RSVP Killer Reservations

January 25, 1999

Abstract

This document describes the Killer Reservation Problem encountered when merging RSVP reservation requests. These requests get merged as they travel up the multicast distribution tree, losing information about individual requests. A request, which would have succeeded on its own, may suffer denial of service when the "merged request" fails admission control. This is the problem for which we present different solutions.
1. Introduction

This document describes the Killer Reservation problem in RSVP. RSVP [3] is a "Resource ReSerVation Protocol" used by a receiver of a multicast group to initiate a reservation request for particular data flows. This request is forwarded upstream by RSVP, to all nodes along the path(s) of the flow(s), and gets merged with requests from other receivers along the way. This receiver oriented approach helps accommodate heterogeneous receiver requirements.

Merging of these heterogeneous requests may lead to a denial of service problem. Specifically, a receiver persisting in making a request, which fails admission control at a node, may end up denying service to other requests that were merged with it downstream. This is the Killer Reservation problem for which we present various solutions.

Two such killer reservation problems have been identified [1]

(1) The first killer reservation problem, KR-I, arises when there is a reservation Q0 already in place. If a receiver now makes a request Q1 > Q0, the merged request may fail admission control upstream, consequently denying reservation to the receiver requesting Q0.

(i) A            B                    A            B
    +-----+      +-----+              +-----+      +-----+
    |     |<--Q1 |   Q0|------ R0     |     |      |   Q0|------ R0
    |   Q0|------|     |              |    *|------|     |
    |     |      |   Q1|------ R1     |     | Q1-->|   Q1|------ R1
    +-----+      +-----+              +-----+      +-----+

Figure 1: Killer Reservation I

Figure 1 illustrates the KR-I scenario. Receiver R0’s reservation request (Q0) is successfully established at both nodes A and B. R1 then makes a request for Q1 (Q1 > Q0). The two requests get merged at node B and the merged request, for Q1, is forwarded upstream (right to left) to A, where it fails. This tears down the existing reservation (Q0) at node A, denying service to R0. A ResvErr message is sent downstream (left to right) to node B, with Q1 as the flowspec in error.

(2) In the second killer reservation problem, KR-II, the sequence is reversed. There is a receiver persistent in making a request Q1, in spite of it failing admission control. A subsequent, smaller request (Q0) will be denied, if merged with Q1, even though it might otherwise have succeeded.
Figure 2: Killer Reservation II

Figure 2 shows the KR-II scenario in which receiver R1 continues to refresh its reservation request (Q1), even though it is failing at node A. With every failure, node A generates a ResvErr message that is sent downstream to node B as depicted in Figure 2 (i). Receiver R0 then tries to reserve Q0 (Q0 < Q1). However, the merge at node B continues to send the greater flowspec (Q1) in the Resv message and prevents Q0 from getting established at node A.

Some provision in the protocol is necessary to avoid such accidental or deliberate denial of service.

Note: Throughout this document we assume flowspecs to be scalar, i.e. one-dimensional in some base unit. Scalar flowspecs can be ordered and the merge of a set of scalar flowspecs is defined as the MAX or the largest element in the set. A failing "merged flowspec" identifies those flowspecs that will fail too, these being flowspecs greater than or equal to the failed flowspec.

On the other hand, multi-dimensional flowspecs may not have any ordering defined for them. The merge of a set of multi-dimensional flowspecs is a flowspec which is at least as large as each flowspec being merged, i.e. the Least Upper Bound or LUB. A failed "merged flowspec" gives no indication of the failure of any individual flowspec. As an example, consider a couple of 2-dimensional flowspecs, A = (10, 1) and B = (1, 10). In general one may not be able to say whether A > B or vice versa. The merge corresponds to the flowspec (10, 10) and its failure does not indicate which one of A or B will fail. In fact, it might be the case that both A and B individually succeed but their merge fails.

Multidimensional flowspecs are not dealt with in this study and are a topic of further research.

