

Systematic Evaluation of Multicast Congestion Control Protocols

Karim Seada, Sandeep Gupta, Ahmed Helmy
Electrical Engineering-Systems Department
University of Southern California, Los Angeles, CA 90089
seada@usc.edu, sandeep@poisson.usc.edu, helmy@usc.edu

Keywords: Systematic protocol testing, congestion control, multicast, TCP-friendliness, STRESS

ABSTRACT

Congestion control is a major requirement for multicast to be deployed in the current Internet. Due to the complexity and conflicting tradeoffs, the design and testing of a successful multicast congestion control protocol is difficult. In this paper we present a framework for systematic testing of multicast congestion control protocols based on the STRESS methodology. STRESS has been used to study the correctness and performance of multipoint protocols. Here we extend it to study multicast congestion control, so as to tackle its new semantics and the high complexity of its verification. Our methodology is applied to a single-rate case study protocol and we are able to generate scenarios that can be used for testing multicast congestion control building blocks. Some of the interesting results are the effect of receivers joining and leaving on the throughput measurements, the effect of feedback suppression on congestion control, and the effect of changes in the special receivers representing the group. We hope that this will provide a valuable tool to expedite the development and standardization of such protocols.

1 INTRODUCTION

The growth of the Internet and its increased heterogeneity has increased the complexity of network protocol design and validation. The need for a systematic method to study and evaluate network protocols is becoming more important. Such methods aim to expedite protocol development and improve protocol robustness and performance. The STRESS (Systematic Testing of Robustness by Evaluation of Synthesized Scenarios) methodology [10][11] provides a framework for the systematic design and testing of network protocols. STRESS uses a finite state machine model of the protocol and search techniques to generate protocol tests.

In this paper we present a framework for testing multicast congestion control protocols based on the STRESS methodology. The goal of this framework is the generation of scenarios to evaluate the protocol correctness, performance, and fairness. The evaluation targets specific protocol mechanisms that are incorporated in current proposed protocols. Our aim is to have an abstract evaluation independent of a specific protocol. But since this is difficult to perform in practice, we have chosen, as a case study, a multicast congestion control scheme called *pgmcc*. *pgmcc* [15] is a single-rate multicast congestion control scheme that is designed to be fair with TCP. Although our case study is on *pgmcc*, we aim to evaluate the protocol behavior based on specific protocol mechanisms and building blocks, so that the results can be generalized to other protocols that use similar mechanisms.

The motivation and contribution of this work lies mainly in two areas. First, extending the STRESS methodology by applying it to a complex problem that has multiple dimensions: multicast,

reliability, and congestion control (with the related issue of TCP-friendliness). STRESS has been applied earlier to multicast routing and transport protocols, but congestion control is a new application that adds new semantics to the methodology such as modeling of sequence numbers, traffic, and congestion window. This problem is also characterized by a high degree of interleaving between events and long-term effects of individual events. New challenges are faced due to the large complexity of the search space, and are handled by introducing new types of equivalences and modifications to the search strategy to reduce its complexity. A mechanism for error filtering is developed to cope with the large number of error scenarios generated.

The second motivation and contribution is to provide a tool to aid in the design of these protocols. Multicast transport has been an area of extensive research and many protocols have been proposed [7][13]. One of the most important criterion that IETF [12] requires multicast protocols to satisfy is to perform congestion control in order to be safely deployed in the Internet. It is believed that the lack of good, deployable, well-tested multicast congestion control mechanisms is one of the factors inhibiting the usage of IP multicast [21]. By providing a systematic framework for evaluating these protocols we aim to expedite the development and standardization in this area. This tool is also targeted to application designers, so that if it is impossible to prevent all problems by the protocol itself, then there should be at least a systematic way to enumerate them.

The rest of this paper is outlined as follows. In Section 2 we provide a background about multicast congestion control and a brief description of *pgmcc*. Section 3 gives a brief background of STRESS and a general overview about our methodology. In Section 4 we present the protocol model and the errors examined. In Section 5 we show the search technique and its complexity. Section 6 shows the problems identified and Section 7 generalizes the error scenarios to specific building blocks. Detailed simulations for some of the identified problems are provided in Section 8. Section 9 shows some of the challenges faced and how they are solved. Conclusions and future work are presented in Section 10.

2 MULTICAST CONGESTION CONTROL

The design of a multicast congestion control protocol that provides high performance, scalability, and TCP-friendliness is a difficult task that attracts a lot of research effort. Multicast congestion control can be classified into two main categories: single-rate and multi-rate. Single-rate has a limited scalability because all receivers must receive data at the same (slowest receiver) rate. It also suffers from feedback implosion and drop-to-zero [2] (where the rate degrades significantly due to independent losses by a large number of receivers) problems. Multi-rate, where different receivers can receive at different rates, is more scalable but it has other concerns such as the complex encoding of data, possible multiple paths in the layered approach, and the effects of

receivers joining and leaving layers. TCP-friendly multicast congestion control can also be classified into window-based and rate-based. Window-based protocols have similar congestion window control as TCP, while rate-based protocols depend on the TCP throughput equation [14] for adjusting the transmission rate. Rate-based protocols typically have smoother rate transitions and are fair with TCP over longer time scales. Rate-based protocols also require accurate RTT computations which is not a simple task in multicast. For more details about these issues see [8][20]. Another possible classification for single-rate protocols is whether they are representative-based or not. Non-representative-based protocols solve the scalability problems using some aggregation hierarchy. This requires complex building of the hierarchy and may need network support. The performance is still limited and [4] shows that even without losses, small variations in delay can cause fast performance degradation with the increase in number of receivers. Representative-based protocols provide a promising emerging approach to solve the scalability problems, where a small dynamic set of receivers is responsible for providing the feedback [6]. The main challenge is the dynamic selection of representatives in a scalable and efficient manner with appropriate reaction to changes in representatives. This still needs further investigation. Examples of single-rate representative-based protocols are *pgmcc* (window-based) [15] and *TFMCC* (rate-based) [21]. We will use *pgmcc* as our case study, and in the rest of this section we will provide a more detailed description of *pgmcc*.

pgmcc [15][16] is a single-rate multicast congestion control scheme that is designed to be TCP-friendly. To achieve fast response while retaining scalability, a group representative called the *acker* is selected and a tight control loop is run between it and the sender. It is called the *acker* because it is the receiver that sends the ACKs. Other receivers can send NACKs when they lose packets, if a reliable transport protocol is used¹. *pgmcc* has been used to implement congestion control in the PGM protocol [18].

