A Modified Approach to Dynamic Source Routing in Mobile Ad-Hoc Networks*

GAUTAM CHAKRABARTI
Mighigan State University, USA

SANDEEP KULKARNI
Michigan State University, USA

Abstract

To ensure uninterrupted communication in a mobile ad-hoc network, efficient route discovery is crucial when nodes move and/or fail. Hence, protocols such as Dynamic Source Routing (DSR) precompute alternate routes before a node moves and/or fails. In this paper, we modify the way these alternate routes are maintained and used in DSR, and show that these modifications permit more efficient route discovery when nodes move and/or fail. Our simulation results show that maintenance of these alternate routes (without affecting the route cache size at each router) increases the packet delivery ratio without incurring any extra traffic overhead. We also show that our approach enables us to provide QoS guarantees by ensuring that appropriate bandwidth will be available for a flow even when nodes move. Towards this end, we show how reservations can be made on the alternate routes while maximizing the bandwidth usage in situations where nodes do not move.

Keywords

Ad-hoc Networks, Routing, Load balancing, Bandwidth Reservation, QoS

1 Introduction

The recent emergence of mobile devices has increased the relevance of mobile ad-hoc networks. Such networks are formed by a collection of wireless nodes that are free to move about, often in a restricted amount of space. Such movement of nodes results in temporary networks, formed by a set of nodes due to their proximity to each other. Often, all mobile hosts in a network may not be in the

*Email: {chakra10, sandeep}@cse.msu.edu
Web: `http://www.cse.msu.edu/~{chakra10,sandeep}`
Tel: +1-517-355-2387

This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, and a grant from Michigan State University.

transmission range of each other. In such scenarios, each node acts not only as a host sending (respectively, receiving) data to (respectively, from) another mobile host, but also as a router. Thus, nodes use multi-hop routes to reach their destinations. This task of routing data through multiple hops to the destination becomes all the more challenging due to the possibility of *router movement* in the middle of transmission.

To ensure uninterrupted communication in the presence of node movement, it is necessary to discover a new route efficiently. More specifically, if an intermediate node finds that it cannot reach the next hop on the route to the destination, then that intermediate node needs to find an alternate route to the destination. For uninterrupted communication, it is necessary that a new route be available as soon as a node becomes unreachable. In other words, it is necessary to identify alternate routes even before a node moves away or fails.

While identifying and using alternate routes, it is important to ensure that other flows using that alternate route are not affected and that appropriate bandwidth is available for the *rerouted* flow. However, one cannot simply reserve the entire requested bandwidth for a data transmission on the alternate routes, as it will lead to underutilization of the network bandwidth in the case where no nodes move. Based on the above discussion, in this paper, we focus on two conflicting goals that need to be met in ad-hoc networks: (1) ensuring the availability of the alternate route that provides the required bandwidth, and (2) maximizing the available bandwidth when no node moves/fails.

We proceed as follows: First we begin with the observation that in source-based routing when an intermediate node detects that it cannot reach the next hop in a source route, it can use precomputed alternate routes to other intermediate nodes mentioned in the packet header, to transmit the packet. This can be achieved by caching routes to all the nodes in the network. If a node caches such routes, then it is highly probable that it may have an alternate route to some of the intermediate nodes even if it does not have an alternate route to the final destination.

Second, we note that a node may have *multiple* alternate routes to reach another node in the network. Hence, a rerouted transmission needs to consume only a small amount of bandwidth from each individual alternate route, and thus cause minimal interference with other flows. This implies that it is important to study the number of alternate routes to a particular node that are available in general, and to see how many of them can be used for rerouting. Availability of such valid alternate routes would help in using the available bandwidth efficiently. It would also enable us to deal with the case where some alternate routes have become invalid due to node movement.

Third, we consider the effect of the rerouted flow on other flows in the network. As argued earlier, reserving the requested bandwidth in a single alternate route would result in inefficient bandwidth usage if there is no node movement/failure. Also, trying to reserve parts of the requested bandwidth along multi-

ple alternate routes would generate high control overhead. Generating more control overhead may in effect be even more harmful since there is no guarantee how long the reserved alternate routes will remain valid. Hence, we cannot afford to explicitly reserve bandwidth along the alternate route(s). At the same time, we need to have at least an *implicit* reservation along alternate routes to maximize the delivery ratio of rerouted packets.

