Decentralized topology aggregation for QoS estimation in large overlay networks

Stefan Wieser, Laszlo Böszörmenyi
Department of Information Technology
Klagenfurt University, Austria
{swieser,laszlo}@itec.aau.at

Institute of Information Technology
University Klagenfurt
Technical Report No TR/ITEC/01/2.11
2 March 2011

Abstract

This paper introduces a scalable approach for efficient, low-cost multi-level Quality of Service (QoS) estimation in large overlay networks (ON). We modify an existing distributed partitioning algorithm [7], and use it to create hierarchical aggregations of any underlying ON's topology and its QoS. We call these aggregations "QoS maps". QoS maps empower applications to quickly predict several QoS metrics for any given route, and to obtain multiple alternative routes to any given target node in the ON, along with an estimate of their QoS. Simulations with large ONs are performed to evaluate the proposed approach and demonstrate its scalability. We show that our modifications of the partitioning algorithm permit the aggregation of large hubs - nodes with a lot of neighbors, commonly found in ONs - but still preserve the sub-linear runtime of the original heuristic. Finally, we provide an algorithm that uses QoS maps to perform QoS estimation and QoS aware routing in any given ON.

1 Introduction

While ONs have received a lot of attention both in practice and in research, routing and QoS prediction in large ONs remain difficult problems [10]. In networks with thousands of nodes, maintaining a global view of the topology is no longer an option.

Our goal is to provide applications in existing large ONs with so-called "QoS maps" that serve two purposes: First, they allow low-cost estimates of the QoS experienced on different paths. Second, they provide applications multiple alternative paths to any target node. Consider, for example, a video conferencing application with certain bandwidth and jitter constraints. Said application may not find any route that satisfies both QoS constraints. We provide this application with a fast, coarse-grained estimate of QoS along different paths in the ON, which may contain routes that satisfy one constraint, but not the other. Let us assume that smoothness is more important for the application than image quality. Therefore, it decides to use a path with low jitter, even if it has low bandwidth and the resolution of the video has to be reduced. At the same time, it discards a route that would have sufficiently high bandwidth for the full resolution video, but has also high jitter. Finally, by using multi-level estimation as described in this paper, the application can refine its estimate step by step if desired and increase the accuracy of the estimate.

To perform efficient, QoS-aware routing in ONs that are too large to have a global view, we simplify their topology by hierarchical aggregation. This aggregation, together with the aggregated QoS information stored throughout the network is what we refer to as a "QoS map". In contrast to many other overlays that create a new hierarchical topology to disseminate data, our QoS maps preserve the *existing* topology of the underlying network.

Every node has a different view of the network, and hence a different QoS map. The QoS map of a single node offers coarse but low-cost QoS estimates of any path in the network. By combining the views of the nodes on a route, that aggregated information can be refined. This allows nodes to obtain more accurate information of routes at the cost of a higher computational and network cost caused by the query.

The main contributions of this paper are as follows: First, it explains how an existing distributed partitioning algorithm should be changed, so that it partitions any ON into sub-networks ("clusters") that can be used for distributed hierarchical aggregation. Second, it introduces three different methods to deal with hubs - nodes with a lot of neighbors - during partitioning, and evaluates their effects on the aggregated topology. It shows that with our modifications, large ONs can be aggregated into low-depth hierarchies with clusters of a small, bounded size. As a result, nodes only need to store little state about the ON to allow QoS aware routing. Finally, it presents an algorithm to perform multi-level QoS estimation using the aggregated information.

QoS maps make use of three domains: Topology aggregation (TA) [21], distributed partitioning, and QoS epitomes [19]. TA is used to increase scalability by avoiding a global view. Distributed partitioning is needed to partition the ON into distinct clusters for TA. Finally, QoS epitomes are used to aggregate the QoS of each cluster. Each of

these approaches is applied hierarchically to facilitate multi-level QoS estimation of the usual common QoS metrics.

2 Related Work

A number of approaches attempt to solve the problem of scalability that large networks face.

Hybrid ONs, such as [17] or [23], and ONs that build distribution trees [6], are popular choices for multicast data dissemination. They differ from our approach in that they either cannot provide alternative routes and QoS estimates to applications, or rely on the underlying network for routing entirely. OverCast [11] uses a two-tier architecture to increase scalability. While somewhat similar to a hierarchical approach, it is limited to only two levels (peers and super-peers). Furthermore, they also do not provide alternative routes or perform QoS estimation. NICE [3] also builds a hierarchy with bounded cluster size, however, it is optimized against a certain metric, such as delay, and builds a new ON based on it. This causes suboptimal routing performance if any other QoS metric is used by the application. In contrast, our approach preserves the structure of the underlying network, permits scalable routing and multi-level QoS estimates based on any QoS metric known to the network.

Finally, pheromones-based approaches mimic the behavior of ants foraging food, and are a popular way to find routes in particular for mobile networks, and networks with high failure rates. They allow finding high bandwidth or low delay routes to target nodes in a completely self-organized manner. Scalability of traffic caused by simulated ants is not trivial and requires careful optimization [1]. Furthermore, despite these optimizations, pheromone-based approaches offer no direct support for QoS estimation. Recently, a novel pheromone-based proxy network specifically for multimedia delivery has been introduced [18]. However, while it provides QoS by prioritizing delivery of data units based on their playback deadline, it still does not go as far as providing estimated QoS information between any pair of nodes.

2.1 Topology Aggregation

We follow the commonly used topology aggregation (TA) approach [21] to increase scalability. By aggregating a topology consisting of several nodes into a new, more compact topology, less data is required to get a high level overview of the whole. In addition, as fewer and smaller updates are needed for localized changes than in the original ON, frequency and size of maintenance traffic decreases. Finally, the reduced complexity of aggregated topologies makes more expensive routing algorithms feasible [21]. The quality of the aggregation is determined by how accurately the reduced topology still represents the underlying network topology, as well as by the amount of information reduction achieved.

QoS epitomes are used to aggregate the QoS of all paths that lead through such a

network. This is done to reduce the amount of information that needs to be advertised to other networks. A large number of approaches exists for that purpose [19]. For example, geometry based QoS epitomes plot QoS metrics, such as delay and bandwidth, from each source to each destination node on a graph. The resulting graph is then approximated by methods such as line fitting, polylines or curve fitting. We assume that QoS epitomes are used to approximate the QoS of the clusters we aggregate. However, we do not limit ourselves to a specific method. The various approximation approaches, their advantages and disadvantages are summarized in [19].

