WN 97W0000018

WORKING NOTE

User's Manual for the SCPS Transport Protocol

March 1997

Robert C. Durst
Patrick D. Feighery
Eric J. Travis

Sponsor:	SMC/AXE	**Contract No.:**	F-19628-94-C-0001
Dept. No.:	W159	**Project No.:**	03970638

Washington C^3 Center
McLean, Virginia

Abstract

This document provides the information necessary to write programs that use the Space Communications Protocol Standards (SCPS) Transport Protocol (SCPS-TP). The primary content of this document is Unix-style manual pages that describe how the Berkeley Socket Interface has been extended to support SCPS-TP.

This document does NOT provide information necessary to build a distribution of the SCPS-TP software. Such information is provided as documentation with reference implementation of the SCPS protocols or with whatever distribution of the software the user might have.

This document assumes that the reader is familiar with the Berkeley Socket Interface and with the basics of the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP).

KEYWORDS: SCPS, SCPS-TP, TCP, UDP, Socket

Table of Contents

List of Tables

Section 1

Introduction

In the fall of 1992, NASA and the DOD jointly established a technical team (the SCPS Technical Working Group, or “SCPS-TWG”) to explore possibilities for developing common space data communications standards, with a principal focus on the activities associated with in-flight monitoring and control of civil and military spacecraft. In practical terms, these activities involve a ground control center conducting a dialog with a remote spacecraft to transmit telecommands, to up-load and verify onboard software loads, and to confirm correct spacecraft performance via a flow of telemetry.

The team adopted a two-pronged approach in its study phase: part of the team conducted a top-down survey of representative civil and military space data communications requirements, while the remainder of the team conducted a bottom-up analysis of available standard data communications protocols. The team compared the results to see how capabilities matched requirements, and formulated recommendations for future work. In evaluating existing capabilities, first priority was given to commercially-supported “off the shelf” standards. However, recognizing unique requirements of the space mission environment (long propagation delays, noise-induced errors, and limited spacecraft data processing resources and communications capacity), the team also considered other options. By the end of 1993 the team concluded that wide segments of the U.S. civil and military space communities have common needs for:

- An efficient file handling protocol, capable of supporting file transfers initiated either from ground-based systems or space-based systems
- A data transport protocol that provides the user with selectable levels of reliability, based on operational need, between computers that are communicating over a network containing one or more space data transmission paths
- Optional data protection mechanisms to assure the end-to-end security and integrity of such message exchange
- An efficient protocol to support connectionless routing of messages through networks containing space data links.

Following the study phase, the SCPS-TWG began development of four specifications, one for each of the protocols, that address the above requirements: the SCPS File Protocol (SCPS-FP), the SCPS Transport Protocol (SCPS-TP), the SCPS Security Protocol (SCPS-SP), and the SCPS Network Protocol (SCPS-NP). These draft specifications have been submitted for adoption as military standards as international standards. At the completion of these standards activities, resulting standards may then be adopted by any military, civil, or commercial organization for use in any space system. It is the intent of NASA and DOD that commercial vendors produce the SCPS protocols as widely-distributed commercial products, thus helping to reduce the cost of space systems while increasing their interoperability.

Part of the protocol development includes development of a reference implementation of each of the protocols. This reference implementation is being made available for the purposes of evaluation and experimentation with the protocols by potential users of the SCPS capabilities. The SCPS-TP has been implemented so as to present a familiar programming interface to users: the Berkeley Socket interface. This document describes the SCPS extensions to the Berkeley Socket interface that allow users to take advantage of the SCPS-TP capabilities.

This user's guide continues in Section 2 with an overview of the major SCPS-TP capabilities. The material in Section 2 explains and motivates the options available in the SCPS-TP programming interface. Section 3 documents this interface by means of a series of Unix manual pages for SCPS-TP. Finally, Section 4 presents suggested reading for those readers not familiar with TCP, UDP, or the Berkeley Socket Interface.

Section 2

Overview of SCPS Transport Protocol Capabilities

The SCPS Transport Protocol is designed to support current and upcoming tactical and space communication environments. The modifications to the base protocols are intended to address the communication environments and resource constraints that systems fielded into these environments typically face.

As a result of the study phase, several technical requirements were allocated to the transport layer of the Open Systems Interconnect model. These requirements include the following:

- support for communication with full reliability, partial reliability, and minimal reliability;
- efficient operation in a wide range of delay, bandwidth, and error conditions;
- efficient operation in space-based processing environments;
- support for precedence (priority);
- support for connectionless multicasting;
- support for packet-oriented applications.

The SCPS Transport Protocol (SCPS-TP) refers collectively to the protocols that provide the full reliability, best-effort reliability, and minimal reliability services. The full reliability service is provided by the Transmission Control Protocol (TCP). The partial reliability service is provided by TCP with minor modifications. The minimal reliability service is provided by the User Datagram Protocol (UDP).

The SCPS-TP addresses the constraints posed by the operating environment with the following extensions and modifications to TCP:

- Window scaling (specified in RFC 1323)— addresses communication environments that may have more than 65k bytes of data in transit at one time,

- Round Trip Time Measurement (specified in RFC 1323) — This enhancement addresses environments that have high loss, changing delays, or large amounts of data in transit at one time.

- Protect Against Wrapped Sequence Numbers (specified in RFC 1323)— This capability, which uses the Round Trip Time Measurement capability, addresses very long delay or very high bandwidth environments.

- Selective Negative Acknowledgment (SNACK - specified in the SCPS Transport Protocol, adapted from RFC 1106) — SNACK improves performance in high loss environments by providing specific information about missing data and requesting its immediate retransmission.

- Record Boundary Indication (specified in the SCPS Transport Protocol) — This feature provides the ability to mark and reliably carry end-of-record indications for packet-oriented applications.

- Partial Reliability Communication (specified in the SCPS Transport Protocol) — This capability provides the ability for an application to select correct, in-sequence, but possibly incomplete delivery of data.

- Header compression (specified in the SCPS Transport Protocol, adapted from RFC 1144)— This technique reduces the overhead of TCP headers, typically by more than 50%, for application in low-bandwidth environments.

- Low-loss congestion control or optional non-use of congestion control (specified in the SCPS Transport Protocol) — Congestion control mechanisms prevent TCP from overdriving shared links. If links are not shared, users can disable congestion control and depend solely on SCPS-TP's rate control mechanism to prevent overdriving the communication channel. Alternatively, SCPS-TP provides a congestion control mechanism that does not depend on data loss to signal congestion.

- Retransmission strategies for mixed-loss environments that accommodate loss due to data corruption, link outages, and congestion (specified in the SCPS Transport Protocol).