Fourth, we consider the issue of reusing the existing route while determining alternate routes. The reuse of the existing route will enable us to use the bandwidth already reserved on that route. Also, if the number of new links on the alternate route is minimized then there is a greater potential that sufficient bandwidth will be available on that alternate route. Moreover, if the existing route is being reused, it is more likely to be valid (except for the next hop) than other cached routes.

In the context of the four issues discussed above, this paper takes up the issues of *bandwidth reservation* and *route maintenance* in the face of node movement. Our approach provides better route availability between hosts and better delivery of data packets compared to the existing Dynamic Source Routing Protocol presented in [1]. We use the network simulator *ns* [2] to compare the performance of DSR and our protocol. We present simulation results showing the performance improvements we have achieved with our modifications.

Organization of the paper. The rest of the paper is organized as follows: In Section 2, we describe the DSR protocol for ad-hoc networks and the optimizations added to it. In Section 3, we present our approach, and in Section 4, we present simulation results comparing the performance of DSR and our protocol. Finally, in Section 5, we present related work and conclude in Section 6.

2 Dynamic Source Routing

In Dynamic Source Routing protocol for ad-hoc networks, the sender of a packet determines the route that the packet should follow in order to reach the destination. The entire *source route* is inserted in the packet header. A node receiving the packet determines from the header if it is an intermediate host in the route or if it is the final destination. If it is an intermediate node, it forwards the packet to the next hop as specified in the source route. If it is the final destination, the packet is instead delivered to the network layer. The basic operations done by the protocol are *route discovery* and *route maintenance*.

Route discovery. When a host wants to send a packet, it consults its cache of previously discovered routes to determine if it has a route to the destination. If the host does not have a valid route, it broadcasts a *route request* packet containing its own address and the destination address for which it is requesting a route. Each host receiving the route request consults its cache to see if it has a route to the destination. If the cache does not yield a valid route, the node inserts its address in the packet header and broadcasts it again. Thus, the route taken by the packet gets stored in its header. If no intermediate host has a cached route and the destination is *reachable* from the source, the route request packet will ultimately

reach the destination. When a packet reaches the destination or a node with a valid cached route, the node replies to the source with a *route reply* packet containing the source route discovered.

Route maintenance. DSR uses a hop-by-hop acknowledgment at the data link level to detect failure of a link. If a host trying to transmit a packet to its next hop determines that the link has failed, a *route error* packet is sent to the source of the packet giving addresses of nodes at both the ends of the failed link. The source host removes the hop from the cache and truncates routes containing that link at the failed point.

Previous Improvements to DSR. The Dynamic Source Routing protocol as presented in [3] has undergone some modifications in [1], some of which are:

- IEEE 802.11 requires an RTS/CTS/Data/ACK exchange for all unicast packets. This implies that data packets can be transmitted only through bidirectional routes. The source routing protocol is modified to use only bidirectional links for data transfer.

- Nodes are modified to work in *promiscuous* mode. A node running this protocol thus overhears packets even if it is not in the source route. This approach allows nodes to learn about route failures by tapping *route error* packets. Moreover, if a node overhears a packet which has its own address listed in the unprocessed portion of the source route, it implies that the node is set to receive the packet through a longer route. In that case, it can let the packet source know about the available shorter route by sending a *gratuitous route reply*.

- When a node forwarding a packet to its next hop discovers that the node is unreachable either due to a link failure or a node movement/failure, it consults its cache to find if it can find an alternate route to the destination. If the node has another route to the destination, it changes the source route appropriately and forwards the packet according to this new route.

The version of DSR implemented by CMU Monarch [4] has a few additional modifications. Notably, when an intermediate node forwards a packet to its next hop, it snoops into the unprocessed portion of the route in the packet header to get a route to the destination. This route is cached and can be used as an alternate route if some existing route fails. This version of DSR has been ported to the Network Simulator, *ns* [2]. We compare our protocol with this implementation.

3 Proposed Improvements

In this section, we describe our approach for route maintenance and bandwidth allocation. First, we describe our approach to validate the hypothesis: when an intermediate node needs to find an alternate route due to node movement/failure, it should try to reuse the existing source route as much as possible. Then, in Section 3.1, we discuss our approach to load balancing and the modification to cache

implementation. Subsequently, in Section 3.2, we discuss our approach for route reservation when a node starts communicating with a destination and when an intermediate node routes packets on alternate routes. The simulation results for our algorithms are presented in Section 4.