2.2 Network Partitioning

To create the aggregated topology, our ON first has to be split into distinct partitions. As graph partitioning is known to be an NP-complete problem [9], heuristics have to be used. Many approaches, such as [16, 12], assume a global view to partition a network for at least the initial partition step, which does not scale for a large dynamic environment. In addition to that, common k-way partitions are not practical for creating an aggregation as the number of clusters is fixed to k, and the size of clusters, a variable, may turn out to be very large. Finally, as the number of nodes in an ON may not be known in advance, choosing a fitting k may be difficult. [20] proposes a top-down approach to hierarchy creation with a bounded number of children per parent, however they require a central root node that assigns nodes to their correct cluster. While they relax this restriction and distribute the root node's tasks among cluster heads of lower hierarchy levels, nodes represented higher in the hierarchy still receive significantly more load than nodes in lower levels. It is not clear how much network load their load-balancing introduces while the hierarchy is still under creation. [4] introduces a distributed and robust algorithm for ad-hoc network clustering, however, the clusters created by the algorithm have a radius of 1. Furthermore, the amount of nodes within a cluster is unbounded. [5] pursues a similar approach, and repeatedly applies their algorithm in order to create a hierarchy. They also suggest a variant where clusters with a predetermined radius can be built. However, they use a separate leader election step, and do not attempt to put a bound on the amount of nodes within a cluster.

[22] uses a "copying influence" model to partition ONs at weak links, which are represented in a stochastic model as vertices with low copying probability (low influences). However, they offer no direct control over partition size. While the partitions produced by this algorithm will likely partition topology aware ONs at domain boundaries, the partitions (and thus the clusters) may be very large. In addition, distributed termination is achieved by explicitly reducing the influences between nodes. Therefore, the quality of the resulting partition depends on choosing an appropriately small influence reduction size, which in turn increases the iterations the algorithm requires. The simulation uses a network of seven nodes, and while it outperforms existing algorithms, [22] notes that it is not clear how the runtime scales with the size of the graph.

[7] proposes a non-sequential and fully distributed variant of the Basic Partition algorithm introduced by [2], in which nodes "battle" for layers of surrounding nodes. The

advantage over other partitioning approaches is that the Basic Partition produces a set of disjoint connected partitions with *bounded radius*, therefore making the number of partitions variable instead of the size. The heuristic provided in [7] has a low runtime and message complexity. It accounts for variable message delivery time and does not depend on synchronized clocks or leader election. However, naïvely applying it repeatedly to create a hierarchy results in topologies with only trivial (single-node) partitions that cannot be aggregated any further.

We thus make several modifications, which allow us to create partitions with a bounded number of nodes in a fully distributed and parallel way. As every node only has a limited local view, this scales well for networks with a large number of nodes. Similar to [5], we build a hierarchy from the partitions. However, we keep the partition size at any hierarchy level bounded, and combine it with QoS epitomes to enable QoS estimates at any hierarchy level. With a small, bounded partition size, nodes with a complete view of their own partition become feasible. This can be exploited for load-balancing, since any node of a partition knows the exact QoS of its own cluster and can provide that information to other nodes.

3 Hierarchy Construction

In this section, we first describe the general topology of the hierarchical aggregation we create. We then address several issues we encountered that required modifications to the original algorithm, and present them with our evaluations.

Consider an existing large ON of n nodes with unique identifiers. For scalability, we aggregate this network into a hierarchy of h levels: The lowest hierarchy level, level 0, is the original ON and therefore contains all n nodes with no aggregation. This level is then partitioned into several clusters based on the nodes' locality in the original ON, while bounding the number of nodes per cluster with the constant s. The identifier of a cluster is the largest identifier of the nodes it contains. Each cluster on level l is represented as a single (virtual) node on level $l + 1$. A virtual node uses the identifier of the cluster it represents as its own identifier. Connections between nodes of different clusters on level l also exist on level $l + 1$, as a link between the virtual nodes of both clusters. We continue building hierarchy levels until the uppermost hierarchy level consists of at most s clusters.

Figure 1 shows this concept with a hierarchy of three levels and $s = 4$. Parts of the ON and the aggregated topology have been omitted for clarity, and links to them are represented as dashed lines. Level 0 contains the original ON, and is partitioned into several clusters. Each cluster is represented by a virtual node at level 1. This process is repeated, and at level 2, the entire network is represented by only three clusters, making the structure of the network easy to grasp.

When we describe the partitioning and aggregation in the remainder of this section, the term "node" shall refer both to a single node in the ON and to an entire cluster represented as a single node on the next hierarchy level. This simplification of terminology is justified

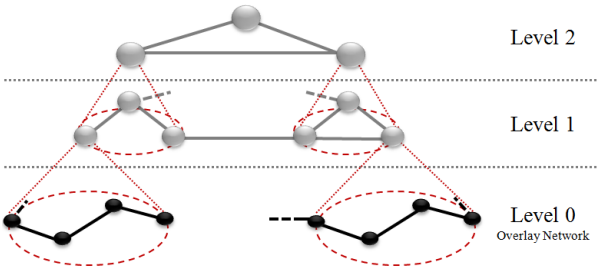

Figure 1: A three level hierarchy with $s = 4$.

by the fact that both entities behave identically for the purpose of the algorithm.

3.1 Distributed Partitioning

To aggregate the topology of the ON, we first partition it. For this, we build up on the partitioning algorithm described in [7], which partitions the network by building clusters. Each node (a potential "cluster head") attempts to add surrounding nodes with increasing hop-count ("layers") to its own cluster. The i^{th} layer consists of all nodes with a distance of i hops from the cluster head that do not already belong to a "finished" cluster. Whenever one node attempts to add another, the node with the highest identifier prevails. A stopping condition is used to decide when to stop adding layers: The original heuristic stops growing a cluster if the ratio between cluster members and neighbors exceeds a threshold that is determined by a configurable constant and the total number of nodes in the network. If, after adding a new layer, this stopping condition is fulfilled, that last layer is dropped again, and the cluster is declared to be "finished". A cluster is also finished if no more nodes are found that could be added. Finished clusters no longer participate in the algorithm.

To partition the ON for our QoS maps, we needed to make several changes to the partitioning algorithm. We replaced its stopping condition with our own, which only requires local knowledge. We changed the message format to allow running partitioning algorithm on multiple levels of the hierarchy simultaneously. This removed the need to explicitly finish partitioning a hierarchy level, before starting with the next. Finally, we modified the heuristic to explicitly deal with hubs.

3.2 Simulation Setup

To evaluate our changes in the light of large ONs, we perform simulations on random graphs with varying density that contain 500 to 5,000 nodes. The simulation advances in discrete steps. Nodes exchange messages to partition the network and aggregate the topology. At each step, every node may read all messages it has received, and may send messages to its *directly connected neighbors*. This is reasonable, as the average time to

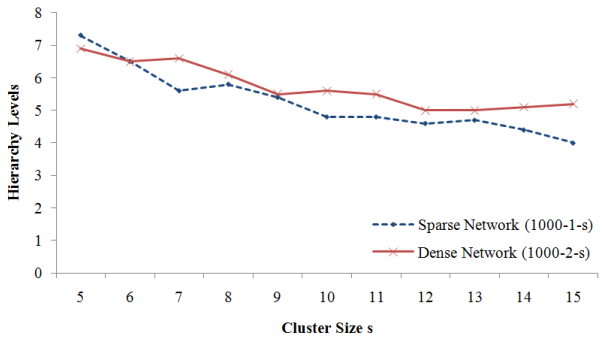

Figure 2: Hierarchy depth with varied cluster size.

parse and respond to a message is negligible (8.38 microseconds on a 2.67 GHz single core lab computer, an average of 312 messages per node on a 5,000 node network, with an average of 4.4 bytes per message). The simulation steps directly relate to the expected runtime on "real" networks if multiplied by the average packet propagation and queuing delay on the network. For example, aggregating a dense network with $3,000$ nodes takes $2,503 \pm 58.3$ simulation steps. If we assume an average propagation and queuing delay of $50ms$, the aggregation takes an average of 125.15 ± 2.92 sec.

