ELSEVIER

Computer Communications 20 (1997) 605–614

computer communications

# Network path caching: Issues, algorithms and a simulation study

Mohammad Peyravian*, Ajay D. Kshemkalyani

*IBM Corporation, P.O. Box 12195, Research Triangle Park, NC 27709, USA*

Received 1 March 1996; accepted 11 September 1996

**Abstract**

Caching of network paths in a connection-oriented communication network provides a means to store computed paths 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. It first identifies and analyses 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. © 1997 Elsevier Science B.V.

*Keywords:* Algorithm; ATM; Connection; Bandwidth; Path selection; Caching; Simulation

## 1. Introduction

With the increasing growth of high-speed wide-area connection-oriented communication networks such as the asynchronous transfer mode (ATM) [14], there is a critical need to be able to set up new connections efficiently and rapidly. 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 [2,5,8,13,16,18]. Running an optimal path selection algorithm is computationally intractable [9]. Although some of the various algorithms in the 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 problem of high overheads for path selection is aggravated due to the following trends in networking. First, the expansion of wide-area networks introduces more nodes and links in the network topology, which adds complexity to the path selection. Second, private networks are being increasingly interconnected using protocols such as ATM Private Network Node Interface (PNNI) [17], which further increases the size of the network topology graph on which a path selection algorithm has to be run. Third, 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. Fourth, 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 [1,3,6], error recovery in multiprocesors [12], distributed systems [7], and network file systems [15]. 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 [4] 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

* Corresponding author. Tel.: +1 919 2547576; e-mail: peyravn@vnet.ibm.com


*PII* S0140-3664(97)00062-5

simple table lookup; hence it uses a very different concept and assumptions than those we use.

We identify and discuss the various issues that arise in manipulating a local cache of network paths computed in the past. 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 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 $N$ 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 $\infty$, and differs in that if a cached path results in a failed connection setup, a path computation attempts to set up the connection before a 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 that are simple to implement, perform very well and substantially reduce the number of path computations.

The rest of this paper is organized as follows: Section 2 presents a simple network model for decentralized/distributed connection-oriented networks. Section 3 discusses the various 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. It then presents two algorithms for network path caching. Section 4 presents a simulation study of network path caching and compares the performance of the two proposed path caching algorithms. Section 5 gives the conclusions.

## 2. Network model

We present below a simple network model for decentralized connection-oriented networks. This model is used for our discussions on the path caching problem. This network model borrows some concepts from ATM PNNI [17], 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 a 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 advertized by broadcasting PTSEs. There is no concept of centralized control, each link manager independent of other link managers decides whether it can accept a new connection when it receives a connection setup request.

There are two reasons why the topology database is never completely synchronized with the real topology: (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.

A connection is set up 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.

When a setup request is processed by a link manager and not enough resources are available, the link manager 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 [10]. 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 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,

[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.

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 performs the connection setup process as described above.

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

For the path caching problem, we assume that a connection request arrives with its QoS requirements, a predefined priority, and a predefined bandwidth. No knowledge of the future arrivals or the holding time 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 connection request.

## 3. Network path caching

### 3.1. Issues

A node that uses a path caching scheme (in addition to path computation) to find routes for connections maintains a 'cache table'. 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, security level, etc.

When a connection requests a path, 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, packet loss probability, etc.). 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.

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

1. Cache update policy. This policy determines whether an entry for a specific path should be inserted into the cache.
2. Cache invalidation policy. This policy determines how an entry in the cache is determined to have become invalid.
3. Cache replacement policy. This policy determines the choice of an entry in the cache that needs to be replaced in order to 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 3.2.

Network path caching varies from the well-studied and implemented forms of caching such as in memory systems [1], multiprocessor memory systems [1,3,6], error recovery in multiprocesors [12], distributed systems [7], and network file systems [15], 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 (on an analog scale) 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 advertized 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.

We make the following assumptions about the network path caching schemes that we consider.

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 discarding cached path entries based on the number of topology updates to any one link in the path to be the most effective and simple criterion from among the criteria we considered.

### 3.2. Algorithms

Based on our analysis of various network path caching schemes, we present two algorithms for network path caching. 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 $\infty$; it differs in that if a cached path results in a failed connection setup, a path computation attempts to set up the connection before a connection request is rejected.

#### 3.2.1. Variables/constants used by algorithms

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

#### 3.2.2. Algorithm 1

##### 3.2.2.1. 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**
2. attempt to set up a path to destination;
3. **if** connection setup succeeds **then**
4. accept the connection request;
5. **else**
6. reject the connection request.
7. **else**
8. invoke path computation;
9. attempt to set up a path to destination;
10. **if** connection setup succeeds **then**
11. **(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;
12. **else**
13. reject the connection request;
14. **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.