- Acknowledgment strategies for highly asymmetric communication channel capacities (specified in the SCPS Transport Protocol).

This section discusses the problems that might be encountered in the tactical and spacecraft communications environments, then identifies the solutions that SCPS-TP uses to address those problems. Finally, this section discusses how to configure SCPS-TP to enable the specific capabilities that provide the required solutions.

2.1 Differences Between Communications Environments

The Transmission Control Protocol (TCP) provides an excellent base of technology for extension. It is a highly robust protocol, widely distributed, and is freely available. Hundreds of individuals, world-wide, work to ensure that TCP continues to meet the needs of the Internet community. The Internet community currently employs a terrestrial communication environment, and TCP is optimized to provide service to this environment. The space and tactical communication environments may have many similar communication characteristics to the terrestrial environment by virtue of operating across terrestrial networks as part of the end-to-end network. However, there are significant differences between the terrestrial and space/tactical environments that affect communication protocol performance. It should be noted that many of the characteristics of the space and tactical environments are also characteristic of mobile and wireless communication. As a result, many of the SCPS enhancements may be applicable to the mobile and wireless communication community.

Table 1 presents a summary of the main differences that affect TCP performance when it operates in the tactical or space communications environments. The following paragraphs discuss these factors.

Table 1. Factors Affecting TCP Performance in Non-Terrestrial Environments

Factor	Terrestrial Communication	Tactical and/or Space Communication
Bit-Error Rate	Typically < 10^{-9}	10^{-4} to 10^{-12}
Round-trip Delay	Milliseconds to seconds	Seconds to hours
Continuity of connectivity	Continuous	Intermittent
Forward and Reverse Link Data Rates	Symmetric	10:1 to 1000:1 forward to reverse link data rate ratio
CPU and Memory Capacity	Relatively large	Relatively small
Communication Goals	Fair access over time High aggregate throughput High reliability	Maximum throughput during contact period Maximum link utilization Selectable reliability level

Primary Sources of Data Loss	Congestion	Congestion Corruption Link Outage

2.1.1 Bit-Error Rates

The error performance of typical terrestrial networks has improved to a point that it is no longer considered as a typical source of data loss. With sufficient channel coding and application of radiated power, some tactical and satellite links can approach the error performance of terrestrial networks. However, this is not the typical case, especially in situations in which the power, weight, and volume of the communications gear is constrained.

The loss of data due to bit-errors has a disproportionately bad effect on TCP performance because TCP interprets any loss as an indication of network congestion. The appropriate response to network congestion is to reduce the offered load to the network. TCP's congestion response reduces the offered load by half, then builds back slowly over several subsequent round trips. The effect of this in response to bit-errors is to significantly underutilize the communication channel.

2.1.2 Round Trip Delay

Round trip delays in the terrestrial communication environment are typically in the tens of milliseconds to low hundreds of milliseconds. (Round trips across the continental United States average approximately one hundred milliseconds.) In the spacecraft communication environment, round trip times of five hundred milliseconds are the minimum that one expects when communicating through a geostationary satellite, with each hop through a satellite adding another five hundred milliseconds. Deep space communications can increase round trip delays to hours.

Long round-trip delays limit the usefulness and effectiveness of TCP's (or any closed-loop system's) feedback from the remote communication endpoint. This causes problems when the protocol needs to react to changes in the network, but does not receive feedback about those changes until long after the change has occurred.

Note that long delays are not exclusively a result of speed-of-light propagation times. Low data transmission rates add delay to a network, as can half-duplex operation. Finally, queuing in intermediate systems is a source of delay (and the primary source, in the terrestrial communication environment).

2.1.3 Continuity of Connectivity

The terrestrial communication environment can be characterized as a network with a very infrequently-changing topology. Orbiting systems have predictable, but possibly highly dynamic, connectivity characteristics. Low Earth Orbiting satellites typically have connectivity through a single ground station 10% of the time or less. Changes to the number of ground stations or the satellite's orbit can improve this, but even NASA's Tracking and Data Relay Satellite System (TDRSS) offers only about 90% coverage. Further, tactical systems have unpredictable connectivity characteristics, due to system mobility and potential system mortality.

2.1.4 Forward and Reverse Link Capacity

In the terrestrial communication environment, communication links are typically duplex with the same data rate in both directions. This is not the case in space environments. Rather, it is not unusual to have large differences in forward and reverse link capacities. Ratios of 1000:1 are not unusual. This degree of asymmetry causes problems for TCP, which uses a stream of acknowledgments as a self-clocking mechanism for transmitting data packets. Thus, very-low-capacity acknowledgment channels limit the transmission rate of data packets.

2.1.5 CPU and Memory Capacity

In the terrestrial communications environment, the availability of computing resources is essentially unrestricted. This is not the case in spacecraft and tactical systems, in which power, weight, and volume are all precious commodities. The amount of computational resource available to any subsystem in a tactical or spacecraft system must be traded off against the benefits of applying that resource elsewhere. Therefore, it is important to be aware of these constraints. Note that restrictions on power may affect other factors listed here. Notably, power restrictions may increase error rates, decrease data rates (increasing delay), and may affect continuity of connectivity.

2.1.6 Communication Goals

A major TCP goal is to provide to its users fair access to the network over time. By fair access we mean that no single user can monopolize a communication channel when others need to use it. TCP also attempts to provide high aggregate throughput, and provides high reliability.

None of these communication goals are inherently bad. However, the tactical and space communication environments may explicitly NOT wish to provide fair access to the communication resources. Rather, access may need to be on a strict precedence basis, with low precedence users getting starved for resources in favor of high precedence users.

Further, TCP does not assume that maximization of link utilization is a priority. It intentionally under-drives the link at the beginning of a connection and after loss, in an attempt to determine the sustainable capacity of the link.

Finally, TCP offers a fully-reliable service, preserving completeness, sequence, and correctness. TCP trades delay (incurred as a result of retransmission) and buffer space to provide these features. Its companion protocol, the User Datagram Protocol (UDP), provides an unreliable service, with no preservation of sequence or completeness. However, for some types of data, such as image data, a partial-reliability service that preserves sequence and correctness, but possibly not completeness, may be appropriate. In the case of image data, the idea is that the possible loss of a single scan line (or a part of a scan line) should not significantly delay the delivery of the remainder of the image, but that the order of the scan lines is important to preserve.

2.1.7 Primary Source of Data Loss

As previously mentioned, data loss due to bit-errors and to topological instability is rare in the terrestrial environment. The primary source of loss in terrestrial networks is congestion, and TCP is optimized to control congestion. The space and tactical communication environments present mixed-loss environments, with losses occurring due to all three causes: bit-errors, topology changes (link outages), and congestion. To treat all losses in as congestion results in unnecessary reductions in offered load. The increased round trip times in these environments delays the restoration of full-rate transmission.

