Path Caching in Connection-Oriented Networks

Mohammad Peyravian and Ajay D. Kshemkalyani
IBM Corporation, P. O. Box 12195, Research Triangle Park, NC 27709, U.S.A.

Abstract

In a connection-oriented communication network, a computed network path can be stored in the local cache of the source node for later reuse. We propose that network path caching can provide an efficient way to eliminate, whenever possible, the expensive path computation algorithm that has to be performed in setting up a network connection. This paper is the first known work on network path caching in decentralized connection-oriented networks. We first identify and analyze the issues that arise in caching network paths. Based on our extensive study of network path caching schemes, we then propose two path caching algorithms to reduce the number of path computations in the network when a new connection is to be established. A simulation study of the two algorithms is then presented. We conclude that both algorithms perform very well and significantly reduce the number of path computations in setting up connections.

1 Introduction

There is a critical need to be able to setup new network connections efficiently and rapidly because of the growth of high-speed wide-area connection-oriented communication networks such as the Asynchronous Transfer Mode (ATM) [13]. A new connection request arrives with its quality of service (QoS) requirements and an appropriate network path that meets the QoS requirements must be determined before the network connection can be established. Paths in decentralized connection-oriented networks are computed at the source by running a path selection algorithm on a topology database that represents a recent topology snapshot of the network. Numerous path selection algorithms have been proposed in the literature to optimize various parameters such as the number of hops or a specific cost function [4, 7, 12, 16]. Running an optimal path selection algorithm is computationally intractable [8]. Although some of the various algorithms in literature use heuristics, they are also very expensive and may introduce large delays before the connection can be established. Furthermore, other network resources are not optimized.

The following trends in networking complicate the problem of high overhead for path selection. Firstly, the expansion of wide-area networks introduces more nodes and links in the network topology, which adds complexity to the path selection. Secondly, private networks are being increasingly interconnected using protocols such as ATM Private Network Node Interface (PNNI) [15], which further increases the size of the network topology graph on which a path selection algorithm has to be run. Thirdly, as more high-speed links are introduced in the system and message transmission times decrease, the bottleneck in establishing a connection becomes the path selection algorithm. Fourthly, as real-time constraints on the establishment of a connection become more common due to the proliferation of real-time applications, the path selection algorithm becomes a bottleneck.

We propose that caching of network paths that have been precomputed can reduce the connection establishment time for subsequent connection requests by an efficient reuse of stored paths without having to rerun a path computation algorithm. Caching schemes have been extensively studied in memory systems [1], multiprocessor memory systems [1, 5], error recovery in multiprocessors [11], distributed systems [6], and network file systems [14]. To the best of our knowledge, there has been no prior published work on caching of network paths in decentralized connection-oriented networks. The closest known work is [2] which restricts its traffic QoS to very specific values required by interactive video. Their scheme builds and maintains a delay table on each node so that path selection can be solved by a simple table lookup; hence it uses a very different concept and assumptions than those we use.

Manipulating a local cache of network paths computed in the past differs from manipulating a cache for a memory system, multiprocessor system, network file system or a distributed system in several aspects. We identify and discuss such aspects. We have examined various schemes for adding cache entries and deleting cache entries to discard obsolete path information. Based on our analysis of various network path caching schemes, we present two algorithms for network path caching. Algorithm 1 discard cached path entries based on the number of topology updates to any one link in the path. We found this to be the most effective and simple criterion from among the criteria we considered. Algorithm 1 discards a cached path if any one of the links on the path has received N topology updates, where N is a tunable parameter, since insertion of the path in the cache. If a cached path entry results in a failed connection setup, then the connection request is rejected. Algorithm 2 is a special case of Algorithm 1 where N is ∞ and differs in that if a cached path results in a failed connection setup, a path computation attempts to setup the connection before the connection request is rejected. We present an extensive simulation study of these two algorithms and compare them with the case when no path caching is done. We conclude that both these algorithms perform very well and substantially reduce the number of path computations. Moreover, the algorithms are simple to implement.

This paper is organized as follows: Section 2 presents a simple network model for decentralized/distributed connection-oriented networks. Section 3 discusses the issues involved in caching network paths, and how path caching differs from traditional forms of caching such as memory caches, multiprocessor caches and caches for network file systems. Section 4 presents two algorithms for network path caching. Section 5 presents a simulation study of network path caching and compares the performance of the two proposed path caching algorithms. Section 6 gives the conclusions.

