Cross-Layer Optimization to Improve TCP Performance in Mobile Adhoc Networks (MANETs)

A Report submitted in partial fulfillment of the requirements for the award of degree of

Master of Technology
in
Information Technology

By
Mahesh Bakshi

Department of Computer and Information Sciences
University of Hyderabad
Hyderabad, India

June, 2007

CERTIFICATE

This is to certify that the project work entitled "**Cross-Layer Optimization to Improve TCP Performance in Mobile Ad Hoc Networks (MANETs)**" being submitted to University of Hyderabad by **Mahesh Bakshi** (Reg. No. 05MCMB16), in partial fulfillment for the award of the degree of Master of Technology in Information Technology, is a bona fide work carried out by him under my supervision.

Ms. Anupama Potluri	Dr. Atul Negi
Project Supervisor,	Project Supervisor,
Department of CIS,	Department of CIS,
University of Hyderabad	University of Hyderabad

Head of Department,	Dean,
Department of CIS,	School of MCIS,
University of Hyderabad	University of Hyderabad

To

My Friends

Acknowledgments

I would like to acknowledge the help of my supervisors, **Ms. Anupama Potluri** and **Dr. Atul Negi**. Their guidance and encouragement enabled me to accomplish my project successfully .

I am thankful to my project mates, who helped me throughout the project.

Mahesh Bakshi

Paper Submitted

1. Submitted a paper titled ***"Comparison of goodput using different flavors of TCP for different transmission rates"*** to M.V. Chauhan students paper contest, IEEE Indian council, August, 2006.

   In this paper we present a simulation study of different flavors of TCP, namely Tahoe, Reno, New Reno, Vegas and SACK with different transmission rates and analyze the goodput using network simulator *ns-2*. We found that the goodput is best for Vegas as compared to other flavors for high transmission rates. We analyze the reasons for this behavior and present our conclusions.

Abstract

Transmission Control Protocol (TCP) has been carefully designed for reliable wired networks. A packet loss event is treated by TCP as a loss due to congestion in the case of wired networks. In MANETs, packet losses are more likely to occur due to losses in wireless transmission, route errors or even due to congestion. TCP performs poorly in terms of the throughput achieved under these conditions. We analyze several schemes in the MANET literature [1], [2], [3], [4], which propose cross layer optimizations to improve TCP performance. It has been shown that such cross-layer optimizations have been effective in improving TCP throughput. From our review we observe that such MANET proposals seem to have an issue of unfairness in the TCP flows [5]. We propose to introduce a cross-layer trigger to TCP which on detection of a route failure freezes(suspends) its transmission on all connections with that destination. Later, when a new route is discovered to a destination that had its transmission suspended, the routing protocol sends another trigger to resume data transmission. This scheme could be used to improve the performance of an existing MANET routing protocol Dynamic source Routing (DSR). We propose to compare the performance of various flavors of TCP such as Reno, NewReno and Tahoe to understand which of these versions works would best respond to our proposed cross-Layer trigger from a throughput perspective. In addition, we also propose to understand the impact on congestion window of the changes on various flavors of TCP.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction to MANETs

The vision of the mobile ad-hoc network[14] is to support robust and efficient operation in mobile wireless networks by incorporating routing functionality into mobile nodes. Such networks are seen to have dynamic, rapidly-changing, random, multihop topologies. MANET has no permanent infrastructure at all. A MANET is depicted in figure 1.1. Mobile Ad Hoc Networking is a technology that enables wireless networking in environments where there is no wired or cellular infrastructure (eg, battlefield, disaster recovery, etc); or, if there is an infrastructure, it is not adequate or cost effective.

1.1.1 Applications of Mobile Ad Hoc Networks

Some of the applications of MANETs include battlefield, homeland security (where unmanned vehicles are deployed in urban areas hostile to man), Mars explorations, disaster recovery etc. where we require to deploy network in an unfriendly, remote infrastructure-less area.

One of the commercial application of MANET which is used to extend a home or Campus network to areas not easily reached by wireless telephony and wireless LANs, that is opportunistic ad hoc networking. It is also useful when infrastructure is cut into pieces due to natural forces or terrorists. Another important area that has propelled the ad hoc concept is sensor networks.

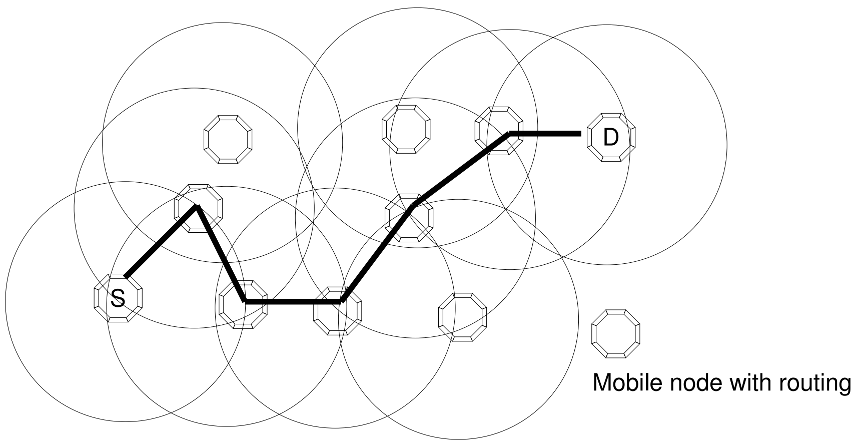


Figure 1.1: Mobile Ad-hoc Network

1.1.2 Salient Characteristics of MANETs

Here we list some of the MANET characteristics.

- Dynamic topologies: Nodes are free to move arbitrarily, which causes topology to change randomly and rapidly at unpredictable times. It may have bidirectional or unidirectional links.

- Bandwidth-constrained, variable capacity links: Wireless links have lower capacity when compared with that of wired parts. In addition, throughput of wireless communication is less because of multiple access, fading, noise, interference and weather conditions, etc.

- Energy-constrained operation: Some or all nodes in the MANETs use limited battery power. The most important design criteria is to optimize the energy conservation.

- Limited physical security: Mobile wireless networks are more prone to physical security threats than are fixed-cable networks. We need to carefully consider eavesdropping, spoofing, and denial-of-service attacks.

1.2 Cross Layer optimization

There has been much said in literature that although layered architectures have served well for wired networks, they are not suitable for wireless networks. Generally speaking, cross-layer design refers to protocol design done by actively exploiting the information in a different layer from another protocol layer to obtain performance gains.

A Definition of Cross-Layer Design[13] One of the definitions of Cross-Layer design is as follows: Protocol design that violates the layered communication architecture of the protocol stack is cross-layer design with reference to that particular layered architechture. Some of the observations of this definition are given below. Examples of violation of a layered architecture include creating new interfaces between layers, redefining the layer boundaries, designing protocol at a layer based on the details of how another layer is designed, joint tuning of parameters across layers, and so on. Violation of a layered architecture involves giving up the luxury of designing protocols at different layers independently. Protocols so designed impose some conditions on the processing at other layer.

A Snap shot of Cross-Layer Design Proposals[13] Following are the different basic ways to violate the layered architechture:

- Creation of new interfaces (Fig. 1.2 A, B, C).
- Merging of adjacent layers (Fig. 1.2 D).
- Design coupling without new interfaces (Fig. 1.2 E).
- Vertical calibration across layers (Fig. 1.2 F).

Creation of new interfaces can be done in three ways, namely: Upward information flow, Downward information flow and Back and forth information flow as shown in the figure 1.2 A, B, C. Designed coupling of new layers can be explained using an example as given below. If the physical layer is capable of receiving multiple packets then the functionality of MAC should be changed accordingly to support this functionality. The motivation of last classification is easy to understand. Basically, the performance seen at a level of the application is the function of the parameters at all the layers below it. Therefore joint tuning can help to achieve better performance than individual setting of parameters. Which means we are calibrating vertically.

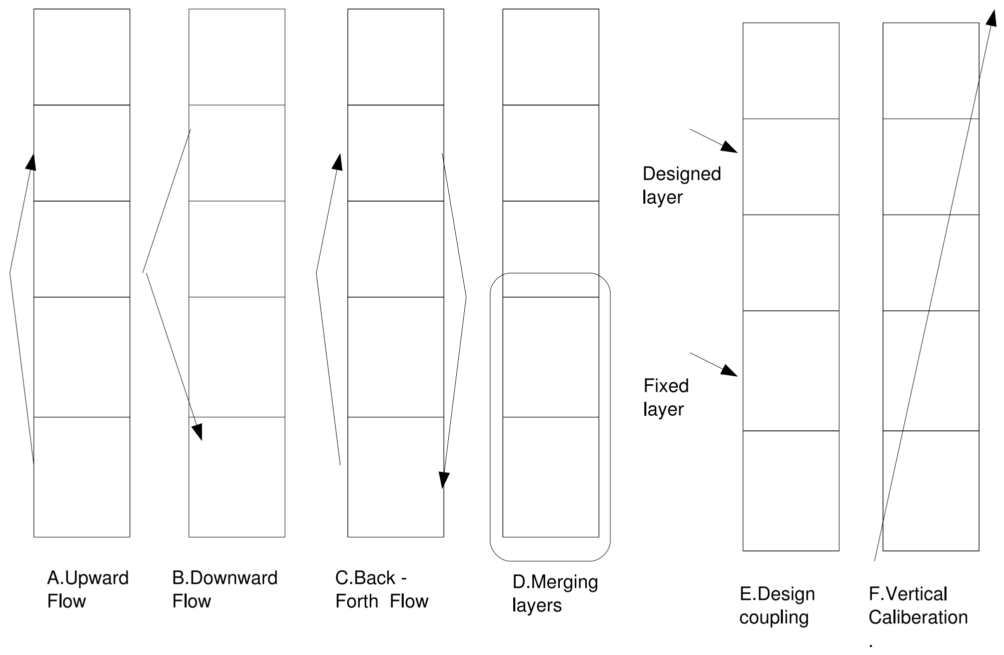


Figure 1.2: Basic cross layer design proposals[13].

1.3 An overview of Transmission Control Protocol

The TCP/IP protocol suite consists of the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) as the transport protocols. UDP is a simplistic transport layer solution that merely provides labeling functionality for applications. TCP is a complex transport layer protocol that provides applications with reliable, end-to-end, connection-oriented and in-sequence data delivery. It performs both flow control and congestion control on behalf of the applications, recovers from packet losses in the network, and handles resequencing of data at the receiver. For more information on congestion control algorithms namely slow start, congestion avoidance, fast retransmit and fast recovery refer [8].

1.4 Overview of Dynamic source Routing [6]

Dynamic Source Routing (DSR) is characterized by the use of source routing. That is, the sender knows the complete route to the destination which is stored in a route cache. The data packets carry the source route in the packet header. When a node attempts to send a data packet to a destination for which the route is not known, it uses a route discovery process to determine a route. Route discovery floods the network with route request packets. Each node receiving a request, rebroadcasts it unless it is the destination or it has a route to the destination in its route cache. Such a node replies to the request with a route reply packet that is routed back to the original source. The request builds up the path traversed so far. The reply routes itself back to the source by traversing this path backward. The route carried back by the reply packet is cached at the source for future use. If any link on a source route is broken, a route error is generated. Route error is sent back toward the source which erases all entries in the route caches along the path that contains the broken link. A new route discovery will be initiated by the source.

1.5 Motivation

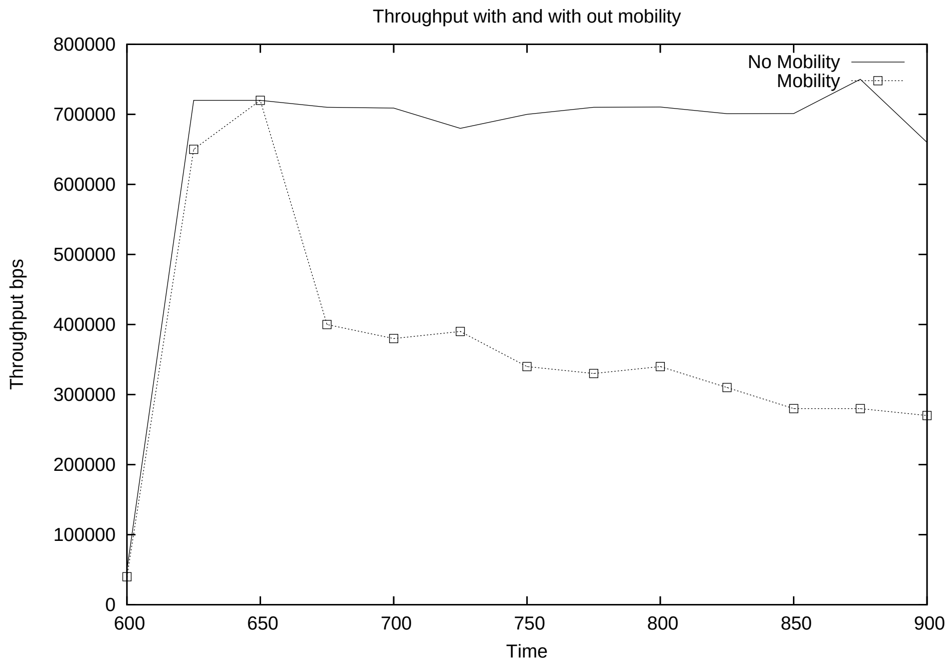


Figure 1.3: Scenarios of TCP unfairness

We motivate our work by showing a simulation. The simulation in figure 1.3 shows plotting of throughput on the y-axis against time. This is done for a single TCP flow from a source to destination for mobile nodes and static nodes, the mobile nodes suffered a much lower throughput. Also for a mobile node it was observed that a number of link failures and route failures had occurred. So from this it is observed that although with a single flow only there is a great reduction in throughput due to packet losses. It is also seen that present mechanisms that control congestion windows are inadequate and do not distinguish between congestion losses or those due to other reasons such as route errors or link failures.

Several cross layer designs have been attempted to improve TCP performance in Mobile Ad Hoc Networks. some of them are TCP-Feedback[2], Explicit Link Failure Notification[3], Ad Hoc TCP[7]. TCP-Feedback is not implemented in real ad hoc network scenario but it is emulated from the view point of the transport layer. Explicit Link Failure Notification[3] and Ad Hoc TCP[7] use probe packets to know about reestablishment of route but it leads to congestion at higher loads. So we want to develop our own cross layer scheme using two cross layer triggers from network layer to transport layer to improve TCP performance by distinguishing between link failures and congestion.