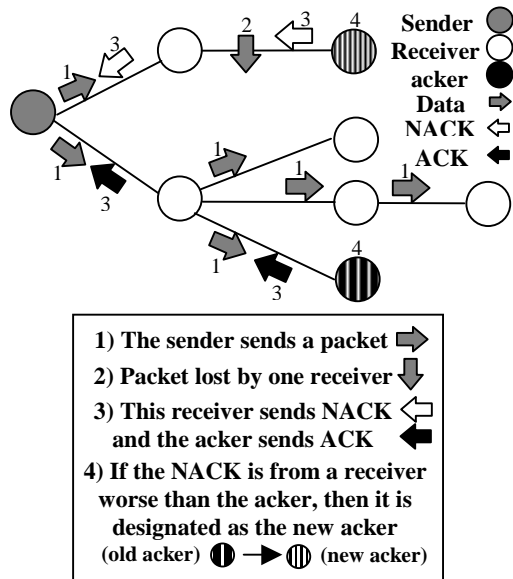


Figure 1: A sample *pgmcc* scenario

The *acker* is the representative of the group. It is chosen as the receiver with the worst throughput to ensure that the protocol will be TCP-friendly. A window-based TCP-like controller based on

¹ In [16] the authors state that *pgmcc* can be used with both reliable and non-reliable transport protocols. Non-reliable protocols may also need some NACKs to be sent for congestion control purposes.

positive ACKs is run between the sender and the *acker*. The feedback in *pgmcc* is provided in receiver reports that are used by the sender to estimate the throughput. They are embedded into the NACKs and ACKs and contain the loss rate and information for computing an estimate for the round trip time of sending receiver. There is a 32-bit field in the ACK called the bitmask, which indicates the receive status of the most recent 32 packets. It is included to help the sender deal with lost and out-of-order ACKs.

The most critical operation of *pgmcc* is the *acker* election and tracking. As mentioned, the receiver with the worst throughput is selected as the *acker*. When another receiver with worse throughput sends a NACK, an *acker* change may occur. Computation of throughputs uses information sent by receivers and the TCP-like formula: $T \propto 1/RTT \sqrt{p}$ where T is the throughput, RTT is the round trip time estimate and p is the loss rate [14].

A window based congestion control scheme similar to that used by TCP is run between the sender and the *acker*. The parameters used are a window W and a token count T . W is the number of packets that are allowed to be in flight and has the same role as the TCP window, while T is used to regulate the sending of data packets by decrementing T for every packet sent and incrementing it for every ACK received. On packet loss, W is cut by half and T is adjusted accordingly. A packet is assumed lost when it has not been acked in a number (normally three) of subsequent ACKs. W and T are initialized to 1 when the session starts or after a stall when ACKs stop arriving and a timeout occurs.

Some multicast transport protocols depend on router support for feedback aggregation. For example in PGM routers the first instance of a NACK for a given data segment is forwarded to the source, and subsequent NACKs are suppressed. During the study of the protocol there are different variations to consider. The main two variations are

- (i) reliable vs. non-reliable transport, and
- (ii) with feedback aggregation (router support) vs. without feedback aggregation.

3 METHODOLOGY FRAMEWORK

STRESS (Systematic Testing of Robustness by Evaluation of Synthesized Scenarios) provides a framework for the systematic design and testing of network protocols. STRESS uses a finite state machine (FSM) model of the protocol and search techniques to generate protocol tests. The generated tests include a set of network scenarios each of which leads to violation of protocol correctness and behavioral requirements. In [9][10] STRESS has been used to verify multicast routing protocols using forward and backward search techniques. In [11] it is used for the performance evaluation of timer suppression mechanisms in multipoint protocols. In [1] Mobile IP and multicast over ATM are studied. Currently STRESS is used for the verification of wireless MAC protocols. In this paper we extend STRESS to study multicast congestion control. In the remainder of this section we give a high-level description for our methodology and how different blocks described throughout the paper fit within. Hopefully, this will simplify reading the rest of the paper as we go into more details.

Figure 2 shows the main framework. The protocol designer or tester inputs the protocol specifications and correctness conditions in the form of a state model (Section 4.1), transition table (Section 4.2), and error model (Section 4.3). The search engine (Section 5) explores the search space for errors and generates a set of error scenarios. These error scenarios are fed into the error filter (Section 9.1) which classifies the scenarios and outputs a compact set of error scenarios that can be used for testing the protocol.

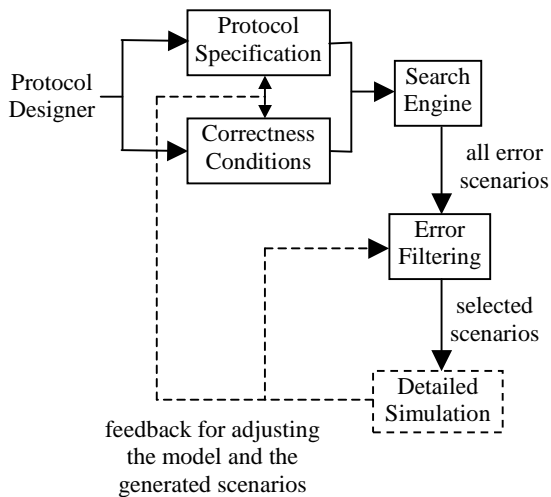


Figure 2: Block diagram of our methodology

These scenarios can be used for more detailed simulation to confirm their ability to disclose errors/low-performance and to ensure that no crucial details are missing. The feedback from the simulations can be used by the designer to adjust the input and the filtering approach. Currently the search engine and error filtering have been automated.

4 PROTOCOL MODEL

In order to process the protocol and apply the STRESS methodology we need to have a suitable model that represents the protocol and the system. The model is built according to the protocol specification and additional assumptions based on our understanding of the protocol behavior and its context. The model consists of

- (i) a representation of the protocol states,
- (ii) a state transition table defining the possible events and their effect on states,
- (iii) a set of errors that represents our correctness criteria.

Critical to our methodology is how the topology is modeled. A multicast transport session topology consists of a sender, a group of receivers, links, and routers. This detailed topology can be very complex for our model and search. Also, many of the details are not required. For a multicast congestion control protocol operation, the topological details are reflected mainly on which packets are received by the sender and receivers, and when these packets are received. Our goal here is to simplify the topology and still capture characteristics as delays, reordering, selective and correlated losses, and feedback suppression.

Figure 3 shows the simplified topology. A virtual end-to-end channel exists between the sender and each receiver. We are able to capture all the required characteristics with this topology by using all permutations of events (this will become clearer when we explain the search) including losses, reception of packets in different orders, and suppression. The order in which packets are received or dropped in this topology emulate the delays, routing effects and feedback aggregation in larger topologies. This simplification may hide some details, but we believe that such details do not affect our protocol evaluation. In addition, to further verify the validity and effect of our generated scenarios, we performed more detailed simulations (see Section 8).

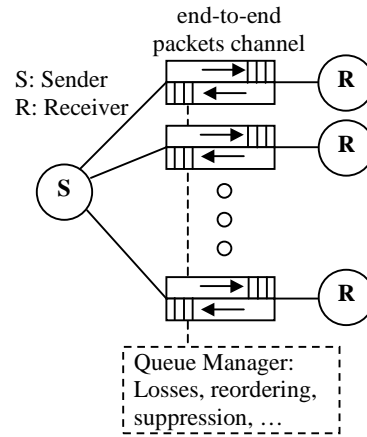


Figure 3: Topology Model: The path between the sender and a receiver is represented by a virtual end-to-end channel. The topological details (delays, losses, suppression, and routing effects) are captured by taking all permutations of loss and packet reception events.