2.2 Modifications to TCP to Address Communication Problems

To address the problems listed above, the SCPS Transport Protocol includes some already-specified enhancements to TCP and some new enhancements. These enhancements are summarized in Table 2, and described in the following paragraphs.

Table 2. SCPS-TP Modifications to TCP to Address Communication Problems

Factor	Tactical and/or Space Communication	SCPS-TP Modifications
Bit-Error Rate	10^{-4} to 10^{-12}	Corruption response SNACK Header compression
Round-trip Delay	Seconds to hours	Window scaling Timer modifications
Continuity of connectivity	Intermittent	Link outage support
Forward and Reverse Link Data Rates	10:1 to 1000:1 forward to reverse link data rate ratio	Rate control Ack frequency reduction Header compression
CPU Capacity and Memory Availability	Restricted	Header precomputation Record boundaries
Communication Goals	Maximum throughput during contact period Maximum link utilization Selectable reliability	Congestion control optional (rate control to support) Header precomputation Separate corruption response SNACK Partial-reliability operation
Primary Sources of Data Loss	Congestion Corruption Link Outage	Separate response per loss type SCMP signaling Configurable default source of loss

2.2.1 Bit-Error Rates

SCPS-TP has developed three capabilities to address the possibility of data loss due to bit-errors. The first is an explicit response to corruption, rather than congestion, as a cause of loss. The second is the Selective Negative Acknowledgment (SNACK) capability. The third is the loss-tolerant header compression mechanisms.

2.2.1.1 Explicit Corruption Response

When TCP responds to an isolated data loss, it reduces its transmission rate by half and doubles its retransmission timer. SCPS-TP's response to corruption does neither of these things. Rather, both the transmission rate (controlled by the congestion window) and the retransmission time out value remain unchanged.

2.2.1.2 Selective Negative Acknowledgment

The SCPS-TP Selective Negative Acknowledgment (SNACK) capability has been developed to identify specific data that requires retransmission, and to request immediate retransmission of that data. The SNACK capability is invoked when the receiver creates and transmits the SNACK option to the data sender on a regular acknowledgment. The receiver does not send the SNACK option immediately upon detecting a data loss, in case packets have become misordered within the network.

2.2.1.3 SCPS-TP Header Compression

SCPS-TP defines a header compression capability to reduce the size of transmitted packets. This header compression capability operates at the endpoints of the SCPS-TP connection. As a result, headers are only compressed once, regardless of the number of hops that the data requires. Further, this header compression scheme is loss-tolerant, meaning that the loss of one packet does not render subsequent packets unintelligible. (This trait is an unfortunate characteristic of the TCP/IP header compression in common use with dial-up serial lines, and makes it unsuitable for use in the space or tactical communication environment.)

2.2.2 Round Trip Delay

SCPS-TP addresses the problems imposed by round-trip delay with two capabilities - one defined by the Internet community and one defined in SCPS-TP.

2.2.2.1 Window scaling

The Window Scaling option (defined in RFC 1323) permits TCP to have more than 64k bytes of data outstanding (unacknowledged) at one time. (Note that at T1 data rates, 1.54 Mbps, a one-half second round trip delay would result in over 96k bytes of data outstanding.) The window scaling option simply imposes a scaling factor to the advertised window, increasing the maximum data that could be outstanding by powers of two, up to 2^{13}.

2.2.2.2 Timer modifications

SCPS-TP increases the range of typical TCP timers to allow round trip delays of minutes to hours. Further, SCPS-TP initializes its retransmission timer based on data from the routing structure. This allows routes to remote systems to be configured with a reasonable initial estimate of the round-trip time, thus avoiding retransmission time-outs at the beginning of a connection.

2.2.3 Continuity of Connectivity

SCPS-TP depends on signaling from the network layer (the SCPS Network Protocol's SCPS Control Message Protocol, SCMP) to identify link outages. This permits SCPS-TP to differentiate between link outages and other causes of packet loss.

2.2.3.1 Signaling of link outages

The SCPS Control Message Protocol entity depends on information from local link interfaces (for example, a satellite communications channel) to determine whether the link is available or not. Such information can be inferred from, for example, the presence of crypto-synch or from explicit data-link layer signaling. The SCPS Network Protocol entity maintains simple state information about the availability of outbound links. When the link's status changes (for example, from "available" to "unavailable"), SCMP sends a signal indicating the change to recent users of that link. This SCMP signal is received by the SCMP entity at the data source. If, in the case of a link transition from "available" to "unavailable," another route to the destination cannot be identified, the "link out" signal is passed up to SCPS-TP.

2.2.3.2 Link outage support in SCPS-TP

When the SCPS-TP receives a message from its local SCMP entity that a link is out, it ceases to transmit new data. Additionally, it stops its normal retransmission timers and periodically "probes" the link to determine if it has been restored. These probes are either packets with a single byte of data, or they are acknowledgments (if there is no data waiting to be transmitted). The transmission of a probe is not counted as a retransmission of data, so the connection will not be terminated as a result of exceeding the maximum retransmission count. When SCPS-TP receives an indication that the remote entity is again reachable, either through new packets being received from the remote SCPS-TP or from an SCMP message indicating that the link is restored, it resumes its normal mode of operation.

2.2.4 Forward and Reverse Link Capacity

Operation of TCP over highly-asymmetric channels tends to result in sustained under utilization of the high-capacity channel, as mentioned above. This is a result of TCP's use of acknowledgments as clocking mechanisms for transmitting data. SCPS-TP has three capabilities that work together to improve the utilization of the high capacity channel: rate control, acknowledgment frequency reduction, and header compression.

2.2.4.1 Rate control

SCPS-TP provides a rate control mechanism to "spread" the transmission of data across a time interval, replacing TCP's acknowledgment-clocking mechanism. SCPS-TP uses a "token-bucket" rate control mechanism, with the rate control parameters associated with a particular route. All SCPS-TP users on a single host that share that route share the capacity of that route. The rate control also provides a means of limiting the rate of transmission of acknowledgments, something that TCP cannot do.

2.2.4.2 Acknowledgment frequency reduction

TCP attempts to acknowledge at least every-other packet that is received. If TCP detects that a packet is missing, it sends an acknowledgment for every packet. As previously mentioned, limitations on acknowledgment channel capacity result in under utilization of the data channel. SCPS-TP breaks the dependency on acknowledgments as clocking mechanisms, and therefore allows the acknowledgment rate to be reduced.

SCPS-TP permits the user to explicitly specify the rate at which acknowledgments will be sent. If channel capacity permits, this rate should be at least twice per round trip.