[13] remarks that adding random connections in an overlay network increases its robustness and decreases local clustering. ONs, in particular, are very prone to churn, as nodes may join or leave the overlay network at any given time. For this reason, we also evaluate the performance of "dense networks". Dense networks are created like sparse networks, except that every node retains at least one additional connection to random other nodes in the overlay network. The name for each simulation is composed of the number of nodes, the minimum number of random connections per node, and finally the target cluster size. For example, "5000-2-10" refers to a random network graph with 5,000 nodes, with each node having a minimum of two random connections, and a target cluster size of ten. Each simulation is repeated ten times.

Once all clusters are in a finished state and the number of clusters in the uppermost hierarchy level is not larger than s, the simulation is considered complete.

3.2.1 Recommended Cluster Size

The depth of the hierarchy is primarily affected by the cluster size s, as it determines how many nodes are aggregated into a cluster. We keep s small to facilitate complete views and more complex routing algorithms within a cluster. Choosing a s that is too small, however, results in a deep hierarchy, which increases the state every node in the ON needs to store. To reduce the problem space, we first determine a reasonable size for clusters within the hierarchy. Figure 2 shows the hierarchy depth with a cluster size s between 5 and 15. It can be seen that the average hierarchy depths created from spare and dense networks are very similar, and ranges from 4 to 7.3 hierarchy levels. The 95% confidence intervals (0.44 ± 0.11 for sparse; 0.35 ± 0.08 for dense networks) were not included in

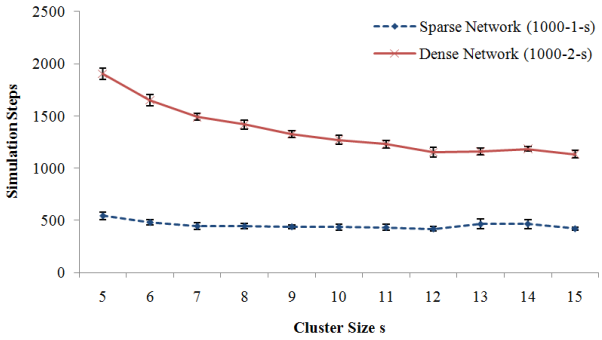

Figure 3: Runtime with varied cluster size.

the graph to preserve clarity. As expected, larger cluster sizes decrease the depth of the hierarchy.

Figure 3 displays the runtime to fully aggregate the overlay network. Dense networks require more iterations, as they execute the "battle phase" of the original algorithm significantly more often than spare networks. Small cluster sizes further increase runtime, because nodes are more likely to exceed $s-1$ neighbors. As described in the previous section, these are broken up into a hub with $s-1$ neighbors, and a virtual node that continues to participate in the partitioning algorithm. As a cluster can contain fewer members for small s, more virtual nodes are introduced, which increases the total number of nodes that need to be aggregated, and hence the hierarchy depth and the runtime.

It can be seen that the runtime and the hierarchy depth decrease quickly if the cluster size is increased to $s = 10$. Further increases only yield little improvements. We therefore recommend a cluster size of $s = 10$, which offers as a good compromise between runtime overhead and hierarchy depth. This means that every node needs to keep track of up to nine other nodes in their cluster. Assuming an average hierarchy depth of $\log_s n$, each node needs to store $\log_s n \cdot (s-1)$ epitomes for our QoS maps.

3.3 Modifications to the Heuristic

In the following, we describe our modifications to the partitioning algorithm in detail:

3.3.1 Stopping Condition

The original algorithm [7] bounds the number of inter-cluster edges, dependent on the network size. This requires knowledge of the total number of nodes in the network graph. Even if we assumed that the network remained static during the measurement, it would be costly to derive this information for large ONs. We regard limiting the number of nodes within a partition as highly relevant. Therefore, we considered introducing a new, simple stopping condition (S) that checks the current partition size against the partition size s. While this works well, we could further reduce the runtime by considering the number of

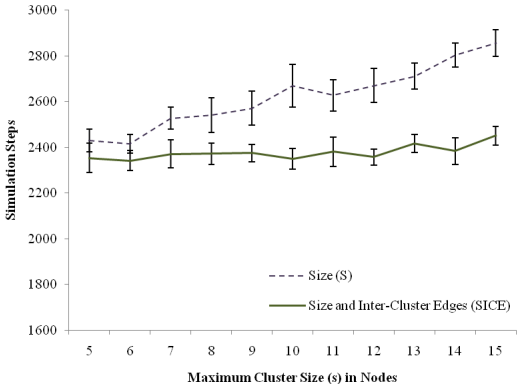

Figure 4: Runtime of different stopping conditions on dense 5,000 node ONs.

inter-cluster edges e in addition to the cluster size (SICE). Figure 4 compares the average simulation steps required to build the QoS map with the two different stopping conditions on a 5,000 node ON.

Limiting the number of inter-cluster edges is also beneficial for QoS maps, as it limits the number of entry and exit points to a cluster. This reduces the complexity of QoS epitomes and the complexity of the aggregation in the next hierarchy level. In our experiments, we found setting $e := s - 1$ reasonable, as it decreases the likelihood of aggregating nodes into hubs with more than $s - 1$ neighbor clusters. Because imposing a hard limit on inter-cluster edges made it impossible to aggregate highly connected networks, we relax this condition for cluster heads that are adding their first layer. In this case, we always add the layer, even if the number of inter-cluster edges then exceeds e. The drawback of this approach is that for highly connected networks, the runtime gain of SICE seen in Figure 4 will be reduced and closer to S. In addition, hubs with more than $s - 1$ neighbor clusters may be created. However, while hubs pose a problem to the partitioning heuristic, we are able to break them up during the partitioning. This is described later in this section.

3.3.2 Multiple Hierarchy Levels

As the algorithm runs fully in parallel, without explicit synchronization, no global knowledge exists of when the current hierarchy level is fully partitioned into clusters, and the next hierarchy level can be built[1]. Any finished cluster immediately begins running the partitioning algorithm in the next highest hierarchy level. Therefore, we must consider that different parts of the network may perform aggregation on different hierarchy levels at the same time. Consider any part of a large network that aggregates itself into clusters on level 0 of the hierarchy. At the same time, another part of the same network may already be aggregated into clusters on level 0, and continues to run the partitioning algorithm on

[1]An exception to this is the uppermost hierarchy level. As all nodes are in the same cluster, they observe that their cluster has no inter-cluster edges, which indicates that the hierarchy has fully been built.

level 1. We therefore include each node's current hierarchy level in every message it sends. Any node that observes a neighbor with a higher hierarchy level considers that neighbor to be part of a finished cluster. This is correct, as a neighbor of a certain hierarchy level is in a final state on all lower hierarchy levels, and would not compete with them in the original algorithm either. Also, a node will not execute the partitioning algorithm until all its neighbors are at least the same hierarchy level as itself. This does not affect the correctness of the algorithm in [7] either, as it uses implicit synchronization and does not rely on message delivery in a bounded time.

