, 0.01, 0.05, and 0.1. For each value of p, we solved the model for values of λ equal to 1×10^{-7}, 1×10^{-6}, 1×10^{-5}, 1×10^{-4}, 1×10^{-3}, 1×10^{-2}, 1×10^{-1}, 1. These values were selected to measure the behaviour of Psync for different applications and different communication channel reliabilities. Figure 9 shows more clearly the effect of p on S near the minimum of each curve, by showing a close-up of figure 8.

As shown in figure 8, for low application processing rates, the expected message stabilizing time is extremely long compared to the ideal case, where p is 0.0. For example, at $\lambda = 1 \times 10^{-5}$, $S > 100$ ms for $p \geq 0.001$, compared to $S = 15$ ms in the ideal case. This is because at low application processing rates, the time between transmitting new messages is long. Since Psync employs a negative acknowledgment scheme, the frequency of sending retransmission requests for lost messages is low at low processing rates, resulting in long message stabilizing times.

As the application processing rate increases, S decreases until an optimal value of S is reached for specific values of λ and p. Increasing the application processing rate above the optimal value for particular values of λ and p increases S. This is because as λ increases, more new messages are generated, and processes experience longer

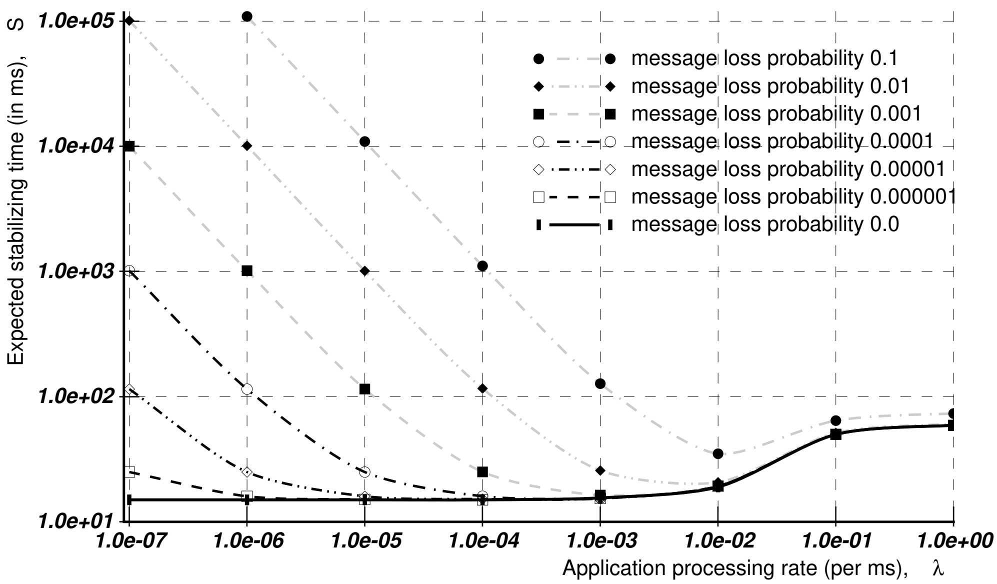

Figure 8. Expected steady-state time for a message to stabilize as a function of processing rate and message loss probability.

