University of Alberta

Improving TCP Resilience

by

Qiang Ye ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

Edmonton, Alberta

Spring 2007

Your file Votre référence
ISBN: 978-0-494-29774-2
Our file Notre référence
ISBN: 978-0-494-29774-2

Abstract

The Transmission Control Protocol (TCP) was designed to provide a reliable end-to-end connection between two hosts. Over time, it has become the de facto transport layer protocol in the Internet. It is important to thoroughly understand the resilience behavior of TCP in the case of network congestion and failure and improve the resilience performance of TCP if possible.

We studied the detailed resilience behavior of SACK, NewReno, and Reno TCP, and recommended the proper resilience objectives for TCP transport in the Internet. Specifically, with SACK TCP, we found two restoration objectives, $\tau_{1,S}$ and $\tau_{2,S}$. With NewReno TCP, we also found two restoration objectives, $\tau_{1,NR}$ and $\tau_{2,NR}$. $\tau_{1,NR}$ is approximately twice as large as $\tau_{1,S}$ when *rwnd* is large. $\tau_{2,NR}$ is essentially the same as $\tau_{2,S}$. With Reno TCP, we found a recovery objective of 200ms for the DS0 access; for DS1 and OC-3c access, there seem to be no critical points from 15ms to 1s.

After understanding the detailed behavior of SACK TCP, we proposed two types of mechanisms, Probing and Pacing, to improve the resilience of SACK TCP. SACK TCP with Probing and Pacing were modeled with Petri nets. The Petri net models were formally analyzed to increase our confidence about the correctness of the proposed changes. Our experimental results demonstrate that SACK TCP with Probing and Pacing is much more resilient than standard SACK TCP.

To test the effectiveness of the proposed protocols, we proposed a hybrid protocol development methodology, embedding formal models into simulation, that combines the use of formal methods and simulation to obtain the advantages of deep formal verification with the broad spectrum testing of simulation. With the assistance of this hybrid methodology, we tested α-min Paced SACK TCP in a NSF-like network. The results show that α-min Paced SACK TCP performs better than SACK TCP in terms of resilience, not only in single session scenarios, but also in multiple session cases, even if these TCP sessions have varied RTTs.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor, Dr. Mike H. MacGregor, for his continuous support over the past six years. PhD takes a long time, without his guidance and encouragement, I could not come this far.

Secondly, I would like to thank the members of my candidacy examination committee: Dr. Janelle Harms, Dr. Pierre Boulanger, Dr. Ehab Elmallah, and Dr. Ivan Fair. Their thoughtful suggestions make this thesis much more valuable.

Thirdly, I am grateful to all the members of the communication network research group for their assistance and friendship. I would also like to thank all my friends in Edmonton who make my life in Edmonton such a memorable period.

Finally, special thanks to my parents and elder sisters. They make such a great family that I feel so warm whenever I think of it. Their confidence and encouragement means much to me.

Contents

List of Tables

List of Figures

List of Symbols and Abbreviations

DS0	A basic digital signaling rate of 64 kbps.
DS1	A T-carrier signaling scheme devised by Bell Labs. The bandwidth is 1.544 Mbps.
OC-3c	OC-3 is a network line with transmission speeds of up to 155.52 Mbps . When OC-3 carries the data from a single source, it is called OC-3c.
cwnd	Congestion Window.
rwnd	Receiver Window.
ssthresh	Slow Start Threshold.
ACK	Acknowledgement.
RTT	Round Trip Time.
RTO	Retransmission Timeout.
FlightSize	The amount of data that has been sent but not yet acknowledged.
SMSS	Sender Maximum Segment Size.
TTI	Transfer Time Increase.

Chapter 1

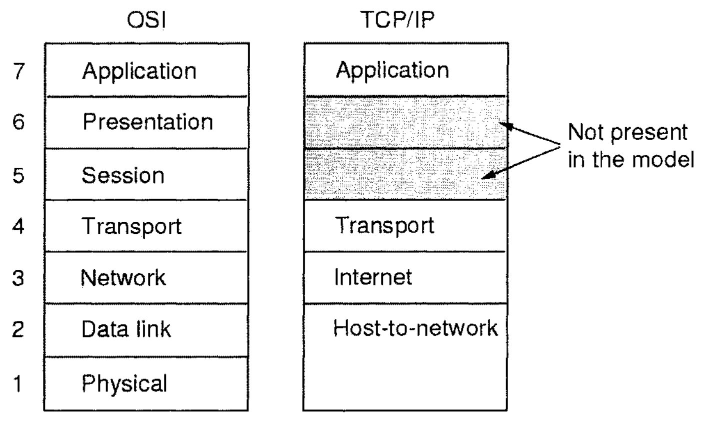
Introduction

The Transmission Control Protocol (TCP) [1, 2] was designed to provide a reliable end-to-end connection between two hosts. Over time, it has become the de facto transport layer protocol in the Internet. It is so widely deployed that even a small improvement in TCP could lead to a large change in overall network performance. Our research is focused on TCP resilience improvement. We first studied the resilience performance of the most widely used TCP versions, SACK, NewReno, and Reno [3]. Based on the resilience study, two types of mechanisms, Probing and Pacing, were proposed to improve the resilience of the most recent version, SACK TCP. To test Pacing in a relatively realistic environment, we developed a hybrid method for protocol development, "embedding formal models into simulation". In this chapter, we first give an introduction to TCP and network resilience. Then we present the contributions and outline of the thesis.

1.1 TCP: the Transport Layer Protocol in the Internet

Most networks employ a layered architecture. That is, the software in the network is structured as a stack of layers, each one built upon the one below it. In 1983, as a first step toward international standardization, International Standards Organization proposed the ISO OSI (Open Systems Interconnection) model [2], shown in Figure 1.1 [2]. The model was designed to connect systems that are open for communication with other systems. Seven layers are included in this model. From bottom up, these layers are Physical Layer, Data Link Layer, Network Layer, Transport Layer, Session Layer, Presentation Layer, and Application Layer. Nowadays, the protocols associated with the ISO OSI model are hardly used any more. However, the model is very general so that the features mentioned at each layer are still valid and important.

Another reference model, the DoD model is different than the ISO OSI model in that the DoD model is simpler and consists of only four layers. The DoD model was first employed by the predecessor of the Internet, ARPANET. As the Internet developed into the dominant worldwide network, the DoD model became more and more popular and the protocols associated with the model became the de facto protocols in practice. As shown in Figure 1.1, the DoD model has four layers in total: Host-to-Network Layer, Internet Layer, Transport Layer, and Application Layer. Internet Layer, Transport Layer, and Application Layer in the DoD model correspond to Network Layer, Transport Layer, and Application Layer in the ISO OSI model, respectively. Also, the DoD model does not have Presentation and Session Layer. In addition, Host-to-Network Layer has the same function as Data Link and Physical Layer in the ISO OSI model. There are two commonly used protocols defined in the Transport Layer of the DoD model. These are TCP and UDP (User Datagram Protocol). UDP is an unreliable, connectionless protocol for applications in which prompt delivery is more important than accurate delivery. On the contrary, TCP is a reliable connection-oriented protocol that allows packets to be transferred from one host to the other without error.



(1) ISO OSI Model (2) DoD Model

Figure 1.1 The ISO OSI Model and DoD Model [2]

TCP has evolved over the past decades. So far the main versions of TCP that are widely deployed are Tahoe TCP, Reno TCP, NewReno TCP and SACK TCP [3]. Tahoe TCP is the oldest version and it only includes the algorithms of Slow Start, Congestion Avoidance and Fast Retransmit. Reno TCP is an improvement over Tahoe TCP in that it incorporates Fast Recovery into Fast Retransmit. Reno TCP can survive single segment loss but suffers seriously from multiple segment loss within one transmission window. NewReno TCP [4] and SACK TCP [5, 6] are two major solutions to the multiple segment loss problem.

As bandwidth has increased and new applications have appeared, various versions of TCP were proposed to improve the congestion control performance of TCP. FAST TCP [7] is a new TCP congestion control algorithm for high-speed long-latency networks. It aims to rapidly stabilize high-speed networks into steady, efficient, and fair operating points. HighSpeed TCP [8] is a modification to standard congestion control for use with TCP connections with large congestion windows. TCP Westwood [9, 10] is based on NewReno TCP and it is aimed at better handling large bandwidth-delay product paths with potential packet loss. XCP [11] generalizes the Explicit Congestion Control Notification [12] proposal and introduces the new concept of decoupling utilization control from fairness control. The decoupling concept allows a more flexible protocol design and opens new avenues for service differentiation.

Despite the outstanding experimental performance of these newer versions of TCP, Reno, NewReno, and SACK TCP are still the most widely used versions [13]. Hence, in our research, we are focused on the resilience of Reno, NewReno, and SACK TCP.

1.2 Network Resilience

Network resilience is the ability of a network to recover traffic in case of failures and it plays an important role in network design. Resilience has been studied much in wired networks [14-24]. As a result, resilience features are embedded into many wired communication protocols.

In a typical TCP/IP/SONET network, separate resilience mechanisms exist in different layers. At the SONET layer, there are two types of resilience mechanisms: Point-To-Point and Self-Healing Ring. Three Point-to-Point mechanisms, One-Plus-One (1+1), One-For-One (1:1), and One-For-n (1: n) [25-27], are defined in SONET. In 1+1 protection, a protection line is provided for every working line, and the working and protection line are on two physically diverse routes. Thus in the case of working line failure, traffic uses the protection line to reach the destination. In 1:1 Protection, two diverse routes are provided for the same connection. What makes 1:1 Protection different from 1+1 Protection is that, with 1:1 Protection, at the transmit end traffic is not bridged to both the working and protection line. Rather it is transmitted only on the working line under normal conditions. 1:1 Protection allows the protection line to carry low-priority traffic in the absence of a failure. 1: n Protection is similar to 1:1 Protection. The only difference is that in 1: n Protection one protection line is used to protect n working lines. A SONET ring is "a collection of nodes forming a closed loop whereby each node is connected to two adjacent nodes via a duplex communication facility" [26]. It is often referred to as "Self-Healing Ring" because it can provide redundant bandwidth for protection.

At the IP layer, resilience mechanisms are inherent in the protocol because Internet Protocol (IP) was designed with survivability in mind. Internet pioneer Dave Clark pointed out in 1988 that "Internet communications must continue despite loss of networks or gateways" [28]. Of course this has much to do with the military origin of ARPANET, as well as the less reliable communication at that time. IP restores network failures by rerouting, which is supported by the dynamic routing protocols running at the IP layer. Broadly speaking, there are two types of routing protocols in IP: Distance Vector Routing and Link State Routing [2]. Both of them are dynamic routing protocols and thus can reroute traffic around failed links (or nodes) in the network. OSPF [29] is one of the widely-deployed Link State Routing Protocols. We use OSPF as an example to illustrate the inherently resilient nature of IP. All routers running OSPF periodically flood Link State Advertisements (LSAs) over the whole network. Each router collects these LSAs in its link state database. With the information stored in its database, each router can construct the topology of the network. By running Dijkstra's shortest path algorithm on the topology, every router builds its own routing table. Topology changes result in some LSAs flooded in the network, and the flooding process goes on until all routers in the network have received the information and updated their routing tables. After this, the traffic is rerouted along the new path.

At the TCP layer, TCP congestion control deals with packet loss events. TCP congestion control does not distinguish network congestion from network failure. As long as packet loss events are detected, TCP congestion control will take actions to restore the affected session. The details of TCP congestion control are presented in Section 2.1

Separate resilience mechanisms might result in redundancy and inefficiency. Hence, various multi-layer resilience mechanisms were proposed [20, 30, 31]. The idea behind multi-layer resilience mechanisms is that multiple layers in the protocol stack should coordinate with each other in order to meet the resilience objectives and improve system efficiency. This has been an active research area for both wired and wireless networks. T. H. Wu recognized the importance of providing synergy between the healing actions of different network layers and summarized the techniques that had been used in optical networks in [15]. P. Demeester and M. Gryseels conducted an in-depth study on cross-layer survivability in wired networks and presented an effective cross-layer resilience mechanism for ATM-over-SDH-over-WDM networks in [30]. In May 2005, C. M. Sadler proposed "Cross-Layer Approach to Self-Healing", in which nodes use information from each layer of the protocol stack to help routing protocol maintain wireless ad hoc network reliability in the presence of failures [31].

This thesis is focused on TCP resilience. We would like to explore whether the resilience performance of TCP is satisfactory. If not, we would like to propose some changes to improve its resilience.

1.3 Thesis Contributions

In this thesis, we present the resilience behavior of SACK, NewReno, and Reno TCP. Two types of mechanisms are also proposed to improve SACK TCP resilience. Finally, a hybrid method for protocol development is illustrated to test the effectiveness of the improved version of SACK TCP. The contributions of the thesis are summarized as follows.

- Most network researchers are aware that TCP has internal mechanisms to deal with packet losses. Many have some idea that SACK TCP and NewReno TCP are two different approaches to improving Reno TCP resilience. But few people really understand the detailed step-by-step interactions that occur in TCP in response to network congestion or failure. This thesis illustrates the detailed interaction in TCP in the case of packet losses.

- The physical layer of the transport network (SONET, SDH) incorporates the ability to restore traffic in less than 50 ms. The 50 ms target was established when voice traffic was the dominant load. Now that the volume of data traffic has surpassed voice, this target might be too slack, or too exacting. Our experimental results demonstrate that the traditional 50 ms target is not suitable for TCP transport on the Internet. Instead, varied resilience objectives are proposed for SACK, NewReno, and Reno TCP, respectively.
- The most up-to-date TCP version, SACK TCP, was designed to be capable of surviving multiple segment loss. However, we found that if too many segments in one transmission window are lost, even if SACK TCP transitions into Fast Recovery, it is still possible that timeout will finally occur and performance will be degraded significantly. In this thesis, we propose two probing mechanisms that decrease the impact of lost segments significantly.
- SACK TCP tends to send out segments in clusters and congest bottleneck routers, which could potentially degrade SACK TCP performance significantly. In this thesis, we present α-min paced SACK TCP that spreads out outstanding segments and reduces the number of segments queued at bottleneck routers.
- Traditionally, when a change in TCP is proposed, the change is often tested using simulation tools. And usually improvement is reported by showing the simulation results. However, the correctness of the implementation can never be proved by simulation itself. Petri nets, originating from the early work of Carl Adam Petri [32], have well-defined semantics that enable us to formally analyze Petri net models. We modeled the improved version of SACK TCP with Petri nets and formally analyzed the Petri net models in order to determine the correctness of our proposals.
- Formal methods enable us to verify the protocol of interest formally. Simulation excels in testing a new protocol in a relatively realistic environment. In this thesis, we present a hybrid protocol development methodology, embedding formal models into simulation, that combines the use of formal methods and simulation to obtain the advantages of deep formal

verification with the broad spectrum testing of simulation.

1.4 Thesis Outline

The rest of the thesis is organized as follows. In Chapter 2, an introduction to TCP congestion control mechanisms, modeling with Petri nets, protocol development with formal methods and simulation are presented. In Chapter 3, we illustrate the detailed resilience performance of SACK, NewReno, and Reno TCP and recommend the proper resilience objectives for TCP transport on the Internet. In Chapter 4, two types of approaches to improving SACK TCP resilience, Probing and Pacing, are presented. In Chapter 5, we talk about the hybrid method for protocol development. Chapter 6 contains our conclusions and future work.

Chapter 2

Related Work

In this chapter we discuss TCP congestion control, modeling with Petri nets, and the role that formal methods and complex simulation play in communication protocol development. Chapter 2 is organized as follows. Section 2.1 presents various TCP congestion control mechanisms. Petri net model methodology is illustrated in Section 2.2. Section 2.3 talks about different formal methods and simulation packages and their application to protocol development.

2.1 TCP Congestion Control

Broadly speaking, there are two types of congestion control mechanisms that can be used at the transport layer in Internet: network-assisted congestion control and end-to-end congestion control. With network-assisted congestion control, explicit feedback about network congestion status is sent back to the transport-layer sender by the network-layer components. This approach was used in IBM SNA and DEC DECnet [33]. It was also proposed for TCP/IP networks [12]. With end-to-end congestion control, the transport layer is more independent of the network layer. No feedback is provided by the network-layer components and the end nodes at the transport layer infer network congestion status by observing the performance of the network, such as packet loss and delay. TCP employs an end-to-end congestion control mechanism.

TCP has evolved over the past decades. So far the main versions of TCP that are widely deployed are Tahoe TCP, Reno TCP, NewReno TCP and SACK TCP [3]. Tahoe TCP is the oldest version and few systems employ Tahoe TCP today. Hence, in our research, we are focused on the resilience of Reno, NewReno, and SACK TCP. The detailed congestion control mechanisms that are used in these versions are presented as follows.

2.1.1 Congestion Control in Reno TCP

As mentioned previously, Reno TCP employs four different congestion control algorithms: Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery. These algorithms are illustrated in detail in this section.

Two terms are widely used in TCP. Sender Maximum Segment Size (SMSS) is the size of the largest segment that the sender can transmit. In practice, we can set this value to a fixed number or use the Path MTU Discovery [34] algorithm to determine the proper value. FlightSize is the amount of data that has been sent but not yet acknowledged. A Retransmission Timer [35] is also used in TCP. When a segment is sent, the retransmission timer is started. If the acknowledgement (ACK) for the segment arrives before the timer expires, the timer is stopped. Otherwise, the sender will assume that the segment is lost due to either congestion or failure and retransmit the segment.

In Reno TCP, three state variables are maintained to deal with network congestion. The congestion window (*cwnd*) is an upper bound imposed by the sender and limits the amount of data that can be transmitted into the network. According to [36], the initial value of *cwnd* must be less than or equal to 2x SMSS. In practice, *cwnd* is usually set to SMSS [1]. The receiver window (*rwnd*) is a receiver limit on the amount of outstanding, unacknowledged traffic. This initial value is set by the receiver during the negotiation process of a TCP connection. TCP never sends out more than the minimum of *cwnd* and *rwnd*. Thirdly, the slow start threshold (*ssthresh*) is the critical value that determines whether TCP is in the "Slow Start" state or in the "Congestion Avoidance" state. The initial value of *ssthresh* may be arbitrarily high. But the de facto value in practice is 64 KB [1]. When *cwnd* is less than *ssthresh*, the Slow Start algorithm is used, while the Congestion Avoidance algorithm is used when *cwnd* is greater than *ssthresh*.

After a TCP connection is established, the sender starts transmitting segments. Since TCP incorporates an end-to-end congestion control mechanism, the sender has no idea of the congestion status of the network when it is ready to send segments. To avoid congesting the network with a large burst of data, TCP is designed to slowly probe the network to determine the available bandwidth. Slow Start algorithm is used for this purpose at the beginning of a transfer.

During Slow Start, *cwnd* is increased by one SMSS for each ACK that acknowledges new data. Namely, the following equation is used to update *cwnd* during Slow Start:

$$cwnd_{new} = cwnd_{old} + \text{SMSS} \tag{2.1}$$

This way, as long as TCP is in Slow Start, the number of segments that can be inserted into the network increases exponentially.

Slow Start ends when *cwnd* exceeds *ssthresh*. Then TCP transitions into Congestion Avoidance state. During Congestion Avoidance, *cwnd* is incremented by one SMSS per round-trip time (RTT). TCP stays in Congestion Avoidance until congestion is detected. The following equation is commonly used to update *cwnd* each time an ACK arrives acknowledging some new data:

$$cwnd_{new} = cwnd_{old} + \text{SMSS x (SMSS} / cwnd_{old}) \tag{2.2}$$

At any time during a transfer, when the sender detects a segment loss event using the Retransmission Timer, it will retransmit the lost segment. In addition, *cwnd* must be set to one SMSS and *ssthresh* must be set to no more than the value given in equation 2.3.

$$ssthresh = (\text{FlightSize} / 2, 2 \text{ x SMSS}) \tag{2.3}$$

With TCP, a receiver sends an immediate duplicate ACK when an out-of-order segment arrives. Thus when a segment is dropped on the way from the sender to the receiver, all segments after the dropped segment will generate duplicate ACKs. From the sender's perspective, the arrival of duplicate ACKs can be viewed as a sign of segment loss. If this sign can be used effectively, the sender will be able to retransmit the lost segment shortly after the loss event occurs before resorting to the costly Retransmission Timer to detect a segment loss event. Fast Retransmit algorithm, based on duplicate ACKs, was designed to detect segment loss and retransmit the lost segment in a much more timely fashion than the Retransmission Timer approach. With Fast Retransmit, the arrival of three duplicate ACKs is considered an indication that a segment has been lost. After Fast Retransmit algorithm retransmits the lost segment, Fast Recovery algorithm governs the transmission of new data until a non-duplicate ACK arrives. Fast Recovery algorithm is used in this phase to prevent the communication path from going empty after Fast Retransmit, thus avoiding the need to Slow-Start to refill the path after a single segment loss. Usually the Fast Retransmit and Fast Recovery algorithms are implemented together as follows:

(1) When the third duplicate ACK arrives, the sender retransmits the lost segment, updates *ssthresh* according to Eq. 2.3, and sets *cwnd* to (*ssthresh* + 3 x SMSS).

(2) For each additional duplicate ACK, *cwnd* is increased by one SMSS.

(3) During Fast Recovery, the sender should send a segment if it is allowed by the value of *cwnd* and *rwnd*.

(4) When an ACK that acknowledges new data arrives, *cwnd* should be set to *ssthresh*. Then TCP transitions into Congestion Avoidance from Fast Recovery.

2.1.2 Congestion Control in NewReno TCP

As illustrated in previous section, each invocation of Reno Fast Retransmit/Fast Recovery results in the retransmission of only one lost segment. If there is only one lost segment, then the ACK corresponding to this retransmitted segment will acknowledge all of the segments that were transmitted before TCP transitioned into Fast Retransmit. In this case, Reno Fast Retransmit/Fast Recovery helps TCP recover from the segment loss event efficiently. However, if there are multiple segments that are dropped from a single window of data, the ACK corresponding to the retransmitted segment will only acknowledge some but not all of the segments transmitted before the Fast Retransmit. This kind of ACK is called a Partial ACK. With Reno Congestion Control, a Partial ACK also makes TCP transition from Fast Retransmit/Fast Recovery to Congestion Avoidance. However, since the Partial ACK does not acknowledge all the segments transmitted before the Fast Retransmit, the FlightSize is still very large after TCP switches to Congestion Avoidance. Actually the FlightSize is very likely to be greater than *cwnd*. Since TCP never sends out more than the minimum of *cwnd* and *rwnd*, the sender will not send anything until the Retransmission Timer expires in this case of multiple segment loss. Then TCP transitions into Slow Start and starts retransmitting the second, the third, etc. lost segment. In short, Reno Congestion Control has to resort to the Retransmission Timer to recover from multiple segment loss. To avoid triggering the costly retransmission timeout, NewReno Congestion Control was proposed [4, 37]. With NewReno Congestion Control, as long as Partial ACKs keep arriving, the lost segments will be retransmitted step by step, and the costly retransmission timeout is thus avoided. NewReno Congestion Control is the same as Reno Congestion Control except that the 4th step of the Reno Fast Retransmit/Fast Recovery algorithm is changed to the following:

> (4) When an ACK that acknowledges new data arrives, the sender checks to see if this ACK is a Partial ACK. If it is a not Partial ACK, *cwnd* should be set to *ssthresh*, then TCP transitions into Congestion Avoidance from Fast Recovery. If it is a Partial ACK, the sender should retransmit the first unacknowledged segment, deflate *cwnd* by the amount of new data acknowledged, then add back one SMSS. Whether it is a Partial ACK, the sender should always send a new segment if allowed by the new value of *cwnd*.

2.1.3 Congestion Control in SACK TCP

NewReno TCP can survive multiple segment loss. SACK TCP [5, 6, 38] is another solution to the multiple segment loss problem. Selective acknowledgement (SACK) together with selective retransmission helps improve TCP performance when multiple segments are dropped within one window of data. Selective acknowledgement is achieved by adding to the ACK a list of the non-contiguous blocks of data that have been received by the receiver. When a valid segment arrives that is in the receive window but is not the next expected segment, the receiver sends back a selective acknowledgement to inform the sender that non-contiguous blocks of data have been received.

Table 2.1 shows how SACK acknowledgements work in an example where all segments are 500 bytes in length. The successful arrival of the first segment with sequence number 500 triggers the receiver to send a normal ACK with acknowledgement number 1000. As sequence numbers really indicate byte offsets in the transmission stream, this ACK indicates that the receiver is requesting the data that begins at offset 1000. If segments 2 through 4, and then 7 and 8 are dropped, they will not result in any ACKs. When segment 5 is received it triggers the receiver to send a selective acknowledgement indicating the span of data that has been received (left edge of 2500 and right edge of 3000) while the data starting at offset 1000 has not been received (ACK number of 1000). When segment 6 is received it again triggers an ACK with number 1000 but this time the left and right edge are 2500 and 3500. If segments 7 and 8 are dropped, the receipt of segment 9 leads to a selective acknowledgement indicating that two non-contiguous data blocks have been received. One block corresponds to segment 5 and 6, and extends from byte 2500 to byte 3499; the receiver indicates this with a left edge of 2500 and a right edge of 3500 because 3500 is the offset of the next byte following the data received in the first block. The left and right edges of the other non-contiguous block of data that has been received are 4500 and 5000, corresponding respectively to the beginning of segment 9 (received) and 10 (not yet seen). SACK reports blocks from most to least recent, so the block from 4500 to 5000 is reported first, and the block from 2500 to 3500 is reported second, as noted in Table 2.1.

Table 2.1 SACK Acknowledgement

Arriving Segment		Returning ACK				
Sequence Number	Transmission Status	ACK Number	First Block		Second Block	
			Left Edge	Right Edge	Left Edge	Right Edge
500	Successful	1000				
1000	Dropped	No ACK				
1500	Dropped	No ACK				
2000	Dropped	No ACK				
2500	Successful	1000	2500	3000		
3000	Successful	1000	2500	3500		
3500	Dropped	No ACK				
4000	Dropped	No ACK				
4500	Successful	1000	4500	5000	2500	3500

In addition to the three state variables maintained in Reno TCP, SACK TCP has another state variable called *pipe*. SACK TCP uses *pipe* to represent the estimated number of segments outstanding in the path during Fast Recovery. SACK TCP also has a data structure called the *scoreboard* that keeps track of the contiguous data blocks that have arrived at the receiver. SACK TCP employs the same Slow Start and Congestion Avoidance algorithm as Reno TCP. However, SACK TCP Fast Retransmit and Fast Recovery algorithm are completely different than those used in Reno TCP. The details are as follows:

(1) When the third duplicate ACK arrives, the sender retransmits the lost segment, updates *ssthresh* according to Eq. 2.3, sets *cwnd* to *ssthresh*, and sets *pipe* to FlightSize.

(2) During Fast Recovery, *pipe* is increased by one SMSS when the sender either sends a new segment or retransmits an old segment. It is decreased by one SMSS when the sender receivers a selective acknowledgement reporting that new data has been received at the receiver.

(3) When *pipe* is less than *cwnd*, the sender will retransmit the earliest unacknowledged segment or send a new segment when there are no such segments.

(4) When an ACK that acknowledges new data arrives, TCP transitions into Congestion Avoidance from Fast Recovery.

2.2 Formal Analysis and Performance Evaluation with Petri Nets

The area of Petri nets, originating from the early work of Carl Adam Petri [32], has been developed tremendously in both the theory and the applications [39-45]. The Petri net method was designed specifically to study information processing systems that are characterized as being concurrent, asynchronous, and distributed [39]. One of the main attractions of Petri nets is the way in which the basic aspects of concurrent systems are modeled both graphically and mathematically [39, 40]. Essentially, a Petri net is a clear graphical representation of a concurrent system. This graphical representation can be used to present an animated view of a simulation, demonstrating how the system works in great detail. In addition, Petri nets have well-defined mathematical semantics which unambiguously define the behavior of the model. The formal semantics of Petri nets form the theoretical foundation that allows us to analyze structural and behavioral properties of Petri net models.

"Petri net" is actually a generic name that is used for a class of net-based models. This class of model can be divided into three main levels. The first level is very fundamental and is especially suitable for a thorough investigation of concurrent systems. The basic model here is called Elementary Net Systems, or EN systems [46]. Although this is the most fundamental type, it is not suited for practical applications because the size of the model explodes even for simple applications. The second level is an intermediate model that is more practical than EN systems. The basic model at this level is Place/Transition nets, or P/T nets [47]. With P/T nets, some repetitive features of EN systems are folded in order to get more compact representations. Finally, the third level is the most advanced and practical. The Petri nets at this level are called high-level nets. With high-level nets, algebraic and logical tools are used to yield compact nets that are suitable for real-life applications. Predicate/Transition nets [48] and Colored Petri nets [40] are the most widely used high-level nets. In our research, Colored Petri nets are chosen to study the performance of TCP resilience. Thus, we are focused on formal analysis and performance evaluation of Colored Petri nets.

In this section, Elementary Net Systems, Place/Transition nets, and Colored Petri nets are first introduced briefly; then we present how to formally analyze Colored Petri nets and how to evaluate the performance of concurrent systems modeled with Colored Petri nets.

2.2.1 Elementary Net Systems

Elementary Net Systems form the most fundamental class of Petri nets [46]. As a net-based model, every EN system is composed of a net, a certain number of tokens, and a set of rules that control the way that tokens move in the net. The net determines the static structure of a concurrent system and the rules decide the dynamic behavior of the system. Actually, different types of nets and rules lead to different classes of Petri nets. We are focused on EN systems in this section.

The net in an EN system is a directed bipartite graph. There are two types of nodes in the graph, "places" and "transitions". In graphical representation, places are drawn as circles or ellipses; transitions are drawn as rectangles. All arcs in the graph are either from a place to a transition or from a transition to a place. Since arcs are directed, we can define the input place of a transition as the place that is linked to the transition by an arc pointing to the transition. Similarly, the place that is linked to a transition by an arc leaving the transition is called its output place. The tokens in the EN system are assigned to the places in the net. In EN systems, each place can hold at most one token, denoted by a dot in the corresponding circle or ellipse. Theoretically, the way that tokens are assigned to places in EN systems can be formulated as a function that maps places to either 0 or 1. If a place is mapped to 0, then there is no token in the place. And if it is mapped to 1, then the place has one token. In EN systems, this kind of function is called a marking, M. Essentially, each marking represents a particular state of the modeled system. Every EN system is associated with an initial marking that determines the way that tokens are assigned to the places in the system. This initial marking is denoted by M_0. In addition, we use N and N_+ to denote non-positive integers {0, 1, 2, ...} and positive integers {1, 2, 3, ...}, respectively. Now we are ready to define EN systems formally.

Definition 1. An Elementary Net System, or an EN system for short, is a 4-tuple EN = (P, T, A, M_0), where:

(1) P is a finite set of places: {P1, P2, ... , P_m}, where $m \in N$.

(2) T is a finite set of transitions: {T1, T2, ... , T_n}, where $n \in N$.

(3) A is a set of arcs: $A \subseteq (P \times T) \cup (T \times P)$.

(4) M_0 is the initial marking, which maps the elements in P to either 0 or 1.

The dynamic behavior of an EN system is simulated by moving tokens from places to places, namely, changing the state of the system from one marking to another one. In EN systems, a marking is changed according to the following transition firing rules:

(1) A transition is said to be enabled if each input place of the transition has one token.

(2) A firing of an enabled transition removes the token from each of its input places and deposits one token to each of its output places.

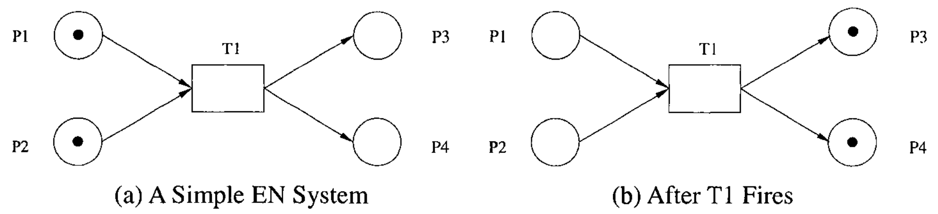


Figure 2.1 An Example EN System

An example EN system is shown in Figure 2.1 (a). Formally, the example EN system is defined as a 4-tupe EN_{ex} = (P, T, A, M_0), where:

(1) P = {P1, P2, P3, P4}.

(2) T = {T1}.

(3) A = {(P1, T1), (P2, T1), (T1, P1), (T1, P2)}.

(4) $M_0(p) = \begin{cases} 1 & \text{if } p = P1 \\ 1 & \text{if } p = P2 \\ 0 & \text{if } p = P3 \\ 0 & \text{if } p = P4 \end{cases}$

With the initial marking in the example EN system, transition T1 is already enabled since each input place of T1 has a token. After T1 fires, the tokens on P1 and P2 are removed and one token is deposited to each of the output places of T1, P3 and P4. At this moment, since no transition is enabled any more, the system arrives at the final stable state, shown in Figure 2.1 (b). This simple example is used to illustrate how EN systems work. We should be aware that, in real applications, the net could be much more complex and there could be a large number of concurrently enabled transitions.

2.2.2 Place/Transition Nets

As mentioned previously, size explosion is a problem with EN system modeling. To make Petri net modeling more practical, a new class of Petri nets, Place/Transition nets, was proposed [47]. There are two main differences between EN systems and P/T nets. First of all, in P/T nets, each place can hold multiple tokens, as opposed to only one token in EN systems. Thus, a marking in P/T nets maps places to non-negative integers. Secondly, in P/T nets, arcs are labeled with their weights (positive numbers). An n-weighted arc in a P/T net can be viewed as n parallel arcs in an EN net. Labels for unit weight are usually omitted. Formally, a P/T net is defined as follows.

Definition 2. A Place/Transition net, or P/T net, is a 5-tuple PTN = (P, T, A, W, M_0), where:

(1) P is a finite set of places: {P1, P2, ... , P_m}, where $m \in N$.

(2) T is a finite set of transitions: {T1, T2, ... , T_n}, where $n \in N$.

(3) A is a set of arcs: $A \subseteq (P \times T) \cup (T \times P)$.

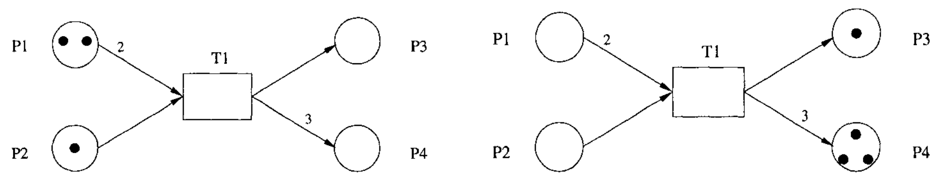
(4) W is a weight function that maps the elements in A to non-negative integers: $A \rightarrow N_+$.

(5) M_0 is the initial marking. Namely, it is a function that maps the elements in P to non-negative integers: $P \rightarrow N_+$.

Due to the definition change, the transition firing rules in P/T nets are changed as follows:

(1) A transition t is enabled if each input place p_{in} of t has at least $w[(p_{in}, t)]$ tokens, where $w[(p_{in}, t)]$ is the weight of the arc from p_{in} to t.

(2) A firing of an enabled transition t removes $w[(p_{in}, t)]$ tokens from each input place p_{in} of t and deposits $w[(t, p_{out})]$ tokens to each outplace p_{out} of t.



(a) A Simple P/T Net (b) After T1 Fires

Figure 2.2 An Example P/T Net

An example P/T net is shown in Figure 2.2 (a). This P/T net is similar to the example EN system mentioned previously. However, in the P/T net, there are two tokens on P1. In addition, the arc (P1, T1) and (T1, P4) are associated with arc weight 2 and 3, respectively. Formally, the P/T net is defined as a 5-tupe, $PTN_{ex} = (P, T, A, W, M_0)$, where:

(1) P = {P1, P2, P3, P4}.

(2) T = {T1}.

(3) A = {(P1, T1), (P2, T1), (T1, P1), (T1, P2)}.

(4) $$W(a) = \begin{cases} 2 & \text{if } a = (P1, T1) \\ 1 & \text{if } a = (P2, T1) \\ 1 & \text{if } a = (T1, P3) \\ 3 & \text{if } a = (T1, P4) \end{cases}$$

$$(5)\ M_0(p) = \begin{cases} 2 & \text{if } p = P1 \\ 1 & \text{if } p = P2 \\ 0 & \text{if } p = P3 \\ 0 & \text{if } p = P4 \end{cases}$$

In the example P/T net, T1 is already enabled with the initial marking. After T1 fires, one token is removed from P2 while two tokens are removed from P1 due to the weight associated with the arc from P1 to T1. The firing of T1 also results in one and three tokens deposited into P3 and P4, respectively. After this event, the P/E net reaches a stable state and do not change any more. This is shown in Figure 2.2 (b).

