Train of Packets
A Novel Routing Algorithm for Mobile Ad Hoc Networks

ilker Basaran
Department of Computer Science, UC Riverside
March 2005

Abstract

We propose Train of Packets routing for mobile ad hoc networks as a technique for reducing the packet header size by caching session specific routing information at intermediate nodes. Thus, if multiple packets belonging to the same session follow the same path, the data packets do not carry a full header. By this way the transmission time and number of transmitted packets decrease dramatically. Given the limitations of mobile devices, such as battery power, we can lengthen the life of a particular node by minimizing the number of packets to transmit. Routing in mobile ad hoc networks as a technique for reducing packet header size is achieved by caching some session specific routing information at intermediate nodes so that subsequent packets using the same path will not need it. In our approach, we have a hybrid routing protocol that combines the advantages of existing proactive and reactive routing protocols. On top of this, Train of Packets approach handles multiple paths between the source and destination to improve perceived throughput, avoid congestion and speed recovery from broken links. Moreover, according to bandwidth availability and delay properties of a particular path, Train of Packets approach uses load balancing which further enhances throughput and distributes the traffic burden to different paths enabling better utilization of the network. This paper describes Train of Packets approach in detail.

1. Introduction

Maintaining the service in the presence of the possible diverse application areas of ad hoc networks face with the problem of frequent changes due to external factors like interference and physical layer signal impairments, such as bit error rates and bandwidth availability. These factors lead to unpredictable changes to link quality. Moreover, nodes are mobile and may join the network at anytime, or get disconnected due to movement to a new place or running out of battery power. Since the traffic patterns and topology of the network may change drastically, one important challenge in mobile ad hoc networks is designing routing protocols that can rapidly adapt their behavior to these changes.

Current ad hoc routing protocols broadly fall into two categories. Proactive protocols, such as DSDV [DSDV], exchange routing information periodically to have a set of paths for all nodes in the network, conversely reactive protocols, such as AODV [AODV], DSR [DSR], and TORA [TORA], try to find a path only when it is required. There are also a few hybrid protocols, such as ZRP [ZRP], and SHARP [SHARP], that merge advantageous parts of proactive and reactive routing strategies. Our approach is similar to hybrid schemes considering route discovery phase, but differs in other directions.

The novelty in our work compared with previous hybrid approaches is the reduction of the packet header size. It is a well-known fact that most of the mobile nodes are limited to battery power and only the efficient use of it may extend the life of a particular node and correspondingly the network. In order to achieve this we introduce the idea of reducing the packet header size.

By doing this, the transfer of data will be completed in fewer packets and less amount of time. The main idea is similar to virtual circuit switching and/or wormhole routing in wired networks. After the discovery of the path to a destination, the source will put all routing information to the first packet of the whole data (as in reactive protocols), together with a streamID. The intermediate nodes in the path receiving this packet will cache the route by its streamID and then forward the packet. Hence, the subsequent packets will only carry a streamID which is used by the intermediate nodes to forward them. As seen the first packet acts a locomotive and the following ones as railway cars, thus the name of our approach comes into existence, *Train of Packets*.

Second difference with previous hybrid strategies is usage of multiple paths. When the route discovery process returns multiple non-overlapping paths or path segments to a destination, our approach makes use of all of them to improve perceived throughput, avoid congestion and faster recovery from broken links. This is achieved by load balancing over available paths to the destination according to their bandwidth availability and delay properties. Hence, from the point of transfer time, the duration will be much shorter and from the point of connectivity, the source and destination will still have a path even if a link fails (if multiple paths exist for that source-destination pair). It is obvious that, this load balancing among multiple paths will further enhance the throughput and distribute the traffic burden to different paths enabling better utilization of the network.

2. Related Work

The Destination-Sequenced Distance-Vector protocol (DSDV) described in [DSDV] is a table-driven algorithm based on the classical Bellman-Ford routing mechanism. The improvements made to the Bellman-Ford algorithm include freedom from loops in routing tables. Every mobile node in the network maintains a routing table in which all of the possible destinations within the network and the number of hops to each destination are recorded. Routing table updates are periodically transmitted throughout the network in order to maintain table consistency. New route broadcasts contain the address of the destination, the number of hops to reach the destination, the sequence number of the information received regarding the destination, as well as a new sequence number unique to the broadcast [DSDV]. The route labeled with the most recent sequence number is always used. The update broadcasts are delayed by the length of the settling time, so the mobile nodes can reduce network traffic and optimize routes by eliminating those broadcasts that would occur if a better route were discovered in the very near future. *Train of Packets* routing scheme differs from DSDV by its neighborhood size. In DSDV neighborhood is the whole network, whereas in our scheme it is only the *k*-neighbors (i.e. the nodes that you can reach in *k* hops). So the overhead of periodic updates will be diminished. The neighborhood size *k* is dynamic and may be adjusted according to traffic load.

