DISTRBUTED CACHE ARCHITECTURE FOR SCALABLE QUALIY OF SERVICS FOR DISRIBUTED NETWORKS

Girish.R.Deshpande[1], Niranjan.S.J[2], Sumayya.Rottiwale

[1]Assistant Professor , Dept. of CSE, GIT,Belgaum-590008
[2] Assistant Professor , Dept. of CSE, KIT,Tiptur- 572202
[3]Student, Dept of CSE,GIT,Belgaum-590008

---***---

***ABSRACT**: Performance degrades significantly in Mobile Ad hoc Networks due to the packet losses. Most of these packet losses result from the Route failures due to network mobility. TCP assumes such losses occur because of congestion, thus invokes congestion control mechanisms such as decreasing congestion windows, raising timeout, etc, thus greatly reduce TCP throughput. However, after a link failure is detected, several packets will be dropped from the network interface queue; TCP will time out because of these packet losses, as well as for Acknowledgment losses caused by route failures.*

***KEY WORDS**: Throughput, timeout, Adhoc-networks, Acknowledgement*

1. INTROUDUCION

1.2 Objective of the Study

Routing Protocols for ad hoc networks can be classified into two major types: proactive and on-demand. Proactive protocols attempt to maintain up-to-date routing information to all nodes by periodically disseminating topology updates throughout the network. In contrast, on demand protocols attempt to discover a route only when a route is needed. To reduce the overhead and the latency of initiating a route discovery for each packet, on-demand routing protocols use route Caches. Due to mobility, cached routes easily become stale. Using stale routes causes packet losses, and increases latency and overhead. In this paper, we investigate how to make on-demand routing Protocols adapt quickly to topology changes. This problem is important because such protocols use route caches to make routing decisions, it is challenging because topology changes are frequent.

To meet the diverse quality-of-service (QoS) requirements of emerging multimedia applications, communication networks should provide end-to-end QoS guarantees. QoS routing is the first step towards this goal. It seeks to find routes that satisfy a set of QoS constraints while achieving overall network efficiency. Therefore, unlike current routing protocols, QoS routing protocols rely on dynamic network state information for computing QoS routes. Frequent route computing and network state updates, especially in large networks, can cause computing and traffic overhead, respectively. Therefore, scalability to large networks has been identified as one of the key issues in designing a QoS routing protocol. It is desirable to minimize these overheads without sacrificing the overall routing performance. In this paper, we address the route computing overhead.

In QoS-capable networks, routes are computed upon arrival of calls. The main advantage of this on-demand approach is its simplicity. However, in large networks with high arrival rates, this approach can cause significant computing load. The pre-computing technique has been proposed and shown to be an effective solution to reduce route computing load. The principle is to compute routes as a background process and use them when a call arrives, therefore reducing the computing load upon each arrival. In this paper, we focus on route caching that has been recently proposed as a solution to reduce the route computing load by reusing already computed routes. In route caching, a newly computed route is stored in a cache for possible use by future calls. Upon arrival of a call, the cache is searched for a route that can satisfy the requested QoS parameters. If no such route is found in the cache, then a new route has to be computed. Because cache size is limited, cache replacement policies should be used when the cache is full. In addition, when several feasible routes are found in the cache, efficient route selection policies are required to maximize network resource efficiency.

While caching is a promising approach to reduce route computing load, we believe that recent proposals have taken very simplistic approaches and several fundamental issues have received no attention. Firstly, the hierarchical architecture of very large networks has not been taken into account. Large networks are potentially partitioned into several domains. A realistic caching scheme should offer an end-to-end solution across multiple domains in a large network. Secondly, for scalability reasons, topology aggregation is identified as an essential technique in large networks with multiple domains. In a large network, an end-to-end route potentially crosses several domains. Considering that each domain represents only an aggregated view of its internal topology and state information, the important question is: how can such an end-to-end route be cached efficiently? Finally, cached routes are subject to changes in the network conditions and should be regularly updated. The simple update techniques that try to periodically re-compute all cached routes can cause considerable computing load.