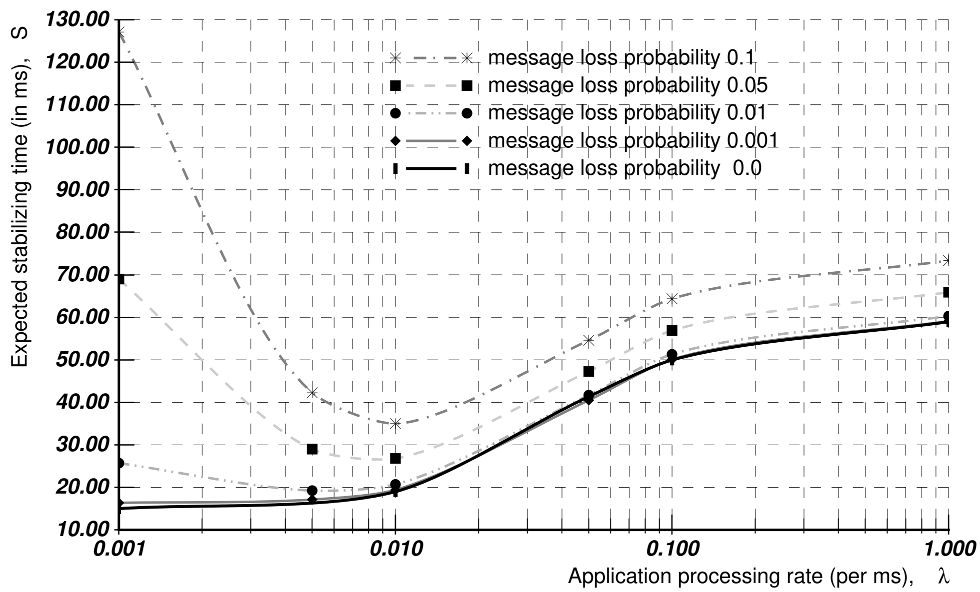

Figure 9. Expected steady-state time for a message to stabilize as a function of processing rate and message loss probability.

delays to access the communication channel. In its turn, increasing the average number of messages (w) in the system beyond some optimal number increases S. As λ increases further, S reaches a bound for some value of λ and remains at that value for higher arrival rates. This is because there can be, at most, three processes waiting to transmit a new message, putting a bound on w. As shown in figure 8, for high processing rates ($\lambda = 1$), w reaches a maximum fixed value ($w \geq 4.0$) and S levels off at a value ≥ 60.

In addition, figure 8 shows that the higher the value of p, the higher the value of S for the same value of λ. For low application processing rates, $\lambda \leq 0.001$, S is very sensitive to p. This is because for low values of $\lambda \leq 0.001$, the value of w is mostly due to the average number of lost messages between the processes. The number of messages on the channel and the number of messages waiting to be transmitted are very small (close to zero) for such small values of λ. As shown in the figure, increasing the value of p by a factor of 10 increases the value of S by a factor of 10 for $\lambda \leq 1 \times 10^{-4}$. However, for high application processing rates ($\lambda = 1$), S is less sensitive to p, compared with low

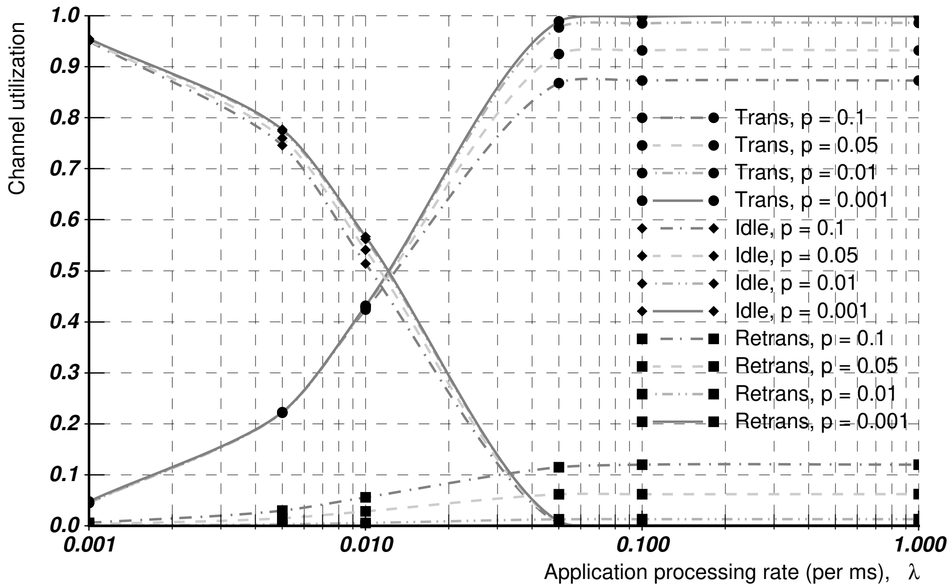

Figure 10. Channel utilization as a function of processing rate and message loss probability.

application processing rates. At such high rates, the value of w is mainly due to the number of messages waiting to be transmitted and the new message on the channel. Increasing p for such high rates increases S, but not by the same factor as was the case for low processing rates.