1.1. KR-I versus KR-II

It should be noted that the two killer reservation problems are not independent but closely related. We now draw analogies between the
A solution to KR-II needs to make the smaller reservation request (Q0) visible in the presence of a large, failing reservation request (Q1). This would allow Q0 to get established. For example, node A in Figure 2 needs to be informed of a downstream request for Q0 or it would not be able to make this reservation.

A solution to KR-I would allow Q0 to remain at node A in Figure 1. However, node A should be able to tear down Q0 when the receiver R0 stops refreshing its reservation request. Hence, simply leaving any existing reservation in place upon admission control failure is not sufficient to correct the KR-I problem. If R1 is persistent, the merge will always result in Q1, the larger reservation request. Node A will not be informed when the smaller reservation request Q0 is torn down, denying it the option of clearing the reservation. Hence, Q0 must be made visible despite the presence of Q1, as in the KR-II problem.

Moreover, once a solution to KR-II leads to the establishment of the smaller reservation request (Q0) at the node where the larger reservation request (Q1) was previously failing admission control, we encounter the same scenario KR-I deals with. There will be a smaller reservation (Q0) in place and a receiver attempting a larger reservation request (Q1), which would fail and tear down Q0 if not prevented from doing so. For example, in the KR-II scenario if Q0 is somehow established at node A, Figure 1 (i) and Figure 2 (ii) will become identical.

Hence the possible solutions we present do not lay any stress on the distinction between the two, but present an overall solution for both.

The rest of this document is organized as follows. Section 2 discusses various policies which can be used to solve the KR problem. Section 3 mentions the additional cost of implementing these policies and factors which influence the choice of one policy over another. The remaining sections outline mechanisms for each of the 3 policies presented.

2. Policies for Dealing with Admission Control Failure

The current RSVP protocol does include some provisions for dealing with the KR problem (Section 2.5 [1], 3.5 [1]). However, we first describe how the base protocol (one without these additional mechanisms) behaves when a reservation request fails admission control.
As illustrated in Figure 1, admission control failure at a node tears down the existing reservation and generates a ResvErr message that is sent downstream. The error message is forwarded downstream to nodes along the path to "culprit" receivers. It does not alter the reservation state at these nodes, create additional state, or trigger new Resv messages. This base mechanism maintains the MAX (merge) of all reservation requests, from the point of merge to the point of failure. As already mentioned, this leads to the undesirable consequence of denying smaller requests which could have succeeded at the failure point and beyond.

To show what we can alternately achieve, we imagine an ideal environment where information about distinct reservation requests is not lost and a node has knowledge of the reservation state at all other nodes. Given this, we would have sufficient knowledge to decide which reservation request to attempt after admission control failure. This section outlines different policies on which this decision could be based.

To elucidate the differences between these policies, we consider the topology shown in Figure 3. There are 3 receivers R1, R2 and R3, making reservation requests for 2, 4 and 8 units respectively. Node A, B and C can support reservations for at most 8, 2 and 7 units respectively on their outgoing links. Node D has no such restriction.

2.1. Global Best Fit Policy

The Global Best Fit Policy corresponds to establishing and maintaining the single greatest reservation request that can succeed along the entire path(s) of the flow(s). For a WF or SE reservation style, this request has to succeed at all nodes lying on paths leading to the sources whose flows share the reservation.

All reservation requests that are greater than the Best-Fit-Reservation will fail at some intermediate node. If such a request gets merged with a smaller reservation request before reaching its point of failure, the merge forwards the smaller request. Hence, at a merge point that does not have any request that will succeed along
the entire path, the merge takes a MIN of all the requests.

Using this policy, we get the same reservation state along the entire path of the Best-Fit-Reservation. Requests lower than the Best-Fit-Reservation are provided the requested service at all nodes from the point they merge with the Best-Fit-Reservation. Higher requests are not granted their request at any node beyond their merge point with the Best-Fit-Reservation.

This policy would lead to the reservation state shown in figure 4.