In this, we propose a novel distributed cache architecture to reduce the route computing load, while addressing the above-mentioned issues.

1.3 Scope of Study

The fundamental component of the distributed cache architecture is its core architecture. The core architecture consists of the cache elements deployed across the networks, basic cache operations that includes saving and reusing the cached routes, and the cache flushing. Before detailing the distributed cache architecture, we briefly discuss our network and routing models.

1.3.1 Network Model

The proposed distributed cache architecture is designed to scale to very large networks. We have considered a large network that is partitioned into several domains. Each domain is connected to the rest of the network through its border nodes. Domains are subject to topology aggregation, a commonly used technique to reduce the overhead traffic caused by network state updates [1].

1.3.2 Routing Model

Link state routing model similar to the ATM PNNI standard [2]. Calls arrive at the border nodes of network domains. QoS routes are computed and setup between two border nodes. All routes are bi-directional. In link state routing, network state information is periodically distributed in the network, so that each border node maintains a link state database containing topology and network state information. In source routing, a route is computed at the node where the call has arrived. If the route has to cross several domains, only a skeleton route is computed for those domains. When the route setup request enters a domain at a border node, the border node constructs the detailed structure of the route for that domain. Finally, we assume a bandwidth-based QoS model, so that calls request for bandwidth as their QoS parameter. This simple, yet realistic, model can represent a broad range of applications, while helps us to focus on other aspects of the cache architecture.

1.3.3 Core Distributed Cache Architecture

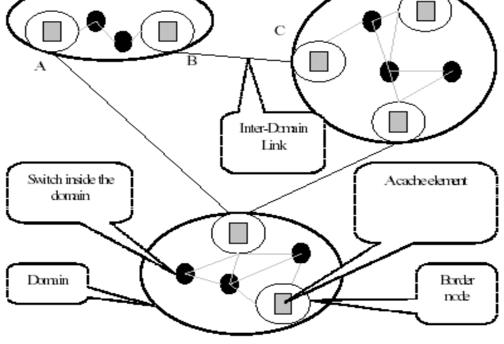

Fig. 1.1: Deployment of the core distributed cache architecture Across a network with multiple domains

Figure 1.1 shows how the core distributed cache architecture is deployed in a large network. As shown, every border node in a domain maintains a cache element. These cache elements are connected together by means of pointers (network addresses) and form the basis of the distributed cache architecture. Each cached route is stored across several cache elements in a distributed fashion and in the form of several segments. By segment, we mean the part of a route that is laid between two adjacent border nodes. Routes enter the border nodes in the form of an ingress segment and leave the border nodes in the form of an egress segment. When a route crosses a border node, the cache element at the border node stores information about both ingress and egress segments. For example, in Figure 1.1, assume that upon arrival of a call at the border node A, a route has been successfully computed between the border nodes A and C. This route also passes through the border node B and is stored across the cache elements at the border nodes A, B, and C as follows. The cache element at the border node A stores the A-B segment, the cache element at the border node B stores the A-B and the B-C segments, and the cache element at the border node C stores the B-C segment. Note that the cache elements at two ends of a route store only one segment. Figure 1.2 shows the internal structure of the cache element at a border node, identifying the following fields:

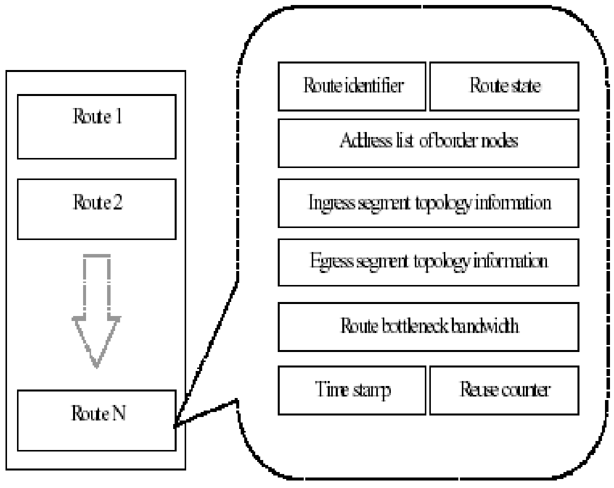
Route 1
Route 2
Route N
Route identifier
Route state
Address list of border nodes
Ingress segment topology information
Egress segment topology information
Route bottleneck bandwidth
Time stamp
Reuse counter

Fig.1.2: The internal structure of a cache element at a border node

· **Route identifier**. This is a number assigned to a route when it is cached. The combination of this number and the address of the source node help to identify different segments of a route that are stored across different cache elements.