Channel utilization. Figure 10 shows how channel utilization changes as application processing rate changes for different message loss probabilities. In this figure, channel utilization due to sending NAKs is not shown because this value is very small. Channel utilization due to sending NAKs ranges between 0.01% for message loss probability 0.001 and processing rate 0.001 to 1.2% for message loss probability equal to 0.1 and a processing rate equal to 1.0. As shown in the figure, channel utilization due to sending new messages and channel utilization due to retransmissions are both proportional to processing rate. In the figure, p is the message loss probability, *Trans* is the channel utilization due to new messages, *Retrans* is the channel utilization due to retransmissions, and *Idle* is the fraction of time the channel is idle.

Effect of MAX on model behaviour. Finally, table 1 gives the fraction of time the total number of lost messages is MAX (MAX = 3). As shown in the table, the probability of reaching the maximum value is small even at a very high message loss probability ($p = 0.1$) and high processing rate ($\lambda = 1.0$). This confirms that setting MAX to three has little effect on the results obtained.

6. Use of null messages

While the results show the efficiency of the protocol (almost no time is spent transmitting negative acknowledgments), they also reveal the extremely long message stabilizing times that can result if the time between transmitting new messages is long. An example of an application that generates infrequent network transmission is a replicated server for which updates are relatively rare. Read requests would go to the local server and not generate Psync traffic, and hence the only traffic would be the updates.

Figure 10 shows when the time between transmitting new messages is 1 s, and the channel is idle 95% of the time. This suggests a modification to Psync that uses excess channel capacity, if available, to transmit 'null' messages. These messages carry no new information but instead contain information about the sender's view of the context graph. To ensure timely transmission of new messages, null messages have lower priority than new messages within a process. When a host receives a null message, it compares the sender's view of the context graph with its view of the context graph and requests retransmission for any message the receiver is missing. Losing null messages does not worsen message stabilizing time because these messages carry no new information and hence are not added to the context graph.

To study the effect of sending null messages, we modified the **transmit** submodel (in figure 5) as shown in figure 11. In this SAN, activity *null_msg* transmits null messages at a fixed rate, with exponential times between transmissions. When activity *null_msg* completes, it places a token in *null_send*. The function of *og2* is modified to set the marking of places *null_en* and *null_send* to zero and set the marking of *mess_to_send* to one. The function of *og1* places a token in *null_en* and places a token in *start_proc* if its marking is zero. Activity *transmit_start* is enabled, in turn, if there is a token in either of the places *null_send* and *mess_to_send*, and the condition discussed in section 3 holds.

The new place *on_channel*, which is global, is used to distinguish between null and new messages on the channel.

Table 1. Probability that the number of total lost messages MAX = 3.

Processing rate	p = 0.1	p = 0.05	p = 0.01	p = 0.001
0.001	5.13×10^{-3}	7.31×10^{-4}	6.5×10^{-6}	6.64×10^{-9}
0.01	7.3×10^{-3}	1.04×10^{-3}	9.24×10^{-6}	9.45×10^{-9}
0.1	9.5×10^{-3}	1.42×10^{-3}	1.31×10^{-5}	1.35×10^{-8}
1.0	9.51×10^{-3}	1.43×10^{-3}	1.32×10^{-5}	1.36×10^{-8}

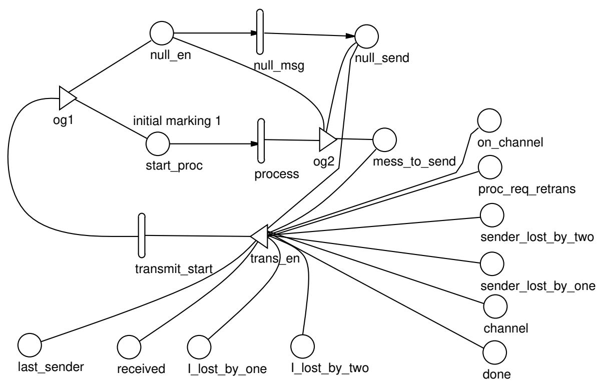

Figure 11. SAN submodel of **transmit** with null messages.