When an intermediate host in a source route cannot reach its next hop along the route, it looks up its cache for alternate routes to reach the destination. If it finds an alternate route, it modifies the source route accordingly and transmits the packet to the newly selected next hop. While selecting such alternate routes, our approach strives to maximize the part of the original route that is preserved in the new route. As mentioned in the Introduction, this helps in reducing interference with other flows and in increasing the probability of finding bandwidth on the alternate routes.

With this intuition, in our protocol, when a node detects that its next hop along a route is unreachable, it tries to find an alternate route to the node that lies at a distance of 2 hops (mentioned henceforth as *hop-2 neighbor*) along the route. An alternate route to the hop-2 neighbor would enable the intermediate node to remove the next hop neighbor from the route, but keep the rest of the route intact. It follows that, if successful, this leads to maximum reuse of the existing source route. If such an alternate route is not present in the cache, our protocol searches for routes to nodes farther away in the source route. We have implemented and tested two versions of our approach:

1. An intermediate node starts scanning the source route from its hop-2 neighbor towards the destination. For each node in this route, it searches its cache for an alternate route to that node. It uses the first alternate route that it obtains from its cache to appropriately modify the source route and send the packet to the newly discovered next hop. Hence, the part of the route starting from the current node to the node to which it found an alternate route is changed. If there is no alternate route available to any of the intermediate nodes, but there is a route to the destination, then it results in an entirely new alternate route from the current node. In the worst case, if there is no alternate route to any of the nodes including the destination, the intermediate node drops the packet.

2. The intermediate node first checks if it has an alternate route to the destination. If not, similar to the previous approach, it searches hop-2 neighbor, hop-3 neighbor, and so on.

For both the versions, the routing protocol takes care *not* to include the next unreachable hop in the alternate routes used for rerouting. This is achieved by first removing all the routes from cache that contain the link from the current node to the unreachable next hop. We first conducted simulations to compare these approaches and to validate the hypothesis that reusing the existing routes is desirable; the first version follows this hypothesis, whereas the second approach

attempts to find an alternate route only when DSR fails to find a route to a destination. Since we find that the first approach indeed outperforms the second, we only present the first approach here.

3.1 Load Balancing

As we discussed in the Introduction, using alternate routes to reroute all packets of a flow may interfere with normal data transmissions that are sending packets through that alternate route. This problem is aggravated if multiple flows facing route failures use the same alternate route to transmit their packets. To reduce the interference with other flows, our routing protocol does *load balancing* among the number of alternate routes that are available. The protocol uses multiple alternate routes (if available) in round robin order for rerouting packets that face a route failure. Our simulations have shown that in general nodes have two or more alternate routes to 90% of the nodes in the network. Hence, with this assumption, while looking for alternate routes to the hop-2 neighbor, we start transmitting to that neighbor if there is any route to it. This is also consistent with our hypothesis that reusing existing routes is desirable. We search for alternate routes to our hop-3 neighbor only if there is no route to the hop-2 neighbor. In addition, our protocol, like DSR, notifies the source of the data transfer about any route failure. Hence, we try to transmit packets using multiple alternate routes until the source finds a new route and stops transmitting packets using the failed route.

3.1.1 Modifications to the Cache

In our protocols, we modify the way cache is updated. The first modification deals with how the protocol learns of new routes, and the second modification deals with how the protocol uses its primary and secondary cache of routes. The total cache size is kept fixed to 64 entries.

In DSR, a node caches a route when it receives a new route in reply to a *route request* packet, or when it *overhears* a packet not addressed to it and snoops into the source route to discover the route contained in that packet. Also, an intermediate node forwarding a packet snoops into the source route in the packet header and extracts the segment of the source route starting from the current node to the destination. Our protocol, in addition, extracts the route segment starting from the current node to the source of the packet. Thus, we can cache alternate routes for both the destination and the source of the packet. We, however, keep the total size of the cache maintained at each node same as that in DSR.