A(8) B(2) C(7) D(*) ---- R1 (2)
+-----+ +-----+ +-----+ +-----+ |
| 2 | 2 | 2 | 4 | 2 |
| 8 | 4 | 8 | 8 | 8 |

Figure 4: Global Best Fit

2.2. Local Best Fit Policy

The Local Best Fit Policy supports partially successful reservation requests. At each node we try to establish the greatest of all downstream requests. If this is rejected we attempt the next greatest request, and so on, till either one downstream request succeeds or all of them fail admission control. The decision of which reservation request to attempt next is NOT influenced by whether that reservation request succeeds at other nodes and the reservation state might vary along the path.

Hence, even if a particular request does not succeed along the entire path, it will be installed in those nodes that can provide the requested quality of service. The receiver is provided the service it requested at as many nodes as possible.

With this policy, we achieve the reservation state shown in Figure 5.

A(8) B(2) C(7) D(*) ---- R1 (2)
+-----+ +-----+ +-----+ +-----+ |
| 8 | 2 | 4 | 4 | 2 |
| 8 | 4 | 8 | 8 |

Figure 5: Local Best Fit
2.3. Greedy Policies

The Greedy Policy has two variants. One (which we call Plain Greedy) first attempts the greatest downstream request at each node. Failing this, it installs the best reservation the node can provide, without regards for the smaller, remaining requests. A receiver is assured that even if its request fails, the network provides it the best possible service. As in the Local Best Fit Policy, the reservation state along the path might be different at different nodes. The reservation state resulting out of this policy, for the configuration in Figure 3, is shown in Figure 6.

![Figure 6: Plain Greedy](image)

The other variant installs Non-Increasing Reservation Amounts upstream from the merge point. The first attempt at a node is to install the greatest of all reservations, in place at its immediate downstream neighbor(s), i.e. only the reservation state at nodes one RSVP hop away are considered. As before, if this attempt fails, it installs the best reservation the node can provide. Following this policy the reservation state in Figure 6 would change to that shown in Figure 7.

![Figure 7: Greedy (Non-Increasing Reservation Amounts)](image)

3. Solution Cost

There are scaling problems with a mechanism requiring knowledge of every downstream request as assumed when introducing the different policies in the previous section. Hence we propose different mechanisms that can support these policies without needing complete information in each Resv message.
The schemes for the Global and Local Best Fit Policies rely on a transient period, during which Resv and ResvErr messages travel back and forth adjusting the reservation state at the nodes. These iterations involve ResvErr messages generating fresh Resv requests, which carry new information. Before the reservation state at a node stabilizes, more than a few of these iterations may be required, the worst case requiring as many iterations as the number of distinct, failing reservation requests. Hence our mechanisms generate more Resv and ResvErr messages. Besides this dynamic overhead, there is also a static overhead of additional state at a node that summarizes the failed requests.

Note that we tradeoff the number of control (ResvErr and Resv) messages and delay in the establishment of the desired reservation with the state maintained at each node for all downstream requests and the size of each control (Resv) message. However, it should be realized that these mechanisms come into play only in the event of admission control failure.

The following sections discuss mechanisms for each of the 3 policies mentioned. The choice of a particular policy should be governed by the complexity of the associated mechanism as well as how a network wishes to manage failed requests. The Global Best Fit policy is preferable when a network desires to reserve resources only for those requests that succeed end to end. Requests, which fail at some intermediate node(s), do not use up resources at any node (however see Section 2.1). For such requests, the Local Best Fit Policy reserves resources at those nodes where they did succeed. The Greedy policy is the most flexible, installing them at nodes where they succeed and reserving as much as possible at nodes where they fail.

4. Blockade State Mechanism (Global Best Fit Policy)

The Blockade State Solution to the Killer Reservation problem, also included in the RSVP document [1], was proposed by Lixia Zhang and Steve Berson [4] [5]. This mechanism implements the Global Best Fit Policy (Section 2.1), hence it maintains the single greatest reservation request, successful along the entire path. To this end it introduces additional state at each node, called "blockade state".