0-7803-3336-5/96 $5.00 © 1996 IEEE

2 System Model

The network model for decentralized connection-oriented networks we use for our study of the path caching problem is described in this section. The network model borrows some concepts from connection setup of [3], and ATM PNNI [15], a decentralized type of network which provides connection-oriented services using the concept of source routing and link state. PNNI provides connections between different networks which are organized in an hierarchical manner. A node is provided complete knowledge of the topology of its own network and an aggregated view of the topology of other networks. Each node maintains a topology database that contains the latest information about the topology of all the networks[1], including the link states. In source routing using link states, the source computes a complete route from the source to the destination based on its knowledge about the current states and utilizations of the links in the global network. Each link is owned by a link manager (LM), and when a significant link state change occurs, the link manager broadcasts the information to all the nodes in the network in PTSEs (PNNI Topology State Elements) using a hierarchical flooding mechanism. Link and node failures are also advertised by broadcasting PTSEs. Each link manager, independent of other link managers, decides whether it can accept a new connection when it receives a connection setup request. Thus, there is no concept of centralized control.

The topology database is never completely synchronized with the real topology for the following reasons: (i) there is a delay in the propagation of the PTSE broadcasts which themselves occur only when a significant change in the link state or node state occurs, and (ii) the aggregated information broadcast from other private networks is inherently inaccurate due to approximation in the aggregation process.

The setup of a connection proceeds as follows: The origin computes a complete route from the origin to the destination based on its knowledge about the current states and utilizations of the links. Then, the origin constructs a connection setup request for the connection and sends it to all the link managers along the computed route. A link manager along the route accepts the connection and returns a positive reply only if it can provide the resources to accommodate the connection. Otherwise, it rejects the connection and returns a negative reply to the origin. If a link manager accepts a connection, it allocates the requested resources for the connection. When the origin receives the replies it determines whether a connection setup is successful. The connection setup is successful only if all the replies are positive. If the connection setup is unsuccessful, the origin computes a new route (which excludes the links that replied unfavorably) and repeats the setup process. When the connection setup is unsuccessful, the origin also sends a path takedown request to the link managers along the path of the connection that replied favorably. When a link manager receives a path takedown request for a connection, it releases the network resources associated with that connection.

A link manager that processes a setup request when not enough resources are available selects connections to be preempted from the set of all connections currently using the link with priorities lower than the priority of the requesting connection [9]. Preemption is triggered at a link by the link manager only if enough resources can be released by preemption to accommodate the requesting connection at that link. For each connection to be preempted, the link manager sends a preemption notification message to the origin of the connection. At the receipt of the preemption notification message, the origin takes some actions to reroute the connection. First, it takes down the connection by sending a path takedown request to all the link managers along the path of the connection. Then, it computes a new route for the preempted connection and starts the setup process.

When a link or an intermediate node along the path of an ongoing connection fails, our protocol switches the connection to an alternate path. Link and node failures are detected by both origin and destination nodes via topology database update broadcasts. When a link or an intermediate node along the path of an ongoing connection fails, both the origin and destination send path takedown requests along the path of the connection. Then, the origin computes a new route which excludes the failed links or nodes and uses the procedure described above to perform the connection setup.

When the origin or destination wants to terminate a connection, it constructs and sends a path takedown request to all the link managers along the path of the connection.

2.1 Problem Statement

A connection request arrives at a node with its QoS requirements, a predefined priority, and a predefined bandwidth. No knowledge of the holding time or the future arrivals of connection requests is available. The path caching problem is to determine how to use a cache of network paths computed in the past when a future connection setup request arrives, so as to minimize the overhead of a new path computation for each arriving connection request.

3 Path Caching

When a path caching scheme in conjunction with path computation is used at a node to find routes for connections, a "cache table" has to be maintained at that node. Each entry in the cache table gives a path to a destination node. Associated with each path entry are attributes such as destination node, number of hops, maximum delay, maximum packet size, packet loss probability, and security level.

To provide a path for a connection request, the cache table is searched first to check whether there is a path entry in the cache table that meets the connection QoS requirements such as maximum delay, maximum number of hops, and packet loss probability. If such a path is found, then the node's local topology database is used to examine whether the links along the path are up and have enough free bandwidth to support the connection. If the local topology database confirms that the links along the path meet the connection's QoS and bandwidth requirements, then the path computation part of the connection establishment process is skipped and the path obtained from the cache table is used instead. Otherwise, the node attempts to compute a route for the connection.

Caching schemes used in each of the different contexts such as memory systems, network file systems, and multiprocessors have the following components:

- **Cache update policy**: This policy determines whether an entry for a specific path should be inserted into the cache.
- **Cache invalidation policy**: This policy determines how an entry in the cache is determined to have become invalid.
- **Cache replacement policy**: This policy determines the choice of an entry in the cache that needs to be replaced in order to

[1] Henceforth, unless otherwise specified, the term "network" will refer to the global network formed by the conglomeration of networks that are connected using PNNI.

make space for another entry that is to be inserted in the cache. This policy arises because the cache is typically of finite size.

The various schemes differ in the implementation of these policies. We now consider how caching of network paths differs from caching used in other contexts. We then consider the various schemes for network path caching that we considered before choosing the schemes used in our algorithms in Section 4.