Like DSR, our protocol also maintains two separate fixed-sized caches of routes: a *primary* cache and a *secondary* cache. The primary cache is used to store routes returned in reply to *route request* packets. The secondary cache stores routes that have been learnt by other ways, for example, by snooping into the header of a packet while it is being forwarded. In general routes are added more frequently to the secondary cache, as a node is able to learn more routes from others' packets than the number of explicit *route replies* it receives. This implies that a route is more quickly eliminated from the secondary cache than it would

if it were in the primary cache. Hence, when a source node uses a route from the secondary cache to transmit its *own* packets, DSR *promotes* the route from the secondary cache to the primary cache. Our protocol, however, needs the routes to remain in the same order throughout their existence in cache. Otherwise, doing load balancing with the possibility of changes in the ordering of routes may make its overhead prohibitive. Hence, we do not promote routes from the secondary to the primary cache. This implies that our protocol uses only a small amount of primary cache, but needs a larger secondary cache. To reflect this requirement, we have reduced the size of the primary cache, and increased the size of the secondary cache while keeping the total cache size constant. In addition, while replacing a route from cache, we try to preserve routes that are currently in use. Thus, in our protocol, the load of the rerouted packets is efficiently shared among multiple available alternate routes.

3.2 Route Reservation

Now we discuss our approach towards reserving bandwidth for original source routes and alternate routes. Our routing protocol tries to provide Quality of Service guarantees to source nodes initiating a data transfer. When a source wants to send data to a destination, it tries to reserve the requested amount of bandwidth along a source route before the data transmission starts. Towards this end, we assume that a host would be able to estimate the available bandwidth, using some link-level techniques. This may also need a close interaction between the link layer and the routing layer for the routing protocol to use this knowledge of bandwidth availability. The approach used to estimate the total available bandwidth is outside the scope of this paper.

In the QoS version of our protocol, when a source host initiates a *route discovery* phase to reach a particular destination, it is required to state the amount of bandwidth, it intends to consume, in its packet header. An intermediate node processing a route request packet checks to see if it has enough available bandwidth to be able to accept the request. If the node determines that it can support the requested flow, it re-broadcasts the route request packet. This continues until either the request reaches the destination, or some intermediate node that does not have enough available bandwidth. In the latter case, the node drops the packet.

As in DSR, once a route request reaches the destination, the node reverses the *route record* so far formed in the packet header, and retransmits the *route reply* packet back to the source. If any of the links in the route record is not bidirectional, the route reply would not reach the source. Hence, the source would only receive routes whose all links are bidirectional (required by IEEE 802.11 as discussed earlier in Section 2). Thus, although a node makes a reservation when a route request packet travels towards its destination, the route may not work out due to any of a variety of reasons such as link failure or due to the source deciding to use a different route. This implies that nodes need to maintain timeout values to determine if a reservation should be kept any more. In our protocol, a reservation

is *not* removed as soon as its timeout expires, but it is removed if its timeout has expired and some new data flow is requesting a reservation for which the node does not have enough free bandwidth.

We maintain three timeout values: *route reply timeout*, *data start timeout*, and *data timeout*. The *route reply timeout* is used to remove reservations for which the node has not seen a route reply packet. This may happen either if some node towards the destination could not provide the requested bandwidth, or because one of the links on that route is unidirectional. The *data start timeout* expires if a node has seen a route reply for a reservation, but has not seen data from the source. This can happen if any of the links from the current node towards the source is unidirectional, or if the source chooses to use a different (probably shorter) route for the data transfer. The *data timeout* value is used to keep track of flows which have transmitted data, but this timeout value has elapsed since the last data packet was forwarded. This can occur, for example, by node movement.

As the QoS version of our protocol aims to reserve bandwidth along a route before starting a data transfer, it requires the route request packet to reach the destination. It does not allow any intermediate node to reply to the source with a cached route it may have for the destination. (DSR and non-QoS version of our protocol continue to be configured to use cached routes during route discovery.)

In the QoS version of our protocol, we store routes returned in *route replies* in the primary cache. Thus, routes in the primary cache have bandwidth allocated for them. When a source node initiates a data transfer, our protocol allows it to look for routes *only* in the primary cache (and not in the secondary cache), since we want it to use a route for which bandwidth has been reserved. It selects the shortest available route from the primary cache to the destination. Routes in the secondary cache are used only for rerouting packets through alternate routes when a route fails. (Note that DSR and the non-QoS version continue to use the routes in the secondary cache.) Such an approach in our QoS protocol still allows the possibility of a source using a route whose reservation has been timed out from some intermediate node. In such a scenario, the intermediate node not having the required reservation would drop the data packets. Our simulation results show that this has only a minimal effect on packet delivery ratio.