4.1. Algorithm

When admission control fails for a reservation request, any existing reservation is left in place. This prevents a failing request from tearing down established ones. This, however, is not sufficient to eliminate the KR-I problem when the failing request is persistent. If we choose not to refresh the existing reservation with the failing request, the established reservation state will eventually timeout.
If we do refresh the existing reservation, we would be left with a stray reservation when its request is torn down by the receiver. We hence choose not to refresh the established reservation state and demonstrate how the following mechanism ensures it does not timeout.

A ResvErr message is sent downstream, to all receivers making offending requests, and it creates or refreshes additional state at each node that it traverses. This is called the blockade state and contains the flowspec from the ResvErr (Qe). An offending (blockaded) request is one with a requested flowspec greater than or equal to the blockade flowspec.

Subsequent Resv refreshes (triggered or periodic) use the blockade state to ignore the offending requests when doing a merge. This makes smaller requests visible upstream allowing them to be established or maintained.

The blockade state is periodically timed out allowing probes for previously failed requests upstream. These might now succeed in the event of greater resource availability. The blockade state timeout interval (Tb) would typically be a multiple of the refresh interval, the suggested default for which is 30 seconds [1].

In case all downstream requests are blockaded the merge takes the MIN of all requests, maintaining it till the failure point. A MIN is more useful than a MAX as it prevents the failed reservations from timing out, only to be restored every time the blockade state expires.

This policy is useful in providing users with the desired QoS along as much of the path as possible and in overcoming transient failures such as route "flaps". It may also give rise to certain pathological cases such as a large reservation request being denied at the last hop but still consuming resources at all nodes downstream.

Finally, if desired, the failed request can also be forwarded upstream from the failure point with a flag to prevent generation of additional error messages. When all requests fail, leap-frogging the failure point in this fashion maintains the MIN at as many nodes as possible.

4.2. Example

To clarify the procedure we consider the example presented in Figure 8. In the figure, the reservation state is labeled next to the downstream interface while the blockade state for the previous hop is shown on the upper left corner.
The two receivers, R0 and R1, make reservation requests for 4 and 8 units respectively. These get merged at node B and a Resv message for 8 units is sent upstream to A. This request fails, generating a ResvErr message (Qe = 8) that is sent downstream to B and forwarded to R1 (Figure 8 (i)).

The ResvErr creates a blockade state (Qb = 8) at node B for the associated previous hop (A). Since the reservation state for R0 (4) is not blockaded, an immediate refresh is triggered. The blockade state causes R1’s reservation state (8) to be ignored, resulting in a fresh Resv for 4 units sent upstream (Figure 8 (ii)).

A detailed sequence of events resulting in a stable reservation state at each node under this algorithm is included in Appendix A.1.

### 4.3. Overhead

We now discuss the overhead and limitations of the blockade state mechanisms. Although this mechanism adds little complexity to the base protocol’s message processing rules (Section 2), it does require additional (blockade) state maintained at each node. Each element of the blockade state consists of a blockade flowspec (Qb) and an associated timer (Tb). Depending on the reservation style there may be a blockade state element per previous hop (wildcard style) or per sender (explicit style).

The control traffic overhead also increases, albeit only in the event of admission control failure. ResvErr messages are sent downstream that cause merging of non blockaded requests and immediate Resv refreshes.

Every Tb there is a fluctuation in the reservation state with the blockade state timing out and requests that had thus far been ignored, being merged in a subsequent Resv refresh and sent upstream. These are established, overriding existing reservations, till the point they are rejected or blockaded again.

Finally in the case when all requests fail and their MIN is sent
upstream, persistent ResvErr messages are generated. Every refresh period the node sees the rejected Resv message and generates a ResvErr in response. The Resv message constantly probes this node for resource availability and the ResvErr keeps refreshing the blockade state downstream.

5. Class Partitioning Mechanism (Local Best Fit Policy)

The Class Partitioning Solution requires partitioning of the reservation requests into two classes, those that have experienced failure (F) and those that have not (S). It is based on an approach suggested by Lee Breslau, Shai Herzog and Bruce Davie [6] and implements the policy of maintaining partially successful reservation requests, i.e. the Local Best Fit Policy (Section 2.2).

