Off-Piste[1] QoS-aware Routing Protocol

Tal Anker[2] Danny Dolev Yigal Eliaspur
The School of Engineering and Computer Science
The Hebrew University of Jerusalem, Israel
{anker, dolev, eliaspur}@cs.huji.ac.il

Abstract

QoS-Aware routing protocols have been the focus of much attention in the past decade. This is due to the interesting challenges posed by the problem of QoS routing, in the absence of precise information - or even partial and altogether imprecise information. In this paper, we present a new QoS-aware routing protocol, OPsAR, which achieves very good performance results at a very reasonable cost, in terms of memory and messages.

OPsAR is based on the Multi-path routing approach, but constrains the number of paths used and leverages on previous resource reservation attempts. Every reservation attempt – successful or not – results in an update to the knowledge-state recorded at all the nodes that had participated in those attempts. This knowledge helps in avoiding network bottlenecks and coping with congested path when they are encountered. We present extensive simulation results and even compare ourselves favorably with a protocol that explores all possible paths.

1. Introduction

This paper demonstrates the benefits of caching, when combined with an adaptive protocol, in reducing the blocking probability of QoS requests. Specifically, we show that keeping track of recent QoS queries drastically improves the quality of the QoS routing, and optimizes communication resources. The information collected while executing our protocol can be used by more sophisticated learning algorithms to dynamically change the protocol's parameters.

Much of the QoS routing research deals with multicast routing. There are benefits to using unicast QoS, and applications using unicast QoS routing have lately emerged (e.g., point to point VPN services). The protocol in this paper is presented in the context of unicast QoS routing, although it could also be applied to multicast routing as well.

The main challenge in QoS routing is to be able to respond to online requests with minimal overhead, while minimizing probability of blocking (failures). A typical request is to reserve a certain resource, typically bandwidth, along a path from a transaction source to a transaction destination[3]. The dynamic nature of the system, in terms of availability of resources, network topology, etc., and the inability to maintain the global state leads to use of adaptive protocols.

Typically, QoS routing protocols adhere to shortest path algorithms, deviating from them only when these fail to offer the requested resources. In that case, full or limited flooding will be employed. This paper presents the Off-Piste qoS-Aware Routing protocol (*OPsAR*). In OPsAR, a node keeps track of recent QoS requests that reach it to learn about resource availability to and from various target points. The learning is reflected in the node's "knowledge-state".

The protocol presented contains two phases, a *Try_phase* and a *Scan_phase*. The Try_phase follows the shortest path as long as it has the required resources. The deviation from the shortest path takes an "off-piste" route that leverages on the knowledge-state to optimize the routing protocol. The protocol reserves the resources along the path toward the transaction target during the Try_phase. Its deviation from the shortest path is bounded. If resources cannot be reserved within that boundary, the resources which have already been reserved are released, and a request is sent to begin the Scan_phase, a limited scanning from the transaction target toward the transaction source (in the reverse direction). The scanning process takes advantage of the knowledge-state to optimize the search. The scan process is based on limited Breadth-First-Search (BFS), which is bounded using a combination of the knowledge-state and different techniques and ideas borrowed from other related works, as specified later.

Our protocol is influenced by QMRP [1] and S-

1 Meaning - off the trail, especially a ski trail.

2 Anker is also with Radlan Computer Communications, Israel.

3 The application data source can be on either side.

Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP'04)
1092-1648/04 $ 20.00 IEEE

QMRP [2]. In [2], when the shortest path fails, the transaction target uses the preexisting multicast tree and deviates from it when it fails to offer the requested resources. We use that deviation concept in both phases of our protocol and gain a factor in efficiency. The scanning method resembles the ideas of [1] and [2]. In [1], a multiple-path routing is invoked once the shortest path fails to allocate resources. A similar approach is used in the scanning process of our protocol, while limiting the blowout of the scanning process, as proposed in [3].

The resources requested by a QoS request can be divided into additive resources (e.g. delay) and nonadditive resources (e.g. bandwidth). The protocol and the simulations in this paper specifically deal with bandwidth reservation, though other nonadditive resource reservation can be handled in a similar way. The protocol can be adapted to handle additive resources, as we indicate later.

The OPsAR protocol presented in the paper meets the following design goals:

- *Efficiency:* OPsAR increases the chances of finding a QoS route while sending fewer messages. It leverages on its knowledge-state, which is updated by previous successful **and** failed attempts to reserve resources.
- *QoS Awareness:* The protocol is aware of availability and non-availability of QoS resource. The protocol uses a decay method to limit the effect of past information that may not be relevant any longer.
- *Scalability and responsiveness:* The protocol maintains a limited information base at each node. It requires a bounded overhead, and it does not flood the network. It responds to requests without wasting resources that are not explicitly requested.
- *Operability:* OPsAR makes use of existing routing protocols. It prefers the default shortest path route when it has available resources. It does not use any global network state but rather use a local state and accumulated information regarding partial QoS resource availability toward some destinations, when such information is available.
- *Adaptivity:* The protocol adapts its routing preferences as a function of ongoing requests, while leveraging on its knowledge-state. Over time, it intelligently covers more and more possible routes, thus increasing the chances of finding a successful one.
- *Loop-free:* The protocol produces loop-free routes and does not require any extra machinery to either break loops or detect them.

The protocol was simulated on the power-law network model ([4, 5]) augmented with bandwidth assignments. The paper describes simulations that simulate streams of requests originating from different sources and targeted at different destinations. The simulation follows all the steps of the protocol, including the actual assignment of bandwidth, random duration of flow, resource releasing and so on, in contrast to some past papers that described simulations in which only an independent probability of success is assigned to each link.

The rest of the paper is organized as follows. Related work is discussed in Section 2. The OPsAR protocol is presented in Section 3. Simulation results are described in Section 4.

2. Related Work

QoS in general, and QoS routing in particular, have received considerable attention from the research community in the past years. A good survey of QoS routing can be found in [6]. Although the current paper is presented in the context of unicast QoS routing, it draws heavily on multicast QoS routing techniques.

The two main approaches used in multicast QoS routing are Single Path Routing (SPR) and Multi Path Routing (MPR):

- Single path routing refers to the traditional multicast routing protocols in which a new group member connects to the multicast tree along the unicast route towards the tree root (e.g., PIM [7], CBT [8]). Extensions to such protocols to support QoS routing have been proposed (e.g., [9]).
- Multi path routing is used to "scan" multiple routes concurrently in order to find the best one that satisfies a QoS reservation request that connects the new member to the multicast tree.

