Efficient Precomputation of Quality-of-Service Routes

Anees Shaikh†, Jennifer Rexford‡, and Kang G. Shin†

†Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122
{ashaikh,kgshin}@eecs.umich.edu

‡Network Mathematics Research
Networking and Distributed Systems
AT&T Labs – Research
Florham Park, NJ 07932-0971
jrex@research.att.com

Abstract

Quality-of-service (QoS) routing satisfies application performance requirements and optimizes network resource usage by selecting paths based on connection traffic parameters and available link capacity. However, QoS-routing protocols can introduce significant network overhead for computing routes and distributing information about link load. These issues must be carefully addressed before deploying QoS-routing protocols in large backbone networks. Route precomputation is an effective way to amortize the cost of the path-selection algorithm over multiple connection requests, at the risk of selecting suboptimal routes. This paper introduces efficient mechanisms for precomputing one or more routes to each destination, and on-demand checking of the suitability of the routes at connection arrival, based on the most recent link-state information. The algorithm uses a small extension of the traditional Dijkstra shortest-path computation to precompute a minimum-cost graph, and an on-demand depth-first search to extract a feasible route for each connection. As part of the extraction process, the algorithm can rerank the routes in the minimum-cost graph to improve the routing choices for subsequent connections. Simulation experiments show that the route precomputation and extraction techniques are effective at lowering the computational overheads for QoS routing, while achieving performance similar to the more expensive on-demand path-selection schemes.

1 Introduction

The success of distributed audio and video applications hinges on having predictable performance in the underlying communication network. The network can provide throughput and delay guarantees by reserving resources for individual connections or flows. The routing algorithm plays a pivotal role in this process by locating paths that can satisfy the performance requirements of arriving connections. This is particularly important for handling high-bandwidth multimedia streams, which often consume a relatively large fraction of the link capacity. Quality-of-service routing has the potential to optimize the usage of network resources, and increase the likelihood of accepting new connections, by selecting paths based on existing network load and connection traffic parameters [1–3]. However, distributing link load information and computing routes for new connections can consume considerable bandwidth, memory, and processing resources. Controlling these overheads in large backbone networks requires a careful trade-off between accuracy and complexity. In this paper, we present an efficient path-selection scheme that precomputes routes, while still capitalizing on the most recent network load information available at connection arrival.

Our study focuses on link-state routing algorithms where the source router or switch selects a path based on connection throughput requirements and the available resources in the network. For example, the

ATM Forum's PNNI standard [4] defines a routing protocol for distributing topology and load information throughout the network, and a signalling protocol for processing and forwarding connection-establishment requests from the source. Similarly, proposed QoS extensions to the OSPF protocol include an "explicit routing" mechanism for source-directed IP routing [5, 6]. Each switch maintains its own view of the available link resources, distributes link-state information to other switches, and selects routes for new connections. To improve the scalability of these protocols in large configurations, the switches and links can be assigned to smaller peer groups or areas that exchange detailed link-state information. In this paper, we focus on reducing the overheads of QoS routing within a single peer group, though the algorithms apply to the general case of hierarchical networks. We consider the case of large, relatively well-connected, backbone topologies that offer the possibility of multiple routes between source-destination pairs. In addition, we compare performance on more sparsely-connected networks that provide fewer routing options.

Efficient QoS routing requires effective techniques for computing routes and exchanging link-state information. Link state is typically propagated in a periodic fashion, or in response to a significant change in available capacity. For example, a link may advertise its available bandwidth metric whenever it changes by more than 10% since the previous update message; in addition, a minimum time between update messages is often imposed to avoid excessive link-state update traffic. When a connection request arrives, the source typically computes a suitable route, using the most recent link-state information, and initiates hop-by-hop signalling to reserve resources along the path to the destination. The set-up attempt fails if one or more of the links no longer has sufficient resources to accept the new connection. After such a signalling failure, the source may decide to try a new route, that excludes the offending link, or may simply block the connection. These route computations and signalling attempts consume processing resources at the switches, and introduce set-up latency for each accepted connection. Minimizing these overheads is particularly important during periods of transient overload, such as bursty connection arrivals or rerouting of traffic after a link failure.

Most previous research on QoS routing has investigated *on-demand* policies that compute a path at connection arrival. Recent work considers *precomputation* or *path caching* schemes that attempt to amortize the overheads of route computation by reusing the paths for multiple connection requests [6–10]. Path precomputation introduces a trade-off between processing overheads and the quality of the routing decisions. Previous work on precomputed routes has focused on quantifying this trade-off and developing guidelines for when to recompute routes. We extend this work by presenting efficient *mechanisms* for precomputing paths under source-directed link-state routing. In particular, this paper addresses several important practical questions:

- How should the precomputed routes be stored?
- How should multiple routes be computed?
- How much work should be performed at connection arrival?
- How should routing and signalling overheads be limited?

We answer these questions with four key elements that are described in detail in Section 2:

- *Coarse-grain link costs:* Path selection is based on link-cost metrics, which are a function of link-state information. Limiting link costs to a small number of values reduces the computational complexity of the path-selection algorithm. Coarse-grain link costs do not significantly degrade performance, and increase the likelihood of having more than one minimum-cost route to a destination.

- *Precomputation of minimum-cost graph:* Each switch or router precomputes a compact data structure that stores all minimum-cost routes to each destination. A small modification to Dijkstra's shortest-path algorithm can locate all minimum-cost routes to each destination. Instead of storing the precomputed paths in a cache or table, route extraction is postponed until connection arrival.

- *Route extraction with feasibility check:* As part of extracting a route, the source checks the feasibility of each link, based on the most recent link-state information and the bandwidth requirement of the new connection. The algorithm performs a depth-first search through the Dijkstra data structure, extracting the first route in the common case. Then, the source initiates signalling in the network to reserve resources along the selected path.

- *Reranking of multiple routes:* The depth-first extraction algorithm imposes an implicit ordering of the links when a node appears in multiple minimum-cost paths. As part of route extraction, the source can rerank these links to improve the path-selection process for the next connection. This provides a simple framework for a number of alternate-routing policies.