The Dynamic Source Routing (DSR) protocol presented in [DSR] is an on-demand routing protocol that is based on the concept of source routing. Mobile nodes are required to maintain route caches that contain the source routes of which the mobile is aware. Entries in the route cache are continually updated as new routes are learned. When a mobile node has a packet to send to some destination, it first consults its route cache to determine whether it already has a route to the destination. If it has an unexpired route to the destination, it will use this route to send the packet. On the other hand, if the node does not have such a route, it initiates route discovery by broadcasting a route request packet. A route reply is generated when the route request reaches either the destination itself or an intermediate node which contains in its route cache an unexpired route to the destination [DSR]. The reply is sent by a cached route to source or if such route does not exist by the reverse route in the route record assuming the links are bidirectional. If the links are unidirectional the node may initiate its own route discovery and piggyback the route reply on the new route request. Both DSR and other known on-demand routing protocol TORA [TORA] supports multiple paths. *Train of Packets*, however, prevents large packet headers that will be generated because of long routes. It is obvious that if the path is healthy there is no need to send all routing information with every packet. On the other hand, if the path breaks, our scheme will also recover from it as DSR. *Train of Packets* will facilitate on-demand routing for the destinations that are not the immediate neighbors of the source node.

The hybrid approaches, such as ZRP [ZRP] and SHARP [SHARP], inherit the advantageous parts of the proactive and reactive protocols. The Zone Routing Protocol (ZRP) was the first hybrid routing protocol with both a proactive and a reactive routing component. ZRP defines a zone around each node consisting of its *k*-neighborhood. Routing within a zone is performed using a proactive routing protocol and routing between nodes in different zones is performed by an on-demand routing protocol. The size of the zone is dynamically determined based on network load. While SHARP has the main concept of hybrid routing with ZRP, the two approaches differs in several fundamental ways. SHARP enables application-specific adaptation strategies to bound loss rate and control jitter, in addition to controlling the overhead of the routing protocols and also maintains proactive routing zones only around those nodes that have significant incoming data. But this introduces extra overhead on the nodes to monitor the traffic continuously. For the sake of simplicity and easier decision process in the nodes, we design a scheme closer to the idea of ZRP.

Our load balancing with multiple paths technique is very much different from that is done in Open Shortest Path First (OSPF) [OSPF] which is used in intra-network routing. In OSPF the multiple links are assigned the same cost and load sharing is done among them. But *Train of Packets* assigns a probability to each path (if multiple paths exist between the source and the destination) to distribute the load among them. The probability of each path is updated when an ACK is received corresponding to that path. The calculation of new probability depends on bandwidth availability and delay properties of that path, which are extracted from a series of ACKs by using the bandwidth estimation technique in [TCPW].

Finally, the streamID field of the packets may be remembered from Multi-protocol Packet Label Switching (MPLS) [MPLS]. But in MPLS, labeling is mainly used for tunneling and supporting differentiated services. Inside MPLS header there is a label for each node the packet will pass. So MPLS has a similar problem with DSR, that is, the packet header size will increase as the number of hops in the route increases. Conversely, in our scheme, the streamID field is fixed in length and allows the intermediate nodes to forward packets easily. That is the main purpose of *Train of Packets* by which the routing information is excluded from the packet and packet header size is reduced dramatically.

3. Train of Packets

3.1. Assumptions and Basics

There are a few assumptions that we should note. First of all, we assume that there is an underlying MAC layer, namely 802.11, that deals with and takes care of collisions and transmission schedules. Second, the links in between the nodes are assumed to be bidirectional, so if a node can hear another node, it is also true vice versa. The network size and nodes' mobility model are subject to simulations and will be discussed later. There is no need for special equipments, such as GPS receiver, on the nodes. Finally, all the nodes are identical, in the sense that none of them has better calculation power or more battery life than the others.

The streamID (8 bits total), which is going to be used in our algorithm, will consist of two parts. The upper 5 bits of streamID will be sessionID that is going to define the session between source and destination. It will be unique to source node and the reasons for that are explained in 3.2. The lower 3 bits will be pathID, which will define each path between source and destination uniquely. The hierarchical structure of streamID is needed in order to handle multiple paths in their most general cases. Hence the final structure of streamID is as follows:

sessionID	pathID

3.2. Route Discovery and Routing

Train of Packets algorithm keeps track of its *k*-neighborhood proactively, so in a table it has all the routing information to reach its *k*-hop neighbors. For further nodes it broadcasts a route request message. From this point on the path (or possibly the paths) is discovered as it is done in DSR. For the sake of possibility of multiple paths to destination, the source does not start data transfer immediately as it learns a path from the first route reply. It waits for a specific, though adjustable, amount of time for other possible route replies. The other important details of handling multiple paths are discussed in 3.3.