If the transmitted message is a null message, the marking of *on_channel* is set to two. If it is a new message, the marking of place *on_channel* is set to one. A recipient can determine the kind of message it has received by looking at the marking of *on_channel*.

This submodel (figure 11) is joined with the **receive** submodel (figure 6) and the **retransmit** submodel (figure 7) described previously. The joined submodel is then replicated three times, representing the three processes in the group. Using *UltraSAN*, Markov regenerative stochastic process representation of the system can be generated, and consists of 116 602 states.

Using the resulting model, we studied the effect of transmitting null messages for two message loss probabilities, $p = 0.1$ and $p = 0.0001$. We selected these values to see how null messages affect message stabilizing time under very lossy environments ($p = 0.1$) and under more reliable channels ($p = 0.0001$). For each message loss probability, we varied the application processing rate and the null message sending rate. We assumed the average transmission delay for a multicast null message was 15 ms. The remaining parameter values in the model are unchanged from the previous model. In each case, we solved the model for $10^{-7} \le \lambda \le 10^{-2}$. Higher application processing rates were not considered since null messages are not needed in this case, and the solution time becomes long, due to the high uniformization rate that is required to solve the model. The results of the runs for $p = 0.1$ and $p = 0.0001$ are shown in figures 12 and 13, respectively.

Figure 12 shows an example of the effect of transmitting null messages on message stabilizing time when there is a very high message loss probability. As shown in the figure, the message stabilizing time decreases rapidly as the null message sending rate increases. Table 2 shows channel utilization due to null messages as a function of null message sending rate. These results show that even with an extremely severe message loss probability ($p = 0.1$), a reasonable message stabilizing time can be achieved with an acceptable channel utilization due to null messages. If we assume a message loss probability equal to 0.0001, we can a achieve a close-to-ideal message stabilizing time with only 0.45% channel utilization, according to figure 13 and table 2. Thus large improvements can be made in the expected message stabilizing time without huge network bandwidth requirements.

Table 2. Channel utilization due to null message as a function of null messages sending rate.

Null message sending rate	Channel utilization
1×10^{-5}	4.5×10^{-4}
1×10^{-4}	4.5×10^{-3}
1×10^{-3}	4.5×10^{-2}
1×10^{-2}	4.3×10^{-1}

7. Conclusions

This paper presents an evaluation of Psync, a group multicast protocol, that accounts for the effect of message loss on the performance of the protocol. Such protocols are an important building block for dependable distributed systems, due to the strong guarantees they make concerning message delivery and ordering.

The paper makes two important contributions. First, it presents useful information regarding the performability of Psync under a wide range of workloads and message loss rates. The results show that while the protocol is extremely efficient in its use of bandwidth for heavy workloads, message stabilizing times are long if use is infrequent and loss probabilities are significant. The timeliness of message stabilization is an important aspect of a multicast protocol's performance, and these long times could prevent Psync's use in harsh fault environments. We then show how to improve message stabilization times through the use of null messages. Null messages contain no new information but are sent in the context of messages within a host's context graph. By exchanging information between hosts regarding their context graphs, null messages induce retransmission requests for lost messages, thus reducing stabilizing times of multicasts within the group. The evaluation results show that an adaptive algorithm that uses null messages as traffic