The two multi-path routing schemes described in QMRP [1] and S-QMRP [2] propose an adaptive approach. In both, the unicast route towards the multicast tree root is checked first. If the unicast route has enough resources to address the QoS request, then the target route follows the unicast route. Otherwise, every protocol deploys its own mechanism in order to bypass congested nodes. Upon hitting a congested network node that cannot satisfy the QoS request, QMRP branches from the unicast route and continues searching on multiple branches, for a route that does satisfy the QoS request.

In contrast, S-QMRP does not branch on the way toward the root of the multicast tree. Instead, it first reaches the root of the tree, and then it scans towards the new multicast group member searching for a branch that will satisfy its QoS request.

Another related protocol is QoSMIC [10]. QoSMIC is best suited to a multicast environment, since it looks for a point on a multicast tree to "hook" on a new receiver. The basic case where a single receiver tries to reserve resources

from a data source is not handled optimally in QoSMIC, since there are no means in QoSMIC to cope with a lack of resources on the first branch from source to destination (i.e., when no other group members exists).

Another relevant work is ticket-based probing [3] which extends [11]. The latter deploys multiple probe messages that strive to reach the destination of the QoS route request. These probes are sent based only on the state of local QoS resources. Ticket-based probing [3] extends this work by applying a new constraint on the flooding process: Every source of the QoS route-request assigns a limited number of "tickets" and distributes them among multiple probe messages. Every probe message is supposed to carry at least one ticket. Thus, when a probe message splits into multiple probes at a network node, it is limited by the total number of splits.

Other unicast routing protocols use link state either with or without aggregation of information. Some of these protocols also enforce an hierarchical structure on the network. The usage of aggregation and hierarchical structures are common approaches to achieve scalability by reducing the size of the network global state (see [6]).

3. Protocol

3.1. The Network Model

The network is described as a graph $G = \{V, E\}$. Each edge (link) has allocated resources (e.g., bandwidth) that may be different for each of its directions. Each node knows the initial resource availability in each direction on each link and the availability of that link.

We assume the existence of a unicast routing protocol. Each node can store some temporary information, and can maintain some local state (e.g., resource availability on every outgoing and incoming link).

We do not assume the existence of any global network state by the nodes.

We assume that a reservation process starts when the edge router receives a request to reserve some bandwidth toward a specific destination (usually reserving bandwidth for an application that transmits data to one or more receivers). The process of reserving the bandwidth will be referred to as a transaction. The edge router is the transaction-source and the target is the transaction-destination.

3.2. Motivation

Our objectives are similar to the objective of [2] that improves those of [1]. Our main objective is to achieve an improved tradeoff between the success ratio of meeting a requested resource allocation and the overhead required. Like many existing protocols ([12, 10, 1, 2]) we also chose a tradeoff between the overhead of the protocol and the success ratio it produces. As was already observed by previous papers, the key to the performance is how to make an efficient path selection. Our method is to make an educated guess of a feasible path by leveraging on the knowledge-state at the nodes.

In addition, we seek a solution that adheres to unicast routing, as long as resources are available, and attempts to bypass a congestion point. When all else fails, the protocol forks to controlled multiple path selection.

The objective is to design a scalable protocol with bounded memory requirements and bounded message overhead per transaction, on average. The OPsAR protocol, described below, strives to achieve these objectives.

3.3. Protocol Overview

OPsAR handles transaction-source requests to reserve QoS resources along a path to the transaction-destination. Each node maintains a local state in which it holds its links' status and the resource availability on them. It also maintains a bounded list of pairs <target nodes, outgoing-link>, and for each one it maintains the resources availability toward the target-node and from the target-node with respect to that outgoing link. This information is updated occasionally and is marked to identify the time of its last update. That information is the node's knowledge-state.

Any message traversing a node is used to update the knowledge-state by updating the resource availability to/from the message origin (whether it is the transaction-source or transaction-destination) and the resource availability along the message's last hop. The protocol messages are: TRY-MESSAGE, SCAN-MESSAGE, ACK-MESSAGE, NACK-MESSAGE and RELEASE-MESSAGE.

A source-node initiates a request by composing a TRY-MESSAGE and initiating the Try_phase of OPsAR. The TRY-MESSAGE contains the QoS resource request (transaction-source), the transaction-destination, and the list of traversed nodes (This list is used to avoid routing loops).

When a TRY-MESSAGE arrives at a node, it tries to choose the best link that can meet the QoS requirements. The choice is made according to the resource availability along the various links toward the target, and according to how recent that information is (based on its knowledge-state). There is a preference for choosing the link that is on the shortest path unicast route, over other links. A counter on the TRY-MESSAGE identifies the number of off-piste decisions in which the unicast route was not chosen. The amount of deviations from the shortest path unicast route that a request can make is bounded, using the off-piste counter. Once that bound is reached, no further off-piste traversal is allowed. In this case if the unicast route cannot be taken at a certain node, that node initi-

ates a backward-path NACK-MESSAGE to release all resource reservation with respect to that request[4]. The TRY-MESSAGE is marked as failed and is sent along the unicast path (of the unicast routing protocol) to the target-node to initiate the Scan_phase.

If the TRY-MESSAGE arrives successfully at the target-node, it responds by sending an ACK-MESSAGE along the reverse path. The ACK-MESSAGE is used also, as mentioned above, to update the resource availability toward and from the transaction-destination node along the path.

If the target-node receives the request to initiate a Scan_phase it invokes a scan toward the transaction-source. The knowledge-state is used to choose intelligently to which links to fork in order to increase the chance of success. The scan forks at nodes along the way until the maximal number of alternative paths is reached. We use a ticketing scheme ([3]) to bound the total number of paths. We also limit the branching degree at each node in order to increase the variety of potential paths to traverse. On the other hand, we limit the distance from the shortest path by using an off-piste counter similar to the Try_phase. When the off-piste limit is reached and the unicast route does not have the resources, or when no outgoing link has the requested resources the SCAN-MESSAGE is dropped.

An important observation is that we neither reserve resources in the Scan_phase nor keep any state that relates to the specific scan.