3.3.3 Aggregation of Large Hubs

Hubs with more than $s-1$ neighbors pose a problem for the original heuristic. Recall that it partitions the network by adding layers of surrounding nodes to the partition of a potential cluster head. If the first layer already exceeds the partition size bound, it is dropped again. This results in a partition with only a single node. Once the entire network consists of such hubs, no further aggregation is possible. Hubs, if existent, will always affect the heuristic, independent from their identifiers. For example, assume we need to aggregate a network into clusters of up to s nodes. The network may contain hubs with s or more neighbors. Depending on their identifiers, each hub may either be a cluster head, or not. If the cluster head is the hub itself, no layer can be added without exceeding the limit of s nodes per cluster. If, on the other hand, the cluster head is one of the hub's neighbors, one layer can be added. This layer would contain the hub itself, because it is a direct neighbor to the cluster head. As a result, the new cluster would then border the other $s-1$ neighbors of the hub, making it a hub itself in the next hierarchy level.

Note that the random graphs used for our simulations usually do not contain many hubs. Hubs emerge primarily during the aggregation of our network. We did not specifically create networks with large hubs at the lowest level, because the algorithm behaves identically for hubs that already exist at the network level and hubs that emerge during aggregation.

Hubs are very common in ONs, and frequently occur in higher hierarchy levels during aggregation. We considered several approaches to deal with them. First, we attempted to permit adding at least one layer, even if it exceeded the desired cluster size s. We refer to this as Naïve Aggregation (NA). Figures 5 and 6 show this simple approach resulted in low runtime, and low hierarchy depth. However, on our dense networks with $5,000$ nodes and a target cluster size of 10, the average size of the clusters created on the topmost hierarchy level is 170.17 ± 20.624 nodes. Excessively large cluster sizes like these defeat the purpose of aggregation. As such, while yielding the lowest runtime and hierarchy depth of the evaluated approaches, the structure of its hierarchy is not suitable for our QoS maps.

We approach this new problem in two steps. The original algorithm either adds all nodes of a layer, or none. We change the algorithm to perform what we refer to as Partial Explorations (PE): If a node is surrounded by more than $s-1$ neighbors, the cluster head

randomly accepts $s-1$ nodes into the cluster. The remaining nodes are dropped, as in the original algorithm. Afterwards, the cluster is switched to a finished state. Figure 5 shows that the runtime of the algorithm increases, ranging from 17% for 500, to 41% for 5,000 nodes: This is expected, as excess nodes are dropped and continue to participate in the partitioning algorithm. NA would have accepted these nodes, creating very large clusters that are unbounded, which explains its low runtime. In the worst case, NA aggregates the entire network into a single cluster with n nodes. In contrast, PE bounds the cluster size. Even with the increase, it retains the sub-linear runtime of the original algorithm. As an upper bound on the cluster size is crucial for scalability, we argue that this trade-off is worthwhile.

While PE successfully bounds the cluster size, Figure 6 highlights a second problem: The hierarchy depth increases sharply. This is, again, caused by large hubs. As the number of a hub's neighbors is only reduced by up to $s-1$ nodes at each hierarchy level, the hierarchy depth increases significantly for hubs with many neighbors. Consider, for example, a hub with a large number of neighbors. After adding as many neighbors as possible using partial exploration, the cluster containing the hub is finished and no longer participates in the algorithm. Other neighbors that are not part of the cluster are blocked by it, and can only expand in a different direction. It is possible that the number of clusters neighboring the hub is reduced further, if at least two of its neighbors would join the same cluster. However, to maintain a local view, our partial exploration does not consider the network structure beyond the hub, and therefore does not specifically promote this behavior. In the worst case, none of the remaining neighbors can be aggregated. A hub with n neighbors and a maximum cluster size of s therefore increases the height of the hierarchy by $\lceil \frac{n}{s-1} \rceil$ levels.

To cope with this issue, we "break up" hubs by introducing Virtual Nodes (VN): Once a cluster completes a partial exploration, the cluster head splits into the finished cluster, and a virtual node. The virtual node is connected to the cluster, as well as all remaining neighbors, and continues to participate in the algorithm. With this change, hubs are broken up very effectively within a single hierarchy level, at the cost of additional load on the hub. Still, we argue that if such large hubs exist in the network or the hierarchy, they are likely sufficiently powerful to handle the additional load. Figure 5 shows the runtime increases over PE and NA. Again, this is expected, as a hub splitting into two nodes effectively introduces new (albeit virtual) nodes into the network. On the other hand, Figure 6 shows that with VN, the hierarchy depth decreased once more. In fact, with a slight - but still sub-linear - increase in runtime, we have removed the large clusters, and still obtain hierarchy depths only one to two levels deeper than the theoretically optimum of $\lceil \log_s n \rceil$. As a result, by using VN, we are now able to reliably build low-depth hierarchies with clusters that are no larger than the desired cluster size, independent from the underlying network's structure - all important properties for QoS maps.

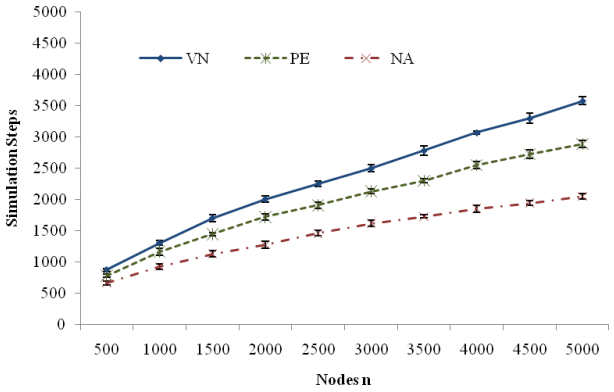

Figure 5: Runtime on random dense ONs; different numbers of nodes.

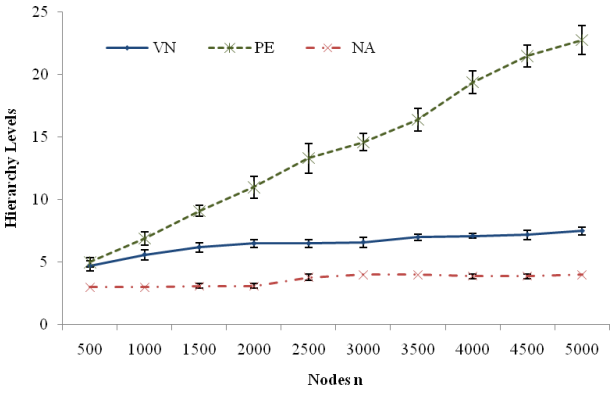

Figure 6: Hierarchy depth on random dense ON; different numbers of nodes.