4.1 States

The global state is an aggregation of local states of a sender and a number of receivers, with a separate end-to-end channel between the sender and each receiver, as shown in the topology in Figure 3. Each entity has a number of state parameters that define its local state. These parameters are as follows:

- Sender state: packets sent, ACKs received, sequence number of next packet to send, sequence number of next ACK expected, current acker, current acker throughput, window size, number of tokens, duplicate count (to count the duplicate ACKs), ignore count (for adjusting the token count after a window cut).
- Receiver state: receiver ID, packets received, sequence number of next packet expected, loss ratio, member or not.
- A Packets channel between sender and each receiver:
 - Packet: type (data, ACK, or NACK), sequence number, acker (if it is a data packet), packets received by acker (if it is an ACK), loss ratio (if it is an ACK or NACK), missing packet (if it is a NACK)

The channel holds the packets between the sender and the receiver with data going in one direction and feedback (ACK or NACK) going in the other direction. The underlying network characteristics such as packet losses, packet reordering (e.g., due to route changes), and network delays are handled by the channel. Multicast group membership is represented by defining at a state whether each receiver is a group member or not. It is assumed that an underlying multicast routing protocol exists and delivers the packets to current members only. In addition to the combination of these local states we added also a *global loss estimator* in our global state, in order to estimate the loss rates on different channels independent of local computations by receivers. As will be shown later, this is required only for the protocol evaluation.

4.2 Transition Table

A simplified version of this protocol's transition table is shown in Table 1. The transition table describes, for each possible event, the condition of its occurrence, and the effect of that event. For example, *Send* is the event of sending a packet, its condition is that the number of tokens is above 1, and its effect is putting the packet in the group member receivers' channels and adjusting the sender parameters. During the search process the conditions of these events are checked at every state, and if satisfied the corresponding

Table 1: A simplified transition table for the protocol

Event	Precondition	Effect (Transition)
Send	Tokens ≥ 1	-Decrement number of tokens ($T=T-1$) -Add packet to sent list and increment the next sequence to send -Add packet to all members' channels
Join	Receiver not member	-Receiver becomes member
Leave	Receiver member	-Receiver becomes non-member
Loss	Packet in channel	-Remove packet from channel -If it is a data packet, update global loss estimation for that channel.
Receive Data	Data packet in channel	-Remove data packet from channel and update global loss estimation -If that receiver is a member, update its receive list, next sequence to receive, loss ratio -If lost packets discovered, generate and send NACKs, update loss ratio -If that receiver is acker send ACK
Receive ACK	ACK in channel	-Remove ACK from channel -If ACK from current acker, update its throughput using its loss ratio and RTT -Update ACK list and sequence of next ACK expected -Increment window and token ($W=W+1/W$, $T=T+1+1/W$) -If packet losses discovered (duplicate ACKs), cut window ($W=W/2$) and adjust token (ignore next $W/2$ ACKs)
Receive NACK	NACK in channel	-Remove NACK from channel -Compare that receiver throughput with acker throughput, and if NACK is from a receiver worse than acker then choose it as new acker -Retransmit

event is triggered and a new state is generated. Slow start is ignored in our model, since it is a transient phase and we are more interested in testing the steady state behavior. Some changes can be made to the transition table to study variations. For example the retransmission of packets can be omitted to consider non-reliable transport. NACK suppression is emulated by NACK dropping.

4.3 Errors Model

Table 2 shows the errors we check for in our model. These errors are not necessarily errors in the protocol itself, since some of them may be compensated for by other layers (e.g., by an application). Nevertheless, they are potential problems that should be considered during the implementation of the protocol or by the applications using it. Some of these errors violate the correctness such as 'gaps in sequence space', which means that, for some reason, the sender has not sent all the packets in sequence. Most of the other errors cause mainly performance degradation such as 'unnecessary starvation', which means that the sender has packets to send but it is stalled, although the group receivers are ready. Duplicate packets means that a receiver received the same packet more than once, which is a waste of the network bandwidth. Wrong congestion notification can lead to an unnecessary window cut and reduction in sending rate. An error that leads to fairness violation is 'wrong acker selection', which means that a receiver

Table 2: The errors checked in our model

Error	When to check	How to check
Unnecessary starvation	At the end of search or at timeout	Tokens < 1 and no more data or ACKs in channels
Duplicate packets	At data reception by receiver	Check receive list for if the packet already exist
Gaps in sequence space	At the end of search	Check sent list of the sender for gaps
Wrong congestion notification	At congestion (when packet loss detected)	Check whether packets are really lost (using global loss estimator)
Wrong acker selection	At sending (or at the end of search)	Comparing acker and receivers throughput (by global loss estimators)

other than the worst receiver is chosen as acker. Since in this protocol we consider fairness (TCP-friendliness) as being synonymous to choosing the worst receiver as acker, 'wrong acker selection' models the fairness violation.

During the search we look for erroneous states by checking for these errors conditions. The point at which an error is checked is important, since this has a significant impact on the search time and on the number of erroneous states generated. It also affects the number of false errors. We want to reduce the error checking as much as possible by considering only the minimum set of states required to discover the error. For example whenever possible we check for errors only at the end of the search path (leaves of search tree). This is possible for errors that become permanent once they appear. Whether an error is permanent or not depends on the events that may change its conditions. For example 'unnecessary starvation' is a permanent error and can be checked at the end, since it leads to a stall, but if we include timeouts in our model, then timeout will be the event that can break the starvation condition and we will need to check for that error at timeouts.

5 SEARCH PROBLEM

The problem of test synthesis can be viewed as a search problem. By searching the possible sequence of events over network topologies and checking for errors at specific points, we can construct the test scenarios that stress the protocol. Forward search is used to investigate the protocol state space. As in reachability analysis, forward search starts from initial states and applies the stimuli (events) repeatedly to produce the reachable state space (or part thereof). Conventionally, an exhaustive search is conducted to explore the state space. In the exhaustive approach all reachable states are expanded until the reachable state space is exhausted. Of course in our case we cannot expand forever, so we limit our search by a maximum sequence number, which is the number of packets to be sent by sender. The search process is completely automated.

At each state during the search, the events preconditions are checked, and transitions are applied to generate new states. A list of all visited (already expanded) and working (still to be expanded) states is kept, so that new states can be checked against it to avoid redundant search. Error checking is performed at specific states during the search according to the type of error. An example of a part of the search tree is shown in Figure 4. Notice that this leads to state space explosion and techniques must be used to reduce the complexity of the space to be searched.