We make implicit reservations for rerouted flows. Consider the case where k disjoint alternate routes are available when a packet needs to be rerouted due to node movement/failure. In this case, $\frac{1}{k}^{th}$ of the flow will be transmitted on each alternate route. With this intuition, we allow a node to reserve only $\frac{k}{k+1}^{th}$ bandwidth while making reservations. The remaining $\frac{1}{k+1}^{th}$ ($\frac{1}{k}^{th}$ of $\frac{k}{k+1}^{th}$) bandwidth is implicitly reserved for rerouted flows. We found that in over 90% cases, two (or more) disjoint alternate routes are available when a flow needs to be rerouted. Hence, while simulating the QoS version of our algorithm, we let $k = 2$. Thus, $\frac{1}{3}^{rd}$ bandwidth is implicitly reserved for rerouted flows. If the redundancy in an ad-hoc network is high, higher values of k can also be used; larger values of k will minimize the bandwidth that is reserved for rerouted flows.

4 Performance Comparison

For our simulation, we use the network simulator *ns* (Version 2.1b8a) that is a discrete event simulator developed as part of the VINT project [2]. Protocols are evaluated with ad-hoc network topologies consisting of 50 wireless nodes, moving about in a rectangular space. The simulation time is 900 seconds for each run. The protocol takes as input a scenario file and a data traffic generation file. The scenario file specifies the movement of each node, and the traffic generation file has the data transfer characteristics giving details such as when each source node starts a data transfer, the number of packets to be transmitted per second, and the size of each packet. The link bandwidth is 2 Mbit/sec for all the results.

We use the *random waypoint model* to model node movement in our simulations. Each run of the protocol is characterized by a *pause time*. At the start of the simulation, each node remains stationary for *pause time* seconds. Then each node selects a destination from the rectangular space randomly, and starts moving towards the target with a speed uniformly distributed between 0 and a maximum speed of 20 meters per second. At the destination, the node again stays there for *pause time* seconds before moving again. For small data rate (Figure 1), we use a rectangular space of dimensions 1500m $\times$ 300m. For large data rate (Figures 2, 3 and 4), we use a space of dimensions 1800m $\times$ 1000m.

We ran our simulations with networks containing CBR (constant bit rate) sources. First, we compute the effect of node movement on the percentage of packets dropped (= total number of packets dropped $\times$ 100 / total number of packets sent) (cf. Figure 1). The graph in Figure 1 (a) is for a network of 10 data sources, while that in Figure 1 (b) is for a network of 20 sources. Each source has a sending rate of 4 packets per second, with a packet size of 64 bytes. We conduct our experiments for the following values of pause time: 0, 30, 60, 120, 300, 600, and 900 seconds. A pause time of 0 means constant mobility, while that of 900 seconds implies no node movement. With high node mobility, the percentage of data packets dropped by the non-QoS version of our protocol is around half of that dropped by DSR. This performance improvement is achieved without any extra control packet overhead. Specifically, our protocols do not send any extra control packets to discover or maintain its alternate routes. As shown in Figure 1 (a), the QoS version is best when the number of sources is 10. This is due to the fact that routes used in the QoS version are more stable; during route reservation, the QoS version validates the route being used. However as number of sources is increased, due to the extra overhead of route reservation, our non-QoS version is better.

(a) **(b)**

Figure 1: Comparison among the three protocols of the percentage of data packets dropped as a function of pause time. (a) Number of sources = 10, (b) Number of sources = 20

Figure 2 compares our QoS protocol with DSR for the percentage of data packets dropped as a function of the data rate of a source. The results are for a

network of 20 sources and 30 flows. The graph in Figure 2 (a) is for a pause time of 600 seconds, while that in Figure 2 (b) is for a pause time of 60 seconds. In our QoS protocol, a source sends data packets only after it gets a reservation. This only requires a coordination between the application sending data and the routing protocol. This is because if an application needs QoS service, it has to get a resource reservation before it can start transmitting packets. For comparison, we modify DSR so that it does not send packets when the sender buffer is full. In other words, in simulating DSR, we ensure that packets are never dropped at the source. As discussed earlier, the results in Figure 2 show that the performance improvement of our QoS protocol over DSR increases with the data rate. Even at high data rates of each source node transmitting at 4096 bytes/second, our protocol manages to deliver 80% to 90% of data packets transmitted, whereas DSR is able to deliver only around 30% to 40% of the data packets. It is also interesting to note in the graphs in Figure 2 that the performance of our QoS protocol actually improves while moving from a packet rate of 2048 bytes/second to that of 4096 bytes/second. This is because at such a high data rate, few flows actually get reservation and hence they can transmit most of their packets even through alternate routes in the event of a route failure. For lower data rates, many flows get reservations initially, but some of them are not able to reroute packets efficiently when a route fails.