If the transaction-source receives several successful scan messages, it initiates the Try_2_phase. It chooses the "best" route from the successful scan messages and asks to reserve the resources along that path by generating a TRY-MESSAGE with the explicit requested route. This reservation request TRY-MESSAGE for explicit route resembles the traditional RSVP, until a reservation failure along the explicit route is detected. From that failure point, the OPsAR tries to route the reservation request message (TRY-MESSAGE) to the transaction-destination using alternative routes that the off-piste mechanism offers. If that fails, this message is not forwarded to the transaction-destination to request another scan process (as oppose to the initial Try_phase). Instead, a NACK-MESSAGE is returned to the transaction-source indicating the need to choose another explicit route from the previous scan results.

As in the classic RSVP, there is always the option to release the reserved resources by sending a RELEASE-MESSAGE message along the reservation path.

4 Note that the backward failure message is used to update the resource availability toward the transaction-target in each node along the path.

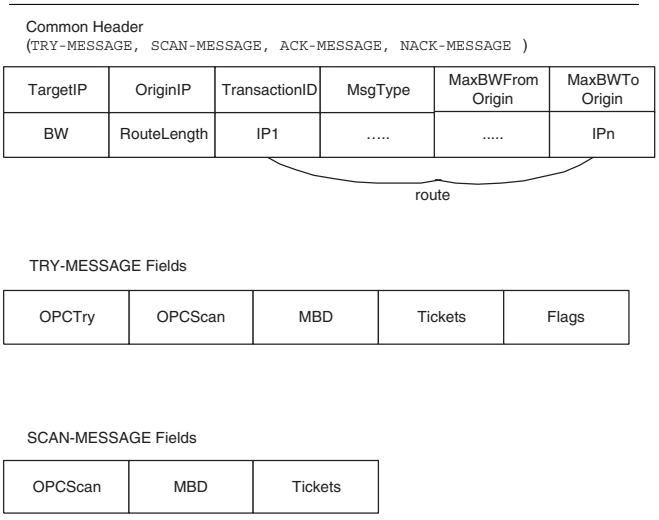

Figure 1: Messages Format

3.4. Detailed Description

3.4.1. Terminology and notations In the protocol description we make use of the following notations:
nh : a candidate next hop;
nu_{dst} : next hop according to the unicast route to $\mathrm{d}st$;
$t_{\mathrm{d}st}$: transaction-destination;
t_{src} : transaction-source;
bw : required bandwidth;
ne : group of all the node's neighbors;
$ne_{\mathrm{fresh}(dst)}$: group of neighbors with recently-updated knowledge-state to/from $\mathrm{d}st$ node;
$ne_{stale(dst)}$: group of neighbors with stale knowledge-state info to/from $\mathrm{d}st$ node;
$ne_{old(dst)}$: group of neighbors with no knowledge-state info to/from $\mathrm{d}st$ node;
$b_{in}(nh)$: the amount of free bandwidth from neighbor nh;
$b_{out}(nh)$: the amount of free bandwidth to neighbor nh.

3.4.2. message format The message format fields are given in Figure 1. The following is a brief description of each field and its usage:
$TargetIP$: The message target IP address, which can be the address of either transaction-destination or transaction-source. The message is forwarded hop by hop according to the protocol (not necessarily according to the IP forwarding table);
$OriginIP$:[5] The message origin IP address, which can be

5 Both $TargetIP$ and $OriginIP$ are presented here only for simplicity. They could be inferred directly from the IP header of the OPsAR

the address of either transaction-destination or transaction-source;
$MaxBWFromOrigin$: The maximum bandwidth available through some route from OriginIP to the previous hop IP[6]. This field is updated at every node that it traverses, according to the node's local knowledge-state;
$MaxBWToOrigin$: The maximum BW available through some route from the previous hop IP to the Origin IP. This field is updated at every node that it traverses, according to the local knowledge-stateat the node;
$P1 - Pn$: The current message's route from the Origin IP to the previous hop IP ;
$RouteLength$: The current message's route length;
BW : The bandwidth requirement.
$OPCTry$: The maximum Off-Piste Count (the number of deviations from the unicast route) allowed for the TRY-MESSAGE;
$OPCScan$: The maximum Off-Piste Count (the number of deviations from the unicast route) allowed for the SCAN-MESSAGE;
MBD : The Maximum Branching Degree allowed for the SCAN-MESSAGE to use at each Branch;
$Tickets$: The maximum number of individual routes allowed for the SCAN-MESSAGE in the Scan_phase;
$flags$: Try_phase flags. TRY_FAILED - means the Try_phase failed to find a route. NO_SCAN - means do not invoke Scan after the Try_phase.

3.4.3. Knowledge State The knowledge-state data structure at a node (denoted as ks) contains a set of records, one per pair of <outgoing-link, target-node>. We maintain a *global_count* variable per endpoint (which is an edge router that participated in a transaction as either a transaction-source or a transaction-destination) that is incremented by one for each update done for any of the records pertaining to that endpoint. Each record contains the following fields:
bw_{out} : is the bandwidth available to the endpoint from the neighbor attached to the link;
bw_{in} : is the bandwidth available from the endpoint to the neighbor attached to the link;
refresh_count: contains the value of the *global_count* at the time of the last updating of the record.

3.4.4. Protocol Flow In the context of the specific protocol we will limit our description to bandwidth reservation requests. Other QoS resources (additive and nonadditive) can be handled in a very similar manner. Also, the protocol is described as if the transmitting application starts the reservation request (starts the Try_phase). However, it can be easily adapted to use the RSVP model in which the receivers start the reservation requests towards the data source.

The reservation process starts when the edge router (transaction-source) receives a request to reserve some bandwidth toward a specific destination (transaction-destination). The transaction-source constructs a TRY-MESSAGE and then processes it as if it was received from the network.

Every intermediate node that receives a protocol message invokes the message dispatcher algorithm (see Figure 8). The message is handled according to its type. Identifying the next hop node for TRY-MESSAGE is handled by the *Try-decision* algorithm (Figure 9) and for a SCAN-MESSAGE by *Scan-decision* (Figure 12).

3.4.5. Maintaining the Knowledge State The knowledge state is organized in records. The records are kept for each pair of a transaction endpoint[7] and a network interface. For every ongoing transaction processed by a node, the node consults its knowledge state. If a corresponding record is not found then a new record is created. Note that a transaction involves at least two records[8]: one record is a knowledge database containing information about the known available bandwidth on the path from the node to the transaction-destination. The other record maintains information about the path back to the transaction-source (the "backwards path").