1.6 Organization of project report

In this chapter we have introduced the problem of TCP behavior adjustment in MANETs. We have looked at certain basic concepts related to cross-layer design and MANETs. In next chapter we explain about the various problem TCP has in adhoc networks and different proposals exists to tackle these problem. We also explain about different cross layer proposal which are present in the literature to distinguish between link failure and congestion separately to improve TCP performance. We also give overview of a scheme from literature to improve TCP unfairness problem. In later chapter we explain about our new cross layer proposal to improve TCP performance in Mobile Ad Hoc networks. After that we have chapter which talk about functionality of DSR and TCP in *ns-2* and what are the modifications done for that code to implement our proposal. Followed by conclusive remarks.

Chapter 2

Existing Work

2.1 TCP in Mobile Ad Hoc Networks

Mobile Ad Hoc Networks are multi-hop wireless networks that consist of mobile nodes that can dynamically self-organize. The following are some of the challenges in MANETs namely: lossy channels, hidden and exposed stations, path asymmetry, network partitions, routing failures and power constraints. Transmission Control Protocol performance degrades in MANETs for the following reasons:

- TCP is unable to distinguish between losses due to route failures and losses due to congestion.
- TCP unfairness.

Transmission Control Protocol (TCP) has been carefully designed for reliable wired networks. Route failures and disruptions are very less since the network is fixed. Therefore, packet loss, which is detected by TCP as a timeout, can be reliably interpreted as a symptom of congestion in the network. A packet loss event is treated by TCP as a loss due to congestion in the case of wired networks. In MANETs, packet losses are more likely to occur due to losses in wireless transmission or due to the mobility of nodes causing routing errors and hence packet losses, in addition to congestion. The more serious problem is that of route errors which can occur very frequently and are unpredictable due to mobility of the nodes. Whenever a route becomes invalid, it will take time to reestablish a route to the destination. During this time no packet is reachable to the destination and packets are queued and some of them may be dropped. This causes a timeout at source and TCP misinterprets this as congestion. Consequently the source:

- Retransmits unacknowledged packets on timeout.
- Exponentially backs off retransmission time and reduces congestion window, thus reducing transmission rate.
- Enters slow start phase before resuming transmission at normal rate to avoid congestion.

Retransmission is undesirable as there is no route to the destination. It wastes mobile hosts battery power and the channel bandwidth. TCP performs poorly in terms of the throughput achieved under these conditions.

We mainly concentrate on the first problem that is, TCP is unable to distinguish between losses due to route failures and losses due to congestion. General discussion of the second problem followed by explanation of NRED is given and then we move to different proposals to solve our first problem.

2.2 TCP unfairness problem

Here we introduce another related issue that of TCP unfairness. Figure figure 2.1(A) explains the problem of TCP unfairness. Ellipse in the figure 2.1 represents the transmission range of the node1. Here all the nine nodes are positioned such that distance between two neighbor nodes is around 250m, i.e., its transmission range. Three FTP flows are started at the same time as shown in the figure 2.1. After making simulation it has been observed that only one of these three flows uses all the resources and the throughput of other two flows is negligible which is shown in figure 2.1(B). Figure 2.1(C) shows the one hop unfairness scenario. In this scenario, we start two flows: one is of one hop and other is of two hops. Irrespective of when we start the flows, the one-hop flow uses all the resources as soon as it starts. In a later part of this section we explain the Neighborhood RED [5] proposal to handle TCP unfairness problem.

2.2.1 Neighborhood RED[5]

TCP unfairness results from the nature of the shared wireless medium and location dependency. Shared wireless constraint implies space is a shared medium. TCP flows that do not even traverse common nodes may compete for shared space and interfere with each other. Locational dependency triggers problems like channel capture,

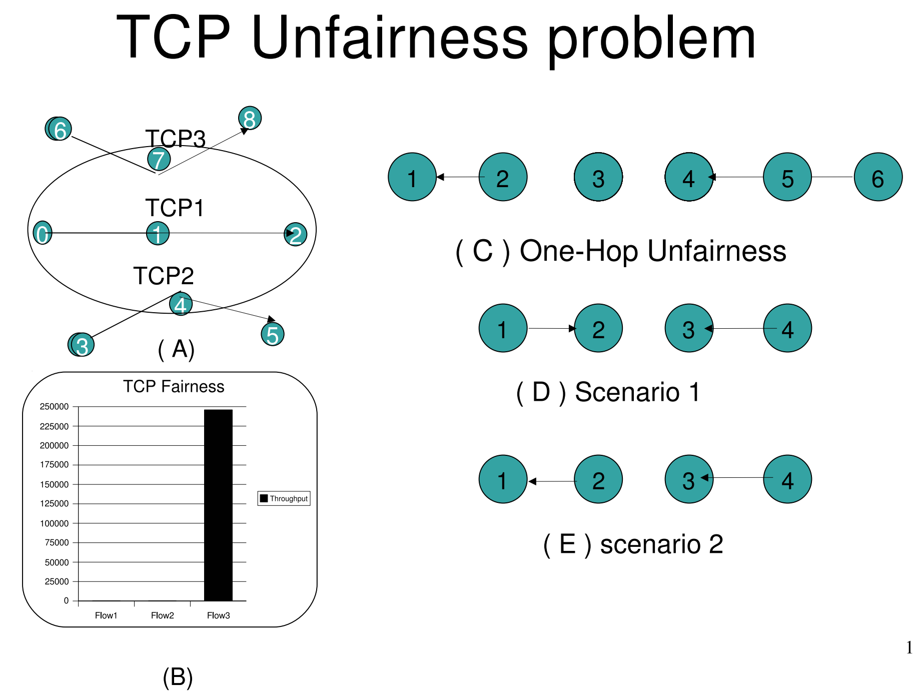


Figure 2.1: Scenarios of TCP unfairness

hidden and exposed terminal conditions and the binary exponential backoff of IEEE 802.11 MAC etc.

Why RED does not work Unlike in wired networks, RED does not solve TCP's unfairness in MANETs. The following are the major factors that stop RED from improving TCP unfairness.

- A TCP connection penalized in channel contention may experience a queue buildup and dropping packets from this queue indeed increases unfairness.
- Congestion does not happen in a single node, but in an entire area involving multiple nodes.

Since there are multiple nodes involved in congestion they should coordinate their packet drops instead of dropping independently which is essentially what Neighborhood RED does.

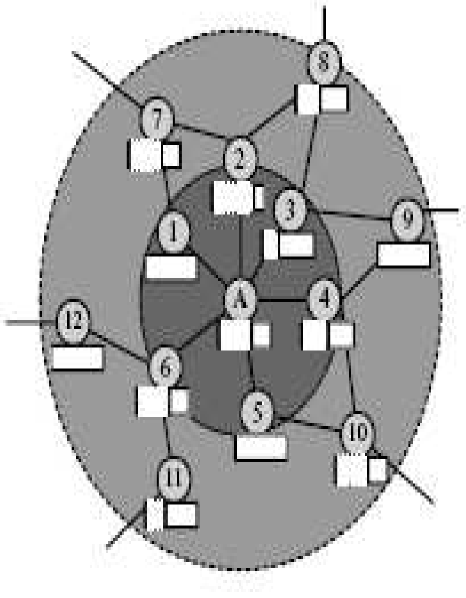


Figure 2.2: 'A' node neighborhood and its distributed queue [5]

2.2.1.1 Simplified neighborhood queue model

Neighborhood*: A node's neighborhood consists of the node itself and the nodes which can interfere with this node's signals.* A node's neighborhood includes both its 1-hop neighbors and 2-hop neighbors. 2-hop neighbors are also included because they may interfere with the nodes that are transmitting to any 1-hop neighbor and collision may happen. Distributed neighborhood queue is shown in figure 2.2 This queue consists of the multiple queues of the neighborhood nodes. It is not a FIFO queue. Nodes multiple sub-queues have different relative priorities in terms of acquiring wireless channels and these priorities of the queues change dynamically due to topology and traffic pattern changes.

2.2.1.2 Neighborhood random early detection

The mechanism of Neighborhood Random early detection can be explained in the following steps:

- Each node keeps estimating the size of the neighborhood queue.
- If queue size is greater than or equal to minimum threshold then get drop probability which is computed using RED algorithm,

- This drop probability is propagated to neighborhood nodes for cooperative packet drops.
- Local queues compute local drop probability based on its channel bandwidth usage and drop packets accordingly.

To do so they propose three mechanisms namely:

1. **Neighborhood Congestion Detection** (NCD): How to detect early congestion or average queue size.
2. **Neighborhood Congestion Notification** (NCN): When and how does a node inform its neighbors about the congestion.
3. **Distributed Neighborhood Drop Probability** (DNDP): How do neighbor nodes calculate their local drop probabilities.

This proposal works well in a single bottleneck node. According to our knowledge this is the only complete proposal present in the literature to improve TCP unfairness problem. But as the number of bottleneck nodes increases it has significant effect on throughput. In general Ad Hoc network scenarios we have multiple bottleneck nodes. We can analyse that if we want to improve un-fairness then there will be sufficient drop in the throughput of TCP. So a new mechanism has to be made which could give right balance between throughput and un-fairness of TCP even at high loads.

2.3 Cross-layer Designs to Improve TCP Throughput

Proposals to solve the problem of TCP being unable to distinguish between packet losses due to congestion and route failure are listed in the figure 2.3. Here we mainly concentrate on the cross-layer proposals to improve TCP performance.

In the remaining sections, we give an overview of the proposals to improve TCP performance using cross-Layering namely TCP-Feedback, Explicit Link Failure Notification, ATRA and Adhoc TCP.

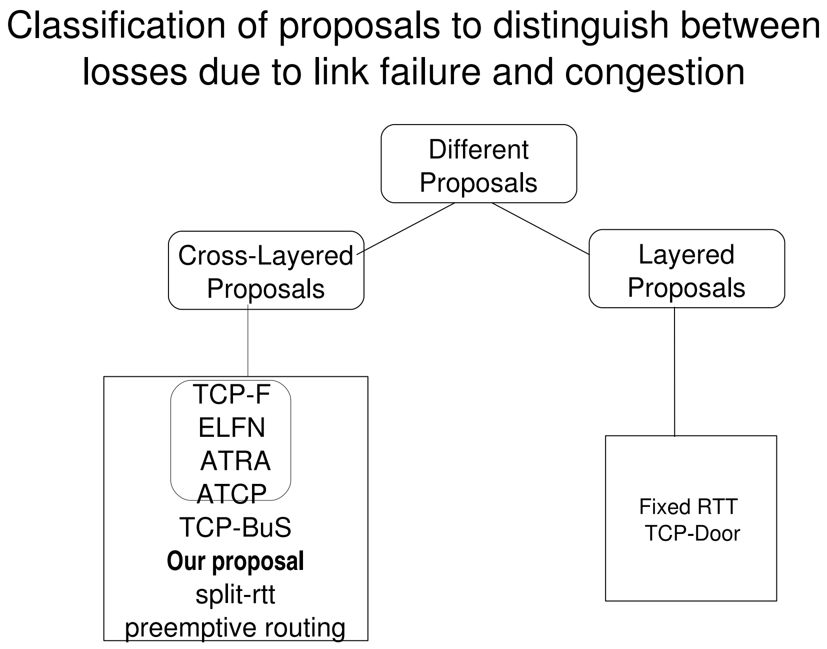


Figure 2.3: Classification of TCP proposals

2.4 TCP Feedback model[2]

Here we review certain schemes in this category. First we will start with TCP-Feedback scheme.

2.4.1 Assumptions of TCP-Feedback

- A reliable data link layer protocol : If a packet is not delivered at link layer, then this event is notified to the network layer.
- We require a suitable routing to perform certain additional actions, which include sending feedback messages to transport layer.
- When a packet cannot be sent by the link layer, they treat the situation as the failure due to mobility. It is assumed that when a route failure occurs, **a finite time elapses** until the route is restored and communication can be resumed.

- The routing protocol may maintain redundant routes between different sources and destinations. All packets carry the source and destination IDs so that the network layer can identify them.
- Each mobile host broadcasts its packets. No assumption is made about medium access protocol. Wireless links are bidirectional.

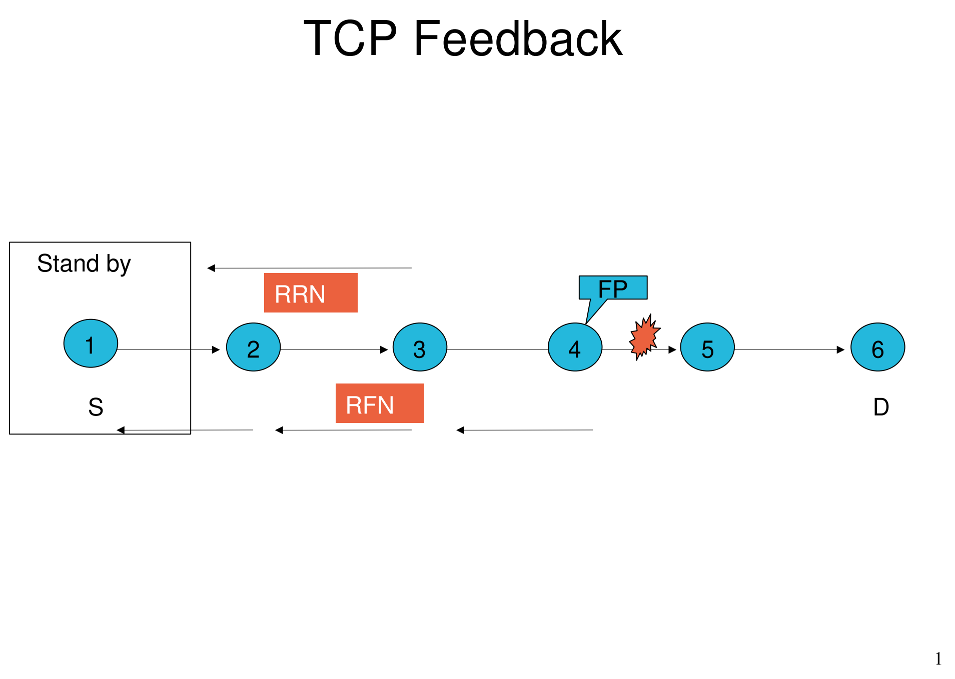


Figure 2.4: TCP Feedback Approach

2.4.2 TCP-Feedback Design

Figure 2.4 shows a pictorial representation of TCP-Feedback scheme which is explained below. A source(S) is sending packets to a destination(D) mobile host(MH). If an intermediate MH (failure point) detects the disruption in the route due to the mobility of the next MH along that route, it sends a route failure notification (RFN) packet to the source and records this event. Each intermediate node on receiving the RFN invalidates that route and prevents packets from passing through that route. If the intermediate node knows of an alternate route to the destination, it can be used for further communication and it discards the RFN.