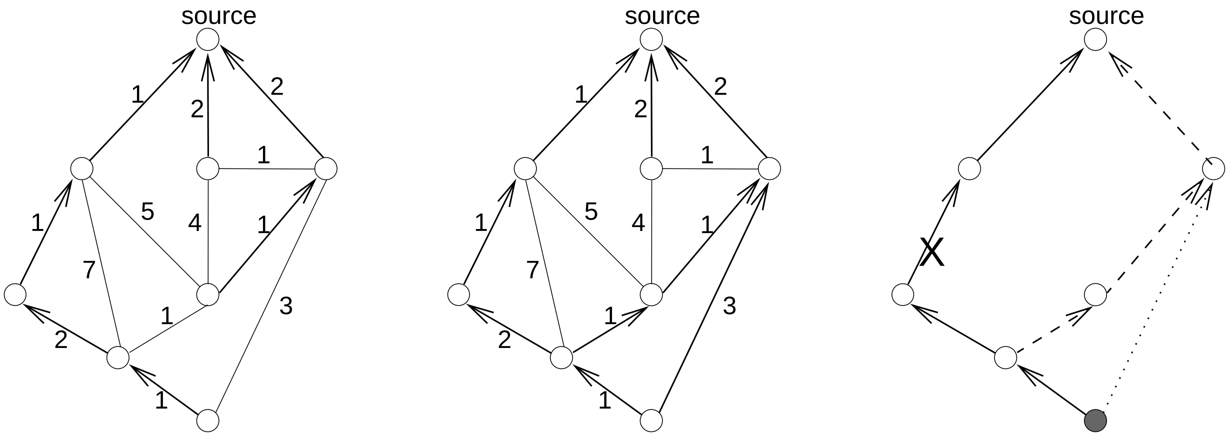
These mechanisms enable a wide range of policies for when to compute new routes, how many candidate routes to try for a new connection, and how often to update link-state information. Coupled with our efficient routing mechanisms, these policy decisions allow the network to significantly reduce processing overheads and set-up delay, in comparison to traditional on-demand algorithms. In addition, the simulation experiments in Section 3 show that the feasibility check, and the potential for multiple candidate routes, results in lower likelihood of rejecting requests and lower signalling overheads, in relation to other precomputation schemes. Section 4 compares our approach to related work on QoS route precomputation, and Section 5 concludes the paper with a discussion of future research directions.

2 Precomputation of QoS Routes

Reducing the overheads of route computation requires careful consideration about how much information and processing are involved on various time scales. The source receives the most accurate information about network load and connection resource requirements upon the arrival of new link-state updates and connection requests. To lower complexity, though, our precomputation scheme does not perform any work upon receiving a link-state message, beyond recording the new load information, and only modest work on connection arrival. Instead, most of the processing is relegated to background computation of a shortest-path graph of routes to each destination. By delaying route extraction until connection arrival, we exploit the most recent link-state to find a suitable path while incurring only modest overhead. Allowing occasional triggered recomputation for individual connection requests further improves the ability to find a path and to select better routes for future connections.

2.1 Compact Storage of Precomputed Routes

Our study of path precomputation focuses on intra-domain routing in large, flat networks with N nodes (switches or routers) and L links. In this context, we focus on topologies with relatively high connectivity, an increasingly common feature of emerging core backbone networks [11, 12], and on multimedia traffic



(a) Shortest-path graph (b) All-shortest-path graph (c) Route extraction

Figure 1: **Delayed Pruning of Shortest-Path Routes:** The left figure shows a network graph with a link-cost metric on each edge. Each node has a parent pointer to the upstream node along a minimum-cost path from the source. By allowing multiple parent pointers at each node, the graph can represent all of the minimum-cost paths for each node, as shown in the center figure. These parent pointers allow efficient extraction of a route for a specific destination, as shown in the right figure. If a link appears infeasible (denoted by the "X"), an alternate minimum-cost route (shown by dashed lines) can be extracted through a search along the parent pointers.

that requires throughput guarantees. Each switch knows the underlying topology and has (possibly out-of-date) information about the unreserved bandwidth on each link. Link state is flooded periodically, or in response to a significant change in available bandwidth, ensuring that the switches have fairly accurate knowledge of network load. The source switch selects a route for an arriving connection, based on the link state and the connection's bandwidth requirement. Route computation is based on the Dijkstra shortest-path algorithm [13], where link cost (or "distance") is a function of the link load. To minimize resource requirements and end-to-end delay, we focus on link-cost functions that favor routes with a small number of links.

The Dijkstra shortest-path algorithm computes a route to a destination node in $O(L \log N)$ time, when implemented with a binary heap [13]. Although advanced data structures can reduce the average and worst-case complexity [14], the shortest-path computation still incurs significant overhead in large networks. In computing a route to each destination, the Dijkstra algorithm generates a shortest-path graph, where each node has a parent pointer to the upstream node in its route from the source, as shown in Figure 1(a). Extracting a route to a particular destination involves traversing these parent pointers, and introduces complexity in proportion to the path length. For on-demand routing of a single connection, the construction of the shortest-path graph terminates upon reaching the destination. Continuing the computation to generate a route for every destination does not significantly increase the processing requirements, and the complexity remains $O(L \log N)$. This allows path precomputation schemes to amortize the overhead of the shortest-path calculation over multiple destination nodes.

Path precomputation schemes typically store one or more routes in a cache or table. In hop-by-hop routing, the router stores only the next hop of the route to each destination, allowing for simple storage in a route table. In contrast, source-directed routing requires the source to maintain a variable-length list of the links on the path to the destination. This list (or stack) becomes part of the signalling message that establishes a connection (e.g., a "designated transit list" in PNNI, or an "explicit route advertisement" in the QoS extensions to OSPF), instructing each intermediate node to forward the message to the next link on the route. To precompute routes, the source could generate paths to one or more destinations and store each route list in a cache, occasionally recomputing one or more routes based on the most recent link-state information. However, computing and extracting multiple routes introduces computational and storage complexity. In addition, since link-state information changes over time, these cached routes must be invalidated and/or recomputed periodically. Instead of caching paths in a separate data structure, we consider a compact representation of the variable-length routes directly in the shortest-path graph, as in Figure 1(a).