2.2.3 Colored Petri Nets

In both EN systems and P/T nets, there is only one type of token. This limits the descriptive capability of Petri nets and contributes to the size explosion problem significantly. If we can equip each token with an attached data value, these problems can be alleviated dramatically. Traditionally, the data value is called the color of the token. Further, if there could be different types of colors, Petri nets could be even more compact. By tradition, each type of token color is called a color set. Actually, the use of "color" and "color set" in Petri nets is very analogous to the use of "value" and "type" in programming languages. This innovative class of Petri nets is called Colored Petri nets, or CP nets [40]. We mentioned previously that P/T nets are more advanced than EN systems since each place in P/T nets can hold any number of tokens. Now we know that CP nets go one step further so that the tokens on a place are distinguishable.

In CP nets, a place is associated with only one color set. All tokens on a given place have token colors that belong to the color set associated with the place. A place in a CP net can have a multi-set of tokens. Intuitively, a multi-set is similar to a set except that all elements in a set are different whereas there could be multiple appearances of the same element in a multi-set. For example, if we add set {a, b, c} to set {c}, we still have the set {a, b, c}. However, if we add multi-set {a, b, c} to multi-set {c}, the result is a multi-set {a, b, c, c}. By convention, the multi-set {a, b, c, c} is usually denoted as 1\`a+1\`b+2\`c, where the number before the \`-operator indicates the number of tokens of each color. Since each place could have a multi-set of tokens, a marking in a CP net maps places to multi-sets. Furthermore, in CP nets, each arc is associated with an arc expression instead of just a weight as in P/T nets. The arc expressions on all input places of a transition determine whether the transition is enabled. In addition, guard expressions can be attached to transitions in CP nets. A guard expression is a Boolean expression defining an additional constraint that must be satisfied before the transition is enabled. Now we are ready to define CP nets formally.

Definition 3. A Colored Petri net, or a CP net, is a 8-tuple CPN = ($\sum$, P, T, A, C, G, E, M_0), where:

(1) $\sum$ is a finite set of non-empty types, called color sets.

(2) P is a finite set of places: {P1, P2, … , P_m}, where m $\in$ N.

(3) T is a finite set of transitions: {T1, T2, … , T_n}, where n $\in$ N.

(4) A is a set of arcs: A $\subseteq$ (P x T) $\cup$ (T x P).

(5) C is a node function that maps the elements in P to the elements in $\sum$. C determines the color set associated with each place.

(6) G is a guard function. It is defined from T into expressions. If the expression is simply "true", the expression is usually omitted in CP nets.

(7) E is an arc expression function. It is defined from A into expressions. These expressions determine how tokens are removed from or added to places.

(8) M_0 is the initial marking. Namely, it is a function that maps each element in P to a multi-set whose elements belong to the same color set.

In CP nets, the transition firing rules are as follows:

(1) When all the input places of a transition contain at least the number of tokens prescribed by the expression associated to each corresponding arc and the guard expression evaluates to "true", the transition is said to be enabled.

(2) A firing of an enabled transition removes tokens from each input place and deposits tokens to each outplace according to the arc expressions associated with each

corresponding arc.

An example CP net is shown in Figure 2.3 (a). In the example net, there are three places, Num_P, Div2_P, and Div3_P; there is only one transition, Tran_1. The color set of Num_P is INT and a multi-set of token 1`1 is associated with Num_P. Div2_P and Div3_P have the same color set BOOL and the same multi-set 1`false. Tran_1 has one guard, “n<4”. Several different expressions are attached to the arcs in this example net. In these expressions, we use n to denote an INT variable; we also use b2 and b3 to denote two boolean variables. Formally, the example net can be defined as an 8-tuple $CPN_{ex} = (\sum, P, T, A, C, G, E, M_0)$, where:

(1) ∑ = {INT, BOOL}.

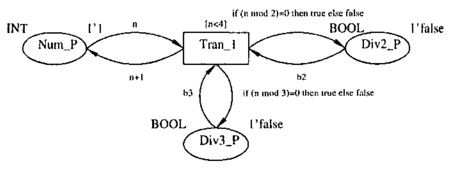
(2) P = {Num_P, Div2_P, Div3_P}.

(3) T = {Tran_1}.

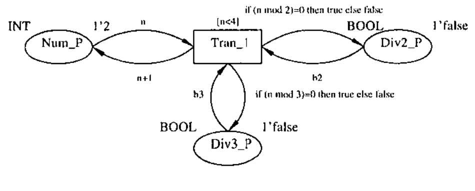
(4) A = {(Num_P, Tran_1), (Tran_1, Num_P), (Div2_P, Tran_1), (Tran_1, Div2_P), (Div3_P, Tran_1), (Tran_1, Div3_P)}.

(5) C(p) =	INT	if p = Num_P
	BOOL	if p = Div2_P
	BOOL	if p = Div3_P
(6) G(t) =	n<4	if t = Tran_1
	true	otherwise
(7) E(a) =	n	if a = (Num_P, Tran_1)
	n+1	if a = (Tran_1, Num_P)
	if (n mod 2) = 0 then true else false	if a = (Tran_1, Div2_P)
	b2	if a = (Div2_P, Tran_1)
	if (n mod 3) = 0 then true else false	if a = (Tran_1, Div3_P)
	b3	if a = (Div3_P, Tran_1)
(8) M_0(p) =	1`1	if p = Num_P
	1`false	if p = Div2_P
	1`false	if p = Div3_P

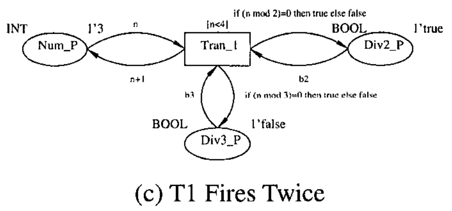
The example CP net is more complex than previous example nets. It actually accomplishes a simple task: each time Tran_1 fires, if 2 divides into the token from Num_P, a multi-set 1`true will be added to Div2_P, otherwise 1`false will be added; and if 3 divides into the token from Num_P, a multi-set 1`true will be added to Div3_P, otherwise 1`false will be added. Now let us have a look at the details. When the net is in the initial state shown in Figure 2.3 (a), Tran_1 is already enabled. At this moment, the multi-set on Num_P is 1`1. Since both 2 and 3 do not divide into 1, after the transition fires, the multi-sets on Div2_P and Div3_P remain the same. However, due to the arc expression n+1, the multi-set on Num_P changes to 1`2 after the firing. The state of the net is shown in Figure 2.3 (b). When this event is over, Tran_1 is enabled again. This time since 2 divides into 2 and 2 does not divides into 3, after Tran_1 fires, 1`true is added to Div2_P and 1`false is deposited in Div3_P. Of course, the multi-set on Num_P becomes 1`3 at the same time. This is shown in Figure 2.3 (c). At this state, Tran_1 is still enabled, as a result of the firing, Num_P has a multi-set 1`4, Div2_P has 1`false, and Div3_P has 1`true. Figure 2.3 (d) illustrates the state of the net after this event. At this moment, Tran_1 is not enabled any more because the guard expression attached to Tran_1 is n<4 while the multi-set on Num_P has only one element 4 which does not satisfy the guard expression. Thus the state corresponding to Figure 2.3 (d) is actually the final state of the CP net.


INT
1'1
n
if (n mod 2)=0 then true else false
[n<4]
BOOL
1'false
Num_P
Tran_1
Div2_P
n+1
b3
b2
if (n mod 3)=0 then true else false
BOOL
1'false
Div3_P

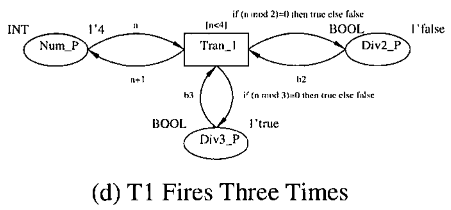
(a) A Simple CP Net


INT
1'2
n
if (n mod 2)=0 then true else false
[n<4]
BOOL
1'false
Num_P
Tran_1
Div2_P
n+1
b3
b2
if (n mod 3)=0 then true else false
BOOL
1'false
Div3_P

(b) T1 Fires Once


INT
1'3
n
if (n mod 2)=0 then true else false
[n<4]
BOOL
1'true
Num_P
Tran_1
Div2_P
n+1
b3
b2
if (n mod 3)=0 then true else false
BOOL
1'false
Div3_P

(c) T1 Fires Twice


INT
1'4
n
if (n mod 2)=0 then true else false
[n<4]
BOOL
1'false
Num_P
Tran_1
Div2_P
n+1
b3
b2
if (n mod 3)=0 then true else false
BOOL
1'true
Div3_P

(d) T1 Fires Three Times

Figure 2.3 An Example CP Net

2.2.4 Formal Analysis and Performance Evaluation with Colored Petri Nets

Petri nets have been widely used to formally analyze system properties [39, 49, 50] and evaluate system performance [39, 51, 52]. Petri nets have many behavioral properties that are worth analyzing, such as reachability, boundedness, liveness, fairness, etc. In our research, we are focused on reachability and boundedness. Occurrence Graphs and Place Invariants are two typical formal analysis methods used in Petri nets [40]. However, the use of Place Invariants is less automatic and involves a great deal of human reasoning. In comparison, an occurrence graph can be constructed automatically. Hence, the occurrence graph is a more error-free and convenient tool with which Petri net models can be analyzed. In our study, we use occurrence graphs to formally analyze CP nets.

We know that the first firing of transitions moves a CP net from the initial marking M_0 to the next marking M_1. This process goes on until there are no more enabled transitions. Then the CP net stays at the final marking, M_n. This sequence of markings, "$M_0, M_1, \dots, M_n$" is called an occurrence sequence. A marking M_k is said to be reachable from the initial marking M_0 if there exits an occurrence sequence M_0 , M_1 , ... , M_k. Essentially, the reachability problem is the problem of finding whether a certain marking M_j is reachable from the initial marking M_0. In addition, a CP net is said to be k-bounded if the number of tokens in each place does not exceed a positive integer k for any marking reachable from M_0.

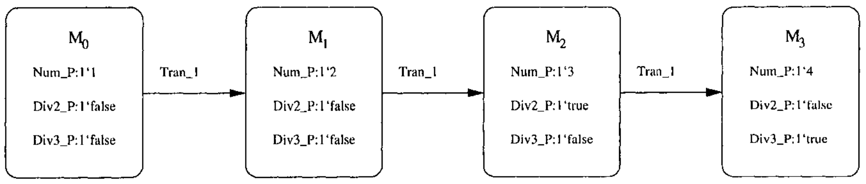


Figure 2.4 Occurrence Graph of the Example CP Net

The basic idea behind Occurrence Graphs is to construct a graph containing a node for each reachable marking and an arc for each enabled transition. Once the occurrence graph is available, reachability and boundedness can be easily analyzed. Using the previous CP net as an example, we can construct the corresponding occurrence graph shown in Figure 2.4. There are altogether four markings for this CP net, M_0, M_1, M_2, and M_3. M_0 is changed to M_1 due to the firing of Tran_1, then to M_2 and M_3 sequentially. From the occurrence graph, we can conclude that M_1, M_2, and M_3 are reachable from M_0; no other markings are not reachable from M_0. And since there is at most one token on each place for any marking reachable from M_0, we can also reach the conclusion that the example CP net is 1-bounded.

So far we have been talking about investigating the logical correctness of a CP net by formally analyzing its behavioral properties. However, CP nets can also be used to evaluate the performance of the modeled system via simulation. To perform this kind of analysis, it is convenient to extend CP nets with a time concept. This type of CP net is called a timed CP net [40]. In our research, a global clock is introduced to represent the modeling time. A timestamp is attached to each token. The execution of a timed CP net is time-driven and it works in a similar way to discrete event simulation.

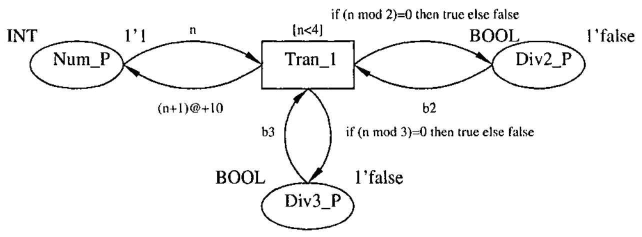

INT
1'1
n
Num_P
[n<4]
Tran_1
if (n mod 2)=0 then true else false
BOOL
Div2_P
1'false
(n+1)@+10
b2
b3
if (n mod 3)=0 then true else false
BOOL
1'false
Div3_P

Figure 2.5 An Example Timed CP Net

To see how a timed CP net simulation works, let us consider the timed CP net in Figure 2.5. This timed CP net is similar to the previous example CP net except that the expression associated with the arc from Tran_1 to Num_P is changed from n+1 to (n+1)@+10. Now let us see how the simulation progresses. When the simulation starts, all tokens are available and Tran_1 will fire immediately. After the firing, 1`false is added to Div2_P and Div3_P; 1`2 is added to Num_P, but it can not be used to enable Tran_1 until 10 time units later because of the arc expression (n+1)@+10. We should be aware that in timed CP nets, arc expressions without special notations, such as "if (n mod 2) = 0 then true else false" that is associated with the arc from Tran_1 to Div2_P, are available immediately after the firing. At this moment, the CP net is in a state similar to the one shown in Figure 2.3 (b). After 10 time units pass, 1`2 becomes available and Tran_1 will be enabled. As a result of the firing of Tran_1, the CP net will change to a state similar to the one shown in Figure 2.3 (c). After this, again, 1`3 is not available until 10 time units later. Then 10 time units go by and Tran_1 fires, the CP net is changed to a state similar to the one shown in Figure 2.3 (d). However, the simulation is not over until 10 time units later. At that time, 1`4 becomes available, but since the guard expression cannot be satisfied, the CP net will not change any more. In total, the simulation of the example timed CP net lasts 30 time units. The data gathered from a simulation can help us to evaluate the performance of the system modeled with a timed CP net.

2.3 Protocol Development with Formal Methods and Simulation

Reliable communication protocols play an important role in providing effective communication services. Hence, measures should be taken to guarantee the quality of protocols during the development process. Formal methods [53] and simulation [54] are two important methodologies that have been widely used to develop new communication protocols. This section presents an overview of several formal methods and simulation packages, and illustrates their application to protocol development using real-life examples.

2.3.1 Protocol Development with Formal Methods

In the early days of protocol development, the process was carried out in an informal way that relied on informal textual documentation, graphical description techniques, etc. The informal methodology, lacking scientific foundation and relying on human expertise, leads to ambiguous definitions and provides no means to analyze protocols of interest. Later, mathematically-based techniques were adopted in protocol development, introducing rigor and reliability into the various steps of the development process. Such techniques are called formal methods (FM). Formal methods allow us to formally analyze the protocol of interest and increase our confidence about the correctness of the protocol. There are many formal methods that can be used for protocol development [53]. An overview of several widely-used formal methods is presented as follows.

SDL (Specification and Description Language) is a formal description language standardized by ITU-T [55]. The latest version is SDL-2000, an object-oriented description language. The effectiveness and the intuitive graphical format of SDL have made it a very popular formal method in both academia and industry. Several standard institutes, including ETSI (European Telecommunications Standards Institute) [56] and 3GPP (Third Generation Partnership Project) [57], have included SDL diagrams in their official specification documents. With SDL, structural and functional aspects are separated. The system structure is described by drawing the building blocks and the channels connecting them. The function of each block is specified by editing the internal processes in the block. SDL is based on Finite State Machine (FSM). Formal analysis of static and dynamic properties of the model in SDL is performed by analyzing the equivalent FSM produced by the combination of control states and data values. SDL has been applied to system specification and analysis in many application areas. The details are documented in SDL proceedings [58].

LOTOS (Language of Temporal Ordering Specifications) is a formal description language approved by ISO in 1989 [59]. Different than SDL, LOTOS is based on process algebra. Process algebras are a diverse family of related approaches to formally modeling concurrent systems. Process algebras provide a tool for the high-level description of interactions, communications, and synchronizations between a collection of independent processes. Although LOTOS was designed for information processing system modeled after the OSI reference model, it has been widely used to specify and validate sequential, concurrent, and distributed systems in various domains. Due to textual format and strong abstraction, LOTOS requires greater effort than SDL to be employed. Some effort has been devoted to support specifications in graphical LOTOS, but there have been no mature products. Nevertheless, LOTOS has been used to prove the validity of formal methods in development of real communication systems.

Estelle, standardized by ISO in 1989, is a formal description technique for specifying distributed and concurrent systems. SDL, LOTOS, and Estelle are commonly referred to as the three Formal Description Techniques (FDTs) that are used for the specification of protocols in standardization documents [60]. Estelle is based on Finite State Machines and it employs Pascal to describe protocols. Due to the inclusion of a programming language, Estelle is more implementation-oriented than SDL and LOTOS. Actually, the formal specification in Estelle can be compiled into an executable model automatically. Estelle has also been used to specify and analyze real-life communication protocols.

PROMELA (Process Meta Language) is a widely-used description language for specification, simulation, and validation of communication protocols [61]. It is based on FSM and supports validation of consistency requirements and invariant assertions. It employs C-like syntax, which increases its applicability in the early stages of the design and makes the implementation phase an automatic process. SPIN (Simple PROMELA Interpreter) is a popular tool for simulating and validating protocols specified in PROMELA. Due to its effectiveness, SPIN has become the most exploited among freeware FM-based tools for formal analysis of communication protocols. Extensive work has been carried out using PROMELA. Theoretical advancements and practical applications are continuously documented in the proceedings of the International SPIN Workshops.

Petri nets are an effective tool for describing and analyzing concurrent and distributed systems. As mentioned in Section 2.2, Petri nets provide a clear graphical representation of protocols of interest. This graphical representation can be used to present an animated view of a simulation, demonstrating how the system works in great detail. In addition, Petri nets have well-defined mathematical semantics that form the theoretical foundation, allowing us to analyze structural and behavioral properties of Petri net models. Petri nets have been widely used in both academic and industrial areas [62].

All of the above formal methods can be applied to protocol development. In this section, we use a real-life example project [63] of Petri net application to illustrate the effectiveness of formal methods. Chapter 4 also illustrates the application of Petri nets to protocol development by modeling two revised versions of SACK TCP with Petri nets. In the real-life example project at Ericsson Telebit A/S, CP nets were used for the specification and validation of an edge router discovery protocol (ERDP) for mobile ad hoc networks. Ad hoc networking is different from traditional networks in that the nodes in the ad hoc network operate in a fully independent and distributed fashion, without any existing communication infrastructure. The mobile nodes in an ad hoc network, such as laptops, personal digital assistants, and mobile phones, are capable of establishing a communication infrastructure for their common use. In a hybrid network where the nodes in the ad hoc networks communicate with the nodes in the core network, ERDP is used between the gateways in the ad-hoc networks and the edge routers in the core network. ERDP supports gateways to discover edge routers and supports edge routers to configure gateways with a globally routable IPv6 address prefix so that all nodes in the hybrid network can communicate with each other. The development of ERDP started with a specification of the initial ERDP in natural language. Then a CP net model corresponding to the specification was created using Design/CPN [62]. The act of creating the CP net model led to the identification of several problems with the initial version of ERDP. After revising the initial specification, the CP net model was modified accordingly. One important feature of ERDP is to configure the gateway with prefixes properly. Occurrence Graphs were used to formally analyze this property of ERDP. The results showed that for a given prefix and state where the gateway has not yet been configured with that prefix, the protocol is able to configure the gateway with the prefix; furthermore, when the gateway has been configured with the prefix, the edge router and the gateway are properly synchronized. Hence, it was concluded that the proposed ERDP functions properly and this is proved by formal analysis. Details of the project can be found in [63].

2.3.2 Protocol Development with Complex Simulation

Simulation is an effective method for protocol design and development [54]. An individual researcher can start from scratch and write specific simulation programs to evaluate protocols of interest. However, this kind of simulation usually employs small-scale topologies and simplified assumptions. High costs prevent an individual researcher from creating a comprehensive and advanced networking simulation environment to run realistic simulations. This limits the understanding of protocols of interest and the reproducibility of simulations constructed by different groups.

Multiprotocol network simulators allow us to run complex simulations in a common simulation environment to solve the previously mentioned problem. Normally, multiprotocol network simulators have abundant built-in protocol models and internal mechanisms to generate complex simulation scenarios. They provide the opportunity to investigate large-scale protocol interaction in a controlled environment and enable easy comparison of proposals from different research groups. Many communication protocols were developed with the assistance of various multiprotocol network simulators. An overview of several multiprotocol network simulators is presented as follows.

GloMoSim (Global Mobile Information System Simulator) [64] is a scalable simulation environment for wireless network systems. It is designed using the parallel discrete-event simulation capability provided by Parsec [65]. GloMoSim 2.0 is the latest version and it supports protocols for a purely wireless network. At the data link layer, CSMA, MACA, and 802.11 are implemented. The network layer protocols that have been included in the simulator are AODV, DSR, Bellman-Ford, Fisheye, LAR scheme 1, ODMRP, and WRP. At the transport layer, TCP and UDP are implemented. The successor to GloMoSim is QualNet, a GloMoSim-based commercial product from Scalable Network Technologies. QualNet extends the base capabilities of GloMoSim and includes some specialized libraries, such as the libraries for MANET, Satellite, and WiMAX. GloMoSim is widely used in wireless network research due to its open source nature.

OPNET [66] is a commercial modeling tool for designing and evaluating communication networks, devices, protocols, and applications with flexibility and scalability. OPNET Modeler is the simulator for wired networks and Modeler Wireless Suite is used for wireless network study. OPNET offers a comprehensive library of detailed protocol and application models, including Voice, HTTP, TCP, IPv4, BGP, EIGRP, RIP, VoIP, OSPFv3, RSVP, Frame Relay, FDDI, Ethernet, ATM, 802.11 Wireless LANs, 802.16, WiMAX, IPv6, MPLS, PNNI, DOCSIS, UMTS, IP Multicast, Circuit Switch, MANET, Mobile IP, IS-IS, satellite capabilities, and many more. Aside from standard protocols, OPNET Modeler and Modeler Wireless Suite have built-in vendor-specific models, allowing more realistic simulations. Compared with GloMoSim, OPNET is more widely used in industry than in research.

ns-2 [67] is a discrete event simulator for networking research. It especially excels in modeling of TCP, routing, and multicast protocols over wired and wireless networks. ns-2 was simply a variant of the REAL network simulator in 1989 and has evolved much over the past decades. ns-2 development was supported by DARPA through the VINT project [68] in 1995. Currently, ns-2 development is support through DARPA with SAMAN [69] and through NSF with CONSER [70]. ns-2 has abundant built-in protocols, especially the protocols for wired networks. Due to its open source nature, ns-2 keeps adding contributed models from other researchers to its library. For example, the wireless code in ns-2 is actually from UCB Daedelus, CMU Monarch Project, and Sun Microsystems. ns-2 is the de facto multiprotocol network simulator for wired network research. Hence, many research projects chose ns-2 as the simulation environment.

All of the above multiprotocol network simulators enable us to run complex simulations to develop communication protocols. In this section, we use ns-2 as the example simulator to illustrate the effectiveness of running complex simulations for protocol development purposes. T. R Henderson and R. H. Katz studied whether some modified TCP congestion avoidance algorithms can improve TCP fairness using ns-2 [71]. Two problems were investigated in the project. The first problem is to examine whether a change to TCP congestion control, as proposed by Floyd, that results in a 'constant rate' *cwnd* increase during Slow Start can correct the bias against connections with long round trip times (RTTs). The second problem is to explore whether TCP connections with long RTTs could independently improve their own throughputs by becoming slightly more aggressive during Slow Start. ns-2 was used to model the several different experimental scenarios and the relatively complex simulations were carried out, resulting in the following conclusions. With regard to the first problem, it is concluded that despite the fact that the constant rate algorithm can improve fairness significantly, it was also found that such a policy is difficult to implement in heterogeneous networks. As for the second problem, the experimental results show that long RTT connections benefit from the aggressive policy and peer connections are not influenced dramatically. Hence, the aggressive policy should be included in the new version of TCP and the constant rate policy should not.

Chapter 3

Resilience Performance of SACK, NewReno, and Reno TCP

The reaction of TCP to packet loss events is a key issue because TCP currently accounts for over 90% of the traffic on the Internet [72]. If TCP reacts inefficiently when packet loss events are experienced, the performance of the Internet will be degraded significantly. Packet loss events can be caused by either network failures or congestion. Physical failures are relatively common in the fiber-optic networks that form the core of the Internet: a top-tier carrier will, on average, experience one fiber failure every three hours [73]. We should be careful to note that the frequency of failure in optical networks, and the massive potential impact of such failures, is hidden from end-users (and availability measures) by sophisticated protection and recovery mechanisms. Optical networks use redundancy to prevent the unreliability of a single component, such as a fiber link, from directly becoming the unavailability of the network. Failures in end-equipment (fiber terminals, cross-connects, switches, etc.) are more common than fiber failures, and again equipment designers use redundancy to ameliorate the effects of individual failures. Of course, as we are all too aware as users of various networks, simple congestion events are even more frequent than failures.

To prevent a fiber failure from making a portion of the network unavailable, the physical layer of the transport network (SONET, SDH) incorporates the ability to restore traffic in less than 50 ms. The 50 ms target was established when voice traffic was the dominant load. Now that the volume of data traffic has surpassed voice, we would like to know whether this target is too slack, or too exacting, in the context of providing service for Internet traffic over optical networks. In the course of this work, we studied the resilience performance of SACK, NewReno, and Reno TCP.

In our research, we consider the reaction of a single TCP session to packet losses on the link from the sender to the receiver. Interactions between multiple TCP flows are not taken into account. We believe that the TCP protocol itself is complex enough that it is necessary to first understand how TCP behaves in this baseline scenario, before setting standards, proposing mechanisms, or exploring the impact of additional variables. The general behavior of SACK, NewReno, and Reno TCP is presented in Section 3.1, 3.2, and 3.3. More details of their resilience performance are included in Section 3.4.

3.1 SACK TCP Resilience

SACK TCP is the most up-to-date version of TCP [3, 5, 6, 38] and has been widely deployed in the Internet [13]. In this section we detail and quantify the reactions of SACK TCP to packet loss events due to network failures and congestion episodes. We present a Petri net model of the resilience aspects of SACK TCP and use it to explain the step-by-step behavior of the protocol in various circumstances. We also use simulation results derived from the Petri net to relate the protocol's behavior during network failures to its reactions to congestion. The step-by-step analysis of TCP's reactions leads to a simple algebraic model that can be used to predict the number of lost segments above which TCP's performance degrades significantly.

Most networkers are aware that SACK TCP has the best resilience mechanism of the versions of TCP released to date. Many have some idea how SACK TCP makes use of selective acknowledgements to recover more quickly from dropped packets. But few people really understand the detailed step-by-step interactions that take place in response to network congestion or failure. We believe that one of the contributors to this lack of detailed understanding is the traditional use of graphs of sequence number vs. time or congestion window vs. time to illustrate the behavior of TCP [3] [74] [75]. Such graphs can present parameter value changes over time clearly, but they lack the ability to demonstrate the internal behavior of TCP.

In contrast, Petri nets can be used to give a clear, step-by-step illustration of the behavior of a protocol in specific circumstances. As mentioned in Section 2.2, Petri nets have many advantages.

First of all, a Petri net is a clear graphical representation of a system. Secondly, Petri nets have well-defined semantics which unambiguously define the behavior of the model. The formal semantics of Petri nets form the theoretical foundation that allows us to analyze Petri net models for properties such as reachability and boundedness. Such analyses can be used to check whether proposed changes in a protocol preserve its correctness, or introduce faults [53, 76, 77]. Thirdly, the Petri net model of a system can be used in simulation mode, in order to sample the system's behavior statistically over a range of parameter values.

Ref. [78] presents a simplified Petri net model of TCP and is focused on the performance analysis of Tahoe and Reno TCP. A generic Petri net model of TCP is illustrated in [76] and it is used to study the performance of a TCP variant, ARTCP [76]. In this section, we present a much more detailed Petri net model than was exhibited in either [76] or [78], and use it to study SACK TCP. Our model has two large components, called TCP Sender and TCP Receiver. The TCP Sender follows the formal specifications for TCP as set out in [3, 5, 6] and is capable of dealing with selective acknowledgements. The TCP Receiver accepts TCP segments and sends out corresponding acknowledgements. The combined behavior of these two components was validated against simulation results from a popular commercial simulator, OPNET [66], and has been shown to reproduce the actions of SACK TCP both qualitatively in terms of the steps it takes, as well as quantitatively in terms of delays, *cwnd*, *rwnd*, and *ssthresh.* And further, we have used our Petri net model to test both SACK TCP and our modified version for reachability, boundedness, and liveness [77]; these are formal, fundamental properties of SACK TCP that have not been tested by others.

This section focuses on leveraging a Petri net model, and the algebraic model for delay derived from it, to explain SACK TCP resilience in detail. The graphical representation of our Petri net model of SACK TCP enabled us to observe visually the detailed interactions that occur inside TCP when network congestion or failure takes place, and lead to the illustrations in this thesis. We also used our Petri net model in "simulation mode" to obtain insight into scenarios with a large number of protocol instances. These simulation results are unique in that this research is the only work we are aware of that focuses on the resilience behavior of SACK TCP, and then derives an algebraic performance model based on results from something that is formally analyzed. We present the Petri net model of SACK TCP in Section 3.1.1 and illustrate the detailed resilience performance of SACK TCP in Section 3.1.2.

3.1.1 Petri Net Model of SACK TCP

SACK TCP is a complex protocol [38]. We focus here on the resilience of the protocol to packet losses caused by network congestion or failure, so only the SACK TCP functions that are closely related to resilience are included in our Petri net model. Other loosely related functions, such as connection establishment, persistence timer and keep-alive timer, are purposely omitted.

We have used Colored Petri Nets (CP nets), one of the many varieties of High-level Petri Nets, together with a Petri net modeling environment called Design/CPN [62]. As mentioned in Section 2.2, CP nets are composed of places, transitions and (directed) arcs. In CP nets, each place has an associated type defining the kind of tokens that a place can contain. The type may be a simple (e.g. integer) or compound (e.g. product type, list type). In Design/CPN, a timestamp can be attached to each token. The timestamp defines the earliest time at which the token can be used to enable a transition. The timestamp concept makes it possible to model concurrent systems involving time using CP nets. Design/CPN also facilitates the modeling of complex systems by providing the capability to describe them top-down as a hierarchy of gradually more detailed behaviors. One transition in the top level of a hierarchy may correspond to tens of transitions and places at the next level of detail. This ability to abstract details into more general behaviors is crucial to assembling complex models, such as that for SACK TCP.

The top-level description of the system is illustrated in Figure 3.1. A function called Segment Discarder is placed on one of the links connecting a SACK TCP sender and receiver. The Segment Discarder is used to simulate packet losses due to congestion or failure. We can specify the number of segments to be dropped within a transmission window as well as being able to drop segments consecutively or at random within a transmission window.

At this level of detail, the SACK TCP Sender (on the left-hand side of the figure) is composed of four parts. **Send Segment** takes care of sending TCP segments when they are ready and **Receive ACK** handles receiving acknowledgements from the receiver when they arrive. **Parameter Management** manipulates the important status parameters at the sender. These include the congestion window (*cwnd*) and the slow start threshold (*ssthresh*). **Retran Timer Management** is responsible for starting, updating and stopping the retransmission timer. When the retransmission timer expires it signals **Send Segment** and **Parameter Management** to retransmit the corresponding segment and update the status parameters.

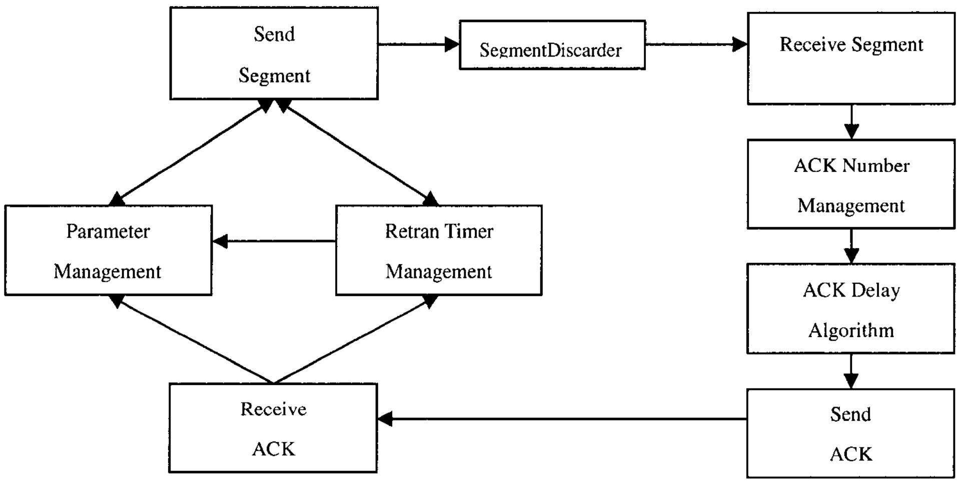
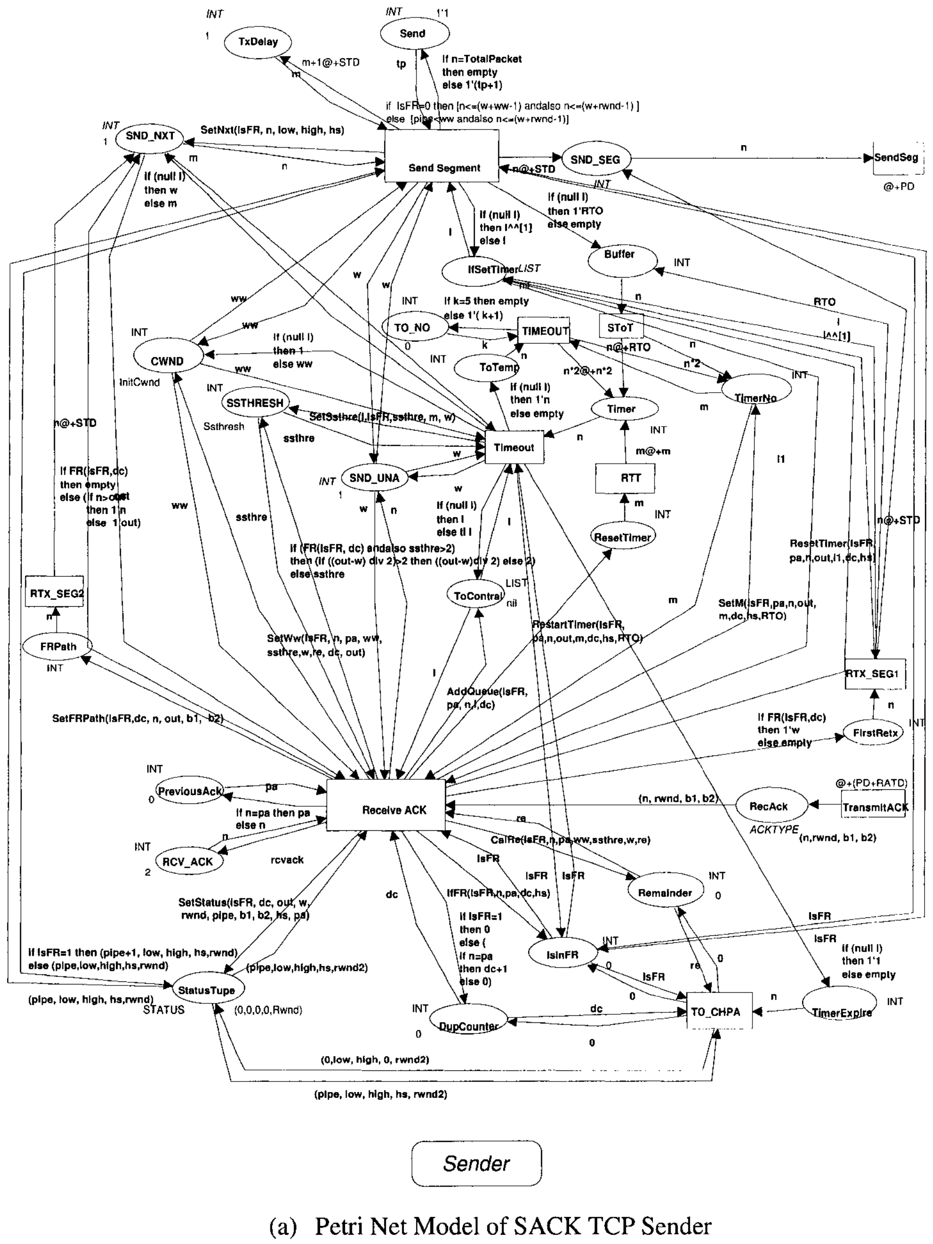


Figure 3.1 Top-Level View of the Petri Net Model

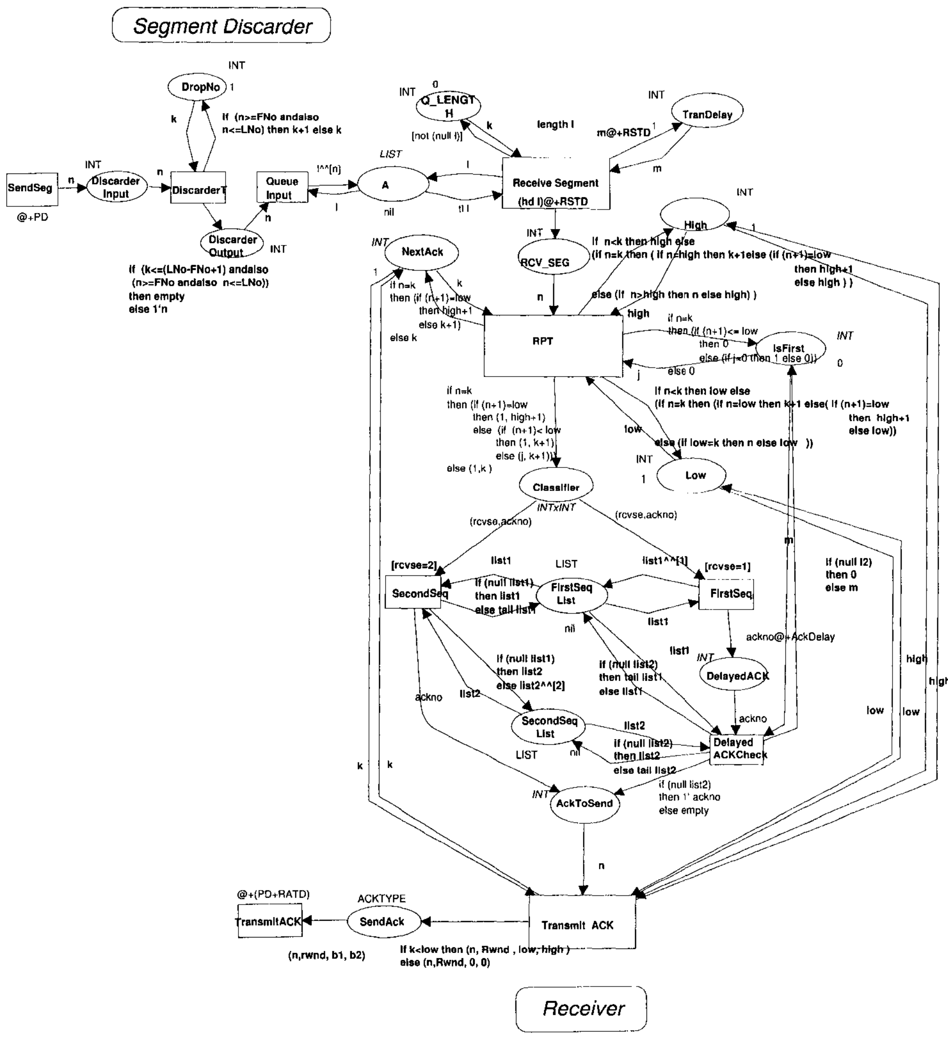
We have used rectangles to draw all four parts of the sender. In Petri net notation these denote transitions. However, in the upper layers of a hierarchical description we use rectangles to denote collections of places and transitions, rather than denoting a top-level transition. This practice also applies to the receiver components on the right-hand side of Figure 3.1.