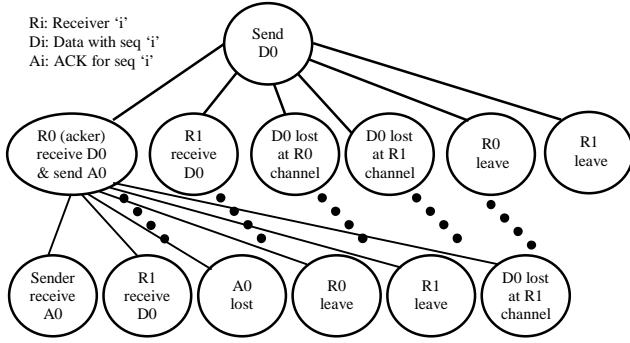


Figure 4: An example of search space expansion

Table 4: Number of comparisons without equivalences and hashing

rcvr \ seq	1	2	3
1	109	4650	2.00e5
2	311	1.48e5	1.50e8
3	17891	1.16e9	

Table 5: Number of comparisons with equivalences and hashing

rcvr \ seq	1	2	3
1	2	16	108
2	4	71	2887
3	57	18632	

Table 6: Number of transitions without equivalences

rcvr \ seq	1	2	3
1	22	220	1880
2	42	1568	6.7e4
3	423	1.8e5	

Table 7: Number of transitions with equivalences

rcvr \ seq	1	2	3
1	7	25	81
2	14	147	1931
3	101	10514	

5.1 Search Complexity

To get a measure of the complexity we are facing and to understand the source of this complexity, we show in Table 3 the number of visited states in a topology of 2 receivers and increasing sequence number (the limit of the search). By using all the events (complete interleaving) we find that the number of visited states increases at a very high rate with the increase in sequence numbers, so that search cannot be completed even for 4 sequence numbers (we are able to reach it after performing some optimizations as will be shown later). The next three columns show the reduction in the number of visited states by doing the search without packet reordering (packet reordering means that the channel delivers packets in all possible orders, so that at each state there is an event triggered for each packet to be received independent of its location in channel. Removing reordering means that packets in the same channel and same direction arrive in order, but interleaving still exist between packets in different channels or different directions), packet losses, and members join and leave, respectively. In the next column we show that without all these three types of events, we can reach 20 sequence numbers with a number of visited states about half of those reached with 3 sequence numbers and complete interleaving. These three events are considered as undesirable events in our search (Another reason for considering them undesirable is that they are the main cause of errors). Table 4 and Table 6 show the total number of comparisons and the number of transitions and their large increase with the increase in number of receivers and sequence numbers.

5.2 Comparison Time

In order to avoid redundancy in state generation, a list of all visited (already expanded) and working (still to be expanded) states is kept, against which new states can be checked. The comparison time is the most time consuming process during the search, since the comparison time for a state is proportional to the number of visited and working states at that time. As the number

Table 3: Number of visited states with 2 receivers. The last column shows the number of visited states when equivalences (see Section 5.3) are exploited

Seq	Complete	Without reorder	Without losses	Without join & leave	Without all	Complete with equivalences
1	64	64	52	16	7	16
2	384	264	260	96	16	96
3	28264	3820	11372	7066	40	6082
:						
20					14520	

of states increases, the comparison time becomes extremely large, which limits the number of sequence numbers that can be covered. Some statistics are shown in Table 4. To reduce the comparison time and expand our search space, we use hashing. A hash table is used for storing all the states (visited and working); the working list is still used for the search process only. Use of hashing reduced the search time by several orders of magnitude and allowed us to cover 4 sequence numbers instead of 3.

5.3 Equivalences

Exhaustive search has exponential complexity and the number of states generated is very large. To reduce this complexity we use the notion of equivalence. The notion of equivalence implies that by investigating an equivalent subspace we can test for protocol correctness. That is, if the equivalent subspace is verified to be correct then the protocol is correct, and if there is an error in the protocol then it must exist in the equivalent subspace. We use three types of equivalence: state equivalence, path equivalence, and error scenarios equivalence. Error scenarios equivalence is shown later when we describe the error filtering approach used to reduce the number of generated erroneous scenarios. State equivalence means that two states are not exactly equal, but they are equivalent from our viewpoint. Path equivalence is to find paths that lead to the same states and prune them, this can cause a major reduction in complexity as will be seen. Table 5 shows the number of visited states when equivalences and hashing are used. This can be compared to the direct search results in Table 4. Table 7 shows the reduction in number of transitions due to equivalences in comparison to Table 6.

5.3.1 State Equivalence

Examples of state equivalences used in our case study are *channel equivalence* and *receiver equivalence*. Channel equivalence is when two global states have similar local states and similar packets in their channels but in different order. They are

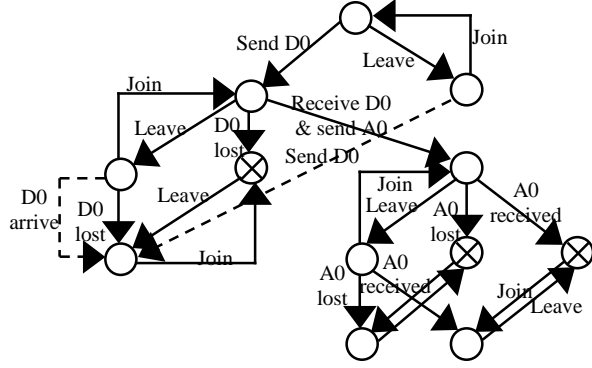


Figure 5: Search tree for one receiver and one sequence number without path equivalence (number of visited states: 12, number of transitions: 22, number of comparisons: 109)

considered equivalent, since we consider all orderings of packet arrival. Receiver equivalence occurs when several receivers have identical local states. In this case it is sufficient to apply the transitions for only one of them, since the same search sub-trees will be generated for others. For example, if we have two receivers R1 and R2 with the same parameters, we do not need to apply the same events for both of them, since this will just create identical states with R1 and R2 swapped. One way to implement that is to use descriptors to represent channels and receiver states where the packet order and receiver ID are ignored in the descriptor representation. A more general approach is to use a global descriptor for representing the whole state taking all the state equivalences into account.

5.3.2 Path Equivalence

Path equivalences are based on the notion that if two events do not affect each other, then it is not necessary to consider interleaving between them. Detecting path equivalences can be very complex, but it leads to large reductions in search space. One way to identify path equivalences is to look at the different events in the transition table and see how they affect each other, so that we need not consider interleaving between events that do not affect each other directly. An example is shown in Figure 5, which is the search tree for one receiver and one sequence number. Notice that there are a lot of redundant paths. Consider for example the leave and join events. After a leave, you need to join only if there is a change (send or receive) in your channel. Also, after a join, you need to leave only if there is a change in your channel. A loss of a packet is affected only by the receiving of that packet, which means that repeating it on all paths is redundant.

Taking these equivalences into account we see in Figure 6 that the complexity is reduced, and with higher number of receivers and sequence numbers the reduction is greater. In Figure 6 the join and leave events of a receiver are associated with sending and receiving of packets on the channel of that receiver and the loss event is triggered only once under the same send sub-tree. So we have two major events (send and receive) and other events are internally associated with them. Larger reduction can be obtained by restricting the interleaving between packets received by different receivers. A receiver is affected only by changes occurring on its channel, and not by other receivers' channels (although the effect of that on the sender is taken into account during the sending of new packets). So instead of taking the permutations of packet arrival of all packets in all channels at a

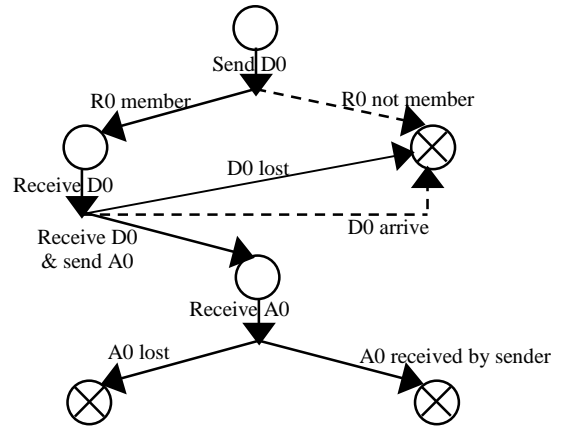


Figure 6: Search tree for one receiver and one sequence number with path equivalence (number of visited states: 6, number of transitions: 7, number of comparisons: 19)

certain state, we need to take only the permutations of each individual channel with the combinations between receivers. This reduces the number of different packet arrival orders from $\sum_{k=0}^l \binom{l}{k} k!$, where $l = \sum_{i=1}^m n_i$ is the total number of packets in m channels (m is the number of receivers, and n_i is the number of

packets in channel i), to $\prod_{i=1}^m \sum_{k=0}^{n_i} \binom{n_i}{k} k!$. E.g. in the case of two receivers ($m=2$), with $n_1=3$ and $n_2=2$ packets in their channels, the number of different packet arrivals reduce from 326 to 80.