2.2.4.3 SCPS-TP Header compression

SCPS-TP header compression, described in Section 2.2.1.3, reduces the size of SCPS-TP headers. By reducing the size of acknowledgments, the load on the (low data rate) acknowledgment is correspondingly reduced. SCPS-TP header compression may be enabled or disabled on a per-route basis.

2.2.5 CPU and Memory Capacity

While current TCP implementations tend to be efficient in their use of CPU and memory resources, SCPS-TP has implemented some further enhancements that take advantage of the environment. These enhancements are header precomputation, the provision of record boundaries, and the implementation of some memory-efficient buffering strategies. Only the record boundary modification is strictly a *protocol* feature (meaning that it has end-to-end significance). The other enhancements are isolated to one endpoint, and do not require the cooperation of the remote system.

2.2.5.1 Header precomputation

The SCPS-TP implementation provides a header precomputation capability to improve CPU use. Its application is in situations where data collection takes place over long periods of time compared to the time when the communication link is available, and it assumes that the data will be transmitted at high rates once the link becomes available. On an existing connection, when the link becomes unavailable, SCPS-TP continues to accept data from the user (to the limits of its available memory), and does all protocol processing possible. When the link becomes available, the timing-related protocol processing is performed and the queued data is transmitted. The effect of the header precomputation is to amortize the bulk of the protocol processing across the time that the link is unavailable, reducing the “spike” in processing required when the link becomes available.

This capability is completely implementation-dependent. There is no protocol mechanism required to support it. Further, in some situations (for example, high-rate data acquisition and low-rate data transmission) it is inappropriate for use. However, in the case of sustained observation and bursty transmission, header precomputation smoothes the CPU utilization over the two periods. The reference implementation of SCPS-TP provides header precomputation as its intrinsic behavior.

2.2.5.2 Record boundaries

TCP provides a byte-stream-oriented transmission capability. That is, it does not guarantee the preservation of record boundaries from end to end. This forces applications to provide their own application-layer framing mechanisms to delimit their data units. SCPS-TP provides a record boundary option that does this application data delimiting function. This results in a memory savings when two or more applications have implemented independent application-layer framing software.

2.2.5.3 Memory buffer strategies

The reference implementation of SCPS-TP provides memory management that is optimized for efficient use of memory. This is the intrinsic behavior, and requires no user action.

2.2.6 Communication Goals

SCPS-TP addresses the communication goals of the space and tactical communication environments with five enhancements to TCP. To address the goals of maximizing throughput and link utilization during a contact period, TCP's congestion control mechanisms are made optional. Header precomputation reduces the protocol processing required at the time that the link is available. And SCPS-TP's corruption response and SNACK capabilities maintain high link-utilization when experiencing bit-errors.

SCPS-TP defines a partial reliability service to address the goal of selectable reliability.

2.2.6.1 Optional congestion control

SCPS-TP makes optional the standard congestion control capabilities within TCP. However, if TCP congestion control is not enabled, system designers must ensure that congestion is either controlled by other means or is not possible in the network due to resource reservation.

2.2.6.2 Header precomputation

Header precomputation is described in section 2.2.1.3. Header precomputation can improve link utilization by reducing the amount of protocol processing required during the time that the link is available. (This benefit accrues if the processor is the performance bottleneck in the system.)

2.2.6.3 Separate corruption response

The SCPS-TP corruption response, described in section 2.2.1.1, improves link utilization by not interpreting data loss due to bit-errors as data loss due to congestion. When responding to corruption, the transmission rate (and therefore link utilization) is not reduced.

2.2.6.4 Selective Negative Acknowledgment

The Selective Negative Acknowledgment (SNACK) capability improves link utilization by providing a means to unambiguously identify and request immediate retransmission of missing data. The SNACK capability is described in section 2.2.1.2.

2.2.6.5 Partial-reliability service

SCPS-TP provides a partial reliability service, called BETS, to ensure correct, in-sequence, but possibly incomplete data delivery. When the BETS capability is enabled, SCPS-TP on the sending side attempts to retransmit packets a user-specified number of times, then continues on as if the packets had been acknowledged (rather than aborting the connection, as standard TCP does). If no retransmissions are desired, the sender discards the packet after its initial transmission. At the receive side, the receiving SCPS-TP entity waits for retransmissions until its receive buffers fill to a user-specified level, then the missing data is signaled to the user. After receiving the signal that a block of data is missing, the receiver can continue reading data beyond that block.

2.2.7 Primary Sources of Data Loss

SCPS-TP addresses mixed-loss environments by providing the ability to respond to different types of loss with responses that are appropriate for that type. The SCPS Control Message Protocol (SCMP) provides signaling mechanisms to inform SCPS-TP about the types of loss being experienced. Finally, the SCPS-TP default response can be configured to invoke either the congestion response or the corruption response.

2.2.7.1 Separate responses for each type of loss

SCPS-TP has separate responses for congestion, corruption, and link outages. The congestion response is the same as that in TCP. The corruption response is described in section 2.2.1.1. The link outage response is described in section 2.2.3.2.

2.2.7.2 SCMP signaling for different loss types

The SCPS Control Message Protocol (SCMP) provides separate signals for congestion (the “source quench” signal), corruption (the “corruption experienced” signal), and link outage (the “link out” and “link redirect” signals). Upon receipt of these signals, SCMP informs SCPS-TP and updates local state information.

2.2.7.3 Configurable default source of loss

If SCMP cannot determine the cause of loss or the signal does not reach SCPS-TP, SCPS-TP must invoke some response to that loss. The reference implementation of SCPS-TP allows each route to be configured with a default source of loss, congestion or corruption. If no signals are received indicating the cause of loss, SCPS-TP will invoke its default response.

2.3 Configuring SCPS-TP to Enable Specific Capabilities

SCPS-TP provides a large degree of flexibility in the enabling and disabling of capabilities, as well as queries and modification of operational parameters. This section describes the mechanisms that allow an application to perform such operations.

2.3.1 Congestion Control

Applications, when using congestion control, have the ability to select the standard TCP congestion control algorithm (“Van Jacobson Congestion Control, - VJCC”) or the newer (and experimental) TCP-Vegas congestion control algorithm. The standard algorithm is robust, but interprets all loss as congestion, and will reduce TCP’s transmission rate accordingly. TCP-Vegas’s algorithm uses changes (increases) in round-trip time to infer congestion, rather than using loss. As a result, TCP-Vegas is better at avoiding congestion, and doesn’t suffer from a misinterpretation of loss as congestion. However, TCP-Vegas *may* be susceptible to misinterpretation of changing round-trip times as congestion.

Note that enabling one congestion control algorithm or another must be done before a connection is established.