5.1. Algorithm

The two classes of reservation requests are merged independently at each node and forwarded upstream in a Resv message. This request is also tagged with a flag that indicates whether a ResvErr should be generated if this request fails admission control upstream.

The resulting tuple carried in a Resv is (Qs, Qf, NoErrorReport) where

1. Qs is the MAX of the flowspecs in the S class
2. Qf is the MAX of the flowspecs in the F class
3. NoErrorReport is the flag set when all merged requests have this flag set. i.e. Merging of the NoErrorReport flag uses AND.

For each downstream neighbor making a request, the node keeps track of this tuple. When a reservation request is first made it is put in the S class and hence corresponds to (Qr, NULL, OFF) where Qr is the requested flowspec.

On receiving a Resv message an attempt is first made to install the MAX of Qs and Qf. If successful, this establishes the highest requested service at this node since all requests are in either one of these two classes.

If this fails we try to install or refresh the MAX of Qc (the existing reservation) and Qs. This merge corresponds to the highest downstream request not to fail at this node.

If both these attempts fail, the following procedure is followed:
If the NoErrorReport flag is not set (OFF) in the Resv message, a ResvErr message is sent downstream with Qs as the offending flowspec.

The currently installed reservation is kept in place, preventing a failing request from wiping out installed service.

The NoErrorReport flag in the saved state for this downstream neighbor is set to ON to reflect the fact that a ResvErr message has already been generated. This failed request is forwarded upstream, merged with other requests at this node.

When a ResvErr message is received at a node, it identifies the offending members of the S class, just as the blockade state identifies blockaded reservation requests. These offending reservations are then moved to the failure (F) class. The ResvErr is forwarded along those branches carrying culprit requests, with the NoErrorReport flag turned OFF. The S class merge for a subsequent Resv message (Triggered or Periodic) would hence make the next highest request visible upstream.

ResvErr messages cease either when a reservation request is successfully installed at each node or when all requests fail. In the former case Qs corresponds to the greatest request to succeed along the entire path(s) of the flow(s). In the latter case Qs is NULL, which obviously succeeds at each node.

The failure (F) class is periodically timed out and its flowspecs moved to the S class. This helps in probing upstream for reservation requests that had thus far been in the F class.

While more powerful, this policy of maintaining partially successful reservation requests may tie up resources for a large request at most of the nodes along the path of the flow. Still, this request may be provided absolutely no guarantees at those nodes where it fails.

5.2. Example

To illustrate this algorithm we consider the same configuration as in Figure 8. As before, the reservation state is labeled next to the downstream interface. The tuple (Qs, Qf, NoErrorReport) for the downstream nodes is shown on the left of the reservation state. (Qf,Qs) represents the tuple (Qs, Qf, OFF) and (Qs’Qf) represents the tuple (Qs, Qf, ON). Figure 9 shows the example.
Initially when R0 and R1 make their requests, the flowspecs are put into the S class with the NoErrorReport flag OFF. This corresponds to (4,0) and (8,0) respectively. These get merged at node B and the tuple (8,0) is sent upstream to A in a Resv message. This request fails, generating a ResvErr message (Qe = 8) that is sent downstream to B and forwarded to R1 (Figure 9 (i)). Node A also forwards the failed request upstream (not shown here) but with the NoErrorReport flag turned off.

The ResvErr moves the offending flowspec (8) to the F class. Since the merge at node B results in a non null flowspec for the S class (4,8), an immediate refresh is triggered, resulting in a Resv for (4,8) sent upstream (Figure 9 (ii)).

Appendix A.2. shows the sequence of events that result in a stable reservation state at each node using class partitioning.

5.3. Overhead

The Class Partitioning mechanism places additional burden on the base protocol too. Not only is it more complex than the blockade state approach, it also requires a change in the format of Resv messages. These must now carry two flowspecs instead of one. Each node needs to keep state for each downstream neighbor making a request in the form of the tuple (Qr, Qf, NoErrorReport) and a timer for the failure class, Tc. This timeout interval should be of the same order as the blockade state timeout interval.