The top-level description of the SACK TCP receiver contains four parts in series. **Receive Segment** and **Send ACK** manage the functions related to receiving segments and sending ACKs. **ACK Number Management** generates ACK numbers. Generally, acknowledgements should be sent after the arrival of every second full-sized segment, and they must be sent within 500ms of the arrival of the first unacknowledged segment [36]. Most implementations use a delay of 200ms instead of 500ms [1]. We follow this common practice in our model. This delayed ACK mechanism is modeled in the transition called **ACK Delay Algorithm**.

The detailed Petri net model of our system is shown in Figure 3.2. For clarity, the original Petri net model was split into two parts. Figure 3.2 (a) shows the model for SACK TCP sender and Figure 3.2 (b) illustrates the model for SACK TCP receiver and Segment Discarder. Note that transition "**SendSeg**" and "**TransmitACK**" appear in both Figure 3.2 (a) and Figure 3.2 (b). They are actually the splitting points, meaning that removing one copy of them and then combining Figure 3.2 (a) and Figure 3.2 (b) will result in the original Petri net model.



(a) Petri Net Model of SACK TCP Sender



(b) Petri Net Model of SACK TCP Receiver and Segment Discarder

Figure 3.2 Detailed Petri Net of SACK TCP

In this section, we only use **ACK Delay Algorithm** as an example to illustrate this level of detail in the hierarchical description of the protocol. As shown in Figure 3.3, **ACK Delay Algorithm** is composed of five places and three transitions. Both **FirstSeqList** and **SecondSeqList** are of type *LIST*. **DelayedAck** and **ACKToSend** are integer type (*INT*), and **Classifier** has the product type (*INT x INT*). *INT x INT* is a Cartesian product of integers. Sample values of this type include (1, 5) and (6, 2). **ACK Delay Algorithm** receives tokens bearing two-tuples denoted as (*rcvse, ackno*), where *rcvse* and *ackno* are two integer variables. The two-tuples carry information about the acknowledgements to be sent. The variable *rcvse* means "receive sequence". If *rcvse* is 1, the two-tuple corresponds to the first segment received by the TCP receiver after the receiver has acknowledged the previous segment. Thus, the corresponding acknowledgement should be delayed. If *rcvse* is 2, then the two-tuple corresponds to the second received segment, and the acknowledgement should be transmitted immediately. The variable *ackno* specifies the acknowledgement number to be included in the final acknowledgement. When **ACK Delay Algorithm** decides that it is time to send out an acknowledgement, it simply outputs the current *ackno* from **ACKToSend**. After receiving the *ackno*, the module responsible for packaging and sending the acknowledgement will finalize the transmission.

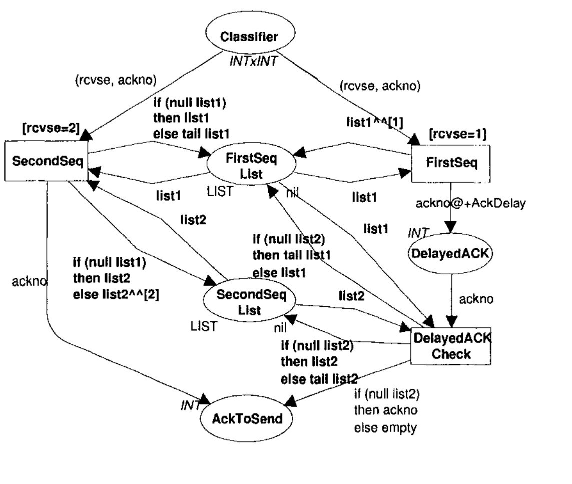


Figure 3.3 Detailed View of ACK Delay Algorithm

To illustrate how the delayed acknowledgement mechanism works in detail, let us consider the arrival sequence of tokens given in Table 3.1. If we assume that all previously received segments have been acknowledged before the token labeled (1, 1000) arrives, then the lists at **FirstSeqList** and **SecondSeqList** will be empty and there will be no tokens in any of the places in **ACK Delay Algorithm** at 100 ms. When **Classifier** receives (1, 1000), *rcvse* is equal to 1 so only **FirstSeq** is enabled. This is the result of the guard [*rcvse=2*] on **SecondSeq**. After **FirstSeq** fires, an element "1" is inserted in the list on **FirstSeqList.** This is indicated by the expression list1^^[1] in Figure 3.3. The token 1000 corresponding to the current value of *ackno* is also moved to **DelayedACK**. This token is not available until 200 ms later due to the arc expression *ackno@+AckDelay* where *AckDelay* is defined to be 200 ms. After this no transition is enabled until the token labeled (2, 1500) arrives at 200 ms. When this occurs, only **SecondSeq** is enabled because *rcvse* is now 2 and the guard on **FirstSeq** is [*rcvse=1*]. After **SecondSeq** fires an element "2" is inserted in the list on **SecondSeqList**. This is indicated by the expression list1^^[2] in Figure 3.3. The *ackno* value of 1500 is also moved in a token to **AckToSend**. A token at **AckToSend** means that an acknowledgement corresponding to the current token should be sent immediately. This illustrates a sequence of events where an acknowledgement is generated for every second full-sized segment.

Table 3.1 Token Arrival Sequence

Received Token	Arrival Time (ms)
(1, 1000)	100
(2, 1500)	200
(1, 2000)	400

The scheduled token at **DelayedACK** becomes available at 300 ms, enabling **DelayedAckCheck**. After **DelayedAckCheck** fires, the current *ackno* is not moved to **AckToSend** because of the expression "*if (null list2) then ackno else empty*" on the arc from **DelayedAckCheck** to **AckToSend**. Recall that the previous firing of **SecondSeq** inserted "2" in the list **list2** at **SecondSeqList**. As a result, the *(null list2)* clause in the expression is false so the "*else empty*" action is performed. The firing of **DelayedAckCheck** removes the element "2" from **list2** due to the expression "*if (null list2) then list2 else tail list2*" associated with the arc from **DelayedAckCheck** to **SecondSeqList**. This sequence of events illustrates how the Petri net manages the expiry of the timer that would have forced an acknowledgement to be sent for (1,1000) if (2,1500) had not arrived and triggered an ACK for both segments.

At 400 ms, the token labeled (1, 2000) arrives. The value *rcvse*=1 indicates that this is the first segment after the last acknowledged segment, so **FirstSeq** is enabled. This moves a token of value 2000 to **DelayedAck**. That token will not be available until 600 ms. If the next input token arrives after 600 ms, then the scheduled token will enable **DelayedAckCheck**. After

`DelayedAckCheck` fires, a token with value 2000 is moved to `AckToSend` to trigger the immediate transmission of an acknowledgement as `list2` is empty. This illustrates a sequence of events where an acknowledgement is forced by the timing rule that no more than 200 ms should elapse after the arrival of the first unacknowledged segment until the sending of an ACK.

3.1.2 SACK TCP Resilience Performance

When network congestion or failures occur, some segments are dropped. In this section, we focus on situations where multiple segments are dropped within one transmission window. Network failure - either node or link failure - will most likely lead to the loss of several consecutive segments. This is because no segments will be able to traverse the network during the outage. In comparison, network congestion usually results in nonconsecutive segments being dropped because the network is still working although it is not in perfect condition. Congestion avoidance mechanisms such as random early detection (RED) [79] can also result in nonconsecutive segment loss.

Network failure and network congestion are completely different by definition. However, in terms of their segment loss patterns, network failure is “just” an extreme form of network congestion. During light network congestion, most of the segments in the current transmission window will traverse the network successfully; and any dropped segments will be spaced widely apart. As the network becomes more congested, segments will be discarded more frequently. Viewed through the transmission window at the sender, the losses become more frequent and more consecutive losses are observed. When congestion becomes extremely serious, or when network failure occurs, many consecutive segments will be discarded. The effects of a temporary network failure can thus be looked at as an extreme congestion event. We note that failures in modern transport networks are in fact quite brief, and are usually restored in 50 ms or less [80, 81].

TCP does not distinguish network failure from network congestion. This is because both situations lead to segment losses which trigger TCP’s congestion control mechanisms. RFC 2018 [5] does not specify in detail the congestion control algorithms for implementations of SACK TCP. It only requires that all implementations preserve the congestion control algorithms present in the *de facto* standard [1]. RFC 3517 [38] presents a conservative SACK-based loss recovery algorithm for TCP. The model discussed here conforms to the algorithm illustrated in [38].

In the following we discuss the specific example of a client with a relatively high-speed access link (DS1, i.e. 1.544 Mbps) connecting to a continental OC-3c backbone. The client is assumed to be downloading a large file from a distant server. The TCP session in our simulation must be long enough to test failures and congestion episodes with varying durations, so we studied the example of a 10 MB file transfer. This is large enough to enable us to study a range of situations given the link rates noted above. In reality, the duration of TCP flows covers a very large range. Although there are many short flows in terms of number, long-running flows account for up to 50% of all traffic [82].

Table 3.2 Parameter Settings

initial *cwnd*	1
ssthresh	45
rwnd	23
Propagation Delay (One Way)	16 ms
RTO	1500 ms

In these experiments, SMSS was set to 1460 bytes. The values of *cwnd*, *rwnd* and *ssthresh* were initialized to 1460 bytes, 32 KB (kilobyte) and 64 KB, respectively. By 32 KB or 64 KB, we mean a multiple of SMSS that is just above 32 KB or 64KB. For example, with an SMSS of 1460 bytes, by 32 KB we mean 1460 * 23= 33580 bytes. For clarity, we consider 1460 byte segments as the data units. Thus, *cwnd*, *rwnd* and *ssthresh* were set to 1, 23 and 45, respectively. Retransmission Timeout (RTO) is also an important parameter in TCP congestion control. RFC 2988 [35] presents a detailed method to calculate RTO. For simplicity, in our experiment we chose a fixed value of 1500 ms. This value approximates the RTO when network failure or congestion occurs in the corresponding OPNET scenario. We know that RTO does not change the TCP performance until network failure and congestion takes place and a segment is retransmitted due to timeout. Thus our simplification does not change the behavior of TCP. A list of all parameters and their values is presented in Table 3.2.

3.1.2.1 Network failures and SACK

A network failure will result in the loss of multiple consecutive segments. If the size of the receive buffer for the TCP session does not change over time, the size of the transmission window during the session is fixed once *cwnd* becomes greater than *rwnd*. In the following we focus on the period during which *cwnd* is already greater than *rwnd* because this accounts for the major part of a long TCP session. Also, the transmission window is normally shifted by one or two segments after the sender receives an ACK. When considering network failures, we assume that the first *n* consecutive segments in the transmission window are discarded. This is equivalent to studying all the other possibilities (i.e. the loss of segments 2 through n+1, 3 through n+2, etc.) because the reaction of TCP is invariant under such shifts.

We define a new metric called Transfer Time Increase, TTI, to quantify the additional time required to complete the download as a result of the failure:

$$\text{TTI} = \text{ATT} - \text{NTT} \qquad (3.1)$$

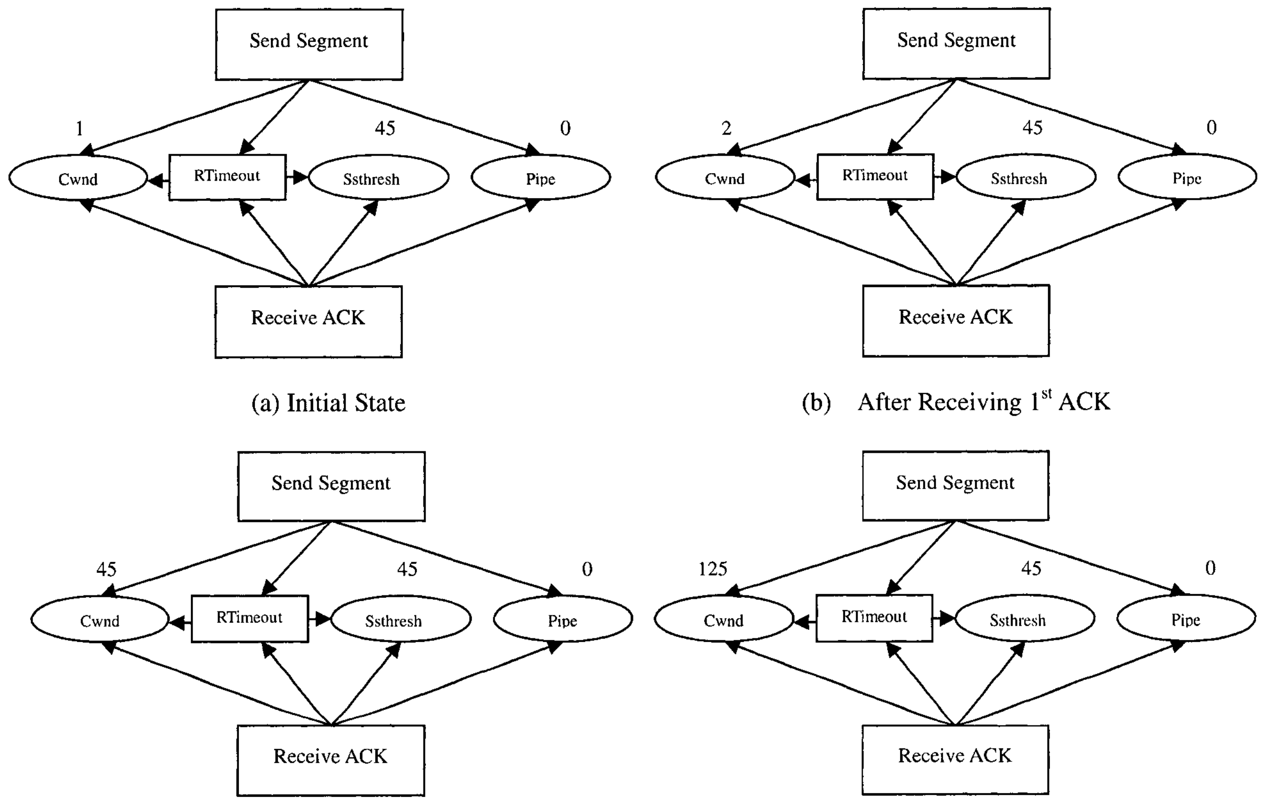
where ATT stands for Actual Transfer Time of the file when a network failure or congestion are experienced, and NTT is the Normal Transfer Time when there is no network failure or congestion.

In the following, a 10 MB file transfer starts at time 0. The Segment Discarder begins to drop segments at time 30 seconds. When viewed graphically, the Petri net shows tokens flowing in different patterns under different segment loss conditions. However, we do not have the liberty of using a dynamic presentation in this thesis. As an alternative we present explanations based on a subset of the full Petri net. This subset includes only the places **`Cwnd`**, **`Ssthresh`** and **`Pipe`**, and the transitions **`Send Segment`**, **`Receive ACK`** and **`RTimeout`** (denoting "Retransmition Timeout"). These components are all on the sender side of the TCP session.

The resilience behavior of SACK TCP can be categorized into four cases:

Case 1: No Dropped Segments

Figure 3.4 illustrates the scenario where no segment is discarded. When the session is first initiated, TCP is in Slow Start. The initial values of *cwnd*, *ssthresh* and *pipe* are 1, 45 and 0, respectively, as shown in Figure 3.4 (a). After the beginning of the file transfer, *cwnd* increases by one for every ACK received until *cwnd* equals *ssthresh*. Figure 3.4 (b) shows the state of the Petri net after the sender receives the first ACK, and Figure 3.4 (c) shows the situation when *cwnd* becomes equal to *ssthresh*. After this, TCP transitions into Congestion Avoidance. Without a network failure, TCP stays in Congestion Avoidance until the file is completely transferred. During Congestion Avoidance, *cwnd* only increases by one every round trip time (RTT). Hence, although many ACKs arrive at the sender during Congestion Avoidance, the final value of *cwnd* is just 125, as shown in Figure 3.4 (d). Note that throughout the file transfer, *pipe* remains at 0 because there are no segments dropped. In short, when there is no failure and thus no dropped segments, *cwnd* increases as ACKs arrive, and *ssthresh* and *pipe* remain fixed.



(a) Initial State

(b) After Receiving 1st ACK

(c) When *cwnd* Equals *ssthresh*

(d) Final State

Figure 3.4 State Changes without Network Failure or Congestion

Case 2: Dropped Segments, Duplicate ACKs and No Timeout

Now we consider the case in which some segments are dropped due to a failure. In this scenario, although the first several segments in the transmission window are discarded, the rest of the segments arrive at the client. The idea that the session will persist across the span of a network failure may appear odd, but recall that current physical-layer restoration mechanisms will react within 50 ms or less. For a client receiving data at 1.5 Mbps in 1460 byte segments, a 50 ms failure corresponds to a loss of no more than 7 segments.

Recall that the concepts of "duplicate ACK" and "Partial ACK" were mentioned in Section 2.1. Namely, a "duplicate ACK" is defined as an acknowledgement for a segment that has been acknowledged before. A "Partial ACK" is defined as an acknowledgement received during Fast Recovery that acknowledges new data, but does not acknowledge all the segments that were outstanding when TCP switched into Fast Retransmit/Fast Recovery.

In Case 2, the arrival of the segments following those that were dropped during the failure triggers the client to send out duplicate ACKs, in order to request the segments that were dropped. TCP transitions into Fast Retransmit/Fast Recovery after the sender receives three duplicate ACKs. It then retransmits the earliest unacknowledged segment and sets *pipe* to the number of outstanding segments (n_O). In normal conditions, n_O should be equal to the size of the receiver window, *rwnd,* which as stated earlier is equal to 23. However, during Fast Retransmit/Fast Recovery, three duplicate ACKs indicate that three segments have been received successfully, but out of order. Thus n_O should be decreased to $(rwnd - 3)$. n_O is further decremented to $[(rwnd - 3) - 1] = (rwnd - 4)$ because the sender assumes that one segment has been dropped. After the earliest unacknowledged segment is retransmitted, n_O is increased to $[(rwnd - 4) + 1] = (rwnd - 3)$ because one more segment has been inserted into the network. In our file transfer example, *pipe* is set to $(23 - 3) = 20$. RFC 3517 [38] also requires the sender to set *ssthresh* and *cwnd* to $\lfloor rwnd / 2 \rfloor$ which in this scenario is $\lfloor 23 / 2 \rfloor = 11$. Thus, we have:

$$cwnd = \lfloor rwnd / 2 \rfloor \tag{3.2}$$

We now discuss the particular instance of the file transfer example where 14 segments are dropped. TCP starts from the state shown in Figure 3.4 (a); *cwnd* increases exponentially from the time at which the session is started until the time at which the failure occurs. The state of the Petri net model immediately before the failure is shown in Figure 3.5 (a). Just before the failure, *cwnd* has a value of 98, which is greater than *rwnd*. After the failure, TCP receives three duplicate ACKs, and transitions into Fast Retransmit/Fast Recovery, setting its parameters accordingly. The state of the Petri net at that moment is shown in Figure 3.5 (b).

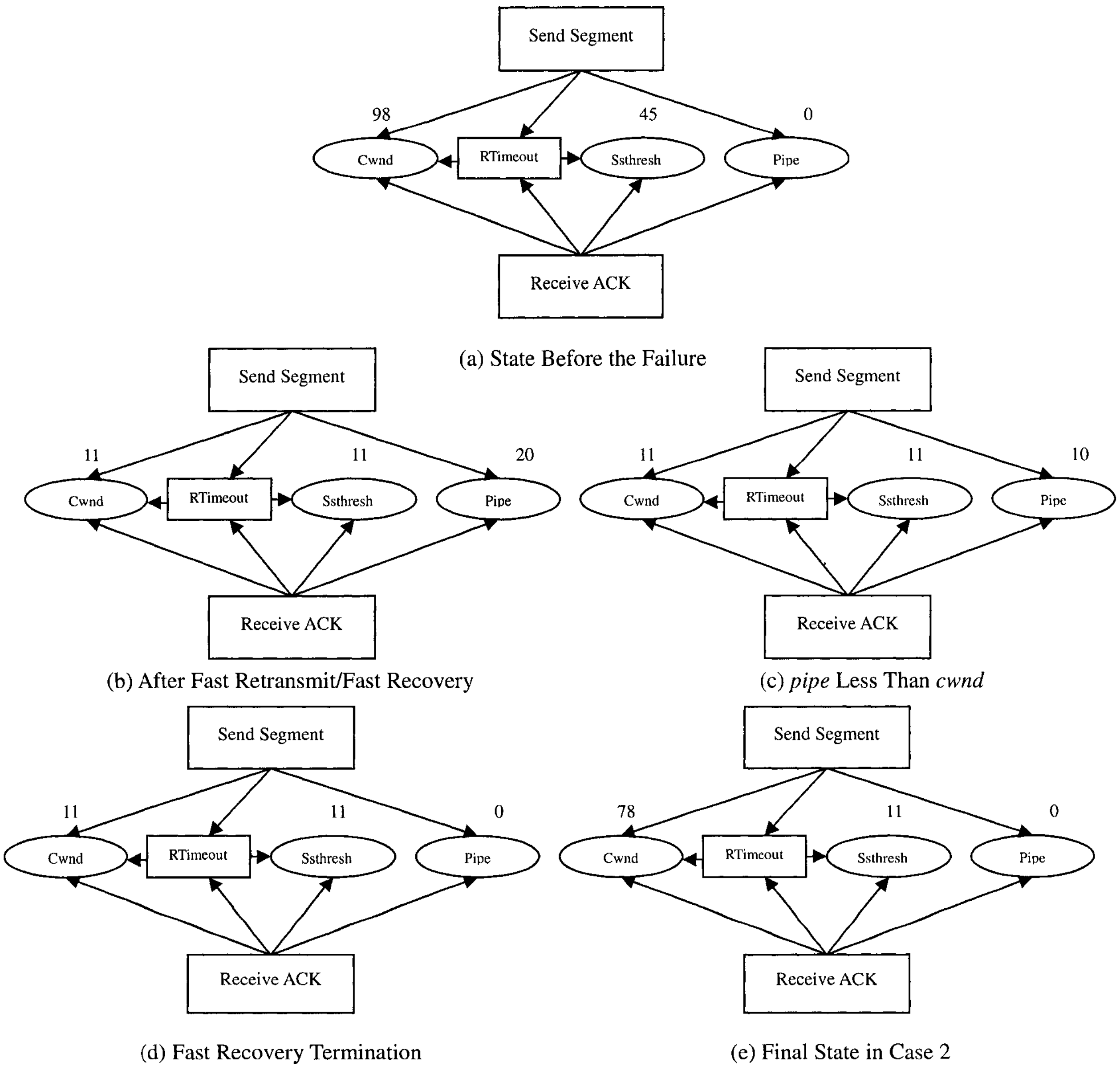


Figure 3.5 State Changes for 14 Dropped Segments

Recall that *pipe* is the sender's estimate of the number of outstanding segments in the network. During Fast Recovery, *pipe* is increased by one when the sender transmits any segment. This could be either a retransmission of an old segment or the first transmission of a new segment. *pipe* is decreased by one for each duplicate ACK received because the duplicate ACK is a signal from the receiver that a segment has left the network. For each Partial ACK, *pipe* is decreased by two because a Partial ACK indicates that two segments have left the transmission link: the original segment that is assumed to have been dropped, and the retransmitted segment. The sender does not transmit any segments during Fast Recovery until *pipe* becomes less than *cwnd.* At that time, the sender will check the *scoreboard* and either retransmit the earliest unacknowledged segment, or transmit a new segment when there are no unacknowledged segments.

We use n_D to denote the number of duplicate ACKs that have been received by the sender by the time that *pipe* has just become less than *cwnd* (i.e. *pipe* = *cwnd* -1) due to duplicate ACKs and the first Partial ACK. Note that when TCP transitions into Fast Retransmit/Fast Recovery, *pipe* is set to (*rwnd* – 3) and the earliest unacknowledged segment is retransmitted. This retransmission will lead to a Partial ACK if the ACK triggered by the retransmission will not take TCP out of Fast Retransmit/Fast Recovery. This Partial ACK will decrease *pipe* by two, so that *pipe* = [(*rwnd* - 3) - 2] = (*rwnd* – 5). In addition, *pipe* will be decreased by n_D when the duplicate ACKs arrive. Thus we have:

$$(rwnd - 5) - n_D = cwnd - 1 \tag{3.3}$$

So:

$$n_D = rwnd - cwnd - 4 \tag{3.4}$$

From Eq. 3.2 and 3.4, we have:

$$n_D = rwnd - \lfloor rwnd / 2 \rfloor - 4 \tag{3.5}$$

If we use $n_{C,S}$ to denote the number of segments dropped in this case, we arrive at:

$$n_{C,S} = rwnd - n_D = \lfloor rwnd / 2 \rfloor + 4 \tag{3.6}$$

We will refer to $n_{C,S}$ as the "critical number" for reasons which will become clear subsequently.

In the normal state the receiver only sends out an ACK for every second full-sized segment, or within 200ms of the arrival of the first unacknowledged segment. Also, out-of-order segments must be acknowledged immediately [36]. Thus when the first out-of-order segment in the window arrives, if there is no unacknowledged segment at the receiver, this segment will trigger the receiver to send out a duplicate ACK. We call this a Type I failure. If instead the first out-of-order segment arrives within 200 ms of an unacknowledged segment, the receiver will not send out a duplicate ACK. The receiver will only transmit an acknowledgement of the previously-unacknowledged segment. We call this a Type II failure. In both Type I and Type II failure, each segment following the first out-of-order segment results in a duplicate ACK.

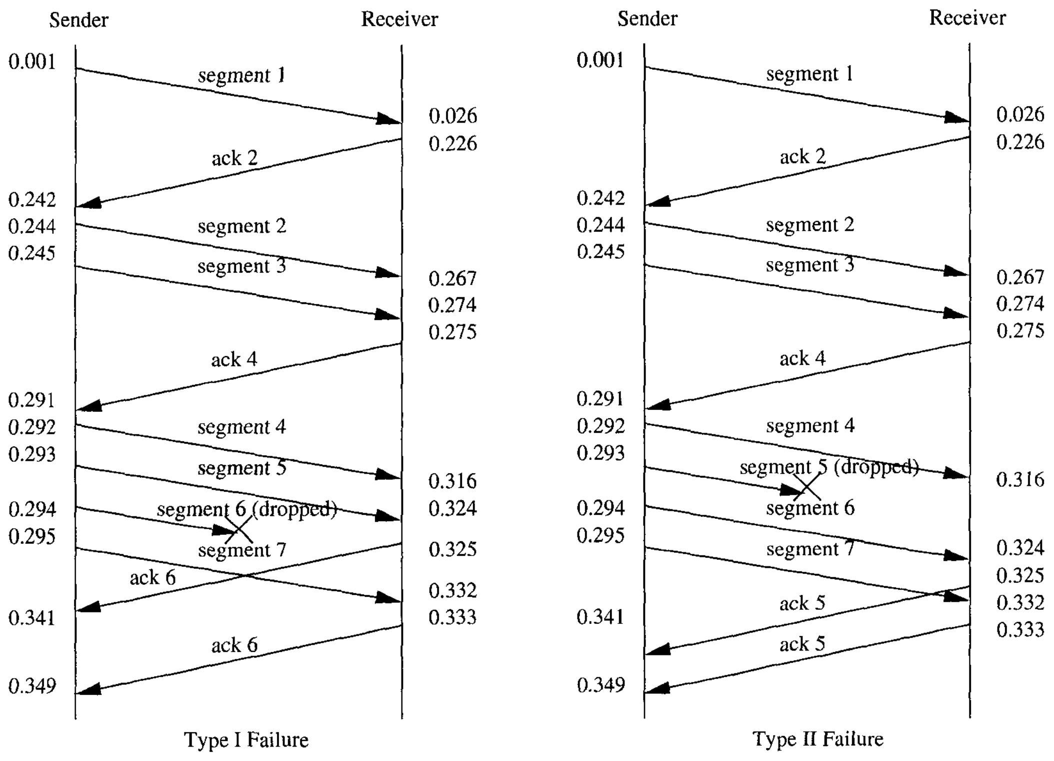


Figure 3.6 Data Flow of Type I Failure and Type II Failure (in seconds)

Figure 3.6 presents the initial data flow of two scenarios to illustrate the difference between Type I failure and Type II failure. The instance where the sixth segment is dropped is a Type I failure, and that where the fifth segment is dropped is a Type II failure. Eq. 3.6 above applies to Type I failure. For Type II failures, Eq. 3.6 must be modified slightly, decreasing $n_{C,S}$ by one to account for the segment triggering the ACK for the previously-unacknowledged segment. Thus, we have:

$$n_{C,S} = \lfloor rwnd/2 \rfloor + 4 \qquad \text{(Type I failure)}$$

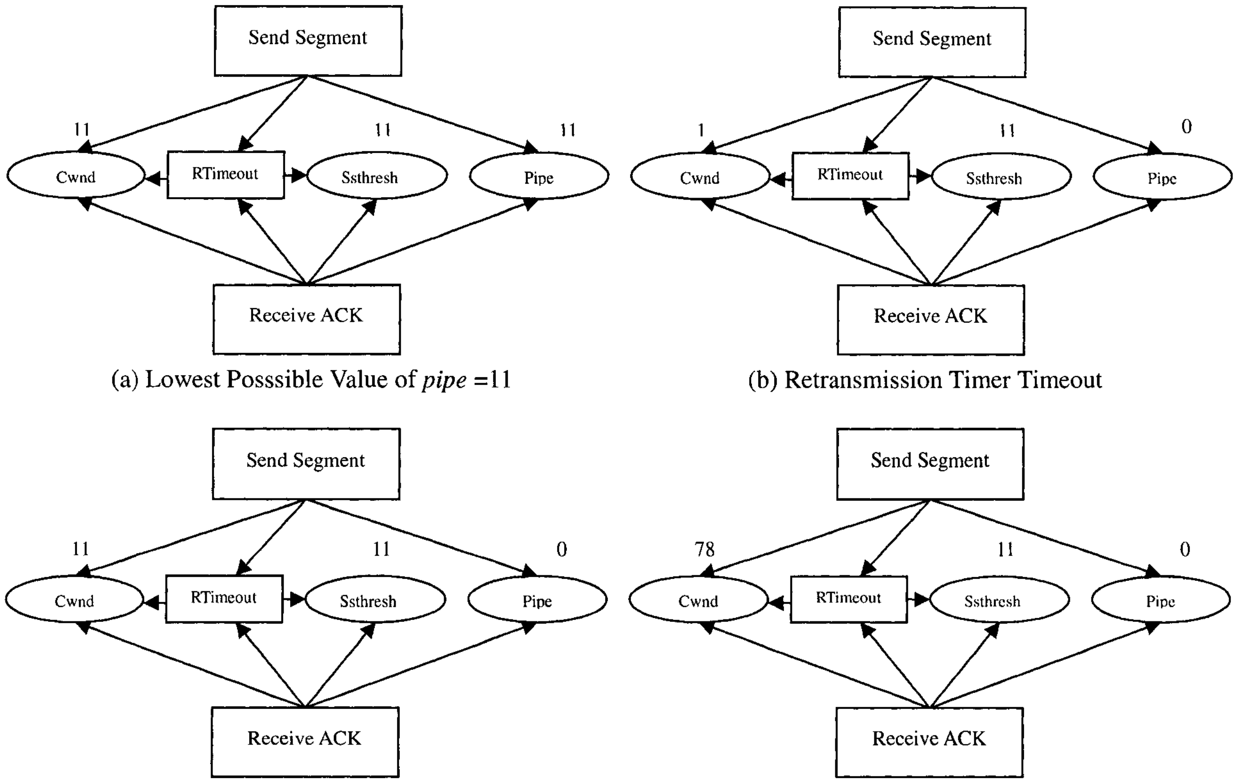
$$n_{C,S} = \lfloor rwnd/2 \rfloor + 3 \qquad \text{(Type II failure)} \qquad (3.7)$$

The instance of the file transfer example in which 14 segments are lost illustrates a Type II failure, so $n_{C,S}$ is 14. If $n_{C,S}$ or fewer segments within the transmission window are dropped, there are still many segments that arrive at the receiver, triggering a large number of duplicate ACKs. These duplicate ACKs, possibly together with the Partial ACK due to the first retransmission when TCP transitions into Fast Retransmit/Fast Recovery, are enough for the sender to make *pipe* less than *cwnd*. Then the sender can retransmit other dropped segments after retransmitting the earliest unacknowledged segment when it first switches into Fast Retransmit/Fast Recovery.

In the instance that 14 segments are dropped, Figure 3.5 (c) shows the state of the Petri net when *pipe* first becomes less than *cwnd*. After this, the sender keeps transmitting segments until a non-duplicate ACK arrives acknowledging all data that was outstanding when Fast Retransmit/Fast Recovery was entered. Then TCP exits Fast Retransmit/Fast Recovery. Because *cwnd* is equal to *ssthresh* after TCP gets back to its normal state, TCP is actually in Congestion Avoidance after Fast Retransmit/Fast Recovery. Figure 3.5 (d) illustrates the state of the Petri net at this moment. TCP stays in Congestion Avoidance until the file transfer is finished, and the final state of the Petri net is shown in Figure 3.5 (e). In the 10 MB file transfer example, *cwnd* has a final value of 78 instead of 125 due to the impact of the network failure. In the instances where from 1 to 13 segments are dropped, *cwnd*, *ssthresh* and *pipe* change in a similar fashion to the instance of 14 dropped segments. Thus from 1 dropped segment to 14 dropped segments, there is no significant difference in terms of TTI.