Network path caching varies from the well-studied and implemented forms of caching such as in memory systems [1], multiprocessor memory systems [1, 5], error recovery in multiprocessors [11], distributed systems [6], and network file systems [14], in the following respects:

1. The penalty for the use of an outdated network path cache entry is higher due to the time and message overhead for a failed connection setup. Hence, it is important to determine a good path caching scheme.
2. Cache size which is usually a constraint in traditional memory system caches is not a constraint in caching network paths because of two reasons. First, the number of network paths originating from a node is relatively small, as compared to the number of variables or pages that a multiprocessor or a distributed memory system may want to cache. Second, the cache memory used for network caches can be primary (main) memory – the savings in using cached paths is the use of precomputed paths which avoids the expensive path selection algorithms, as opposed to the savings offered by traditional memory caches by avoiding access to main memory and instead accessing fast volatile memory. In this sense, the usage of the term "cache" to refer to storing of precomputed network paths in primary memory is a misnomer.
3. For memory caches, multiprocessor caches, and caches used in operating systems, the currency of a cache entry is boolean: valid or invalid. Therefore, an invalid cache entry is easily detected by the receipt of a single invalidation message that is broadcast. There is no easy way to determine when a particular cache entry for a precomputed network path is invalid because the validity of a cached network path is "analog". When a PTSE about a change in the link capacity of a certain link is received in a network, the cached path entry for the path that contains the link is not quite invalidated; only some of its parameters may change based on the information in the received PTSE. The path itself may be good for future connection requests, with the difference that its characteristics are slightly different from those advertised in the cache entry for the path. Therefore, an invalid cached path entry is extremely difficult to detect even using the information in the arriving PTSEs.

The following assumptions about network path caching schemes are made in our study:

1. When a PTSE about a link or a node is received, the algorithm will not scan and update the cached path entries for paths that pass through the link or the node using the changed capacity/metrics in the PTSE because that is computationally expensive.
2. In the simulation study of path caching we do not put a hard bound on the cache size. This is mainly because the number of paths with a fixed source node is relatively small compared to the cache sizes used for memory caches and caches in multiprocessor systems. In addition, as explained before, main memory which is virtually unbounded can be used for storing precomputed paths; the fast but expensive volatile cache memory that is required by traditional forms of caching is not required. Hence, we do not specifically formulate a cache replacement policy for simulation purposes. However, for implementation in real systems, in case cache shortage occurs, the path with the longest lifetime since its last successful use should be deleted.

We have examined various schemes for cache invalidation to discard obsolete path information. For example, (i) deleting entries for those paths that have low bandwidth because they are more likely to be preempted, or (ii) deleting entries for those paths that have high hop count because these paths are more likely to cease existence due to involvement of multiple links, or (iii) monitoring the change in bandwidth of the links involved in each path based on information in the topology database and deleting entries for those paths for which the change in bandwidth for any link crosses a tunable threshold. The scheme (iii) is not favored because it is expensive due to the expensive updating of bandwidths for cached paths on a continuous basis. More complicated schemes such as deleting path entries based on their lifetime since insertion in the cache were also considered but were not pursued because they require a lot of information that adds overhead to the cache maintenance. We found that from among the criteria we considered, the most effective and simple criterion was to discard cached path entries based on the number of topology updates to any one link in the path.

4 Path Caching Algorithms

We present two algorithms for network path caching based on our analysis of various network path caching schemes. Algorithm 1 discards cached path entries based on the number of topology updates to any one link in the path. We found this to be the most effective and simple criterion from among the criteria we considered. Algorithm 1 discards a cached path if any one of the links on the path has received N updates, where N is a tunable parameter, since insertion of the path in the cache. If a cached path entry results in a failed connection setup, then the connection request is rejected. Algorithm 2 is a special case of Algorithm 1 where N is ∞; it differs in that if a cached path results in a failed connection setup, a path computation attempts to setup the connection before a connection request is rejected.

Variables/Constants used by algorithms:

- N: constant integer (used by Algorithm 1 only);
- P_i: $\langle \langle e_{i_1}, update_{i_1} \rangle, \langle e_{i_2}, update_{i_2} \rangle, \ldots\ldots \langle e_{i_l}, update_{i_l} \rangle \rangle$;

where P_i is a path of length l in cache, e_{i_j} is a link along the path P_i, and $update_{i_j}$ stores the number of PTSEs received for link e_{i_j}. Algorithm 2 does not use the variable $update$. (The path attributes such as number of hops, maximum delay, maximum packet size and packet loss probability are not shown here.)

4.1 Algorithm 1

Receive a connection request with destination and QoS requirements

1. **if** the cache contains a path to the same destination that satisfies the QoS of the request and the topology database confirms this path **then**
   attempt to setup a path to destination.
   if connection setup succeeds **then**
   accept the request
   else
   reject the connection request.
   else
   invoke path computation;
   attempt to setup a path to destination.
   if connection setup succeeds **then**
   (Cache update policy:) store the connection's path along with its attributes in the cache and for each link e_{i_j} in the path, set $update_{i_j}$ to 0.
   else
   reject the connection request
2. **Cache invalidation policy:** Before storing a new path in the cache, remove all the exiting entries that have the same destination. An implementation may choose to keep more than one path for the same destination in the cache; this can be done by storing paths based on their QoS parameters. This has the danger of letting the cache grow big.