· **Route state.** A cached route can be in one of several states.

· **Address list of border nodes.** This is a list containing addresses of all border nodes that a route has crossed. This list is used to traverse a route. It contains the addresses of source, destination, and all intermediate border nodes that a route has crossed. This list is arranged as a bi-directional linked list.

· **Route bottleneck bandwidth**. The QoS parameter used in our architecture. This is the minimum available bandwidth in an end-to-end cached route.

· **Topology information of ingress and egress segments**. For each route, each cache stores topology information of up to two segments connected to it. The topology information is used by operations such as route selection or cache snooping.

1. It is essential to store topology information of both ingress and egress segments of the cached routes because cached routes can be re-used from both ends. Therefore, navigation through segments may be required in both directions.

2. Each segment (either ingress or egress) belongs only to a specific cached route. Segments are not shared between different cached routes. In other words, each cached route is saved in the form of multiple interconnected segments.

· Time stamp. This field keeps the time when the route was computed and stored in the cache for the first time.

· Re-use counter. This is a counter that is increased by one each time the cached route is re-used successfully.

1.3.4 Using the Cached Routes

When a new call arrives at a border node, based on the requested bandwidth and the destination address, the cache at the border node is searched for a feasible route. If no feasible route is found in the cache, a route has to be computed. If only one feasible route is found in the cache, the route setup will be started as follows. Starting from the cache element at the source border node, where the route is found, the first segment of the route is extracted from the cache and is setup in the network. If the segment is setup successfully, the corresponding entry in the cache element is labelled as "in-use" and route setup proceeds to the next border node. As setup proceeds across border nodes, the address list and the route identifier fields are used to keep track and find the route information and the next border node. This process continues until the last segment of the route is setup. If at any stage the route setup fails, all corresponding cache entries will be labelled as "stale" and an on-demand route computing will be started. If more than one feasible route is found in the cache, one of them should be selected to go through setup process. In such situation, we use one of the following route selection policies:

· **Most Recently Computed (MRC**). This policy aims to minimize the probability of selecting an obsolete cached route by choosing the youngest route. The principle is that the younger route has been subject to less fluctuation in the network states.

· **Least Frequently Used (LFU).** This policy attempts to choose the least popular route, therefore increasing the availability of more popular routes for future calls.

· **Widest**. This policy chooses the widest one among all feasible routes. Its goal is to balance the network load. It also attempts to reduce bandwidth fragmentation.

· **Tightest.** This policy attempts to choose the best fit. It leaves wider routes for future calls with possibly higher bandwidth requirement.

1.4 Advantages of Distributed Cache Architecture

- It can scale to very large networks since it has a distributed nature. It has been designed to be easily deployable in networks with multiple domains.
- A cache content management/replacement technique called cache flushing has been developed. It suits the distributed nature of our cache architecture. The traditional cache replacement techniques take action when the cache is full and a new entry has to be added. In contrast, the cache flushing works in the background and always maintains some free space in the cache elements of the proposed distributed cache architecture.
- Once a route is cached, our distributed cache architecture does not rely on network state updates and operates independently. Therefore, our cache architecture does not suffer from inaccuracy of the network state information caused by topology aggregation, delays in the distribution of the network states, or network state update interval. Instead, our architecture directly monitors only those parts of the network that are more likely to be used. In this way, it intelligently adapts to the changes in the network states. This is done by a novel technique called cache snooping, which has been developed to alleviate the effects of network state fluctuations on the cached routes with minimum overhead.
- Cache snooping increases the routing tolerance to inaccurate network state information. This improves the overall routing performance, especially in the presence of highly inaccurate network state information.
- We have considered simplicity as a key design issue for distributed cache architecture and its associated techniques. Therefore, the distributed cache architecture relies on simple but efficient algorithms and techniques so that the added complexity to the network is minimized.