When a protocol message is received by a node, the node retrieves the knowledge record corresponding to the message *origin* endpoint and the incoming interface. It then updates the record according to the message content. In addition, the node updates the message content to reflect the maximum bandwidth available to/from the message origin, based on its updated knowledge-state information. The next hop router will later uses this information to update its own knowledge-state.

The set of records per endpoint is divided into three categories:

- The first category is for the recently updated records (corresponding to the $ne_{fresh(dst)}$ group).
- The second category is for stale records that have not been recently updated (corresponding to the $ne_{stale(dst)}$).
- The third category contains records that are candidates to be aged out, or already aged out (corresponding to the $ne_{old(dst)}$).

packet.

6 The previous hop IP can be identified from the full route piggyback on the OPsAR packet (see $P1 - Pn$).

7 Whether it is the source or the destination.

8 Observe that for every transaction multiple protocol messages may pass through a node.

Refer to *Try-decision* algorithm (Figure 9) and *Scan-decision* algorithm (Figure 12) that specify the use of these groups. There are two thresholds used by the algorithm: *Stale Threshold* and *Aged-out Threshold*. The assignment of a record to a category is done in the following way: if *(global_count - record.refresh_count)* is smaller than the stale threshold then the record belongs to the fresh records set. If (*global_count - record.refresh_count)* is between the stale and the aged out thresholds then the record belongs to the stale records set. If *(global_count - record.refresh_count)* is greater than the aged-out threshold, then the record belongs to the aged out records set.

Record assignment and aging out are ongoing processes. Each query/access of a knowledge-state data structure about a specific endpoint triggers the aging and re-assignment process as needed, for the records that correspond to that specific endpoint.

Apart from aging within the set of records belonging to a specific endpoint, the algorithm also has a mechanism for aging out a whole set of records belonging to a specific endpoint. This means that, whenever the algorithm reaches a predefined memory limit, the records associated with the least updated endpoint are removed. Note that *all* the records in all the sets (fresh/stale/aged-out) of the node are removed.

4. Simulation

We have validated our protocol using extensive simulations. This section describes both the simulation model, the environment used and the result achieved.

4.1. Simulation Model

The OPsAR protocol was simulated extensively using the NS-2 simulator [13]. Our simulations were conducted on Power-Law network topologies [4]. The Power-Law topologies are based on the results reported in [4], which showed that the node degrees in the Internet obey a power log law: most nodes have small degrees and a small number of nodes have large degrees; as the degree increases, the number of nodes with that degree decreases exponentially. We used the topology generator described in [5].

The simulations were done in networks with 600, 1300, and 2000 nodes. The bandwidth on the links was uniformly distributed from {10,34,45,100 } Mb/s. In order to make sure that the congestion would first occur in the core network and not on the first or last hop, we re-assigned the bandwidth of the endpoints (the nodes that would act as transaction-source and transaction-destination) to 1000Mb. This reflects the case where the last hop is a server with a gigabit network interface card (or a last hop router with a gigabit interface receiving the reservation request to servers/hosts residing in its directly attached networks). We also conducted tests with hierarchical bandwidth assignment chosen from {10Mb,100Mb,1G,10G} bits per second, capacities that are in use in Metro and backbone networks. The networks we have simulated with these link capacities (and with the number of nodes varying from 600 to 2000) are usually considered as over-provisioned (especially in the case of gigabit+ links). In these over-provisioned networks we have confirmed that there is almost no congestion for the current usual bandwidth reservation requests. Therefore, the topologies simulated were only large edge networks (autonomous systems or large enterprise networks) and ISP like networks.

As already noted, the topology used for the simulations were generated according to the the power-law network model ([4, 5]). Although there is a research that studies ISPs' topologies [**?**], other research efforts in the QoS routing area used the lower-law self-generated networks. Thus, in order to compare our results with previous results we have generated the topology with similar tools. Note that the assignment of bandwidth to the links in the simulated topology is not a trivial task. We have chosen the above link bandwidth assignment model in order to be somewhat close to real life network scenarios.

The transaction bandwidth allocation was uniformly distributed between [1,100] Mb. A transaction was invoked every 0.5 seconds and the transaction lifetime was uniformly distributed out of [1,1000] seconds; thus an average of 1000 transactions concurrently existed in the network. About one fifth of the nodes in the simulated network were used as endpoints of the transactions. These nodes were selected based on their distance (hop count) from the network core. The network core is the set of routers with which the power-law network generator starts.

In the simulations conducted, in contrast to the other simulations done in [1, 2], we practically allocated the QoS requirement on the links and released it when the transaction's lifetime was expired. This simulated rather closely a "real" world model of transaction allocation and thus led to more realistic results.

4.2. Simulation Results

For the basic set of simulations, a network of 600 nodes was used. The transaction-source and transaction-destination were uniformly selected out of 120 edge nodes. In most of the simulations, each point in the simulation results' graphs is the result of 10,000 transactions performed on six different generated topologies.

To address the scalability issue, we also used a larger number of nodes. To measure the memory requirement and

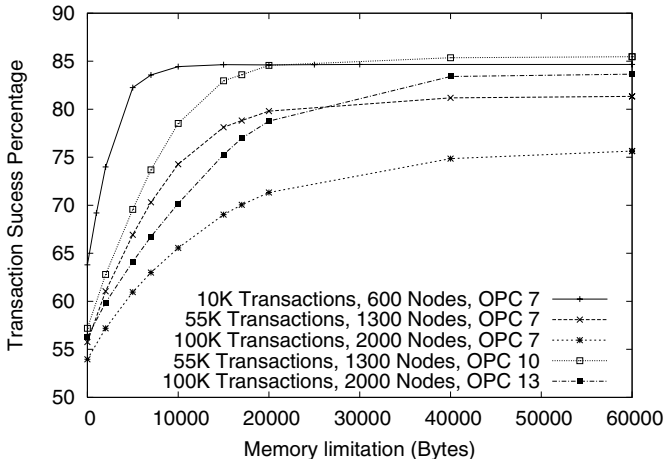

Figure 2: Memory usage vs. success ratio

impact on reservation success ratio we conducted simulations with 55,000 and 100,000 transactions.