Receive a PTSE for link e_{i_j} **(Cache invalidation policy):**

for each path P_i in the cache **do**
if e_{i_j} belongs to the path **then**
$P_i.update_{i_j} := P_i.update_{i_j} + 1$
if $P_i.update_{i_j} > N$ **then**
delete path P_i from cache.

There are two components of the cache invalidation policy by which a path is removed from the cache. First, when a new path to the same destination with the same QoS is stored, the existing cache entry for the path is deleted. Second, a parameter (N) which depends on the number of PTSEs that a node receives from a LM associated with a link, is used to delete cache entries. After a node receives $N+1$ PTSEs from a link, it removes all the paths that use that link from its cache table. In a sense, a node ignores N PTSEs received from a link and acts when it receives the $(N+1)$th PTSE. N is a tunable parameter and its value can be anywhere from 0 to ∞. When $N = 0$, no PTSE is ignored, that is every time a PTSE is received from a link all the paths that use that link are removed from the cache table. When $N = \infty$, no path is removed from the cache table based on receiving a PTSE from a link. In a sense, entries are not removed from the cache table for $(100.N)/(N+1)$ percent of the PTSEs received from a link. The underlying reason for this scheme is to keep the paths in the cache table current.

4.2 Algorithm 2

Algorithm 2 is the same as Algorithm 1 except as follows. Algorithm 2 gives a connection one more chance to be setup if the previous setup failed and the path was obtained from the cache table; the second setup attempt finds a path (if one is available) using the path computation method. In addition, Algorithm 2 does not remove any paths from the cache table when a PTSE (from a link) is received. This is the same as setting N to ∞ in Algorithm 1. Algorithm 2 is now presented.

Receive a connection request with destination and QoS requirements

1. **if** the cache contains a path to the same destination that satisfies the QoS of the request and the topology database confirms this path **then**
   attempt to setup a path to destination.
   if connection setup succeeds **then**
   accept the request
   else
   invoke path computation;
   attempt to setup a path to destination.
   if connection setup succeeds **then**
   (Cache update policy:) store the connection's path along with its attributes in the cache.
   else
   reject the connection request
   else
   invoke path computation;
   attempt to setup a path to destination.
   if connection setup succeeds **then**
   (Cache update policy:) store the connection's path along with its attributes in the cache
   else
   reject the connection request.
2. **Cache invalidation policy:** Before storing a new path in the cache, remove all the exiting entries that have the same destination. An implementation may choose to keep more than one path for the same destination in the cache; this can be done by storing paths based on their QoS parameters. This has the danger of letting the cache grow big.

5 Simulation

5.1 Model

A connection-level simulation was used to study path caching and compare the two proposed algorithms in a dynamic network environment where connections come and go. The simulation model has most mechanisms of typical connection-oriented networks. Its main components are a path selection algorithm which selects a minimum-hop path between an origin-destination pair, a connection setup and takedown protocol, and a topology information distribution protocol. In addition to the above components, the model also has a connection preemption protocol and a path-switch mechanism which reroutes connections preempted due to link/node failure or preemption. The simulation program is written in C and SIMSCRIPT and has about 4000 lines of code and consists of a number of processes which execute several dynamic objects and routines. A process is created at a simulated time and it performs a sequence of events separated by lapses of time. The process concept is used to represent connections, connection generation, and messages, while routines are used to represent static objects such as route computation.

The input to the simulation program includes a network configuration — the nodes, the transmission links with their propagation delays and capacities, — source/destination distribution, connections' characteristics, link failure events, and other controlling parameters such as simulation time, simulation seeds, and maximum connection hops. As will be described later, the program collects and reports a number of statistics.

The program simulates the lives of connections from the time they are created until they terminate. Connection interarrival times are exponentially distributed. Upon arrival of a connection to the network,

its source and destination nodes, priority, bandwidth, holding time, and delay are chosen probabilistically. Once the connection's parameters are selected, the source process examines the local cache to determine if a path satisfying the connection's parameters is already precomputed and stored in the cache. If the local cache does not contain such a path, a path selection algorithm is run and a path in the network is determined. This algorithm attempts to find a path that has a minimum number of hops while satisfying the connection's quality of service parameters. If there are several eligible paths with the same number of hops then one of them is chosen based on lowest "weight" of the path. This weight is the sum of weights of the individual links. This path selection algorithm and the notion of link weights are described in detail in [10].

Once a path has been selected, the connection control protocol described in Section 2 attempts to establish the connection. Basically, when a connection request arrives, a connection is established if the network has the bandwidth to support the connection. Once established, the connection begins its "talk" phase. However, if there is not enough bandwidth to establish the connection, then if there are sufficient low-priority connections that can be preempted to free enough bandwidth for this connection, then those low-priority connections will be preempted and the connection request gets satisfied. When the connection request cannot be accommodated, it is rejected. When a connection is preempted, it is treated like a new connection. When a connection successfully completes its talk phase, it gets taken down. So, note that a successfully completed connection may have been rerouted one or more times due to preemption, link failure, or node failure.