An application may enable standard TCP congestion control (VJCC) on a socket in the following manner:

```
one = 1;

scps_setsockopt(socket, SCPSTP_VJ_CONGEST, one, sizeof(one));
```

Similarly, an application may enable TCP-Vegas congestion control on a socket in the following manner:

```
one = 1;

scps_setsockopt(socket, SCPSTP_VEGAS_CONGEST, one, sizeof(one));
```

Applications have the ability to disable congestion control on a per socket basis. This can be achieved by making the following scps_setsockopt() call:

```
null = 0;

scps_setsockopt(socket, SCPSPROTO_TP, SCPSTP_CONGEST, null, sizeof(null));
```

Congestion control should only be disabled when operating in highly managed network environments, in which other traffic control policies exist to prevent congestion collapse of the network.

2.3.2 Selective Negative Acknowledgment

The default behavior for SCPS-TP should be to use SNACK capability on all connections. An application can query whether or not a socket is configured to support the SNACK capability by executing the following getsockopt() call:

```
scps_getsockopt(socket, SCPSPROTO_TP, OPT_SNACK, &result, sizeof(result));
```

The value of result provides a boolean indication as to whether SNACK is enabled on this socket.

Similarly, the setsockopt() call can be used to enable or disable the SNACK capability on a particular socket:

```
scps_setsockopt(socket, SCPSPROTO_TP, OPT_SNACK, &result, sizeof(result));
```

2.3.3 SCPS-TP Header Compression

The default behavior for SCPS-TP should be to use header compression capability on all connections. An application can query whether or not a socket is configured to support the SNACK capability by executing the following getsockopt() call:

```
scps_getsockopt(socket, SCPSPROTO_TP, OPT_COMP, &result, sizeof(result));
```

The value of result provides a boolean indication as to whether header-compression is enabled on this socket.

Similarly, the setsockopt() call can be used to enable or disable the header-compression capability on a particular socket:

```
scps_setsockopt(socket, SCPSPROTO_TP, OPT_COMP, &result, sizeof(result));
```

2.3.4 Rate Control

SCPS-TP provides a capability for imposing a maximum transmission rate on a connection. The rate is configured on per route basis, all connections sharing a route also share the aggregate allocated rate. The rate available on a routing socket can be queried using the scps_getsockopt() call:

```
scps_getsockopt(route_socket, SCPS_ROUTE, SCPS_RATE, &rate, sizeof(rate));
```

returns the current rate allocated in bits per second.

Applications with sufficient runtime authority may change the maximum transmission rate for a routing socket by using the scps_setsockopt() call:

```
rate = 1000000;

scps_setsockopt(route_socket, SCPS_ROUTE, SCPS_RATE, &rate, sizeof(rate);
```

changes the maximum transmission rate to 1Mbps.

2.3.5 Acknowledgment Frequency Reduction

An application may query and change the acknowledgment frequency on a socket basis. To query the current acknowledgment frequency, the application must make use of scps_getsockopt(). The following call will return the current delay between acknowledgments (in milliseconds) in the variable del:

```
scps_getsockopt(socket, SCPSSOCK_TP, SCPSTP_ACKDELAY, &del, sizeof(del));
```

Similarly, the acknowledgment frequency for a given socket can be modified using the scps_setsockopt() call:

```
scps_setsockopt(socket, SCPSSOCK_TP, SCPSTP_ACKDELAY, &del, sizeof(del));
```

2.3.6 Record Boundaries

Record boundaries are enabled by creating a SOCK_SEQPACKET socket instead of the more typical SOCK_STREAM packet.

```
sock = socket(PF_SCPS, SOCK_SEQPACKET, 0);
```

When using a SOCK_SEQPACKET socket, all writes are treated as atomic records by the transport stream. Read requests also respect record boundaries. Read requests that attempt to span record boundaries are truncated at the termination of the currently pending record.

2.3.7 Partial Reliability

Partial reliability transport service (BETS) must be negotiated at the establishment of a transport connection. If both parties in the connection do not offer the BETS capability, partial reliability will not be supported on the resulting connection. Applications must explicitly enable BETS capability on a socket ***prior*** to attempting to initiate a connection (connect() or listen()). BETS is enabled through the scps_setsockopt() call:

```
scps_setsockopt(socket, SCPSSOCK_TP, OPT_BETS, &value, sizeof(value));
```

Following the establishment of a connection, an application can verify whether partial reliability service is indeed available using the scps_getsockopt() call. Depending on the outcome, the application may decide to terminate the connection if partial reliability is not available but required.

Section 3

Manual Pages for SCPS-TP

The following manual pages specify the Application Programming Interface (API) to the reference implementation of the SCPS Transport Protocol (SCPS-TP). The style and content of these manual pages is consistent with the documentation provided with the Unix socket interface.

```
#include <scps.h>

int
scps_socket(int family, int type, int protocol)

     PARAMETERS:
          family: {PF_INET | PF_SCPS}
          type   : {SOCK_STREAM | SOCK_DGRAM | SOCK_SEQPACKET}
          protocol: {UDP | TCP | 0 == default protocol}

     RETURN VALUES:
          positive integer upon success
          -1 on failure and sets scps_errno to indicate the
                    error

     ERRORS
          ENOBUFS  could not allocate a socket for use
```

The SCPS transport API introduces a new protocol family, PF_SCPS.

When a socket is created and the SCPS protocol family is specified, this signals that the socket is to use SCPS-TP for data transport, rather than its Internet protocol equivalent.

Protocol Family	**Socket Type**	**Protocol**	**Resulting Protocol**
PF_INET	SOCK_DGRAM	IPPROTO_UDP	UDP
PF_INET	SOCK_STREAM	IPPROTO_TCP	TCP
PF_INET	SOCK_SEQPACKET		
PF_SCPS	SOCK_DGRAM	IPPROTO_UDP	UDP
PF_SCPS	SOCK_STREAM	IPPROTO_TCP	SCPS-TP
PF_SCPS	SOCK_SEQPACKET	IPPROTO_TCP	SCPS-TP (with Record Boundaries)

```
#include <scps.h>

int
scps_bind(int sockfd, void *myaddr, int addrlen)


     PARAMETERS:
          sockfd:   A valid sockfd
          myaddr:   A pointer to a sockaddr_in address
                    structure
          addrlen:  The size of(struct sockaddr_in)

     RETURN VALUES:
          0         upon success
          -1        on failure and sets scps_errno to indicate
                    the error

     ERRORS:
          EBADF     sockfd is not a valid descriptor
          EINVAL    addrlen is invalid
          ENOTSOCK  socket is wrong type (routing socket)
```