2. REASEARCH BACKGROUND

Maltz et al. [3] were the first to study the cache performance of DSR. They found that the majority of ROUTE REPLIES are based on cached routes, and only 59% of ROUTE REPLIES carry correct routes. They also observed that even ROUTE REPLIES from the target are not 100% correct, since routes may break while a ROUTE REPLY is sent back to the source node. They concluded that efficient route maintenance is critical for all routing protocols with route caches.

Holland and Vaidya [4] showed that stale routes degrade TCP performance. They observed that TCP experiences repeated route failures due to the inability of a TCP sender's routing protocol to quickly recognize and remove stale routes from its cache. This problem is complicated by allowing nodes to respond to route discovery requests with routes from their caches, because they often respond with stale routes.

Hu and Johnson [5] studied the design choices for cache structure, cache capacity, and cache timeout. They proposed several adaptive timeout mechanisms for link caches. In Link-MaxLife, the timeout of a link is chosen according to a stability table in which a node records its perceived stability of each other node. A node chooses the shortest-length path that has the longest expected lifetime. When a link is used, the stability metric for both endpoints is incremented by the amount of time since the link was last used, multiplied by some factor. When a link is observed to break, the stability metric for both endpoints is multiplicatively decreased by a different factor. Link-MaxLife was shown to outperform other adaptive timeout mechanisms. Marina and Das proposed wider error notification and timer-based route expiry. Wider error notification aims at increasing the speed and extent of ROUTE ERROR propagation. With wider error notification, a node receiving a ROUTE ERROR rebroadcasts the packet if the node caches a route containing the broken link and the route was used to forward packets. There are three differences between this technique and our work.

First, with this technique, a node detecting a link failure does not know which neighbors have cached the link, and thus cannot notify all nodes that need to be notified. Second, this technique uses broadcast. Broadcast will introduce overhead to the nodes that do not cache a broken link, and some nodes that cached a broken link may not receive notifications because broadcast is unreliable. Broadcast will also interfere other transmissions. In contrast, our algorithm uses unicast packets to notify only the nodes that have cached a broken link. Third, stale routes propagated through ROUTE REPLIES and cached for future use will not be removed. Under timer-based expiry, an average lifetime is assigned to all routes, which is obtained using the lifetime of all broken routes in the past. This approach works well when routes break uniformly, but mobility may not be uniform in time or space. Lou and Fang proposed an adaptive link timeout mechanism that adjusts link lifetime based on the moving average of link lifetime statistics.

2.1 Goal of Proposed Work:

The goal of this paper is to proactively disseminating the broken link information to the nodes that have that link in their caches. We define a new cache structure called a cache table and present a distributed cache update algorithm. Each node maintains in its cache table the information necessary for cache updates. When a link failure is detected, the algorithm notifies all reachable nodes that have cached the link in a distributed manner. We show that the algorithm outperforms DSR with path caches and with Link-MaxLife [5], an adaptive timeout mechanism for link caches.

2.4 Analysis of Existing Network

- TCP performance degrades significantly in Mobile Ad hoc Networks due to the packet losses. Most of these packet losses result from the Route failures due to network mobility.
- TCP assumes such losses occur because of congestion, thus invokes congestion control mechanisms such as decreasing congestion windows, raising timeout, etc, thus greatly reduce TCP throughput.
- However, after a link failure is detected, several packets will be dropped from the network interface queue; TCP will time out because of these packet losses, as well as for Acknowledgement losses caused by route failures.
- There is no intimation information regarding about to the failure links to the Node from its neighboring Node's. So that the Source Node cannot able to make the Route Decision's at the time of data transfer.
- The Stale route causes packet losses if packets cannot be salvaged by intermediate nodes.
- The stale route increases packet delivery latency, since the MAC layer goes through multiple retransmissions before concluding a link failure.
- Use Adaptive time out mechanisms.
- If the cache size is set large, more stale routes will stay in caches because FIFO replacement becomes less effective.