5.4 Protocol Mechanisms

Due to this large complexity, it is not expected that the sequence number limit can be increased too much, so a good question here is "how many sequence numbers are required to provide a reasonable evaluation of the protocol?" To have some hints about that we analyze the protocol mechanisms and identify minimum sequence numbers that are necessary to trigger all of them. Table 8 shows the protocol mechanisms presented in our model. These mechanisms cover a wide range including mechanisms for sending of packets, receiving of packets, window management, loss ratio and throughput calculation, acker management, membership management, and mechanisms related to channel losses and reordering. Using reachability analysis we can empirically show that 6 sequence numbers are necessary to trigger all these mechanisms (including window cut and token adjustment) if the number of duplicate packets for loss indication is 3. But by using 1 duplicate packet this can be reduced to 4 sequence numbers only. Also from the errors table and transition table we can show that no errors will occur without triggering of the undesirable events (packet reordering, losses, leave & join). Using reachability analysis again we found that only 3 sequence numbers are required to trigger all of the undesirable events in different combinations. This indicates that the error scenarios detected using 4 sequence numbers likely cover most of the protocol errors. Another indication is that the same erroneous scenarios were found when we increased from 3 to 4 sequence numbers. We are still investigating further to ensure that no other error scenarios can be found by considering higher sequence numbers.

Table 8: Protocol mechanisms in our model

• Data send by sender	• Receiver leaving & joining
• ACK send by Acker	• Acker leaving & joining
• NACK send by receiver (&Acker)	• Channel
• Data receive by receiver (&Acker)	• Data reordering (to Acker, to receiver)
• ACK receive from current Acker (by sender)	• ACK reordering (from new Acker, from old Acker)
• ACK receive from old Acker	• NACK reordering (from Acker, from receiver)
• NACK receive by sender	• Data loss (to Acker, to receiver)
• Acker changing (sender)	• ACK loss (from new Acker, from old Acker)
• Window & token increase (sender)	• NACK loss (from Acker, from receiver)
• Window cut (sender)	• Data removal due to a non-member receiver
• Loss ratio calculation (receiver)	
• Throughput estimation (sender)	
• Throughput comparison (sender)	
• Token adjustment after cut (sender)	
• Duplicate ACKs (sender)	

Table 9: Summary of detected problems

• Unnecessary starvation	• Wrong Acker selection
• All data to Acker lost	• Very bad Acker (all data or ACKs lost), this can lead to wrong Acker selection after timeout
• All ACKs lost	• Very bad receiver (data lost, ACKs or NACKs lost)
• Some data, some ACKs lost	• NACK suppression
• Acker leaves (crashes)	• False NACKs
• Duplicate packets	• Wrong loss ratio estimate by receiver due to leaving and joining
• False NACKs due to out-of-order (OOO) data causing retransmissions	• Gaps in sequence space
• Sender or receiver crashes (non-deterministic)	• Sender or receiver crashes (non-deterministic)
• Wrong congestion notification (wrong window cut)	
• OOO data packets	
• Acker switch causing OOO ACKs	

6 DETECTED PROBLEMS

Using our methodology we are able to generate scenarios that lead to the following problems. We want to emphasize that these problems are not necessary errors in the protocol itself. The reason is that they may be handled by other layers, by the application, or during the implementation. However these problems are important and should be taken into account by the protocol designer or by applications that use the protocol. In Table 9 we list a summary of all the problems detected. Some of these problems are obvious and known, but here they are detected in a systematic way using an automated tool. In the following subsections we will explain the non-obvious and important problems.

6.1 Unnecessary Starvation

We define unnecessary starvation as a scenario where the sender has packets but it is not able to send, although the group receivers are ready to receive more packets. In practice, the state may remain stalled until the sender starts sending again, e.g. after a timeout. This leads to a reduction in throughput and longer delays experienced by receivers. There are two types of scenarios that can lead to unnecessary starvation. The first type of scenarios is when all the data to acker or the ACKs from acker are lost, since the sender depends on the ACKs for generating new packets. This is an expected problem and, according to the protocol definition, in this case the rate should follow the worst receiver. The other scenario that can lead to unnecessary starvation is when the group representative (the acker in our case study) leaves the group without notification (or crashes) and there is no more feedback to the sender. The sender will wait for some time (until a timeout) before start sending again and switching to another representative.

6.2 Duplicate Packets

In these scenarios a receiver receives the same packet more than once, which wastes the network bandwidth. A scenario leading to duplicate packets is when data packets arrive out-of-order at the receiver causing it to send NACKs for packets still on their way. Adding some delays in sending the NACKs by receiver and the retransmissions by sender can reduce this effect, but there will be a

tradeoff between the delay value and the responsiveness in case of packet losses. Another possible reason for duplicate packets is sender or receiver crashes ending in a non-deterministic state.

6.3 Gaps in Sequence Space

Gaps in sequence space means that the sender is not sending packets in sequence, which can also occur due to crashes leading to non-deterministic states. Sender crashes may be considered unlikely, and if they occur the session can be restarted. However receiver crashes have a high probability as the number of receivers increase. Hence we need to make sure that malicious behavior by a single receiver will not affect the entire session. For example, if a receiver sends an ACK or NACK with a sequence number greater than the highest sequence number sent by the sender, how the sender will respond to that? Such malicious behavior can be due to a non-deterministic state caused by a receiver crash or may be an intentional denial of service attack. We believe that such cases should be considered if the protocol is to be scaled to large number of receivers.

6.4 Wrong Congestion Notification

Wrong congestion notification means that the sender receives a congestion notification (duplicate ACKs) and cuts its window, although there is no congestion and this reduction in sending rate is not necessary. This can occur due to out-of-order data arrival at the receiver causing wrong interpretation of loss. Another scenario that leads to wrong congestion notification is when an acker switch occurs between receivers with large difference in delay. This leads to ACKs arriving out-of-order, which can cause severe performance degradation if not handled properly by the sender as shown in the simulation we performed in [17].

6.5 Wrong Acker Selection

Wrong acker selection means that a receiver other than the worst receiver is selected as the acker. This leads to fairness violation and affects the TCP-friendliness of the protocol. Several scenarios can lead to wrong acker selection. If there is a very bad receiver such that it is not even able to send feedback, another receiver will be selected as the acker. The other receiver may have a higher rate,

which can make things worse on the bad receiver path. This is complementary with the unnecessary starvation case. In unnecessary starvation, the problem is on the multicast session itself, while here we are concerned about competing traffic. Simulations in Section 8 will show some interesting tradeoffs between these two cases.

Out-of-order packets causing false NACKs can also lead to wrong acker selection. For example, if data packets arrive at a receiver out-of-order (e.g. due to route change), the receiver sends NACKs for the missing packets with a high loss ratio. This loss ratio may cause the sender to designate it as the acker, although it is a good receiver. The effect of this scenario will depend on how loss ratio is computed in the case of out-of-order packets, how long the receiver waits before sending the NACK, and what happens to the loss ratio after the receiver discovers that the packets were not lost. In many cases such specific details are not specified or even articulated during protocol design.