The control traffic overhead is about the same as the Blockade State mechanism, except that

(1) More Resv messages are generated since failed requests are forwarded upstream

(2) There are fewer ResvErr messages since no persistent error messages are generated when all requests fail.

Note that additional control traffic is incurred only upon admission
control failure.

Failure class timeouts do not cause fluctuations in the reservation state at a node. All reservations less than or equal to the highest request are maintained i.e. an existing reservation (Qc) is overwritten only when it lies beyond MAX(Qs, Qf), which implies that some reserving node has gone away.

However, by maintaining reservations in this way we run the risk of keeping stray reservations in place. When a reserving node goes away, its reservation request will still be kept in place if it’s less than MAX(Qs, Qf). Since the F class is periodically timed out these intermediate requests become visible at a node once every Tc. An additional timer can help, by deleting them if they are not detected within some number of failure class timeout periods. Whether this additional complexity and state is worth the effort would depend on whether such stray reservations can be tolerated.

6. Greedy Solutions (Greedy Policies)

Finally we consider the Greedy Solutions. These implement the Greedy Policies (Section 2.3) of compensating a failed reservation request with the best service the network can provide. Even though a large reservation request still overshadows those it gets merged with, by not rejecting it entirely we do not deny service to the smaller ones. The Killer Reservation problem is successfully avoided in this way.

6.1. Algorithm

A node first attempts to establish the merged request it sees. If there are sufficient resources to guarantee this request, it succeeds and the protocol works as usual. The reservation installed (Qc) is the same as the reservation requested (Qr).

If the request fails, the greedy policy dictates that the node install the greatest reservation it can support. Hence it determines the maximum Q, less than Qr, which can be supported and makes that reservation locally. Qc will be less than Qr in this case.

In either case a request is forwarded upstream. The flowspec sent in this request depends on the variant of the Greedy Policy being supported. For the Plain Greedy policy, the flowspec sent upstream is the same as the service requested (Qr). For Greedy with Non-Increasing Reservation Amounts, the installed service (Qc) is forwarded. Hence while the former requires state for both Qc and Qr, the latter needs state only for the level of service installed, Qc.

Generating a ResvErr message at each node that could not provide the
service requested, would create multiple and persistent ResvErr messages whenever there exists any failing request. To prevent this anomaly, we do not generate ResvErr messages upon admission control failure. Instead we advertise the installed service in an advertisement message or in the Path message sent downstream from source. These report the minimum level of service guaranteed along the entire path. It requires taking the MIN of the flowspec in the incoming advertisements and the service installed on the link.

This mechanism can be used to provide feedback to receivers whose request did not succeed along the entire path of the flow. Based on this feedback, they can decide whether to continue with their reservation or not.

This policy might generate serious concerns because it does not prevent a naive or malicious receiver to make a large request that can block all of the available resources along the entire path of the request.

6.2. Overhead

Although processing of Resv and ResvErr messages is simpler, the Greedy Mechanism depends on advertisement of installed service, additional state and extra control traffic. State needs to be maintained not only for the reserved (Qc) service (as in the base protocol) but also for that requested (Qr). The frequency with which the advertisements are generated decide the extra control traffic burden.

7. Summary

The Killer Reservation problem arises out of the desire to support heterogeneous reservation requests and the need to merge these flowspecs to scale to large multicast groups. While removing the heterogeneity support would define away the problem, we need some provision in the existing protocol to accommodate heterogeneity, but avoid the denial of service problem.

Various alternative policies are discussed on which solutions to the Killer Reservation Problem can be based. Mechanisms are described that can be used to implement these and their overheads discussed. In choosing one policy over another, one might consider not only what the policy achieves but also the cost of the associated mechanism.