There is no difference in use of the bind() socket call for use with the SCPS protocol suite.

```
#include <scps.h>

int
scps_connect(int sockfd, void *servaddr, int addrlen)

     PARAMETERS:
          sockfd:   A valid sockfd.
          servaddr: A pointer to a sockaddr_in address
                    structure
          addrlen:  The sizeof(struct sockaddr_in)

     RETURN VALUES:
          0         upon success
          -1        on failure and sets scps_errno to indicate
                    the error

     ERRORS:
          EBADF     invalid socket descriptor
          EFAULT    invalid address length
          ENOMEM    Could not get the buffer space required to
                    build a SYN segment
```

There is no difference in use of the connect() socket call for use with the SCPS protocol suite.

```
#include <scps.h>

int
scps_listen (int sockfd, int backlog)

     PARAMETERS:
          sockfd:   A valid sockfd
          backlog:  Maximum number of sockets that can be
                    queued while waiting for an accept.

     RETURN VALUES:
          0         upon success
          -1        on failure and sets scps_errno to indicate
                    the error

     ERRORS:
          EBADF     invalid socket descriptor
```

There is no difference in use of the listen() socket call for use with the SCPS protocol suite.

```
#include <scps.h>

int
scps_accept(int sockfd, void *peer, int *addrlen)

     PARAMETERS:
          sockfd:   The sockfd of a currently listening
                    socket.
          peer:     A pointer to a sockaddr_in address
                    structure to be filled in upon a
                    successful accept. This is the address of
                    the client.
          addrlen:  The size (in bytes) of the peer address
                    provided above.

     RETURN VALUES:
          positive integer representing new sockfd on success
          -1        on failure and sets scps_errno to indicate
                    the error

     ERRORS:

          EBADF     invalid socket descriptor
```

There is no difference in use of the accept() socket call for use with the SCPS protocol suite.

```
#include <scps.h>

int
scps_read(int sockfd, void *data, int size)
scps_recv(int sockfd,  void *data, int size, int flags)
scps_recvfrom (int sockfd, void *data, int size, void *ina,
          int *ina_len)

     PARAMETERS:
          sockfd:   The sockfd of a currently valid socket.
          data:     A pointer to the data that is to be
                    written/read
          size:     The amount of data to be written/read
          flags:    MSG_OOB | MSG_PEEK | MSG_DONTROUTE
          ina:      Pointer to sockaddr_in structure (for
                    remote address).
          ina_len:  Integer values corresponding to length of
                    address contained in *ina

     RETURN VALUES:
          The actual number of bytes read upon success
          -1        on failure and sets scps_errno to indicate
                    the error

     ERRORS:

          EBADF          invalid socket descriptor
          ENOTCONN       socket is not currently connected
          ECONNRESET     connection was reset by peer
          EWOULDBLOCK    operation would block
          EBETS          BETS hole detected at receiver
```

Read/receive operations for applications utilizing the SCPS transport protocol are slightly more complicated than their TCP/UDP counterparts. The behavioral differences stem from the SCPS Record-Boundary and Best Effort Transport Service (BETS).

When reading data from a socket of type SOCK_SEQPACKET, responses to read requests will not cross embedded record-boundaries. Attempts to read lengths greater than that of the next pending record will return only the next record. It is, however, legal to read portions of records, as the record-boundaries are remain persistent in the socket receive buffer until consumed.

The length of bytes read will be the minimum of the requested length and the amount

Record-Boundary operations under SCPS are provided only for sockets of type SOCK_SEQPACKET. When using SOCK_SEQPACKET sockets, all write/send operations are treated as atomic transfers that generate a new record, and therefore, a record-boundary into the transmitted octet-stream.

When operating over a transport connection with BETS enabled, the delivery of a BETS hole in the incoming data stream is signaled via an EBETS error message. Upon receiving such an error, an application should immediately getsockopt() using the SO_BETS_RHOLE_SIZE qualifier in order to determine the size of the missing data. The next read/receive operation will resume the delivery of the sequenced data stream.

```
#include <scps.h>

scps_write(int sockfd, void *data, int size)
scps_send(int sockfd, void *data, int size, int flags)
scps_sendto(int sockfd, void *data, int size, int flags, void
*to, int addrlen)

     PARAMETERS:
          sockfd:   The sockfd of a currently valid socket.
          data:     A pointer to the data that is to be
                    written
          size:     The amount of data to be written
          flags:    MSG_OOB | MSG_PEEK | MSG_DONTROUTE
          to:       Pointer to sockaddr_in structure
                    containing destination address.
          addrlen:  Integer value corresponding to the length
                    of the address in *to

     RETURN VALUES:
          The actual number of bytes written upon success
          -1        on failure and sets scps_errno to indicate
                    the error
     ERRORS:

          EBADF          invalid socket descriptor
          ENOTCONN       socket is not currently connected
          ECONNRESET     connection was reset by peer
          EWOULDBLOCK    operation would block
          EBETS          BETS hole detected at receiver
```

Write/send operations for applications utilizing the SCPS transport protocol are slightly more complicated than their TCP/UDP counterparts. The behavioral differences stem from the SCPS Record-Boundary and Best Effort Transport Service (BETS).

Record-Boundary operations under SCPS are provided only for sockets of type SOCK_SEQPACKET. When using SOCK_SEQPACKET sockets, all write/send operations are treated as atomic transfers that generate a new record, and therefore, a record-boundary into the transmitted octet-stream.

When operating over a transport connection with BETS enabled, unacknowledged holes in the outbound data stream are ***not***

automatically signaled to the sending application.
Applications that need to be aware of BETS holes on the transmitted data stream need to periodically poll for their existence, location and size.
This polling is achieved through the getsockopt() system call. An application can query both the number of previously unreported
BETS holes (SO_BETS_NUM_SEND_HOLES) and also retrieve the position and size of the first 50 unreported holes. If there are more than 50 unreported holes to be retrieved, then multiple getsockopt() calls should be made.

```
#include <scps.h>

int
scps_close(int sockfd)

     PARAMETERS:
          sockfd:       A valid sockfd

     RETURN VALUES:
          0    on success.
          -1   on failure and sets errno to indicate the error

     ERRORS:
          EBADF          invalid socket descriptor
          ESOCKOUTSTATE   TP connection is not in ESTAB state
```

There is no difference in use of the close() socket call for use with the SCPS protocol suite.

```
#include <scps.h>

int
scps_shutdown(int sockid, int how)

    PARAMETERS:
          sockid:   a valid socket
          how:      method for closing connection
```

The shutdown() call causes all or part of a full-duplex connection on the socket associated with s to be shut down.