Our simulations were done in the context of unicast reservations. However, with minor adaptations, OPsAR can be used for QoS multicast routing as well. For example, this can be accomplished by terminating the search for a destination as soon as a node which is not a tree is encountered.

We ran each simulation on 5 different protocol types:

- Traditional RSVP - We adapted the traditional RSVP to allocate the QoS requirement along the unicast route toward the transaction-destination;
- S-QMRP* - We adapted the S-QMRP protocol [2] so that it would work on top of unicast routing. The adaptation was done by applying the S-QMRP protocol on a "degenerated" multicast tree that is comprised of a single node, which is the transaction-destination in the simulation. Also, S-QMRP* reserves bandwidth and does not handle delay gurantees (as S-QMRP does). Thus the branching decision in the Scan_phase is not based on the end to end delay information;
- S-QMRP*D - We enhanced the S-QMRP* protocol to allow deviations from the shortest path on the Try_phase;
- OPsAR - The Off-Piste QoS-aware Routing Protocol;
- OPT - Implemented as a BFS from transaction-source to transaction-destination which finds the shortest path that fulfills the bandwidth QoS requirements. Every transaction was handed to the OPT protocol. At every transaction the initial state of the protocol was derived from the accumulated state of the transactions previously handled by the protocol.

S-QMRP [2] is an enhancement of QMRP [1]. The expected performance of QMRP for non-additive metrics is similar to S-QMRP ([2]). Therefore we did not implement it, nor did we compare it to OPsAR.

As was described in Section 2, a relevant protocol to compare with is QoSMIC [10]. However, as described, there are some inherent differences between OPsAR and QoSMIC. As such, direct comparison with QoSMIC is beyond the scope of this paper. Nevertheless, QMRP [1] shows better results than QoSMIC for non-additive metrics.

We performed different simulations to compare different parameters and their relationship to the reservation success ratio: 1. Message overhead; 2. Memory usage; 3. Amount of concurrent transactions; 4. Number of edge nodes; 5. Number of destination nodes.

The simulations proved that the results of the OPsAR protocol were very close of the results of the OPT, and more closely resembled those results than any of the other protocols.

4.3. Memory Usage vs Success Ratio

Due to the nature of OPsAR and its usage of cache, it is important to measure the effect that the amount of available memory has on its success. In order to evaluate the memory usage of OPsAR and the performance implications of the amount of available memory, we have conducted the following test. We ran the simulation with different amounts of memory limit. The implementation of "memory limit" was that in case a new cache entry was to be saved and the amount of consumed memory was above the limit, we have deleted some old information from the cache in order to accommodate the new data. In addition to running the protocol with different amounts of memory limitations, we used simulated networks of 600, 1300 and 2000 nodes, and 10000, 55000, 100000 transactions correspondingly, with several off-piste counter values. This verified the scalability of memory requirements over network load and size.

Figure 2 shows the consolidated graph of the simulation runs. Each line in the graph is a result of keeping the network size, transaction number and a value of off-piste counter fixed, while iterating with increasing memory sizes. Each point in each line is the average of several runs with the same parameters. In the graph, one can see that each set of parameters resulted in a convergence to a point beyond which an additional amount of memory had minimal, if any, impact on the success ratio.

The reason for the convergence phenomenon is that no new knowledge-state can be effectively used beyond a certain point, because the success ratio limitation is bounded by another protocol parameter. For example, increasing the off-piste counter does increase the success ratio, as can be seen in the graph (though resulting in more messages and

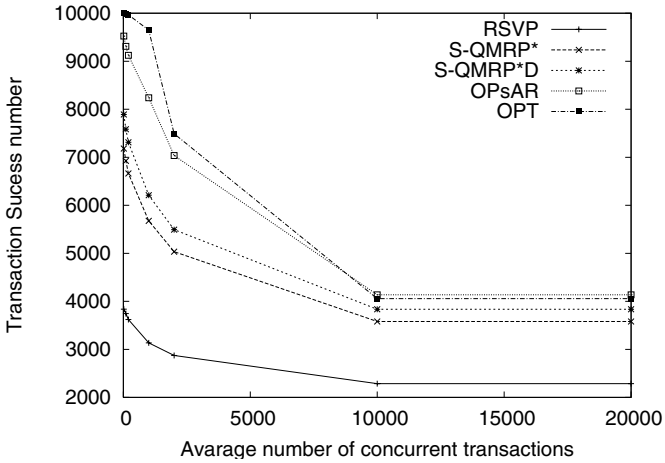

Figure 3: Concurrent transactions vs. success ratio

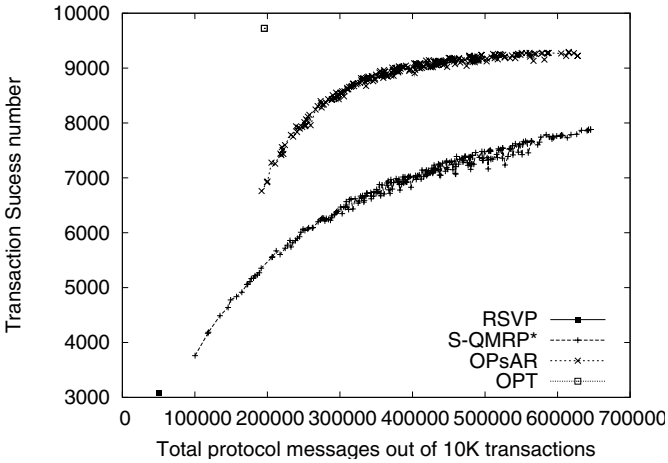

Figure 4: Message overhead vs. success ratio

longer routes). The graph shows variation of the off-piste value from 7 to 13, in a 2000-node network where we ran 100,000 transactions. By varying the off-piste values, we improved the results from 76% to 85%.

All in all, we can conclude from the graph that the amount of memory sufficient to achieve about 85% of success ratio is very reasonable. Furthermore, the memory bound is theoretically bounded by the out degree of a core node times the number of possible transaction-destination nodes. The protocol makes use of the aged-out threshold in order to limit the amount of neighbors per destination for which the knowledge-state is kept (in our simulations this threshold was set to 9). For example, in the largest network simulated (2000 nodes and 400 participating endpoint where each knowledge-state record in the simulation code consumes 40B) it was about 160KB. In practice, a much smaller memory size suffices. The average memory consumption was about 10% of the theoretical bound, where 60KB was the actual bound set in the simulation code. Our simulations showed that increasing the number of neighbors for which knowledge-state per endpoint is kept (aged-out threshold), for example from 9 to 60, did not produce any better results.