Once the path (or paths) is discovered to destination, the data is prepared to be sent. The first packet, along with the first data chunk, carries the full path to the destination. The first packet also has a sessionID field within streamID that is a unique number to the source only. The main reasons why it needs to be unique to the source only are as follows. First of all, there will be three kinds of nodes that will need to differentiate flows. These are source, intermediate and destination nodes. The destination node can easily identify flows by checking sessionID together with the source ID. So sessionID need not be unique to the destination node. Similarly the intermediate nodes can cache the sessionID together with source and destination IDs, so when they need to differentiate flows; they check the table that holds source-destination-streamID (note that sessionID is hidden inside streamID) trios. Considering these facts we conclude that the sessionID field need not be unique to both destination and intermediate nodes, since they can differentiate flows by checking source-streamID pair (the destination) or source-destination-streamID trio (the intermediate nodes). Secondly, we also evaluated the idea of using a signaling technique similar to ATM Virtual Circuits. In this scenario a connection is first established by allocating a unique forwarding number on each hop from source to destination. Once a Virtual Circuit is established between source and destination, the source can send packets into the Virtual Circuit with the appropriate Virtual Circuit Number. Because a VC has a different VC Number on each hop and also each hop has a VC-Number translation table, the packets are easily forwarded by the use of these tables. When an intermediate node receives a packet, it first checks its VC-Number translation table and decides to which node it should forward the packet. The table also has the information of the unique number that the next hop is using for this VC. This information is exchanged in the signaling phase of establishing the connection. The intermediate node, then, replaces the VC Number in the incoming packet with the number corresponding to that VC for outgoing packets. By this way the next hop will be able to identify the VC and forward the packet accordingly. After briefly describing how ATM VCs work, one immediately realizes that the overhead is too much. As discussed, we do not need to introduce the overheads such as signaling phase and replacing the numbers on each hop. We can achieve this simply by using a unique number to the source only.

Having decided to use a unique sessionID to the source only, the intermediate nodes, then, cache the *full* path together with the corresponding source-destination-streamID trio by extracting the information from the first packet. The cached paths will form the *learned paths*. The subsequent packets following the first packet do not carry any kind of routing information other than source and destination addresses and streamID. The intermediate nodes will know how to route the packets by looking it up in their caches. We should emphasize the reason why the intermediate nodes cache the full path. It is for the acknowledgements that will return to the source by using the same path. Also the cache of the intermediate nodes (learned paths) will be soft state cache that will allow them to delete idle or broken paths without explicit indication. The decision heuristic of expunging paths in intermediate node caches is discussed in 3.4.

In the source node, for each path, the following information will be kept: full path, RTT, how many times this path has been timed out, bandwidth availability and probability of usage of this path. The first two arguments are trivial. The third parameter (number of time-outs) is going to be used by route maintenance procedure. Bandwidth availability and probability measures are going to be used for load balancing purposes. The path list to a particular destination will not grow (i.e. a new path will not be added) while data transmission is ongoing. Only after all paths to a particular destination are lost or bandwidth estimation of all the paths in path list are lower than a given threshold, then route discovery mechanism will be re-initiated for the destination. This is due to the fact that multiple paths will allow the data transmission to continue even a path breaks. Thus the routing is more resilient to path losses.

3.3. Multiple Routes and Route Repair

As stated in 3.2, a source node, after initiating a route request for a particular destination, will receive route replies. The amount of time to wait for the replies is adjustable. It should not be too short that will prevent discovering good paths and also it should not be too long that will insert unnecessarily long and delayed paths to the path list. There are two important points to emphasize here. The first is, even if the received multiple paths share an intermediate node in the middle of the path, those paths will be counted as multiple paths. We might think that this will introduce bottlenecks in the network. But carefully considering the behavior of MAC layer (802.11) we know that the usage of Request To Send and Clear To Send messages, shortly RTS and CTS, blocks the transmission of all nearby nodes where a communication takes place. So even if we try to have node-disjoint paths to prevent bottlenecks, because of the inherent characteristic of wireless communications, there is still a high probability that the packets of the same flow will compete for the same bandwidth. Also it has been stated in [YKT] that as the distance between a source and its destination is increased, one could find no more than a very limited number of paths between them. So before recklessly discarding these paths we should try to exploit their use. In [YKT] the authors try achieve a multi-path routing framework for providing enhanced robustness to node failures. In order to support reliability they avoid single point of failure by considering only node-disjoint paths in particular segments of a path. They introduce the idea of having *reliable nodes* in the network and routing the packets over them by using *reliable paths*. Concatenating a sequence of reliable segments creates the *reliable paths*. A segment is defined to be reliable if the number of node-disjoint paths that can be found between the end nodes of the segment is at least equal in number to k, which is a preset threshold, or, if the segment entirely consists of reliable nodes. We will further consider their work in section 4. Since our objective is to balance loads, increase throughput and reduce transmission time of data, we do not limit the multiple paths to be completely node-disjoint. This is also true for intermediate nodes. The intermediate nodes, when forwarding a route reply, should not ignore the subsequent route replies after relaying the first one for the same source destination pair. By this way we will be able to have paths as in Figure 1.