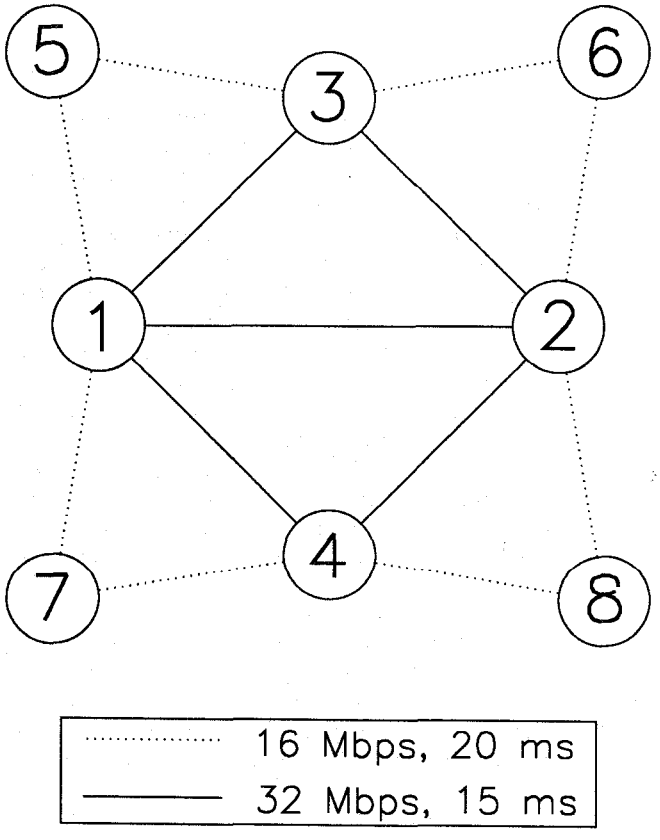

Figure 1. Network topology and structure.

Nodes	Source probability	Destination probability (given that source ≠ destination)
1,2	0.47	0.47
3,4,5,6,7,8	0.01	0.01

Table 1. Load Distribution A (asymmetric load: for expts. 1 and 2).

Nodes	Source probability	Destination probability (given that source ≠ destination)
1,2	0.2	0.2
3,4,5,6,7,8	0.1	0.1

Table 2. Load Distribution B (symmetric load: for Expt. 3).

When a connection is accepted on a link or removed from a link, a bandwidth reservation table for that link is updated. When a significant change in the link bandwidth reservation occurs, a PTSE is broadcast to every node in the network. This is done only if the change in the reservation level for the link is significant, i.e., if it exceeds some threshold value defined for that link. A PTSE is also broadcast when a link fails or comes up. So, a connection setup request may not be successful for two reasons: the topology database at the originating node may not be "current" and/or multiple sources may compete for a limited available bandwidth by sending connection setup requests to a particular link almost simultaneously.

5.2 Experiments

We have conducted wide-range simulation with various network conditions such as network topology, number of priority levels, link bandwidth, and traffic pattern to study path caching and the behavior of the two proposed algorithms.

Network Structure: The simulation experiments used the following network model which is an abstraction of a real network. The network, shown in Figure 1, is two-tiered and consists of 8 nodes and 26 unidirectional links. The inner links are 32 Mbps links with a propagation delay of 15ms, and the outer links are 16 Mbps links with a propagation delay of 20ms. In the experiments, two types of network load distributions were used by adjusting the selection of the origin and destination pairs for the connections. The two network load distributions, denoted Distribution A and Distribution B, were obtained by varying the probabilities of origin and destination pairs of connections, as shown in Tables 1 and 2, respectively. In Distribution A, the origin and destination pairs for the connections were selected such that the

Connect-ion priority	Bandwidth range (in Kbps)	Number of BW types in BW range (uniform distribution)	Mean holding time (in secs) (exponential distribution)	Delay (in ms) (uniform distribution)
1	800	1	100	10-60

Table 3. Traffic Characteristic A (for Expt. 1).