2.5 Proposed System

- Prior work in DSR used heuristics with ad hoc parameters to predict the lifetime of a link or a route. However, heuristics cannot accurately estimate timeouts because topology changes are unpredictable.

- Prior researches have proposed to provide link failure feedback to TCP so that TCP can avoid responding to route failures as if congestion had occurred.
- We propose proactively disseminating the broken link information to the nodes that have that link in their caches. We define a new cache structure called a cache table and present a distributed cache update algorithm. Each node maintains in its cache table the Information necessary for cache updates.
- The Source Node has the information regarding about the Destination and the Intermediate Node links failure, So that it is useful from Packet loss and reduce the latency time while data transfer throughout the Network.
- Proactive cache updating also prevents stale routes from being propagated to other nodes.
- We defined a new cache structure called a cache table to maintain the information necessary for cache updates. We presented a distributed cache update algorithm that uses the local information kept by each node to notify all reachable nodes that have cached a broken link. The algorithm enables DSR [7] to adapt quickly to topology changes.
- The algorithm quickly removes stale routes no matter how nodes move and which traffic model is used.

3. ARCHITECTURAL DESIGN:

3.1 The Distributed Cache Update Algorithm

In this section, we first describe the cache staleness issue. We then give the definition of a cache table and present two algorithms used to maintain the information for cache updates.

On-demand Route Maintenance results in delayed awareness of mobility, because a node is not notified when a cached route breaks until it uses the route to send packets. We classify a cached route into three types:

pre-active, if a route has not been used;

active, if a route is being used;

post-active, if a route was used before but now is not.

It is not necessary to detect whether a route is active or post-active, but these terms help clarify the cache staleness issue. Stale pre-active and post-active routes will not be detected until they are used. Due to the use of responding to ROUTE REQUESTS with cached routes, stale routes may be quickly propagated to the caches of other nodes. Thus, pre-active and post-active routes are important sources of cache staleness.

When a node detects a link failure, our goal is to notify all reachable nodes that have cached that link to update their caches. To achieve this goal, the node detecting a link failure needs to know which nodes have cached the broken link and needs to notify such nodes efficiently. This goal is very challenging because of mobility and the fast propagation of routing information.

Our solution is to keep track of topology propagation state in a distributed manner. Topology propagation state means which node has cached which link. In a cache table, a node not only stores routes but also maintain two types of information for each route:

(1) How well routing information is synchronized among nodes on a route.

(2) Which neighbor has learned which links through a ROUTE REPLY. Each node gathers such information during route discoveries and data transmission.

The two types of information are sufficient; because each node knows for each cached link which neighbors have that link in their caches. Each entry in the cache table contains a field called Data Packets. This field records whether a node has forwarded 0, 1, or 2 data packets. A node knows how well routing information is synchronized through the first data packet.

When forwarding a ROUTE REPLY, a node caches only the downstream links; thus, its downstream nodes did not cache the first downstream link through this ROUTE REPLY. When receiving the first data packet, the node knows that upstream nodes have cached all downstream links. The node adds the upstream links to the route consisting of the downstream links. Thus, when a downstream link is broken, the node knows which upstream node needs to be notified.

The node also sets Data Packets to 1 before it forwards the first data packet to the next hop. If the node can successfully deliver this packet, it is highly likely that the downstream nodes will cache the first downstream link; otherwise, they will not cache the link through forwarding packets with this route. Thus, if Data Packets in an entry is 1 and the route is the same as the source route in the packet encountering a link failure, downstream nodes did not cache the link. However, if Data Packets is 1 and the route is different from the source route in the packet, downstream nodes cached the link when the first data packet traversed the route. If Data Packets is 2, then downstream nodes also cached the link, whether the route is the same as the source route in the packet. Each entry in the cache table contains a field called Reply Record. This field records which neighbor learned which links through a ROUTE REPLY. Before forwarding a ROUTE REPLY, a node records the neighbor to which the ROUTE REPLY is sent and the downstream links as an entry. Thus, when an entry contains a broken link, the node will know which neighbor needs to be notified. The algorithm uses the information kept by each node to achieve distributed cache updating.