*3.2.2.2. Receive a PTSE for link $e_{i_j}$ (**Cache invalidation policy**)*

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

The cache invalidation policy by which a path is removed from the cache has two components. 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 $\infty$. 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.

*3.2.3. Algorithm 2*

*3.2.3.3. 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**
2. attempt to set up a path to destination;
3. **if** connection setup succeeds **then**
4. accept the connection request:
5. **else**
6. invoke path computation;
7. attempt to set up a path to destination;
8. **if** connection setup succeeds **then**
9. **(Cache update policy:)** store the connection's path along with its attributes in the cache.
10. **else**
11. reject the connection request;
12. **else**
13. invoke path computation;
14. attempt to set up a path to destination;
15. **if** connection setup succeeds **then**
16. **(Cache update policy:)** store the connection's path along with its attributes in the cache;
17. **else**
18. reject the connection request;
19. **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.

Algorithm 2 is the same as algorithm 1 except as follows. Algorithm 2 gives a connection one more chance to be set up 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 $\infty$ in algorithm 1.

## 4. Simulation

### *4.1. Model*

We used a connection-level simulation to study path caching and to 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 static objects such as route computation are represented using routines.

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. The program collects and reports a number of statistics as will be described later.

The program simulates the lives of connections from the time they are created until they terminate. Connection inter-arrival 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 pre-computed 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 [11].

Then 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, 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.

Upon acceptance of a connection on a link or removal of a connection 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 send connection setup requests to a particular link almost simultaneously, competing for a limited available bandwidth.

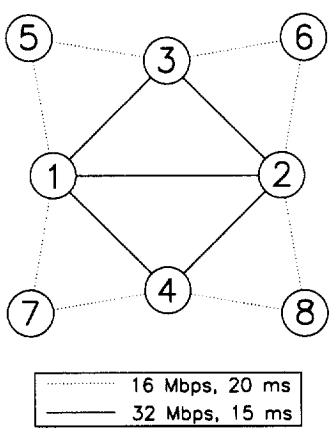



Fig. 1. Network topology and structure.

Table 1
Network load distribution A

| 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 |

Asymmetric load: for experiments 1 and 2.

### 4.2. Experiments

#### 4.2.1. Network structure

We have conducted wide-range simulation with various network conditions (i.e. network topology, number of priority levels, link bandwidth, traffic pattern, etc.) to study path caching and the behavior of these two algorithms.

The following network model (which is an abstraction of a real network) was used in the simulation experiments. The network is two-tiered consisting of 8 nodes and 26 unidirectional links, (Fig. 1). The inner links are 32 Mbps links with a propagation delay of 15 ms, and the outer links are 16 Mbps links with a propagation delay of 20 ms. 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 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.

Table 2
Network load distribution B

| 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 |

Symmetric load: for experiment 3.

Table 3
Network traffic characteristic A (for experiment 1)

| Connection priority | Bandwidth range (Kbps) | Number of BW types in BW range (uniform distribution) | Mean holding time (s) (exponential distribution) | Delay (ms) (uniform distribution) |
|---|---|---|---|---|
| 1 | 800 | 1 | 100 | 10–60 |

*4.2.2. Traffic profile*

In the simulation experiments, two types of traffic profiles were used. The two traffic profiles, denoted characteristic A and characteristic B, are shown in Tables 3 and Table 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 s. 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 and 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 s.

*4.2.3. 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 four measures: average link reservation level, cache hit probability, connection success probability, and number of path computations.

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 set up 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.

*4.2.4. Nature of experiments*

We report three experiments that we have run in the example network shown in Fig. 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.

*4.2.5. Simulation results for experiment 1*

The simulation results of experiment 1 are presented in Table 5. Statistics are collected for 10 000 s 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 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

Table 4
Network traffic characteristic B (for experiments 2 and 3)

| Connection priority | Bandwidth range (Kbps) | Number of BW types in BW range (uniform distribution) | Mean holding time (s) (exponential distribution) | Delay (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 5
Simulation results of experiment 1

| Run | Path caching algorithm | Connection inter-arrival period for entire network (s) | $N$ for algorithm 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 |

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.

For any given set of runs, as the cache hit probability drops, the number of path computations also drops. This

Table 6
Simulation results of experiment 2

| Run | Path caching algorithm | Connection inter-arrival period for entire network (s) | $N$ for algorithm 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 7
Simulation results of experiment 3.

| Run | Path caching algorithm | Connection inter-arrival period for entire network (s) | $N$ for algorithm 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 | 31 788 |
| 2H | Algo. 1 | 0.4 | 0 (0%) | 0.813 | 0.787 | 0.668 | 18 501 |
| 3H | Algo. 1 | 0.4 | 1 (50%) | 0.811 | 0.771 | 0.666 | 17 399 |
| 4H | Algo. 1 | 0.4 | 3 (75%) | 0.809 | 0.761 | 0.654 | 16 157 |
| 5H | Algo. 1 | 0.4 | 9 (90%) | 0.808 | 0.743 | 0.628 | 14 998 |
| 6H | Algo. 2 | 0.4 | N/A | 0.831 | 0.705 | 0.666 | 21 734 |
| 1M | No caching | 0.7 | N/A | 0.620 | N/A | 0.927 | 15 291 |
| 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 | 11 413 |
| 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 |

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, fewer path computations are attempted, as per algorithm 1.