Some multicast transport protocols depend on router support for suppressing NACKs to avoid the NACK implosion problem. NACK suppression can cause the NACKs of worse receivers to be suppressed, especially if they are far away, which leads to wrong acker selection. In [17], we have shown using simulation, how this problem affects TCP-friendliness.

Another type of scenarios that can lead to wrong acker selection is where loss ratio is estimated incorrectly by receivers leaving and joining the group. If a receiver leaves the group and then joins again after a while, the loss ratio estimate by that receiver will not be an accurate estimate of the current channel situation. Moreover, it may consider packets not received as lost in computing the loss rate, although they were not lost. This problem can have a large effect on the operation of the protocol, if receivers are not stable or if there are problems in the underlying multicast routing and group membership protocol. Since the throughput computation depends on the local loss ratio computed by receivers, inaccuracy in that value can significantly affect the protocol behavior. In pgmcc, the throughput is used only for acker selection, but in rate-based protocols that depend on throughput computations for determining the sending rate, this problem can have a larger impact.

7 GENERALIZATION OF RESULTS

Table 10: Multicast transport building blocks

• Data reliability	• Congestion control
• Loss detection/notification	• Congestion feedback
• Loss recovery	• Rate regulation
• Group membership	• Receiver controls
• Membership notification	• Security
• Membership management	

In this section we describe how our scenarios can be used to test other protocols that use similar mechanisms. In order to identify the protocols that can be tested by these results, we look at how they can be decomposed into specific building blocks. IETF provides a framework for reliable multicast transport building blocks [19] in order to support the development and standardization of these protocols. The protocols can be composed of these building blocks, and the building blocks can be reused across multiple protocols. This building block approach is also useful in supporting the testing of protocols, by simplifying it to testing of individual building blocks. Table 10 shows some of the components presented in [19] that can be tested by our scenarios.

Here we describe those scenarios generated by our framework for the specific case study protocol and how they can be used to test various abstract components:

- All data to acker² or ACKs lost: This can be used to test the congestion feedback component and how it reacts when there is no feedback. It also tests the receiver controls component, since a single receiver here can starve the whole multicast session.
- Acker leaves or crashes: This tests the membership management component and how it deals with special members, such as ackers. This is also important for other protocols having group representatives [6] that are responsible for providing the feedback to sender. For example, in TFMCC [21], a special receiver called CLR provides the feedback for increasing the transmission rate.
- Out-of-order data or ACKs: This tests the congestion feedback and how feedback is provided and handled. It tests also the rate regulation and how the rate is adjusted due to feedback.
- Acker switch: Also tests the congestion feedback and rate regulation in case of change in the feedback provider.
- Very bad receiver: This tests the receiver controls since we want to stop that receiver from starving the whole session. Some protocols (e.g. [5]) prune bad paths from the multicast tree or force some bad receivers to leave, in order to have an acceptable rate. It also tests rate regulation, since we do not want to make it worse for that receiver.
- Sender or receiver crashes: This tests how the group membership is managed.
- False NACKs: This tests several components such as loss detection, notification, and recovery, for example how losses are handled and whether timers are set before sending notifications. Congestion feedback is affected by false NACKs, since NACKs carry the feedback.
- Receiver leaves and joins: This tests the membership management and also the congestion feedback if the receiver performs computations for things like the loss rate and RTT. In pgmcc the receiver computes the loss rate. In other protocols such as TFMCC, it computes the loss rate and also the RTT.
- NACK suppression: This tests the congestion feedback and its interaction with router support.

Another building block that can be considered here is security. Many of the problems presented may not only be caused by the network, but they can also be due to intentional denial of service attacks. This is important if applications are covering a wide range of unknown receivers. This denial of service can be for the multicast session or for other competing (e.g. TCP) sessions. Examples of these scenarios are receivers sending wrong feedback, false NACKs, or not sending feedback at all. This may lead to performance or fairness problems.

8 SIMULATION OF GENERATED SCENARIOS

To show the utility of our methodology, we have conducted a set of detailed simulations for the generated error scenarios. These simulations also demonstrate how the low-level errors detected lead to high-level performance problems and long-range effects.

² The term ‘acker’ here can be substituted by any group representative responsible for providing the feedback to the sender in other protocols.

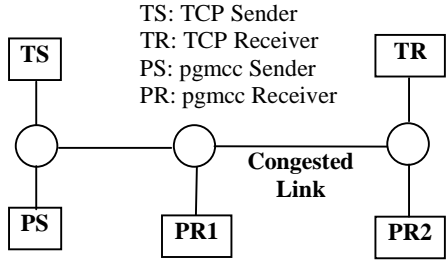


Figure 7: A pgmcc session with two receivers PR1 and PR2, where PR2 is the worst receiver (accordingly the acker) due to the congested link. A TCP session is also competing for the congested link.

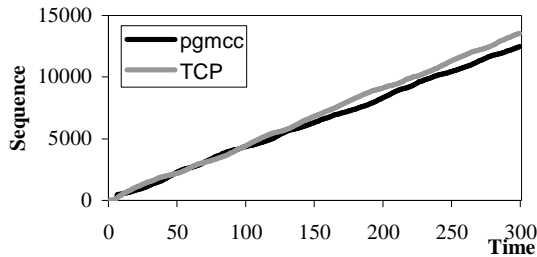


Figure 9: Throughput of pgmcc and TCP without acker switch

We show the results obtained using the NS-2 simulator [3]. Due to space limitations we show only simulations for the *acker leaving* problem, which causes *unnecessary starvation*, and the *receiver leaving and joining* problem, which can cause *wrong acker selection*. Other simulation results are provided in [17].

The source models used in the simulation are FTP sources with packet size of 1400 bytes. The links have propagation delay of 1ms, and bandwidth of 10Mb/s, unless otherwise specified. The queues have a drop-tail discard policy (RED is also examined and the results are similar) and FIFO service policy, with capacity to hold 30 packets. For TCP sessions, both Reno and SACK are examined and they support similar conclusions. In the graphs we show the sequence numbers sent by the sender vs. the time. This has the same effect as showing the throughput.

8.1 Unnecessary Starvation

A generated scenario that leads to unnecessary starvation is that of the acker leaving the group (or crashing), as was shown in Section 6.1. Guided by that scenario we build the topology in Figure 7, where we have a pgmcc session with a sender and two receivers. The path to the receiver PR2 has a congested link (1Mb/s), hence PR2 is the worst receiver and becomes the acker. A TCP session is added to generate extra traffic over that path. The experiment runs for 300 seconds. Every 10 seconds the acker PR2 leaves the group for 1 second. During this period the sender receives no ACKs and is not able to send any packets until it timeouts. This leads to drastic reduction in throughput by about 75% as shown in Figure 8.