If how is 0, then further receives will be disallowed. If how is 1, then further sends will be disallowed. If how is 2, then further sends and receives will be disallowed.

```
     RETURN VALUES
          0         on success.
          -1        on failure and sets scps_errno to indicate
                    the error.

     ERRORS
          EBADF     s is not a valid scps socket descriptor.
          ENOTCONN  The specified socket is not connected.
          ESOCKOUTSTATE  TP connection is not in ESTAB state
```

There is no difference in use of the shutdown() socket call for use with the SCPS protocol suite.

```
#include <scps.h>

scps_select(int sockid, scps_fd_set *readset,
            scps_fd_set *writeset,
            scps_fd_set *nullval, struct timeval *time)

     PARAMETERS:
          sockid    a valid scps socket identifier
          readset   the file descriptors to be tested for
                    reading
          writeset  the file descriptors to be tested for
                    writing
          nullval   a null value for compatibility with select
          time      the amount of time to wait before
                    returning if no descriptors are readable
                    or writable

     RETURN VALUES:
          select() returns a non-negative value on success. A
                    positive value indicates the number of
                    ready descriptors in the descriptor sets.

          0 indicates that the time limit referred to by time
                    expired.

          On failure, select() returns -1, sets scps_errno to
                    indicate the error, and the descriptor
                    sets are not changed.

     ERRORS:
          EINVAL    accept called with no socket descriptors
                    identified.
```

When selecting on a socket using SCPS-TP with BETS enabled, the creation of a BETS hole will cause select() to return with an EBETS error message. Since the creation of a BETS hole also indicates that there is new data available for consumption by the application, there is no need to call select() again prior to attempting to perform a read()/recv() operation.

```
#include <scps.h>

int
scps_getsockopt(int sockid, int level, int optname, void
*optval, int *optlen)
scps_setsockopt(int sockid, int level, int optname, void
*optval, int optlen)

PARAMETERS:
          sockfd:   The sockfd of a currently valid socket.
          level:    SOL_SOCKET
                    SCPSPPROTO_TP
                    SCPS_ROUTE
          optname:  (dependent on level)

     RETURN VALUES:
          0         on success
          -1        on failure and sets scps_errno to indicate
                    the error.

     ERRORS:

          EBADF               invalid socket descriptor
          EFAULT              address length is invalid
          ENOPROTOOPT         The option is unknown at the
                              level indicated.
          EPROTONOSUPPORT     Attempt made to perform SCPSTP
                              operations on another type of
                              socket.
          EOPNOTSUPP          Operation not supported
          ESOCKOUTSTATE       Connection in incorrect state
                              for this operation
          ENOBETS             BETS not supported on this
                              connection
          ENOMEM              Unable to build a list of
                              current BETS holes.
```

SCPS introduces a number of new options that are not present in the standard Berkeley socket interface. These are grouped according to operations that are:

- socket specific (level = SOL_SOCKET)
- transport connection specific (level = SCPSPROTO_TP)
- route specific (level = SCPS_ROUTE)

Socket specific option handling (level = SOL_SOCKET):

SCPS enhancements to the SOL_SOCKET level socket option processing are for the querying of BETS holes in the transmission stream, both on the sending and receiving sides of a connection.

SO_BETS_RHOLE_SIZE
Retrieves the size of a signaled BETS hole at the receiver

SO_BETS_RHOLE_START
Retrieves the relative sequence number of the start of the
signaled BETS hole

SO_BETS_NUM_SEND_HOLES
Retrieves the number of BETS holes currently detected at the sender (sender has experienced one or more BETS timeouts on its data stream)

SO_BETS_SEND_HOLES
Retrieves the first 50 BETS holes currently detected at the sender, and then resets the counter SEND_HOLE counter to zero.

Connection specific socket option handling (LEVEL = SCPSPROTO_TP):

SCPS-TP provides the application using the transport protocol much more flexibility in the configuration of connection specific parameters. These values should be changed from their system-wide default values with great care and planning.

```
SCPSTP_MAXSEG
     Sets/reads the maximum segment size (bytes) for use for
     this socket.

SCPSTP_ACKDELAY
     Sets/reads the value in (milliseconds) of the delayed
     ACK timer to be used for TP operation for this socket.

SCPSTP_ACKFLOOR
     Sets/reads the value (in milliseconds) for the minimum
     delay between ACK transmissions for TP operation for
     this socket. (Don't ACK more often than this).

SCPSTP_RTOMIN
     Sets/reads the minimum retransmission time (in
milliseconds)
     for TP operation for this socket.

SCPSTP_RETRANSMITTIME
     Sets/reads the retransmission timer call interval(in
     milliseconds) for TP operation for this socket.

SCPSTP_PERSISTTIME
     Sets/reads the delay (in milliseconds) until a zero-
     window probe is sent.

SCPSTP_TIMEOUT
     Sets/reads the maximum number of retransmissions allowed
     (integer value) before transmitter aborts the connection
     or declares a BETS_SEND_HOLE and moves on.

SCPSTP_LONGTIMEOUT
     Sets/reads the maximum number of retransmissions allowed
     (integer value) for a SYN before aborting a connection

SCPSTP_2MSLTIMEOUT
     Sets/reads the 2MSLtimeout value (in milliseconds) for a
     TP socket. In general, this value should not be changed
     from the system default.

SCPSTP_BETS_RTIMEOUT
     Sets/reads the BETS_Receive Timeout value (in
milliseconds)
```

The following options must be changed prior to connection
establishment (either connect() or listen()) in order to have
any
effect.

SCPSTP_TIMESTAMP
     Enables or disables use of TIMESTAMP on a TP connection.

SCPSTP_COMPRESS
     Enables or disables use of header compression on a TP
     connection.

SCPSTP_SNACK
     Enables or disables use of Selective Negative Acks on a
     TP connection.

SCPSTP_BETS
     Enables or disables use of BETS on a TP connection.

SCPSTP_CONGEST
     Enables or disables use of congestion control on a TP
     connection. Disabling of congestion control on a specific
     transport connection should only be done in managed
     network environments.

SCPSTP_VJ_CONGEST
     Enables or disables the use of the TCP standard
     congestion control algorithm (Van Jacobson Congestion
     Control) on a TP connection. SCPSTP_CONGEST must be
     enabled for this to have an effect.

SCPSTP_VEGAS_CONGEST
     Enables or disables the use of TCP-Vegas congestion
     control on a TP connection. SCPSTP_CONGEST must be
     enabled for this to have an effect.

Route specific options (LEVEL = SCPS_ROUTE):

SCPS allows management agents or applications running with sufficient privilege to change the certain default values on a route specific basis:

SCPS_RATE
: Sets/reads the rate control setting (in bps) for the

  global routing socket.