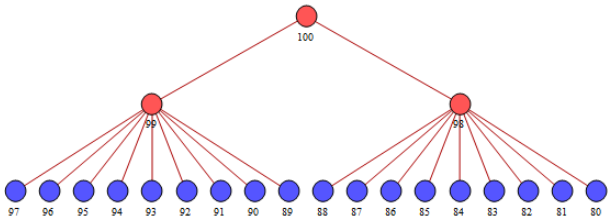

Figure 7: The worst case for aggregating the topology at any level for $s = 3$. The three nodes located at the top of this network form a single cluster, blocking all other nodes from exploring.

3.3.4 Hierarchy Depth and Simulation Steps

The optimal case for the hierarchy depth of a network with n nodes and a cluster size of s is $\lceil log_s n \rceil$. However, as we use a heuristic approach and favor convergence speed, the clustering may be sub-optimal. We measure how deep the hierarchies are, and compare them against the optimal hierarchy size.

It should be noted that the height of the hierarchy, and with it the number of simulation steps, depends on the quality of the clustering. To illustrate the best and the worst case, consider the number of nodes at each hierarchy level, and recall that the hierarchy is complete once the uppermost level consists of at most s clusters. Therefore, the optimal algorithm would reduce the number of nodes on each level as much as possible. As each partition may only contain up to s nodes, it needs to create as few partitions as possible, without exceeding the maximum partition size s. In the following paragraph, n describes the number of nodes the hierarchy has at any given level.

For the optimal case, the algorithm creates $\lceil \frac{n}{s} \rceil$ partitions in each level. As the next level contains only one node for each partition, the number of nodes is reduced by $n - \lceil \frac{n}{s} \rceil$ nodes at each level.

In the worst case, $n - 1$ partitions are created. The partial exploration explained in the previous section guarantees that at least one partition with s nodes is created at each hierarchy level. Therefore, the worst case occurs if one partition with s nodes is created, and all the remaining partitions only contain a single node.

Consider the network in Figure 7 as an input for creating a hierarchy with $s = 3$. The node shown in the top of the figure has the highest identifier, and exactly $s - 1$ neighbors. All remaining nodes have lower identifiers, and are attached to the $s-1$ nodes surrounding the top node. Because of their high identifiers, the top node and its immediate neighbors are aggregated into a single cluster. As they reached the maximum cluster size, they then switch to a finished state. The remaining nodes do not find any direct neighbours that are not part of a finished cluster, and become singleton clusters. Therefore, the number of nodes is reduced by only $s - 1$ in this example. However, even in this worst case, the next higher hierarchy level will consist of one hub node bordering all the other nodes. This hub with all its neighbors is then aggregated into $\lceil \frac{n}{s} \rceil$ clusters, which again represents the

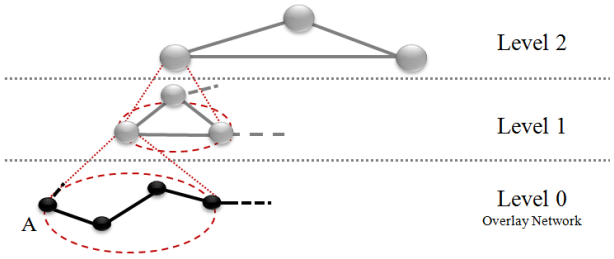

Figure 8: Node A's view of the hierarchy.

optimal case.

In all the other cases, the methods described in the previous section will maximize the number of nodes in a partition. Placement of nodes with high identifiers can still cause partitions to be smaller than the target size, therefore, we expect hierarchies to be slightly higher than the optimal case.

The original partition algorithm itself runs in sub-linear time with regards to network size in the common case, and in linear time in the worst case. Since we apply the partitioning algorithm once for every hierarchy level, we expect a sub-linear time for creating the hierarchy as well. In the worst case, we expect a time complexity close to $n \cdot log_s n$ for creating the hierarchy.

4 QoS Estimation

Once the hierarchy is created, it can be used for efficient routing and QoS estimation in the original ON. In this section we present our approach for QoS estimation. Recall that each node has a local view of the topology, which contains the topology of its own cluster, and the topology of the clusters alongside the hierarchy that contain it. For example, Figure 8 shows the view of node "A" for the topology first shown in Figure 1. To perform QoS estimation and routing, we *additionally* require that each node also knows the identifiers of the clusters that directly border its view. In Figure 8, these would be the identifiers of the clusters the dashed lines lead to. This is reasonable, as that information is already obtained when each cluster is finalized by the partitioning algorithm.

4.1 QoS Epitomes

Whenever a cluster is finalized, QoS epitomes of any desired QoS metric are created. QoS epitomes approximate the QoS experienced for traversing that cluster from any entry node to any exit node. This reduces the amount of information that needs to be advertised to other clusters. While focusing on the concept of QoS maps themselves, we used a simple implementation that stores the QoS from each entry to each exit point of a cluster in a matrix. Common approaches to QoS epitomes such as [21] are likely to reduce the amount of data better and thus further improve our QoS maps. However, a comparison of the

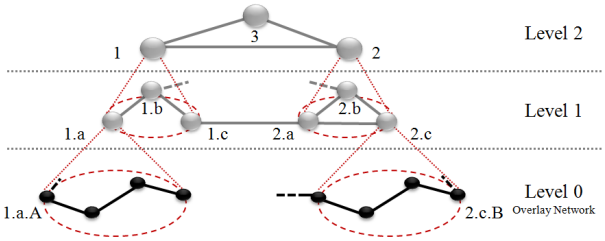

Figure 9: A simple topology with node identifiers.

large amount of algorithms and their pay-off if used for our QoS maps is subject of future work.

Any cluster in our QoS map may be the target cluster of a route. In this case, they do not have an exit node. We therefore compute the worst QoS metrics from each entry to any node in the cluster. As the cluster size is bounded, obtaining this information has little overhead.

The resulting QoS epitomes are included in the topology updates that are sent to the cluster members during aggregation. Therefore, every node also knows the QoS epitomes of each cluster in its local view. To use QoS maps, each node in this hierarchy needs to store $s \cdot (\lceil log_s n \rceil - 1)$ epitomes of its reduced view of the topology with $s \cdot \lceil log_s n \rceil$ nodes.

The topology of the ON is aggregated without considering QoS. Therefore, it may be possible that a cluster contains paths and nodes with vastly different quality. This is intended: QoS maps aim at giving a view (indeed a map) of the landscape of the given ON. We do *not* intend to change the landscape itself. By comparing the range of QoS metrics in the QoS epitomes, they can be used to identify poor topology-awareness in the underlying ON and thus help to find a better one. In any case, our QoS maps allow QoS estimation regardless of how well or poorly the underlay is organized.

4.2 Estimation Algorithm

In this section, we describe how our QoS map can be used to provide QoS estimates with a simple algorithm. A node that wishes to predict the QoS experienced on a path to a target node first uses its locally stored QoS epitomes to get a coarse, low-cost estimate. This estimate can later be refined.