When a node detects a link failure while forwarding a packet, the algorithm checks the Data Packets field of the cache entries containing the broken link:

(1) If it is 0, indicating that the node has not forwarded any data packet using the route, then no downstream nodes need to be notified because they did not cache the broken link.

(2) If it is 1 and the route being examined is the same as the source route in the packet, indicating that the packet is the first data packet, then no downstream nodes need to be notified but all upstream nodes do.

(3) If it is 1 and the route being examined is different from the source route in the packet, then both upstream and downstream nodes need to be notified, because the first data packet has traversed the route.

(4) If it is 2, then both upstream and downstream nodes need to be notified, because at least one data packet has traversed the route.

The algorithm notifies the closest upstream and/or downstream nodes and the neighbors that learned the broken link through ROUTE REPLIES. When a node receives a notification, the algorithm notifies selected neighbors: upstream and/or downstream neighbors, and other neighbors that have cached the broken link through ROUTE REPLIES. Thus, the broken link information will be quickly propagated to all reachable nodes that have that link in their caches.

3.2 Modules Used

Module 1: Route Request

When a source node wants to send packets to a destination to which it does not have a route, it initiates a Route Discovery by broadcasting a ROUTE REQUEST. The node receiving a ROUTE REQUEST checks whether it has a route to the destination in its cache. If it has, it sends a ROUTE REPLY to the source including a source route, which is the concatenation of the source route in the ROUTE REQUEST and the cached route. If the node does not have a cached route to the destination, it adds its address to the source route and rebroadcasts the ROUTE REQUEST. When the destination receives the ROUTE REQUEST, it sends a ROUTE REPLY containing the source route to the source. Each node forwarding a ROUTE REPLY stores the route starting from itself to the destination. When the source receives the ROUTE REPLY, it caches the source route.

Module 2: Route Maintenance

Route Maintenance, the node forwarding a packet is responsible for confirming that the packet has been successfully received by the next hop. If no acknowledgement is received after the maximum number of retransmissions, the forwarding node sends a ROUTE ERROR to the source, indicating the broken link. Each node forwarding the ROUTE ERROR removes from its cache the routes containing the broken link.

Module 3: Cache Updating

When a node detects a link failure, our goal is to notify all reachable nodes that have cached that link to update their caches. To achieve this goal, the node detecting a link failure needs to know which nodes have cached the broken link and needs to notify such nodes efficiently. Our solution is to keep track of topology propagation state in a distributed manner.

In a cache table, a node not only stores routes but also maintain two types of information for each route: (1) how well routing information is synchronized among nodes on a route; and (2) which neighbor has learned which links through a ROUTE REPLY. Each node gathers such information during route discoveries and data transmission, without introducing additional overhead. The two types of information are sufficient; because each node knows for each cached link which neighbors have that link in their caches.

3.3 Example Usage

Example 1:

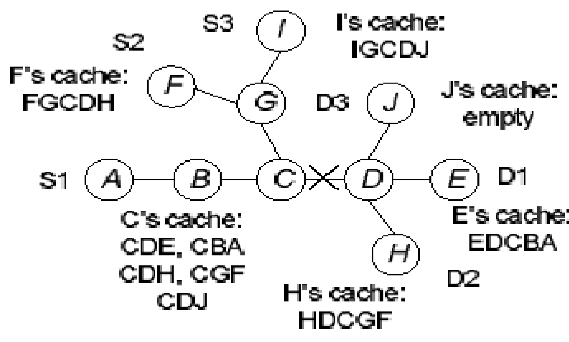

Fig. 2: Routing Caching in DSR

We show an example of the cache staleness issue. In Figure 2 assume that route ABCDE is active, route FGCDH is post-active, and route IGCDJ is pre-active. Thus, node C has cached both the upstream and the downstream links for the active and post-active routes, but only the downstream links, CDJ, for the pre-active route. When forwarding a packet for source A, node C detects that link CD is broken. It removes stale routes from its cache and sends a ROUTE ERROR to node A. However, the downstream nodes, D and E, will not know about the broken link. Moreover, node C does not know that other nodes also have cached the broken link, including all the nodes on the post-active route, F, G, D, and H, and the upstream nodes on the pre-active route, I and G. Stale routes have several adverse effects:

- Using stale routes causes packet losses if packets cannot be salvaged by intermediate nodes;
- Using stale routes increases packet delivery latency, since the MAC layer goes through multiple retransmissions before concluding a link failure;
- Using stale routes increases routing overhead, since the node detecting a link failure will send a ROUTE ERROR to the source node;
- Using stale routes degrades TCP performance, since TCP will invoke congestion control mechanisms for packet losses caused by route failures.

We use algorithms add Route and find Route to collect and maintain the information necessary for cache updates. Algorithm add Route is called when a node attempts to add a route to its cache table.

Example 2:

We use the network shown in Figure 5.3 for our examples. Initially, there are no data flows and all caches are empty. We use S-D for Source Destination and DP for Data Packets in the tables describing the content of caches.

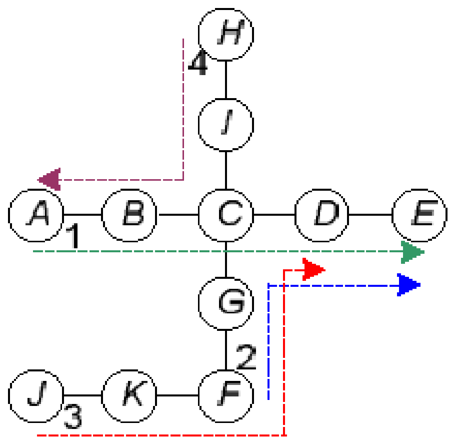

Fig. 3: Networks Used in Routing Protocols

Node A initiates a route discovery to node E, and E sends a ROUTE REPLY to A. Each node forwarding the ROUTE REPLY creates a cache table entry. For instance, node C creates an entry consisting of four fields: the route consisting of the downstream links, the source and destination pair, the number of data packets the node has forwarded using the route, and which neighbor will learn which links through the ROUTE REPLY.

	Route	S-D	DP	ReplyRecord
C:	CDE	A E	0	B ← CDE

When node A receives the ROUTE REPLY, it creates a cache table entry.

	Route	S-D	DP
A:	ABCDE	A E	0

When node A uses this route to send the first data packet, it increments Data Packets to 1. Each intermediate node receiving the first data packet updates its cache table entry. For instance, node C increments Data Packets to 1, adds the upstream links to route CDE, and removes the Reply Record entry, as the complete route indicates that the upstream nodes, A and B, have cached the downstream links, CDE.

	Route	S-D	DP
C:	ABCDE	A E	1

When node E receives the first data packet, it creates a cache table entry

	Route	S-D	DP
E:	ABCDE	A E	1

When a node on this route receives the second data packet, it increments Data Packets to 2. Assume that after transmitting at least two data packets for flow 1, node C receives a ROUTE REQUEST from G with source F and destination E. Before sending a ROUTE REPLY to node G, node C adds a Reply Record entry to its cache

	Route	S-D	DP	ReplyRecord
C:	ABCDE	A E	2	G ← CDE

Reply Record Before sending a ROUTE REPLY to node F, node G creates a cache table entry.

	Route	S-D	DP	ReplyRecord
G:	GCDE	F E	0	F ← GCDE

When node F receives the ROUTE REPLY, it creates a cache table entry

	Route	S-D	DP
F:	FGCDE	F E	0

When node C receives a ROUTE REQUEST from I with source H and destination A, it adds the second Reply Record entry to its cache.