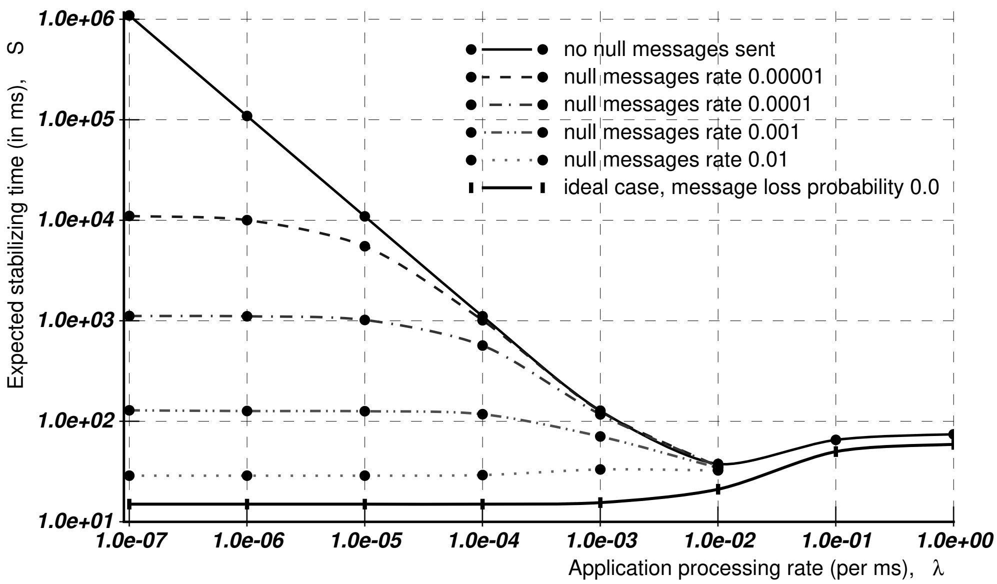

Figure 12. Expected steady-state time for a message to stabilize as a function of processing rate and null message rate, when message loss probability is fixed at 0.1.

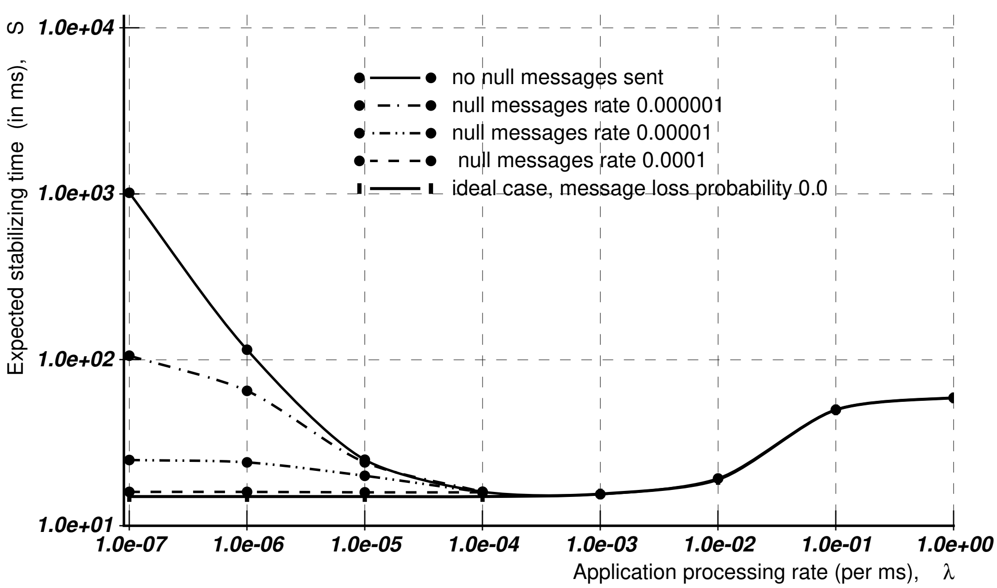

Figure 13. Expected steady-state time for a message to stabilize as a function of processing rate and null message rate, when message loss probability is fixed at 0.0001.

diminishes can significantly reduce message stabilization time with minimal use of network bandwidth. Similar insights about other multicast protocols could undoubtedly be gained using modelling techniques similar to those used in this paper.

Second, the paper illustrates the appropriateness of stochastic activity network models for analytically predicting the performance of group multicast protocols. By representing the behaviour of Psync as a composed stochastic activity network model, we are able to accurately characterize the protocol's mechanism for handling lost messages in a precise manner, and then automatically generate a stochastic process representation of the model. Reduced base model construction methods are used to reduce the size of the resulting Markov regenerative stochastic process, and recently developed numerical techniques for models with deterministic and exponential delays are used to obtain the desired performance measure. If reduced base model construction techniques had not been used, and the states of the process were to be the stable (also known as 'tangible') reachable markings, 82 856 and 737 702 states would have been needed for the first and second models, respectively. Thus with respect to this objective, the results show that it is indeed possible to model complex protocol operations as SANs and, by using reduced base model construction methods, obtain a Markov regenerative stochastic process that can be solved in reasonable time. The results bode well for the use of stochastic activity networks and reduced base model construction on practical protocol evaluations.