Case 3: More Than $n_{C,S}$ Segments Dropped, Timeout Occurs

If more than $n_{C,S}$ segments in the transmission window are dropped and there are still at least three (in Type I failure) or four segments (in Type II failure) remaining in the window, SACK TCP can still transition into Fast Retransmit/Fast Recovery and retransmit the earliest unacknowledged segment. This is because the segments left in the transmission window can still trigger at least three duplicate ACKs. However, these duplicate ACKs together with the ACK due to the first retransmission will never make *pipe* less than *cwnd*.



(a) Lowest Posssible Value of *pipe* =11

(b) Retransmission Timer Timeout

(c) *cwnd* Equals *ssthresh*

(d) Final State

Figure 3.7 State Changes for 15 Dropped Segments

We use the instance of the file transfer example where 15 segments are dropped to illustrate SACK TCP resilience in Case 3. The protocol goes through the same process as in Case 2 until the transition to Fast Retransmit/Fast Recovery. However, after the transition to Fast Recovery *pipe* cannot become less than *cwnd*. For the instance where 15 segments are dropped, Figure 3.7 (a) shows the state of the Petri net when *pipe* reaches its minimum value of 11. This is equal to, but not less than the current *cwnd* value of 11. Thus the sender will not retransmit other dropped segments after retransmitting the earliest unacknowledged segment. It will simply wait until timeout occurs. Then TCP will transition into Slow Start. At that time, *cwnd* is set to one and then starts increasing. In these circumstances, the transition **RTimeout** will fire. As a result, *cwnd* is changed to 1 and *pipe* is set back to 0. The state of the model at that point is presented in Figure 3.7 (b).

After this, TCP starts from the beginning, following the process presented in Case 1. That is, *cwnd* first increases exponentially. After *cwnd* becomes equal to *ssthresh*, illustrated by Figure 3.7 (c), TCP switches into Congestion Avoidance. After the failure, *ssthresh* is 11 instead of the original 45. In the file transfer example, TCP remains in Congestion Avoidance until the session is terminated. The final state is presented in Figure 3.7 (d).

The overall transfer time increases significantly in this case because timeout takes place. This is why we call $n_{C,S}$ the "critical" number of segments dropped. At or below this number, TCP will not experience timeout. However, when more than $n_{C,S}$ segments are lost, TCP is doomed to timeout and suffer a significant performance degradation due to timeout. In addition, setting *cwnd* back to 1 after the failure also impacts TCP performance after the failure because the available bandwidth is not fully utilized. In short, SACK TCP suffers significantly in this scenario. In the file transfer example, this sequence of events occurs in the instances where 15 to 19 segments are dropped.

If less than three (in Type I failure) or four (in Type II failure) segments remain in the transmission window, Fast Retransmit/Fast Recovery will not occur because there will be less than three duplicate ACKs. This also leads to timeout. When the retransmission timer expires TCP will transition into Slow Start. In this scenario, TCP switches into the timeout state shown by Figure 3.7 (b) directly from the state shown in Figure 3.5 (a), without going through the *pipe* decrease process illustrated by Figure 3.7 (a). During the rest of the file transfer, TCP experiences the same sequence of events after timeout occurs as in the instances of 15 to 19 dropped segments. In the file transfer example, the instances where 20 to 23 segments are dropped belong to this category. There is no significant difference in terms of TTI between the instances of 15 to 19 dropped segments, and those where 20 to 23 segments are dropped. This is because timeout is the main factor influencing transfer time increase. Hence, although TCP does not experience Fast Retransmit/Fast Recovery in the instances of 20 to 23 dropped segments, it still suffers timeout and we categorize these instances as Case 3.

For TCP, resorting to timeout is very costly. RFC 2988 [35] specifies that the value of the timeout should be at least 1 second. If SACK TCP does not experience timeout it can recover relatively quickly. There is some performance degradation in a non-timeout case due to the reduction in *cwnd*. However this is not very serious compared to the impact of the session having to wait at least 1 second for timeout. Thus, the increase in TTI from Case 2 to Case 3 is large.

Case 4: Retransmitted Segment Also Dropped

If the network failure lasts long enough that the segment retransmitted due to timeout is also dropped, performance decreases again. This is because when the first retransmitted segment is sent out, the retransmission timer has been doubled [35]. If the retransmission fails, the sender will wait for twice the previous RTO before timing out and retransmitting the earliest unacknowledged segment a second time. This corresponds to the instance of the file transfer example in which 24 segments are dropped. In this case, TCP first switches from Congestion Avoidance into Slow Start because of the first timeout. It stays in Slow Start until the second timeout occurs. Then TCP retransmits the earliest unacknowledged segment again. If this second retransmission is successful, after receiving the acknowledgement for the second retransmission, *cwnd* starts increasing. Then the session follows the process discussed in Case 1 until the end of the transfer. In terms of state transitions, the behavior of the file transfer in Case 4 is very similar to the instances of 20-23 dropped segments. The difference is that TCP has to wait much longer in Case 4 before *cwnd* starts increasing exponentially. Waiting for twice the previous RTO increases TTI significantly. If the second retransmission also fails, the timeout doubling process goes on until TCP gives up on the session.

In the file transfer example, any segments discarded after the first 23 are actually outside the current transmission window. However, because they are not normal segments but segments retransmitted due to timeout, and their loss has a significant impact on SACK TCP resilience, we have documented these instances as well.

Figure 3.8 presents the performance of the SACK TCP file transfer example as a graph of

TTI as a function of the number of dropped segments. From 1 to 14 dropped segments, TTI does not increase much. At 15 dropped segments, TTI increases dramatically. Once 24 segments are dropped, TTI increases dramatically again. The large increases in TTI correspond to the instances where first one and then two timeouts occur.

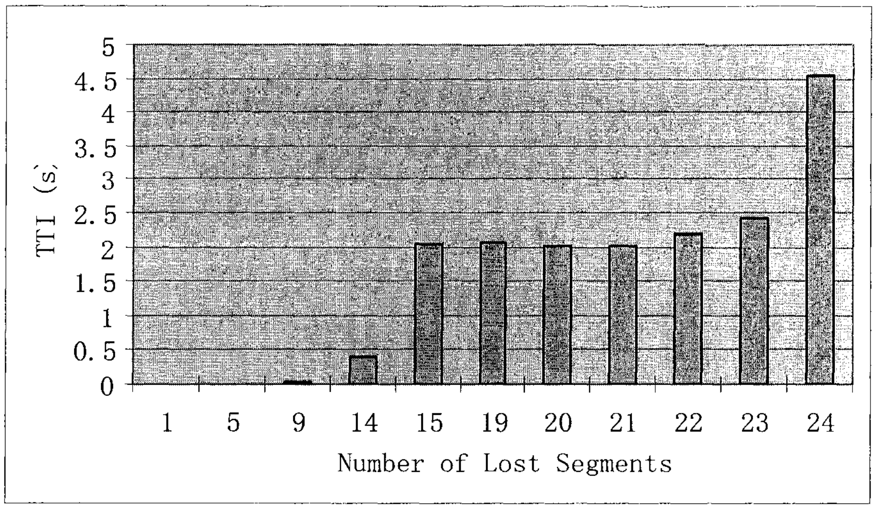


Figure 3.8 Performance during Congestion and Failure

3.1.2.2 Network Congestion and SACK

To study the impact of network congestion on SACK TCP, the Segment Discarder can be configured to simulate losses of nonconsecutive segments. As with consecutive segment loss we consider those transmission windows in which the first segment is dropped. In the file transfer example, the transmission window has room for 23 segments. If we study only those instances where the first segment is dropped, there are 2^{22} or about 4 million possible different sequences of segments being either successfully received or dropped. To study this large set of possibilities, we use the Petri net model in simulation mode to sample the behavior of the protocol. For the file transfer example, we ran the model ten times for each possible number of nonconsecutive discards, from 1 to 23. Each time, the segment discarder generated a different drop pattern. These runs could more accurately be characterized as "not necessarily consecutive discards" because it is still possible for consecutive segments to be discarded. However, the loss pattern does allow for nonconsecutive discards, whereas the earlier study of link failure enforced strictly consecutive losses. Table 3.3 illustrates ten runs during each of which 5 segments are discarded. In the first run, segments 1, 4, 10, 16, 22 are dropped.

Table 3.3 Simulation Runs

	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10
1st Drop	1	1	1	1	1	1	1	1	1	1
2nd Drop	4	3	3	3	3	3	3	3	3	3
3rd Drop	10	8	4	4	6	5	17	8	5	10
4th Drop	16	14	5	10	11	10	18	13	14	11
5th Drop	22	21	14	13	22	19	20	15	22	21

Table 3.4 Comparison between Network Failure and Congestion

Cases	5 Drops	9 Drops	14 Drops	15 Drops	19 Drops
TTI Mean (Network Congestion)	0 s	0.000479 s	0.086865 s	1.911193 s	1.917387 s
TTI Variance (Network Congestion)	0	2.29E-06	0.000171	8.32E-05	1.15E-05
TTI (Network Failure)	0 s	0 s	0.082161 s	1.961781 s	1.918702 s
Cases	**20 Drops**	**21 Drops**	**22 Drops**	**23 Drops**	**24 Drops**
TTI Mean (Network Congestion)	1.928925 s	1.926476 s	1.927976 s	2.139896 s	4.539896 s
TTI Variance (Network Congestion)	2.80E-05	1.05E-08	1.07E-05	0	0
TTI (Network Failure)	1.926474 s	1.926474 s	1.926474 s	2.139896 s	4.539896 s

When subjected to these loss patterns, the numerical results show that SACK TCP behaves almost the same during congestion as during a network failure, both in terms of Petri net transitions and in terms of TTI. In the file transfer example, from 1 to 14 dropped segments, Fast Retransmit/Fast Recovery is triggered, but timeout does not occur. As a result, TTI does not change much. When it comes to 15 drops, timeout takes place and TTI increases significantly. After that, TTI remains nearly constant until 24 segments are dropped whereupon TTI increases significantly again. Hence, whether the dropped segments are consecutive or not does not matter – it is the absolute number of dropped segments that is important. This is because the segments in the current transmission window following the first dropped segment will all lead to duplicate ACKs. As long as the number of segments that are transmitted successfully remains the same, there will be the same number of duplicate ACKs arriving at the sender. As a result TCP behaves in the same manner for congestion events as it does during network failures, as long as the number of dropped segments is the same. Table 3.4 summarizes the sample mean and variance of TTI in the file transfer example for situations of network congestion. The TTI for the network failure scenario is also included for comparison.

3.2 NewReno TCP Resilience

In Section 3.1, we talked about SACK TCP resilience in the case of network failures and congestion. In Section 3.2, 3.3, and 3.4, we are focused on the reaction of TCP to network failures. Specifically, we present the general resilience performance of NewReno and Reno TCP in Section 3.2 and 3.3, respectively; the detailed resilience performance of SACK, NewReno, and Reno TCP in the scenario of network failures is discussed in Section 3.4. In this section, we describe the details of our simulations and present NewReno TCP resilience.

3.2.1 Simulation

We carried out simulations using OPNET [66] to study the resilience of SACK, NewReno, and Reno TCP in the case of network failures. The experimental results illustrated the resilience behavior of SACK, NewReno, and Reno TCP. As mentioned previously, the results were also used to validate our Petri net model of SACK TCP. For this study, a client and server are connected across a continental-scale network. Each node is connected to a local router via a high-speed LAN link. The local routers are connected to the core network via access links. Three access link rates are commonly used in real-life systems: DS0 (64 Kbps), DS1 (1.544 Mbps) and OC-3c (155 Mbps). Based on the fact that servers are usually connected to the Internet via high-speed links while client-side access link rates vary a lot, our simulation fixes the server-side access at OC-3c and varies the client-side access link from DS0 to DS1 and OC-3c. The core network in our simulation has an NSFNET-like topology shown in Figure 3.9. Core routers (Cisco 12008) are connected via OC-192, which is common in backbone networks nowadays. The client resides in Palo Alto and the server is located at Princeton.

As shown in Figure 3.9, a packet discarder model, used to simulate outages, is on the link connecting Salt Lake City to Palo Alto. We can specify either the number of packets to be dropped or a certain time period during which all packets are dropped. Our experiments simulate a unidirectional failure of packets going from Salt Lake City to Palo Alto (i.e. in the server-to-client path). Packets traveling the other way get to their destination safely. A unidirectional failure would be unusual in a transport network. However, this assumption was made by many network researchers to reflect the reality of today's Internet: routes for IP packets are often asymmetric [12]. Thus a failure in the underlying network will often only affect a session in one direction.

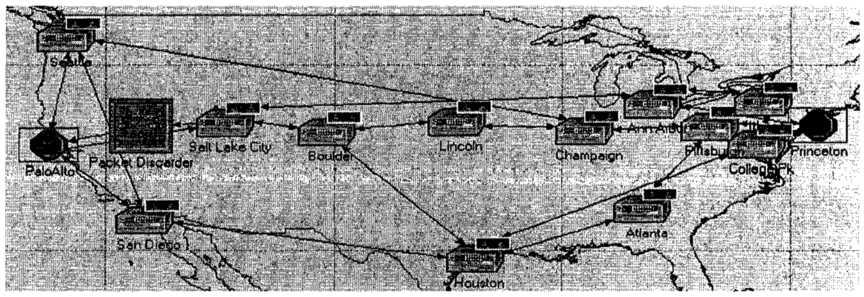


Figure 3.9 Core Network Topology

There is only one routing domain in our simulations, and the NSFNET-like topology is relatively old. However, this thesis focuses on the TCP-layer view of failures. That is, this thesis tries to find out, in the absence of any compensating mechanisms, how TCP congestion control mechanisms will react to outages. This first-step experiment generated many valuable results. We believe it is an important first step in understanding TCP resilience thoroughly.

The receiver window *rwnd* advertised by the receiver plays an important role in TCP performance. During a TCP session, the sending TCP continuously compares the outstanding unacknowledged traffic with *cwnd* and *rwnd.* Whenever the outstanding traffic is less than the smaller of these two variables by at least one SMSS (sender maximum segment size), the sender will send out some segments if there are any waiting to be sent. Note that *rwnd* is different from the receive buffer because *rwnd* only represents the space that has not been occupied in the receive buffer. However, for simplicity, we assume that any data is read immediately after it is written into the buffer. Thus, in our research, *rwnd* is a constant that is always equal to the receive buffer. Generally the receive buffer size (*rbuff*) is set as:

$$rbuff = \text{bandwidth} \ * \ \text{round-trip-time} = r * \tau \qquad (3.8)$$

where r stands for bandwidth and τ is round-trip-time (RTT). Namely, in TCP, *rwnd* should be set using Eq. 3.8. This is commonly called the bandwidth-delay product [1].

The TCP session in our simulation must be long enough to test scenarios with varying failure durations. We chose FTP as the application-layer protocol and made the transmitted file large enough to fulfill this requirement. For DS0, DS1 and OC-3c client-side access links, we used 5 MB, 10 MB and 20 MB files, respectively. In reality, the duration of TCP flows covers a very large range. Although there are many short flows in terms of number, long-running flows account for up to 50% of all traffic [82]. Our research focuses on long-running TCP flows.

3.2.2 NewReno TCP Resilience Performance

We use a scenario with DS1 client access and 32 KB receiver window as a typical example to illustrate the general behavior of NewReno TCP in the case of network failures. Reno TCP general behavior in the DS1-32KB case will be presented in Section 3.3. Note that actually the general behavior of SACK TCP has already been presented using Petri nets in Section 3.1. In our simulations, SMSS is 1460 bytes; *cwnd* and ssthresh were initialized to 1460 bytes and 64KB, respectively. By 32KB or 64KB, we mean a multiple of SMSS that is just above 32KB or 64KB. For clarity, we consider 1460 byte segments as the data units. Thus, cwnd, rwnd, and ssthresh were equal to 1, 23, and 45, respectively.

In our example, a long NewReno TCP session starts at time 0 and the Packet Discarder begins to drop packets at 30 seconds, as shown in Figure 3.10. The four curves in Figure 3.10 illustrate the changes in the sender's congestion window over time in four different cases. Before the failure starting point (marked by "X"), the four curves overlap each other because during that period they describe essentially the same conditions. After point "X", they split into four different curves.

When the TCP session starts, the congestion window is equal to one and TCP is in Slow Start. *cwnd* increases exponentially as the sender receives acknowledgements, until *cwnd* equals *ssthresh* (initially *ssthresh* is 45). The sparse points at the beginning of the curve correspond to this Slow Start period. Then TCP transitions into Congestion

Avoidance during which *cwnd* increases by one every RTT. This increase is comparatively slow and as a result the points for the Congestion Avoidance period are very close together. Without a network failure, TCP stays in Congestion Avoidance until the file is completely transmitted and the TCP session is terminated. This is shown by the curve labeled "0 Drop".

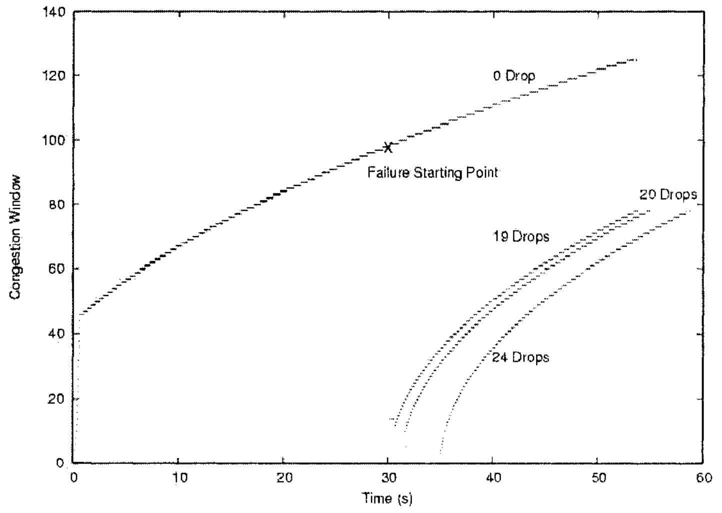


Figure 3.10 Congestion Window vs. Transfer Time (NewReno DS1-32K Case)

Now we consider the case in which some segments are lost during a failure. In this case, after the lost segments, the client receives subsequent segments over the restored link. As the result, the sender gets three duplicate ACKs. Then it transitions into Fast Retransmit/Fast Recovery and retransmits the earliest unacknowledged segment. It also sets *ssthresh* and *cwnd* to $\lfloor rwnd / 2 \rfloor$ and ($\lfloor rwnd / 2 \rfloor + 3$), respectively. If we use $n_{C,NR}$ to denote the critical number of lost segments when there are just enough subsequent surviving segments in the window of data to trigger three duplicate ACKs, it is easy to know usually the critical number is equal to (*rwnd* - 3). Due to the TCP acknowledging mechanism illustrated in Section 3.1.2.1, we can get the following formula:

$$n_{C,NR} = \begin{cases} rwnd - 3 & \text{in Type I Failure} \\ rwnd - 4 & \text{in Type II Failure} \end{cases} \qquad (3.9)$$

Note that the NewReno DS1-32K example also illustrates a Type II failure, thus $n_{C,NR}$ is 23 - 4 =19.

If less than $n_{C,NR}$ segments in a window of data are lost, enough surviving segments can arrive at the receiver and trigger enough duplicate ACKs to make TCP transition into Fast Retransmit/Fast Recovery. In this case, the earliest unacknowledged segment is retransmitted and the retransmission leads to a partial ACK. The partial ACK will then make the sender retransmit the earliest unacknowledged segment at that moment. This retransmitted segment will result in another partial ACK, and thus leads to another retransmission. This process goes on until a non-duplicate ACK arrives acknowledging all data that was outstanding when TCP transitioned into Fast Retransmit/Fast Recovery, then TCP switches into Congestion Avoidance by setting *cwnd* back to *ssthresh*. We should note that each time the sender receives a partial ACK, it does one retransmission and thus recovers one lost segment. Namely, it takes NewReno TCP a whole RTT to recover one lost segment. Thus in a sense, RTT determines the final TTI value. If RTT is comparatively long, TTI increases dramatically with the number of lost segments; otherwise, TTI almost remains unchanged. In the NewReno DS1_32K case, RTT is relatively small, thus TTI does not increase much. In this case $n_{C,NR}$ is 19, so from 1 lost segment to 19 lost segments, all the curves are similar. For clarity, we only include the 19-drop curve in Figure 3.10.

SACK TCP has a different mechanism to deal with partial ACKs. In Section 3.1.2.1, we have mentioned that *pipe* is decremented by one for each additional duplicate ACK, but it is decreased by two rather than one for each partial ACK. This additional decrease in *pipe* results in a faster recovery process: one partial ACK leads to two retransmissions. The two retransmissions will trigger another two partial ACKs and eventually will lead to another four retransmissions. This process goes on until a non-duplicate ACK arrives acknowledging all data that was outstanding when TCP transitioned into Fast Retransmit/Fast Recovery. Hence, within one RTT, usually many more lost segments can be recovered with SACK TCP than with NewReno TCP. This is why with SACK TCP, TTI does not increase much when less than $n_{C,S}$ segments within one window are lost, regardless of the length of RTT. In contrast, the TTI of NewReno is influenced by RTT in this situation.

On the other hand, if more than $n_{C,NR}$ segments in a window of data are dropped, Fast Retransmit/Fast Recovery will not occur because there will not be enough duplicate ACKs. This leads to timeout. When the retransmission timer expires TCP will transition into Slow Start and retransmit the first lost segment. In this scenario, timeout plays the major role in terms of TTI, and thus the overall transfer time does not increase much with the number of lost segments. In the NewReno DS1_32K case, from 20 lost segments to 23 lost segments, all the curves are similar. We only include the 20-drop curve in Figure 3.10 for clarity.

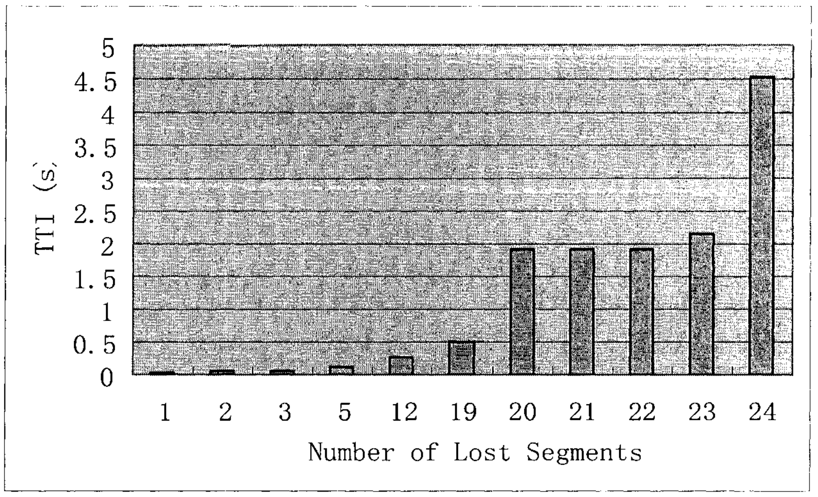


Figure 3.11 TTI vs. No. of Lost Segments (NewReno DS1-32K case)

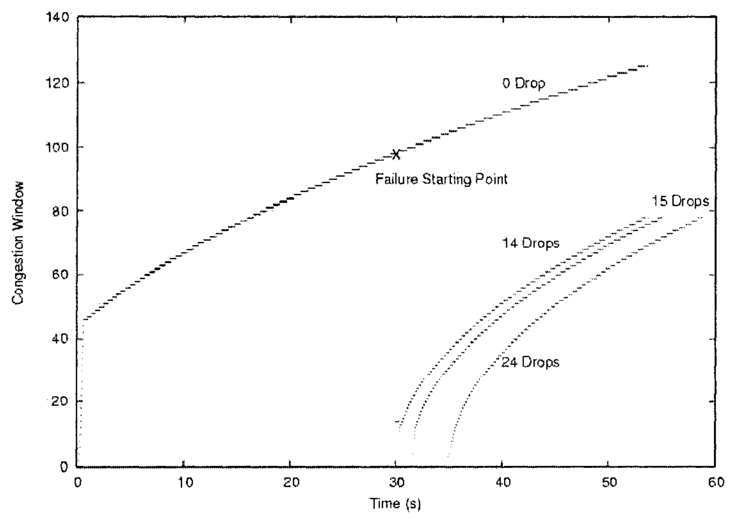


Figure 3.12 Congestion Window vs. Transfer Time (SACK DS1-32K Case)

If the network failure lasts long enough so that the segment retransmitted due to timeout is also dropped, NewReno TCP will experience the same doubling calculation that is illustrated in Section 3.1.2.1. A similar 24-drop curve is included in Figure 3.10. For clarity, we did not include the curves of 25, 26, etc. dropped segments. Figure 3.11 illustrates the overall trend by presenting TTI changes vs. the number of dropped segments. We presented the figure of "TTI changes vs. the number of dropped segments" for SACK TCP in Section 3.1.2.1. For comparison and completeness, the way that *cwnd* changes with transfer time in the case of SACK TCP is summarized in Figure 3.12.

3.3 Reno TCP Resilience

For Reno TCP, again, we use a similar experimental setup. As shown in Figure 3.13, a long Reno TCP session also starts at time 0 and the Packet Discarder begins to drop packets at 30 seconds. Before the failure starting point (marked by "X"), the four curves overlap; after point "X", they split into four different curves. Similarly, the curve labeled "0 Drop" is the same one presented in Section 3.3.1.A. Without network failures SACK and Reno TCP behave in the same fashion.

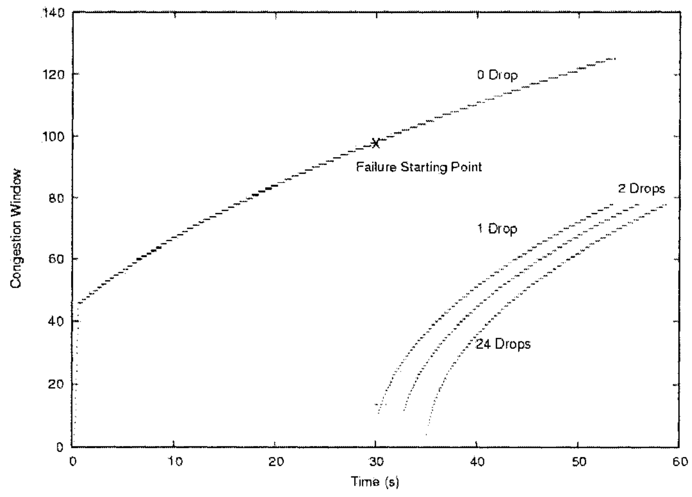


Figure 3.13 Congestion Window vs. Transfer Time (Reno DS1-32K Case)

Now we consider the case in which one segment is lost during a failure. As a result of the client receiving subsequent segments, TCP transitions into Fast Retransmit/Fast Recovery after the sender gets three duplicate acknowledgements. The sender retransmits the earliest unacknowledged segment, sets *ssthresh* to $\lfloor rwnd / 2 \rfloor$ and sets *cwnd* to ($\lfloor rwnd / 2 \rfloor + 3$). For each additional duplicate ACK, *cwnd* increases by one. This process goes on until the retransmitted segment reaches the receiver and a new ACK acknowledgement for all outstanding segments is received by the sender. At that point *cwnd* is set back to the current *ssthresh*, and Congestion Avoidance starts again because *cwnd* is now equal to *ssthresh*. In short, Reno TCP can usually recover effectively from the loss of one segment. The overall transfer time does not increase much in this case.

Losing two segments makes a difference. Before the ACK for the first retransmission is received by the sender, the recovery process is similar to that when only one segment is lost. As before, the ACK for the first retransmission only acknowledges the first lost segment. This segment has been retransmitted due to Fast Retransmit/Fast Recovery, while the second lost segment has not been retransmitted yet. The Fast Retransmit/Fast Recovery algorithm in Reno TCP assumes that only one segment has been lost, so the sender does not immediately retransmit the second lost segment. Because two segments have been lost, TCP will eventually time out, and switch into Slow Start. TCP will have to retransmit the earliest segment that has not been acknowledged, which in this case is the second lost segment. This timeout results in the large gap between the 1-drop and 2-drop curves in Figure 3.13.

In the case of more than two lost segments, if there are still three or more non-retransmitted segments following the lost segments that arrive at the receiver, enough duplicate ACKs will reach the sender to trigger Fast Retransmit/Fast Recovery. In this scenario, losing more than two segments leads to the same recovery process as losing two segments. On the other hand, if less than three segments follow the lost segments, Fast Retransmit/Fast Recovery will not occur because there will not be enough duplicate ACKs. When the retransmission timer expires, TCP transitions into Slow Start and retransmits the first lost segment. Although TCP experiences different transitions than in the case of two lost segments, total transfer time does not increase dramatically because timeout is the main factor. Hence, from 3 lost segments to 23 lost segments, all the curves are similar to the 2-drop curve in Figure 3.13.

If the network failure lasts long enough so that the segment retransmitted due to timeout is also dropped, things change again. This is because when the retransmitted segment is sent out, the retransmission timer has been doubled. If the retransmission fails, the sender will wait for twice the previous RTO before timing out and retransmitting the earliest unacknowledged segment again. This corresponds to the 24-drop curve in Figure 3.13. If the repeated retransmission does not succeed, the sender has to wait for four times the original RTO to retransmit a third time. This process goes on until TCP gives up this connection. For clarity, we did not include the curves of 25, 26, etc. segments dropped, but it is not difficult to imagine what they should look like in Figure 3.13. We can observe the Reno TCP general behavior in terms of TTI in Figure 3.14.

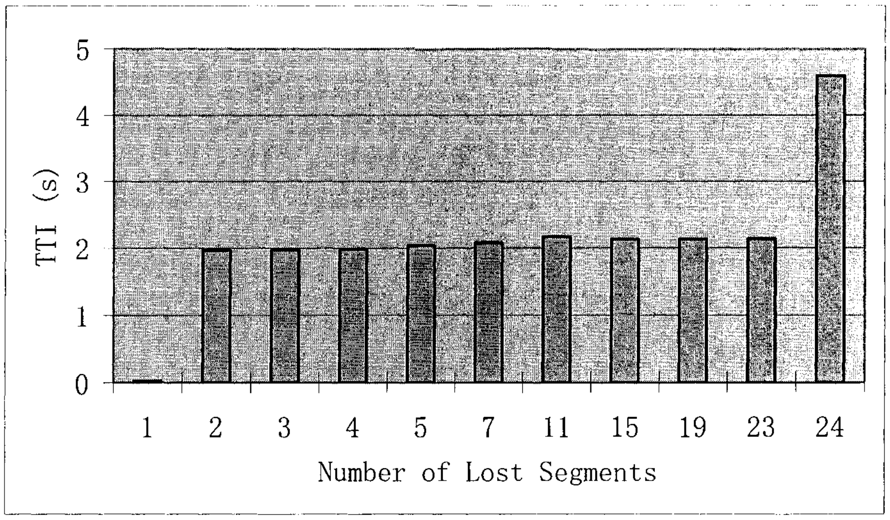


Figure 3.14 TTI vs. No. of Lost Segments (Reno DS1-32K Case)

3.4 Several Details of TCP Resilience

The bandwidth-delay product, rτ, is commonly used to size *rwnd*. In our simulations, the RTT for DS0, DS1 and OC-3c access is 210ms, 41ms and 26ms respectively, so rτ has values of 1680, 7913 and 505440 bytes respectively. For each access link rate, we experimented with 8 different *rwnd* sizes, from 8 KB to 1024 KB. By 8 KB, we mean a multiple of SMSS that is just above 8 KB. For example, in our simulation, SMSS is 1460 bytes, so by 8 KB, we mean 1460 * 6= 8760 bytes.

3.4.1 SACK TCP Details

We have demonstrated that losing less than $n_{C,S}$ segments typically does not increase SACK TCP transfer time significantly. Losing ($n_{C,S}$ + 1) segments makes a difference, and subsequent losses have little impact until it comes to the loss of the retransmitted copy. To link the number of lost segments to outage duration, we define SACK Level-1 Fault Tolerance Time ($\tau_{1,S}$) as the period from the moment network failure occurs to the moment just before the segment following the dropped $n_{C,S}$ segments arrives. We define SACK Level-2 Fault Tolerance Time ($\tau_{2,S}$) as the period from the moment network failure occurs to the moment just before the copy retransmitted due to timeout arrives. Thus $\tau_{1,S}$ is the time during which ($\lfloor rwnd / 2 \rfloor$ + 4) or ($\lfloor rwnd / 2 \rfloor$ +3) segments pass the failure point. In our simulations, this is mostly influenced by the client access rate, which is essentially the bandwidth of "r" used to calculate the bandwidth-delay product, so that:

$$\tau_{1,S} = \begin{cases} ((\lfloor rwnd/2 \rfloor + 4) * \text{SMSS})/r & \text{in Type I Failure} \\ ((\lfloor rwnd/2 \rfloor + 3) * \text{SMSS})/r & \text{in Type II Failure} \end{cases} \qquad (3.10)$$

Obviously, $\tau_{1,S}$ increases with *rwnd.* If *rwnd* is large enough so that we can approximate $\tau_{1,S}$ as follows:

$$\tau_{1,S} = ((\lfloor rwnd/2 \rfloor) * \text{SMSS})/r \qquad (3.11)$$

then $\tau_{1,S}$ approximately doubles as *rwnd* doubles. This is illustrated in Figure 3.15, 3.16 and 3.17. $\tau_{2,S}$ is not as straightforward because it is slightly greater than RTO, and RTO is influenced by many factors [1]. RTO usually increases with *rwnd,* and has a minimum value of 1 second. Thus, when RTO is greater than 1 second, $\tau_{2,S}$ increases with *rwnd.* In our experiments, when the client access is DS0 or DS1, the RTO is greater than 1 second, resulting in $\tau_{2,S}$ increasing with *rwnd.* For example, $\tau_{2,S}$,corresponding to the second significant TTI increase, is around 3.8 seconds in the case of DS0-8K and it is about 6.4 seconds in the scenario of DS0-16K. The details are illustrated in Figure 3.15 and 3.16. When RTO is at its minimum of 1 second, $\tau_{2,S}$ does not change much and is independent of *rwnd.* In our experiments, when the client access is OC-3c, the RTO is fixed at 1 second, leading to an independent $\tau_{2,S}$. Figure 3.17 shows that no matter how large *rwnd* is, the second significant TTI increase always takes place at around 1.3 seconds. In any case, $\tau_{2,S}$ is always greater than 1 second.

Either $\tau_{1,S}$ or $\tau_{2,S}$ can be chosen as a restoration objective. If the restoration can be finished within $\tau_{1,S}$, the overall transfer time will not increase much in the case of network failures. If the restoration time is in the range of $\tau_{1,S}$ to $\tau_{2,S}$, the overall transfer time is increased but it is guaranteed that the TTI is a fixed value. Other thresholds can be defined on the basis of a third timeout and so on. However, we know that $\tau_{2,S}$ is certainly greater than 1 second. This is already much larger than the de facto target of 50 ms.