Run	Path Caching Algo.	Connection inter-arrival period for entire network (sec)	N for Algo. 1 (% of PTSEs ignored)	Average link reservation level	Cache hit Probability (if path caching is used)	Connection success probability	Number of path computations
1H	no caching	0.2	N/A	0.819	N/A	0.626	50014
2H	Algo. 1	0.2	0 (0%)	0.690	0.521	0.542	9258
3H	Algo. 1	0.2	1 (50%)	0.620	0.471	0.494	5225
4H	Algo. 1	0.2	3 (75%)	0.575	0.436	0.452	3431
5H	Algo. 1	0.2	9 (90%)	0.494	0.364	0.374	1713
6H	Algo. 2	0.2	N/A	0.843	0.814	0.624	32560
1M	no caching	0.3	N/A	0.778	N/A	0.871	33383
2M	Algo. 1	0.3	0 (0%)	0.570	0.718	0.733	3290
3M	Algo. 1	0.3	1 (50%)	0.540	0.690	0.704	2472
4M	Algo. 1	0.3	3 (75%)	0.500	0.634	0.646	1854
5M	Algo. 1	0.3	9 (90%)	0.456	0.591	0.598	1196
6M	Algo. 2	0.3	N/A	0.815	0.891	0.874	12816
1L	no caching	0.4	N/A	0.530	N/A	0.987	24922
2L	Algo. 1	0.4	0 (0%)	0.398	0.833	0.836	1730
3L	Algo. 1	0.4	1 (50%)	0.404	0.844	0.847	1535
4L	Algo. 1	0.4	3 (75%)	0.412	0.840	0.842	1354
5L	Algo. 1	0.4	9 (90%)	0.402	0.808	0.809	1032
6L	Algo. 2	0.4	N/A	0.578	0.968	0.988	2114

Table 5. Simulation results of Experiment 1.

Connect- - ion priority	Bandwidth range (in Kbps)	Number of BW types in BW range (uniform distribution)	Mean holding time (in secs) (exponential distribution)	Delay (in ms) (uniform distribution)
1	320-3200	10	100	10-60
2	320-3200	10	100	10-60
3	320-3200	10	100	10-60
4	320-3200	10	100	10-60

Table 4. Traffic Characteristic B (for Expt. 2 & 3).

load in the network is asymmetric, with nodes 1 and 2 experiencing very high load. In Distribution B, the selection of the source and destination pairs for the connections was such that the network load is uniformly distributed.

Traffic Profile: Two types of traffic profiles were used in the simulation experiments. The two traffic profiles, denoted Characteristic A and Characteristic B, are shown in Tables 3 and 4, respectively. For Characteristic A, all network connections were of a single priority and had a bandwidth of 800 Kbps. The connections' holding times are assumed to be exponentially distributed with a mean of 100 seconds. For Characteristic B, many connection types in terms of bandwidth size, holding time, and delay requirement were used along with four priority levels. The distribution of the priority levels is uniform, i.e., on the average the number of connection requests for each priority level is the same. The bandwidth range for connections is between 320 Kbps to 3200 Kbps. The distribution of bandwidth within this range is also uniform. The connections' holding times are assumed to be exponentially distributed with a mean of 100 seconds.

Performance Metrics: The simulation program collects and reports a large number of statistics which are averaged over the life of simulation. In this study we concentrate on the following four measures.

1. Average Link Reservation Level: This is the percentage of the links' reservable capacity used by connections averaged over all links.
2. Cache Hit Probability: This is the probability that a path obtained from the cache table results in a successful setup.
3. Connection Success Probability: This is the probability that a connection is successfully setup and completes the talk phase. Note that a connection can be rerouted due to preemption or due to link failure and may still be able to complete.
4. Number of Path Computations: This is the total number of path computations during the simulation life for the entire network. There are separate measures for PTSEs sent due to different events.

Nature of experiments: We report three experiments that we have run in the example network shown in Figure 1. Each experiment consists of 3 sets of runs and each set consists of 6 runs. In each experiment, we considered three connection arrival rates (or network load): high, medium, and low. These correspond to sets H, M, and L, respectively. Within each set, Run 1 is for the case in which no path caching is done. For runs 2 through 5, we run Algorithm 1 with the value of N chosen to be 0, 1, 3, and 9, respectively. Run 6 is for Algorithm 2.

Simulation results for Experiment 1: Table 5 presents the simulation results of Experiment 1. Statistics are collected for 10000 seconds of simulation time and runs are made for network load distribution A (Table 1) and network traffic characteristic A (Table 3). As all connections are of same priority note that there is no preemption due to priorities.

When comparing runs $*H$,$*M$,$*L$, where $*$ is a number from 1 to 6, which correspond to the varying network loads, the connection success probability decreased as the network load increased. Also, for any given set of runs, the connection success probability was approximately the same without caching, and with Algorithm 2; the connection success probability for Algorithm 1 was lower than this value and decreased as the parameter N increased.