On receiving RFN at source, it goes into *snooze* state and performs the following functions:

- It stops sending packets and marks all the existing timers invalid.
- Freeze send window and retransmit timer value.
- Starts a *Route failure timer* which corresponds to worst case route reestablishment time. It avoids blocking in *snooze* state and invokes congestion control mechanism after this timer expires.

It remains in snooze state until it receives Route Reestablishment Notification (RRN). Let one of the intermediate nodes which has received RFN have an alternate route to the destination. Then it sends a RRN to source through intermediate nodes. On receiving RRN the source changes its state from *snooze* to *active*. It then flushes out all unacknowledged packets in its current window. The number of retransmitted packets directly depends on the current window size. It resumes sending packets at the same rate as before RFN. TCPs congestion control mechanism can now take over and adjust to the existing load in the system.

Features of TCP-Feedback that are affecting an improved throughput are as follows:

- Assumes routing protocol has a mechanism to transfer feedback.
- Assumes worst case RRD for handling congestion.
- Intermediate nodes are responsible for RRN.

2.5 Explicit Link Failure Notification model[2]

The main objective of Explicit Link Failure Notification model is to distinguish between packet losses due to link failures and congestion control separately to increase the throughput. There are several ways in which one can send ELFN messages. The following are the two different ways in which one can send ELFN messages: one is to use a host unreachable ICMP message to notify the sender; the second method is to use the route failure message of the routing protocol to notify the sender about the link failure. The second approach has been used by ELFN model. The DSR route error message is modified to carry a payload similar to host unreachable ICMP

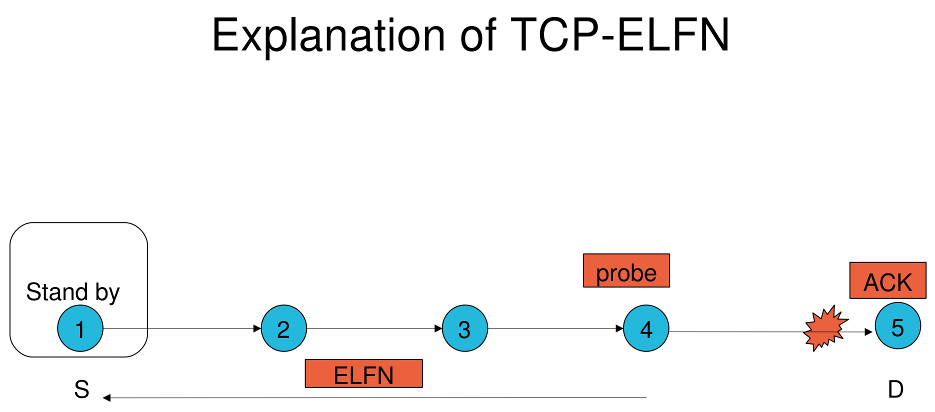


Figure 2.5: TCP ELFN model

message. In particular it includes the fields of the TCP/IP headers of the packet to identify the connection. It also includes the TCP sequence number. Figure 2.5 shows the functionality of this proposal.

The TCP source, on receiving the link failure message, disables the congestion control mechanism until the route has been restored to that destination. On receiving ELFN message, the TCP source disables its retransmission timers and enters into the *standby* mode. TCP sender comes to know about route reestablishment using probe packets. In standby mode, a probe packet is sent at periodic intervals to see if a route has been established. On receiving acknowledgment, it comes out of the standby mode and restores its retransmission timers. Then normal TCP action is taken over.

The probe messages needed to determine the route reestablishment lead to a delay in restoring TCP to its normal state from the *standby* state. In addition, these are additional overhead if there is congestion. ELFN performance is worse than basic TCP in heavy load scenario with mobility due to aggressive probing.

2.6 ATRA framework

Here we look at ATRA framework [4] that was proposed to improve TCP performance.

2.6.1 Performance degradation components identified by ATRA

This framework argues that ELFN is a necessary but *not* sufficient mechanism to improve TCP performance over ad-hoc networks.

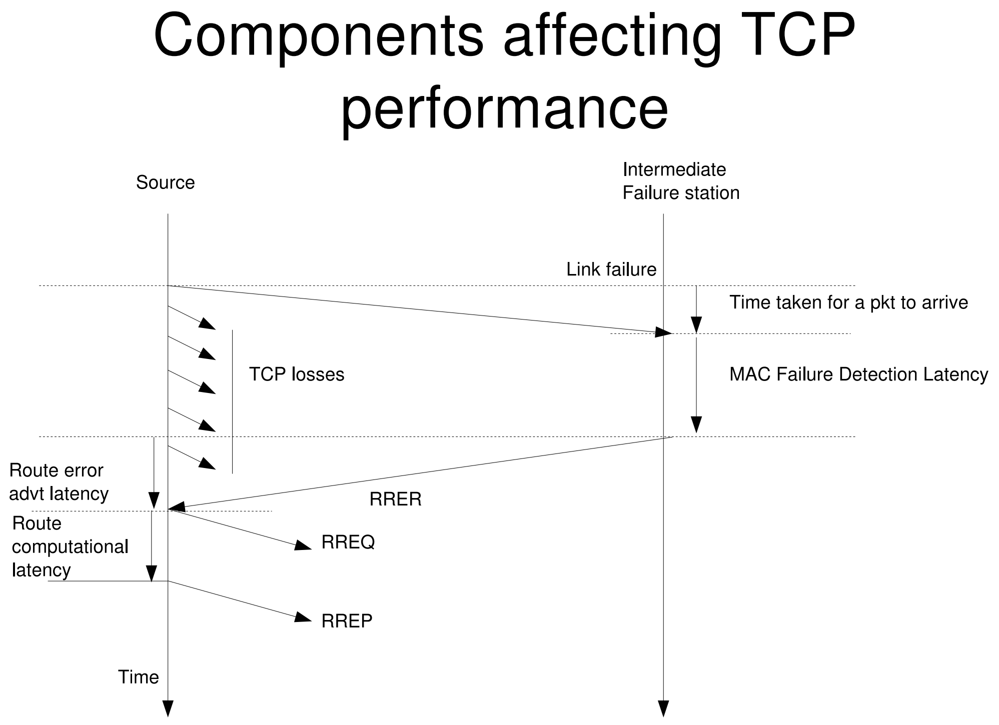

Components affecting TCP performance
Source
Intermediate Failure station
Link failure
Time taken for a pkt to arrive
TCP losses
MAC Failure Detection Latency
Route error advt latency
RRER
Route computational latency
RREQ
RREP
Time

Figure 2.6: ATRA performance degradation components

They identified the following different components that contribute to performance degradation in the presence of mobility which are a superset of the factors of ELFN (Figure 2.6 depicts these components.) :

- **TCP losses**: Every route failure induces TCP-window worth of packet losses. they have direct impact on performance degradation.
- *MAC Failure detection time*: It is the actual time taken to detect link failure from when the failure occurs. This time increases with network load. If this

time is high then the likelihood of TCP source pumping data into network is also high.

- **MAC packet arrival**: If a link failure is detected then it is only informed to the source of the packet that triggered detection. If other source is also using the same link then it will take time to know about link failure.
- **Route computational time**: It is the time taken to recompute the route to destination.

2.6.2 Framework of ATRA

ATRA framework consists of three simple and easily implementable mechanisms at the MAC and routing layers to enhance TCPs performance. ATRA framework is based on the following three goals.

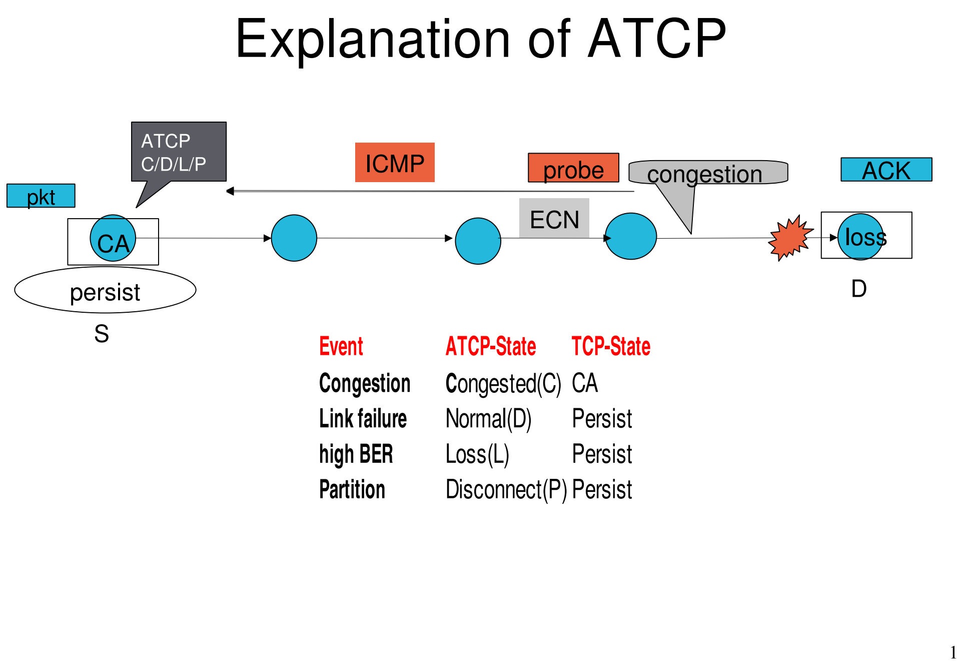
- To reduce the probability of route failures.
- To predict route failures in advance to find alternate route before existing route fails.
- To minimize the latency in conveying route failure information to the source.

ATRA framework has three mechanisms targeted toward each of the above goals respectively.

- **Symmetric Route Pinning**: ACK path of a TCP connection is always kept the same as the data path.
- **Route Failure Prediction**: to predict the occurrence of a link failure based on the the signal strength.
- **Proactive Route Errors**: If multiple source nodes use the same link which has failed, all will be notified immediately after link failure. This is done using the cache each node maintains about all the sources using that link

2.7 Ad Hoc TCP (ATCP) [7]

This model uses network layer feedback from intermediate hops to put TCP source into persistent state, congestion state or retransmit state. Thus when the network



Event	ATCP-State	TCP-State
Congestion	Congested(C)	CA
Link failure	Normal(D)	Persist
high BER	Loss(L)	Persist
Partition	Disconnect(P)	Persist

Figure 2.7: Ad Hoc TCP functionality

is partitioned TCP sender is put into persistent mode where it does not needlessly transmit and retransmit packets. Otherwise if packets are lost due to error and not due to congestion then those packets are retransmitted but it will not invoke congestion control mechanism. Finally when network is really congested then TCP will invoke congestion control mechanism normally. Figure 2.7 explains the mechanism of Ad Hoc TCP proposal.

It does not modify TCP but inserts a thin layer between IP and TCP. It listens to the network layer information through ECN and ICMP destination unreachable message. Then it puts TCP in its corresponding state. ATCP is transparent in the sense that it is interoperable with standard TCP and connections are handled in the normal manner. As in ELFN, it uses probe packets for route reestablishment which can lead to congestion as traffic increases. It performs worse than normal TCP at high traffic loads.

Table 2.1: Performance comparisons of various schemes

S.No.	*Performance parameters*	*Adhoc TCP [7]*	*ELFN [3]*	*TCP-F [2]*
1	***Packet losses due to high BER***	Handled	Not handled	Not handled
2	***Route changes***	ICMP-persist state	ELFN-standby	RFN-snooze state
3	***Network partition***	ICMP-persist state	ELFN-standby	RFN-snooze state
4	***Packet reordering***	Done by ATCP	not handled	not handled
6	***CWND***	Reset for each new route	uses old cwnd	uses old cwnd

2.8 TCP flavors description

In this section we try to explain the description of all severalflavors of TCP namely Tahoe, Reno, New Reno, Vegas, SACK.

2.8.1 TCP Tahoe[8]

In TCP Tahoe the congestion window is reduced to half of its size when congestion is detected.

Here the packet loss is detected in 2 ways.

- Through the use of the duplicate acknowledgments (*dupacks*).
- If the acknowledgment is not received before RTO then it indicates a packet loss.

Typically Tahoe waits for 3 *dupacks* before inferring the packet loss and immediately resends the packets. This mechanism is called Fast Retransmit which is followed by normal actions. TCP Tahoe does not deal well with multiple packet drops within a single window of data.

2.8.2 TCP Reno[9]

It is an improvement over Tahoe by changing the way it reacts after detecting a loss through *dupacks*. Reno introduces Fast Recovery[8] which is activated after Fast Retransmit. Here if the sender is still receiving acks from the receiver, then the sender should not fall into slowstart since the congestion is not too heavy. So, the sender can keep on sending as the flow still exists but should use resources less vigorously.

The above mechanism is implemented by halving the size of the congestion window(*cwnd*) and the value is stored in *ssthresh*. Then the *cwnd* is set to *ssthresh*+(3*MSS) which include only upto half of the number of segments just before the Fast Recovery. As the receipt of *dupacks* under Tahoe does not move the window, inflating the size of **cwnd** upon each *dupack* is the only way to shift the window. Upon the receipt of each *dupack* the *cwnd* is inflated by one segment. After receiving $w/2$ dupacks (where w is the size of *cwnd* before Fast Recovery) the window will be ready to include newer ones. After all the *dupacks* have been received the next *ack* will be the normal *ack*. In this case the sender should exit Fast Recovery and set its *cwnd* to *ssthresh* ($w/2$). This value is still the value of *cwnd* before Fast Recovery.

By using Fast Recovery the sender is using a *cwnd* that is half the size of *cwnd* before the loss. It increases cwnd by one segment only when one segment leaves the network. So this forces Reno to send less vigorously. This leads to reduced utilization of the network.

Though Reno is better than Tahoe while dealing with single packet loss, it is not better than the latter when multiple packets are lost within a window of data.

2.8.3 TCP New Reno[10]

TCP New Reno is same as TCP Reno with more intelligence during Fast Recovery. This modification of Reno showed that Reno can be improved without adding SACKs (selective acknowledgments).

TCP New Reno utilizes the idea of the partial *acks*: when there are multiple packet drops the *acks* for the retransmitted packet will acknowledge some, but not all the segments sent before the Fast Retransmit. In New Reno the partial *acks* are taken as an indicator of another lost packet unlike in Reno where partial *acks* do not take New Reno out of Fast Recovery.

In the above way the New Reno retransmits one packet per RTT until all the lost packets are retransmitted and avoids multiple Fast Retransmits from a single window. The disadvantage of this is to recover from a loss, many RTTs are required.