8. Acknowledgements

The author would like to acknowledge members of the RSVP research group (USC/ISI) and Lee Breslau (Xerox PARC) for their valuable
comments. Special appreciation goes out to Steve Berson (USC/ISI) for his detailed review of the draft. Bob Braden (USC/ISI) provided the original motivation, and helped with extensive discussions and reviews every step of the way, resulting in increased clarity and fewer bugs.

9. References


[7] Breslau, L., "Options For Dealing With Admission Control Failures", RSVP WG 7/19/95
APPENDIX. Message Sequence Leading to Stable State

This appendix shows the sequence of messages leading to a stable reservation state for the blockade state and class partitioning algorithms. The sample configuration shown in Figure 3 is used. An awk based simulator has been designed for this purpose and can be downloaded from http://www-scf.usc.edu/~mtalwar/projects/rsvp.

As before, in these examples all flowspecs are one dimensional. The reservation state is labeled next to the downstream interface. Arrows to the left (upstream) represent reservation messages (Resv) and those in the opposite direction (downstream) denote reservation errors (ResvErr). The reservation amount that a node can support is tagged in parenthesis next to the node name.

A.1. Blockade State

The blockade state for the previous hop is shown on the upper left corner of the node.

```
A(8)         B(2)         C(7)         D(*)    +------ R1
+-----+      +-----+      +-----+      +-----+   |
|     |      |     |      |     |<-- 8   8|---+  
|     |------|     |------|     |---+     |---+  
|     |      |     |      |     |---+     |---+  
+-----+      +-----+      +-----+      +-----+   |
+------ R3
```

R1 sends a Resv for 8 units that succeeds at D and is forwarded.

```
A(8)         B(2)         C(7)         D(*)    +------ R1
+-----+      +-----+      +-----+      +-----+   |
|     |      |     |      |     |      | 8   8|---+  
|     |------|     |------|     |------|     |---+  
|     |      |     |      |     |---+     |---+  
+-----+      +-----+      +-----+      +-----+   |
+------ R3
```

The Resv fails at C generating a ResvErr.

```
A(8)         B(2)         C(7)         D(*)    +------ R1
+-----+      +-----+      +-----+      +-----+   |
|     |      |     |      |     |      | 8   8|---+  
|     |------|     |------|     |------|     |---+  
|     |      |     |      |     |---+     |---+  
+-----+      +-----+      +-----+      +-----+   |
+------ R3
```
The ResvErr cause a blockade state to be setup in D with Qb = 8 and
is forwarded to R1, the responsible receiver.

Now R2 sends a Resv for 4 units that succeeds at D and triggers a
refresh. The merge ignores R1’s request (8) and results in a Resv
for 4 units being forwarded upstream.

The Resv succeeds at C and is forwarded.

The Resv fails at B generating a ResvErr that is sent downstream.

The ResvErr sets up blockade state in C with Qb = 4 and is sent
downstream to D.
The ResvErr alters the blockade flowspec in D to 4 and is forwarded to both R1 and R2.

R2 generates a Resv for 2 units that succeeds at D. The change in reservation state triggers a refresh. The blockade state causes the merge to ignore requests for 8 and 4. This results in a Resv for 2 units being forwarded upstream.

The Resv causes the reservation state in C to change to 2.
The Resv for 2 units is forwarded upstream to A and B, and is installed in both these nodes. This is a stable state and does not alter till either the blockade state times out or some receiver changes its request.

A.2. Class Partitioning

The tuple $(Q_s, Q_f, \text{NoErrorReport})$, for the downstream nodes making a reservation, is labeled to the left of the reservation state. $(Q_f, Q_s)$ represents the tuple $(Q_s, Q_f, \text{OFF})$ and $(Q_s'Q_f)$ represents the tuple $(Q_s, Q_f, \text{ON})$.

R1 sends a Resv for 8 units that succeeds at D and is forwarded.

The Resv fails at C generating a ResvErr. The failed Resv is forwarded upstream. The NoErrorReport flag is turned ON for the forwarded request and the saved tuple.
The ResvErr is forwarded to R1 and causes R1’s request (8) to be moved to the F class in D. The Resv request continues to travel upstream to A, after it fails at B too. Note that there is no ResvErr generated at B since the NoErrorReport flag is ON.