References

[1] Schneider F 1990 Implementing fault-tolerant services using the state machine approach: A tutorial *ACM Comput. Surveys* **22** 299–319

[2] Birman K, Schiper A and Stephenson P 1991 Lightweight causal and atomic group multicast *ACM Trans. Comput. Syst.* **9** 272–314

[3] Powell D (ed) 1991 Delta-4: A generic architecture for dependable computing *Research Reports ESPRIT* vol 1 (Berlin: Springer)

[4] Rodrigues L and Verissimo P 1992 xAMP: A multi-primitive group communication service *Proc. 11th Symposium on Reliable Distributed Systems (Houston, TX)* (Los Alamitos, CA: IEEE Computer Society Press) pp 112–21

[5] Kopetz H, Damm A, Koza C, Mulazzani M, Schwabl W, Senft C and Zainlinger R 1989 Distributed fault-tolerant real-time systems: The Mars approach *IEEE Micro* February pp 25–40

[6] Kaashoek M F and Tanenbaum A 1991 Group communication in the amoeba distributed operating system *Proc. 11th Symp. on Distributed Computing Systems (Arlington, TX)* (Los Alamitos, CA: IEEE Computer Society Press) pp 882–91

[7] Mishra S, Peterson L L and Schlichting R D 1992 Consul: A communication substrate for fault-tolerant distributed programs *Distrib. Syst. Engng* **1** 112–21

[8] Moser L, Melliar-Smith P, Agrawak D, Budhia R, Lingley-Papadopoulos C and Archambault T 1995 The Totem system *Proc. 25th Symp. on Fault-Tolerant Computing (Pasadena, CA)* (Los Alamitos, CA: IEEE Computer Society Press) pp 76–84

[9] Amir Y, Dolev D Kramer S and Malki D 1992 Transis: A communication sub-system for high availability *Proc. 22nd Int. Symp. on Fault-Tolerant Computing* (Los Alamitos, CA: IEEE Computer Society Press) pp 76–84

[10] Meyer J F 1980 On evaluating the performability of degradable computing systems *IEEE Trans. Comput.* **C-22** 720–31

[11] Goyal A, Shahabuddin P, Heidelberger P, Nicola V F and Glynn P W 1922 A unified framework for simulating Markovian models of highly dependable systems *IEEE Trans. Comput.* **41** 36–51

[12] Meyer J F, Movaghar A and Sanders W H 1985 Stochastic activity networks: Structure, behavior, and application *Proc. Int. Workshop on Timed Petri Nets (Torino)* (Los Alamitos, CA: IEEE Computer Society Press) pp 106–15

[13] Sanders W H and Meyer J F 1991 Reduced base model construction methods for stochastic activity networks *IEEE J. Selected Areas Commun.* **9** 25–36

[14] Peterson L L, Buchholz N C and Schlichting R D 1989 Preserving and using context information in interprocess communication *ACM Trans. Comput. Syst.* **7** 217–46

[15] Mishra S, Peterson L L and Schlichting R D 1993 Experience with modularity in Consul *Software Practice Experience* **23** 1059–75

[16] Sanders W H 1988 Construction and solution of performability models based on stochastic activity networks *Computing Research Laboratory Technical Report CRL-TR-9-88* The University of Michigan, Ann Arbor, MI

[17] Kulkarni V 1995 *Modelling and Analysis of Stochastic Systems* (London: Chapman-Hall)

[18] Lindemann C 1991 An improved numerical algorithm for calculating steady-state solutions of deterministic and stochastic Petri net models *Proc. 4th Int. Workshop on Petri Nets and Performance Models (Melbourne)* (Los Alamitos, CA: IEEE Computer Society Press) pp 176–85

[19] Shah B P 1993 Analytic solution of stochastic activity networks with exponential and deterministic activities *Master's thesis* The University of Arizona

[20] Sanders W H, Obal W D II, Qureshi M A, and Widjanarko F K 1995 The *UltraSAN* modeling environment *Perform. Evaluation J.* **24** 89–115