2.2 Precomputation of Multiple Minimum-Cost Routes

Path precomputation schemes benefit from having multiple candidate routes to each destination, to balance network load and have additional routing choices in case of a signalling failure. However, the multiple routes should have similar cost, to avoid selecting paths that make inefficient use of network resources. A switch could conceivably precompute the $k > 1$ shortest paths (in terms of hop-count or other cost) to each destination. Alternatively, a routing algorithm could compute all paths within some additive or multiplicative factor ϵ of the best path. However, these approaches introduce considerable computational complexity. For example, computing the k shortest paths for a single destination in a directed graph has complexity as high as $O(kN^3)$ [15, 16]. In addition, the k shortest paths (or paths within ϵ of optimal) to one node may not be part of the best routes to other destinations; hence, it is not usually possible to store these multiple routes in a compact, shortest-path graph representation.

Instead, we focus on an efficient special case of computing multiple *equal-cost* paths to each destination. This formulation permits a compact representation of the multiple routes, as shown in Figure 1(b). Computing all minimum-cost routes requires a small modification to the traditional Dijkstra computation to store multiple parent pointers for each node; each pointer identifies an upstream node along a shortest path to the destination. The nodes and parent pointers form a directed, acyclic subgraph of the original graph, rooted at the source switch. Each node in the graph has a circular list of its parent pointers (to facilitate simple traversal), or a bit-mask to indicate which upstream nodes reside on shortest-path routes. Figure 2 summarizes the algorithm, which is similar to a traditional Dijkstra computation, except for the addition of lines 4–5. The algorithm uses a heap to visit the nodes in order of their distance from the source, and relaxes each outgoing link. A parent pointer is assigned if a link is on a minimum-cost path (so far) to the neighboring node. To minimize the storage requirements and complexity of route extraction, the algorithm can impose a limit on the number of parents for each node (e.g., 2 or 3).

The likelihood of having multiple minimum-cost routes to a destination depends on the link cost function and the underlying network topology. To increase the chance of "ties" in the route computation, we limit the link costs to a small number of discrete values (say, $C = 5$ or $C = 10$). Although fine-grain link costs (larger values of C) usually result in lower blocking probabilities, a moderately coarse link-cost function does not significantly degrade performance, as shown in Section 3. Moreover, when link-state information is somewhat out-of-date, the benefit of a fine-grain link cost function is greatly diminished.

```
Heap = set of all N nodes;
while (Heap is not empty) {                          # visit each node
    u = PopMin(Heap);
    foreach node v ∈ Neighbors(u) {                  # relax each link
       if (dist[u] + cost[u,v] == dist[v])           # equal route
         parent[v] = parent[v] ∪ {u};
       else if (dist[u] + cost[u,v] < dist[v]) {     # cheaper route
         parent[v] = {u};
         dist[v] = dist[u] + cost[u,v];
       }
    }
}
```

Figure 2: **Dijkstra algorithm with multiple parent pointers:** Initially, each node `u` has distance `dist[u]`$=\infty$, except for the source which has a distance of 0. Starting with the source, the algorithm visits the node u with the smallest distance and considers the cost `cost[u,v]` of the link to each neighboring node `v`, extending or reassigning the set of parent pointers `parent[v]` if the new route has equal or lower cost, respectively. At the end, each connected node has one or more parent pointers to upstream nodes along minimum-cost routes from the source.

Perhaps more importantly, coarse-grain link costs reduce the processing requirements of the shortest-path computation by reducing heap complexity. The complexity of Dijkstra's algorithm, and the variation in Figure 2, decreases from $O(L \log N)$ to $O(L + CN)$ [13], while more advanced data structures offer even further reduction [14]. Hence, these coarse-grain link costs become particularly appropriate in large, well-connected networks (large L and N) where the switches have out-of-date link-state information.

2.3 Delayed Extraction of Precomputed Routes

Rather than extracting and caching routes in a separate data structure, we store the precomputed routes in the shortest-path graph and delay the extraction operation until a connection request arrives. During the path extraction the source applies the most recent link-state information in selecting a route. The extraction process operates on the subgraph of nodes and parent pointers along minimum-cost routes to the destination, as shown in Figure 1(c). Though the switch could conceivably run a new Dijkstra computation on this subgraph to select the "best" precomputed route, we instead optimize for the common case of extracting the first route by following a single set of parent pointers (e.g., the leftmost path in Figure 1(c)). More generally, we perform a depth-first search through the reduced graph to extract the first *feasible* route, based on the current link-state information and the bandwidth requirement of the new connection. If the extraction process encounters a link that does not (appear to) have enough available bandwidth, the algorithm backtracks and tries a different parent pointer.

The depth-first search and the feasibility check effectively "simulate" hop-by-hop signalling, using the most recent link-state information. Note that this operation is purely local at the source, and much less costly than discovering an infeasible link by sending and processing signalling messages in the network. Starting at the destination node, the algorithm builds a stack of the nodes in the route from the source, as shown in Figure 3. Each node has a circular list of parent pointers, starting with the `head` parent; a

```
 1:  push(dest);                                # start at the destination node
 2:  ptr[dest] = head[dest];                    # start at dest's first parent
 3:  while ((top != NULL) and (top != src)) {   # continue until reaching src
 4:     new = ptr[top].nodeid;                  # explore parents of top node
 5:     if (feasible(new, top)) {
 6:        push(new);                           # explore next node in route
 7:        ptr[new] = head[new];
 8:     }
 9:     else {
10:        while (ptr[top].next == head[top])
11:           pop();                            # backtrack from exhausted node(s)
12:        if (top != NULL)
13:           ptr[top] = ptr[top].next;         # sequence to next parent pointer
14:     }
15:  }
```

Figure 3: **Depth-first route extraction:** Starting at the destination node (`dest`), the algorithm performs a depth-first traversal of parent pointers to construct a stack of the nodes in the route. By the end of the algorithm, the top of the stack points to the source (`src`) if a feasible route exists; otherwise, the stack is empty (`top == NULL`). The algorithm explores multiple parent pointers by sequencing through a circular list, starting with the `head` parent.

second pointer `ptr` is used to sequence through the list of one or more parents, until `ptr` cycles back to `head`. Each iteration of the `while` loop considers the link between the node at the top of the stack and its current parent (pointed to by the `ptr` pointer and denoted by `new` in line 4). If the link is feasible, `new` is added to the stack and its parents are considered in the next iteration (lines 5–8). Otherwise, if the feasibility check fails, we proceed to the next parent, or backtrack until we find a node whose parent list has not been exhausted (lines 10–13). The algorithm terminates when we reach the source via a feasible path (`top == src`), or when the path stack becomes empty (`top == NULL`) because no feasible path was found. In the former case, the route is read by popping the stack from source to destination.