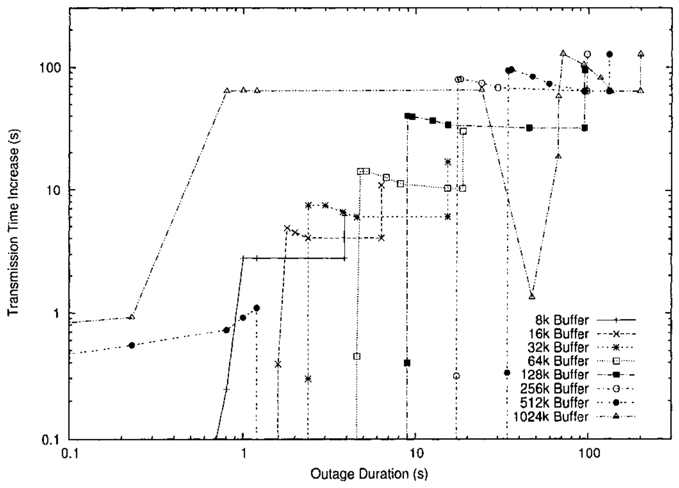


Figure 3.15 TTI vs. Outage Duration (SACK DS0 Access)

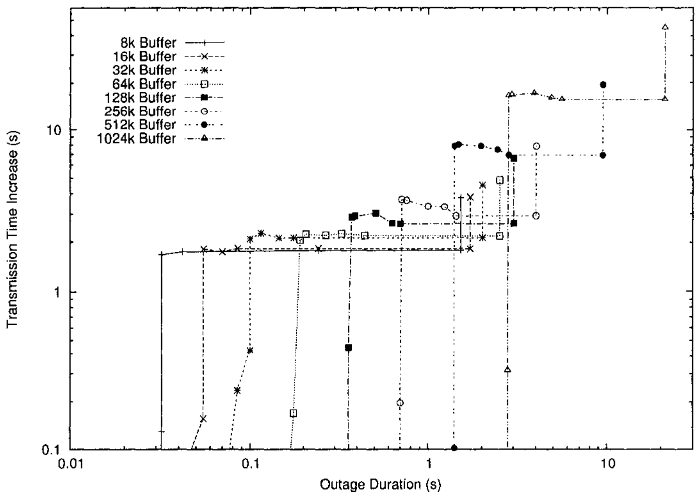


Figure 3.16 TTI vs. Outage Duration (SACK DS1 Access)

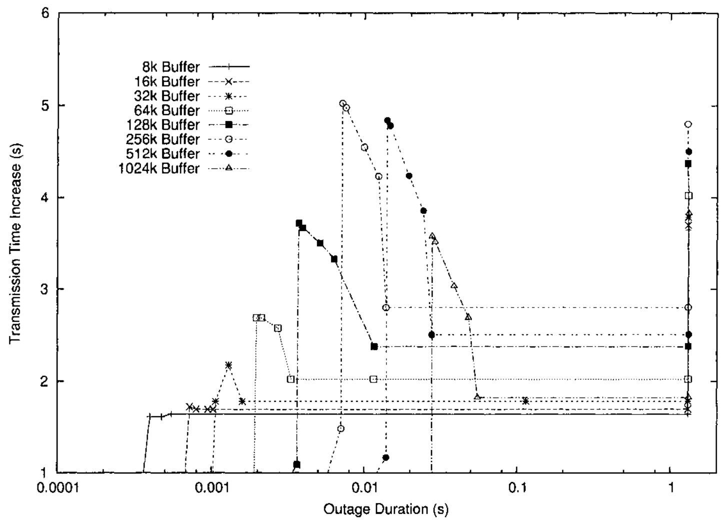


Figure 3.17 TTI vs. Outage Duration (SACK OC-3c Access)

An interesting observation is that TTI is not always greater than outage duration. When *rwnd* is greater than rτ, some segments will be buffered in the network. These buffered segments help keep traffic flowing between the sender and the receiver in the case of network failures. Thus TTI could be shorter than outage duration in some scenarios. We can observe this trend in Figure 3.15 and 3.16.

There are some exceptions to these typical cases. First, the 512 KB and 1024 KB curve in Figure 3.15 illustrate situations in which very large receive buffers lead to a calculated value of RTO that is greater than the TCP-defined maximum of 64 seconds. This puts TCP into Slow Start many times unnecessarily and dramatically changes the normal recovery process. Thus the 512 KB and 1024 KB curve are very irregular. Secondly, in Figure 3.17, we note that a 1024 KB buffer mostly leads to a shorter TTI than does a 512 KB buffer. This is exceptional because generally TTI increases with *rwnd*. However, the bandwidth-delay product for OC-3c access is 505440 B, and after the failure *ssthresh* is set to half the current flight size, which is around 256 KB in the case of a 512 KB buffer. Setting *ssthresh* to a value less than rτ hurts link utilization and leads to a longer TTI. Thirdly, in Figure 3.17, we observe that when outage duration is between $\tau_{1,S}$ and $\tau_{2,S}$, TTI decreases dramatically with outage duration. This is again the result of the large value of rτ. When outage duration is in this range, the sender times out and finally gets into

Congestion Avoidance. In the case of OC-3c access, *cwnd* increases with the number of lost segments (corresponding to longer outage duration) when TCP transitions into Congestion Avoidance. In this scenario, the network connection is not fully utilized after the failure because *cwnd* is always less than $r\tau$, so a larger *cwnd* due to longer failure time leads to shorter TTI. Fourthly, in Figure 3.17, after $\tau_{1,S}$, TTI increases as the *rwnd* increases from 8 KB to 256 KB and it decreases as the *rwnd* increases from 256 KB to 512 KB. We know that in the case of OC-3c access $r\tau$ is 505440 bytes. Hence, the curves for 8 KB to 256 KB are for receiver window sizes of less than $r\tau$ and those for 512 KB to 1024 KB are for sizes greater than $r\tau$. A value for *rwnd* less than $r\tau$ leads to poorer link utilization and so to larger NTT [1]. NTT is the baseline value used to calculate TTI in Eq. 3.1. Thus, we have two different classes in terms of TTI, above and below $r\tau$. They are essentially not comparable.

The receiver window size plays a significant role in SACK TCP resilience. First, from the viewpoint of network link utilization, *rwnd* should be set to at least $2r\tau$ to anticipate the case that one network failure takes place during the transmission. Anything lower results in impaired resilience. This is because even if a timeout occurs, *ssthresh* is still equal to at least $r\tau$ when it is initially set to $2r\tau$. Secondly, the *rwnd* should be set as large as possible in order to increase $\tau_{1,S}$. Finally, when RTO is at its minimum of 1 second, receive buffer size does not affect $\tau_{2,S}$; when RTO is greater than 1 second, larger receive buffers lead to longer $\tau_{2,S}$. We can observe these trends in Figure 3.8, 3.9 and 3.10.

3.4.2 NewReno TCP Details

We have illustrated that when less than $n_{C,NR}$ segments are lost, TTI is affected by RTT. Losing ($n_{C,NR}$ +1) segments makes a difference, and subsequent losses have little impact until it comes to the loss of the retransmitted copy. We define NewReno Level-1 Fault Tolerance Time ($\tau_{1,NR}$) as the period from the moment of network failure to the moment just before the segment following the dropped $n_{C,NR}$ segments arrives. $\tau_{1,NR}$ is the time during which (*rwnd* - 3) or (*rwnd* - -4) segments pass the failure point. In our simulation, it is mostly affected by the client access rate, which is essentially r, so:

$$\tau_{1,NR} = \begin{cases} ((rwnd\text{-}3) * SMSS)/r & \text{in Type I Failure} \\ ((rwnd\text{-}4) * SMSS)/r & \text{in Type II Failure} \end{cases} \tag{3.12}$$

Apparently, $\tau_{1,NR}$ increases with *rwnd*. If *rwnd* is large enough so that we can approximate $\tau_{1,NR}$ as follows:

$$\tau_{1,NR} = (rwnd * SMSS)/r \tag{3.13}$$

then $\tau_{1,NR}$ approximately doubles as *rwnd* doubles. This is illustrated in Figure 3.18, 19 and 20. From Eq. 3.11 and 3.13, we conclude that $\tau_{1,NR}$ is approximately twice as large as $\tau_{1,S}$ when *rwnd* is large.

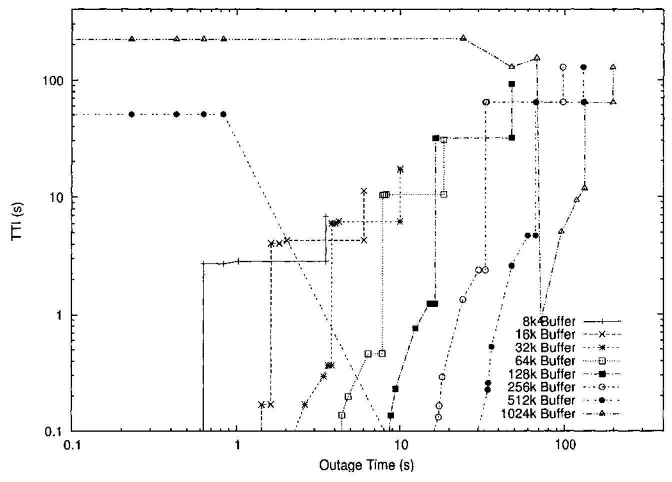


Figure 3.18 TTI vs. Outage Duration (NewReno DS0 Access)

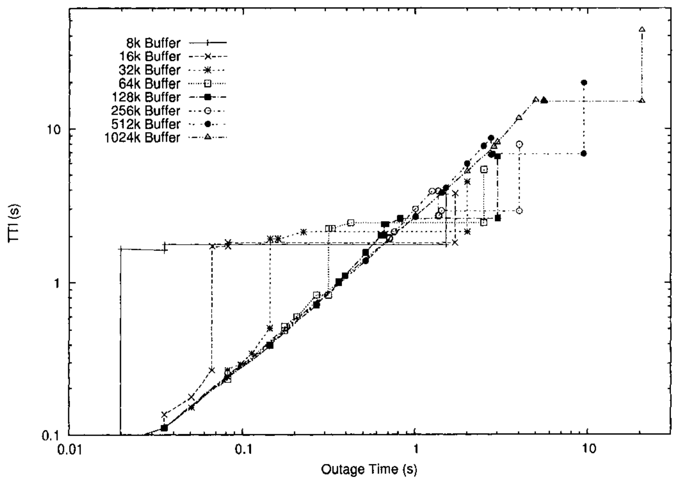


Figure 3.19 TTI vs. Outage Duration (NewReno DS1 Access)

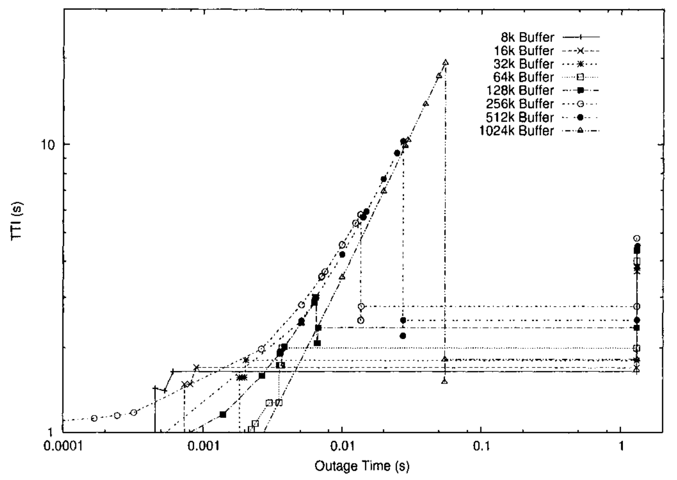


Figure 3.20 TTI vs. Outage Duration (NewReno OC-3c Access)

We define NewReno Level-2 Fault Tolerance Time ($\tau_{2,NR}$) as the period from the moment network failure occurs to the moment just before the copy retransmitted due to timeout arrives. $\tau_{2,NR}$ is essentially the same as $\tau_{2,S}$. So all conclusions about $\tau_{2,S}$ also apply to $\tau_{2,NR}$.

Either $\tau_{1,NR}$ or $\tau_{2,NR}$ can be chosen as a restoration objective. But with NewReno TCP, if the restoration can be finished within $\tau_{1,NR}$, the overall transfer time is influenced by RTT. This is because it takes NewReno TCP one RTT to recover one lost segment, as illustrated in Section 3.2.2. Thus if RTT is relatively small, the overall transfer time does not change much as restoration time increases; otherwise, TTI increases with restoration time. In Figure 3.18, 3.19 and 3.20, we observe that, when restoration time is less than $\tau_{1,NR}$, TTI does not change much in the DS0 scenario, but RTT plays a role in the DS1 scenario and TTI increases dramatically with outage duration in the OC-3c scenario. It is interesting that in the OC-3c scenario, for large receive buffers, restoration times longer than $\tau_{1,NR}$ lead to better resilience (i.e. decreased TTI). In addition, TTI could be less than outage duration in some cases due to large receiver buffer and this can be observed in Figure 3.18 and 3.19.

The exceptions due to large *rwnd*, *ssthresh* halving and insufficient rτ presented in Section 3.4.1 also apply to NewReno TCP and they can be observed in Figure 3.18, 3.19 and 3.20. The exception with SACK TCP that TTI decreases with outage duration when restoration is finished between $\tau_{1,S}$ and $\tau_{2,S}$ in the OC-3c access case does not occur in NewReno TCP because with

NewReno TCP, after $\tau_{1,NR}$ there are only 3 or 4 segments left in the window of data, these segments do not make a significant change to TTI. For receive buffer sizing, all rules illustrated for SACK TCP previously also apply to NewReno TCP.

3.4.3 Reno TCP Details

Previously, we demonstrated that losing one segment typically does not change Reno TCP performance dramatically. However, two losses make a difference, and subsequent losses have little impact until it comes to the loss of the retransmitted copy due to timeout. Considering the absolute time period rather than the number of lost segments, we can find the relationship between TTI and network failure duration. Since two losses make a difference, the segment arrival pattern and not the raw outage duration determines the final result. The mechanism of generating acknowledgements is related to the segment arrival pattern. RFC 2581 [36] suggests that an ACK should be generated for at least every second segment or within 500 ms of the arrival of the first unacknowledged segment (typically it is 200 ms instead). Based on this mechanism, for DS0 access, an ACK usually acknowledges just one segment and thus triggers transmitting only one segment; for DS1 access and OC-3c access, the link is so fast that it is usually two consecutive segments that result in an ACK, which makes the sender transmit segments in pairs. As a result, for DS0 access, there is usually a 200ms gap between segments. For DS1 access and OC-3c access (with buffer sizes of 512 K and 1024 K), segments cluster in pairs, and the gaps between two consecutive clusters are 15ms and 0.15 ms, respectively. If the receive buffer is less than $r\tau$, there will be some additional gaps because the network is not fully utilized and sometimes the sender has to wait when actually there are some segments to be transmitted. OC-3c access with buffer size less than 512 K corresponds to this scenario. Figure 3.21, 3.22 and, 3.23 show the impact of network failure on Reno TCP performance. For DS0 access with buffer size less than 512 K, 200 ms is a critical value. With a 200 ms restoration time, TCP can guarantee recovery with a comparatively low impact; if the restoration time is longer, TCP has a much larger transfer time increase. For DS1 access and OC-3c access, there seem to be no critical points from 15ms to 1s. Since 15 ms is too small for recovery to be accomplished, 1s is recommended.

The exceptions due to large receive buffers, *ssthresh* halving and insufficient $r\tau$ presented in Section 3.4.1 also apply to Reno TCP and they can be observed in Figure 3.21, 3.22, and 3.23. The exception with SACK TCP that TTI decreases with outage duration when restoration is finished between $\tau_{1,S}$ and $\tau_{2,S}$ in the OC-3c access case does not apply to Reno TCP because there is no Level-1 Tolerance time in Reno TCP.

To size *rwnd* properly, first of all, *rwnd* should be set to at least $2r\tau$ to fully utilize network link capacity in the case of network failure. Secondly, when RTO is at its minimum of 1 second, *rwnd* does not affect resilience; when RTO is greater than 1 second, larger receive buffers lead to better resilience performance. We can observe these trends in Figure 3.21, 3.22, and 3.23.

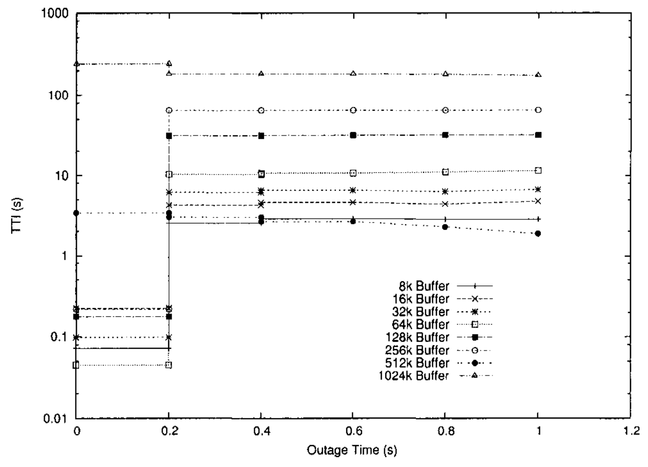


Figure 3.21 TTI vs. Outage Duration (Reno DS0 Access)

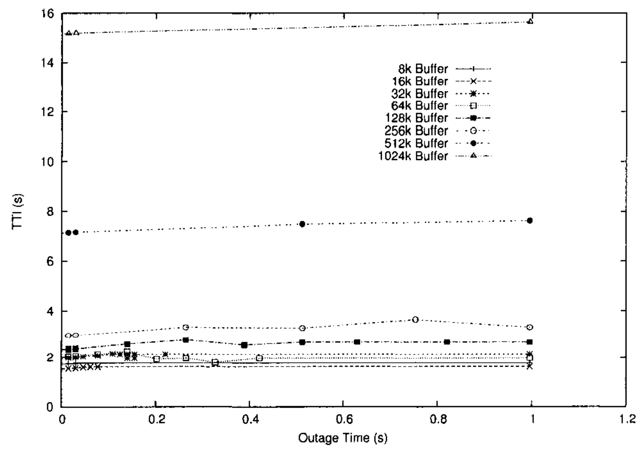


Figure 3.22 TTI vs. Outage Duration (Reno DS1 Access)

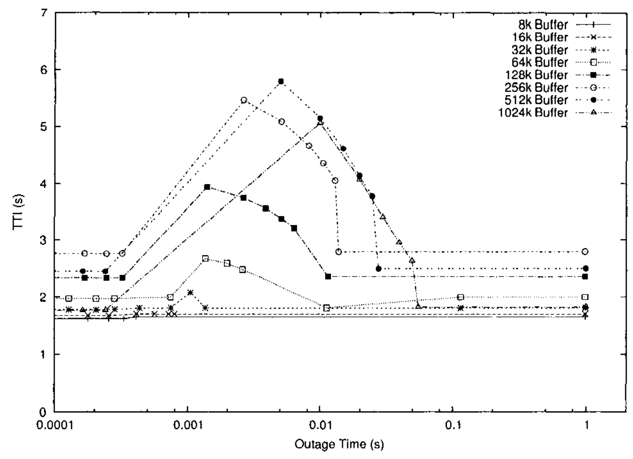


Figure 3.23 TTI vs. Outage Duration (Reno OC-3c Access)

3.4.4 Appropriate Restoration Objectives

The results presented here demonstrate that the traditional 50 ms recovery time is not suitable for TCP transport on the Internet. With SACK TCP, we found two restoration objectives, $\tau_{1,S}$ and $\tau_{2,S}$. $\tau_{1,S}$ is given by Eq. 3.10, and $\tau_{2,S}$ is closely related to RTO. With NewReno TCP, we also found two restoration objectives, $\tau_{1,NR}$ and $\tau_{2,NR}$. $\tau_{1,NR}$ is given by Eq. 3.12, and $\tau_{2,NR}$ is essentially the same as $\tau_{2,S}$. $\tau_{1,NR}$ is approximately twice as large as $\tau_{1,S}$ when *rwnd* is large. With Reno TCP, we found a recovery objective of 200ms for the DS0 access; For DS1 and OC-3c access, there seem to be no critical points from 15ms to 1s. For SACK TCP, if restoration can be finished within $\tau_{1,S}$, TTI does not increase much with recovery time. For NewReno TCP, if network failures can be restored within $\tau_{1,NR}$, TTI is influenced by RTT. If RTT is relatively small, TTI does not change much; otherwise, TTI increases with restoration time. In addition, we find that TTI could be less than outage duration in some cases due to large receive buffer.

Table 3.5 Resilience Objectives for SACK, NewReno and Reno TCP

Types / Access Rates	SACK TCP	NewReno TCP	Reno TCP
Low-rate Access	$\tau_{1,S}$	$\tau_{1,NR}$	200ms
High-rate Access	$\tau_{2,S}$	$\tau_{2,NR}$	1s

For low-rate access, we recommend $\tau_{1,S}$ or $\tau_{1,NR}$ to be the recovery objective if SACK and NewReno TCP are used. This is because in this situation $\tau_{1,S}$ or $\tau_{1,NR}$ is the threshold after which

TTI increases markedly. In the DS0 access scenario, both values are above 600 ms in all experimental cases. If Reno TCP is the transportation layer protocol, 200ms is recommended.

For high-rate access, if SACK and NewReno are used, $\tau_{2,S}$ or $\tau_{2,NR}$ is recommended because $\tau_{1,S}$ or $\tau_{1,NR}$ is probably too small to be realistically attainable, and any additional outage up to $\tau_{2,S}$ or $\tau_{2,NR}$ does not increase TTI significantly. In the OC-3c access scenario, both $\tau_{1,S}$ and $\tau_{1,NR}$ are below 15 ms in almost all experimental cases. Hence, a recovery objective of $\tau_{2,S}$ or $\tau_{2,NR}$ is appropriate. In the OC-3c access scenario, $\tau_{2,S}$ and $\tau_{2,NR}$ are approximately 1 s. For Reno TCP, 1s can be set to be the recovery objective.

Table 3.5 summarizes the resilience objectives for SACK, NewReno and Reno TCP. For real systems, Table 3.5 can be used as a guideline to design transport networks carrying TCP traffic. For example, if the TCP variant used in a transport network (e.g. SONET) is SACK TCP and the client access rate is DS0, then according to Table 3.5, $\tau_{1,S}$ should be adopted as the restoration target for the transport network. If we can further assume that *rwnd* is 16 KB, the segment size is 1460 bytes, and the failures involved are Type II Failure, then we will arrive at a restoration objective of 1643 ms using Eq. 3.10.

Chapter 4

Reactive and Proactive Approaches to Improving SACK TCP Resilience

SACK TCP, the most up-to-date TCP version, was designed to be capable of surviving multiple segment loss. However, as mentioned previously, if more than $n_{C,S}$ segments in one transmission window are lost, even if SACK TCP transitions into Fast Retransmit/Fast Recovery, a timeout will still occur and performance will be degraded dramatically. Also, SACK TCP tends to send out segments in clusters and congest bottleneck routers, which could potentially degrade SACK TCP resilience significantly. In this chapter, we present two types of approaches to improving SACK TCP Resilience. The first approach, "Probing", is a reactive approach that is used to solve the avoidable timeout problem. The second approach, "Pacing", is a proactive approach that spreads out outstanding segments and reduces the number of segments queued at bottleneck routers.

Traditionally, when a change in TCP is proposed, the change is often implemented in simulation tools, such as ns-2 [67] or OPNET [66]. And usually improvement is reported by showing the simulation results. However, the correctness of the implementation can never be proved by simulation itself. Delaying the proof of the correctness to real system implementation is of high risk and could lead to a significant waste of time and money. Petri nets have well-defined semantics which unambiguously define the behavior of each Petri net model. These semantics form the theoretical foundation that allows us to formally analyze Petri net models. By formally analyzing system properties such as reachability, boundedness and liveness, researchers can be more confident about the correctness of their models.

We developed Petri net models of SACK TCP with the proposed changes, based on the Petri net model presented in Section 3.1, and verified formally that our changes leave TCP "correct" in the sense that all states are bounded and the desired final state is always reachable from the initial state. The details of "Probing" and "Pacing" are presented in Section 4.1 and 4.2, respectively.

4.1 Reactive Approach: Probing

In Section 3.1.2.1 we noted that when more than $n_{C,S}$ segments are dropped and there are still at least three (in Type I Failure) or four (in Type II Failure) non-retransmitted segments following the lost segments that arrive at the receiver, the sender will transition into Fast retransmit/Fast recovery and retransmit the earliest unacknowledged segment. The retransmitted segment leads to a Partial ACK, but in this case this single Partial ACK, possibly together with the duplicate ACKs before this Partial ACK, will not make *pipe* less than *cwnd.* The sender will finally time out and this results in a serious TTI increase. However, this increase is preventable. When the Partial ACK arrives at the sender, it indicates that after the sender transitions into Fast retransmit/Fast recovery, the transmission link is somehow working although it may not be in perfect condition. In this case, the sender should try transmitting some segments after the Partial ACK arrives. Of course, the sender should not insert too many segments into the network because it is already congested.

We also noted that if less than three (in Type I Failure) or four (in Type II failure) segments follow the lost segments, Fast retransmit/Fast recovery will not occur because there will not be enough duplicate ACKs. In this case, TCP is again doomed to timeout. For TCP, timeout is always the last way to restore failures. This is because on the one hand, a timeout puts TCP into Slow Start which requires *cwnd* to increase from one. This limits the number of outstanding segments and leads to inefficient bandwidth usage. On the other hand, the period for the retransmission timer to expire is very costly. RFC 2988 [35] specifies a detailed way to calculate RTO and recommends that if the calculated value is less than 1 second then RTO should be rounded up to 1 second. This means that RTO is at least 1 second. Hence, when TCP is not in Fast retransmit/Fast recovery, after stopping receiving ACKs for some time, TCP should also do something to avoid the costly timeout.

In both of these situations, a reactive approach, "probing" the link, can be used to solve the timeout problem. The sender can insert "probe" segments into the link instead of simply waiting for timeout. Probing the link helps the sender to try to transmit some segments during the time it would otherwise just spend waiting for timeout. When TCP is in Fast retransmit/Fast recovery, this may also lead to more Partial ACKs. These Partial ACKs may make *pipe* less than *cwnd,* and the timeout will then be avoided. The reason that we consider "probing" a reactive approach is that it is triggered after some packets are already dropped, namely, a failure occurs before "probing" takes action. The details of the proposed "probing" approach are presented Section 4.1.1. The resilience improvement through probing and the formal analysis of the changed SACK TCP are described in Section 4.1.2 and Section 4.1.3, respectively.

4.1.1 The Details of the Probing Approach

Formally, we propose the following two types of probing mechanisms to improve SACK TCP resilience:

(1) **Type I Probing**: In the case that TCP is in Fast retransmit/Fast recovery, when a Partial ACK arrives, TCP checks to see whether this Partial ACK will make *pipe* less than *cwnd.* If so, TCP behaves as usual. If not, TCP will either retransmit the earliest unacknowledged segment, or transmit a new segment when there are no unacknowledged segments.

(2) **Type II Probing**: When TCP stops receiving acknowledgements for max(300ms, 2RTT) and the retransmission timer has not yet expired, TCP switches into Fast retransmit/Fast recovery (if it is not yet in Fast retransmit/Fast recovery), retransmits the latest unacknowledged segment, and then retransmits the earliest unacknowledged segment.

Traditionally, only ACKs are used to determine the condition of the network. The arrival of three duplicate ACKs is considered to be a strong indication that the network has become congested. Hence, TCP transitions into Fast retransmit/Fast recovery after receiving three duplicate ACKs. However, a relatively long "quiet period" without any ACKs arriving at the sender also indicates the network is probably not working normally. The general idea behind probing is to use retransmitted segments to determine the condition of the network, and actively generate new ACKs. This is particularly useful in the case of a network that can fail and then quickly and automatically be restored, as is the case in the modern SONET / SDH transport systems that carry much of today's Internet. The rather brief 50 ms failure window of these transport systems is likely to leave all the TCP sessions carried by a restored link waiting for timeout. We propose to use probes to bring these sessions into Fast retransmit/Fast recovery without suffering timeouts.

Type I Probing is relatively straightforward. The idea is to generate enough Partial ACKs so that *pipe* can become less than *cwnd,* thus avoiding timeout. Type II Probing is somewhat more complex. The general idea is to signal to the receiver the sender's picture of the segments that have not been acknowledged, and thus may need to be retransmitted. There are several reasons that we choose max(300 ms, 2RTT) as the length of the "quiet period". Intuitively, the length of this period should be related to RTT. If the link has just been restored, it has likely been rerouted over physical facilities that follow a longer geographical route so that the RTT for the session will increase after the link is restored. Thus some multiple of the pre-failure RTT should be used. Here, we use 2RTT to illustrate the benefits of probing. Further study and optimization is required. The value of 300 ms was chosen because the TCP receiver only sends out an ACK for every second segment, or within 200 ms of the arrival of the first unacknowledged segment. Therefore, an ACK could reasonably be delayed 200 ms and we do not want to burden the network unnecessarily by sending probes before this interval expires.

Type II Probing triggers TCP to send out both the latest and the earliest unacknowledged segment to avoid the 200 ms ACK delay that could result if only the earliest unacknowledged segment was sent. Recall that the receiver sends out an ACK immediately after receiving an out-of-order segment. Thus if the latest unacknowledged segment in the current transmission window has actually been properly received, all following retransmissions will be seen as out-of-order segments with each generating an immediate ACK rather than a 200 ms wait. The sender does not know whether the receiver has received the latest unacknowledged segment. With Type II Probing, the sender assumes that the latest unacknowledged segment has been lost and retransmits it before retransmitting any other unacknowledged segments. The cost is just one more duplicate ACK if the receiver actually has received the latest unacknowledged segment.

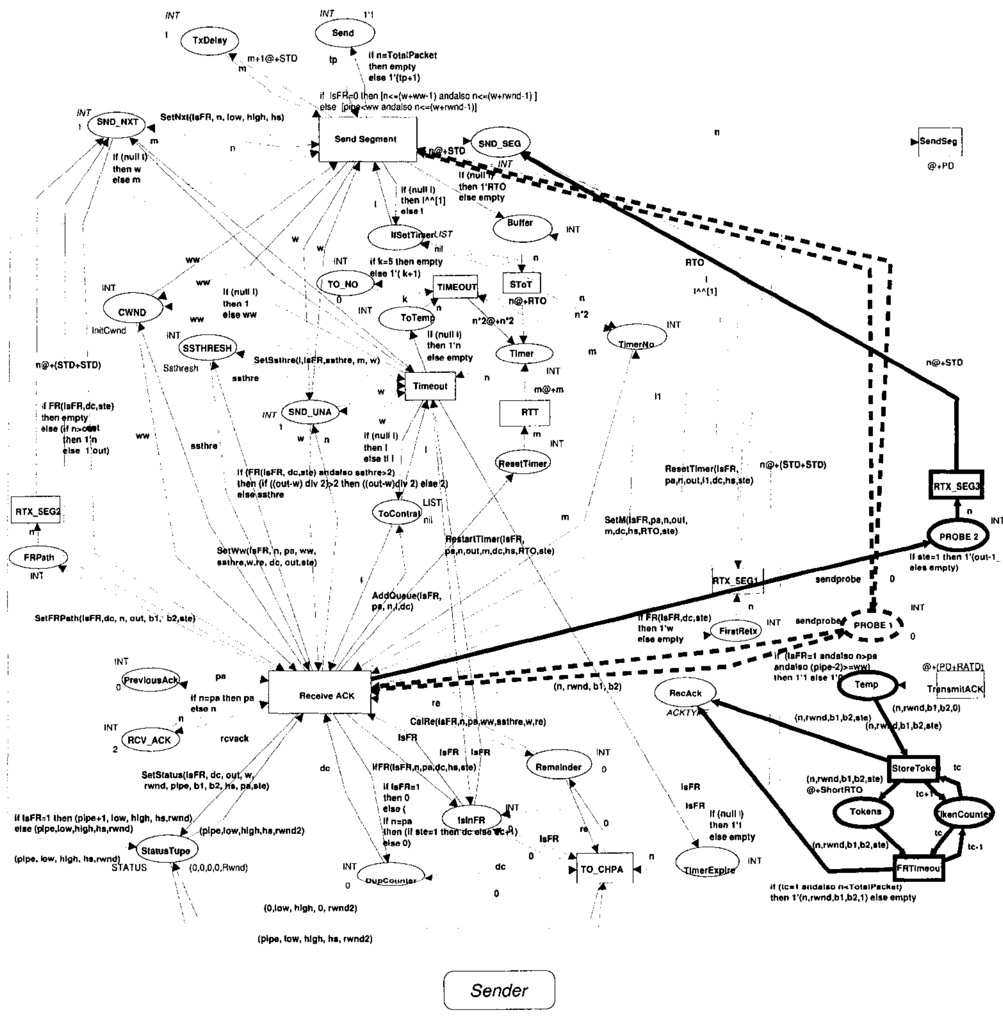
Type I Probing takes place when TCP is in Fast retransmit/Fast recovery and a Partial ACK arrives. Type II Probing occurs when less than three (in Type I Failure) or four (in Type II failure) segments follow the lost segments and thus TCP can not transition from Congestion Avoidance into Fast retransmit/Fast recovery. These two probing mechanisms eliminate the timeout problems in the two cases above.

However, Type II Probing also deals with a wider picture. That is, when the sender does not receive an ACK for max (300ms, 2RTT), it will assume that the network is malfunctioning and send out two segments to probe the network. This occurs whether TCP is in Congestion Avoidance, Fast retransmit/Fast recovery, or even Slow Start. And this will always save TCP from unnecessary timeouts and thus lead to better resilience performance.

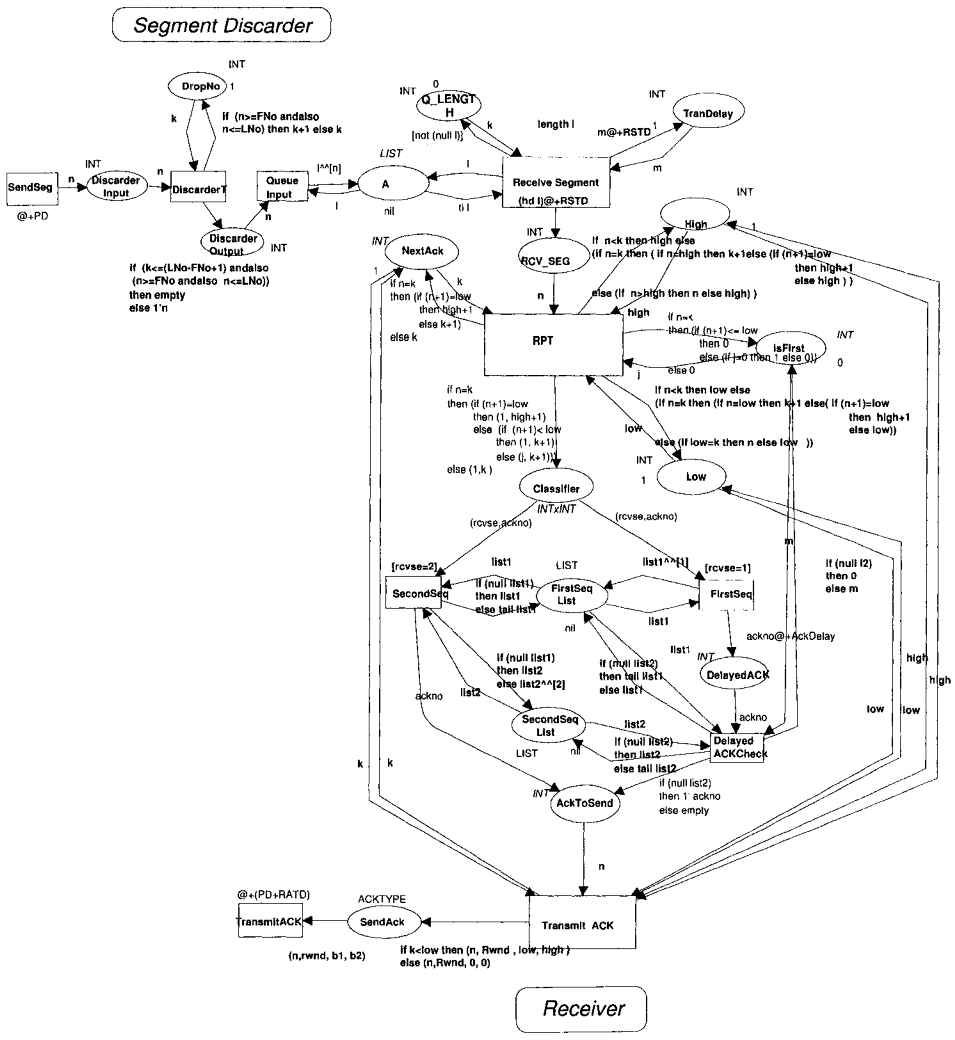
The Petri net model of standard SACK TCP is presented in Section 3.1.1. We modified the model and incorporated our proposed mechanisms. Figure 4.1 shows the Petri net model of SACK TCP with our proposed changes. For clarity, the original Petri net model was split into two parts. Figure 4.1 (a) shows the model for the modified SACK TCP sender and Figure 4.1 (b) illustrates the model for the modified SACK TCP receiver and Segment Discarder. Note that transition "**SendSeg**" and "**TransmitACK**" are the splitting points. Since our proposed mechanisms only modify the sender side TCP, Figure 4.1 (b) is essentially the same as Figure 3.2 (b). The components drawn with dashed thick lines correspond to Type I Probing and the components drawn with solid thick lines correspond to Type II Probing. The place **PROBE1** is connected to the transitions **Send Segment** and **Receive ACK**. When the sender is in Fast retransmit/Fast recovery and a Partial ACK arrives, if this Partial ACK will not make *pipe* less than *cwnd*, the token on **PROBE1** will be changed from "0" to "1", which will trigger the sender to either retransmit the earliest unacknowledged segment or transmit a new segment. The place **PROBE2** is connected to **Receive ACK** and **RTX_SEQ3**. When the sender stops receiving ACKs for max (300ms, 2RTT) and the retransmission timer has not expired, TCP switches into Fast retransmit/Fast recovery if it is not yet in Fast retransmit/Fast recovery. The latest unacknowledged segment will then be passed to **PROBE2** to be transmitted by **SND_SEG**.