The second subtle point is that, even we are using multiple paths; the source destination pair will use the same sessionID on different paths. The destination will check the source ID together with the sessionID to determine where the packet belongs. So the flows on different paths but with the same source and destination will have the same sessionID. This case will not introduce extra overhead for handling out-of-order packet delivery, packet losses and duplicate deliveries. The sliding window algorithm that is used in TCP will take care of these problems since there is going to be a corresponding sliding window for each flow.

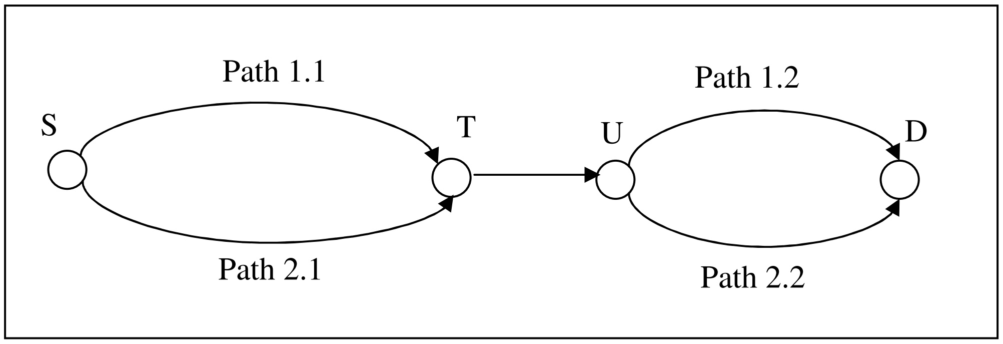


Figure 1 – The network after route discovery phase

Now suppose we have a situation as shown in Figure 1. We will consider two cases in which some of the path segments break but the overall connectivity still exists. So the route maintenance routine will not be facilitated. The first case is the breakage of path 2.1. In that case T will stop receiving packets from that path. But the data transmission will continue through path 1.1 to T and eventually to U. At that point, since the data has been received through path 1.1, U will always forward it through path 1.2. Hence the available path (path 2.1) will not be utilized. The solution for this case is as follows. The intermediate nodes will store a probability for their outgoing links as the source nodes do. The probability calculation and its details will be discussed shortly. We cache the learned paths in a table in the intermediate nodes and this table will now have an additional field, i.e. the probability of a particular outgoing link. As an example, node T will have 1 as the probability for its outgoing link since it has only one. But node U will calculate the probabilities for path 1.2 and 2.2 and store them in the cache next to corresponding path. By this way, we will distribute the load among available paths and achieve a better utilization of them.

The second case is the breakage of both path 2.1 and 1.2. This case is apparently the closest to the worst case in which we totally lose the connectivity. The data packets will continue to flow through path 1.1 up to U. At that point, the node U will try to forward the packet through path 1.2 since it received the packet from path 1.1. But unfortunately path 1.2 is broken. The action to take in such a situation is checking the learned paths cache. If U can find a path with the same source-destination pair that has the same sessionID, then it will continue to forward the packets through that path. In our example that path will be path 2.2. Hence the data transmission will not be interrupted. If U does not have such a path in its cache, then it will execute route maintenance routine.

Finally we should consider the ACKs for both cases. T will send the ACKs for both path 1.2 (for case1) and path 2.2 through 1.1. As the ACKs are received, the sliding window in the source will act accordingly, as we describe in 3.4 in more detail.

3.4. Load Balancing

In order to achieve load balancing, a probability for each path to a particular destination will be calculated. The initial probabilities of the paths will be

$$\mathrm{p_i} = \frac{b_i}{\sum_{\mathrm{i}}^{\mathrm{N}} b_i} \qquad [1]$$

where $\mathrm{b_i}$ is the bandwidth of an individual path. Hence the initial probabilities will be the same for all paths as we initialize each $\mathrm{b_i}$ with the same value. The probabilities, then, are going to be updated as the ACKs arrive, using bandwidth estimation values for the corresponding path. The bandwidth estimation heuristic used in [TCPW] is appropriate for our case. In this heuristic, the source monitors the ACK reception rate and estimates the data packet rate currently achieved by the path. The end-to-end behavior of the algorithm does not require any support from lower layers, and thus adheres to layer separation and modularity separations [TCPW]. The ACK-based bandwidth measurement procedure in [TCPW] is as follows. The authors first argue the importance of end-to-end principle in TCP, which guarantees the delivery of data over any kind of heterogeneous network. So they propose that a source perform an end-to-end estimate of the bandwidth available along a TCP connection by measuring and averaging the rate of returning ACKs.