A potential drawback of this approach is that a subsequent connection request may have to duplicate some of the same backtracking as the preceding request, particularly since the traversal always starts with the parent marked by the `head` pointer. We can avoid this problem by performing some simple pointer manipulations at the end of a successful route extraction. As we pop each node in the route from the path stack, we may alter the position of its `head` pointer so that a subsequent extraction attempt for the same destination (or any destination along the path) visits a different set of links. Three possibilities are:

- Leave parents in existing order (do nothing).
- Make the selected parent the new head (`head[node] = ptr[node]`). This amounts essentially to "sticky" routing where we stay with the first route that was found to be feasible in the last extraction.
- Round-robin to the next parent (`head[node] = ptr[node].next`). This policy attempts to balance load by alternating the links that carry traffic for a set of destinations.

These alternation policies differ from traditional alternate routing, where the switch rotates among a set of cached *paths*. Here, the rotation for one destination node influences the order in which links are visited

Policy	Description
on-demand or precomputed	recompute routes for every request or try first to extract from the existing shortest-path graph
allow multiple routes	limit extraction to just one route instead of allowing backtracking to find alternate routes
feasibility checking	whether to use feasibility checking during route extraction
re-rank multiple routes	policy to arrange `head` pointers in the shortest-path graph during route extraction (none, sticky, round-robin)
periodic recomputations	specification of a background route computation frequency
recompute on routing failure	trigger recomputation when initial route extraction fails
recompute on signalling failure	trigger recomputation on signalling failure
reextract on signalling failure	try to reextract another feasible route (rather than recompute) when signalling fails
prune on recomputation	whether to omit the links that failed the admission test when recomputing after a signalling failure
maximum signalling attempts	limit the number of times each connection request may attempt signalling

Table 1: **Routing and signalling policy options:** This table summarizes some possible policies for connection request handling. The choice of policies may be dictated by objectives such as minimizing processing overhead, connection set-up delay, or blocking probability.

for other destination nodes along the path. Moreover, changing the position of the `head` pointer is simpler than extracting and rotating entire paths, and may actually provide added benefit because the alternate routing is performed on a more global scale.

2.4 Route Computation Policy Options

The path precomputation and extraction algorithms provide a useful framework for computing, storing, and selecting from multiple quality-of-service routes. These techniques provide significant latitude in handling individual connection requests, depending on the inaccuracy of the link-state information, connectivity of the underlying network, and tolerance to set-up delay. Table 1 summarizes the key policy decisions, starting with four options discussed in the previous subsections. The remaining design decisions concern how and when to recompute routes and initiate signalling for new connections. The simplest approach relies entirely on a periodic, background computation of the shortest-path graph. When a request arrives, the source extracts a route and initiates signalling. The source blocks the request if the extraction process does not produce a route, and connection set-up delay is determined almost entirely by signalling delay. Periodic recomputation also simplifies CPU provisioning, at the expense of higher connection blocking probabilities and the selection of suboptimal routes for other connections.

Instead, the source could trigger recomputation of the shortest-path graph after a failure in route extraction or connection signalling. Recomputing routes with the most recent link-state information would reduce connection blocking, at the expense of additional processing load and set-up delay. Inaccurate link state or rapidly arriving connection requests may require frequent path recomputation, introducing computation overhead comparable to on-demand routing. To bound set-up delay and limit processing overheads, the source can impose a maximum number of route computations and signalling attempts for each connection. Since the source recomputes the entire shortest-path graph, the overhead is amortized over all destinations and can benefit subsequent connection requests. To bound the worst-case computational load, the source can impose a minimum time between route recomputations, even in the presence of routing and signalling failures.

When routes are recomputed after a signalling failure, the source can choose to omit the link that failed the admission test. In effect, the signalling failure provides more recent information about the load on the offending link. Pruning this link from the graph is particularly important if the source attempts to compute and signal a new route for the same connection. This new route should avoid using the link that caused the previous failure. In addition, pruning the heavily-loaded link is useful for future connection arrivals, particularly if the shortest-path graph is dedicated to connections with the same (or similar) bandwidth requirements. When connections have more diverse quality-of-service parameters, the source can support a small number of different bandwidth classes (e.g., audio and video), with separate precomputed routes that are tailored to the performance requirements [6, 7]. Employing a different link-cost function and path computation policy to each class enhances the network's ability to route high-bandwidth traffic.

3 Performance Evaluation

In this section, we evaluate the proposed routing algorithm under a range of recomputation periods and link-state update policies. The experiments show that precomputation of the minimum-cost graph, coupled with a feasibility check, approximates the good performance of on-demand routing and the low computational overhead of path caching. The feasibility check is especially effective in reducing the likelihood of expensive signalling failures, even when link-state information is somewhat out-of-date.

3.1 Simulation Model

To evaluate the cost-performance trade-offs of precomputed routes, we have developed an event-driven simulator that models link-state routing at the connection level. A route is chosen for each incoming connection based on a throughput requirement (bandwidth b) and the available bandwidth in the network, based on the source's view of link-state information. Then, hop-by-hop signalling reserves the requested bandwidth at each link in the route. That is, if a link has a reserved bandwidth with utilization u, admitting the new connection increases the reservation to $u = u + b$. A signalling failure occurs if a link in the path cannot support the throughput requirement of the new connection (i.e., if $u + b > 1$). For the simulations in this section, we assume that a connection blocks after a signalling failure, though we briefly summarize other experiments that allow multiple signalling attempts. Blocking can also occur during path selection if the feasibility check suggests that none of the candidate routes can support the new connection; these routing failures impose less of a burden on the network than signalling failures, which consume resources at downstream nodes. Although our experiments do not explicitly model propagation and processing delays, we gain an estimate of average set-up latency through the connection blocking rate and frequency of route computation.