2.8.4 TCP SACK[11]

TCP may experience poor performance when multiple packets are lost from one window of data. SACK uses Selective Acknowledgment (SACK) mechanism, combined with a selective retransmission policy. The SACK option is to be sent by a data receiver to inform the data sender of non-contiguous blocks of data that have been received and queued. The data receiver awaits the receipt of data (perhaps by means of retransmissions) to fill the gaps in between received blocks. When missing segments are received, the data receiver acknowledges the data normally by advancing the left window edge in the Acknowledgment Number Field of the TCP header. Each contiguous block of data queued at the data receiver is defined in the SACK option by two 32-bit unsigned integers in network byte order: Left Edge of Block is the first sequence number of this block; Right Edge of Block is the sequence number immediately following the last sequence number of this block. Each block represents received bytes of data that are contiguous and isolated; that is, the bytes just below the block, (Left Edge of Block - 1), and just above the block, (Right Edge of Block), have not been received. This allows the sender to resend all missing segments without waiting for a timeout.

2.8.5 TCP Vegas[12]

In Reno the window size continues to increase until the packet loss occurs, then the size of the window is reduced and this results in the degradation of the throughput of that TCP connection. The TCP Reno can detect congestion only by packet loss. However, decreasing the window size is not enough when the connection itself causes the congestion because of its larger window size.

The key idea of TCP Vegas is controlling the window size so that the packet loss does not occur in the network and hence the degradation of the throughput due to the reduction of the window size can be avoided.

TCP Vegas controls its window size by observing the RTTs of the packets that the sender has sent before. If these values are large it recognizes the congestion and reduces window size, if the values are small the TCP Vegas host determines that the network congestion is reduced and hence the size of the window is increased.

TCP Vegas also has another feature called Slow Start. Here in this phase the rate of increasing the window size is half of that in TCP Reno and TCP Tahoe. TCP Vegas has been shown to achieve throughput better than Reno.

In this chapter we presented various problems of TCP and gave overview of different cross-layer proposals present in literature to improve TCP performance. We also gave description of various TCP flavors. In the next chapter we will gives description of our new proposal to improve TCP performance in Mobile Ad Hoc Networks.

Chapter 3

Cross Layer Optimization to Improve TCP Performance in MANETs

3.1 Behavior of Proposed scheme

Transmission Control Protocol (TCP) has been carefully designed for reliable wired networks. Whenever there is packet loss, TCP treats it as congestion. In Mobile Ad Hoc Networks (MANETs), packet losses occur due to less reliable wireless medium of communication as well as mobility of the nodes that leads to loss of routes. Thus, TCP performs poorly in terms of the throughput achieved. Our approach is to avoid extra signaling messages to signal TCP that the loss is due to route failure and not congestion. We modify DSR so that RERR messages are used to send a cross-layer trigger to TCP. We explain the scheme in detail in the rest of the chapter.

3.1.1 Our approach

The main purpose of our approach is to distinguish link failures occurring due to mobility of the nodes from congestion. We aim to achieve this using cross layer information. We want to maintain a global list which stores all the references of the active TCP connections which uniquely identifies each TCP connection. During a session, a node on the path to the destination may move away, resulting in a route break. The node experiencing forwarding problem sends RERR message to the source. When the source receives this message, a cross layer trigger named freezeTCP is sent to all the TCP flows which are using this route and these TCP connections

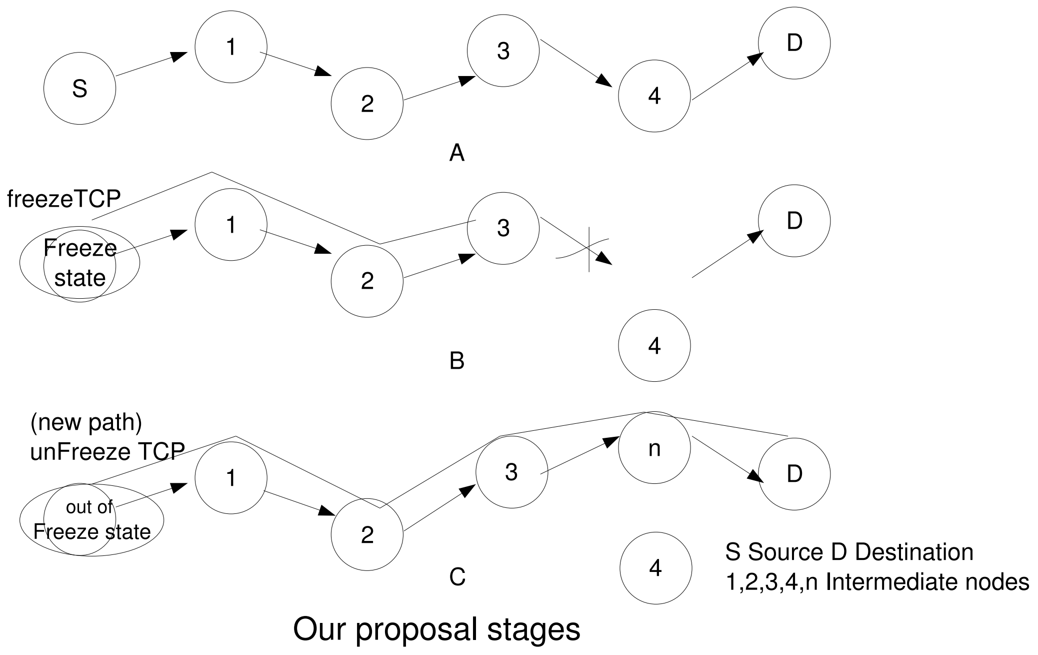


Figure 3.1: Stages in new proposal

go into a freeze state. In this freeze state, we want to stop further transmission of packets along the route and freeze its retransmission timer. When a new route to the destination is discovered, we want to send another cross-Layer trigger which we name as unFreezeTCP to start transmission of data and restart all the timers which are frozen.

3.1.2 Explanation of our proposal using block diagram

Here we give the detailed explanation of our proposal using a block diagram as shown in figure 3.2. Here every block represents an action to be performed and arrows correspond to resource dependency. Now let us explain each and every block's functionality. Whenever a new object for a TCP agent is created then the reference of that object is stored in a global list using *addTCPList*. This global list is named as *TCPList*. In the same way, whenever an object is destroyed then we remove that object's reference from our global list (*TCPList*) using *delTCPList*. This global list (*TCPList*) corresponds to the list of active connections of Transmission Control Protocol(TCP). This information of active connections can be used for transfer of our

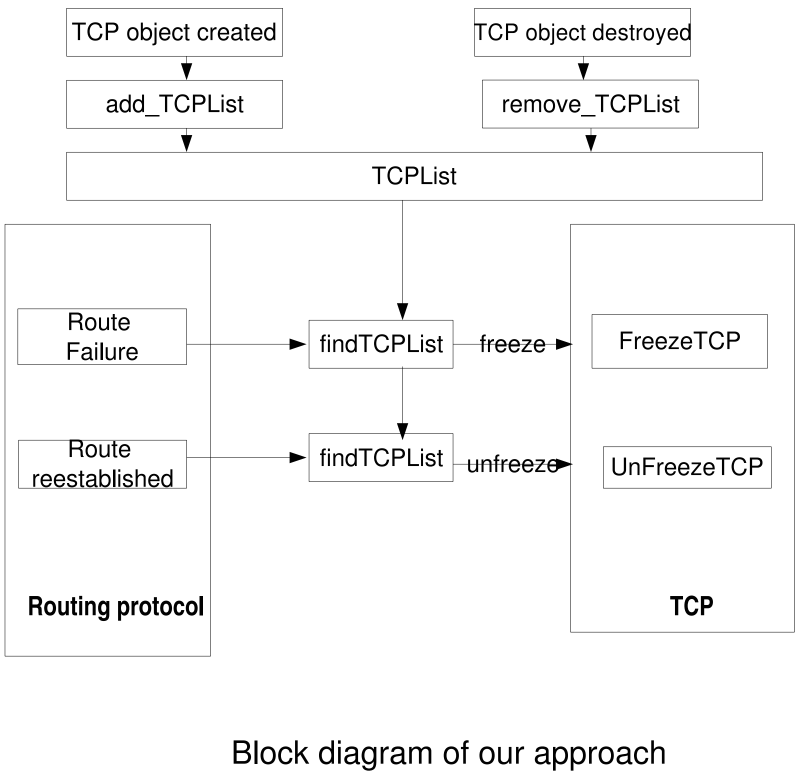


Figure 3.2: Block diagram of new proposal

cross-layer trigger from network layer to transport layer, which we explain in later part of this section. If a node is transferring data to its neighbor, if its neighbor moves out of its transmission range because of mobility and no alternate route is found using local repair then a *route failure* is triggered at the routing layer of the failure point. Then this node, where route failure is detected sends this route failure to the source which has originated this flow. We need to use this route failure as a mechanism to send our cross-layer trigger. We use a global list(TCPList) to find the reference of the corresponding TCP flow in the list. We use *findTCPList* to find the corresponding reference of the TCP object. *findTCPList*, in turn, compares the source-destination addresses of all the TCP objects in the global list(*TCPList*) with that of source-destination address pair of the route for which we got route failure.

Using the reference of the corresponding TCP object, we send a cross-layer trigger named as *freeze* from the network layer to transport layer. This cross-layer trigger invokes a new API in the transport layer named *FreezeTCP*. The functionality of this API is to set a flag named *freezeFlag*, cancel all the timers of the TCP. *freezeFlag* indicates whether our TCP agent is in freeze state or not. When ever we are in freeze state we do not transfer data at the network layer and freeze our congestion window size.

At the same time routing protocol sends a route request from network layer to find an alternate route to the destination. If an alternate route has been found(*Route re-established*), then using *findTCPList* we get the TCP object reference of the corresponding flow as explained before. We then send a cross-layer trigger named *unfreeze* from network layer to transport layer. This cross-layer trigger unfreeze invokes an API named *UnFreezeTCP* at the transport layer. *UnFreezeTCP* resets *freezeFlag* and restarts all the timers which are frozen by *FreezeTCP* API. If *freezeFlag* is not set then normal transmission of data takes place at the transport layer and congestion window is modified according to standard TCP.

3.1.3 Explanation of our proposal using an example

Let us explain this proposal using simple scenario as shown in figure 3.1 A. Let S, 1, 2, 3, 4, n, D be the mobile nodes. In which nodes S and D denotes source and Destination respectively of a FTP connection. Nodes 1, 2, 3, 4 and n are the intermediate nodes. Using one of the routing protocols let us assume that the route has been established from source S to destination D through the intermediate nodes 1, 2, 3, 4 as shown in figure 3.1 A. As each and every mobile node can move at any time. Let us assume that node 4 has moved out of the transmission range of node 3. Using MAC layer of node 3, when it is unable to receive any CTS after sending consecutive RTS then it comes to know that the link to that next hop is broken as shown in figure 3.1 B. Then network layer of the node tries to make local repair, if it can not find alternate route then it records that event and sends a route error message to the source using intermediate routes. All intermediate nodes record that event and stop forwading data through that link. At the same time we want to trigger a cross layer trigger *freezeTCP* at the source node from the network layer to transport layer. A global list is maintained to store the references of all the active TCP connections. Using this global list we want to trigger cross-layer triggers. TCP changes the state of that particular TCP connection from *established* to *freeze* state as shown in figure 3.3 which describes the state diagram of our proposal. In this *freeze* state we will stop all the data transmission and cancel all the timers. At the same time we want to maintain a buffer to store all the packets which are in transmission but not acknowledged.

At the same time source node tries to establish an alternate route to the destination. If it is successful in establishing an alternate route to destination, as shown in figure 3.1 C, then source nodes network layer sends another cross-layer trigger *unFreezeTCP* to transport layer. It once again uses the global list to find the corresponding flow using source and destination addresses. It then makes that flow come out of the freeze state if it is already freezed. It now starts transmission of data (including buffered packets) using that flow and reschedules all its timers.

As explained earlier the state transition diagram of TCP is modified to include a new state which we name it as **Freeze**. When a TCP is in established state on receiving *freezeTCP* trigger it moves into **Freeze** state from the **Established** state. On receiving *unFreezeTCP* from network layer, it moves from **Freeze** state to **Established** state as shown in figure 3.3.

3.2 Performance analysis metric of proposed scheme

We want to use **throughput** as a metric to compare our results of proposed scheme. Here we calculate throughput as the amount of data that is delivered, using all the FTP flows to there corresponding destinations. We use bits per second as the unit to measure throughput.

We want to calculate the throughput in two different cases one is for all the flows in a simulation using our standard TCP and network layer routing protocol and the other is to calculate the throughput using our modified TCP and routing layer protocol to implement our proposal. At the same time we want to analyse the congestion window sizes for all the flows.

In this chapter we present details of our new proposal to improve TCP performance in Mobile Ad Hoc Networks. In next chapter we will give implementation details of DSR and TCP in *ns-2*. Followed by details of implementation of our proposed scheme in *ns-2*.