Now R2 sends a Resv for 4 units that succeeds at D and triggers a refresh. The S and F classes are merged independently, resulting in the Resv (4,8) being forwarded upstream. R1’s request has in the meanwhile succeeded at A and there exists a reservation for 8 units at that node.

The Resv (4,8) succeeds at C, reserving 4 units and is forwarded. The saved tuple at node C is altered to (4,8).

The Resv fails at B generating a ResvErr that is sent downstream.
The failed request is forwarded to A with the NoErrorReport flag ON.

A(8)         B(2)         C(7)         D(*)    +------ R1
+-----+      +-----+      +-----+      +-----+   |
|4'8  |      |4'8  |      |0,8  |      |0,8 8|---+
| 8    |------|     |------| 4    |------|4,0 4|---------- R2
+------+      +------+      +------+      +------+   |

A(8)         B(2)         C(7)         D(*)    +------ R1
+-----+      +-----+      +-----+      +-----+   |
|4'8  |      |4'8  |      |0,8  |      |0,8 8|---+
| 8    |------|     |------| 4    |------|0,4 4|---------- R2
|      |      |     |      |      |      |2,0 2|---+
+------+      +------+      +------+      +------+   |

The ResvErr moves the offending S class member (4) to the F class, altering the saved tuple at C from (4,8) to (0,8). The ResvErr is then sent downstream to D. Meanwhile, the Resv message refreshes the reservation at A (8).

A(8)         B(2)         C(7)         D(*)    +------ R1
+-----+      +-----+      +-----+      +-----+   |
|4'8  |      |4'8  |      |0,8  |      |0,8 8|---+
| 8    |------|     |------| 4    |------|0,4 4|---------- R2
|      |      |     |      |      |      |2,0 2|---+
+------+      +------+      +------+      +------+   |

The ResvErr moves R2’s request (4) into the F class and is forwarded to R2. Since R1’s request is already in the F class, it does not receive a ResvErr.

A(8)         B(2)         C(7)         D(*)    +------ R1
+-----+      +-----+      +-----+      +-----+   |
|4'8  |      |4'8  |      |0,8  |      |0,8 8|---+
| 8    |------|     |------| 4    |------|0,4 4|---------- R2
|      |      |     |      |      |      |2,0 2|---+
+------+      +------+      +------+      +------+   |

R3 generates a Resv for 2 units that succeeds at D. The change in reservation state triggers a refresh. The merge results in a Resv (2,8) being forwarded upstream.

A(8)         B(2)         C(7)         D(*)    +------ R1
+-----+      +-----+      +-----+      +-----+   |
|4'8  |      |4'8 <=2,8 |2,8 |      |0,8 8|---+
| 8    |------|     |------| 4    |------|0,4 4|---------- R2
|      |      |     |      |      |      |2,0 2|---+
+------+      +------+      +------+      +------+   |

The Resv updates the tuple in C to (2,8) and is forwarded to B. The
reservation at C (4) is refreshed.

```
<p>| A(8) | B(2) | C(7) | D(*) |
|-------+------|------|------|
| 4'8   &lt;2,8 | 2,8 | 2,8  | 0,8 8 |</p>
<table>
<thead>
<tr>
<th>8</th>
<th>2</th>
<th>4</th>
<th>0,4 4</th>
</tr>
</thead>
</table>
```

A reservation for 2 units is installed at B, the tuple updated and the request forwarded.

```
<p>| A(8) | B(2) | C(7) | D(*) |
|-------+------|------|------|
| 2,8   | 2,8  | 2,8  | 0,8 8 |</p>
<table>
<thead>
<tr>
<th>8</th>
<th>2</th>
<th>4</th>
<th>0,4 4</th>
</tr>
</thead>
</table>
```

The Resv (2,8) updates the saved tuple and refreshes the installed reservation (8) at A. We now have a stable state that does not alter till some receiver changes its request.