The source selects a minimum-hop route with the least cost[1]; previous studies show that algorithms

[1]A careful assignment of link weights w_i permits a single invocation of the Dijkstra algorithm to produce a minimum-cost, shortest path. In a network with diameter D, and link costs in the range $0 < c_i \leq 1$, the link weights $w_i = D + c_i$ ensure that

Parameter	MCI Internet	5-ary 3-cube
offered load	$\rho = 0.65$	$\rho = 0.85$
bandwidth	$b = (0, 4\%]$	$b = (0, 6\%]$
arrival rate	$\lambda = 1$	$\lambda = 1$
connection duration	$\ell = 46.8$	$\ell = 46.8$

Table 2: **Simulation invariants:** This table lists the parameters that remain fixed throughout the simulation experiments., In the MCI topology $N = 19$, $L = 64$, diameter $D = 4$, and average node degree is 3.4. The 5-ary 3-cube has $N = 125$, $L = 750$, $D = 6$, and degree 6.

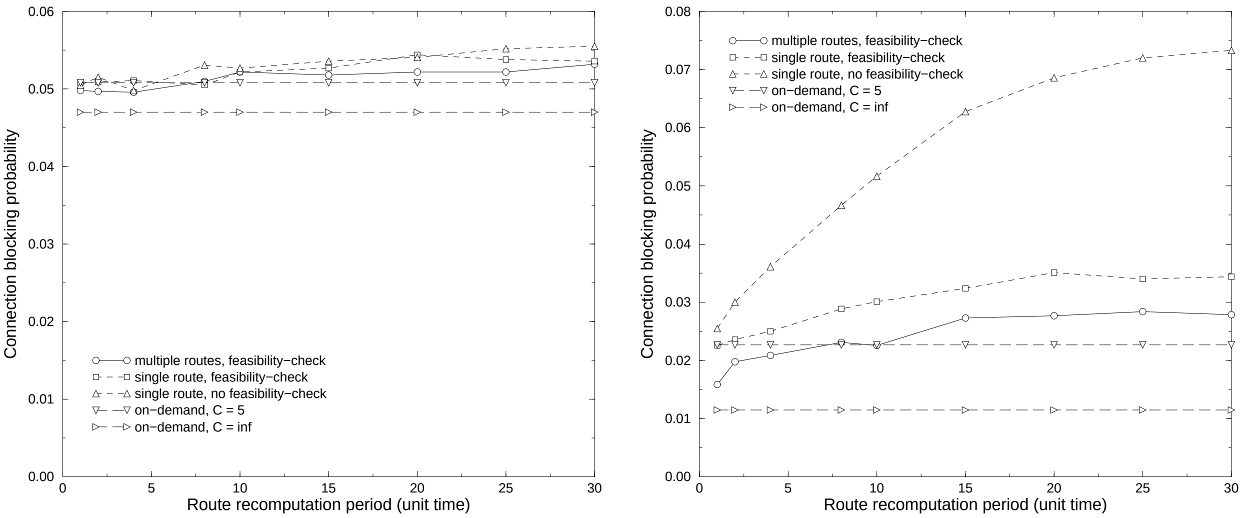
with a strong bias toward minimum-hop routes almost always outperform algorithms that do not consider the hop-count [17–20]. To distinguish among paths of the same length, each link has a cost in the set $\{1/C, 2/C, \ldots, C/C\}$. For the experiments in this paper, a link with reserved capacity u has cost $c = (\lceil u^2 \cdot (C-1) \rceil + 1)/C$; our experiments with link-cost functions show that an exponent of 2 biases away from routes with heavily-loaded links, without being too sensitive to small changes in link-state information. For simplicity we assume that links are bidirectional, with unit capacity in each direction. We evaluate the routing algorithms on a "well-known" core topology (an early version of the MCI Internet backbone [7, 20]) and a uniformly connected 125-node 5-ary 3-cube topology (with 5 nodes along each of 3 dimensions). The relationship between size and connectivity in the 5-ary 3-cube is similar to existing commercial networks, and allows us to study the potential benefits of having multiple minimum-hop routes between pairs of nodes.

Connection requests arrive at the same rate at all nodes and destinations are selected uniformly. Connection holding times have mean ℓ and follow a Pareto distribution with shape parameter 2.5 to capture the heavy-tailed nature of connection durations. Connection interarrival times are exponentially distributed with mean $1/\lambda$, and requested bandwidths are uniformly distributed with equal spread about a mean size $\bar{b}$. The experiments focus on mean bandwidth in the range of 2–3% of link capacity. While smaller bandwidths may be more realistic, the large requests provide some insight to the behavior of the high-bandwidth multimedia traffic (e.g., video) we are interested in. Also, smaller bandwidths result in very low blocking probabilities, making it very difficult to gain sufficient confidence on the simulation results in a reasonable time. With a connection arrival rate λ at each of N nodes, the offered network load is $\rho = \lambda N \ell \bar{b} \bar{h}/L$, where L is the number of links and $\bar{h}$ is the mean distance (in hops) between nodes, averaged across all source-destination pairs. Table 2 summarizes the simulation parameters for the two network topologies.

3.2 Accurate Link-State Information

The initial simulation experiments compare the performance and overhead of the routing algorithms under accurate link-state information, as shown in Figure 4 and 5. We vary the background period for path precomputation, and also allow a switch to recompute its shortest-path graph when the route extraction does not produce a candidate route (due to failed feasibility checks) and after a signalling failure. Since we allow only one signalling attempt for each connection, recomputing after a signalling failure only benefits

paths with fewer hops always appear cheaper. Such an assignment for w_i results in a relatively small number of possible path cost estimates, thus reducing the complexity of the computation.