We assume that the node already knows the target node it wants to communicate with, and what part of the hierarchy it is located in. This information can be provided by the application, the ON, or a distributed hashtable. For example, our QoS maps are currently used by a resource reservation algorithm [15]: There, the algorithm provides us with the node and its location. In other cases, the location may be returned together with the node's identifier by the ON. A node's location is specified by concatenating the identifiers of its clusters from the top to the bottom hierarchy level. For example, in Figure 9, the full identifier of node A is "1.a.A", as it is a member of cluster "a", which is itself a member of cluster "1".

To estimate the QoS from a source to a target node, we compute routes on the highest hierarchy level that contains both the cluster of the source and the cluster of the target node. Depending on how much local information is available, we then either use QoS epitomes to make an estimation, or recurse and estimate the QoS at the next lower hierarchy level.

Algorithm 1 Multi-Level QoS estimation from node a to b.

```
1: estimateQoS(a, b) : −
2: l ← getLowestCommonLevel(a, b)
3: r ← getRoute(a, b, l)
4: p ← empty
5: estimate ← getQoSOnLink(empty, empty)
6: for i ← 1 to getRouteLength(r) do
7:    c ← getClusterOnRoute(r, i)
8:    qos_l ← getQoSOnLink(p, c)
9:    if l = 0 then {a and b are in the same cluster}
10:      qos_c ← empty
11:   else if i = 1 then {source-cluster}
12:      b ← findBorderNode(a, c, l − 1)
13:      qos_c ← estimateQoS(a, b)
14:   else if i =getRouteLength(r) then {target-cluster}
15:      qos_c ← getQoSForTraversal(c, p)
16:   else {transit-cluster}
17:      n ← getClusterOnRoute(r, i + 1)
18:      qos_c ← getQoSForTraversal(c, p, n)
19:   end if
20:   estimate ← combineQoS(estimate, qos_l, qos_c)
21:   p ← c
22: end for
23: return estimate
```

Algorithm 1 displays the aforementioned process using pseudo-code. It uses the following high level functions:

- getLowestCommonLevel(a, b) returns a number indicating the lowest hierarchy level l that contains a and b (or any of their parent clusters).
- getRoute(a, b, l) returns an array with cluster identifiers that represents a route r from a to b on hierarchy level l. Any common routing algorithm such as in [8] can be used to compute the route, and to provide alternative routes. As the cluster size is bounded by a constant, the complexity of finding this route is constant as well.
- getRouteLength(r) returns the length of route r in hops.
- getClusterOnRoute(r, i) returns the identifier of the i^{th} cluster in route r.

- findBorderNode(a, b, l) returns a node n in cluster a that borders cluster b on hierarchy level l. As every node in each cluster has full knowledge of its own cluster and its inter-cluster edges, this amounts to local search through a list with a constant number of nodes.

- getQoSOnLink(a, b) returns the most favorable QoS metric of any links connecting a and b. This is the maximum value for restrictive QoS metrics, and the minimum otherwise. If a is "empty", it instead returns an initial value for the estimate. For additive QoS metrics such as delay or jitter, it returns zero. Multiplicative QoS metrics, such as loss probability, are initialized with zero as well. For restrictive QoS metrics (such as bottleneck bandwidth) it returns the maximum possible value representable by the data type used by the implementation.

- combineQoS(e_1, e_2, e_3) combines the QoS estimates e_1 and e_2. If e_3 is not empty, it combines the intermediate result from combining e_1 and e_2 with e_3. In either case, the final result is returned. To combine additive QoS metrics, the individual values are added. For restrictive QoS metrics, the minimum is taken. For loss probability, the independent probabilities of the links are combined.

- getQoSForTraversal(a, b, c) returns an estimate for the QoS needed to traverse cluster a, if entered from cluster b and exited at cluster c. If c is omitted, the worst QoS from the entry point b to any node in the cluster is returned.

getRoute and findBorderNode may return alternative routes. Their existence can be conveyed to the application, which may request their computation, in case it is not satisfied with the first result or wants to explore additional routes.

To estimate the QoS from a source to a target node, we first locate the highest hierarchy level that contains both the cluster of the source and the cluster of the target node by comparing the aforementioned identifiers. As each node knows its own location in the hierarchy, it can now find possible routes to the cluster of the target node on that hierarchy level. Using any common routing algorithm is feasible despite the large number of nodes present in the ON, as the cluster size and thus the number of nodes in the view of each node is bounded on every hierarchy level (and expected to be small).

Each path derived in this way can contain three types of clusters: a source cluster, a target cluster and zero or more transit clusters. Note that source and target clusters are always different from each other, except for the trivial case in which both nodes are in the exact same cluster at the lowest hierarchy level. The three types of clusters differ in the amount of local information the source node has, and are treated differently for QoS prediction purposes:

- *Transit clusters* are traversed on route to the target node. As QoS epitomes for that cluster contain the aggregated QoS from each entry to each exit point, they can be used directly to get a high level estimate of the cost of traversing them.

- *The target cluster* is the least accurate cluster type in the estimation process. As with transit clusters, we consider the entry point. However, as they do not have an exit point, it would be necessary to have a view of the target cluster in order to provide a more precise estimation. Without said view, we have to assume the worst case QoS from its entry point to any node in the cluster.

- *The source cluster*, which contains node A, can be broken down to the next lowest hierarchy level. This increases the quality of the QoS prediction, as it uses less aggregated QoS epitomes. We set the new target node to any exit point we consider, the QoS prediction algorithm is applied recursively. This process guarantees that the next hierarchy level we consider in the recursion will be at least one level lower than the previous one. Level 0, the trivial case in line 10, contains the unaggregated ON of our own cluster that we have a full view of.

The QoS epitomes for each cluster are combined as described. As a result, the source node now has a coarse estimate of the QoS for each path, without consuming any network resources.

There are two ways to increase the accuracy of this estimate. First, recall that we do not have a view of the target cluster. If the target node is cooperative, it can estimate the QoS from each entry point of its cluster to itself as described before, and transmit that information to the source node. At this point, the accuracy of the estimate is only limited by the accuracy of the QoS epitomes themselves.

In addition, with the cooperation from nodes of any transit cluster, its next-lowest hierarchy level can be considered in place of the QoS epitome. As we use less aggregated information of said transit cluster, the accuracy of the estimation improves. This process is repeated for every transit cluster until the desired accuracy is reached. Note that level 0 contains the original ON: Therefore, if the topology of any cluster is traversed down to level 0, it will contain the most accurate and current QoS information for that cluster. This kind of stepwise refinement could be used by a number of applications in a very reasonable way. For example, a Video-on-Demand (VoD) system could, on the most coarse level, find a path, through which delivery can start with a minimal start-up delay. After having started the (batch of) stream(s), it can start a fine-grained QoS estimation in parallel. If a better path is found, the VoD application could switch to this. Clients could be totally unaware of this and be satisfied by fast start-up and good overall quality.