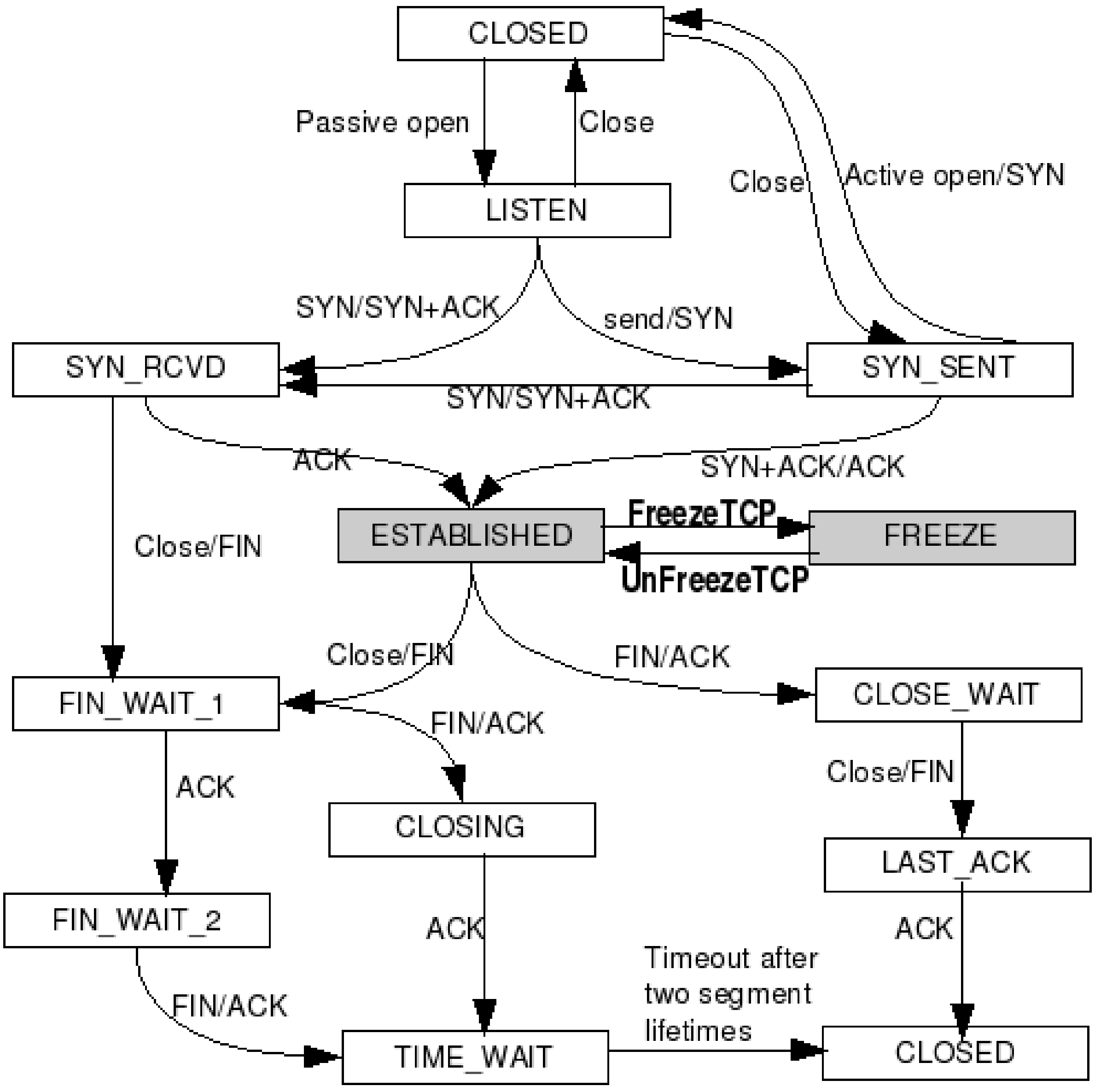


Figure 3.3: State-machine of new proposal

Chapter 4

Implementation and simulation results

4.1 Overview of *ns-2* code

This chapter goes like this: we first give the overview of the *ns-2* interface with the interpretor which acts as frontend. Explain the functionality of DSR and TCP code. Then we take one example tcl script and explain the functionality of TCP agent, FTP application and DSR agent in *ns-2* which is useful for our proposal. Then we give our design document. Followed by the details of the modifications performed to implement our proposal and simulation results.

Overview of *ns-2* interface *ns-2* is an object oriented simulator which is written in C++, with an OTcl interpreter as a frontend. *ns-2* use C++ (a systems programming language) which effectively manipulate bytes, packet headers, and implement algorithms that run over large data sets. Here run-time speed is important. Network research largely involves slightly varying parameters or quickly exploring number of scenarios. Here as *ns-2* change parameters only once run-time is less important. Here turn-around time which involves run simulation, find bug, fix bug, recompile, re-run is important so we use OTcl.

The code to interface with the interpreter resides in a separate directory, tclcl. Some of its classes used in *ns2* are given below.

- **Tcl** contains methods that C++ code will use to access the interpreter.

- **TclObject** is a base class for all the simulator objects that are in interpreted hierarchy and also mirrored in the compiled hierarchy.
- **TclClass** defines the interpreted class hierarchy and the methods to permit the users to instantiate TclObjects.
- **Command** is used to load global interpretor commands.
- **InstVar** contains methods to access C++ member variables as OTcl instance variables.

4.2 Inheritance diagrams

Following are the inheritance diagrams of TcpAgent, DSRAgent and Application which are shown in the figure 4.1 and figure 4.2

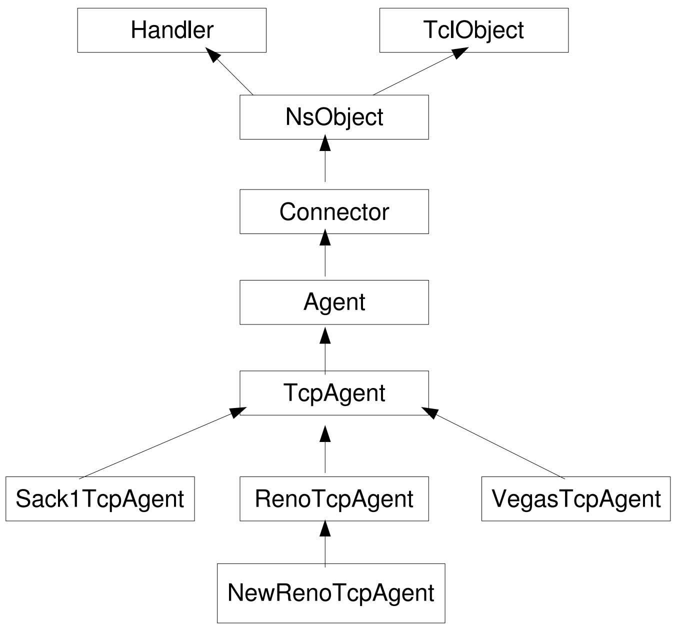
4.3 Explanation of *ns-2* functionality of TCP Agent and FTP Application

We start with an overview of class Agent and class Application. **Agents** represents end points where network-layer packets are constructed or consumed. It has the following member functions:

- **allocpkt**: Allocates new packets and assigns its fields.
- **allocpkt(int)**: Allocate new packet with a data payload of n bytes and assign its fields.
- **recv**(Packet*, Handler*): Receiving agent main receive path.

Here first two member functions are not overridden but receive and timeout (other member function) member functions are overridden according to the subclass functionality.

Application sit on top of transport agents in *ns-2*. Although objects of class Application are not meant to be instantiated, we do not make it an abstract base class so that it is visible from OTcl level. The class provides basic prototypes for application behavior (send(), recv(), resume(), start(), stop()), a pointer to the transport agent to which it is connected, and flags that indicate whether a OTcl-level upcall should be made for recv() and resume() events. Here we use *send(nbytes)* as an API between application and transport protocol as we use sockets in real world.



Inheritance diagram of TCP Agent versions

Figure 4.1: Inheritance Diagram of TCP Agent versions

We will briefly explain the functionality of the code in ns-2 as follows: When ever we start FTP then it calls **send(nbytes)** API with nbytes =-1. Here argument -1 corresponds to an infinite(in *ns-2* it replaced by a large number, 2 $\hat{3}0$) send; that is, the TCP agent will act as if its send buffer is continually filled by the application. This send method internally calls **send_much**, this function attempts to send as many packets as the current sent window allows. Which invokes **output** function that sends one packet with the given sequence number and updates the maximum sent sequence number. It also stores the time stamps and dynamically increases time stamp array if required. **output** function invokes connector::**send(packet, handler)** which calls **recv** method of the target.

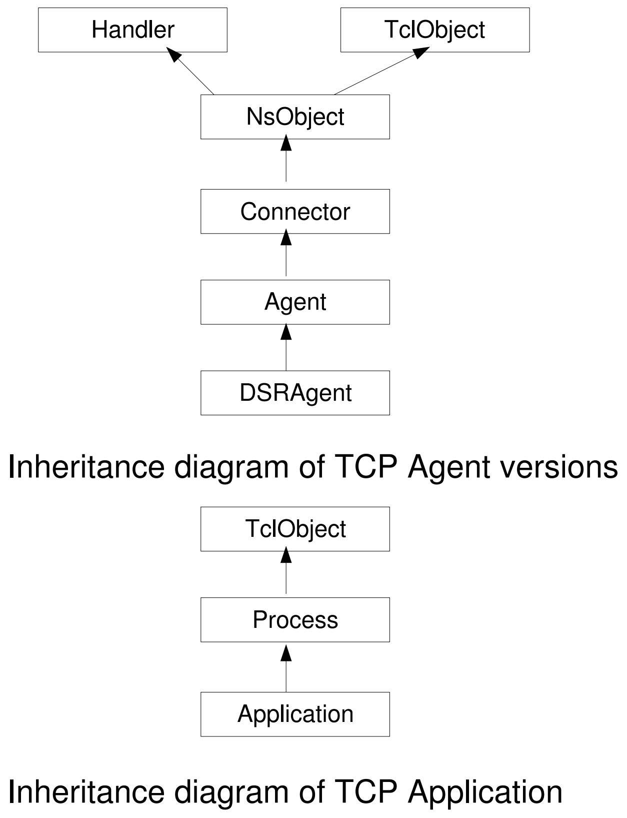

Handler
TclObject
NsObject
Connector
Agent
DSRAgent
Inheritance diagram of TCP Agent versions
TclObject
Process
Application
Inheritance diagram of TCP Application

Figure 4.2: Inheritance Diagram of DSR Agent versions

Explanation using an example Let us start with the explanation of part of tcl scripts which we use for attaching simulated application traffic to TCP agent.

1. set tcp [new Agent/TCP] ; # create sender agent
2. set sink [new Agent/TCPSink] ; # create receiver agent
3. $ns attach-agent $n0 $tcp ; # put sender on node $n0
4. $ns attach-agent $n3 $sink ; # put receiver on node $n3
5. $ns connect $tcp $sink ; # establish TCP connection
6. set ftp [new Application/FTP] ; # create an FTP source "application"

7. $ftp attach-agent $tcp ; # associate FTP with the TCP sender

8. $ns at 1.2 "$ftp start" ; #arrange for FTP to start at time 1.2 sec

The first five lines of code illustrates that agents are first attached to a node via attach-agent. Next, connect procedure sets each agents destination target to the other agent. In *ns-2*, connect() simply establishes the destination address for an agent, but does not set up the connection. First call to send() will trigger the SYN exchange.

After applications are instantiated, they must be connected to a transport agent. The attach-agent method can be used to attach an application to an agent It sets the agent_ pointer in class Application to point to the transport agent, and then it calls attachApp() in agent.cc to set the app_ pointer to point back to the application. The ftp application is started at time 1.2.

Flow of code in *ns-2*

1. set ftp [new Application/FTP]

2. $ftp attach-agent $tcp

3. $ns at 1.2 "$ftp start"

After we attach ftp to transport agent The application is started at the time provided as argument here it is 1.2. It invokes start method which is instantaneous procedure which calls send function with -1 as argument to send function.

```
1) Application/FTP  instproc start {} {
2) [$self  agent] send -1; # Send indefinitely
3) }
```

Here argument -1, this corresponds to an infinite send; that is, the TCP agent will act as if its send buffer is continually filled by the application. which then invokes send(int nbytes) and then sendmsg which is in TCpAgent class, which calls send_much. Generally the sending TCP never actually sends data (it only sets the packet size). send_much(force, reason, maxburst), this function attempts to send as many packets as the current sent window allows. It also keeps track of how many packets it has sent, and limits to the total to maxburst. It calls output function for each packet. The function output(seqno, reason) sends one packet with the given sequence number and updates the maximum sent sequence number variable to hold the given sequence number if it is the greatest sent so far. This function also assigns the various fields in the TCP header (sequence number, timestamp, reason for transmission). This function also sets a retransmission timer if one is not already pending.

Reducing congestion window Now let us look at the reasons for reducing congestion window. They are categorized into three namely:

1. CWND_ACTION_DUPACK which is caused because of duplicate acks or fast retransmit.
2. CWND_ACTION_TIMEOUT which is caused when retransmission timeout occurs.
3. CWND_ACTION_ECN which is caused when ECN bit set, CWND_ACTION_EXITED which is caused when congestion recovery has ended.

When ever we want to reduce our congestion window then *slowdown(how)* function is called. The argument how tells how to change *cwnd* and *ssthresh*. Following are the different ways *ns-2* change cwnd and ssthresh. that is *how*:

- CLOSE_SSTHRESH_HALF
- CLOSE_CWND_HALF
- CLOSE_CWND_RESTART
- CLOSE_CWND_INIT
- CLOSE_CWND_ONE
- CLOSE_SSTHRESH_HALF
- CLOSE_CWND_HALVE
- CLOSE_QUARTER_SSTHRESH
- CLOSE_CWND_HALF_WAY
- CLOSE_CWND_WITH_MIN
- TCP_IDLE
- NO_OUTSTANDING_DATA

4.4 Explanation of *ns-2* functionality of DSR Agent

When a packet first arrives at recv(packet, handler), if a packet has no source route then it may be broadcast packet. If so it is checked for incoming or outgoing broadcast packet. Then *ns-2* will handle that packet accordingly. If the packet is not a broadcast packet then it must have been an outgoing packet and a source route is already present.

When a packet first arrives at recv(Packet, Handler), if the packet did have a source route then packets destination address is checked against node's net_id and the broadcast address. If it matches with either of these addresses then it is send to handlePacketReceipt(p) for further processing. If packet is of type route_reply then it invokes acceptRouteReply(p) function. If a packet is of type route_error then it invokes processBrokenRouteError(p).

If the destination does not match either address then the packet must be one of these types namely: route request, route error, a packet to be forwarded or an invalid packet. It is processed accordingly.

4.5 Design Document

Introduction Transmission Control Protocol (TCP) has been carefully designed for reliable wired networks. Whenever there is packet loss, TCP treats it as congestion. In MANETs, packet losses occur due to less reliable wireless medium of communication as well as mobility of the nodes that leads to loss of routes. Thus, TCP performs poorly in terms of the throughput achieved. This is a design document to use explicit notifications. Which informs all the TCP flows to freeze its data transmission which are using that particular route which is failed until it gets an event when it can resume its normal data transmission.

Brief explanation of the scheme A global list is maintained to store the references of the TcpAgent objects whenever they are created. We want to use this as cross-Layer information. During a session, a node on the path to the destination may move away, resulting in a route break. The node experiencing forwarding problem has to detect the situation and try to repair the path. In the process of route repair, it sends route error packet to the source of that flow. When the routing protocol of the source receives that route error then using global list we will send a cross layer trigger to the TcpAgent which we name it as freezeTcp. On receiving freezeTcp it goes into freeze state, where it will not forward data, cancels all the timers and sets a flag named freezeFlag. When route reestablished to the destination then routing protocol of the source sends another trigger named unFreezeTCP which changes TCP state from freeze to normal. In this document we only give the modifications added but not existing functionality.

class TcpAgent

This class have the complete functionality of TCP Agent. Here we add some of the new interfaces namely unFreezeTCP, freezeTcp, cancel_timers and set_timers. Here we have a flag to store whether TCP object is in freeze state or not.