4.4. Concurrent Transactions vs Success Ratio

In order to evaluate the performance implications of the number of concurrent transactions, we ran the simulation with a different transaction lifetime average. Figure 3 depicts the results and shows that the success value improvement of the OPsAR over other protocols gets lower as the transaction lifetime grows, and that the performance results of OPsAR are quite close to that of the OPT.

4.5. Message Overhead vs. Success Ratio

The OPsAR protocol is flooding based that explores portions of the BFS tree rooted at the transaction-source. The flooding capabilities of the protocol can be configured by changing one of the following three parameters: branching degree, scanning deviation, and number of tickets. We studied all the parameter's possible combination within a specific range and ran the simulation on all the flooding based protocols (OPsAR,S-QMRP*,S-QMRP*D). Each simulation result generated one point in the graph: the transaction success number as a function of messages overhead. Figure 4 depicts the results. Since S-QMRP*D performed minorly better then S-QMRP*, we chose not to present it in the graph (for clarity). Note also that OPT and RSVP are presented as single dots in the graph. We can observe that for the same amount of message overhead, the OPsAR improves the success ratio up to 30% more than S-QMRP*.

The message overhead required to achieve a given ratio of success was also compared. For example, in order to achieve success in 7500 transactions out of 10,000, the OPsAR required about 230,000 messages while in order to achieve the same success the S-QMRP* required about 530,000 messages. Another interesting point to note is that since the RSVP protocol is not affected by the adjusted parameters, it was more successful when the topology and transactions were constant. The success ratio of RSVP in any given set of topology and transactions was 3,000 out of 10,000, where the amount of messages it used was about 50,000. The OPsAR used 260,000 in order to achieve success of 8,700 transactions. Thus, increase of overhead by five times yields about three times the success ratio. Consider the OPT algorithm, which uses 200,000 messages to achieve a success of 9,700 transaction, with an over-

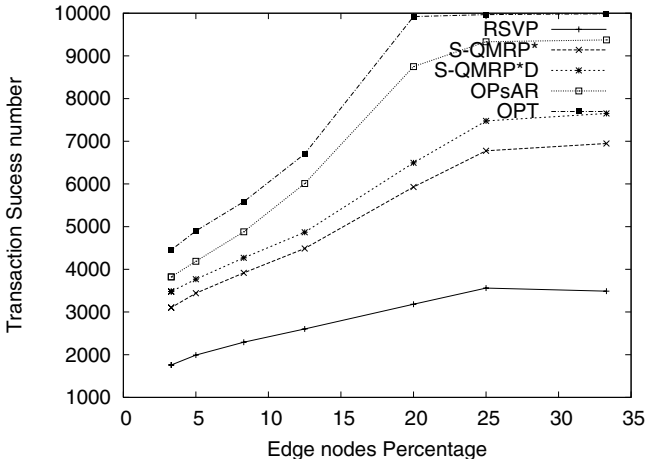

Figure 5: Number of edge nodes vs. success ratio

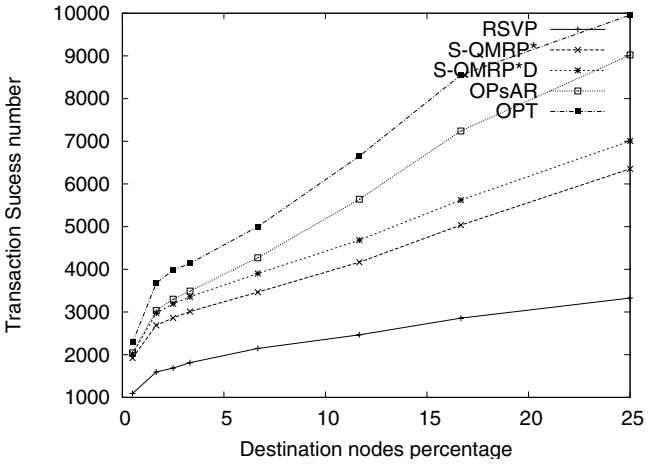

Figure 6: Number of destinations nodes vs. success ratio

head/sucess ratio (number of messages per success transaction) of 20.6. The overhead/sucess ratio of the OPsAR, on the point above, is 29.8 which is only 30% more then the OPT. Another point to consider is that the average path length is about 8 hops when deviation is allowed and 4 hops when deviation is forbidden (e.g. RSVP).

4.6. Number of Edge Nodes vs Success Ratio

In order to evaluate the performance implications of the distribution of edge nodes, we ran the simulation with varying number of edge nodes. For each of those runs we extracted the total transaction success value. Figure 5 depicts the results. In all the protocols we noticed that when the ratio of edge nodes increases (while the total number of concurrent transactions is constant) the success ratio improves. The reason is that the transactions are spread over larger areas of the network and there are less bottlenecks. We can also observe that OPsAR always succeeds more than others and closely adheres to OPT.

4.7. Number of Destinations Nodes vs. Success Ratio

To further base the findings of the previous subsection, we increased the ratio of the destination nodes, while keeping the number of edge nodes and concurrent transactions the same. We ran the simulation with a constant number of 25% edge nodes (as opposed to the 20% we usually used). The number of *candidate* destination nodes (out of which the transaction destination nodes were chosen) varied from 1% up to the whole set of edge nodes (25%). The candidate set of source nodes was always the whole set of edge nodes. Only the links from those destination nodes were assigned a bandwidth capacity of 1000Mb. For each of those runs, we extracted the total transaction success value. Figure 6 depicts the results. We can observe the success ratio of protocols improves drastically as the number of destination nodes is increased. Here, too, the results of OPsAR closely resembled the results of the ultimate protocol OPT.

4.8. The Cost and Performance Gain of Using Try&Scan Phases

The OPsAR protocol improves on past results by using a two stages approach: a Try phase followed (when necessary) by a Scan phase. This double stage process naturally takes longer time than using only a Try phase. We analyzed the simulation results to compare the relative success rate and cost of using only the Try phase to that of using both phases. Figure 7 describes the number of successful transactions using both Try and Scan, and how much of those were obtained at the Scan phase. We can see that in OPsAR less than 25% of the transactions required the Scan phase. Thus, most of the transactions did not required the extra time and messages of the Scan phase. Our simulations showed that the time to complete a Try followed by a Scan is three times the time it takes to complete the Try phase alone. Thus, the expected cost is an increase of about 50% in the time it takes to complete a successful transaction. Observe, that when the network is congested it is more costly to ensure the success of a transaction.