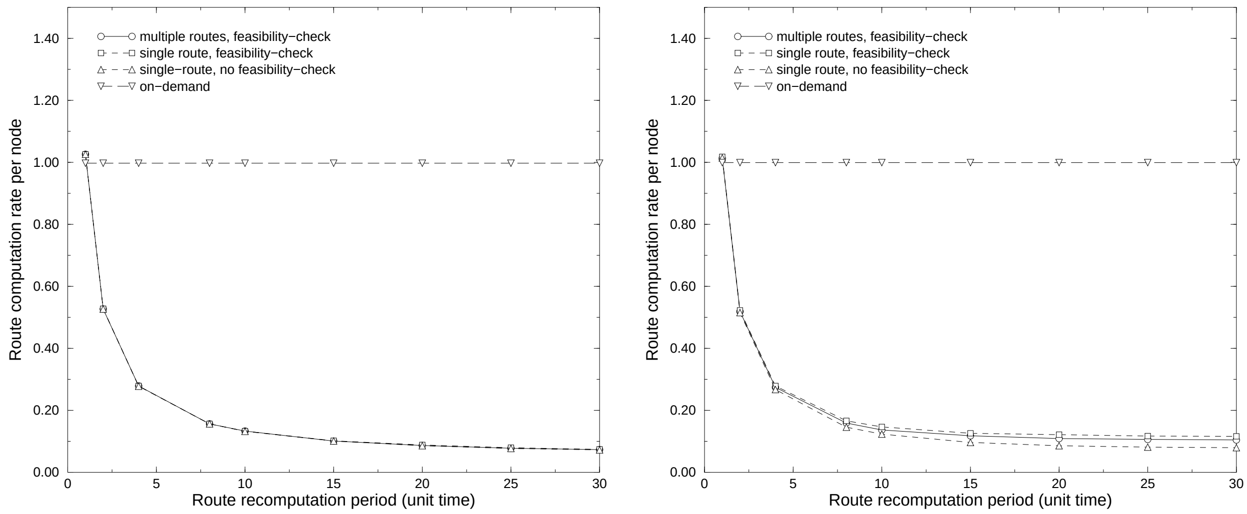
(a) MCI Internet topology (b) 5-ary 3-cube

Figure 4: **Performance with accurate link state:** These graphs plot the blocking probability as a function of the background path computation period under accurate link-state information, for both the MCI and 5-ary 3-cube topologies.

future arrivals. The single-route algorithms use a precomputed shortest-path graph with one route, as in Figure 1(a). The multiple-route approach offers greater flexibility by allowing the source to perform feasibility checks on several paths in the graph, as in Figure 1(b). We also consider two on-demand algorithms that compute routes on a per-request basis. One has a link-cost discretization of $C = 5$, identical to the precomputed routing algorithms, and the other has discretization limited only by machine precision (referred to as $C = \infty$).

Figure 4 plots the connection blocking probability as we increase the recomputation period to more than 30 times the connection interarrival time. The precomputation algorithms perform well, relative to the more expensive on-demand schemes. Feasibility checking substantially improves performance over traditional path-caching techniques, allowing the use of much larger computation periods. In addition, under accurate link-state information, feasibility checking completely avoids signalling failures by correctly "simulating" the effects of signalling during the route extraction process. This is particularly helpful in the 5-ary 3-cube network, since the richly-connected topology frequently has an alternate shortest-path route available when the first choice is infeasible. Other experiments illustrate that, without a feasibility check, precomputed routing can only achieve these performance gains by allowing multiple signalling attempts for a connection, at the expense of longer set-up delays and higher processing requirements.

The ability to precompute multiple candidate routes does not substantially reduce the blocking probability in Figure 4(a), since the sparsely-connected MCI topology typically does not have multiple shortest-path routes, let alone multiple routes of equal cost. In contrast, the 5-ary 3-cube experiment in Figure 4(b) shows a more substantial performance benefit. We expect a more significant gain under nonuniform traffic loads, since alternate routes would enable connections to circumvent regions of congestion. This is especially true during transient fluctuations in network load, caused by bursty connection arrivals or rerouting after a link failure. In these cases, the precomputation of multiple routes allows the source to survive longer intervals of time without computing a new shortest-path graph. The algorithm with multiple precomputed



(a) MCI Internet topology

(b) 5-ary 3-cube

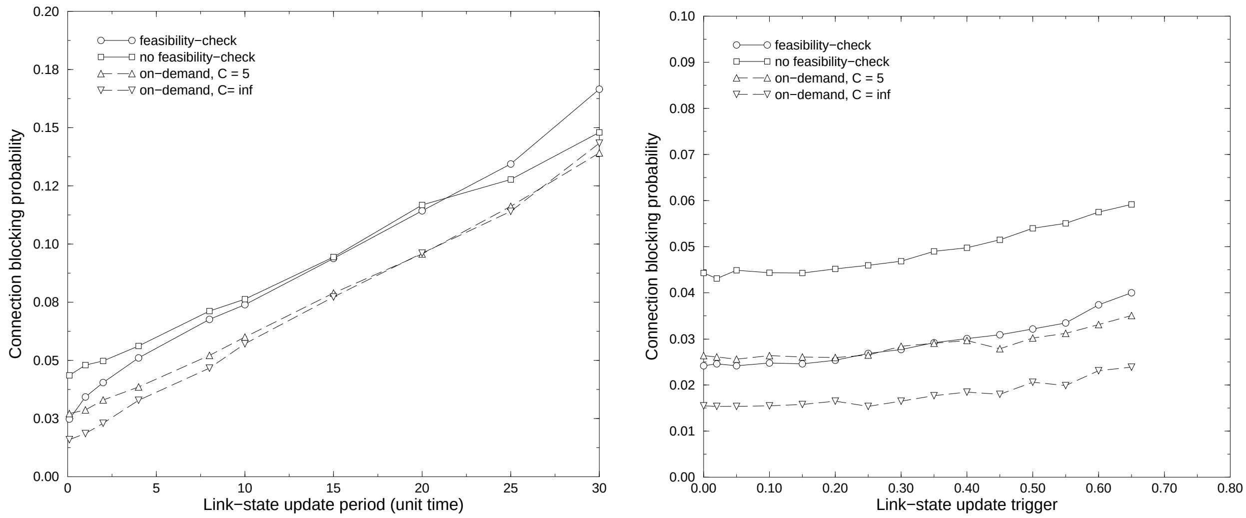
Figure 5: **Overhead with accurate link state:** These graphs plot the route computation frequency as a function of the background path computation period under accurate link-state information for both the MCI and 5-ary 3-cube topologies.

routes, coupled with the feasibility test, performs almost as well as on-demand routing with the same number of cost levels. However, using $C = \infty$ offers a noticeable performance advantage in Figure 4, since the accurate link-state information enables the fine-grain link-cost function to locate the "best" routes.

The lower blocking probabilities for on-demand routing come at the expense of a significant cost in processing load, as shown in Figure 5. In both the MCI and 5-ary 3-cube topologies, the route computation frequency is lowered by a factor of 10 below that of on-demand routing. In addition, the precomputation schemes have much lower complexity for each route computation, relative to the $C = \infty$ on-demand algorithm. Comparing the different precomputation algorithms, the processing load is dominated by the background recomputation of the shortest-path graph, though the feasibility test introduces slightly more triggered recomputations. As the period increases, the graphs flatten since triggered recomputations become increasingly common for all of the algorithms. Although disabling these triggered recomputations results in more predictable processing overheads, additional simulation experiments (not shown) indicate that this substantially increases the blocking probability.