4.5.1 Protected Variable

int freezeFlag;

4.5.2 Public Functions

4.5.2.1 set_timers

```
TcpAgent:: set_timers()
Description: This function will reschedule all the timers of the TcpAgent objec
Inputs:
Outputs:   reschedules all timers.
Return Value: None
Pseudo Code:
        rtx_timer_.resched(rtt_timeout());
        burstsnd_timer_.resched(rtt_timeout());
        delsnd_timer_.resched(rtt_timeout());
```

4.5.2.2 cancel_timers

```
void cancel_timers(
Description: .This function will reschedule all the timers of the TcpAgent obje
Inputs: None
Outputs: cancels all the timers.
```

Return Value: None

Pseudo Code:

```
    rtx_timer_.force_cancel();
            burstsnd_timer_.force_cancel();
            delsnd_timer_.force_cancel();
```

4.5.2.3 freezeTcp

void freezeTcp(TcpAgent *cur)

Description: This function cancels all the existing timers, sets freezeFlag and

Inputs: pointer to TcpAgent object

Outputs: moves TCP to freeze state

Return Value: None

Pseudo Code:

```
  if(cur->freezeFlag==0){
cur->freezeFlag=1;
                cur->cancel_timers();
}
```

4.5.2.4 unFreezeTcp

TcpAgent::unFreezeTcp(TcpAgent *cur)

Description: This function reschedules all the existing timers, unsets freezeFl

Inputs: pointer to TcpAgent object

Outputs: moves TCP back to established state from freeze state

Return Value: None

Pseudo Code:

```
if(cur->freezeFlag==1){
             cur->freezeFlag=0;
                cur->set_timers();
}
```

4.5.2.5 TcpAgent

TcpAgent::TcpAgent()

Description: This is a constructor of this class. Here we append the reference

Inputs: None

```
Outputs: Adds an element to the tcp_list
Return Value: None
Pseudo Code:

  tcp_list[i++]=this;
```

class DSRAgent

This class contains the complete functionality of the DSR routing protocol. Here we modify the functionality of acceptRouteReply, processBrokenRouteEroor, slow-down and replyFromRouteCache functionality. Here we present only modifications and retains all other functionality.

4.5.3 Public Functions

4.5.3.1 processBrokenRouteError

```
DSRAgent::processBrokenRouteError(SRPacket& p)
Description: This function compares the source and destination address of the r
 Inputs: packet
Outputs: invokes TCP API freezeTCP
Return Value:  None
Pseudo Code:
 class TcpAgent *t;
        for(int j=0;j<i;j++) {
                t=tcp_list[j];
                if(t->addr()==(int)p.src.getNSAddr_t()&&t->daddr()==(int)p.dest
                        t->freezeTcp(t);
                }
}
```

4.5.3.2 acceptRouteReply

```
void DSRAgent::acceptRouteReply(SRPacket &p)
Description: This function compares the source and destination address of the r
Inputs: packet
Outputs: invokes TCP API unFreezeTCP.
Return Value:  None
```

```
Pseudo Code:
 for(int j=0;j<i;j++)  {
                t=tcp_list[j];
                if(t->addr()==(int)reply_route[0].addr&&t->daddr()==(int)reply_
                        t->unFreezeTcp(t);
                }
}
```

4.5.3.3 replyFromRouteCache

```
DSRAgent::replyFromRouteCache(SRPacket &p)
Description: This function compares the source and destination address of the r
Inputs: packet
Outputs: invokes unFreezeTCP API of TCP
Return Value: bool
Pseudo Code:
  class TcpAgent *t;
         for(int j=0;j<i;j++) {
                t=tcp_list[j];
                if(t->addr()==(int)complete_route[0].addr&&t->daddr()==(int)comp
                        t->unFreezeTcp(t);
                }
}
```

4.5.3.4 slowdown

```
void slowdown(int how);
Description: This function is used to reduce congestion window. We modify the f
Inputs: None
Outputs: None
Return Value: None
Pseudo Code:
if(freezeFlag!=1)
normal operatio
```

4.5.3.5 recv

```
DSRAgent::recv(Packet* packet, Handler*)
```

```
Description: Modify recv functionality such that whenever we are going to invok
Inputs: packet reference and handler reference
Outputs: None
Return Value: void
Pseudo Code:
if(freezeFlag!=1)
send_much()
```

4.6 Explanation of modifications required to implementation of new proposal

We want to maintain a global list which we want to name as tcpList of type TcpAgent * to store the references of the Tcp objects. whenever we attach a TCP agent to a node then it creates a new object of TcpAgent. So we want to insert an element into the tcpList whenever we call the constructor of TcpAgent. We want to use this global list as a cross-layer information from transport layer to network layer.

Modifications required in DSR code In DSR code whenever we receive a packet of type route error then it calls processBrokenRouteError. Here we want to compare the source and destination addresses of the route with that of the source and destination addresses of all the elements of the tcpList. If any one of these elements matches then we want to trigger the freezeTcp API which is implemented in TCPAgent class using the reference of the TCP agent which is present in the tcpList.

In DSR code whenever we receive a packet of type route reply (route_reply()), which contains the route piggybacked into the route request then it calls acceptRouteReply. We want to compare the source and destination addresses of the route with that of the source and destination addresses of all the elements of the tcpList. If any one of these elements matches then we want to trigger the unFreezeTcp API which is implemented in TCPAgent class using the reference of the TCP agent which is present in the tcpList. Alternately when a route to destination is found in cache then also we want to trigger unFreezeTcp API.

Modifications done in the code of TCPAgent class We want to declare a new protected variable named freezeFlag of type int. We need to initialize its value to 0. We included two new functions which are named as set_timers() and cancel_timers.

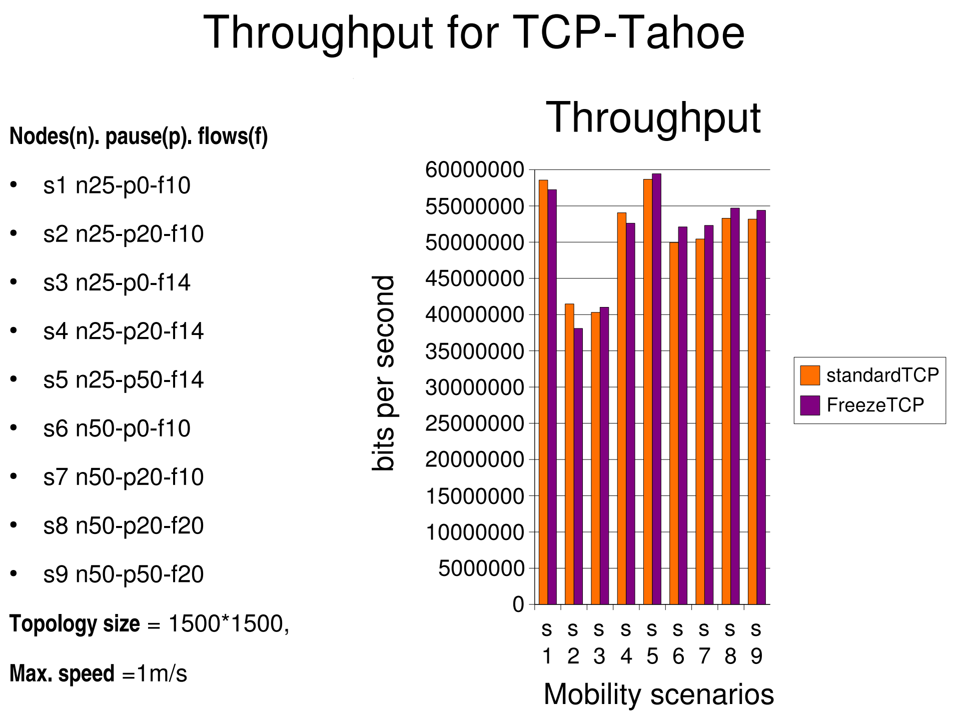
Table 4.1: Mobility scenarios used in simulations

S.No.	*Topology*	*Nodes*	*Pause time*	*max. speed*	*FTP Flows*	*link changes*	*route changes*	*destination unreach-able*
s1	***1500*300***	25	0	1	10	236	1090	24
s2	***1500*300***	25	20	1	10	219	1826	0
s3	***1500*300***	25	0	1	14	236	1090	24
s4	***1500*300***	25	20	1	14	219	1826	0
s5	***1500*300***	25	50	1	14	195	1713	0
s6	***1500*300***	50	0	1	10	798	5279	0
s7	***1500*300***	50	20	1	10	990	4863	0
s8	***1500*300***	50	20	1	20	990	4863	0
s9	***1500*300***	50	50	1	10	788	4863	0

In set_timers we reschedule all the timers used by this class. In cancel_timer we cancel all the timers which are active. We introduced two new APIs which we named as freezeTcp and unFreezeTcp. In freezeTcp we want to cancel all the timers and set the variable freezeFlag to 1. In unfreezeTcp we restore freezeFlag to 0 and reschedule all the timers. When ever there is a need to reduce congestion window for various reasons then TcpAgent calls a function slowdown. We want to modify or reduce the congestion window only if freezeFlag is unset. By this we want to ensure that whenever there is a route failure due to mobility it will not alter its congestion window size. Application FTP uses send(nbytes) as an API to generate traffic. It sends argument as -1 which ensures that it wants to generate data continuously.

4.7 Simulation results

We implemented our proposal in *ns-2* and performed simulations for various mobility patterns and with different number of FTP flows. In table4.1 we present details of the parameters we used for our simulations. We used 1500*300 topology size, packet size of 512, maximum speed of a mobile node is 1 m/s. We varied pause time(0, 20, 50 seconds), number of nodes(25, 50) and number of FTP flows(10, 14, 20) as shown in the table 4.1. We used default values of *ns-2* for all other parameters. We made simulations using three different TCP flavors namely Tahoe, Reno and New Reno. We repeated simulations using existing standard protocols and with our new proposal. We used throughput as a metric to compare our results with that of standard proto-



Nodes(n). pause(p). flows(f)

- s1 n25-p0-f10
- s2 n25-p20-f10
- s3 n25-p0-f14
- s4 n25-p20-f14
- s5 n25-p50-f14
- s6 n50-p0-f10
- s7 n50-p20-f10
- s8 n50-p20-f20
- s9 n50-p50-f20

Topology size = 1500*1500,

Max. speed =1m/s

Figure 4.3: Throughput comparison for TCP-Tahoe

cols. Figure 4.3 shows the comparison result of throughput for TCPTahoe. Similarly figure 4.4 and figure 4.5 shows the comparison results of throughput for TCPReno and New Reno flavors of TCP. We also collected statistics of congestion window sizes for all the FTP flows in the above simulations and the comparison of congestion windows is done. From these simulation results it is clear that our proposal improves TCP performance in most of the cases when compared to that of standard protocols even when mobility of nodes and traffic is more.

Following are some of the observations we made using our simulation results. When number of link failures and route failures increases then our proposal gives better results than that of the standard protocols irrespective of number of flows. When number of link failures and route failures are relatively less then Reno and New Reno performs well when we have less number of FTP flows.

In this chapter we gave implementation details of *ns-2* and our proposal. We also presented simulation results of our proposal. We present the concluding remarks in the last chapter.

Throughput for TCP-Reno

Nodes(n). pause(p). flows(f)

- s1 n25-p0-f10
- s2 n25-p20-f10
- s3 n25-p0-f14
- s4 n25-p20-f14
- s5 n25-p50-f14
- s6 n50-p0-f10
- s7 n50-p20-f10
- s8 n50-p20-f20
- s9 n50-p50-f20

Topology size = 1500*1500,

Max. speed =1m/s

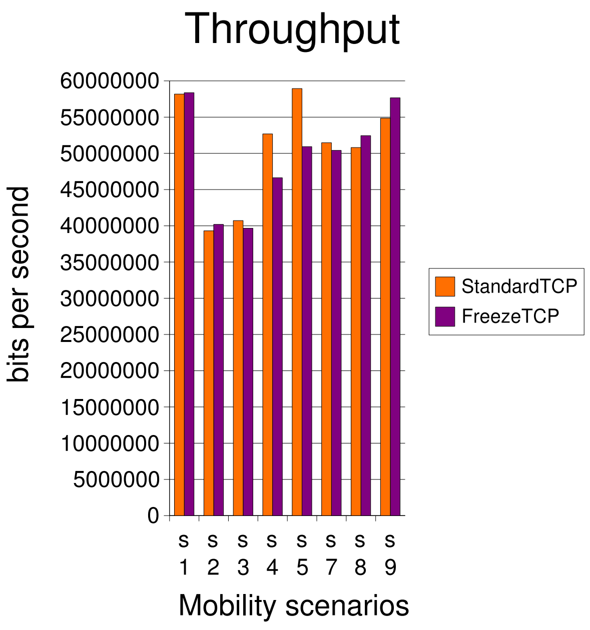


Figure 4.4: Throughput comparison for TCP-Reno

Throughput for TCP-New Reno

Nodes(n). pause(p). flows(f)

- s1 n25-p0-f10
- s2 n25-p20-f10
- s3 n25-p0-f14
- s4 n25-p20-f14
- s5 n25-p50-f14
- s6 n50-p0-f10
- s7 n50-p20-f10
- s8 n50-p20-f20
- s9 n50-p50-f20

Topology size = 1500*1500,

Max. speed =1m/s

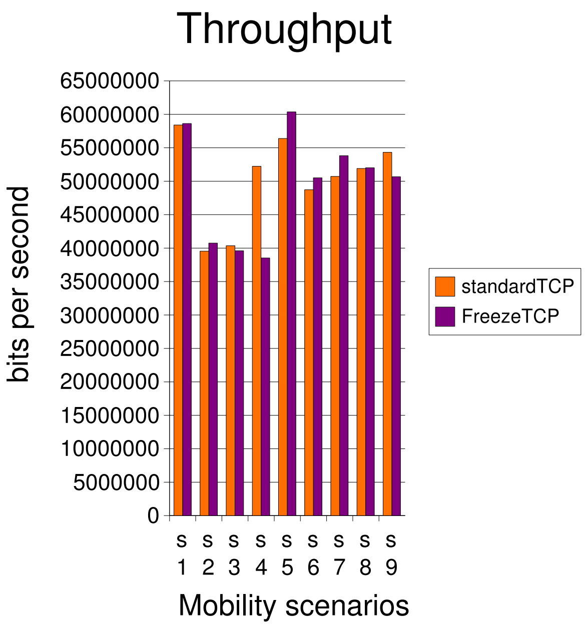


Figure 4.5: Throughput comparison for TCP-New Reno

Chapter 5

Concluding remarks

We have proposed a solution which appears to be a lighter loading solution as compared to other approaches since it does not add new probe packets into the network. We implemented our proposal in *ns-2* simulator. We made simulations using standard protocol and by using our proposal for various mobility scenarios, which include modifying number of nodes, pause time, speed of the nodes, scenario topology size and by varying number of FTP flows. We repeated simulations for three TCP flavors namely Tahoe, Reno and New Reno. In most of the cases our implemented proposal give slightly better results than that of the standard proposal. We also made a study on congestion window sizes of our simulations.

We can also extend this proposal to another routing protocol like AODV. We can also make triggering of route error and route reestablishment events independent of routing protocols. As we are acting reactively after the route failures occur, we can modify our proposal so that proactively we can trigger route errors using signal strength.