4.1.2 Resilience Improvements through Probing

We noted in Section 3.1.2.1 that when there are only three (in Type I Failure) or four (in Type II Failure) segments remaining in the transmission window, the sender can still transition into Fast retransmit/Fast recovery. We define $n`_{C,S}$ as the critical number of dropped segments when there



(a) Petri Net Model of Modified SACK TCP Sender



(b) Petri Net Model of Modified SACK TCP Receiver and Segment Discarder

Figure 4.1 Modifications to Petri Net Model of SACK TCP

are only three or four segments remaining in the window. The following formula illustrates the relation between $n`_{C,S}$ and *rwnd*:

$$n`_{C,S} = \begin{cases} rwnd - 3 & \text{in Type I Failure} \\ rwnd - 4 & \text{in Type II Failure} \end{cases} \tag{4.1}$$

Our experiment illustrates a Type II Failure, and in this particular scenario $n`_{C,S}$ is 23-4=19.

In our experiments the RTT is approximately 200 ms, thus max (300 ms, 2RTT) = 400 ms. That is, when TCP stops receiving acknowledgements for 400 ms and the retransmission timer has not expired, Type II Probing should take place.

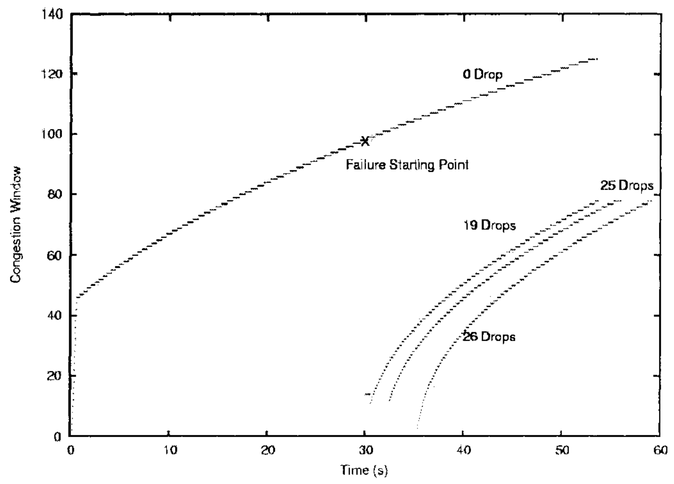


Figure 4.2 Congestion Window vs. Transfer Time (SACK with Probing)

The four curves in Figure 4.2 illustrate the typical changes in the sender's congestion window over time. The "0 drop" curve is the same curve as in Figure 3.12. If less than $n_{C,S}$ segments within one window are dropped, TCP can usually recover. The transfer time does not increase much in this case. This was already illustrated in Section 3.1.2.1. If the number of dropped segments falls in the range $n_{C,S}$ to $n`_{C,S}$ , Type I Probing is started. The first several Partial ACKs help decrease *pipe* and also trigger the sender to transmit some probes. This saves the sender from timing out and thus decreases the transfer time. Thus from 1 drop to $n`_{C,S}$ drops, TCP experiences almost the same *cwnd* change and transfer time increase. For clarity, only the 19 drop curve is included in Figure 4.2. Note that TCP experiences different recovery processes with less than and more than $n_{C,S}$ losses. That is, with less than $n_{C,S}$ losses TCP recovers using the standard Fast retransmit/Fast recovery mechanism. When more than $n_{C,S}$ losses occur TCP returns to a normal state because of both Type I Probing and the standard Fast retransmit/Fast recovery mechanism.

If more than $n`_{C,S}$ segments are dropped, less than three (in Type I Failure) or four (in Type II failure) segments follow the lost segments. In this case, normally TCP will not transition into Fast retransmit/Fast recovery because there will not be enough duplicate ACKs. However, Type II Probing will start once 400 ms have passed without an ACK. TCP then transitions into Fast retransmit/Fast recovery and two more probes, segments (*rwnd*+1) and (*rwnd*+2), are sent out. If these two probes arrive at the receiver safely, TCP will avoid timeout, and the transfer time will increase only slightly. The curves for these scenarios are not included because they are similar to the 19 drop curve. If segment (*rwnd*+1), the latest unacknowledged segment, is lost and segment (*rwnd*+2) arrives at the receiver, Fast Recovery will occur until TCP transitions from Fast retransmit/Fast recovery into Congestion Avoidance. Although in this case TCP goes through Fast Recovery, the transfer time increases significantly. This is because the latest unacknowledged segment is already lost, and thus all following retransmissions will experience the 200 ms ACK delay when one ACK can trigger only one retransmitted segment. The (*rwnd*+1) drop curve is not included for clarity. If segment (*rwnd*+2) is also dropped, TCP will finally time out and the transfer time will increase dramatically. This is illustrated by the 25 drop curve; this corresponds to the 23 drop scenario in Section 3.1.2.1. The 26 drop curve is actually the counterpart of the 24 drop curve in Figure 3.12.

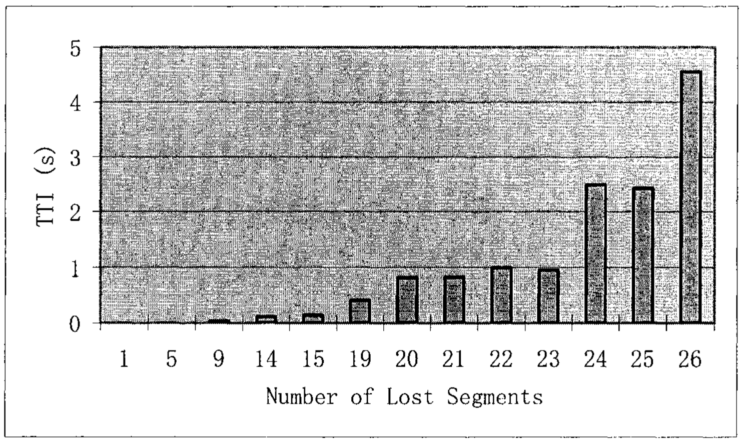


Figure 4.3 TTI vs. No. of Lost Segments (SACK with Probing)

Figure 4.3 presents the resilience of our improved SACK TCP from the perspective of TTI vs. the number of dropped segments. From 1 drop to 23 drops, TTI does not change much. But TCP experiences three different recovering stages when the number of dropped segment is in this range: Fast retransmit/Fast recovery, Type I Probing and Type II Probing. The cases of 24 and 25 drops show TTI increases, but the former increase is because of the 200 ms ACK delay and the latter is due to timeout. With 26 drops, a second timeout occurs leading to a further large increase in TTI. The combination of Type I and Type II Probing decreases TTI by at least 50% over the range from

1 drop to 23 drops. SACK TCP with Type I and Type II Probing has much better resilience than the original protocol.

4.1.3 Formal Analysis of SACK TCP with Probing

Petri nets can be used to study the performance of concurrent systems by simulation. More importantly, Petri nets allow us to formally analyze systems of interest, increasing our confidence about the correctness of Petri net models. In our research, we formally analyzed both the Petri net model of standard SACK TCP and the model of SACK TCP with Probing. We first present the analysis results of standard SACK TCP.

As mentioned in Section 2.2.4, Occurrence Graphs are commonly used in the analysis of Petri nets. For example, the occurrence graph of a Petri net can be used to determine whether a particular final state is always reachable from a given initial state, no matter what trajectory the model follows through its state space. In the case of file transfer, for example, we need to be certain that it is always possible to reach the state where all segments have arrived at the receiver and been acknowledged. Reachability analysis allows us to verify this property of the protocol more generally than by just examining the execution of some finite number of simulation runs. Occurrence Graphs also allow us to make sure the number of tokens at any place in the model (for example, the number of segments or acknowledgements) remains bounded by certain pre-specified values. Again, formal analysis allows us to be certain that these bounds are always obeyed.

We used Design/CPN to derive the occurrence graph of our Petri net model of standard SACK TCP and analyze its boundedness and reachability. We constructed occurrence graphs for three typical scenarios: 0 drop, 14 drops and 15 drops. The "Overview" section in Table 4.1 summarizes the number of nodes and arcs in each of the three occurrence graphs. The properties of these graphs are summarized in Table 4.1. For clarity, we present the boundedness for only the major places in the Petri net model.

Design/CPN defines the upper integer bound as the maximum possible number of tokens in an individual place and defines the lower integer bound as the minimum possible number of tokens in an individual place. Further, the upper multi-set bound of a place is defined as the smallest multi-set which is larger than all reachable markings of the place. The lower multi-set bound of a place is the largest multi-set which is smaller than all reachable markings of the place.

In our model, the places **Sender.RCV_ACK**, **Sender.CWND**, **Sender.SSTHRESH** and **Receiver.Q_LENGTH** should have one token at any time. The "Integer Bounds" information in Table 4.1 verifies this because the upper integer bound and the lower integer bound for these places are always 1. **Receiver.SND_ACK** should have either one token or no token in any case and this is confirmed by the upper integer bound of 1 and lower integer bound of 0. **Sender.SND_SEG** and **Sender.RCV_SEG** should have zero, one or two tokens at any time; the fact that their upper integer bounds are 2 and their lower integer bounds are 0 verifies this behavior.

In the 0 drop case, the upper multi-set bound of **Sender.SSTHRESH** includes only 45 because there is no failure in this scenario and *ssthresh* stays at 45. In the 14 drop and 15 drop case, the bound contains both 11 and 45 because *ssthresh* is set to $\lfloor rwnd / 2 \rfloor$ after the failure occurs and thus *ssthresh* could be either 11 or 45 in these cases. In all three cases, the upper multi-set bound of **Receiver.Q_LENGTH** has 19 elements and the largest is 18. This shows that there are at most 18 segments waiting for service at the receiver. This is reasonable because *rwnd* is set to 23 in this scenario and the number of segments queued at the receiver should be less than *rwnd*. The lower multi-set bounds for the two places are rather simple. They have either one element or no element at all. This is reasonable because both of these places should have one token at any time. In this case, if the upper multi-set bound of the place has only one token, then the corresponding lower multi-set bound should also only contain that token; if the upper multi-set bound of the place has multiple tokens, the corresponding lower multi-set bound should be empty. In short, the boundedness results provided by the occurrence graphs show that the places are bounded as SACK TCP requires.

In Design/CPN, a dead marking is a marking with no enabled transitions. When deriving an occurrence graph, Design/CPN numbers the markings sequentially. That is, the initial marking is numbered one and the next calculated marking is numbered consecutively. In the "Liveness

Properties" section of Table 4.1, each of the three scenarios has a dead marking. Knowing the numbers of the dead markings, we analyzed the dead markings and verified that they are the markings corresponding to the last steps of the data transfer where all segments have arrived at the destination. With the query tool of Design/CPN, we verified that these markings are reachable from the initial markings where all segments are at the source. This shows that all the segments at the sender can be transmitted successfully in the three scenarios.

In the simulation of a concurrent system, multiple transitions will often be enabled at the same time. Most simulators will simply select from the enabled transitions randomly. Hence, a successful simulation only indicates the correctness of one of many possible execution sequences. By generating occurrence graphs for our Petri net model and analyzing its reachability, we can conclude that for all possible execution sequences, the data transfer always completes successfully. This is because each occurrence graph has only one dead marking that corresponds to the last step of the corresponding data transfer and this dead marking is always reachable from the initial marking. The reachability information in Table 4.1 verifies the correctness of our SACK TCP model.

In Design/CPN, a dead transition is defined as a transition that is never enabled in any reachable marking. In our model, **Sender.RTX_SEG1** and **Sender.RTX_SEG2** are related to Fast retransmit/Fast recovery. **Sender.TIMEOUT** and **Sender.TO_CHPA** are related to timeout. In the 0 drop case, TCP should not experience Fast retransmit/Fast recovery and timeout, so the above four transitions should never be enabled. The occurrence graph shows that these transitions are dead in the 0 drop case. In the 14 drop case, Fast retransmit/Fast recovery occurred, thus only the transitions related to timeout appeared to be dead transitions. In the 15 drop case, TCP did not switch into Fast retransmit/Fast recovery because there were not enough ACKs. Instead it finally timed out. As a result, the two transitions related to Fast retransmit/Fast recovery became dead transitions. The dead transition information in Table 4.1 further verifies the correctness of our model.

We analyzed the boundedness and reachability of the Petri net model of SACK TCP with

Probing in a manner similar to the studies performed for the standard SACK TCP. Results for the 0 drop, 19 drop and 20 drop cases are included in Table 4.2.

Table 4.1 Occurrence Graph Data for 0-Drop, 14-Drop and 15-Drop Case (Standard SACK TCP)

Occurrence Graph Statistics		0 Drop Case		14 Drop Case		15 Drop Case	
Overview	Nodes	95921		96474		96040	
	Arcs	123331		124466		123489	
Integer Bounds	Typical Places	Upper	Lower	Upper	Lower	Upper	Lower
	Sender.SND_SEG	2	0	2	0	2	0
	Sender.RCV_ACK	1	1	1	1	1	1
	Sender.CWND	1	1	1	1	1	1
	Sender.SSTHRESH	1	1	1	1	1	1
	Receiver.RCV_SEG	2	0	2	0	2	0
	Receiver.SND_ACK	1	0	1	0	1	0
	Receiver.Q_LENGTH	1	1	1	1	1	1
Upper Multi-set Bounds	Sender.SSTHRESH	1`45		1`11+1`45		1`11+1`45	
	Receiver.Q_LENGTH	1`0+1`1+1`2+ 1`3+1`4+1`5+ 1`6+1`7+1`8+ 1`9+1`10+1`11+ 1`12+1`13+ 1`14+ 1`15+ 1`16+ 1`17+1`18		1`0+1`1+1`2+ 1`3+1`4+1`5+ 1`6+1`7+1`8+ 1`9+1`10+1`11+ 1`12+1`13+ 1`14+ 1`15+ 1`16+ 1`17+1`18		1`0+1`1+1`2+ 1`3+1`4+1`5+ 1`6+1`7+1`8+ 1`9+1`10+1`11+ 1`12+1`13+ 1`14+ 1`15+ 1`16+ 1`17+1`18	
Lower Multi-set Bounds	Sender.SSTHRESH	empty		empty		empty	
	Receiver.Q_LENGTH	1`45		empty		empty	
Liveness Properties	Dead Markings	[95921]		[96474]		[96040]	
	Dead Transitions	Sender.RTX_SEG1 Sender.RTX_SEG2 Sender.TIMEOUT Sender.TO_CHPA		Sender.TIMEOUT Sender.TO_CHPA		Sender.RTX_SEG1 Sender.RTX_SEG2	

Table 4.2 is very similar to Table 4.1. However, Table 4.2 has data for the places **Sender.PROBE1** and **Sender.PROBE2**. In normal conditions, **Sender.PROBE1** has only one token of 0. When the number of dropped segments falls in the range $n_{C,S}$ to $n`_{C,S}$, the token is changed to 1 a few times triggering the sender to transmit some probes. This is verified by the data

in Table 4.2. Normally, **Sender.PROBE2** has zero tokens. When TCP stops receiving acknowledgements for 400 ms and timeout has not occurred, **Sender.PROBE2** gets the token representing the latest unacknowledged segment, which is 3855.

Table 4.2 Occurrence Graph Data for 0-Drop, 19-Drop and 20-Drop Case (Probing SACK TCP)

Occurrence Graph Statistics		0 Drop Case		19 Drop Case		20 Drop Case	
Overview	Nodes	102773		103045		103079	
	Arcs	130183		130618		130673	
Integer Bounds	Typical Places	Upper	Lower	Upper	Lower	Upper	Lower
	Sender.PROBE 1	1	1	1	1	1	1
	Sender.PROBE 2	0	0	0	0	1	0
	Sender.SND_SEG	2	0	2	0	2	0
	Sender.RCV_ACK	1	1	1	1	1	1
	Sender.CWND	1	1	1	1	1	1
	Sender.SSTHRESH	1	1	1	1	1	1
	Receiver.RCV_SEG	2	0	2	0	2	0
	Receiver.SND_ACK	1	0	1	0	1	0
	Receiver.Q_LENGTH	1	1	1	1	1	1
Upper Multi-set Bounds	Sender.PROBE 1	1`0		1`0+1`1		1`0+1`1	
	Sender.PROBE 2	empty		empty		1`3855	
	Sender.SSTHRESH	1`45		1`11+1`45		1`11+1`45	
	Receiver.Q_LENGTH	1`0+1`1+1`2+ 1`3+1`4+1`5+ 1`6+1`7+1`8+ 1`9+1`10+1`11+ 1`12+1`13+ 1`14+ 1`15+ 1`16+ 1`17+1`18		1`0+1`1+1`2+ 1`3+1`4+1`5+ 1`6+1`7+1`8+ 1`9+1`10+1`11+ 1`12+1`13+ 1`14+ 1`15+ 1`16+ 1`17+1`18		1`0+1`1+1`2+ 1`3+1`4+1`5+ 1`6+1`7+1`8+ 1`9+1`10+1`11+ 1`12+1`13+ 1`14+ 1`15+ 1`16+ 1`17+1`18	
Lower Multi-set Bounds	Sender.PROBE 1	1`0		empty		empty	
	Sender.PROBE 2	empty		empty		empty	
	Sender.SSTHRESH	1`45		empty		empty	
	Receiver.Q_LENGTH	empty		empty		empty	
Liveness Properties	Dead Markings	[102773]		[103045]		[103079]	
	Dead Transitions	Sender.RTX_SEG1 Sender.RTX_SEG2 Sender'RTX_SEG3 Sender.TIMEOUT Sender.TO_CHPA		Sender'RTX_SEG3 Sender.TIMEOUT Sender.TO_CHPA		Sender.TIMEOUT Sender.TO_CHPA	

With regard to dead transitions, Table 4.2 is also different than Table 4.1. For the 0 drop and 19 drop case, Table 4.2 has an extra dead transition of **Sender.RTX_SEG3**. This transition is related to Type II Probing and hence in the 0 drop and 19 drop cases, it is never invoked. For the 20 drop case, **Sender.TIMEOUT** and **Sender.TO_CHPA** appear in Table 4.2, while **Sender.RTX_SEG1** and **Sender.RTX_SEG2** do not. As mentioned previously, **Sender.TIMEOUT** and **Sender.TO_CHPA** are related to timeout; **Sender.RTX_SEG1** or **Sender.RTX_SEG2** are related to Fast retransmit/Fast recovery. With Type II Probing, timeout is avoided and TCP transitions into Fast retransmit/Fast recovery in the 20 drop case. The corresponding dead transition information confirms this improvement due to Type II Probing.

Based on the occurrence graph data and applying the same analysis method, we also verified that the improved version of SACK TCP is bounded, and the final step of the successful transfer is always reachable from the initial state. Thus, the correctness of the improved SACK TCP is also verified by formal analysis.

4.2 Proactive Approach: Pacing

SACK TCP can survive multiple segment loss. However, there exists a critical number $n_{C,S}$ above which SACK TCP will resort to timeout instead of Fast Recovery to recover from the loss event. Resorting to timeout degrades TCP performance significantly. Spreading the outstanding segments across each RTT reduces the possibility that SACK TCP loses more than $n_{C,S}$ segments within one transmission window. This corresponds to the traditional paced TCP. However, this version of paced TCP might suffer from negative feedback and could possibly congest bottleneck routers. In this section, we introduce a simple but effective method, leveraging Chebyshev Inequality, to calculate the bottleneck bandwidth and present a new pacing approach, α-min Paced SACK TCP, that improves standard SACK TCP resilience. Our results show that α-min Paced SACK TCP works well over a relatively wide range of RTT. In our research, α-min Paced SACK TCP was modeled with Petri nets. The Petri net model was formally analyzed to prove its "correctness". The details of α-min Paced SACK TCP is presented in Section 4.2.1 and the formal analysis results are summarized in Section 4.2.2.

4.2.1 Paced SACK TCP

To investigate the performance of α-min Paced SACK TCP, a simulation configuration similar to that in Section 4.1 was set up. Namely, *cwnd*, *rwnd*, and *ssthresh* were set to 1, 23 and 45, respectively; the receiver access rate is DS1 and the sender access rate is OC-3c; a network topology shown in Figure 3.9 is adopted. However, to study α-min Paced SACK TCP, we do not need the packet discarder any more. Figure 4.4 shows the abstract topology used in this section. Also, rather than studying the behavior of a 10 MB file transfer, we consider a 1 MB file transfer from the sender to the receiver. This change shortens simulation time and does not have any impact on TCP performance. In addition, the simulation in this section is trace-driven. We gathered a network delay trace from an OPNET TCP model [83] for the same network model and scenario. The average round trip time (RTT) in the trace is 48.24 ms. To study the performance of TCP with longer RTT, we scaled the trace to have an average RTT of 250 ms.

As mentioned previously, $n_{C,S}$ is vital to SACK TCP because it suffers from timeout when more than $n_{C,S}$ segments within one transmission window are lost. Since timeout is such a costly event, we should try to constrain the number of lost segments below $n_{C,S}$. We define **Loss Tolerance Time (LTT)** as the shortest period of time during which $n_{C,S}$ segments could be discarded during a TCP session. Although the exact discarding pattern of each network varies, from the viewpoint of TCP, if LTT is longer then it is less likely $n_{C,S}$ segments could be discarded resulting in better TCP resilience. In our experiment, *rwnd* is 23, thus $n_{C,S}$ is equal to 14 or 15 according to Eq. 3.7. Either value can be used to illustrate TCP resilience. We chose the Type II case to study the resilience of SACK TCP in this section. Thus, to improve TCP resilience, any 14 consecutive segments should be spread out as far as possible.

As an ACK-clocking protocol, TCP uses the minimum of *cwnd* and *rwnd* to determine the number of segments that are allowed to be sent out and uses the receipt of acknowledgements to trigger the sending of packets. We define the minimum of *cwnd* and *rwnd* as "usable window". In this thesis, we use w_U to denote "usable window". Intuitively, ACK-clocking should spread the data across each RTT. However, a number of factors, including bandwidth-delay product, *cwnd*, *rwnd,* etc., cause TCP to be bursty.

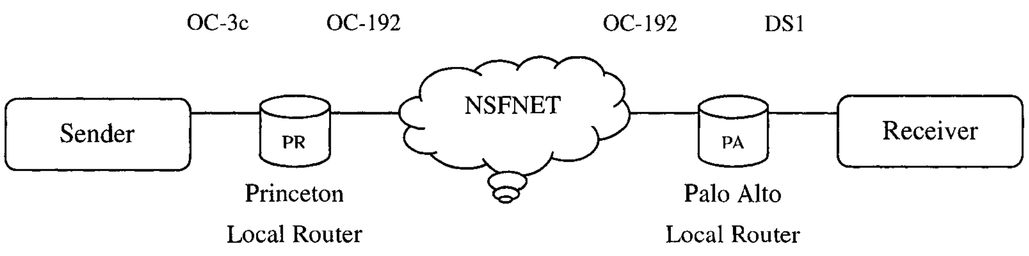


Figure 4.4 Abstract Topology of the Modeled System

Figure 4.5 presents the packet arrival pattern at the Palo Alto router (PA) for 1 MB file transfer with 250 ms RTT. For clarity, only the first 200 segments are included. Before around 2.5 seconds, TCP is in Slow Start and *cwnd* is less than *rwnd*. During this period, *cwnd* increases exponentially from one. As the result, segment clusters become larger and larger. After 2.5 seconds, *cwnd* becomes greater than *rwnd*. However, during this rest of the session, the number of segments per RTT is fixed at w_U because only w_U segments are allowed to be sent out within one transmission window. During this period, segments also tend to cluster. Clustering will decrease rather than increase LTT.

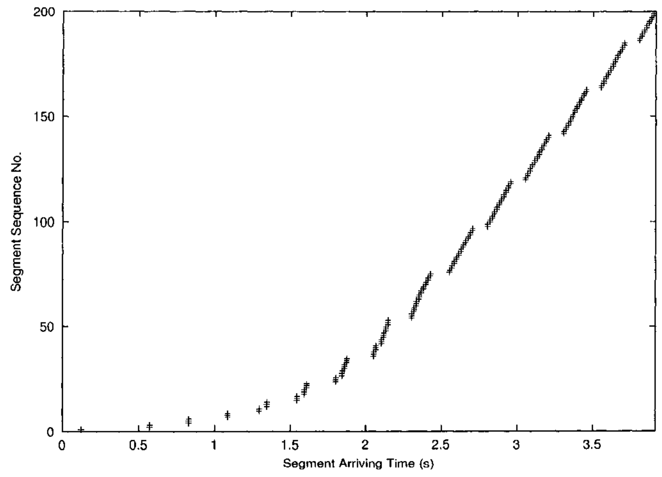


Figure 4.5 Packet Arrival Pattern at Router PA (250 ms RTT)

We also analyzed the bursty nature of TCP in this scenario quantitatively. Since in this case the average RTT is 250 ms, suppose the rwnd segments are evenly spread within one RTT, LTT should be (250 / 23) x 13 = 141.30 ms, where 23 is the value of *cwnd* and 13 corresponds to the number of gaps resulting from $n_{C,S}$ (14 in our experiments) segments. The LTT from the simulation is just 93.86 ms, which is much less than the ideal LTT. Thus there is much room to improve SACK TCP resilience.

Figure 4.6 shows the packet arrival pattern at router PA when the average RTT is 50 ms. In this case, after around 0.75 second, *cwnd* becomes greater than *rwnd*. If the segments are spread evenly across RTT after 0.75 second, the ideal LTT would be (50 /23) x 13 = 28.26 ms. However, the LTT from the simulation is 91.83 ms, which is greater than the ideal LTT. The queuing of segments at PA explains the observation.

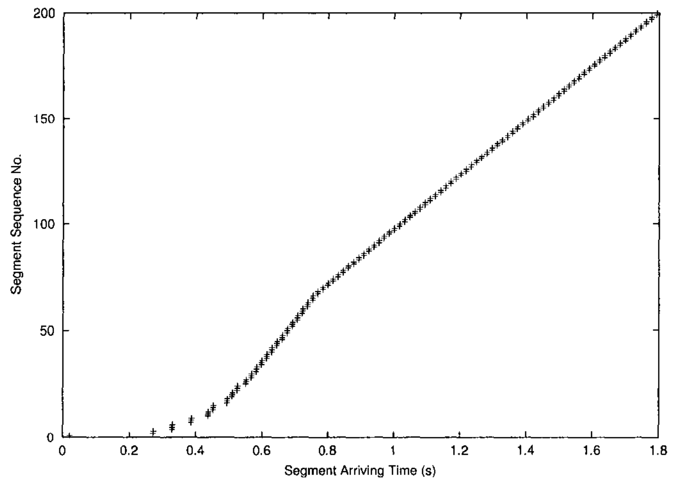


Figure 4.6 Packet Arrival Pattern at Router PA (50 ms RTT)

Figure 4.7 illustrates the number of segments queued at PA in the 250 ms RTT case. Before 2.5 seconds, the number of segments queued fluctuates between 1 and 5. This corresponds to the Slow Start state, during which *cwnd* increases exponentially and thus triggers more packets to be transmitted back-to-back. After 2.5 seconds, the number of segments queued is always 1. The value remains constant because during this period increasing *cwnd* does not introduce more segments into the network and the sending rate is clocked by returning acknowledgements paced by the bottleneck DS1 link. Figure 4.8 shows the number of segments queued in the 50 ms RTT case. Before around 0.75 second, this value increases quickly, which corresponds to Slow Start state. After 0.75 second, the number of segments queued remains at 16 or 17 due to the fixed w_U.

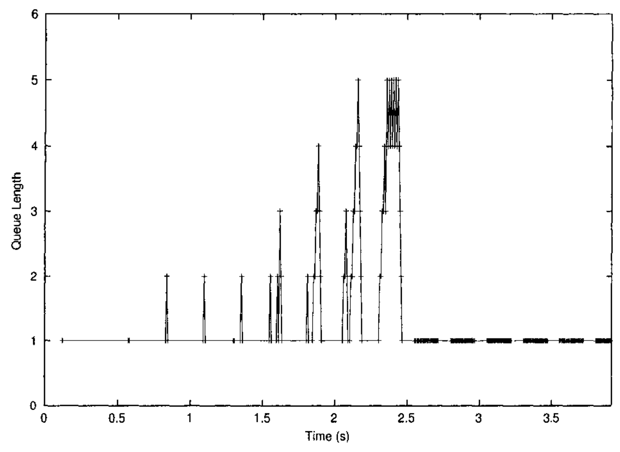


Figure 4.7 The Number of Segments Queued at Router PA (250 ms RTT)

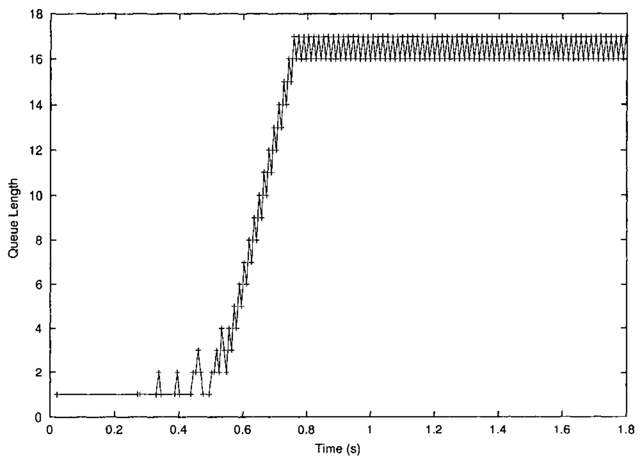


Figure 4.8 The Number of Segments Queued at Router PA (50 ms RTT)

Queuing affects TCP resilience because it increases the possibility of queue overflow, which is one of the major causes of segment loss. To improve TCP resilience, we should try to reduce the number of segments queued. The 50 ms case in our research introduced a very lengthy queue at PA, this harms TCP resilience significantly.

4.2.1.1 α-Paced SACK TCP

Pacing is not a new idea. Kulik [84] proposed a paced TCP that sends out segments in multiple, small bursts during each RTT and claimed that the paced TCP performs better than standard TCP. The Berkeley WebTP group has combined pacing and other specialized techniques into a transport protocol optimized for web traffic [85]. Aggarwal [86] evaluated TCP pacing performance in detail. Akyildiz [87] developed a TCP variant, TCP-Peach, that is based on the novel concept of using dummy segments to probe the availability of network resources without carrying any new information to the sender. With TCP-Peach, the real segments and dummy segments are spread out over each RTT. Razdan [88] proposed a paced version of TCP in which pacing is used during "slow start" in order to improve TCP performance when the receive buffer is small. Several of these previous pacing mechanisms [84] [85] [86] [87] suggested segments should be paced at (w_U /RTT) while one [88] recommended that the pacing rate should be the bottleneck bandwidth. However, to our knowledge, no research into pacing has been focused on TCP resilience and we believe that our suggestion that the pacing rate should be the minimum of the bottleneck bandwidth and (w_U /RTT) is new. In this section we illustrate a pacing mechanism, α-paced SACK TCP that paces segments at a rate of approximately (w_U /RTT). In Section 4.1.2.2, we present an advanced version, α-min paced SACK TCP, that uses the minimum of the bottleneck bandwidth and (w_U /RTT) as the pacing rate.

As mentioned previously, standard TCP is a window-based protocol. It uses w_U to determine the number of segments that are allowed to be sent out and uses the receipt of acknowledgements to trigger the sending of packets. This is in contrast to pure rate-based protocols which use rates to determine how much and when to send. TCP pacing is rather a hybrid scheme between these two approaches. It uses TCP congestion control to determine how much to send and uses rates to determine when to send. The intention is to spread out data across each RTT. This coincides with the idea of increase LTT.

The problem of the rate TCP should follow is difficult. Traditionally, (w_U / RTT) is used to pace TCP. Namely, RTT is divided into intervals of duration RTT/ w_U and during each interval at most one segment is allowed to be transmitted. Figure 4.9 illustrates the resilience improvement that this type of TCP pacing could achieve. Figure 4.9 corresponds to the case of 250 ms RTT. Apparently, TCP pacing spreads data out successfully compared with standard SACK TCP. Quantitatively, the LTT in the paced TCP case is 144.11 ms, which is close to the ideal LTT. Furthermore, the number of segments queued at PA remains at 1 throughout the TCP session.

However, Pacing TCP at the rate (w_U / RTT) might introduce a serious problem. After *cwnd* becomes greater than *rwnd*, w_U is equal to *rwnd*, which is a fixed value. If for some reason, the network becomes congested for a short period of time and leads to a temporary longer RTT, then (w_U / RTT) decreases. This reduced pacing rate will in turn affect the measurement of RTT because TCP uses returning ACKs to calculate RTT. That is, if TCP paces the segments at (w_U / RTT), once w_U becomes stable, it is very easy for TCP to reduce the pacing rate, but difficult increase it. When the pacing rate decreases to the extent that the network bandwidth is not fully used, the TCP session's throughput is reduced by the pacing mechanism.

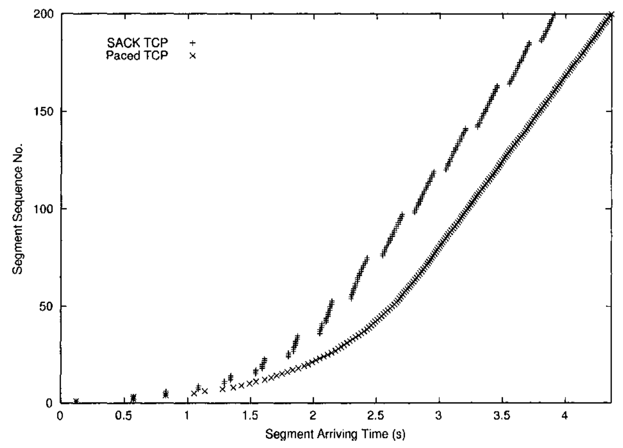


Figure 4.9 SACK TCP vs. α-Paced TCP (250 ms RTT)

We propose α-paced SACK TCP to solve this problem. Our α-paced SACK TCP employs pacing throughout the entire TCP session. The idea behind α-paced SACK TCP is that TCP should pace the segments at the rate α x (w_U / RTT), where α is a positive number that is greater than 1. That is, by intentionally increasing the pacing rate by the factor α, TCP actively tries to avoid the potential problem mentioned previously. We should note that this approach does not lead to an unlimited pacing rate. This is because no matter how fast the sending rate is, the measured RTT is always greater than the sum of propagation delay and transmission delay. If we use RTT_{min} to denote the sum of propagation and transmission delay, the maximum pacing rate is limited by α x (w_U / RTT_{min}). This is never faster than SACK TCP because SACK TCP sends out segments immediately after receiving ACKs. If we consider SACK TCP to be an extremely-high-rate paced TCP, then we conclude that the performance of α-paced SACK TCP becomes more similar to

SACK TCP as α increases. Hence, α should be chosen carefully so that α-paced SACK TCP not only keeps the advantage of pacing but also send out segments fast enough. Starting from the Petri net model of SACK TCP, α-paced SACK TCP was also modeled with Design/CPN. Actually we simply added several places and transitions to calculate (w_U / RTT).

We experimented with different values for α in the 250 ms RTT case and summarized the resulting LTTs in Table 4.3. Note that when α is equal to 1.0, α-paced SACK TCP is just the traditional paced SACK TCP. Table 4.3 illustrates that as α increases, LTT decreases and gets closer to the LTT in SACK TCP. Figure 4.10 also illustrates the same trend. For clarity, only the scenarios corresponding to α=1.0 and α=1.5 are included. Also, for all the α values in Table 1, the number of segments queued at PA remains at 1 throughout the session.