4.3 Complexity Analysis

In order to consider a large number of paths to a target node, a low computational complexity of the estimation algorithm is important. Of the high level functions introduced in the previous section, all but two operate within a cluster. As the number of nodes within a cluster is bounded by the constant s, the complexity of the intra-cluster functions is O(1). *getLowestCommonLevel* in line 2 needs to compare the identifiers of every hierarchy level of the target node, with every hierarchy level of the local node. The identifiers of

the local nodes can be precomputed and stored in a hashtable, for quick retrieval. This is reasonable, as they only change if the position of the local node within the hierarchy itself changes. With this optimization, the lowest common level can be determined in $O(\log n)$ iterations.

The inner loop from lines 6–22 is executed at most s times. This worst case occurs if all nodes within the source cluster are connected serially, and the QoS estimation needs to traverse the entire cluster. Even in this worst case, however, the number of iterations is still a constant. Within the loop, in line 13, the function recurses to obtain more accurate QoS estimations for the clusters it is in. In the worst case, the lowest common level of a and b is the top level of the hierarchy h. This causes the function to recurse h times. If we assume the common case of a hierarchy depth close to $\log_s n$, the runtime complexity of the QoS estimation is $O(\log^2 n)$.

To have sufficient local information for QoS estimation, each node needs to store the identifiers and the epitomes of all nodes in its local view of the hierarchy, as illustrated in Figure 8. On the ON level, level 0, each node stores the identifiers and the QoS metrics of all other $s-1$ nodes of their cluster instead of a QoS epitome. In all remaining hierarchy levels, a node stores up to s identifiers and s QoS epitomes from its local cluster, and up to s identifiers of the neighbor clusters that its own cluster borders (illustrated by dashed lines in Figure 8). Assuming the common case of $\log_s n$ hierarchy levels, each node therefore needs to store $O(\log n)$ identifiers, and $O(\log n)$ epitomes. Since identifiers and epitomes are likely to be very small, this approach scales very well with the number of nodes in the network.

4.3.1 QoS Estimation Example

Consider that in the example network in Figure 9, node 1.a.A wishes to contact node 2.c.B. In order to know its location in the hierarchy, it needs the cluster identifiers "2.c" and "2". Recall that node 1.a.A only has a limited view of the network, as seen in Figure 8. By comparing the cluster identifiers received from node 2.c.B with the identifiers of its own view, it finds that the lowest layer containing a cluster identifier from node 2.c.B is layer 2, which contains cluster 2.c, which in turn contains node 2.c.B. Computing the paths from its own cluster 1 to cluster 2 results in two paths, namely "1-2" and "1-3-2". Node 1.a.A can now use the QoS epitome of cluster 3 to get a QoS estimate for traversing this cluster. As it does not know the structure of cluster 2, the worst case of having to traverse this cluster as well has to be assumed. Finally, as it is part of cluster 1 and knows its lower hierarchy level structures, it can estimate the QoS from itself to the nodes bordering clusters 2 and 3 by recursively applying the algorithm.

The QoS epitomes for each cluster can then be combined, respecting the different QoS metrics: Delay is added, loss is multiplied, and for bottleneck bandwidth, a restrictive QoS metric, the minimum is taken. As a result, node 1.a.A now has a coarse estimate of the QoS to expect for each path, without consuming any network resources.

There are two ways to increase the accuracy of this estimate. First, recall that we

assumed we needed to traverse cluster 2 in our example. If node 2.c.B is cooperative, it can estimate the QoS from the entry point of cluster 2 to itself as described before, and transmit that information to node 1.a.A. At this point, the accuracy of the estimate is only limited by the accuracy of the QoS epitomes themselves.

To achieve a yet even greater accuracy, the topology of the next-lowest hierarchy level can be considered in place of a QoS epitome. This improves the accuracy, as the QoS epitome aggregates that topology and reduces information. In Figure 9, node 1.a.A only has a coarse QoS epitome aggregating the entire cluster 3. With the cooperation of any node from cluster 3, it can obtain an accurate view of this cluster and increase the quality of the QoS estimate. This process is repeated, for example, with clusters 1.b and 1.c, until the desired accuracy is reached. It should be noted that no aggregation is performed at level 0. Therefore, if the topology of any cluster is traversed down to level 0, it will contain the most accurate and current QoS information for that cluster.

To find a node from cluster 3 to cooperate with, node 1.a.A invokes the routing algorithm with the cluster identifier 3. The routing algorithm is described in the following section.

4.4 Routing Algorithm

Similar to the previous section, we assume that the source node (node a) already knows the cluster identifiers of the target node (node b) it wants to communicate with. These cluster identifiers can be included in the data packet, to pass this information to intermediate nodes as well. Each node uses the hierarchical aggregation to compute the next hop to the target node, again starting at the highest common hierarchy level. The application may wish to use a specific route, based on its QoS estimation. In that case, the high level route should be included in the data packet as well. To reduce the message size and the time needed for routing subsequent packets, techniques such as DLDS may be used [14]. The routing is performed as shown in Algorithm 2.

After running the routing algorithm, node a knows the next nodes in its cluster to route the packet to, as well as a high level route through the network outside its cluster. This information may be stored in a routing table, as well as included in the packet itself, in order to further improve routing performance.

4.4.1 Complexity Analysis

In the worst case, the lowest common hierarchy level is the top of the hierarchy itself. This level consists of only a single cluster that contains all other clusters, and is therefore common to all nodes. After finding the lowest common hierarchy level, the algorithm moves one hierarchy level lower, and sets the new target to the node in its own cluster that borders the next cluster closer to the target node. Therefore, if s is the maximum cluster size and n the number of nodes in the overlay network, the outer loop of the algorithm executes $\log_s n$ times (1). Similar to the QoS estimation, `getLowestCommonLevel` needs

Algorithm 2 Multi-Level Routing from node a to b on hierarchy level l.

```
1: computeRoute(a, b, l) : −
2: repeat
3:    l ← getLowestCommonLevel(a, b)
4:    r ← getRoute(a, b, l)
5:    n ← getClusterOnRoute(r, 1)
6:    if l > 0 then
7:       repeat
8:          l ← l − 1
9:          b ← findBorderNode(a, n, l)
10:         if b = a then
11:            n ← getIdentifier(n, l)
12:         end if
13:      until b ≠ a or l = 0
14:   end if
15: until l = 0
16: return n
```

to perform up to $\log_s n$ iterations (2) to determine the lowest common hierarchy level. All other procedures have constant complexity. The runtime complexity of our routing algorithm is therefore identical to the runtime complexity of the QoS estimation algorithm:

$$\textit{Runtime Complexity} = \overbrace{\underbrace{\log_s n}_{(1)} \cdot \underbrace{\log_s n}_{(2)}}^{\log_s^2 n} = O\left(\log^2 n\right)$$

4.4.2 Routing Example

Consider again Figure 9, in which node 1.a.A wishes to contact node 2.c.B. *target* is set to 2.c.B. By exchanging cluster identifiers, node 1.a.A finds that 2.c.B is in a different cluster, and that the lowest level of the hierarchy with a cluster identifier from node 2.c.B is level 2, containing cluster "2". It now invokes a routing algorithm to find the next cluster closer to "2". For the sake of simplicity, assume that the routing algorithm returns cluster "2", and not the route over cluster "3". Node 1.a.A now moves one hierarchy level lower, to level 1, and sets *target* to cluster 1.c, as it borders cluster "2". As *target* is again in a different cluster than itself, node 1.a.A invokes the routing algorithm on level 1, which returns node 1.b. As before, node 1.a.A moves one hierarchy level lower, to level 0, to find the node bordering 1.b. As this is its own cluster, it has complete information on it. It finds that it itself is bordering node 1.b, and thus relays the packet to node in 1.b. The node in 1.b has complete information on its own network, and therefore can transmit the packet to 1.c. Likewise, the node in cluster 1.c knows which nodes are bordering cluster 2, and pass the packet to 2.a. 2.a relays the packet to the cluster 2.c, which is the cluster node 2.c.B resides in.