When comparing runs $*H$,$*M$,$*L$, where $*$ is a number from 1 to 6, which correspond to the varying network loads, the number of

Run	Path Caching Algo.	Connection inter-arrival period for entire network (sec)	N for Algo. 1 (% of PTSEs ignored)	Average link reservation level	Cache hit Probability (if path caching is used)	Connection success probability	Number of path computations
1H	no caching	0.4	N/A	0.791	N/A	0.613	32964
2H	Algo. 1	0.4	0 (0%)	0.792	0.670	0.555	17520
3H	Algo. 1	0.4	1 (50%)	0.789	0.661	0.550	16117
4H	Algo. 1	0.4	3 (75%)	0.783	0.644	0.540	15095
5H	Algo. 1	0.4	9 (90%)	0.769	0.623	0.526	12962
6H	Algo. 2	0.4	N/A	0.815	0.641	0.581	24924
1M	no caching	0.7	N/A	0.681	N/A	0.898	16287
2M	Algo. 1	0.7	0 (0%)	0.639	0.828	0.814	4339
3M	Algo. 1	0.7	1 (50%)	0.614	0.814	0.801	3870
4M	Algo. 1	0.7	3 (75%)	0.615	0.804	0.791	3474
5M	Algo. 1	0.7	9 (90%)	0.607	0.793	0.774	3137
6M	Algo. 2	0.7	N/A	0.732	0.788	0.878	7906
1L	no caching	0.9	N/A	0.506	N/A	0.968	11611
2L	Algo. 1	0.9	0 (0%)	0.451	0.895	0.892	1956
3L	Algo. 1	0.9	1 (50%)	0.452	0.893	0.890	1770
4L	Algo. 1	0.9	3 (75%)	0.455	0.885	0.881	1724
5L	Algo. 1	0.9	9 (90%)	0.454	0.869	0.861	1465
6L	Algo. 2	0.9	N/A	0.574	0.881	0.971	2883

Table 6. Simulation results of Experiment 2.

path computations during the simulation time decreased as the network load decreased. Also, for any given set of runs, the number of path computations was highest when no caching was used; followed by the number of computations when Algorithm 2 was used; the number of path computations for Algorithm 1 was lower than both these values and decreased as the parameter N increased.

When comparing runs $*H$,$*M$,$*L$, where $*$ is a number from 1 to 6, which correspond to the varying network loads, the cache hit probability increased as the network load decreased. Also, for any given set of runs, the cache hit probability for Algorithm 2 was higher than it was for Algorithm 1 although this difference decreased as the network load decreased. For any given set of runs, the cache hit probability for Algorithm 1 decreased as the parameter N increased.

When comparing runs $*H$,$*M$,$*L$, where $*$ is a number from 1 to 6, which correspond to the varying network loads, the average link reservation level decreased as the network load decreased. Also, for any given set of runs, the average link reservation level was somewhat higher for Algorithm 2 than what it was without caching; the average link reservation level for Algorithm 1 was lower than both these values and decreased as the parameter N increased.

It is interesting to observe that for any given set of runs, as the cache hit probability drops, the number of path computations also drops. This appears counterintuitive but can be explained as follows: In Algorithm 1, the cache hit ratio drops as the value of parameter N increases. However, as N increases, the entries in the cache are invalidated less frequently and therefore the cache contains more entries. (The lower cache hit ratio simply reflects the fact that these entries are more outdated and result in a lower percentage of successful connections). As the cache contains more path entries, therefore fewer path computations are attempted, as per Algorithm 1.

Simulation results for Experiment 2: Table 6 presents the simulation results of Experiment 2. Statistics are collected for 10000 seconds of simulation time and runs are made for network load distribution A (Table 1) and network traffic characteristic B (Table 4).

The observations are largely similar to those made for Experiment 1, although the range of network loads considered is lower for Experiment 1. The only differences are the following. For any given set of runs, the cache hit probability for Algorithm 2 was approximately the same as it was for Algorithm 1.

Simulation results for Experiment 3: Table 7 presents the simulation results of Experiment 3. Statistics are collected for 10000 seconds of simulation time and runs are made for network load distribution B (Table 2) and network traffic characteristic B (Table 4).

The observations are largely similar to those made for Experiment 1, although the range of network loads considered is lower for Experiment 1. The only differences are the following.

For any given set of runs, the connection success probability was slightly higher without caching than with Algorithm 2; the connection success probability for Algorithm 1 was lower than this value without caching, and decreased as the parameter N increased.

For any given set of runs, the number of path computations was highest when no caching was used; the number of path computations for Algorithm 1 was lower than this value and decreased as the parameter N increased.

For any given set of runs, the cache hit probability for Algorithm 2 was lower than it was for Algorithm 1 although this difference decreased as the network load decreased.

For any given set of runs, the average link reservation level for Algorithm 1 was approximately the same as that without caching and did not show any variation pattern as N was varied.

Comparison of Algorithms: Algorithm 1 required fewer path computations than Algorithm 2 although its connection success probability was slightly lower. Algorithm 1 performs reasonably well in terms of the connection success probability, particularly at high network loads, and very significantly reduces the number of path computations when compared to the case without path caching. In terms of the connection success probability, Algorithm 2 consistently performs very well, almost as good as without path caching, while reducing the number of path computations significantly.