(a) **(b)**

Figure 2: Comparison between the two protocols of the percentage of data packets dropped as a function of the data rate of a source node. There are 20 sources and 30 data flows. (a) Pause time = 600 seconds, (b) Pause time = 60 seconds

While Figures 1 and 2 look at collective data loss, Figures 3 and 4 focus on the effect of data loss on individual flows. More specifically, Figures 3 and 4 plot the number of data flows that have the percentage of data packets dropped in a specific range. For example, the number of data flows that have their percentage drop between 0 and 10% (inclusive of both) is plotted corresponding to 10 in the X-axis, the number of flows having percentage drop greater than 10% and less than or equal to 20% is plotted corresponding to 20, the number of flows having percentage drop greater than 20% and less than or equal to 30% is plotted corresponding to 30, and so on. Figure 3 shows the simulation results for 600 seconds pause time, while Figure 4 shows the results for 60 seconds pause time. For both figures, the data rate at which each source node transmits is 256 bytes/second, 512 bytes/second, 1024 bytes/second, 2048 bytes/second, and 4096 bytes/second respectively.

These results also show that the performance improvement of our QoS-protocol is more for higher traffic rates, with the highest rates showing explicitly how effective our protocol can be. In Figures 3 (a) and 3 (b), around 25 data flows have less than 10% packet drop ratio for both our protocol and DSR. In Figure

3 (c), 16 flows have less than 20% drop ratio for our protocol, whereas there are only 11 flows in this range for the DSR protocol. For higher data rate of 2048 bytes/second, Figure 3 (d) shows that our protocol has 9 flows having percentage drop less than 10, while DSR only has 2. Similarly in Fig 3 (e), there are 16 flows with percentage drop 10% or less for our protocol, whereas DSR does not have any flow with such a low drop percentage. Also for higher data rates and lower pause time (cf. Figures 4 (d) and 4 (e)), our version shows marked improvement over DSR. For example, for a data rate of 4096 bytes/second in Figure 4 (e), there are 27 flows for our protocol that have a percentage drop less than or equal to 40%, whereas DSR does not have any flow in this range.

As seen from the results, our protocol has large improvement over DSR for high data traffic rate. This is because for such scenarios, our protocol effectively uses its route reservations, and hence can guarantee high packet delivery ratio for most of the reserved data flows. On the other hand, DSR tries to transmit packets in every flow, effectively achieving less packet delivery ratio for all the flows. Moreover, as shown in Figure 2, by allowing some flows (with reservations) to transmit data into the network (and disallowing other flows), we achieve a higher total packet delivery ratio compared to DSR.

Our results in Figures 3 and 4 show that in most cases, our QoS protocol is able to deliver most flows with small loss rate. In case of small loss rates, techniques such as forward error correction [5, 6] can effectively provide 100% packet delivery ratio. In the simplest implementation of FEC, given two parameters m and n, $n > m$, for each group of m packets, $n-m$ additional parity packets are sent. If the receiver receives any m packets (some data packets and some parity packets), it can obtain all m data packets. Hence, even considering the case where we have the highest drop ratio, we can use $n = 2 \times m$ in an FEC implementation so that the effective delivery ratio is close to 100%. Moreover, for smaller data loss rates, the number of parity packets can be reduced even further.

(a) **(b)** **(c)**

(d) **(e)**

Figure 3: Comparison between the two protocols of the number of data flows that have their percentage data packets dropped in a certain range, for a pause time of 600 seconds, involving 20 sources and 30 data flows. (a) Data rate = 256 bytes/sec, (b) Data rate = 512 bytes/sec, (c) Data rate = 1024 bytes/sec, (d) Data rate = 2048 bytes/sec, (e) Data rate = 4096 bytes/sec

5 Related Work

Ad-Hoc routing protocols have been categorized [7] as Table-driven and Source-initiated on-demand. Table-driven routing protocols always try to maintain a consistent view of the network using one or more routing tables. These tables hold