The TCPW sender monitors ACKs to estimate the bandwidth currently used by, and thus available to the connection. More precisely, the sender uses (1) the ACK reception rate and (2) the information an ACK carries regarding the amount of data delivered to the destination. For now, let assume that an ACK is received at the source at time *tk*, notifying that *dk* bytes have been received at the TCP receiver. Then, one can measure the following *sample* bandwidth used by that connection as $b_k = d_k/\Delta_k$, where $\Delta_k = t_k - t_{k-1}$ and t_{k-1} is the time the previous ACK was received.

Since congestion occurs whenever the low-frequency input traffic rate exceeds the link capacity the authors employ a low-pass filter to average sampled measurements and to obtain the low-frequency components of the available bandwidth. This averaging is also useful in filtering out the noise due to delayed acknowledgments. The authors propose the following discrete-time filter, which is obtained by discretizing a continuous low-pass filter using the Tustin approximation [KB]

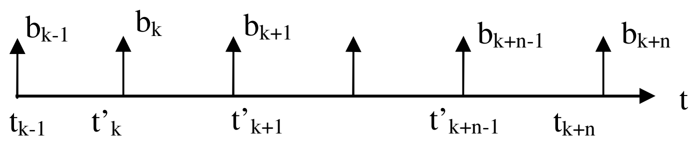
$$b'_k = \alpha_k b'_{k-1} + (1 - \alpha_k)\left(\frac{b_k + b_{k-1}}{2}\right) \qquad [2]$$

where b'$_k$ is the filtered estimate of the available bandwidth at time $t = t_k$, $\alpha_k = (2\tau - \Delta_k)/(2\tau + \Delta_k)$, and $1/\tau$ is the cutoff frequency of the filter.

We must note that the coefficients α_k are made to depend on Δ_k to counteract the effect of non deterministic interarrival times. In fact, when the interarrival time Δ_k *increases*, the last value b'_{k-1} should have less significance. On the other hand, a recent b'_{k-1} should be given higher significance. The coefficient α_k decreases when the interarrival time increases, and thus the previous value b_{k-1} has less significance with respect to the last two recent samples which are weighted by $(1 - \alpha_k)/2$. As an example, let $t_k - t_{k-1} = \Delta_k = \tau/10$. Then

$$b'_k = \frac{19}{21} b'_{k-1} + \frac{2}{21} \left(\frac{b_k + b_{k-1}}{2}\right)$$

The new value b'_k is thus made up of approximately 90% of the previous value b'_{k-1} plus approximately 10% of the arithmetic average of the last two samples b_k and b_{k-1}.



Since the TCPW filter has a cutoff frequency equal to $1/\tau$, all frequency components above $1/\tau$ are filtered out. But in order to sample a signal with bandwidth $1/\tau$ a sampling interval less than or equal to $\tau/2$ is necessary. Since the ACK stream is asynchronous, the sampling frequency constraint cannot be guaranteed. Thus, to guarantee this constraint, the authors of TCPW establish that if a time τ/m $(m \geq 2)$ has elapsed since the last received ACK without receiving any new ACK, then the filter assumes the reception of a *virtual* null sample $b_k = 0$. The situation is shown in Figure 2, where t_{k-1} is the time an ACK is received, t'_{k+j} are the arrival times of the virtual samples, with $t'_{k+j+1} - t'k+j = \tau/m$ for $j = 0, n - 1$; and $b_{k+j} = 0$ for $j = 0, n - 1$ are the virtual samples. Then, $b_{k+n} = d_{k+n}/\Delta_{k+n}$ is the bandwidth sample at t_{k+n}.

Thus we will have the bandwidth estimation values to update the probability of a particular path. The final bandwidth estimation calculation is [2]. When we substitute the bandwidth estimation values in [1], we will have a dynamic probability for each path as time progresses. The usage of RTT values, packet loss rates in probability calculation and how to use this scheme in intermediate nodes is further discussed in section 4.

The usage of multiple available paths is as follows: if there is data to send and there exists available space in the sliding window, *Train of Packets* will choose a path from the path list according to its probability. Then the packet will be sent along that path to the destination. At this point we introduce a slight addition to sliding window that will also keep track of which packet is sent through which path. This is going to allow us to update all required information of that path (i.e. RTT, how many times it has been timed out, bandwidth availability and thus probability) when we receive an ACK for that packet. The sliding window will also act accordingly when an ACK is received.

3.5. Route Maintenance

The idea of not having any kind of routing information in the intermediate packets raises the question what is going to happen when a link breaks in the path. We should consider this carefully. First of all, the source routing schemes should not try to *exhaustively* recover a broken link locally. This is a well-known fact since the source will always discover a better (or equally good in the worst case) path to destination than the intermediate node will. This can be further explained by an example.