Run	Path Caching Algo.	Connection inter-arrival period for entire network (sec)	N for Algo. 1 (% of PTSEs ignored)	Average link reservation level	Cache hit Probability (if path caching is used)	Connection success probability	Number of path computations
1H	no caching	0.4	N/A	0.818	N/A	0.712	31788
2H	Algo. 1	0.4	0 (0%)	0.813	0.787	0.668	18501
3H	Algo. 1	0.4	1 (50%)	0.811	0.771	0.666	17399
4H	Algo. 1	0.4	3 (75%)	0.809	0.761	0.654	16157
5H	Algo. 1	0.4	9 (90%)	0.808	0.743	0.628	14998
6H	Algo. 2	0.4	N/A	0.831	0.705	0.666	21734
1M	no caching	0.7	N/A	0.620	N/A	0.927	15291
2M	Algo. 1	0.7	0 (0%)	0.622	0.925	0.901	4811
3M	Algo. 1	0.7	1 (50%)	0.619	0.922	0.896	4242
4M	Algo. 1	0.7	3 (75%)	0.632	0.904	0.880	3899
5M	Algo. 1	0.7	9 (90%)	0.650	0.890	0.863	3393
6M	Algo. 2	0.7	N/A	0.697	0.848	0.895	5699
1L	no caching	0.9	N/A	0.507	N/A	0.966	11413
2L	Algo. 1	0.9	0 (0%)	0.513	0.965	0.952	2788
3L	Algo. 1	0.9	1 (50%)	0.509	0.958	0.946	2290
4L	Algo. 1	0.9	3 (75%)	0.513	0.958	0.944	1774
5L	Algo. 1	0.9	9 (90%)	0.534	0.942	0.929	1477
6L	Algo. 2	0.9	N/A	0.585	0.922	0.957	2109

Table 7. Simulation results of Experiment 3.

6 Conclusions

We have proposed network path caching as a means to reduce the connection setup time for a connection request in a general decentralized connection-oriented network by bypassing the time-consuming path computation phase. This is an important contribution because in high-speed networks where the transmission times are low, the path computation becomes the bottleneck for setting up the connection quickly. The lengthy path computation process also lowers utilization of other network resources. To the best of our knowledge, this is the only study of network path caching in a decentralized/distributed network. We investigated several issues in the network path caching problem. Then we proposed two simple and efficient algorithms for network path caching. We presented a comprehensive simulation study of the two path caching algorithms which were seen to perform very well and significantly reduce the number of path computations. The reductions in the number of path computations in the simulation experiments were anywhere from 80% to 90%. Our simulation study also provided insights into network path caching and network dimensioning problems in order to achieve a desired level of network availability.

References

[1] J. Archibald, J. Baer, "Cache Coherence Protocols: Evaluation using a multiprocessor simulation model," ACM Trans. on Computer Systems, 4(4):273-298, Nov. 1986.

[2] C.-C. Chou, K. G. Shin, "A Distributed Table-Driven Route Selection Scheme for Establishing Real-time Video Channels," Proc. 15th IEEE Int. Conf. on Distributed Computing Systems, 52-59, June 1995.

[3] I. Cidon, I. Gopal, A. Segall, "Fast Connection Establishment in High-Speed Networks." Proc. SIGCOMM 1990.

[4] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," Numerische Mathematik I, 1957.

[5] M. Dubois, C. Scheurich, F. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors," IEEE Computer, 21(2), 9-21, Feb. 1988.

[6] D. Duchamp, "Optimistic Lookup of Whole NFS Paths in a Single Operation," Proc. 1994 Summer Usenix Conference, 161-169, June 1994.

[7] C. Galand, P. Scotten, "Automatic Network Clustering for Fast Path Selection," Proc. 5th Int. Conf. on High Performance Networking, IFIP Transactions C: Communication Systems, July 1994, Elsevier.

[8] M. R. Garey and D. S. Johnson, "Computers and Interactability," W.H. Freeman, San Francisco, 1979.

[9] M. Peyravian, "Providing Different Levels of Network Availability in High-Speed Networks," Proc. Globecom'94, 941-945, 1994.

[10] L. Gün and R. Guérin, "Bandwidth Management and Congestion Control Framework of the Broadband Network Architecture," Computer Networks and ISDN Systems, 26(1), 61-78, Sept. 1993.

[11] B. Janssens, W. K. Fuchs, "The Performance of Cache-Based Error Recovery in Multiprocessors," IEEE Trans. on Parallel and Distributed Systems, 5(10), 1033-1043, Oct. 1994.

[12] D. Kandlur, K. G. Shin, D. Ferrari, "Real-Time Communication in Multihop Networks," IEEE Trans. on Parallel and Distributed Systems, 5(10), 1044-1056, Oct. 1994.

[13] D. E. McDysan and D. L. Spohn, "ATM: Theory and Application," McGraw-Hill, New York, 1994.

[14] M. Nelson, B. Welch, J. Ousterhout, "Caching in the Sprite Network File System," ACM Trans. on Computer Systems, 6(1):134-154, Feb. 1988.

[15] PNNI Draft Spec., ATM Forum 95-0471R14, Dec. 1995.

[16] A. Przygienda, "Link State Routing with QoS in ATM LANs," PhD thesis, Eidgenossischen Technischen Hochschule, Zurich, 1995.