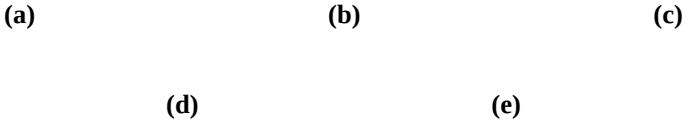

Figure 4: Comparison between the two protocols of the number of data flows that have their percentage data packets dropped in a certain range, for a pause time of 60 seconds, involving 20 sources and 30 data flows. (a) Data rate = 256 bytes/sec, (b) Data rate = 512 bytes/sec, (c) Data rate = 1024 bytes/sec, (d) Data rate = 2048 bytes/sec, (e) Data rate = 4096 bytes/sec

routing information for communication between any pair of nodes in the network. In the event of network topology changes, routing table updates are exchanged between nodes in the network. Some of the prominent protocols in this category are Destination-Sequenced Distance-vector (DSDV) Routing [8], Wireless Routing Protocol [9], and the QoS routing protocol presented in [10]. These protocols differ in the way they update the routing tables. A problem often attributed to these protocols is that even in the absence of data traffic between hosts, nodes would continue to exchange broadcast messages to keep their routing tables updated.

Source-initiated on-demand routing protocols discover a route only when needed. The source node does a *route discovery*. Once a route has been obtained, it is inserted into packets flowing from the node dictating the route the packet should take. Some of the prominent protocols in this category are the Dynamic Source Routing Protocol [3], and the Neighborhood Aware Source Routing [11].

The Neighborhood Aware Source Routing (NSR) [11] protocol maintains information about the two-hop neighborhood of a node. A node, while forwarding a data packet, tries to take advantage of the information it has about the status of the two links along the source route in its 2-hop neighborhood. A node periodically broadcasts *hello* packets to its neighbors to notify its presence. These packets are also used to collect link-state information. We, however, in our protocol try to maintain information about the 2-hop neighborhood by using the information already contained in the control packets, and hence do not generate any extra control overhead. Moreover, our protocol and NSR differ in the way the 2-hop neighborhood information is used. NSR uses the information to detect a failed link, and reroute the packets after finding an alternate route. Our protocol, on the other hand, uses the information to efficiently find alternate routes for rerouting packets (possibly to the 2-hop neighbor) when the next hop node moves and/or fails.

CEDAR [12] is another QoS routing algorithm for ad-hoc networks. It uses a self-organizing *core* of nodes. A core node performs route computations on behalf of other nodes present in its domain. When a node wants to communicate to a destination, it contacts its dominator node in the core to find a route to the destination which meets the QoS requirement. Nodes in the core use *core broadcast* to exchange network topology information with other nodes in the core. By contrast, our algorithm does not send any extra control packets.

6 Conclusion

In this paper, we focused on the problem of route maintenance and bandwidth allocation in ad-hoc networks. We presented two protocols, a QoS version and a non-QoS version. The main features of these protocols are as follows: (1) When the route specified by the source breaks due to a node movement/failure, that source route is reused –as much as possible– while rerouting the packets on an alternate route. (2) When the route specified by the source breaks, intermediate nodes use multiple alternate routes so that the rerouted flow does not interfere with other flows in the network. We find that in most cases, these alternate routes are disjoint and, hence, if k alternate routes are available then only $\frac{1}{k}^{th}$ bandwidth is used on each alternate route. (3) In the QoS version of our protocol, a source explicitly reserves the requested bandwidth before transmitting. However, for the case where the route chosen by the source breaks, no explicit reservations are made on the alternate routes used. However, implicit reservations are maintained on all links. More specifically, if k alternate routes are available for load balancing, $\frac{1}{k+1}^{th}$ bandwidth is used for rerouted flows and a node permits reservations for $\frac{k}{k+1}^{th}$ of the maximum limit. The simulation results in Section 4 show that by letting $k=2$, such implicit reservation typically provides the required bandwidth.

The simulation results in Section 4 show that our protocols are better than DSR for both low and high data rates, as well as for networks with low or high node mobility. With low data rates, our protocol delivers close to 100% of transmitted data packets. With high data rates, our protocol efficiently uses the available bandwidth and tries to maximize the number of flows having very low data loss rates. In addition, nodes in our protocol maintain a running average of data being transmitted for each source node. This implies that our protocol can verify if a source node is really transmitting at the requested rate. Also, the fact that our protocol keeps track of the amount of data a source is sending enables it to charge the respective source nodes for the QoS service it provides.