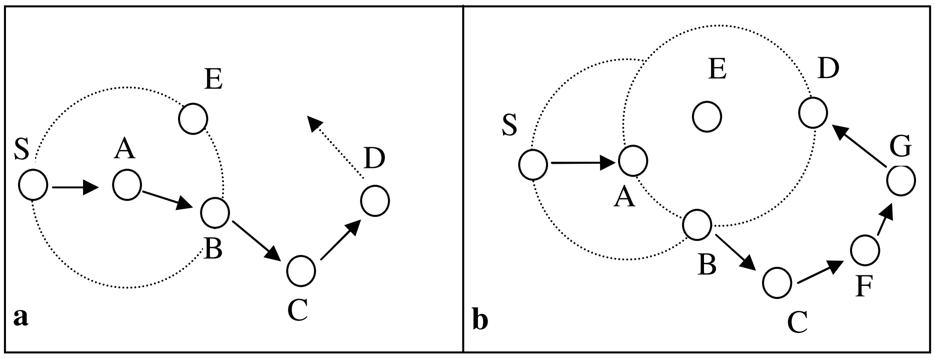


Figure 3 - Intermediate node recovering from a link breakage

In Figure3-a, the source-initiated path is using the nodes A, B and C to reach from S to D. Now suppose node D started to move through the dashed arrow, so that the link between C and D breaks and D enters the power range of E. If the intermediate node C tries to *exhaustively* recover the broken link, it will broadcast a Route Request and end up with a path as shown in Figure 3-b. As we can see, since C tried to recover the broken link, the total length of the path increased, whereas if the source had initiated a Route Request, the algorithm would find the path S-A-E-D which is a much better choice. If this worst case scenario repeats a couple of times, then we will have an unnecessarily long path that will dramatically increase the transmission time as well as introducing a higher probability of congestion, packet loss and link breakage. Also, we should note a minor, yet subtle, point that as long as the path is usable the hops are almost always the same. Hence, there is no need to carry the extra routing information in each packet especially considering that it is not used to recover from broken links locally.

After pointing out the risks of local recovery from a link breakage, we should propose our idea how to handle this problem. The idea is avoiding exhaustive recovery by an intermediate node while authorizing that node to communicate with its *k*-hop neighbors. The intermediate node, then, will send a Route Request to its *k*-hop neighbors together with the flow information (source-destination-streamID trio) and wait for a certain amount of time. If, a neighbor receiving this request has a current flow with the same source-destination-streamID trio, it will inform the intermediate node that initiated the Route Request. Then, the intermediate node that is trying to locally recover the link breakage will understand that the node replied to its Route Request is on a path that is being used by the source. So it will forward the flow through the node that had replied aggregating the traffic on it. We have discussed the reasons why we allowed overlapping multiple paths in section 3.3, so this kind of local recovery does not contradict with the overall routing scheme.

If the intermediate node receives multiple replies from its *k*-hop neighbors it should decide which one to use. In order to make this decision in an effective way we introduce the idea of hop counts. The details are as follows. We introduce two additional fields in the caches of intermediate nodes. These are the hop counts from the destination and to the source. When a neighbor replies to an intermediate node that is trying to recover a broken link, it appends these two numbers to its reply. So, now, the intermediate node will have a good criterion to decide which neighbor to forward the flow through. It will select the neighbor, which is closer to the source. If the intermediate node times out without a reply then it informs the source about this link breakage by sending a route error packet.

Train of Packets achieves the discussed issues as follows. In the case of a link breakage it will continue to transfer data as long as there is a path to the destination in the path list. The broken link will be tried to be locally recovered by the intermediate node on the path that successfully receives but unable to deliver the packets to next node on the path. The intermediate node will decide that the link to the next hop is broken by consulting *k*-neighborhood table which is updated periodically, and then try to locally recover from this breakage as discussed in the previous paragraph. If the intermediate node times out without a reply, it sends the “link broken” message to the source. This will reduce the time needed to purge the unnecessary paths in the intermediate node caches. There may be the case that a node is still in the path and there is no link breakage but that particular node drops the packets maliciously. We can handle such a situation either in the intermediate nodes or in the source node. In order to keep simpler states in intermediate nodes, we decided to solve this case as an end-to-end behavior. If a particular path times out on an ACK for a data packet sent, the time-out field in the path list will be increased by one. When the time-out field of a path becomes 3, then it will be removed from the path list and is not going to be used any more. Since the caches of the intermediate nodes are soft state, the deleted path in the source will eventually be removed from their cache. There is no need to initiate a path retrieval message by the source, thus preventing extra traffic burden on the network.