Appendix A

Simulation results

A.1 Throughput comparison with and with-out Mobility

Here we present the simulation results of throughput for different flavors of TCP namely: Tahoe, Reno, NewReno and Veno. Using different routing protocols namely: DSR, DSDV and AODV. Using 10 and 20 FTP flows We made simulations using three different mobility patterns. We present our mobility pattern in the following format: (scen topology(scen), nodes(n), max speed(s), pause time(p), Link changes(l), Route changes(r)). The following are the three different mobility scenarios used:

- scen-670*670-n50-s20-p600-l953-r2334 as scenario1
- scen-670*670-n50-s20-p600-l1041-r2877 as scenario3
- scen-670*670-n50-s20-p600-l1048-r3339 as scenario2

We made use of throughput as a metric, which is defined as number of bits received at the destination(s) per second during a simulation. In these simulations we want to compare throughput with or without mobility for different mobility patterns, routing protocols and different FTP flows. We can observe that using mobility scenarios 1 and 3 we can say in general that the throughput with out mobility is less than that of with mobility. We can say that this might be because of hidden and exposed terminal problems, which comes into effect when nodes are static. But where as using mobility scenario 2 we can see that the throughput without mobility is larger than that of with mobility. Here as we have more number of route changes when compared to that of scenarios 1 and 3, this might be the reason for decline in the throughput with mobility.

DSDV Mobility Scenario1

- DSDV
- T10=Tahoe10 flows
  NR10=Newreno10flows
  R10=Reno10flows
  V10=Veno10flows
  T20=Tahoe20 flows
  NR20=Newreno20flows
  R20=Reno20flows
  V20=Veno20flows
- scen-670*670-n50-s20-p600,l1041,r2877

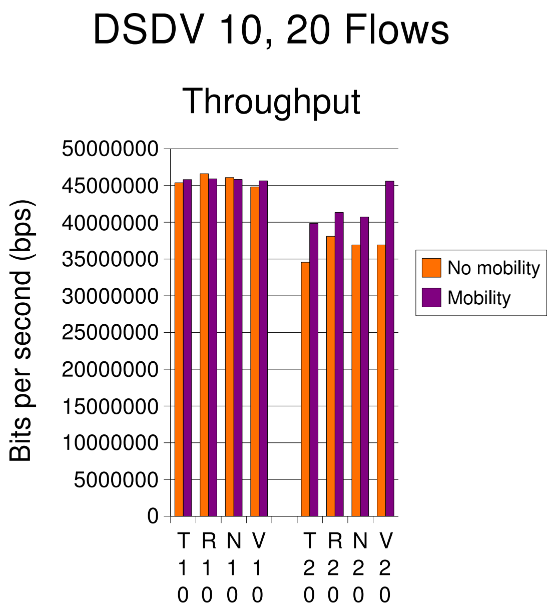
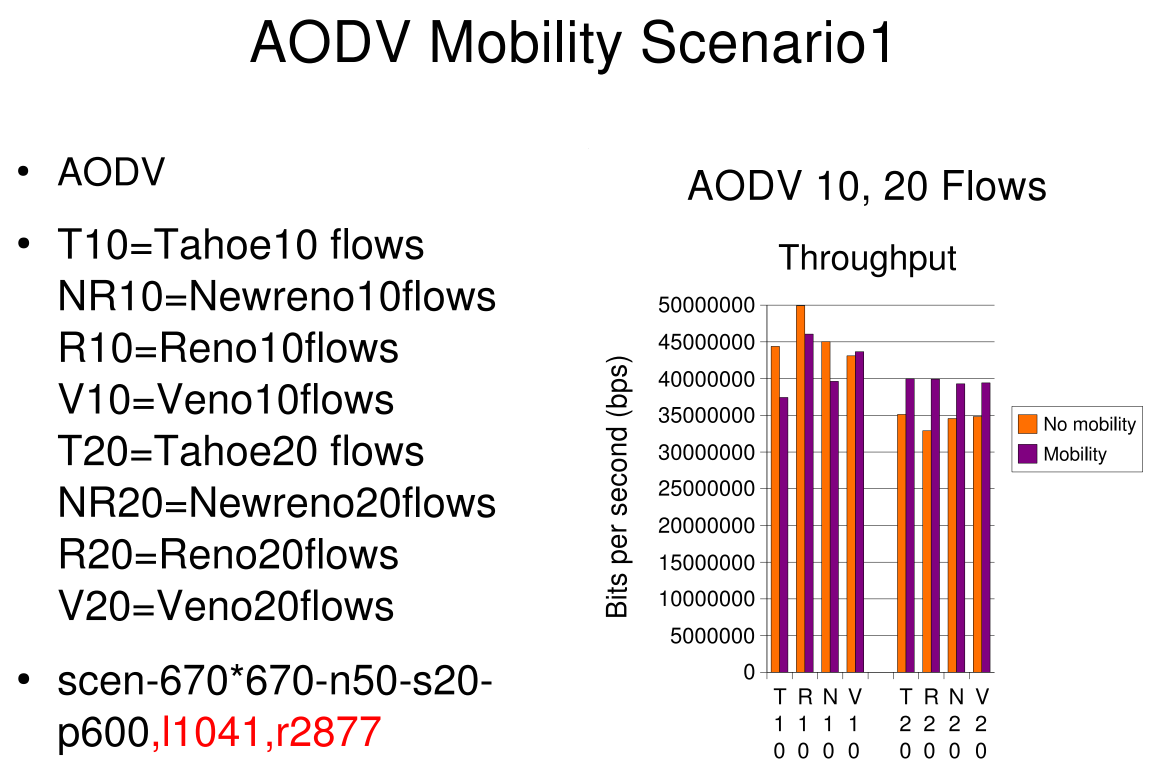


Figure A.1: Thoughput plot for scenario 1 using DSDV

Table A.1: Thoughput (bits) for scenario 1 using DSDV

S.No.	***TCP Flavors***	***No-Mobility-10Flows***	***Mobility-10Flows***	***No-Mobility-20Flows***	***Mobility-20Flows***
1	***Tahoe***	45380260	45807440	34559640	41341700
2	***Reno***	46608860	45908200	41341700	38091620
3	***NewReno***	45828700	46085220	36914840	40716000
4	***Veno***	44806980	45635840	36914840	45597420



AODV Mobility Scenario1

- AODV
- T10=Tahoe10 flows
  NR10=Newreno10flows
  R10=Reno10flows
  V10=Veno10flows
  T20=Tahoe20 flows
  NR20=Newreno20flows
  R20=Reno20flows
  V20=Veno20flows
- scen-670*670-n50-s20-p600,l1041,r2877

Figure A.2: Thoughput plot for scenario 1 using AODV

Table A.2: Thoughput (bps) for scenario 1 using AODV

S.No.	***TCP Flavors***	***No-Mobility-10Flows***	***Mobility-10Flows***	***No-Mobility-20Flows***	***Mobility-20Flows***
1	***Tahoe***	44378680	37430320	3511940	39971740
2	***Reno***	49916000	46057600	32899600	39922100
3	***NewReno***	45033700	39621340	34557440	39287160
4	***Veno***	43102380	43665240	34829220	39416900

DSR Mobility Scenario1

- DSR
- T10=Tahoe10 flows
  NR10=Newreno10flows
  R10=Reno10flows
  V10=Veno10flows
  T20=Tahoe20 flows
  NR20=Newreno20flows
  R20=Reno20flows
  V20=Veno20flows
- scen-670*670-n50-s20-p600,l1041,r2877

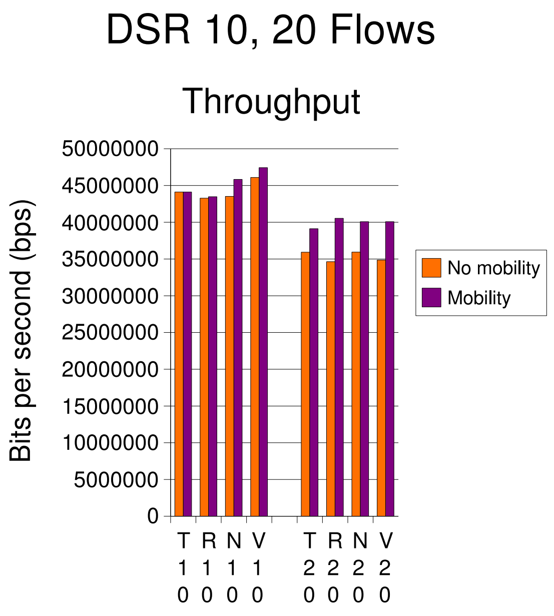


Figure A.3: Thoughput plot for scenario 1 using DSR

Table A.3: Thoughput (bps) for scenario 1 using DSR

S.No.	***TCP Flavors***	***No-Mobility-10Flows***	***Mobility-10Flows***	***No-Mobility-20Flows***	***Mobility-20Flows***
1	***Tahoe***	44114508	46092104	35931084	39127784
2	***Reno***	43280280	43478368	34693128	40540472
3	***NewReno***	43520088	45833576	35940916	40081836
4	***Veno***	46094244	47432916	34854340	41224724

DSDV Mobility Scenario2

- DSDV
- T10=Tahoe10 flows
  NR10=Newreno10flows
  R10=Reno10flows
  V10=Veno10flows
  T20=Tahoe20 flows
  NR20=Newreno20flows
  R20=Reno20flows
  V20=Veno20flows
- scen-670*670-n50-s20-p600,l1048,r3339

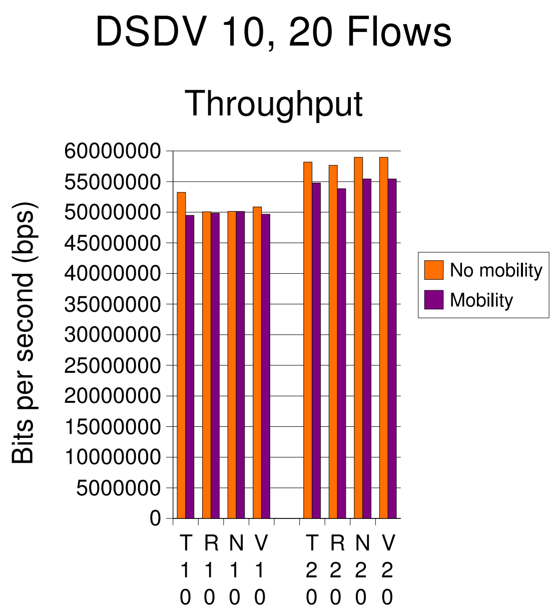


Figure A.4: Thoughput plot for scenario 2 using DSDV

Table A.4: Thoughput (bps) for scenario 2 using DSDV

S.No.	***TCP Flavors***	***No-Mobility-10Flows***	***Mobility-10Flows***	***No-Mobility-20Flows***	***Mobility-20Flows***
1	***Tahoe***	53248640	49499420	58185260	54778360
2	***Reno***	50075060	49851400	57659260	53851680
3	***NewReno***	50145020	50139720	58970660	55434440
4	***Veno***	50871060	49674320	58970660	55434440

AODV Mobility Scenario2

- AODV
- T10=Tahoe10 flows
  NR10=Newreno10flows
  R10=Reno10flows
  V10=Veno10flows
  T20=Tahoe20 flows
  NR20=Newreno20flows
  R20=Reno20flows
  V20=Veno20flows
- scen-670*670-n50-s20-p600,l1048,r3339

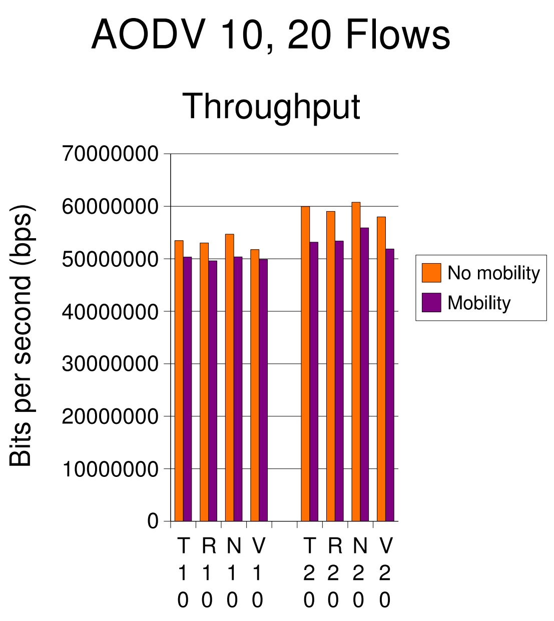
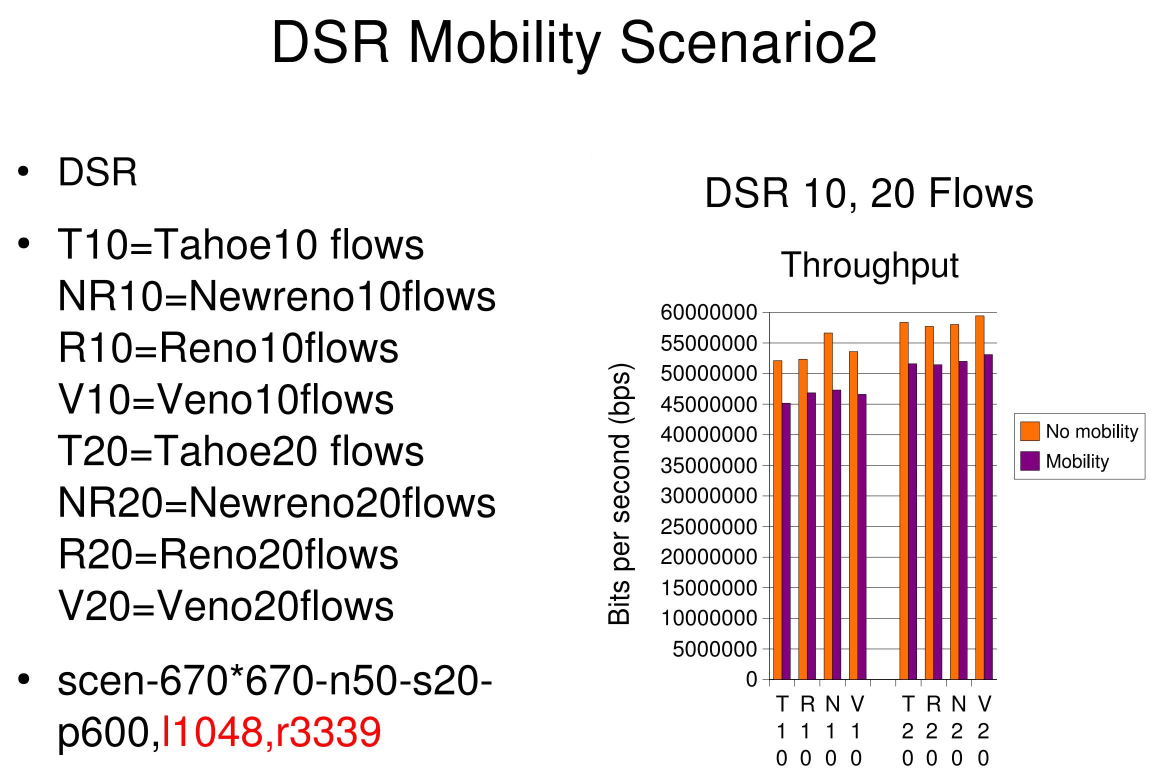