4.5 Conclusion

The multi-level QoS estimation algorithm introduced in this section uses our hierarchical aggregation in combination with QoS epitomes. This allows us to predict QoS on any path between two nodes while using only local information. With a low runtime complexity of $O(\log^2 n)$, a large number of routes can be evaluated in little time. This allows an application to chose the most beneficial route from many alternative routes. Since the QoS estimation algorithm starts out at the highest hierarchy level, the initial alternative routes cover a large part of the overlay network. Route changes become more and more localized, as lower layers are evaluated in greater detail and infeasible routes are discarded.

5 Conclusion and Future Work

We introduced QoS maps, a decentralized hierarchical approach to perform scalable QoS estimation in large scale ONs. We have shown that by modifying a decentralized version of the Basic Partition algorithm [7], we are able to aggregate the network topology into a hierarchy with small cluster sizes and low depth. We introduced a multi-level QoS estimation algorithm that uses QoS maps in combination with QoS epitomes. With it, we predict QoS on any path between two nodes starting with only local information. A large number of routes can be evaluated in little time, due to the low runtime complexity of $O(\log^2 n)$. This allows an application to chose the most beneficial route from many alternative routes. Since the QoS estimation algorithm starts out at the highest hierarchy level, the initial alternative routes cover large parts of the ON. Route changes become more and more localized, as lower layers are evaluated in greater detail, and infeasible routes are discarded. A first application uses our QoS maps to determine underutilized routes and perform resource reservation in large ONs. Current experiments show promising results [15].

There are a number of points subject to further research: First, the partitioning algorithm, and hence the topology aggregation introduced in this paper are not resilient against failure. A node that leaves the network during the partitioning causes the partitioning algorithm to stall, as nodes are indirectly synchronized with each other. A possible solution would be to restart the partitioning algorithm for clusters that have stalled. Also, reaggregation needs to be considered: The aggregated information of the QoS epitomes will become stale over time, as the QoS of any cluster changes. Reaggregation of QoS epitomes is a well studied problem, and considerable work exists of how and how often reaggregation should be performed [21]. For topology changes and churn, we currently re-run the partitioning algorithm for the affected part of the topology. This may not always be necessary: Our assumption is that nodes leaving the ON might only affect few clusters, in particular if its cluster's QoS epitome is not affected by the departing node. By responding with small, localized changes to the hierarchy - such as merging sparse and splitting large clusters, as performed by [3] - the amount of reaggregation can be minimized. However, the implementation and evaluation of this approach for our QoS

maps is left for future work.

References

[1] S. S. Aman, M. R. Akbarzadeh-Totonchi, and M. Naghib zadeh. A novel approach to distributed routing by Super-AntNet. In *Proc. IEEE CEC 2008*, pages 2151–2157, 2008.

[2] B. Awerbuch and D. Peleg. Sparse partitions. In *IEEE Symposium on Foundations of Computer Science*, pages 503–513. IEEE, 1990.

[3] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. *SIGCOMM Computer Communication Review*, 32:205–217, August 2002.

[4] S. Basagni. Distributed clustering for ad hoc networks. In *Proc. 4th Int. Symposium on Parallel Architectures, Algorithms, and Networks, I-SPAN '99*, pages 310–315, 1999.

[5] J. Beal. Leaderless distributed hierarchy formation. *MIT AI Lab Memo*, 2002.

[6] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: high-bandwidth multicast in cooperative environments. *SIGOPS 2003*, 37:298–313, October 2003.

[7] B. Derbel, M. Mosbah, and A. Zemmari. Sublinear fully distributed partition with applications. *Theory Comput. Syst.*, 47(2):368–404, 2010.

[8] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[9] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *Proc. 6th Annual ACM Symposium STOC '74*, pages 47–63, New York, NY, USA, 1974. ACM.

[10] R. Hou, K.-S. Lui, K.-C. Leung, and F. Baker. Routing with QoS information aggregation in hierarchical networks. In *17th Int. Workshop on Quality of Service*, pages 1–9, July 2009.

[11] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr. Overcast: reliable multicasting with on overlay network. In *Proc. 4th Symposium on Operating System Design & Implementation*, OSDI'00, pages 14–30, CA, USA, 2000. USENIX Association.

[12] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, 1998.

[13] J. Leitão, J. Pereira, and L. Rodrigues. Topology aware gossip overlays. Technical report, INESC-ID, January 2008.

[14] P. Montessoro. Distributed linked data structures for efficient access to information within routers. In *Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2010 Int. Congress on*, pages 335 –342, October 2010.

[15] P. L. Montessoro, S. Wieser, and L. Böszörmenyi. An efficient and scalable data-structure for resource reservation and fast packet forwarding in large scale multimedia overlay networks. Submitted for publication.

[16] F. Pellegrini and J.-H. Her. Efficient and scalable parallel graph partitioning. In *5th Int. Workshop PMAA'08*, Neuchâtel Suisse, 2008.

[17] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Proc. 3rd Int. COST264 Workshop NGC '01*, pages 30–43, London, UK, 2001. Springer-Verlag.

[18] A. Sobe, W. Elmenreich, and L. Böszörmenyi. Towards a self-organizing replication model for non-sequential media access. In *Proc. ACM Multimedia 2010 Workshop SAPMIA 2010*. ACM, 2010.

[19] W. Y. Tam, K. S. Lui, S. Uludag, and K. Nahrstedt. Quality-of-service routing with path information aggregation. *Comput. Networks*, 51(12):3574–3594, 2007.

[20] D. G. Thaler and C. V. Ravishankar. Distributed top-down hierarchy construction. In *Proc. IEEE INFOCOM '98*, pages 693–701, 1998.

[21] S. Uludag, K.-S. Lui, K. Nahrstedt, and G. Brewster. Analysis of topology aggregation techniques for QoS routing. *ACM Comput. Surveys*, 39(3), 2007.

[22] Y. Wan, S. Roy, A. Saberi, and B. Lesieutre. A stochastic automaton-based algorithm for flexible and distributed network partitioning. In *Proc. SIS 2005*, pages 273–280, 2005.

[23] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. 11th Int. Workshop NOSSDAV '01*, pages 11–20, New York, USA, 2001. ACM.

Institute of Information Technology
University Klagenfurt
Universitaetsstr. 65-67
A-9020 Klagenfurt
Austria

http://www-itec.uni-klu.ac.at

University Klagenfurt