8.2 Wrong Acker Selection

Incorrect loss ratio estimation by receivers, due to leaving and joining the group, can lead to wrong acker selection as detected by our methodology (Section 6.5). We again use the topology in Figure 7, but this time it is the other receiver PR1 that leaves the group for 1 second, every 10 seconds. After rejoining PR1 can

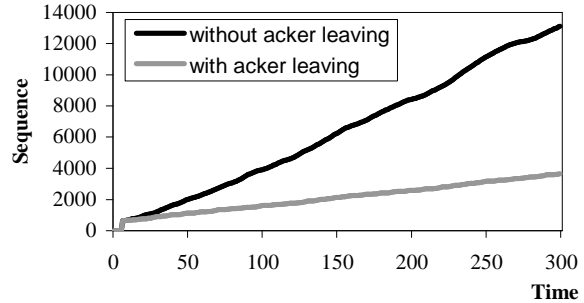


Figure 8: Throughput of pgmcc without and with acker leaving

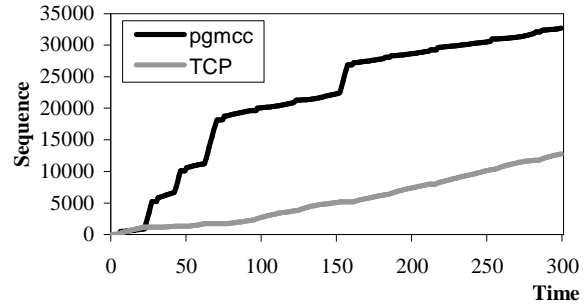


Figure 10: Throughput of pgmcc and TCP with wrong acker switch

have an incorrect estimation of the loss ratio, due to the packets missed during its leave and it starts sending NACKs for the missing packets. The sender may select PR1 as the acker instead of PR2. A longer delay (10ms) is added to the link leading to PR1 to increase the likelihood of that switch. It is interesting to study the effect of that wrong acker switch on both the pgmcc and the competing TCP sessions.

Figure 9 shows the throughput of pgmcc and TCP without the receiver leaving and joining (no acker switching). Figure 10 shows the effect of receiver leaving and joining on the throughput. In that figure the rate of pgmcc increases highly during the periods of wrong acker switching (when PR1 becomes the acker), which is seen in the high slope of the throughput. The low slope represents the periods where the correct acker (PR2) is chosen, which is close to the TCP slope. Although the sender throughput is high, these packets are received by PR1, but PR2 throughput is much less. At the high rate periods the congested link is not able to transfer all packets to PR2 and hence it loses these packets and in a reliable session they have to be retransmitted.

9 CHALLENGES

Several challenges are faced in the design of the framework and the application of our methodology. Some of these challenges are already discussed, such as the reduction of complexity and comparison time mentioned in Section 5. In this section we consider other challenges and how we manage to solve them.

9.1 Error Filtering

One of the problems we faced is the large number of generated error scenarios³. The reason is that, in our approach, all different error states are stored. These error states may represent actually a

³ An *error scenario* is a sequence of states (and events) that ends with an *error state* based on our error model.

small set of distinct error scenarios, but due to the large number of parameters in our state definition, a large number of states represent the same error. Consider as an example out-of-order data packets causing wrong congestion notification. A large number of scenarios with different ordering of data packets and interleaving of the other events are generated, although all these scenarios represent the same problem. Our solution for that is to filter the error states to keep only those scenarios that are really different. This provides a way to remove similar error scenarios and classify detected problems (each error scenario represents a class of problems). We provide two heuristic methods for error filtering, *critical parameters method* and *critical transition method*.

9.1.1 Critical Parameters Method

The idea of this method is choosing a set of parameters depending on the kind of error, and keeping scenarios leading to those error states that are different only on those parameters. The parameters are chosen in such a way to differentiate between error states based on the cause of errors and ignoring parameters that have no effect on the generation of the error, so that their different combinations are not included. A more formal algorithm is provided for automation:

For each error do the following:

- From the error table identify the error condition parameters (the parameters used for detecting the error).
- For each of these parameters, check the transition table for the events that can cause the error condition of the parameter to be satisfied.
- For each error parameter that is affected by more than one event, identify parameters that change due to the transitions caused by these events.
- Use the parameters identified in the above step for classification, and store error states that differ only on those parameters.

Example: Identify the critical parameters of the ‘unnecessary starvation’ error.

- From the error table (Table 2) the condition parameters for unnecessary starvation are *tokens* and *channels*.
- According to the transition table (Table 1), the *tokens* decrease due to the *send* event only and the packets are removed from the *channels* due to the *loss* and *receive* events.
- Since channels can get empty due to different events: *loss* or *receive* (data or ACK), other parameters that change due to these transitions are (see Table 1) the *receive list* and *ACK list*. These two parameters can differentiate between losses, data receive and ACK receive.

By using these two parameters for classification, the number of error scenarios generated using 2 receivers and 3 sequence numbers reduced from 120 to 7.

9.1.2 Critical Transition Method

In this method classification is done based on the transitions that lead to the error. We identify the different *critical* transitions in the detected error scenarios and keep scenarios different only on those transitions. We argue that the critical transitions detected by the following algorithm are adequate for classifying error scenarios.

Starting with a set of error scenarios and valid scenarios:

- For each error scenario find the closest valid scenario and identify the first different transition as the critical transition (by closest we mean the scenario that has the largest number of identical subsequent states starting from the initial state).
- Classify error scenarios according to these error-triggering transitions.

For example in Figure 11 we show a simple error scenario leading to unnecessary starvation and one of its closest valid scenarios. The loss of data is the critical transition here.

Example: Identify the critical transitions of the ‘unnecessary starvation’ error.

Using the set of generated error scenarios and valid scenarios, the critical transitions are the *loss of data*, *loss of ACK*, and *receive of data*. The last critical transition is due to the case when data arrives at the acker while it is not a member, which we call the *acker leave* (or *crash*) problem. This reduces the number of error scenarios generated using 2 receivers and 3 sequence numbers from 120 to 3.

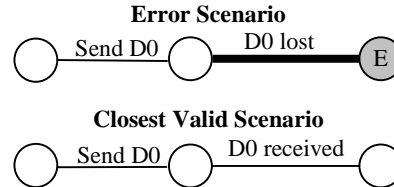


Figure 11: An example of an error scenario and its closest valid scenario. The last state in the error scenario is an error and the thick line is the critical transition

9.1.3 Comparison

Each of these methods has its advantages and drawbacks. The critical transition method is more general and can be applied with different applications similarly. It is also more accurate and compact (in the amount of filtered scenarios). For example, a packet loss causing a certain error will be the same whether it happens at sequence number 5 or at sequence number 50. The disadvantage of this method is the high computational overhead for identifying the closest scenario of each erroneous scenario. The critical parameters method has lower overhead and the identification of the critical parameters requires only the scanning of the transition and errors table and can be done offline. At runtime it needs only to compare error states at those parameters. But critical parameters method is less accurate and generates larger number of error scenarios, especially as the search space gets larger. The number of error scenarios is fixed in critical transition method, since it depends only on the number of transitions. So with higher sequence numbers we have fixed number of error scenarios in critical transition method versus fixed parameters identification time in critical parameters method.

We can combine the advantages of both methods by applying them in sequence, starting with the critical parameters method followed by the critical transition method. This gives lower computational overhead and more compact scenarios.

9.2 Error Hiding

This problem is the inverse of equivalent error scenarios. Here the same error state can be reached by different paths. Consider the case in Figure 12, the error in this case is unnecessary starvation, which can happen due to data loss or acker leaving. Since both scenarios reach the same state and the state is kept only once, the second scenario will not be shown.