#### 4.2.6. Simulation results for experiment 2

The simulation results of experiment 2 are presented in Table 6. Statistics are collected for 10 000 s 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.

#### 4.2.7. Simulation results for experiment 3

The simulation results of experiment 3 are presented in Table 7. Statistics are collected for 10 000 s 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 increased.

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.

#### 4.2.8. 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.

## 5. Conclusions

In this paper, we 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 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. Comput. Syst. 4 (4) (1986) 273–298.

[2] D. Bertsekas, R. Gallager, Data Networks, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[3] L.M. Censier, P. Feautier, A new solution to coherence problems in multicache systems, IEEE Trans. Comput. 27 (2) (1978) 1112–1118.

[4] C.-C. Chou, K.G. Shin, A distributed table-driven route selection scheme for establishing real-time video channels, 15th IEEE International Conference on Distributed Computing Systems, 1995, pp. 52-59.

[5] E.W. Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik I, 1957.

[6] M. Dubois, C. Scheurich, F. Briggs, Synchronization, coherence, and event ordering in multiprocessors, IEEE Comput. 21 (2) (1988) 9–21.

[7] D. Duchamp, Optimistic look up of whole NFS paths in a single operation, Proceedings of the 1994 Summer Usenix Conference, 1994, pp. 161-169.

[8] C. Galand, P. Scotten, Automatic network clustering for fast path selection, Proceedings 5th International Conference on High Performance Networking, IFIP Transactions C: Communication Systems, Elsevier, Amsterdam, 1994.

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

[10] M. Peyravian, Providing different levels of network availability in high-speed networks, Proceedings of GLOBECOM'94, 1994, pp. 941-945.

[11] L. Gün, R. Guérin, Bandwidth management and congestion control framework of the broadband network architecture, Comput. Networks ISDN Syst. 26 (1) (1993) 61–78.

[12] B. Janssens, W.K. Fuchs, The performance of cache-based error recovery in multiprocessors, IEEE Trans. Parall. Distrib. Sys. 5 (10) (1994) 1033–1043.

[13] D. Kandlur, K.G. Shin, D. Ferrari, Real-time communication in multihop networks, IEEE Trans. Parall. Distrib. Sys. 5 (10) (1994) 1044–1056.

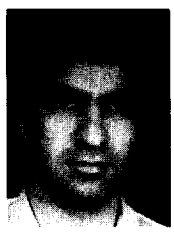
[14] D.E. McDysan, D.L. Spohn, ATM: Theory and Application, McGraw-Hill, New York, 1994.

[15] M. Nelson, B. Welch, J. Ousterhout, Caching in the sprite network file system, ACM Trans. Comput. Syst. 6 (1) (1988) 134–154.

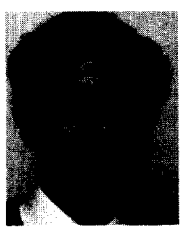
[16] H. Nussbaumer, Teleinformatique II, Presses Polytechniques Romandes, 1987.

[17] PNNI Draft Specification, ATM Forum 95-0471R14, 1995.

[18] A. Przygienda, Link State Routing with QoS in ATM LANs, PhD thesis, ETH, Zurich, 1995.



*Mohammad Peyravian is a senior engineer/scientist at IBM, Research Triangle Park, where he is involved with protocol and algorithm design for high-speed networking technologies. He recieved the B.S. and M.S. degrees from the University of Miami and the Ph.D. degree from the Georgia Insitute of Technology in electrical engineering in 1987, 1989 and 1992, respectively. He does research in the areas of networking, telecommunications and cryptography. Dr. Peyravian has published papers extensively in various journals and proceedings. He has received several IBM invention and technical achievement awards. Dr. Peyravian is a member of the IEEE computer and communications societies, Tau Beta Pi and Eta Kappa Nu.*



*Ajay D. Kshemkalyani received the B. Tech degree in Computer Science and Engineering from the Indian Institute of Technology, Bombay, India, in 1987, and the M.S. and Ph.D. degrees in Computer and Information Science from Ohio State University, USA, in 1988 and 1991, respectively. He is currently an advisory engineer with IBM Corporation, Research Triangle Park, North Carolina, USA. He is also an adjunct assistant professor in Electrical and Computer Engineering at North Carolina State University. His current research interests include distributed computing, operating systems, computer architecture and networking.*