3.3 Inaccurate Link-State Information

While the previous experiments assume that the source has accurate knowledge of network load, Figure 6 considers the effects of stale link-state information, for both periodic and triggered link-state updates, with a background recomputation period of 5 time units. As staleness increases, the $C = \infty$ and $C = 5$ curves gradually converge, since fine-grain link costs offer progressively less meaningful information about network load. However, large link-state update periods also degrade the effectiveness of the feasibility check in our precomputed routing scheme, as shown in Figure 6(a). Periodic updates can lead the source switch to mistakenly identify infeasible links as feasible, and feasible links as infeasible. When link-state information is extremely stale (e.g., an update period that is 20 times larger than the mean connection interarrival time), signalling blindly without a feasibility check offers better performance. In fact, under such large



(a) Periodic link-state updates

(b) Triggered updates

Figure 6: **Performance with stale link state:** These graphs plot the blocking probability for the 5-ary 3-cube topology for periodic and triggered link-state updates, with a background path computation period of 5 time units.

update periods, none of the QoS-routing algorithms perform well; under the same network and traffic configuration, even *static* shortest-path routing (not shown) can achieve a blocking probability of 16%. Experiments with the MCI topology configuration do not show as much benefit from feasibility checking, due to the lower number of shortest-path routes, though the feasibility test does reduce the likelihood of signalling failures.

Despite the effects of large update periods, we find that the feasibility check still offers significant advantages under more reasonable levels of link-state staleness. This is particularly true when link-state updates are triggered by a change in available link capacity, as shown in Figure 6(b). For example, a trigger of 0.2 spawns a link-state update message whenever available capacity has changed by 20% since the last update, due to the establishment and termination of connections. Triggered link-state updates, like triggered path recomputations, generate new routing information during critical fluctuations in network load. Under triggered link-state updates, the switches typically have accurate information about heavily-loaded links, even for large trigger values. Consequently, the connection blocking probability is fairly insensitive to the trigger, across all of the routing algorithms. In addition, feasibility checking remains very effective across the range of update triggers and competitive with on-demand routing (with $C = 5$), in contrast to the results for update periods in Figure 6(a). We find also that using triggered updates allows feasibility checking to reduce blocking even in the MCI topology, though the difference between feasibility and no feasibility checking is less significant than in the 5-ary 3-cube.

Although the blocking probability remains nearly constant as a function of the link-state trigger, small triggers result in fewer signalling failures, as long as the source performs a feasibility test. In contrast, signalling failures account for *all* connection blocking when routing does not involve feasibility checking. As the trigger grows, the feasibility test sometimes mistakenly concludes that a heavily-loaded link is feasible, and the connection blocks in signalling instead of experiencing a routing failure. Still, even for a 50% trigger, signalling failures only contribute 30% of the connection blocking under feasibility checking. Also, despite the fact that periodic updates cause significant staleness and worsen overall connection blocking, feasibility checking still manages to avoid signalling for connections that ultimately block about $30-40\%$ of the time. The ability of feasibility checking to reduce blocking inside the network is an important benefit since signalling failures consume processing resources and delay the establishment of other connections. Routing failures, on the other hand, are purely local and do not consume additional resources beyond the processing capacity at the source switch.

Examining the blocking relative to hop-count shows that feasibility checking at the source also helps in routing connections between distant source-destination pairs. Since bandwidth must be reserved on more links, signalling blindly without checking recent link state has a lower probability of finding a successful route. Other experiments show that when a hold-down timer is used to impose a minimum time between consecutive update messages, the blocking probability rises slightly for all algorithms, though the benefit of feasibility checking remains. The hold-down timer is useful, however, in reducing link-state update overhead, especially when using small trigger levels. For example, with a hold-down timer equal to the mean connection interarrival time, the update generation rate can be reduced by over 35% for triggers in the range $0-0.1$. Moreover, even with a hold-down timer, coarser triggers do not degrade performance. The combination of triggered updates and a small hold-down timer, coupled with feasibility checks and multiple precomputed routes, offer an efficient and effective approach to quality-of-service routing in large networks.

4 Related Work in QoS Route Precomputation

Previous work on route precomputation has focused on path caching policies, performance evaluation, and algorithmic issues. Our work complements these studies by emphasizing lower-level mechanisms and introducing an efficient framework for precomputing and storing QoS routes, which applies to a variety of routing and signalling policies.

Research on path caching has focused on storing routes in a separate data structure and considering different policies for updating and replacing precomputed routes. The work in [8] introduces a policy that invalidates cache entries based on the number of link-state updates that have arrived for links in the precomputed paths. The proposed algorithms also check the current link-state when selecting a path from the cache and allow recomputation when the cached paths are not suitable, similar to the feasibility check in this paper. This earlier work does not, however, address route computation or path extraction mechanisms. Another study proposes a set of route precomputation policies that optimize various criteria, such as connection blocking and set-up latency [10]. The algorithms try to locate routes that satisfy several QoS requirements through an iterative search of precomputed paths (optimized for hop-count) followed, if necessary, by several on-demand calculations that optimize different additive QoS parameters, one at a time.

Other research has focused on detailed performance evaluation to compare precomputed and on-demand routing under different network, traffic, and staleness configurations. The work in [21] evaluates the performance and processing overhead of a specific path precomputation algorithm. The study adopts a Bellman-Ford-based algorithm from [6] and evaluates a purely periodic precomputation scheme under a variety of traffic and network configurations. The study presents a detailed cost model of route computation to compare the overhead of on-demand and precomputed strategies. As part of a broader study of QoS routing, the work in [7] evaluates a class-based scheme that precomputes a set of routes for different bandwidth classes. The evaluation compares the performance of several algorithms for class-based path computation to on-demand computation. These two studies do not propose any particular strategy for path storage or extraction but instead focus on performance trends.