Table 4.3 Achieved LTT for Different α (250 ms RTT)

α	1.0	1.1	1.2	1.3	1.4	1.5
LTT (ms)	141.11	138.83	129.14	126.67	120.03	111.96

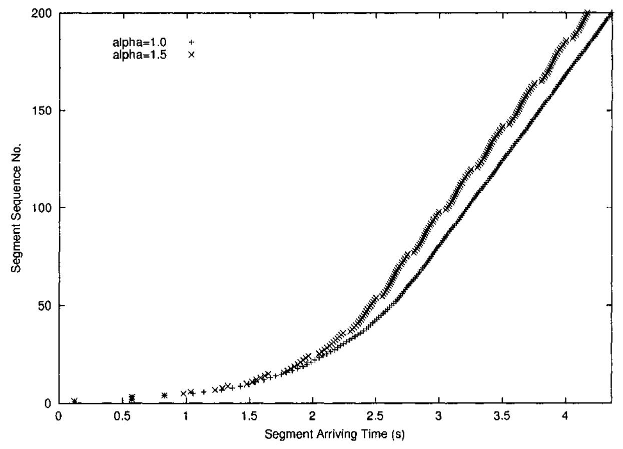


Figure 4.10 α-Paced TCP with Varying α (250 ms RTT)

For the case of 50 ms RTT, α-paced SACK TCP does not have an obvious impact. With different α values, the achieved LTT and the number of segments queued at PA remain almost the same as in the SACK TCP scenario. However, the large number of segments queued at PA poses a serious threat to TCP resilience.

4.2.1.2 α-min Paced SACK TCP

With SACK TCP, segments tend to cluster in the case with 250 ms RTT, but few segments are queued at PA. In the case of 50 ms RTT, many segments are queued at PA, but they are already clocked by ACKs. α-paced SACK TCP spread out the segments when it can, but does not reduce the number of segments queued in the short-RTT case. This leads us to propose adaptive pacing.

Let us first consider what proper pacing rate should be. Traditionally, bandwidth-delay product is recommended to size *cwnd* in SACK TCP. By bandwidth-delay product, "bandwidth" means the lowest transmission rate achieved along the round-trip between the sender and the receiver, denoted as r_B; "delay" just means RTT. When *rwnd* is taken into account, the sender's usable w_U should be equal to (r_B x RTT) to make full use of the network. From the usage of the network, all scenarios can be categorized into three cases as follows.

Case 1: If $w_U = (r_B \times RTT)$, there should be no segments queued and data should be spread out evenly.

Case 2: If $w_U > (r_B \times RTT)$, SACK TCP will insert more than enough segments into the network. As a result, (w_U - (r_B x RTT)) segments will be queued at the bottleneck.

Case 3: If $w_U < (r_B \times RTT)$, not enough segments can be inserted into the network, thus segments are not transmitted continuously. Those segments that are sent out continuously will form a cluster. When the sender stops sending segments due to the limitation of w_U, the gap between clusters is introduced.

The description of these three cases can be converted as follows:

$$\begin{cases} \text{If } (w_U / RTT) = r_B & \text{Case 1} \\ \text{If } (w_U / RTT) > r_B & \text{Case 2} \\ \text{If } (w_U / RTT) < r_B & \text{Case 3} \end{cases} \tag{4.2}$$

Note that w_U is the number of segments that are allowed to be sent out during one RTT. Hence, (w_U / RTT) is the expected rate from TCP's viewpoint. If we define r_E as (w_U / RTT), then the description of the three cases can be further changed as follows:

$$\begin{cases} \text{If } r_E = r_B & \text{Case 1} \\ \text{If } r_E > r_B & \text{Case 2} \\ \text{If } r_E < r_B & \text{Case 3} \end{cases} \tag{4.3}$$

In Case 1, everything is in perfect condition. In Case 2, r_E should be limited to adapt to the bottleneck bandwidth r_B, preventing TCP from saturating the network. In Case 3, TCP should just send out segments at r_E to prevent them from clustering. Hence, paced TCP should send out segments adaptively at min (r_E , r_B). As noted previously, the pacing rate should be scaled up by a factor α. The final ideal pacing rate is [α x min (r_E , r_B)]. We call this version of pacing "α-min Paced SACK TCP".

w_U and RTT are available from TCP, so r_E can be calculated once TCP calculates its first RTT. r_B is a property of the path between the sender and the receiver, and it is not available to TCP sender. r_B-related information can be gathered at the receiver or intermediate routers, with the information transferred to the sender. But this type of approach requires to the receiver or intermediate routers, which limits the usefulness. We proposed a method to measure r_B at the sender. This method can be adopted gradually.

r_B can be measured by monitoring the arrival rate of ACKs. The r_B measurement is synchronized with the RTT update in TCP. From the beginning of each RTT measurement, the sender starts timing the gaps between consecutive ACKs and counting the data acknowledged by each ACK. We use Δt_i to denote the gap between the ith ACK and the (i-1)th ACK, and use Δd_i to denote the amount of data acknowledged by the ith ACK. If n ACKs arrives within the current RTT measurement, the sender will have two series Δt_1, Δt_2 ...Δt_n and Δd_1, Δd_2 ...Δd_n. The straightforward way to estimate r_B is illustrated as follows:

$$r_B = (\Delta d_1+\Delta d_2 +\ldots+\Delta d_n) / (\Delta t_1+\Delta t_2 +\ldots+\Delta t_n) \qquad (4.4)$$

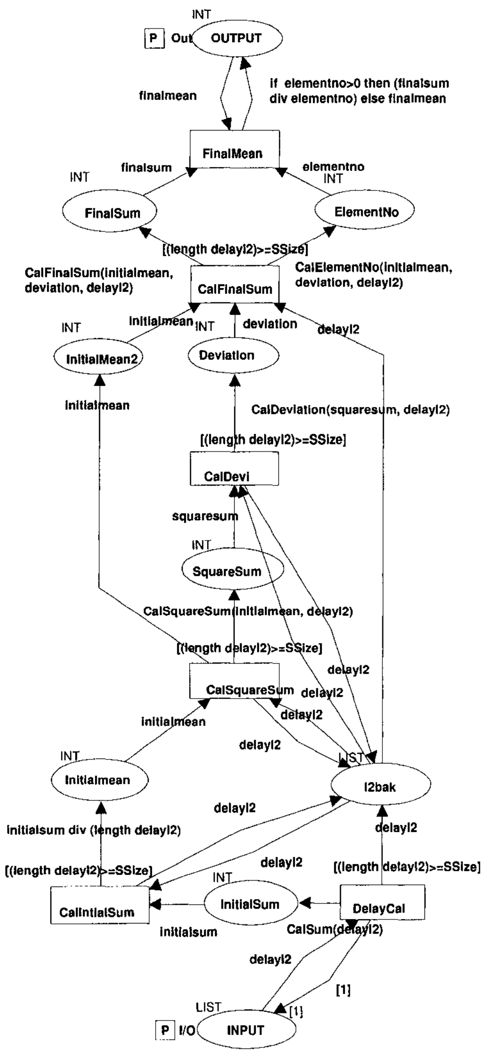
However, Δt_1, Δt_2 ...and Δt_n are not constant intervals. Due to the bursty nature of SACK TCP and network status, they might vary significantly. Some extreme large Δt might introduce significant errors into r_B. The Chebyshev Inequality provides a simple but effective way to remove the outliers in a set of observations.

Chebyshev Inequality [89]: The probability that a standardized random variable has absolute magnitude greater than or equal to some positive number k is always less than or equal to $1/k^2$. Namely,

$$P((|X-\mu| / \sigma) >= k) <= 1/k^2 \qquad (4.5)$$

For example, if k=3, then 1/9 of all the observations of a random variable fall out of the range

(μ-3σ, μ+3σ) no matter what the distribution the random variable follows. In our experiment, after getting Δt_1, Δt_2 ...Δt_n and Δd_1, Δd_2 ...Δd_n, we first apply Chebyshev Inequality to remove outliers. We then use Eq. 4.4 to calculate r_B with the remaining data. The resulting Petri net model is illustrated in Figure 4.11.



INT
P
Out
OUTPUT
finalmean
if elementno>0 then (finalsum div elementno) else finalmean
FinalMean
finalsum
INT
elementno
INT
FinalSum
ElementNo
[(length delayl2)>=SSize]
CalFinalSum(initialmean, deviation, delayl2)
CalElementNo(initialmean, deviation, delayl2)
CalFinalSum
INT
initialmean
INT
deviation
delayl2
InitialMean2
Deviation
initialmean
CalDeviation(squaresum, delayl2)
[(length delayl2)>=SSize]
CalDevi
squaresum
INT
SquareSum
CalSquareSum(initialmean, delayl2)
[(length delayl2)>=SSize]
delayl2
CalSquareSum
delayl2
initialmean
delayl2
delayl2
INT
LIST
Initialmean
l2bak
delayl2
initialsum div (length delayl2)
delayl2
[(length delayl2)>=SSize]
delayl2
INT
[(length delayl2)>=SSize]
CalInitialSum
InitialSum
DelayCal
initialsum
CalSum(delayl2)
delayl2
[1]
LIST
[1]
P
I/O
INPUT

Figure 4.11 Petri Net Model of Chebyshev Inequality

Table 4.4 illustrates how to apply Chebyshev Inequality to remove outliers. The initial series of Δt is from the case with 250ms RTT. Apparently the first Δt represents a burst and it should be removed as an outlier. After getting μ and σ, each Δt is compared with the range (-48.86, 94.12), all Δt outside the range should be considered to be outliers and removed from the series. In this example, only the first Δt is removed. Note that when Δt_i is removed from the sample, the corresponding Δd_i should also be discarded.

Table 4.4 Chebyshev's Inequality Example

Items	Data
Initial Δt Series (ms)	94.48, 15.51, 15.58, 15.72, 15.57, 15.51, 15.54, 15.58, 15.51, 15.56, 15.52
μ	22.73
σ	23.79
(μ-3σ, μ+3σ)	(-48.65, 94.12)
Outliers	94.48
Final Δt Series (ms)	15.51, 15.58, 15.72, 15.57, 15.51, 15.54, 15.58, 15.51, 15.56, 15.52

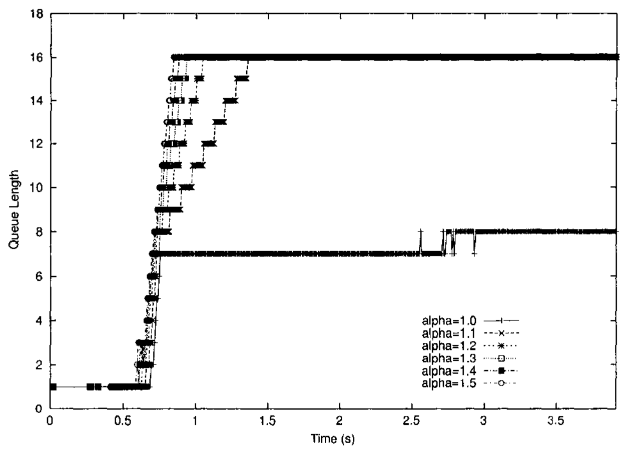


Figure 4.12 Queue Length for Different α (50 ms RTT, α-min Paced TCP)

α-min Paced SACK TCP including Chebyshev Inequality rejection of outliers was modeled with Design/CPN. The final Petri net is simply a combination of Figure 3.2 and Figure 4.11. Our simulation results show that, as expected, the performance of the file transfer in the case with 250 ms RTT remains the same with α-min Paced SACK TCP as with α-paced SACK TCP. α-paced SACK TCP has already improved the resilience of TCP in that case. However, the number of segments queued at PA is significantly reduced with α-min Paced SACK TCP for the 50 ms RTT case (see Figure 4.12). As mentioned in Section 4.2.1.1, α should be a positive number that is greater than 1 in order to avoid possible pacing rate regression. Note that our experiments included the scenario in which α equals 1.0 only for the purpose of comparison. According to

Table 4.3 and Figure 4.12, in the experimental networks, the smaller α is, the better the performance of α-min Paced SACK TCP is. Among the values studied in our experiments, choosing α= 1.1 leads to the best resilience performance.

Table 4.5 Occurrence Graph Data for SACK TCP, α-Paced TCP and α-min Paced TCP

Occurrence Graph Statistics		SACK TCP		α-Paced TCP		α-min Paced TCP	
Overview	Nodes	11603		11209		18093	
	Arcs	15995		15201		25922	
Integer Bounds	Typical Places	Upper	Lower	Upper	Lower	Upper	Lower
	Sender.SND_SEG	2	0	2	0	2	0
	Sender.RCV_ACK	1	1	1	1	1	1
	Sender.CWND	1	1	1	1	1	1
	Sender.SSTHRESH	1	1	1	1	1	1
	Sender.Rate-E	N/A	N/A	1	1	1	1
	Sender.Rate-B	N/A	N/A	N/A	N/A	1	1
	Receiver.RCV_SEG	2	0	2	0	2	0
	Receiver.SND_ACK	1	0	1	0	1	0
	Receiver.Q_LENGTH	1	1	1	1	1	1
Upper Multi-set Bounds	Sender.SSTHRESH	1`45		1`45		1`45	
	Receiver.Q_LENGTH	1`0+1`1+1`2+ 1`3+1`4+1`5+ 1`6+1`7+1`8+ 1`9+1`10+1`11+ 1`12+1`13+ 1`14+ 1`15+ 1`16+ 1`17+1`18		1`0+1`1+1`2+ 1`3+1`4+1`5+ 1`6+1`7+1`8+ 1`9+1`10+1`11+ 1`12+1`13+ 1`14+ 1`15+ 1`16+ 1`17+1`18		1`0+1`1+1`2+ 1`3+1`4+1`5+ 1`6+1`7+1`8+ 1`9+1`10+1`11+ 1`12+1`13+ 1`14+ 1`15+ 1`16+ 1`17+1`18	
Lower Multi-set Bounds	Sender.SSTHRESH	1`45		1`45		1`45	
	Receiver.Q_LENGTH	empty		empty		empty	
Liveness Properties	Dead Markings	[11603]		[11209]		[18093]	
	Dead Transitions	Sender.RTX_SEG1 Sender.RTX_SEG2 Sender.TIMEOUT Sender.TO_CHPA		Sender.RTX_SEG1 Sender.RTX_SEG2 Sender.TIMEOUT Sender.TO_CHPA		Sender.RTX_SEG1 Sender.RTX_SEG2 Sender.TIMEOUT Sender.TO_CHPA	

4.2.2 Formal Analysis of α-min Paced SACK TCP

We used Design/CPN to calculate the occurrence graph and analyze the boundedness and reachability of our models. We constructed different occurrence graphs for three typical cases: SACK TCP, α-paced SACK TCP with 50 ms RTT and α-min Paced SACK TCP with 50 ms RTT. The properties of these graphs are summarized in Table 4.5. The "Overview" section in Table 4.5 summarizes the number of nodes and arcs in each of the three occurrence graphs. For clarity, only the boundedness property of some major places is included in Table 4.5. Based on the occurrence graph data and applying the same analysis method as in Section 4.1.3, we also verified that the α-paced SACK TCP and α-min Paced SACK TCP are bounded, and the final step of the successful transfer is always reachable from the initial state. Thus, the correctness of α-min Paced SACK TCP is also verified by formal analysis.

Chapter 5

A Hybrid Method for Protocol Development

As mentioned in Section 2.3, formal methods enable us to verify the protocol of interest. However, formal verification of a single protocol can be quite difficult, while a new protocol really needs to be tested in a relatively realistic environment in which it interworks (or at least co-exists) with earlier or different versions of the same or similar protocols. Simulation excels in meeting such challenges. However, the correctness of a protocol can never be proved by simulation alone. In this chapter we present a hybrid protocol development methodology of embedding formal models into simulation that combines the use of formal methods and simulation to obtain the advantages of deep formal verification with the broad spectrum testing of simulation. In our research, Petri net modeling and ns-2 are chosen to be the example formal method and simulation software, respectively, to illustrate the hybrid methodology. In addition, α-min Paced SACK TCP, is used as the example protocol under test. However, the proposed hybrid methodology applies to other formal methods and simulation software. Also, the example of combining Petri nets with ns-2 is a generic approach that can be that can be used to study the performance of any new network protocol that is intended to run in an existing, complex network environment.

5.1 Embedding Petri Net Models into ns-2

Formal methods can be used to model and analyze a single communicating pair of protocol entities in great detail, while simulation allows us to create and simulate complex network scenarios containing many protocol entities, but at a lower level of detail and with no formal analysis. If we can make use of the advantages of these two types of methods, the protocol of interest will be studied more thoroughly and the research results will be more convincing. In our research, we use Petri nets and ns-2 as the example formal method and simulation software to illustrate the innovative hybrid methodology for protocol development, embedding formal models into simulation. Thus, the hybrid approach becomes a specific methodology, embedding Petri net models into ns-2. In Chapter 5, α-min Paced SACK TCP is used as the example protocol under investigation. However, embedding Petri net models into ns-2 is a general method that can be used to study the performance of various network protocols. This section presents an overview of this methodology. A detailed example using this methodology will be illustrated in the following sections.

We first model the protocol of interest with Petri nets, then analyze the formal properties of the model, and finally embed the Petri net model in an ns-2 simulation and carry out the simulation study. Note that most Petri net tools support simulation mode, too. However, they typically only work with a single pair of communicating entities. The results from such studies may not be convincing when the new protocol must run in a complex network with hundreds to thousands of concurrent sessions that use a variety of protocol revisions. For example, the active TCP sessions in a real network would use some mix of Tahoe, Reno, New Reno and SACK versions of the protocol. This degree of variety should be tested in any simulation scenario that is attempting to be realistic. With the proposed methodology, we only need to model the protocol that interests us with Petri nets. After the Petri net has been formally analyzed, the model is embedded into an ns-2 simulation environment and extensive simulations are carried out. This method aims to make use of the complementary advantages of Petri nets and simulation by embedding a formally verified Petri net model into a complex, representative ns-2 simulation environment.

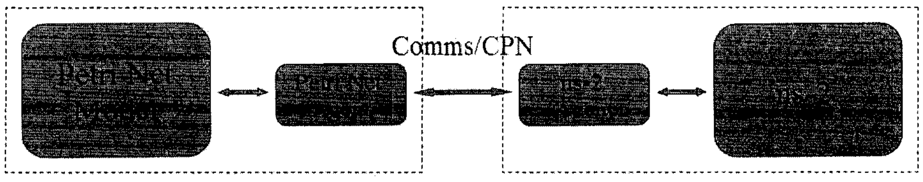


Figure 5.1 Embedding a Petri Net Model into ns-2

A high-level overview of the proposed methodology is illustrated in Figure 5.1. As mentioned previously, to study the performance of a network protocol, we need to make a Petri net model of the protocol first. Furthermore, to embed the Petri net model into ns-2, we need to develop two agents: a Petri net agent and an ns-2 agent. They are attached to the Petri net model and ns-2, respectively. These agents are used to facilitate the interaction between the Petri net model and ns-2. The detailed implementation of the agents is as follows.

To establish the interaction between a Petri net model and ns-2, there has to be a method for them to communicate. Design/CPN [62] is one of the most widely used Petri net modeling tools. Design/CPN has a special external communication library called "Comms/CPN" [90]. Comms/CPN is based on socket programming. It provides a way for Petri net models to communicate with non-Petri-net models. That is, any external process that is capable of sending or receiving messages via sockets can exchange information with Petri net models via Comms/CPN. Of course, there are other methods to transfer messages between Petri net models and ns-2 if we have access to the source codes of Design/CPN and ns-2. However, since socket programming is widely implemented in different operating systems, using "Comms/CPN" to exchange information between ns-2 and Petri net models guarantees the feasibility of the proposed methodology.

Once there is a way to communicate, more advanced issues must be taken into consideration. Synchronization is an important issue we have to address when we attempt to make two discrete-event systems, such as Petri nets and ns-2, interact. Suppose, for example, that there are 4 possible events in a simulation: A1, A2, A3, and A4. Suppose also that we have a Petri net model with 3 events: B1, B2, and B3, and that the events in both the simulation and the Petri net take place sequentially. If the combined execution sequence is A1, B1, A2, B2, B3, A3, A4, then some synchronization mechanism must be in place to guarantee the execution sequence in the combined system. This is shown in Figure 5.2.

Basically there are two types of mechanisms that can be used to guarantee synchronization: tick-based and event-based. A tick is a time unit defined in a specific system, such as 1 second or 0.5 second. The length of a tick depends on specific systems. With a tick-based mechanism, whenever a tick has passed, a tick-end notice is sent out to synchronize Petri nets and ns-2. This mechanism is correct but involves many synchronization notices, making it less practical. With an event-based mechanism, an event-end notice is sent only at the end of each event. For instance, in the system shown in Figure 5.2, at the end of A1, ns-2 will notify Petri net that A1 has been completed and B1 can be executed. After sending out the notice, ns-2 is suspended and waits for the notice from the Petri net. When B1 is completed, the Petri net will send a notice to ns-2 so that the execution of A2 can begin. This process guarantees the correct ordering of the combined execution sequence.

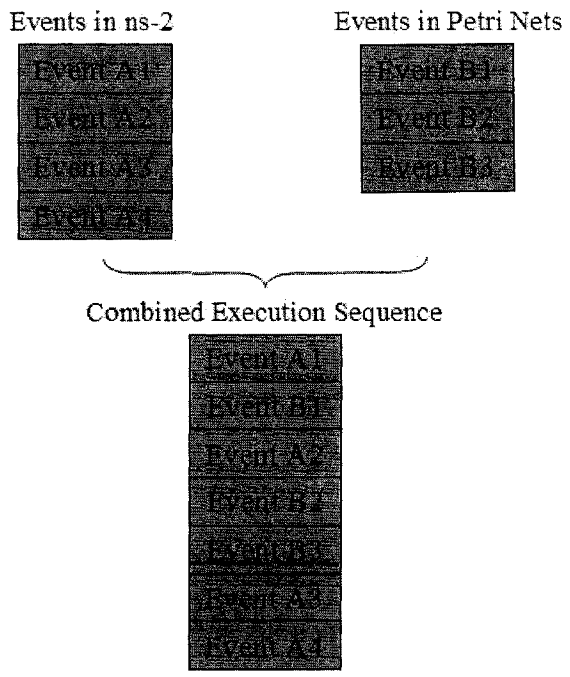


Figure 5.2 Synchronization between Two Discrete-Event Simulation Systems

Other than synchronization information, semantic information, such as packet sequence number, packet length, etc. is also exchanged between ns-2 and the Petri net to carry out the simulation. In practice, synchronization and semantic information are both encapsulated into a single message which is sent out whenever an event takes place. After receiving the message, the receiver, either ns-2 or the Petri net, will process the message accordingly. This leads to an event-based message exchange mechanism. Figure 5.3 illustrates how messages are processed with such a mechanism.

Once the problems of communication, synchronization, and semantic information exchange are solved, we can embed a formally analyzed Petri net model into an ns-2 simulation of any complexity and study the protocol of interest thoroughly. Finally, it is worth mentioning that although this section is focused on embedding Petri nets into ns-2, the hybrid methodology of embedding formal models into simulation is very general and it actually applies to many other modeling tools. For example, we might want to embed a PROMELA model into ns-2 or embed a Petri net into OPNET.

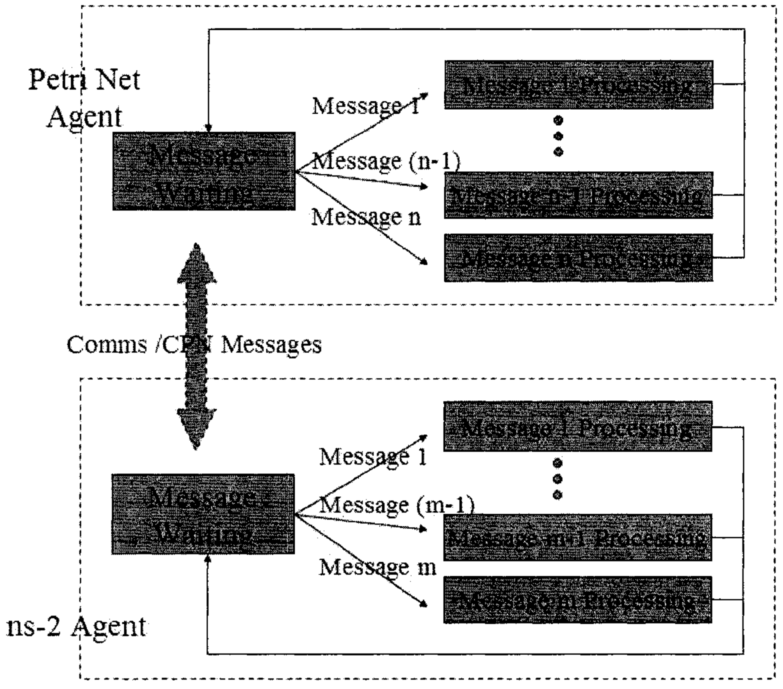
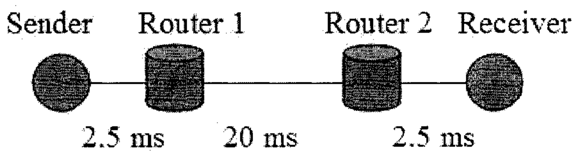


Figure 5.3 A General Event-Driven Message Exchange Mechanism

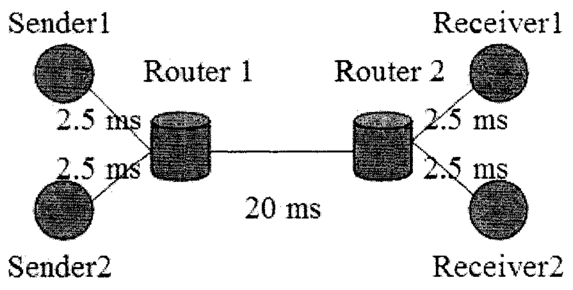
5.2 Simulation

As illustrated in Chapter 4, compared with SACK TCP, α-min Paced SACK TCP was proved to be very resilient in the case of a single TCP session. In this section, the proposed hybrid methodology was used to test the resilience performance of α-min Paced SACK TCP in a complex network environment. We were able to study three types of scenarios: (1) all TCP sessions in the network use α-min Paced SACK TCP, (2) all TCP sessions use ns-2's standard SACK TCP and (3) the scenario in which half of the sessions use α-min Paced SACK TCP and the other half use standard SACK TCP. This gives us results that show how our proposed change to TCP would behave if it were the only version of TCP in the network, and more importantly, helps us understand how the new version interworks with an existing, widespread version of TCP.

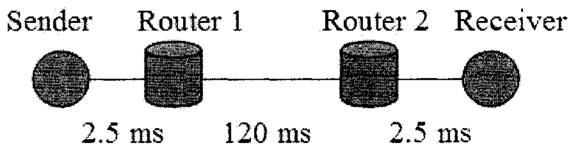
As stated previously, α-min Paced SACK TCP should pace packets at [α x min (r_E , r_B)]. However, it is not easy to determine r_B when there are multiple TCP sessions. For simplicity, we assume that the queuing policy employed by the routers in our simulation is "fair queuing" [2]. Thus, if there are n TCP flows passing a bottleneck link whose bandwidth is b_B, then for each TCP session, r_B can be set to (b_B / n). Note that (b_B / n) is the lower limit since r_B is always no less than (b_B / n). Furthermore, in our simulation, α is set to 1.



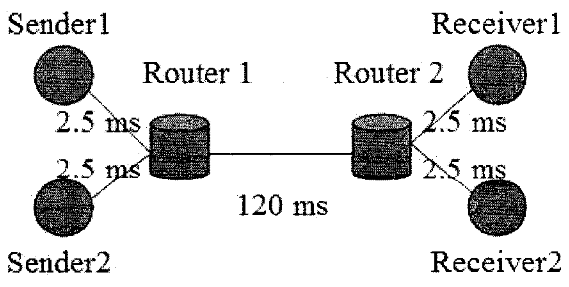
(a)



(B)



(C)



(D)

Figure 5.4 Basic Simulation Topologies

In our research, two sets of simulations were carried out. The first set of simulations correspond to the basic scenarios in which there are at most two TCP sessions at any time and the concurrent TCP sessions have the same RTT, either 50 ms or 250ms. Four network topologies were used and are shown in Figure 5.4. Two routers, Router 1 and Router 2, are connected by a core network link. To simulate different delay conditions, the one-way delay of the core network link is either 20 ms or 120 ms. The bandwidth of the core link is DS1, namely, 1.544 Mbps. One or two TCP senders (or receivers) are connected to Router 1 (or Router 2) by 10 Mbps edge links. The delay of the edge links is fixed at 2.5 ms. Thus, in the first set of simulations, the round-trip time for any TCP session is either 50 ms or 250 ms. We selected these values to be similar to the round-trip times in continental and transoceanic networks, respectively.

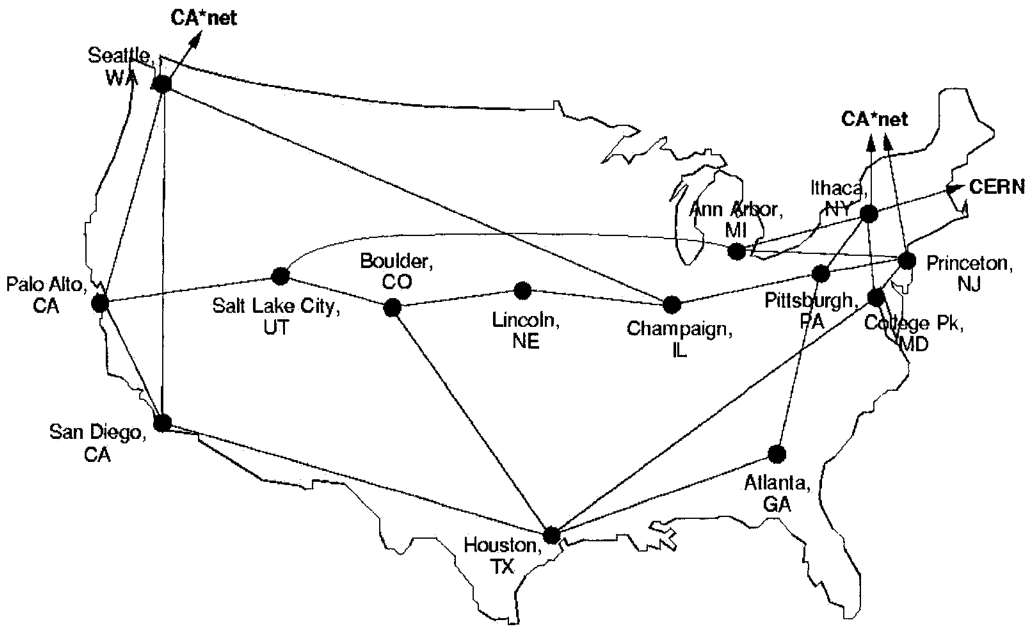


Figure 5.5 NSFNET-Like Network Topology [91]

The second set of simulations were configured to study whether α-min Paced SACK TCP works well in a more realistic situation, namely, when there are more than two concurrent TCP sessions and these TCP sessions do not have the same RTT. In the second set of simulations, a NSFNET-like topology [91] is used, as shown in Figure 5.5. Fourteen routers, each of which corresponds to a node in Figure 5.5, form the core network. The bandwidth of the core link is set to DS1 (1.544 Mbps) so that we can compare the results to those from the first set of simulations. Due to varied geographical distances, the links between these routers have different delays. Assuming that the NSFNET-like network is a fiber optic network in which light travels at 0.66c, we can calculate the corresponding delays due to geographical distances. The details are summarized in Table 5.1. For clarity, only the links that are used by different TCP sessions in the second set of simulations are included. There are 8 TCP sessions altogether in the NSFNET-like network. The senders are all attached to the router at Palo Alto via "10Mbps-2.5ms" edge links. The same type of link is used to connect receivers with the routers at Boulder, Lincoln, Champaign, Pittsburgh, Princeton, College Park, Ann Arbor, and Houston, respectively. Figure 5.6 illustrates the details. The dashed lines in Figure 5.6 indicate that these lines represent multiple hops rather than a single link. The detailed routes for the TCP sessions in the second set of simulations are summarized in Table 5.2.

Table 5.1 Link Delays in the NSFNET-Like Network

Link Start	Link End	Distance (km)	Delay (ms)
Palo Alto	Salt Lake	1230.885	6.154425
Salt Lake	Boulder	843.116	4.21558
Boulder	Lincoln	807.718	4.03859
Lincoln	Champaign	891.386	4.45693
Champaign	Pittsburgh	767.493	3.837465
Pittsburgh	Princeton	540.624	2.70312
Princeton	College Park	286.402	1.43201
Salt Lake	Ann Arbor	2606.58	13.0329
Boulder	Houston	1671.751	8.358755

In our simulation, a sender (or receiver) can be either an ns-2 "SACK TCP" sender (or receiver) or a Petri net "α-min Paced SACK TCP" sender (or receiver). Thus, for the same network topology, there could be several different simulation scenarios. In this section, NS-S and NS-R denote ns-2 "SACK TCP" sender and receiver, respectively, while PE-S and PE-R denote

Petri net "α-min Paced SACK TCP" sender and receiver, respectively. Also, a TCP sender and receiver of the same type are paired. For example, if there is one NS-S in a simulation, then there must be one and only one corresponding NS-R. Different combinations of NS-S, NS-R, PE-S, and PE-R using the basic topologies shown in Figure 5.4 and the NSFNET-like topology shown in Figure 5.5 were studied, and the experimental results are presented in Section 5.3.

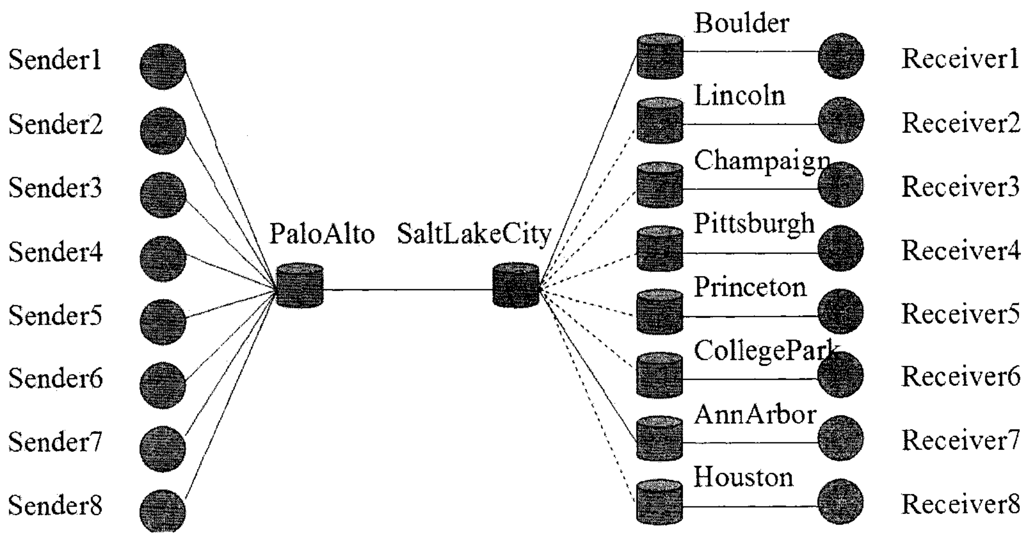


Figure 5.6 TCP Sessions in the NSFNET-Like Network

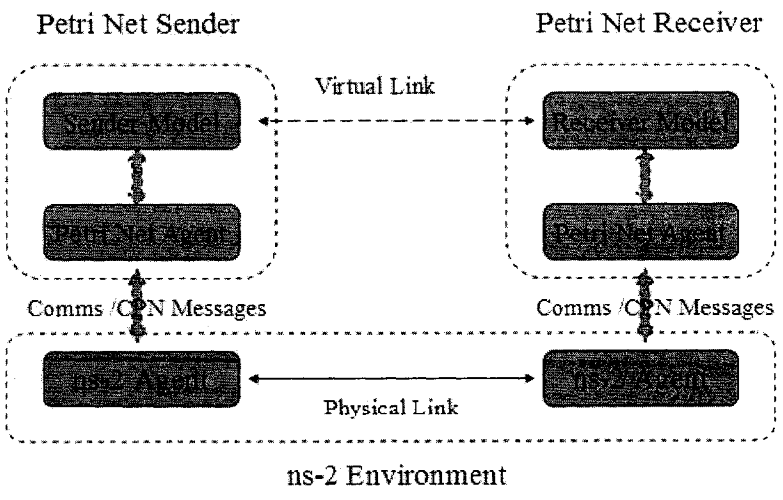


Figure 5.7 Physical and Virtual Links

Table 5.2 Routes of the TCP Flows in the NSFNET-Like Network

Flow ID	Sender Location	Receiver Location	Route
1	Palo Alto	Boulder	PaloAlto>SaltLakeCity> Boulder
2	Palo Alto	Lincoln	PaloAlto> SaltLakeCity> Boulder> Lincoln
3	Palo Alto	Champaign	PaloAlto>SaltLakeCity>Boulder>Lincoln>Champaign
4	Palo Alto	Pittsburgh	PaloAlto>SaltLakeCity>Boulder>Lincoln>Champaign >Pittsburgh
5	Palo Alto	Princeton	PaloAlto>SaltLakeCity>Boulder>Lincoln>Champaign >Pittsburgh>Princeton
6	Palo Alto	College Park	PaloAlto>SaltLakeCity>Boulder>Lincoln>Champaign >Pittsburgh>Princeton>CollegePark
7	Palo Alto	Ann Arbor	PaloAlto>SaltLakeCity>AnnArbor
8	Palo Alto	Houston	PaloAlto>SaltLakeCity>Boulder>Houston

The NS-S and NS-R protocol entities are implemented by ns-2 and to incorporate them in a simulation we do not have to make any change to the corresponding ns-2 model. However, including PE-S or PE-R entities means that a synchronization and semantic information exchange mechanism has to be in place to trade necessary information between the Petri net model and ns-2. The general method to solve the embedding problem was discussed in Section 5.1. We present the detailed implementation of the method in this section. As mentioned previously, a Petri net agent and an ns-2 agent are required. We implemented the mechanism illustrated in Figure 5.7 to complete a TCP session between a PE-S and a PE-R. Each PE-S and PE-R consists of code running in both Design/CPN and ns-2. There is an agent on each "side" that passes information across the program boundaries. That is, each sender or receiver has both an ns-2 agent and a Petri net agent. Comms/CPN is used to transfer messages between these two agents. The network that transfers TCP packets from the PE-S to the PE-R is simulated in ns-2 between the ns-2 agents. The need for a mechanism to synchronize the behaviors of the ns-2 and Petri net models was also discussed in Section 5.1. The mechanism we used to synchronize the models is based on three types of messages: Send, ACK and Timeout (see Figure 5.8).