Keeping all paths is a very expensive solution, which will increase the memory and comparison time tremendously. Since we are suffering already from a search space explosion problem as was shown by the number of visited states, this option is not considered. Other more applicable solutions are the use of some global estimators to differentiate between such scenarios. For example, by using the global loss estimator the two error states in Figure 12 are not equal, since there is a packet loss in one path and

not in the other. But this solution also can increase the number of states highly, since the states implicitly have some path information. Another simple solution is the use of different combinations of events to trigger different error scenarios. For example in Figure 12, if we disable the packet loss event, errors generated will be due to the leave and join.

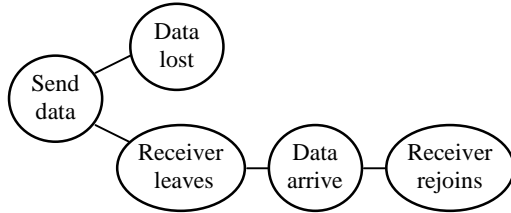


Figure 12: Different paths leading to the same state

10 CONCLUSIONS

We have described a framework for the systematic evaluation of multicast congestion control protocols based on the STRESS methodology. STRESS is extended to cover this type of protocols. The application of congestion control adds new semantics to STRESS, such as the modeling of sequence numbers, channels traffic, and congestion window. New challenges are faced due to the explosive increase in search space, the high level of interleaving between events, and the long-term effects of individual events. These challenges are tackled by introducing new kinds of state and path equivalences. Also some modifications are added to the search mechanism to provide more efficient coverage of the search space. Two algorithms are presented for error filtering to reduce the number of error scenarios generated.

Several problems have been discovered in our case study and a set of scenarios are generated, the generated scenarios can be used for testing multicast congestion control building blocks. Some of the interesting problems identified include the effect of receivers joining and leaving on the throughput measurements, the effect of NACK suppression on feedback, and the effect of having special receivers (as the acker) which can leave or change. Identifying these problems in a systematic way is helpful for the protocol designer and for the applications using that protocol. We also validated our results through detailed packet-level simulations.

The main limitation of our methodology is its high complexity, which constrains the search space covered and limits us to the detection of low-level errors. However these low level errors can lead to high-level performance problems and long-range effects, if not handled properly, as was shown by the simulations. Furthermore the low level errors are more likely to occur in larger topologies and real-life networks. Our future work is to extend the study on the effect of these microscopic problems on performance, using this tool in generating higher-level scenarios and guiding simulations of a larger scale in time and space.

REFERENCES

- [1] S. Begum, M. Sharma, A. Helmy, S. Gupta. "Systematic Testing of Protocol Robustness: Case Studies on Mobile IP and MARS." *Proceedings of the 25th annual IEEE conference on Local Computer Networks (LCN), Florida*, November 2000.
- [2] S. Bhattacharyya, D. Towsley, and J. Kurose. "The Loss Path Multiplicity Problem for Multicast Congestion Control." *Proceedings of the IEEE Infocom'99, New York*, March 1999.
- [3] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. "Advances in Network Simulation." *IEEE Computer*, vol. 33, No. 5, p. 59-67, May 2000.
- [4] A. Chaintreau, F. Baccelli, C. Diot. "Impact of Network Delay Variation on Multicast Session Performance with TCP-like Congestion Control." *Proceedings of the IEEE Infocom 2001, Anchorage, Alaska*, April 2001.
- [5] D. M. Chiu, M. Kadansky, J. Provino, J. Wesley and H. Zhu. "Pruning Algorithms for Multicast Flow Control." *NGC2000, Stanford, California*, November 2000.
- [6] D. DeLucia and K. Obraczka. "Multicast Feedback Suppression using Representatives." *Proceedings of the IEEE Infocom'97, Kobe, Japan*, April 1997.
- [7] C. Diot, W. Dabbous, and J. Crowcroft. "Multipoint Communications: A Survey of Protocols, Functions, and Mechanisms." *IEEE Journal of Selected Areas in Communications*, April 1997.
- [8] S. J. Golestani and K. K. Sabnani "Fundamental Observations on Multicast Congestion Control in the Internet." *Proceedings of the IEEE Infocom'99, New York*, March 1999.
- [9] A. Helmy, D. Estrin, S. Gupta. "Fault-oriented Test Generation for Multicast Routing Protocol Design." *Proceedings of Formal Description Techniques & Protocol Specification, Testing, and Verification (FORTE/PSTV), IFIP, Kluwer Academic Publication, Paris, France*, November 1998.
- [10] A. Helmy, D. Estrin, S. Gupta. "Systematic Testing of Multicast Routing Protocols: Analysis of Forward and Backward Search Techniques." *The 9th International Conference on Computer Communications and Networks (IEEE ICCCN), Las Vegas, Nevada*, October 2000.
- [11] A. Helmy, S. Gupta, D. Estrin, A. Cerpa, Y. Yu. "Systematic Performance Evaluation of Multipoint Protocols." *Proceedings of FORTE/PSTV, IFIP, Kluwer Academic Publication, Pisa, Italy*, October 2000.
- [12] A. Mankin, A. Romanow, S. Bradner, and V. Paxson. "IETF Criteria for Evaluating Reliable Multicast Transport and Application Protocols." *RFC 2357*, June 1998.
- [13] K. Obraczka. "Multicast Transport Mechanisms: A Survey and Taxonomy." *IEEE Communications Magazine*, January 1998.
- [14] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. "Modeling TCP Throughput: A Simple Model and its Empirical Validation." *ACM SIGCOMM 1998, Vancouver, BC, Canada*, September 1998.
- [15] L. Rizzo. "pgmcc: A TCP-friendly Single-Rate Multicast Congestion Control Scheme." *ACM SIGCOMM 2000, Stockholm, Sweden*, August 2000.
- [16] L. Rizzo, G. Iannaccone, L. Vicisano, and M. Handley. "PGMCC Single Rate Multicast Congestion Control: Protocol Specification." *Internet-Draft, draft-ietf-rmt-bb-pgmcc-00.txt*, 23 February 2001.
- [17] K. Seada, A. Helmy. "Fairness Evaluation Experiments for Multicast Congestion Control Protocols." *Technical Report 02-757, University of Southern California, CS Department*, April 2001.
- [18] T. Speakman, D. Farinacci, J. Crowcroft, J. Gemmell, S. Lin, A. Tweedly, D. Leshchiner, M. Luby, N. Bhaskar, R. Edmonstone, K. M. Johnson, T. Montgomery, L. Rizzo, R. Sumanasekera, and L. Vicisano. "PGM Reliable Transport Protocol Specification." *Internet-Draft, draft-speakman-pgm-spec-05.txt*, 24 November 2000.
- [19] B. Whetten, L. Vicisano, R. Kermod, M. Handley, S. Floyd, M. Luby. "Reliable Multicast Transport Building Blocks for One-to-Many Bulk-Data Transfer." *RFC 3048*, January 2001.
- [20] J. Widmer, R. Denda, and M. Mauve. "A Survey on TCP-Friendly Congestion Control." *Special Issue of the IEEE Network Magazine*, May 2001.
- [21] J. Widmer, M. Handley. "Extending Equation-based Congestion Control to Multicast Applications." *ACM SIGCOMM 2001, San Diego, California*, August 2001.