The remaining studies consider different ways to precompute paths for multiple destination nodes and connection QoS requirements. The work in [6] proposes a Dijkstra-based algorithm that computes minimum-hop paths for different bandwidth classes. Another algorithm, introduced in [9], precomputes a set of extremal routes to all destinations such that no other route has both higher bottleneck bandwidth *and* smaller hop-count. The Bellman-Ford-based algorithm in [6] uses a similar optimization criterion to construct a next-hop routing table with multiple routing entries for each destination. The emphasis of these last three proposals is on algorithmic issues, such as reducing complexity. In addition to a focus on computational overheads, we also consider efficient ways to store precomputed paths and apply the most recent link-state information.

5 Conclusions and Ongoing Work

In this paper, we have proposed efficient mechanisms for precomputing quality-of-service routes, while still applying the most recent link-state information at connection arrival. Route computation employs a small extension to Dijkstra's algorithm, coupled with discretized link costs, to generate a shortest-path graph with one or more routes to each destination. On connection arrival, route extraction involves a depth-first search with a feasibility test, which returns the first route in the common case. Simulation experiments show that the algorithm offers substantial reductions in computational load with only a small degradation in performance, compared to more expensive on-demand algorithms. Our precomputation scheme continues to perform well under stale link-state information, particularly under triggered link-state updates. In addition to having lower blocking probabilities than traditional path-caching schemes, the feasibility test reduces network overhead by decreasing the frequency of signalling failures.

As part of future work, we are investigating enhancements to our path precomputation and extraction algorithms. To increase the likelihood of having multiple candidate routes to each destination, we are considering heuristics for generating *near*-minimum-cost alternate routes, while still storing the precomputed routes in a compact graph representation. In addition, we are pursuing other ways to test the suitability of routes during the extraction process. For example, instead of checking *link* feasibility, the source could use the most recent link-state information to compute new *path* costs as part of the depth-first search for a route; whenever the accumulated path cost exceeds a certain threshold, the algorithm can backtrack to consider other precomputed routes. This approach is well-suited to precomputing paths that balance network load without requiring information about the traffic or QoS parameters of individual connections. Finally, we are performing more extensive simulation experiments to evaluate our path-selection schemes under a wider range of network topologies, communication workloads, and routing/signalling policies.

References

[1] W. C. Lee, M. G. Hluchyj, and P. A. Humblet, "Routing subject to quality of service constraints in integrated

communication networks," *IEEE Network Magazine*, pp. 46–55, July/August 1995.

[2] Z. Whang and J. Crowcroft, "Quality-of-service routing for supporting multimedia applications," *IEEE Journal on Selected Areas in Communications*, vol. 14, pp. 1228–1234, September 1996.

[3] E. Crawley, R. Nair, B. Rajagopalan, and H. Sandick, "A framework for QoS-based routing in the Internet." Internet Draft (draft-ietf-qosr-framework-03.txt), March 1998.

[4] PNNI Working Group, *ATM Forum 94-0471R13 PNNI Draft Specification.* Document available at ftp://ftp.atmforum.com/pub/contributions.

[5] Z. Zhang, C. Sanchez, B. Salkewicz, and E. S. Crawley, "Quality of service extensions to OSPF or quality of service path first routing (QOSPF)." Internet Draft (draft-zhang-qos-ospf-01.txt), September 1997.

[6] R. Guerin, A. Orda, and D. Williams, "QoS routing mechanisms and OSPF extensions," in *Proceedings of IEEE GLOBECOM*, (Phoenix, AZ), November 1997. Extended version appears as Internet Draft (draft-guerin-qos-routing-ospf-03.txt), March, 1998.

[7] Q. Ma and P. Steenkiste, "On path selection for traffic with bandwidth guarantees," in *Proceedings of IEEE International Conference on Network Protocols*, (Atlanta, GA), October 1997.

[8] M. Peyravian and A. D. Kshemkalyani, "Network path caching: Issues, algorithms and a simulation study," *Computer Communications*, vol. 20, pp. 605–614, 1997.

[9] J.-Y. Le Boudec and T. Przygienda, "A route pre-computation algorithm for integrated services networks," *Journal of Network and Systems Management*, vol. 3, no. 4, pp. 427–449, 1995.

[10] A. Iwata, R. Izmailov, H. Suzuki, and B. Sengupta, "PNNI routing algorithms for multimedia ATM internet," *NEC Reserach & Development*, vol. 38, January 1997.

[11] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee, "How to model an internetwork," in *Proceedings of IEEE INFOCOM*, pp. 594–602, March 1996.

[12] A. G. Greenberg and R. Srikant, "Computational tecniques for accurate performance evaluation of multirate, multihop communication networks," *IEEE/ACM Transactions on Networking*, vol. 5, pp. 266–277, April 1997.

[13] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* Cambridge, MA (New York): MIT Press (McGraw-Hill), 1990.

[14] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, "Shortest-path algorithms: Theory and experimental evaluation," *Mathematical Programming*, vol. 73, pp. 129–174, May 1996.

[15] J. Y. Yen, "Finding the K shortest loopless paths in a network," *Management Science*, vol. 17, pp. 712–716, July 1971.

[16] E. L. Lawler, "A procedure for computing the K best solutions to discrete optimization problems and its application to the shortest path problem," *Management Science*, vol. 18, pp. 401–405, March 1972.

[17] H. Ahmadi, J. S. Chen, and R. Guerin, "Dynamic routing and call control in high-speed integrated networks," in *Teletraffic and Datatraffic in a Period of Change: Proceedings of the International Teletraffic Congress* (A. Jensen and V. B. Iversen, eds.), vol. 14 of *Studies in Telecommunication*, pp. 397–403, Copenhagen, Denmark: North-Holland, June 1991.

[18] R. Gawlick, C. Kalmanek, and K. Ramakrishnan, "Online routing for virtual private networks," *Computer Communications*, vol. 19, pp. 235–244, March 1996.

[19] C. Pornavalai, G. Chakraborty, and N. Shiratori, "QoS based routing in integrated services packet networks," in *Proceedings of IEEE International Conference on Network Protocols*, (Atlanta, GA), October 1997.

[20] Q. Ma and P. Steenkiste, "Quality-of-service routing for traffic with performance guarantees," in *Proc. IFIP International Workshop on Quality of Service*, (Columbia University, New York), pp. 115–126, May 1997.

[21] G. Apostolopoulos and S. K. Tripathi, "On the effectiveness of path pre-computation in reducing the processing cost of on-demand qos path computation," in *to appear in Proceedings of IEEE Symposium on Computers and Communication*, (Athens, Greece), June 1998.