	Route	S-D	DP	ReplyRecord	ReplyRecord
C:	ABCDE	A E	2	G ← CDE	I ← CBA

A node creates a cache table entry to store source route if a route consisting of the downstream links in the source route does not exist in its cache. Assume that flow 2 starts. When it reaches node D, node D adds the second entry to its cache, because the sub-route CDE has been completed by flow 1. When receiving the first data packet, node D knows that its upstream nodes have cached the downstream link DE.

	Route	S-D	DP
D:	ABCDE	A E	2
D:	FGCDE	F E	1

When node F receives a ROUTE REQUEST from node K with source J and destination D, it extends its cache entry.

	Route	S-D	DP	ReplyRecord
F :	FGCDE	F E	2	K ← FGCD

4. CONCLUSION & SCOPE FOR FUTURE WORK

4.1 Conclusion

In this paper, we have introduced a distributed cache architecture to reduce the route computing load caused by the execution of the QoS routing algorithms, assuming bandwidth-based QoS requirements. Considering the distributed nature of cache architecture, to minimize the added complexity to the network, simplicity is a key design issue in our approach. Therefore, we have designed and incorporated simple yet efficient algorithms and techniques. The distributed cache architecture is easily scaled to large hierarchical networks. The cached routes are stored in the form of multiple interconnected segments across several cache elements. Cache snooping was proposed as a distributed technique to alleviate the effects of rapid changes in the network states so that the route computing load is reduced more efficiently. In addition, cache snooping helps to increase the tolerance of QoS routing in the presence of inaccurate network state information caused by long network state update intervals. This means that the proposed distributed cache architecture can also reduce the overhead traffic caused by the frequent distribution of the network state information, while achieving a good performance. Also route borrowing was introduced as a simple but effective technique to improve the performance of the distributed cache architecture. While cache snooping improves the cache hit ratio, route borrowing significantly increases the cache utilization ratio. We considered realistic network topologies, routing algorithms, traffic models, and topology aggregation techniques to show that our solution is deployed in real life large networks.

4.2 Scope for Future Work

As with other applications, there is certainly a scope for improvement in this application too. New modules are in pipeline for to increase the compatibility of the project. Once these improvements have been done, the majority of the features that make an application an excellent one would be there and the usage would become wider and more expensive. Here, there a some of decision's for to make our project effectively and efficiently in the future

- ➢ Implement Non-Adaptive Routing or Link state Routing while Message Transfer
- ➢ Send the messages in the Encrypted format show that the Network hackers are not able to interfere while transmission.
- ➢ Establish Key agreement process between the Source and the Destination nodes
- ➢ Implement the Bidirectional route information between the source and the destination nodes.

REFERENCES

[1] B. Awerbuck, Y. Du, B. Khan, and Y. Shavitt. Routing Through Networks with Hierarchical Topology Aggregation. Journal of High Speed Networks, 7(17):57-73, 1998,

[2] ATM Forum. Private Network-Network Interface specification version 1.0. Technical report af-pnni-0055.000, ATM forum, March 1996.

[3] J. Broch, D. Maltz, D. Johnson, Y.-C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc networkrouting protocols. In Proc. 4th ACM MobiCom, pp. 85–97, 1998.

[4] G. Holland and N. Vaidya. Analysis of TCP performance over mobile ad hoc networks. In Proc. 5th ACM MobiCom,pp. 219–230, 1999.

[5] Y.-C. Hu and D. Johnson. Caching strategies in on-demand routing protocols for wireless ad hoc networks. In Proc. 6th ACM MobiCom, pp. 231–242, 2000.

[6] D. Johnson and D. Maltz. Dynamic Source Routing in ad hoc wireless networks. In Mobile Computing, T. Imielinski and H. Korth, Eds, Ch. 5, pp. 153–181, Kluwer, 1996.

[7] D. Maltz, J. Brooch, J. Jetcheva, and D. Johnson. The effects of on-demand behavior in routing protocols for multi-hop wireless ad hoc networks. IEEE J. on Selected Areas in Communication, 17(8):1439–1453, 1999.

[8]
http://scitec.uwichill.edu.bb/cmp/online/cs22l/waterfall_model.htm

[9] Software Engineering: A Practitioner's Approach, seventh edition by Roger Pressman