5. Conclusions and Future Work

In this paper, we have demonstrated that maintenance of a limited knowledge-state, combined with constrained multi-path routing, can improve the success ratio for find-

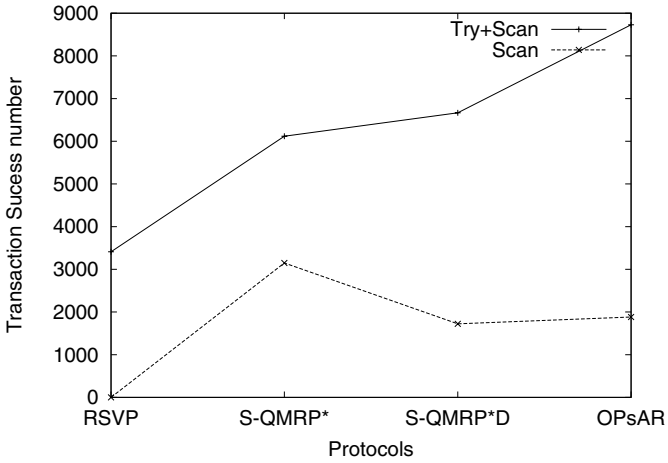

Figure 7: The relative success rate of scan vs. try+scan

ing a path. We defined a knowledge state as a cache of recent route information about resource availability. The results can be explained by observing that the overall scheme of our protocol is an intelligent choice of routes from a full Breadth-First-Search algorithm (BFS). Future research can focus on improving the educated choice of routes while limiting the overhead in memory. We expect to find ways to use machine learning techniques to achieve that goal.

Future work can also try to save in memory by aggregating the information, using techniques like longest prefix matching on transactions destination. For example, knowledge-state entries can be aggregated according to addresses of destinations that share a common prefix. In this case, some aggregation method should be applied to the knowledge-state as well. One such method is to take the minimum on all available bandwidth towards all the endpoints (in the knowledge-state) covered by the aggregated prefix.

Currently, packet losses are not handled by the suggested protocol. However, the protocol can be easily extended to cope with losses. The protocol is sensitive to ACK-MESSAGE and NACK-MESSAGE losses. Using timeouts one can detect the loss event. The protocol can be extended to invoke a special re-try message. Reservations that were made but were not acknowledged would then be released after a short timeout. Moreover, the protocol can be turned into a soft-state based protocol by requiring that the reservation state be refreshed periodically, using a longer timeout.

Currently the OPsAR protocol already includes mechanisms to cope with link/node failures and to overcome inaccuracies in knowledge-state (e.g., old information that was not updated). When a transaction hits a failed link/node an NACK-MESSAGE message is sent back to the transaction-source. This message would update the knowledge-state along its path and would make sure future transactions would not follow the exact same path towards the transaction-destination. Turning the protocol into a soft state protocol will also take care of remnants of the past reservations that are affected by the failure.

In this paper, we have presented a non-additive resource reservation (bandwidth). Handling additive resources, like delay, requires minor changes to the protocols presented. For example, we need to maintain the accumulated delay and to test the route of choice accordingly.

The OPsAR protocol presented invokes the transaction request from the data source towards the data destination. However, this methodology was selected in order to simplify the protocol presentation. The protocol can be modified, so that it is invoked from the data destination towards the data source, while making the reservation from the data source toward the data destination. In some network environments, such as shared media links, the handling of a failed TRY-MESSAGE may require a single link backtrack.

In the course of the simulations, while tuning the knowledge-state tunable parameters, we made some interesting observations. For example, we noted the threshold which defines the size of the $ne_{fresh(dst)}$ group and the size of $ne_{stale(dst)}$ group. We found dependencies between the sizes of these groups. For example, we noticed that linear increase in the $ne_{fresh(dst)}$ group size did not necessarily increase the success ratio. Also, even increasing the size of both groups (by increasing the memory size) did not guarantee increase in the performance of the protocol, and in some cases it even caused degradation in the overall success ratio. The source of this phenomenon has to do with the fact that the protocol was trying routes with state information which is not "fresh" enough. Further research must be conducted in order to explore the inter-dependencies among the various variables of OPsAR, and to automatically learn and choose the optimal values, possibly using machine-learning techniques.

5.1. Gradual Deployment within RSVP Framework

The OPsAR protocol can be designed to fit in the RSVP framework (See [14]). Furthermore, there is no inherent limitation in the protocol that prohibits its use in an incremental manner. For example, future work can focus on testing the OPsAR protocol deployed only within several core routers and measure the impact of having the ability to divert from the shortest path (the approach that RSVP take) within the main core routers.

References

[1] Shigang Chen, Klara Nahrstedt, and Yuval Shavitt, "A qos-aware multicast routing protocol," in *INFOCOM (3)*, 2000, pp. 1594–1603.

[2] Shigang Chen and Yuval Shavitt, "Scalable Distributed QoS Multicast Routing Protocol," TR 2000-18, Aug. 2000, DIMACS Research Report.

[3] Shigang Chen and Klara Nahrstedt, "Distributed QoS Routing with Imprecise State Information," in *Proceedings of 7th IEEE International Conference on Computer, Communications and Networks*, Lafayette, LA, October 1998, pp. 614–621.

[4] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos, "On power-law relationships of the internet topology," in *SIGCOMM*, 1999, pp. 251–262.

[5] Reuven Cohen, Danny Dolev, Shlomo Havlin, Tomer Kalisky, Osnat Mokryn, and Yuval Shavitt, "On the tomography of networks and multicast trees," .

[6] S. Chen and K. Nahrstedt, "An overview of quality-of-service routing for the next generation high-speed networks: Problems and solutions," 1998.

[7] A. Helmy D. Thaler S. Deering M. Handley V. Jacobson C. Liu P. Sharma D. Estrin, D. Farinacci and L. Wei., "Protocol independent multicast-sparse mode (PIM-SM): Protocol specification," RFC 2117, June 1997, Internet Engineering Task Force, Network Working Group.