In the presence of node movement, it is impossible to efficiently deliver 100% of messages. However, if the percentage of packets dropped is small then proactive techniques such as forward error correction [5, 6] can effectively provide 100% delivery ratio. Our results in Figures 3 and 4 show that for most flows the percentage of messages lost is small. Moreover, the data loss rate for our protocol is significantly less than that for DSR. Hence, we expect that combining our protocol with FEC-based techniques will be especially attractive.

The approach for load balancing and route reservation is also applicable in other domains, e.g., in sensor networks. We have used our approach in an application that is a variation of the beam experiment used in [13]. In this problem, the network consisted of a simply supported elastic 2-D grid. Each node in the grid consisted of a sensor-actuator pair, where the sensor provided velocity measurements and the actuator applied force. This grid was subject to external vibrations and the sensors and actuators were used to minimize the vibration at all nodes.

Due to the correlation between sensor values at multiple nodes, it was necessary to communicate sensor values of a node to other nodes in the network. By using our approach for load balancing and implicit bandwidth reservations, we found that the quality of vibration control is maintained when nodes fail. More specifically, we found [14] that if our approach is used to deal with node failure then the quality of vibration control is almost identical to the case where no nodes fail.

There are several possible extensions to this work; we assumed that the number of available alternate routes is fixed (at 2). One possible extension is to change this dynamically based on the actual number of alternate routes available. When the number of alternate routes available is high, the amount of implicit reservations for rerouted flows is small. We are focusing on what information a node should share about its alternate routes with other nodes in the network, and how often it should share it. Another extension of this work is to identify the effectiveness of our protocol when combined with forward error correction. As mentioned in Section 4, the loss rate suffered by our protocol is small. Hence, we expect that by sending only a small number of parity packets, we will be able to recover all the packets at the receiver.

References

[1] J. Broch, D. A. Maltz, D. B. Johnson, Y. C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'98), Dallas, Texas, USA*, October 1998.

[2] Kevin Fall and Kannan Varadhan. editors, The ns manual. November 1997. The VINT Project, UC Berkeley, LBL, USC/ISI, Xerox PARC, Available at: http://www.isi.edu/nsnam/ns/.

[3] David B. Johnson and David A. Maltz. Dynamic source routing in ad hoc wireless networks. *Mobile Computing*, 5:153–181, 1996.

[4] The CMU monarch project: Wireless and mobility extensions to ns-2. http://monarch.cs.cmu.edu/cmu-ns.html.

[5] R. E. Blahut. *Theory and Practice of Error Control Codes*. Addison-Wesley, Reading, MA, 1983.

[6] A. J. McAuley. Reliable broadband communication using a burst erasure correcting code. *ACM SIGCOMM*, 1990.

[7] E. M. Royer and C-K Toh. A review of current routing protocols for ad-hoc mobile wireless networks. *IEEE Personal Communications Magazine*, pages 46–55, April 1999.

[8] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. *Computer Communications Review*, pages 234–244, October 1994.

[9] S. Murthy and J. J. Garcia-Luna-Aceves. An efficient routing procotol for wireless networks. *ACM Mobile Networks and Applications Journal, Special Issue on Routing in Mobile Communication Networks*, pages 183–197, October 1996.

[10] Chunhung Richard Lin and Jain-Shing Liu. QoS routing in ad hoc wireless networks. *IEEE Journal on Selected areas in Communications*, 17(8), August 1999.

[11] Marcelo Spohn and J.J. Garcia-Luna-Aceves. Neighborhood aware source routing. *Proceedings of ACM MobiHoc, Long Beach, California*, October 2001.

[12] Raghupathy Sivakumar, Prasun Sinha, and Vaduvur Bharghavan. CEDAR: a core-extraction distributed ad hoc routing algorithm. *IEEE Journal on Selected areas in Communication*, 17(8), August 1999.

[13] A. Ledeczi, M. Maroti, and I. Bartok. Simple NEST application simulator. Technical report, Institute for Software Integrated Systems, 2001. Also available at http://www.isis.vanderbilt.edu/projects/nest/index.html.

[14] A. Arora, M. Gouda, T. Herman, S. Kulkarni, and M. Nesternko. Self-stabilization in networked embedded software technology (NEST). Available at: http://www.darpa.mil/ipto/research/proceedings/nest2002feb/OhioStateU200202.pdf, February 2002.