Figure A.5: Thoughput plot for scenario 2 using AODV

Table A.5: Thoughput (bps) for scenario 2 using AODV

S.No.	***TCP Flavors***	***No-Mobility-10Flows***	***Mobility-10Flows***	***No-Mobility-20Flows***	***Mobility-20Flows***
1	***Tahoe***	53467000	50324100	59957160	53159380
2	***Reno***	53015440	49607540	59026480	53378800
3	***NewReno***	54680700	50356960	60769000	55893000
4	***Veno***	51751980	49849280	57982800	51868720



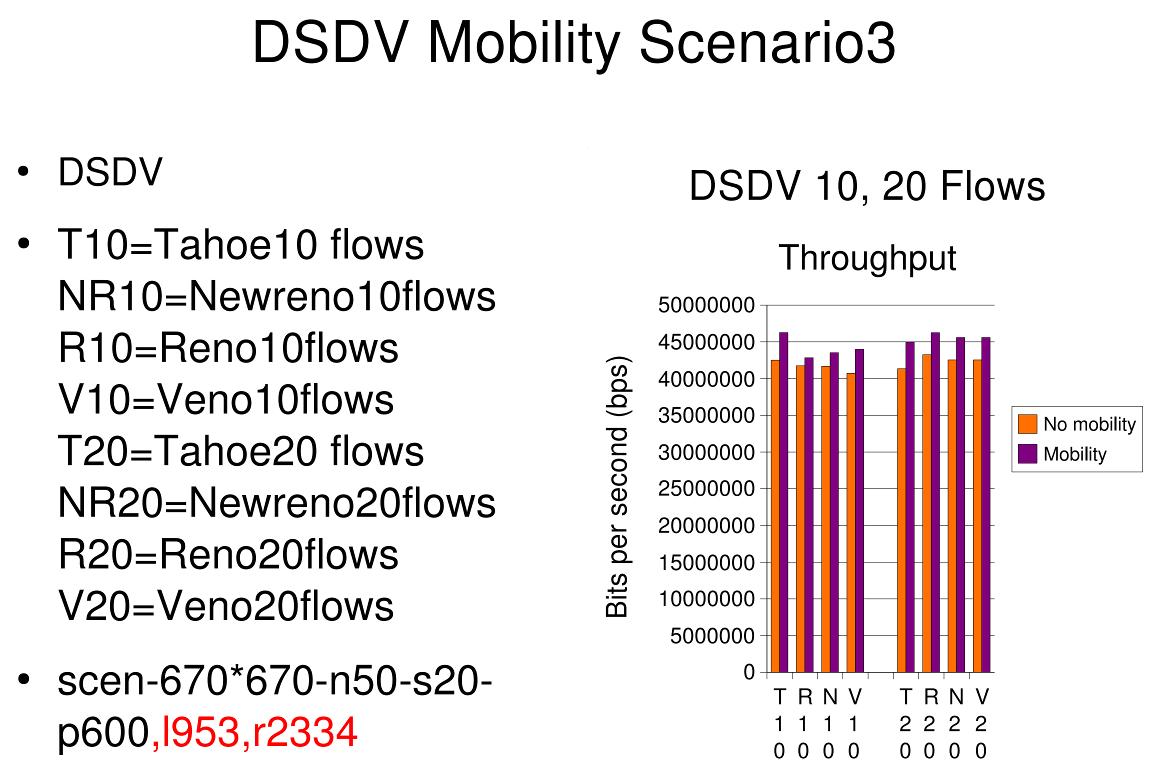
DSR Mobility Scenario2

- DSR
- T10=Tahoe10 flows
  NR10=Newreno10flows
  R10=Reno10flows
  V10=Veno10flows
  T20=Tahoe20 flows
  NR20=Newreno20flows
  R20=Reno20flows
  V20=Veno20flows
- scen-670*670-n50-s20-p600,l1048,r3339

Figure A.6: Thoughput plot for scenario 2 using DSR

Table A.6: Thoughput (bps) for scenario 2 using DSR

S.No.	*TCP Flavors*	*No-Mobility-10Flows*	*Mobility-10Flows*	*No-Mobility-20Flows*	*Mobility-20Flows*
1	***Tahoe***	527117960	45161444	58361876	51595388
2	***Reno***	52340520	46847436	57703688	51445676
3	***NewReno***	56628440	47305924	58020452	51990636
4	***Veno***	53589720	46593276	59424176	53101024



DSDV Mobility Scenario3

- DSDV
- T10=Tahoe10 flows
  NR10=Newreno10flows
  R10=Reno10flows
  V10=Veno10flows
  T20=Tahoe20 flows
  NR20=Newreno20flows
  R20=Reno20flows
  V20=Veno20flows
- scen-670*670-n50-s20-p600,l953,r2334

Figure A.7: Thoughput plot for scenario 3 using DSDV

Table A.7: Thoughput (bps) for scenario 3 using DSDV

S.No.	***TCP Flavors***	***No-Mobility-10Flows***	***Mobility-10Flows***	***No-Mobility-20Flows***	***Mobility-20Flows***
1	***Tahoe***	42509840	46266480	41342520	44933800
2	***Reno***	41745580	42839500	43256000	46255800
3	***NewReno***	41683040	43526380	42540380	45597420
4	***Veno***	40725800	43980000	42540380	45597420

AODV Mobility Scenario3

- AODV
- T10=Tahoe10 flows
  NR10=Newreno10flows
  R10=Reno10flows
  V10=Veno10flows
  T20=Tahoe20 flows
  NR20=Newreno20flows
  R20=Reno20flows
  V20=Veno20flows
- scen-670*670-n50-s20-p600,l953,r2334

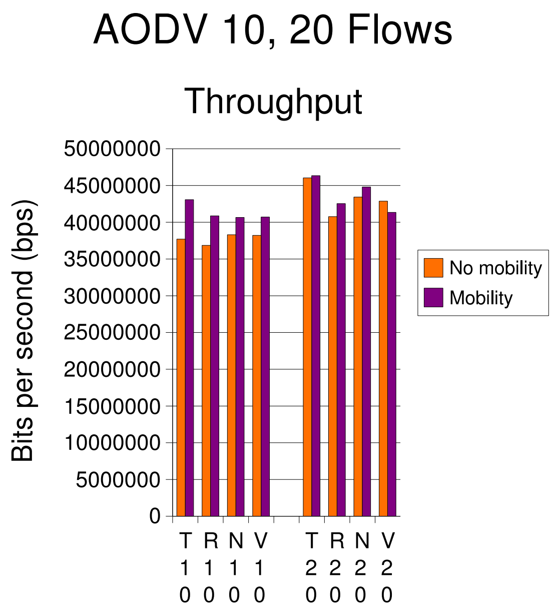
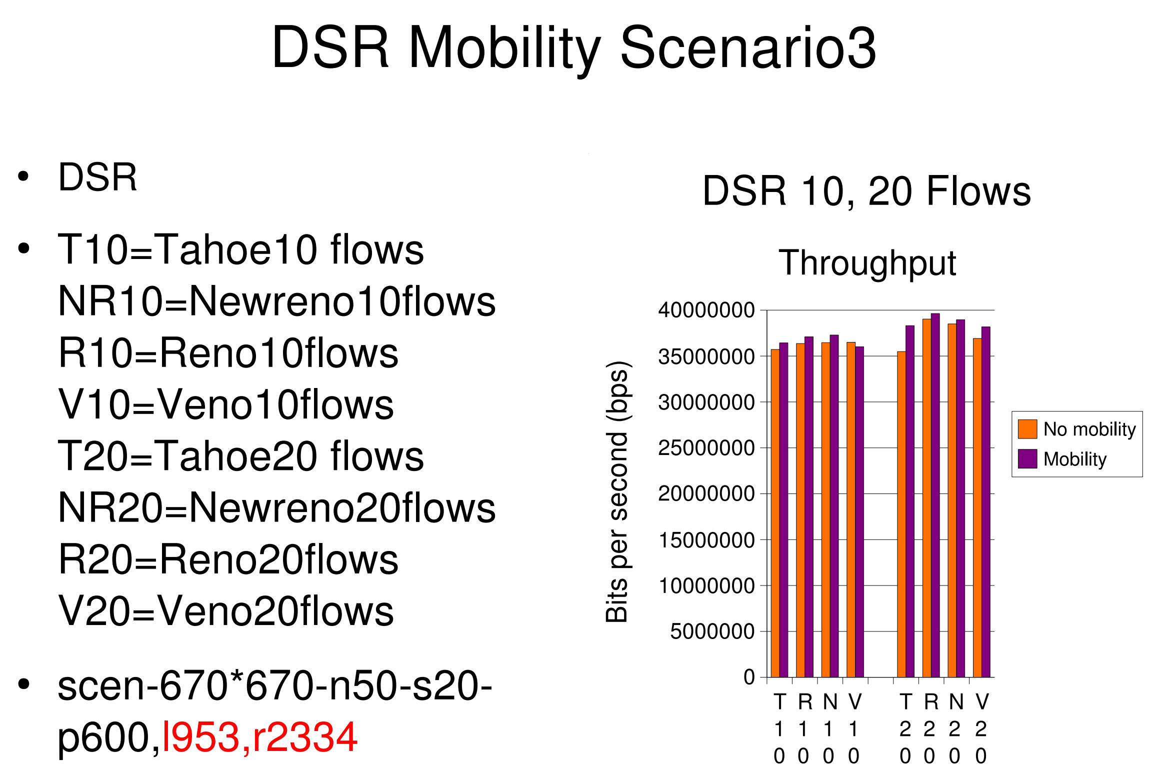


Figure A.8: Thoughput plot for scenario 3 using AODV

Table A.8: Thoughput (bps) for scenario 3 using AODV

S.No.	***TCP Flavors***	***No-Mobility-10Flows***	***Mobility-10Flows***	***No-Mobility-20Flows***	***Mobility-20Flows***
1	***Tahoe***	37709040	43074760	46038540	46343820
2	***Reno***	36859980	40873140	40766920	42537120
3	***NewReno***	38304880	40658080	43428580	44802340
4	***Veno***	38219080	40712200	42874440	41351220



DSR Mobility Scenario3

- DSR
- T10=Tahoe10 flows
  NR10=Newreno10flows
  R10=Reno10flows
  V10=Veno10flows
  T20=Tahoe20 flows
  NR20=Newreno20flows
  R20=Reno20flows
  V20=Veno20flows
- scen-670*670-n50-s20-p600,l953,r2334

Figure A.9: Thoughput plot for scenario 3 using DSR

Table A.9: Thoughput (bps) for scenario 3 using DSR

S.No.	***TCP Flavors***	***No-Mobility-10Flows***	***Mobility-10Flows***	***No-Mobility-20Flows***	***Mobility-20Flows***
1	***Tahoe***	35715492	36438652	35488976	38312396
2	***Reno***	36361308	37099584	39028892	39634308
3	***NewReno***	36461992	37291968	38503488	38959840
4	***Veno***	36497012	36016392	36916156	38181612

Bibliography

[1] E. A. Ahmad Al Hanbali and P. Nain, “A survey of tcp over ad hoc networks,” in *IEEE communications surveys, third quarter*, vol. 7, pp. 22 – 36, 2005.

[2] S. V. Kaetik chandran, sudharshan Raghunathan and R. Prakash, “A feedback-based scheme for improving tcp performance in ad hoc wireless networks,” in *IEEE Personal Communications*, vol. 8 of *1*, pp. 34 – 39, Feb 2001.

[3] G.Holland and N.Vaidya, “Analysis of tcp performance in mobile ad hoc networks,” in *ACM Wireless networks*, vol. 8, pp. 275 – 288, Mar 2002.

[4] V. Anantharaman and R. Sivakumar, “Tcp performance over mobile ad hoc networks: A quantitative study,” in *J. wireless commun. and mobile computing*, vol. 4, pp. 203 – 222, Mar 2004.

[5] K. X. et al., “Tcp unfairness in ad hoc wireless networks and a neighborhood red solution,” in *Wireless Networks*, pp. 383 – 399, Mar 2005.

[6] D. A. M. David B. Johnson and J. Broch, “Dsr the dynamic source routing protocol for multihop wireless ad hoc networks,” in *Ad Hoc Networking, Chapter 5*, no. 139 - 172, 2001.

[7] J. Liu and S. Singh, “Atcp: Tcp for mobile ad hoc networks,” in *IEEE JSAC*, vol. 19, pp. 1300 – 1315, July 2001.

[8] S. F. M. Mathis, J. Mahdavi and A. Romanow, *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms.* Request for Comments: 2001, 1999.

[9] M. Allman, V. Paxson, and W. Stevens, *TCP Congestion Control.* Request for Comments: 2581, 1999.

[10] S. Floyd, T. Henderson, and A. Gurtov, *The NewReno Modification to TCP’s Fast Recovery.* Request for Comments: 3782, 1999.

[11] S. F. M. Mathis, J. Mahdavi and A. Romanow, *TCP Selective Acknowledgment Options.* Request for Comments: 2018, 1996.

[12] J. Singh and B. Soh, "Tcp new vegas: Improving the performance of tcp vegas over high latency links," *IEEE International Symposium on Network Computing and Applications*, Sep 2005.

[13] V. srivastava and M. Motani, "Cross-layer design: A survey and the road ahead," in *IEEE Communications magazine*, pp. 112–119, Dec 2005.

[14] S. Corson and J. Macker, *Mobile Ad hoc Networking (MANET):Routing Protocol Performance Issues and Evaluation Considerations.* Network Working Group, Request for Comments: 2501, Category: Informational, Jan 1999.

[15] G. Holland and N. Vaidya, "Analysis of tcp performance in mobile ad hoc networks part ii: Simulation detils and results," tech. rep., Texas A and M University, Feb 1999.

[16] J. Monks, P. Sinha, and V. Bharghavan, "Limitations of tcp-elfn for ad hoc networks," 2000. J. P. Monks, P. Sinha and V. Bharghavan, Limitations of TCP-ELFN for Ad Hoc Networks, MOMUC 2000.