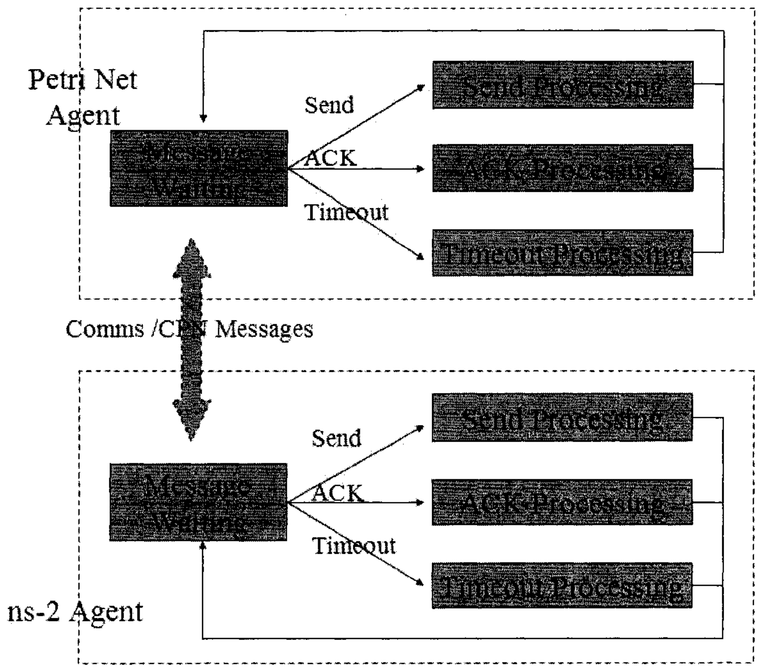


Figure 5.8 A Specific Event-Driven Message Exchange Mechanism

In the study that motivated the development of this method, we study the resilience of TCP after it has been transmitting for awhile and reaches a stable state where *cwnd* becomes greater than *rwnd.* We chose a 1 MB file transfer from the sender to the receiver for every experimental TCP session. So there is always some application data for TCP during the file transfer. Namely, whenever TCP is ready to send a segment, it can transmit the segment immediately. SMSS was set to 1460 bytes. The values of *cwnd*, *rwnd* and *ssthresh* were initialized to 1460 bytes, 32 KB and 64 KB, respectively. By 32 KB or 64 KB, we mean a multiple of SMSS that is just above 32 KB or 64KB. For example, in our model, SMSS is 1460 bytes, so by 32 KB, we mean 1460 * 23= 33580 bytes. For clarity, we consider 1460-byte packets as the data units. Thus, *cwnd*, *rwnd* and *ssthresh* were set to 1, 23 and 45, respectively.

5.3 Experimental Results

α-min Paced SACK TCP spreads outstanding packets along the path and thus reduces the number of packets queued at the bottleneck router, decreasing the possibility of packet discard due to limited buffer space. For the topologies shown in Figure 5.4, Router 1 turned out to be the bottleneck router. For the NSFNET-like topology, Palo Alto is the bottleneck. If there is too much traffic in the network, an excessive number of packets will be queued at Router 1 or Palo Alto. We assume that all routers have sufficient memory to buffer all incoming packets so that no packets are dropped due to the length of the queue. Thus, in our simulations, the length of the queue at Router 1 or Palo Alto indicates the resilience of different versions of TCP. That is, a shorter queue indicates better performance. The detailed experimental results from our simulations are summarized as follows.

5.3.1 Continental Networks: Uniform 50 ms Round-trip Time

With the topologies shown in Figure 5.4 (a) and (b), the round-trip-time is 50 ms. For a network corresponding to Figure 5.4 (a), there are two possible cases: (1 PE-S, 1 PE-R) and (1 NS-S, 1 NS-R). That is, there is only one α-min Paced SACK TCP session or one ns-2 SACK TCP session. The experimental results from these scenarios are shown in Figure 5.9. Apparently, if there is only one ns-2 SACK TCP session, a 15-packet queue is built up at Router 1 soon after the TCP session starts; comparatively, if there is only one α-min Paced SACK TCP session, there is at most a 1-packet queue.

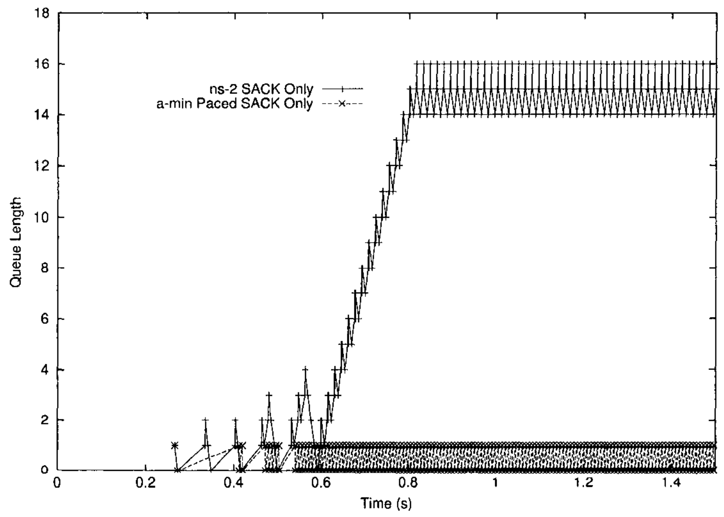


Figure 5.9 Single TCP Session with 50 ms RTT

For the network shown in Figure 5.4 (b), there are three possible cases: (1 PE-S + 1 NS-S, 1 PE-R + 1 NS-R) , (2 PE-S + 0 NS-S, 2 PE-R + 0 NS-R), and (0 PE-S + 2 NS-S, 0 PE-R + 2 NS-R), which correspond to the scenario of one α-min Paced SACK TCP session and one ns-2 SACK TCP session, two α-min Paced SACK TCP sessions, and two ns-2 SACK TCP sessions, respectively. The resilience performance from these scenarios is shown in Figure 5.10. In the case of two α-min Paced SACK TCP sessions, there is only a very short queue at Router 1. The case of two ns-2 SACK TCP sessions leads to the worst resilience performance because a long queue of around 36 packets appears shortly after the start of the simulation. In the case of one α-min Paced SACK TCP session and one ns-2 SACK TCP session, the outstanding resilience of α-min Paced SACK TCP is significantly held back by ns-2 SACK TCP. As a result, there is still a 20-packet queue.

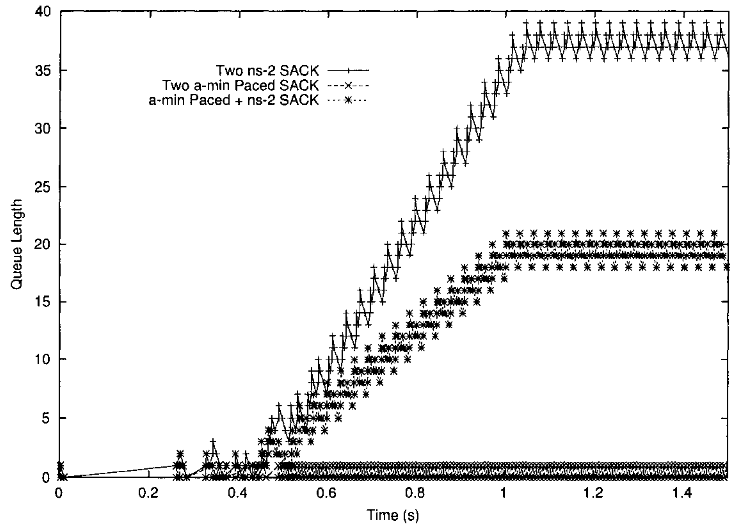


Figure 5.10 Two TCP Sessions with 50 ms RTT

5.3.2 Transoceanic Networks: Uniform 250 ms Round-trip Time

Figure 5.4 (c) and (d) are meant to represent transoceanic communication networks, in which longer round-trip times are often observed. In our simulations, the RTT is fixed at 250 ms. Similar to Figure 5.4 (a), Figure 5.4 (c) leads to two possible cases: (1 PE-S, 1 PE-R) and (1 NS-S, 1 NS-R), which correspond to the scenario of one α-min Paced SACK TCP session only, and one ns-2 SACK TCP session only, respectively. The experimental results from these scenarios are shown in Figure 5.11. The queue resulting from ns-2 SACK TCP in this case is not as long as before: after a few seconds, the length of the queue drops to 1 packet. This is because there is not much traffic in the network in this scenario. Compared with ns-2 SACK TCP, α-min Paced SACK

TCP leads to a very short queue throughout the session.

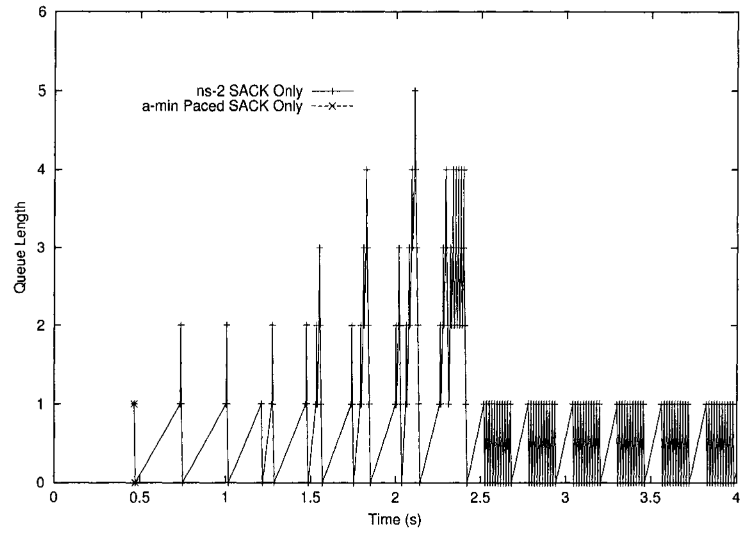


Figure 5.11 Single TCP Session with 250 ms RTT

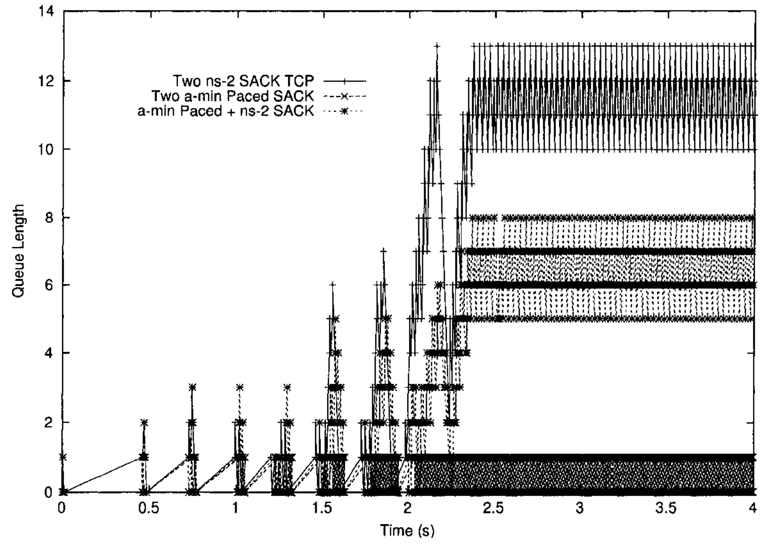


Figure 5.12 Two TCP Sessions with 250 ms RTT

Similar to Figure 5.4 (b), with the topology shown in Figure 5.4 (d), there are three possible cases: (1 PE-S + 1 NS-S, 1 PE-R + 1 NS-R) , (2 PE-S + 0 NS-S, 2 PE-R + 0 NS-R), and (0 PE-S + 2 NS-S, 0 PE-R + 2 NS-R), which correspond to the scenario of one α-min Paced SACK TCP session and one ns-2 SACK TCP session, two α-min Paced SACK TCP sessions, and two ns-2 SACK TCP sessions, respectively. The resilience performance from these scenarios is shown in

Figure 5.12. Similarly, we found that the case of two α-min Paced SACK TCP sessions leads to the best resilience, the case of two ns-2 SACK TCP sessions results in the worst resilience, and the length of the queue in the case of one α-min Paced SACK TCP session and one ns-2 SACK TCP session is somewhere in between.

5.3.3 NSFNET-Like Networks: Varied Round-trip Time

In the second set of simulations, there are eight TCP sessions in the NSFNET-like network. Each sender-receiver pair could be either α-min Paced SACK TCP or ns-2 SACK TCP, so there are 2^8 different cases in total. We chose three typical cases to study the performance of α-min Paced SACK TCP. In the first case all eight TCP sessions are ns-2 SACK TCP sessions, and in the second all TCP sessions are Petri net α-min Paced SACK TCP sessions. The third case is composed of four Petri net α-min Paced SACK TCP sessions and four ns-2 SACK TCP sessions. Specifically, the TCP sessions whose receivers are at Boulder, Lincoln, Champaign, and Pittsburgh are Petri net α-min Paced SACK TCP sessions; the TCP sessions whose receivers are at Princeton, College Park, Ann Arbor, and Houston are ns-2 SACK TCP sessions.

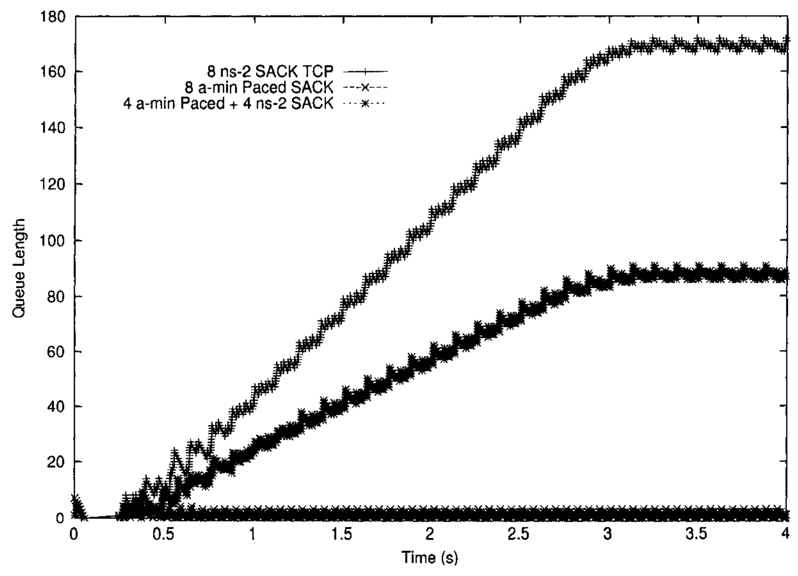


Figure 5.13 Eight TCP Sessions with Varied RTTs in a NSFNET-Like Network

As in the simpler networks, we find that if all TCP sessions use α-min Paced SACK TCP, only a few packets are queued at the bottleneck router in Palo Alto. If all TCP sessions use ns-2

SACK TCP, the length of the queue at Palo Alto increases from the beginning of the run to about 170. If half of the sessions use α-min Paced SACK TCP and the other half use ns-2 SACK TCP, an intermediate state is reached in which the length of the queue at the bottleneck router reaches 90. The detailed results are illustrated in Figure 5.13.

Hence, we can conclude that α-min Paced SACK TCP performs better than SACK TCP in terms of resilience, not only in single session scenarios, but also in multiple session cases, even if these TCP sessions have varied RTTs. We were able to reach this conclusion by using the proposed hybrid methodology to study the resilience of α-min Paced SACK TCP in a relatively complex environment. We can have confidence in the results because our simulations use the formally verified model of our proposed variation of TCP, rather than another representation of it programmed independently. The methodology proposed here makes it possible to test the performance of a verified module in a more complex environment than would ordinarily be employed in a formal verification environment, whilst retaining the certainty of correctness.

Chapter 6

Conclusions and Future Work

In this chapter we present our conclusions and future work. Specifically, Section 6.1 gives the conclusions and Section 6.2 talks about the future work.

6.1 Conclusions

We studied the resilience behavior of SACK, NewReno, and Reno TCP. Our experimental results demonstrate that the traditional 50 ms recovery time is not suitable for TCP transport on the Internet. With SACK TCP, we found two restoration objectives, $\tau_{1,S}$ and $\tau_{2,S}$. $\tau_{1,S}$ is given by Eq. 3.10, and $\tau_{2,S}$ is closely related to RTO. With NewReno TCP, we also found two restoration objectives, $\tau_{1,NR}$ and $\tau_{2,NR}$. $\tau_{1,NR}$ is given by Eq. 3.12, and $\tau_{2,NR}$ is essentially the same as $\tau_{2,S}$. $\tau_{1,NR}$ is approximately twice as large as $\tau_{1,S}$ when *rwnd* is large. With Reno TCP, we found a recovery objective of 200ms for DS0 access; For DS1 and OC-3c access, there seem to be no critical points from 15ms to 1s.

We found that if more than $n_{C,S}$ segments in one transmission window are lost, even if SACK TCP transitions into Fast Retransmit/Fast Recovery, a timeout will still occur and performance will be degraded dramatically. To avoid unnecessary timeout with SACK TCP, we proposed Type I and Type II Probing that insert probes into the network in the case of network malfunctions. Our experimental results show that both Type I and Type II Probing decrease the impact of lost segments significantly.

SACK TCP tends to send out segments in clusters and congest bottleneck routers, which could potentially degrade SACK TCP resilience significantly. α-min Paced SACK TCP, a paced version of SACK TCP based on the estimation of bottleneck rate, was proposed to spread out segments over the path between the sender and the receiver. Our experimental results demonstrate that α-min Paced SACK TCP spreads out segments and reduces the number of segments queued at the bottleneck routers, avoiding potential segment losses due to congestion. SACK TCP with Type I and Type II Probing, as well as α-min Paced SACK TCP, were modeled with Petri nets. The Petri net models were formally analyzed to increase out confidence about the correctness.

Formal methods enable us to verify the protocol of interest. While simulation excels in testing a new protocol in relatively realistic environment in which it interworks (or at least co-exists) with earlier or different versions of the same or similar protocols. We proposed a hybrid protocol development methodology, embedding formal models into simulation, that combines the use of formal methods and simulation to obtain the advantages of deep formal verification with the broad spectrum testing of simulation. Petri net modeling and ns-2 were chosen to be the example formal method and simulation software, respectively, to illustrate the hybrid methodology. With the assistance of the hybrid method, we tested α-min Paced SACK TCP in a NSF-like network. The results show that α-min Paced SACK TCP performs better than SACK TCP in terms of resilience, not only in single session scenarios, but also in multiple session cases, even if these TCP sessions have varied RTTs.

6.2 Future Work

TCP is the de facto Transport Layer protocol in the Internet. As the Internet evolves, there are numerous possibilities for future work:

- TCP was designed for low-speed networks. As the access rate keeps increasing, TCP needs to evolve accordingly. For example, when a packet loss event is detected by receiving three duplicate ACKs, the current version of TCP halves its *cwnd* after it recovers from the malfunction. For high-speed networks, it takes a long time for *cwnd* to climb back to the value that was in use before the malfunction. This could reduce bandwidth utilization significantly. TCP resilience in high-speed networks could be a promising research area.
- In Type II Probing, max (300 ms, 2RTT) is chosen as the length of the "quiet period". We mentioned previously that the length of this period should be related to RTT and 300 ms was

selected because the TCP receiver only sends out an ACK for every second segment, or within 200 ms of the arrival of the first unacknowledged segment. Further study and optimization is required.

- Our experimental results show that SACK TCP with Type I and Type II Probing improves SACK TCP resilience significantly in single session scenarios. Further study of the improved versions of SACK TCP in more complex environments needs to be accomplished to verify their effectiveness.
- Embedding Petri net models into ns2 is used to demonstrate the proposed hybrid method for protocol development. The idea of the hybrid method is a generic approach that can be used with other formal methods and simulation software. However, embedding a specific formal model into a specific simulation environment involves designing the interface between them manually. A more automatic approach needs to be developed to speed up the process and reduce possible errors.

Bibliography

[1] W.R. Stevens, *TCP/IP Illustrated, Volume 1*, Addison Wesley, 2000.

[2] A.S. Tanenbaum, *Computer Networks: 4th edition*, Prentice-Hall, 2003.

[3] K. Fall and S. Floyd, *Simulation-based Comparisons of Tahoe, Reno and SACK TCP*, Computer Communication Review, Vol. 26, No.3, pp. 5-21, July1996.

[4] S. Floyd and T. Henderson, *RFC 2582: The NewReno Modification to TCP's Fast Recovery Algorithm*, April, 1999.

[5] M. Mathis et al., *RFC 2018: TCP Selective Acknowledgement Options*, October, 1996.

[6] S. Floyd et al., *RFC 2883: An Extension to the Selective Acknowledgement (SACK) Option for TCP*, July, 2000.

[7] C. Jin et al., *FAST TCP: From Theory to Experiments*, IEEE Network, Vol. 19, No.1, January/February, 2005.

[8] S. Floyd, *RFC 3649:HighSpeed TCP for Large Congestion Windows*, December 2003.

[9] M. Gerla et al., *TCP Westwood: Congestion Window Control Using Bandwidth Estimation,* Proceedings of IEEE Globecom 2001, pp. 1698-1702, San Antonio, Texas, USA, November 25-29, 2001.

[10] A. Zanella et al., *TCP Westwood: Analytic Model and Performance Evaluation,* Proceedings of IEEE Globecom 2001, pp. 1703-1707, San Antonio, Texas, USA, November 25-29, 2001.

[11] D. Katabi, M. Handley and C. Rohrs, *Congestion Control for High Bandwidth-Delay Product Networks*, ACM SIGCOMM Computer Communication Review, Vol. 32, No.4, pp. 89 - 102, August , 2002.

[12] K. Ramakrishnan, S. Floyd and D. Black, *RFC 3168: The Addition of Explicit Congestion Notification (ECN) to IP*, September, 2001.

[13] J. Padhye and S. Floyd, *On Inferring TCP Behavior,* Proceedings of ACM SIGCOMM 2001, pp. 287-298, San Diego, CA, USA, August 27-31, 2001.

[14] T. Wu, *Fiber Network Service Survivability*, Artech House, 1992.

[15] T.H. Wu, *Emerging Technologier for Fiber Network Survivability*, IEEE Communications Magazine, Vol. 33, No.2, pp. 58-74, February 1995.

[16] O. Gerstel, *Opportunities for Optical Protection and Restoration,* Proceedings of Optical Fiber Communication Conference, pp. 269-270, San Jose, CA, USA, February 22-27, 1998.

[17] S. Ramamurthy and B. Mukherjee, *Survivable WDM Mesh Networks: Part I Protection,* Proceedings of IEEE INFOCOM 1999, pp. 744-751, New York, NY, USA, March 21-25, 1999.

[18] A. Fumagalli et al., *Survivable Networks Based on Optimal Routing and WDM Self-Healing Rings,* Proceedings of IEEE INFOCOM 1999, pp. 726--733, New York, NY, USA, March 21-25, 1999.

[19] G. Sahin and M. Azizoglu, *Optical Layer Survivability: Single Service-Class Case,* Proceedings of OptiComm 2000: Optical Networking and Communications, pp. 267-278, Dallas, TX, USA, October 24, 2000.

[20] S. Arakawa, M. Murata and H. Miyahara, *Design Methods of Multilayer Survivability in IP over WDM Networks,* Proceedings of OptiComm 2000: Optical Networking and Communications, pp. 279-290, Dallas, TX, USA, October 24, 2000.

[21] O. Gerstel and R. Ramaswami, *Optical Layer Survivability - An Implementation Perspective*, IEEE Journal on Selected Areas in Communications, Vol. 18, No.10, pp. 1885-1899, October, 2000.

[22] G. Ellinas, A.G. Hailemariam and T.E. Stern, *Protection Cycles in Mesh WDM Networks*, IEEE Journal on Selected Areas in Communications, Vol. 18, No.10, pp. 1924-1937, October, 2000.

[23] S. Ramamurthy and B. Mukherjee, *Survivable WDM Mesh Metworks: Part II Restoration,* Proceedings of IEEE ICC 1999, pp. 2023-2030, Vancouver, BC, Canada, June 6-10, 1999.

[24] W.D. Grover, *The Self-Healing Network: A Fast Distributed Restoration Technique for Networks Using Digital Crossconnect Machines,* Proceedings of IEEE Globecom 1987, pp. 1090-1095, Tokyo, Japan, November, 1987.

[25] W.J. Goralski, *SONET(Second Edition)*, Mcgraw-Hill, 2000.

[26] ANSI, *T1.105.01-1998, Synchronous Optical Network(SONET)-Automatic Protection Switching*, 1998.

[27] T.E. Stern and K. Bala, *Multiwavelength Optical Networks*, Addison-Wesley, 1999.

[28] D. Clark, *The Design Philosophy of the DARPA Internet Protocols*, ACM SIGCOMM Computer Communication Review, Vol. 18, No.4, pp. 106-114, August, 1988.

[29] J. Moy, *RFC 2328: OSPF Version 2*, April, 1998.

[30] P. Demeester et al., *Resilience in Multilayer Networks*, IEEE Communications Magazine, Vol. 37, No.8, pp. 70 - 76, August, 1999.

[31] C. Sadler, L. Kant and W. Chen, *Cross-Layer Self-Healing Mechanisms in Wireless Networks,* Proceedings of 6TH WORLD WIRELESS CONGRESS, Palo Alto, CA, USA, May 24 - 27, 2005.

[32] C. Petri, *Kommunikation mit Automaten (Communication with Automata)*, PhD Thesis, University of Bonn, 1962.

[33] J. F.Kurose and K.W. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*, Addison Wesley, 2005.

[34] J. Mogul and S. Deering, *RFC 1191: Path MTU Discovery*, November, 1990.

[35] V. Paxson and M. Allman, *RFC 2988: Computing TCP's Retransmission Timer*, November, 2000.

[36] M. Allman, V. Paxson and W. Stevens, *RFC 2581: TCP Congestion Control*, April, 1999.

[37] S.Floyd, T. Henderson and A. Gurtov, *RFC 3782: The NewReno Modification to TCP's Fast Recovery Algorithm*, April, 2004.

[38] E. Blanton et al., *RFC 3517: A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP*, April, 2003.

[39] T. Murata, *Petri Nets: Properties, Analysis and Applications*, The IEEE Proceedings, Vol. 77, No.4, pp. 541-580, April, 1989.

[40] K. Jensen, *Colored Petri Nets: Basic Concepts (Volumn 1)*, Springer-Verlag, 1992.

[41] K. Jensen, *Colored Petri Nets: Analysis Methods (Volumn 2)*, Springer-Verlag, 1992.

[42] N.G. Leveson and J.L. Stolzy, *Safety Analysis Using Petri Nets*, IEEE Trasaction on Software Engineering, Vol. SE-13, No.3, pp. 386-397, 1987.

[43] S.C. Brofferio, *A Petri Net Control Unit for Highspeed Modular Signal Processors*, IEEE Trasaction on Communications, Vol. C35, No.6, pp. 577-583, 1987.

[44] J. Desel, *Basic Linear Algebraic Techniques for Place/Transition Nets*, Lectures on Petri Nets I: Basic Models (LNCS 1491), pp. 257-308, 1998.

[45] A. Valmari, *The State Explosion Problem*, Lectures on Petri Nets I: Basic Models (LNCS 1491), pp. 429-528, 1998.

[46] G. Rozenberg and J. Engelfriet, *Elementary Net Systems*, Lectures on Petri Nets I: Basic Models (LNCS 1491), pp. 12-121, 1998.

[47] J. Desel and W. Reisig, *Place/Transition Petri Nets*, Lectures on Petri Nets I: Basic Models (LNCS 1491), pp. 122-173, 1998.

[48] E. Smith, *Principles of High-Level Net Theory*, Lectures on Petri Nets I: Basic Models (LNCS 1491), pp. 174-210, 1998.

[49] C. Capellmann, H. Dibold and U. Herzog, *Using High-Level Petri nets in the Field of Intelligent Networks*, Application of Petri Nets to Communication Networks (LNCS 1605), pp. 1-36, 1999.

[50] G. Wheeler, *The Modelling and Analysis of IEEE 802.6's Configuration Congtrol Protocol with Colored Petri Nets*, Application of Petri Nets to Communication Networks (LNCS 1605), pp. 69-92, 1999.

[51] K. Jensen, *An Introduction to the Practical Use of Colored Petri Nets*, Lectures on Petri Nets I: Applications (LNCS 1492), pp. 237-292, 1998.

[52] T.S. A. B. Mnaouer, Y. Fujii, T. Ito, H. Tanaka, *Colored Petri Nets Based Modeling and Simulation of the Static and Dynamic Allocation Policies of the Asynchronous Bandwidth in teh Fieldbus Protocol*, Application of Petri Nets to Communication Networks (LNCS 1605), pp. 93-130, 1999.

[53] F. Babich and L. Deotto, *Formal Methods for Specification and Analysis of Communication Protocols*, IEEE Communication Surveys and Tutorials, Vol. 4, No.1, pp. 2-20, Third Quarter, 2002.

[54] S. Bajaj et al., *Improving Simulation for Network Research (Technical Report 99-702b, University of Southern California)*, March, 1999.

[55] ITU-T, *Recommendation Z.100: Specification and Description Language (SDL)*, 1999.

[56] European Telecommunications Standards Institute (ETSI), Available from:

http://www.etsi.org/.

[57] Third Generation Partnership Project (3GPP), Available from: http://www.3gpp.org/.

[58] SDL Proceedings Online, Available from: http://www.sdl-forum.org/Publications/Proceedings/Proceedings.htm.

[59] T. Bolognesi and E. Brinskma, *Introduction to the ISO Specification Language LOTOS*, Computer Networks and ISDN Systems, Vol. 14, No.1, pp. 92-100, April, 1987.

[60] K.J. Turner, *Using Formal Description Techniques: An Introduction to Estelle, LOTOS and SDL*, John Wiley, 1993.

[61] G.J. Holzmann, *Design and Walidation of Computer Protocols*, Prentice Hall, 1991.

[62] Design/CPN, The CPN Group, Department of Computer Science, University of Aarhus, Denmark, Available from: http://www.daimi.au.dk/designCPN/.

[63] L.M. Kristensen and K. Jensen, *Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad Hoc Networks*, ntegration of Software Specification Techniques for Applications in Engineering: Priority Program SoftSpez of the German Research Foundation (LNCS 3147), pp. 248-269.

[64] GloMoSim 2.0, Available from: http://pcl.cs.ucla.edu/projects/glomosim/.

[65] PARSEC, Available from: http://pcl.cs.ucla.edu/projects/parsec/.

[66] OPNET Modeler, Available from: http://www.opnet.com.

[67] ns-2, Available from: http://www.isi.edu/nsnam/ns/.

[68] VINT Project, Available from: http://www.isi.edu/nsnam/vint/index.html.

[69] SAMAN, Available from: http://www.isi.edu/saman/index.html.

[70] CONSER, Available from: http://www.isi.edu/conser/index.html.

[71] T.R. Henderson and R.H. Katz, *On Improving the Fairness of TCP Congestion Avoidance,* Proceedings of IEEE GlobeCom 1998, pp. 539-544, Sydney,NSW, Australia, November 8-12, 1998.

[72] K. Claffy, G. Miller and K. Thompson, *The Nature of the Beast: Recent Traffic Measurements from an Internet Backbone,* Proceedings of INET 1998, Geneva, Switzerland, July 21-24, 1998.

[73] A. Antonopoulos, *Metrication and Performance Analysis on Resilience of Ring-Based Transport Network Solutions,* Proceedings of GLOBECOM 1999, pp. 1551-1555, Rio de Janeiro, Brazil, December 5-9, 1999.

[74] L. Brakmo and L. Peterson, *TCP Vegas: End to End Congestion Avoidance on a Global Internet*, IEEE Journal on Selected Areas in Communications, Vol. 13, No.8, pp. 1465-1480, October, 1995.

[75] J. Hoe, *Improving the Start-up Behavior of a Congestion Control Scheme for TCP,* Proceedings of ACM SIGCOMM 1996, pp. 270-280, Stanford, CA, USA, August 28-30, 1996.

[76] D.J. Chaly and V.A. Sokolov, *An Extensible Coloured Petri Net Model of a Transport Protocol for Packet Switched Networks,* Proceedings of The Seventh International Conference on Parallel Computing Technologies (LNCS 2763), pp. 66-75, Nizhni Novgorod, Russia, September 15-19, 2003.

[77] Q. Ye and M.H. MacGregor, *Petri Net Approach to Improving SACK TCP Resilience,* Proceedings of Second Annual Conference on Communication Networks and Services Research (CNSR 2004), pp. 146-155, Fredericton, NB, Canada, May 19-21, 2004.

[78] J.C.A.D. Figueiredo and L.M. Kristensen, *Using Colored Petri Nets to Investigate Behavioral and Performance Issues of TCP Protocols,* Proceedings of Second Workshop on Practical Use of Colored Petri Nets and Design/CPN, pp. 21-40, Aarhus, Denmark, October 11-15, 1999.

[79] S. Floyd and V. Jacobson, *Random Early Detection gateways for Congestion Avoidance,* IEEE/ACM Transaction on Networking, Vol. 11, No.4, pp. 397-413, August, 1993.

[80] Bellcore, *GR-499-CORE: Transport Systems Generic Requirements (TSGR): Common Requirements*, December, 1998.

[81] ITU, *ITU-T G.841: Types and Characteristics of SDH Network Protection Architectures*, October, 1998.

[82] N. Brownlee and K. Claffy, *Understanding Internet Traffic Streams: Dragonflies and Tortoises*, IEEE Communication Magazine, Vol. 40, No.10, pp. 110-117, October, 2002.

[83] Q. Ye, M.H. MacGregor and W. Shi, *SACK TCP Resilience Improvement with OPNET,* Proceedings of OPNETWORK 2004, Washington D.C., USA, August 30-September 3, 2004.

[84] J. Kulik et al., *A Simulation Study of Paced TCP, BBN Technical Memorandum No. 1218*, August, 1999.

[85] R. Gupta et al., *WebTP: A Receiver-Driven Web Transport Protocol,* Proceedings of INFORMS 2000 Telecommunications Conference, Boca Raton, Florida, USA, March 5-8, 2000.

[86] A. Aggarwal, S. Savage and T. Anderson, *Understanding the Performance of TCP Pacing,* Proceedings of IEEE INFOCOM 2000: The Conference on Computer Communications, pp. 1157 - 1165, Tel-Aviv, Israel, March 26 - 30, 2000.

[87] I.F. Akyildiz, G. Morabito and S. Palazzo, *TCP-Peach: A New Congestion Control Scheme for Satellite IP Networks*, IEEE/ACM Transaction on Networking, Vol. 9, No.3, pp. 307 - 321, June 2001.

[88] A. Razdan et al., *Enhancing TCP Performance in Networks with Small Buffers,* Proceedings of IEEE ICCCN 2002, pp. 39 - 44, Miami, Florida, USA, October 14-16, 2002.

[89] R.L. Winkler and W.L. Hays, *Statistics, Probability, Inference, and Decision*, Rinehart and Winston Inc., 1971.

[90] Comms/CPN. The CPN Group, Department of Computer Science, University of Aarhus, Denmark, Available from: http://www.daimi.au.dk/designCPN/libs/commscpn/.

[91] An Atlas of Cyber Spaces: Historical Maps, Available from: http://www.cybergeography.org/atlas/historical.html.