We are not sure if the packets, which are already sent through the broken paths, are received by the destination. At this moment, the additional field in the sliding window that keeps track of which packet is sent through which path helps us to re-send those packets through another path currently in the path list. But if we can exploit the use of ACKs on the sender side, we will not need to re-send the already sent packets. Since we are using multiple paths, as non-duplicate ACKs from other paths arrive, it indicates that all previous packets, irrelevant of from which path they were sent, have been received. Hence, if the packets sent from a particular path have been received, but not their ACKs due to a link breakage, will not be sent again. Because the non-duplicate ACKs will implicitly notify the source about the reception of all previously sent packets. If the source receives duplicate ACKs for a particular packet, it means there is a problem with the packet corresponding to that ACK number and the path that has been used to send that packet. *Train of Packets* then chooses another path to re-send that packet. If the broken link cannot be recovered locally by the immediate intermediate node, then the source will be informed, so that path will be removed from the path list. If the packets that have been sent through that path have not ACKed yet, the source waits for subsequent ACKs. If it receives duplicate ACKS, it re-sends those packets via another path. If it receives non-duplicate ACKs the source will understand that the packets have arrived safely.

A re-broadcast for route request on the source node will take place only if the path list becomes totally empty or the bandwidth estimation is lower than a given threshold. Interestingly, this scheme will prevent premature broadcasts as the data transmission continues, as well as precluding to be stuck with a very low bandwidth path. When the data transmission stops due to loss of all paths, the destination will wait for an adjustable amount of time before discarding the previously received packets. If this timer expires, then the received packets will be removed from the buffer. If a new path is established before the time-out, the source will be restricted to use the same streamID as before, which will enable the sliding window work properly.

4. Packet Structures

The packet structures for route requests and route replies to/from the nodes that are not in our *k*-neighborhood and route errors will be similar to the ones used in DSR. The beacons that are exchanged among the neighbors will be similar to the ones in DSDV.

First Data Packet Header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
                +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                | Type  | Flags |        Header Length          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Source Address                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Destination Address                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    streamID   |          Packet ID                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Address[1]                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Address[2]                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               ...                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Address[n]                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Type: 7. Indicates this is the first packet of a flow. Nodes not understanding this option will ignore it.

Flags: The first bit (the least significant bit) of the Flags field will be set to 1 if the source asks for an acknowledgement.

Header Length: 16-bit unsigned integer. Length of the header, in bytes, excluding Type, Flags and Header Length fields.

Source Address: Set to the address of the node sending the data packet.

Destination Address: Set to the address of the node that the data packet is targeted. This address is obtained from local beacon exchanges (DSDV) or route replies (DSR).

streamID: Set to an integer that is not recently used for a particular destination. It should only be unique to the sender, since the intermediate nodes and the destination node will differentiate flows by binding source address with streamID.

Packet ID: 24-bit unsigned integer. It is used to distinguish the packets of a flow while acknowledging the source and handling out-of-order packets.

Address[1..n]: If the destination node is not a local neighbor, then it is the source route being returned by the route reply. On the other hand, if the destination node is a local neighbor then it will be a maximum of *k*-hops that is available in the routing table of the source. The number of addresses present in the Address[1..n] field is indicated by the Header Length field
(n = (Header Length - 15) / 4).

Intermediate Data Packet Header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
                                                +-+-+-+-+-+-+-+-+
                                                | Type  | Flags |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Source Address                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Destination Address                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    streamID   |           Packet ID                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The fields are explained above, but there are, however, two points to note. The first is that, Type field will be 11 indicating that this is an intermediate packet of a flow. Flags, source and destination addresses and streamID fields are the same as above. Second, the Packet ID field will be set to all *1*s to indicate that is the last packet of a flow.

Acknowledgement Packet Header

The structure of the acknowledgement packet header is the same as intermediate data packet header. The important points to gloss over are as follows. The Type field is set to 13. Most significant bit of Flags field will be set to 1 if this acknowledgement is sent to inform that there is a breakage in the path. The Packet ID and streamID are used to acknowledge a particular packet of a particular flow.

5. Future Work

In the assumptions section (3.1) we have stated that we assumed MAC layer as 802.11 and therefore used bidirectional links. One future direction of this work may be generalizing the algorithm to work with unidirectional links. The previous work by Molle et al. [YM] points out useful methods to introduce directed links into our scheme. In order to achieve this we will have to touch MAC layer and schedule the ongoing traffic such that the collisions will be avoided.

The current probability calculation merely depends on bandwidth estimation. Although RTT and packet losses are implicitly merged into this calculation we want to have a more specific probability calculation technique that will explicitly take delay and packet loss rate into account. We should also consider that these quantities implicitly exist in bandwidth estimation calculation. Also in a TCP session the total time spent to transfer a large amount of data to destination mostly depends on the available bandwidth.

The affect of RTT comes into picture only on a packet loss event. So we cannot ignore the fact that estimated bandwidth should dominate the probability calculation.