[8] P. Francis T. Ballardie and J. Crowcroft, "Core-based trees (cbt) an architecture for scalable inter-domain multicast routing," in *ACM SIGCOMM*, 1993, pp. 85–95.

[9] Bin Wang Hung-Ying Tyan, Chao-Ju Hou and Yao-Min Chen, "Qos extension to cbt," .

[10] Michalis Faloutsos, Anindo Banerjea, and Rajesh Pankaj, "Qosmic: Quality of service sensitive multicast internet protocol," in *SIGCOMM*, September 1998, pp. 144–153.

[11] Shigang Chen and Klara Nahrstedt, "Distributed quality-of-service routing in high-speed networks based on selective probing," in *LCN*, 1998, pp. 80–89.

[12] K. Carlberg and J. Crowcroft, "Building shared trees using a one-to-many joining mechanism," 1997.

[13] "UCB/LBNL/VINT Network Simulator - ns (version 2), 1997," URL: http://www.isi.edu/nsnam/ns.

[14] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification," RFC 2205, September 1997, Internet Engineering Task Force, Network Working Group.

Main-message-dispatcher(message, in_link):
Knowledge-state-update(message, in_link);
Message-bw-update(message);
switch (message.MsgType)
case TRY-MESSAGE:
/* Message's TargetIP is t_{dst}. */
if (message.Flags = TRY_FAILED)
send($message, nu(t_{dst})$);
return;
nh = Try-decision();
if ($nh = NULL$ or
($nh \neq nu(t_{dst})$ and $message.OPCTry = 0$))
/* Inverse route and message Source/TargetIP. */
/* NACK message TargetIP is t_{src}. */
message-nack.route = inverse(message.route);
send($message - nack, pop(message - nack.route)$);
/* if Try_phase (and not Try_2_phase). */
if (message.Flags $\neq$ NO_SCAN);
message.Flags = TRY_FAILIED;
send($message, nu(t_{dst})$);
return;
if ($nh \neq$ nu(t_{src}))
$message\_tmp.OPCTry - -$;
reserve($nh, message.BW$);
send($message, nh$);
case SCAN-MESSAGE:
/* Message's TargetIP is t_{src}. */
$\{nh\}$ = Scan-decision();
$nh\_size = \min(message.Tickets, message.MBD)$;
$\{nh'\}$ = take nh_size nodes out of $\{nh\}$;
while ($\{nh'\} \neq \emptyset$)
nh = *pop*($\{nh'\}$);
$message\_tmp = message$;
if ($nh \neq$ nu(t_{src}))
if ($message.OPCScan = 0$) then continue;
$message\_tmp.OPCScan - -$;
$message\_tmp.Tickets/ = nh\_size$;
send($message\_tmp, nh$);
case ACK-MESSAGE:
/* Message's TargetIP is t_{src}. */
$nh =$ pop($message.route$);
send($message, nh$);
case NACK-MESSAGE:
/* Message's TargetIP is t_{src}. */
$nh =$ pop($message.route$);
free reservation ($nh, message.BW$);
send($message, nh$);

Figure 8: Message dispatcher at intermediate node.

Try-decision():

/* In the two steps below we search for the neighbor with maximum end-to-end BW from the 'fresh' knowledge-state group that satisfies the BW requirement. The first step checks whether the unicast next hop meets the requirements. */

if $((\min(ks[nu(t_{\rm dst}), t_{\rm dst}].bw_{out}, b_{out}(nu(t_{\rm dst}))) > bw)$
and $(nu(t_{\rm dst}) \in ne_{fresh(dst)}))$
return $nu(t_{\rm dst})$;

if $(\max_{nh \in ne_{fresh(dst)}} \{\min(ks[nh, t_{\rm dst}].bw_{out}, b_{out}(nh))\} > bw)$
return nh;

/* Take the next unicast hop if it is not in the 'fresh' knowledge-state group but satisfies the next hop BW requirement. If it is in the 'fresh' knowledge-state group, then we have information indicating insufficient BW towards destination (end-to-end) */

if $((b_{out}(nu(t_{\rm dst})) > bw)$ and $(nu(t_{\rm dst}) \notin ne_{fresh(dst)}))$
return $nu(t_{\rm dst})$;

/* Take the neighbor with maximum end-to-end BW from the 'stale' knowledge-state group that satisfies the BW requirement.*/

if $(\max_{nh \in ne_{stale(dst)}} \{\min(ks[nh, t_{\rm dst}].bw_{out}, b_{out}(nh))\} > bw)$
return nh;

/* Take the neighbor with maximum next hop BW from the 'stale' or 'old' knowledge-state groups that satisfies the BW requirement. We can not use the 'fresh' group because the neighbors from the 'fresh' knowledge-state group do not have enough end-to-end BW availability (according to the knowledge-state). */

if $(\max_{nh \in ne_{stale(dst)} \cup ne_{old(dst)}} \{b_{out}(nh)\} > bw)$
return nh;

/* Take the neighbor with the maximum next hop BW from the entire neighbors group.*/

if $(\max_{nh}\{b_{out}(nh)\} > bw)$
return nh;

return NULL;

Figure 9: Decision making in the try message process.

Knowledge-state-update(message, in_link):

if ($ks[in\_link, message.OriginIP]$ = NULL) then create it;
$ks[in\_link, message.OriginIP].bw_{out}$ = message.MaxBWToOrigin;
$ks[in\_link, message.OriginIP].bw_{in}$ = message.MaxBWFromOrigin;
$ks[in\_link, message.OriginIP].refresh\_count = global\_count$;

Figure 10: Updating the knowledge-state data base.

Message-bw-update(message):

message.MaxBWToOrigin =
$\max_{nh}\{\min(ks[nh, message.OriginIP].bw_{out}, b_{out}(nh))\}$
message.MaxBWFromOrigin =
$\max_{nh}\{\min(ks[nh, message.OriginIP].bw_{in}, b_{in}(nh))\}$

Figure 11: Updating the message's BW fields.

Scan-decision():

The scan decision is identical to the Try-decision() described in Figure 9 with the following exceptions: 1. Every out should be replaced by in 2. Each "return" should not return the best (max) nh that fulfills the condition but rather collect all nhs that fulfill it, and return the list sorted by the bandwidth availability.

Figure 12: Decision making in the scan message process.

*