The probability calculation in intermediate nodes needs better attention. Since the scheme in [TCPW] is end-to-end, we have to find a way to use it in intermediate nodes as well as in the source node. This is necessary to handle multiple paths as we discussed in 3.3. One possible solution may be caching the ACK reception times from downstream for a particular path and calculate the bandwidth estimation and probability accordingly. Further study on this issue is needed.

The scheduling technique that is going to be used by the source to distribute data among multiple paths should also be considered. Currently, we are using the simplest way, which is tossing a coin and selecting a path accordingly. But this introduces the possibility to use the same path consecutively. To resolve this issue we will consider other scheduling techniques such as round robin, fair queuing and weighted fair queuing.

One another possible direction for future study is trying to achieve loose source routing rather than the strict one we are using. The idea is similar with [YKT] in the sense that there will be segments constructing the whole path. The point is that these segments will be connected by some *reliable nodes* as in [YKT] and the source will only force a packet to follow a subset of these reliable nodes through the destination. The packet will be routed freely inside the segment to the next reliable node. The last reliable node that is closest to the destination will, then, be responsible for the delivery of the packet. There are many issues regarding this scheme such as, placement of the reliable nodes, handling multiple paths, balancing traffic and adapting header reducement technique to loose routing. Fortunately the work by Krishnamurthy et al. [YKT] may help us solve many of the cases.

A final note is the evaluation of the algorithm. In order to have concrete results we need to simulate our algorithm on a network simulator. Our intention is using [ns-2] to model our scheme and have the results. By correctly analyzing the results we may have many other directions to improve this idea.

6. Conclusion

Train of Packets technique has the potential to lower the traffic on the network and decrease the number of packets to transmit by reducing the packet header size and making use of multiple paths. For the networks that have a large number of nodes, the paths between the nodes will consist of many hops. It is unnecessary to put this routing information to each packet of a flow as shown by *Train of Packets* scheme. Load balancing among multiple paths will also improve the receiver perceived throughput and distribute the traffic burden to disjoint paths enabling better utilization of the network.

Consequently, *Train of Packets* technique is a promising work that will (i) reduce traffic control messages due to its hybrid routing approach, (ii) lower data transmission time and number of transmitted packets, (iii) and finally better utilize the network by the use of multiple paths.

7. Acknowledgements

I want to thank Professor Mart Molle for his priceless feedback and patience to my slow progress. Also I am grateful for valuable discussions that I had with Assistant Professor Michalis Faloutsos and PhD candidate Yihua He.

8. References

[AODV] Elizabeth M. Royer and Charles E. Perkins. Ad-hoc On-Demand Distance Vector Routing. In 2nd IEEE Workshop on Mobile Computing Systems and Applications, Feb 1999.
[DSDV] C. Perkins and P. Bhagwat. Highly dynamic destination sequenced distance vector routing for mobile computers. In Proceedings of ACM SIGCOMM, 24(4), Oct 1994.
[DSR] D. Johnson, D. Maltz, Y Hu and J Jetcheva. The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks. IETF MANET Internet Draft, Feb 2002.
[MPLS] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, J. McManus, Requirements for Traffic Engineering Over MPLS
Internet Draft: http://rtg.ietf.org/rfc/rfc2702.txt
[KB] K.J. Astrom and B. Wittenmark, Computer controlled systems (Prentice-Hall, Englewood Cliffs, NJ, 1997).
[ns-2] ns-2 network simulator (ver 2). LBL, URL: http://wwwmash.cs.berkeley.edu/ns.
[OSPF] John Moy, Open Shortest Path First Version 2.
Internet Draft: http://www.ietf.org/rfc/rfc2178.txt
[SHARP] Venugopalan Ramasubramanian and Zygmunt J. Haas and Emin Gun Sirer. SHARP: A hybrid adaptive routing protocol for mobile ad hoc networks, MobiHoc '03, Proceedings of the 4th ACM international symposium on Mobile ad hoc networking, 2003
[TCPW] S. Morris and C. Casetti. "TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links". In MobiCom 2001
[TORA] V. D. Park and M. S. Corson. A Highly Adaptive Distributed Routing Algorithm for Mobile Wirelss Networks. In IEEE INFOCOM, Japan, Apr 1997.
[YKT] Ye, Z., Krishnamurthy S.V., and Tripathi, S.K., -- A Framework for Reliable Routing in Mobile Ad hoc Networks, IEEE INFOCOM 2003.
[YM] An improved topology discovery algorithm for networks with wormhole routing and directed links *Huang, Y.-Y.; Molle, M.L.;* Computer Networks, Volume 31, Issues 1-2, 14 January 1999, Page(s): 79-88
[ZRP] Zygmunt J. Haas and Marc R. Pearlman. The Performance of Query Control Schemes for the Zone Routing Protocol. IEEE/ACM Transactions on Networking, Aug 2001.