SCPS_MTU
: Sets/reads the maximum MTU size (in bytes) for the global routing socket.

SCPS_RTT
: Sets/reads the *initial* RTT estimate used by TP connections.

Section 4

Suggested Reading

This section presents some suggested reading for readers with varying levels of familiarity with TCP, UDP, and the Socket Interface.

Readers with little previous familiarity with TCP or UDP should consider reviewing an introductory text on the subject. One excellent example is TCP/IP Illustrated, Volume 1, by W. Richard Stevens (Copyright 1994, Addison-Wesley Professional Computing Series). Chapters 1, 11, and 17-24 are particularly relevant.

The Berkeley Socket interface is described in detail, along with many various programming examples in Unix Network Programming, by W. Richard Stevens (Copyright 1990, Prentice Hall Software Series). Sections 1 through 6 are very appropriate.

The internals of the TCP and UDP code are explained in detail in TCP/IP Illustrated, Volume 2, The Implementation, by Gary R. Wright and W. Richard Stevens (Copyright 1995, Addison-Wesley Professional Computing Series).

The following Requests for Comments provide the specifications for the Internet Protocols on which SCPS-TP is based. Universal Resource Locators (URLs) are provided for world-wide web access.

J. Postel. *User Datagram Protocol*. IAB STD 6. RFC 768, August 28, 1980. <URL: http://ds.internic.net/rfc/rfc768.txt>.

J. Postel. *Transmission Control Protocol*. IAB STD 7. RFC 793, September 1, 1981. <URL: http://ds.internic.net/rfc/rfc793.txt>.

D. Borman, R. Braden, and V. Jacobson. *TCP Extensions for High Performance*. RFC 1323, May 13, 1992. <URL: http://ds.internic.net/rfc/rfc1323.txt>.

J. Nagle. *Congestion Control in IP/TCP Internetworks*. RFC 896, January 6, 1984. <URL: http://ds.internic.net/rfc/rfc896.txt>

K. McCloghrie and M. Rose. *Management Information Base for Network Management of TCP/IP-Based Internets: MIB-II*. AIB STD 17. RFC 1213, March 26, 1991. <URL: http://ds.internic.net/rfc/rfc1213.txt>

Several articles have been published that explain particular aspects of TCP operation. The following are of particular interest:

P. Karn & C. Partridge. “Round Trip Time Estimation.” In Proceedings of SIGCOMM ‘87: Symposium on Communications Architectures and Protocols, August 1987.

V. Jacobson. “Congestion Avoidance and Control.” In Proceedings of SIGCOMM ‘88: Symposium on Communications Architectures and Protocols, August 1988.

S. Brakmo, S. W. O’Malley, and L. L. Peterson. “TCP Vegas: New Techniques for Congestion Detection and Avoidance.” In Proceedings of SIGCOMM ‘94: Symposium on Communications Architectures and Protocols, August 1994.

Appendix

Example Code

The following is an example of an application developed using the SCPS software reference implementation. This code must be compiled and linked with the SCPS library “libscps.a”. This particular example is the source of a typical source/sink networking application.

The programs main () is minimalistic and contains the following code necessary for any application to use the SCPS protocol stack.

```
 main(argv, argv)
 {

   /* Initialize the thread scheduler data structures */
   init_scheduler();

   /* create a thread for the SCPS protocols */
   scheduler.run_queue[0] = create_thread(tp);

   /* create a thread for the source application. */
   scheduler.run_queue[1] = create_thread(source_application);

   /* Initialize the SCPS protocols' data structures */
   (void) scps_Init();

   /* Pass control to the thread scheduler */
   start_threads();

   /* Exit when done */
   exit(0);
  }
```

Writing programs using SCPS TP sockets is not very different than writing programs that make use of TCP sockets. The SCPS socket calls are prefaced with “scps_”, but otherwise their use is basically identical to their TCP counterparts. For example the SCPS equivalent to the TCP socket call “accept ()” is “scps_accept ()”.

To set the various SCPS options the scps_setsockopt () function is used. The corresponding scps_getsockopt () function is used to get the various SCPS options. In this example the following constants are used.

```
#define SCPS_ACKDELAY           20              /* Sets the acknowledgment timer
                                                   for SCPS TP to 20 milliseconds */
#define SCPS_SOCKBUFSIZE        65536           /* Sets the maximum send and receive
                                                   buffer to 65536 bytes */
#define SCPS_COMPRESS           1               /* Enable SCPS Compression */
#define SCPS_RATECONTROL        1000000         /* Sets SCPS TP rate control to
                                                    1 million packets per second */
```

To set a particular option remove the comment block from the corresponding #define statement in the followingsource code and recompile. The corresponding scps_setsockopt () function will be called. For example if the socket buffer size is to be set then remove the comment block from the following statement

```
#define SCPS_SOCKBUFSIZE        65536
```

The will result in the following block of code to be executed.

```
#ifdef SCPS_SOCKBUFSIZE
{
        int sockbufsize = SCPS_SOCKBUFSIZE;
        scps_setsockopt (tp_sock, SCPS_SOCKET, SO_RCVBUF, &sockbufsize,
                               sizeof(sockbufsize));
        scps_setsockopt (tp_sock, SCPS_SOCKET, SO_SNDBUF, &sockbufsize,
                               sizeof(sockbufsize));
}
#endif SCPS_SOCKBUFSIZE
```

```
/*******************************************************
*
* This software was developed by The MITRE Corporation
* and was produced for the US Government under Contracts
* DAAB0796-C-E601, F19628-94-C-0001, and NAS5-32607 and
* is subject to Department of Defense Federal
* Acquisition Regulation Clause 252.227.7013, Alt. 2,
* Clause 252.227.7013 and Federal Acquisition Regulation
* Clause 52.227-14, Rights in Data - General
*
* NOTICE
*
*
* MITRE PROVIDES THIS SOFTWARE "AS IS" AND MAKES NO
* WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY,
* CAPABILITY, EFFICIENCY, OR FUNCTIONING OF THE PRODUCT.
* IN NO EVENT WILL MITRE BE LIABLE FOR ANY GENERAL,
* CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR
* SPECIAL DAMAGES, EVEN IF MITRE HAS BEEN ADVISED OF THE
* POSSIBILITY OF SUCH DAMAGES.
*
* You accept this software on the condition that you
* indemnify and hold harmless MITRE, its Board of
* Trustees, officers, agents and employees, from any and
* all liability or damages to third parties, including
* attorneys' fees, court costs, and other related costs
* and expenses, arising our of your use of the Product
* irrespective of the cause of said liability, except
* for liability arising from claims of US patent
* infringements.
*
* The export from the United States or the subsequent
* reexport of this software is subject to compliance
* with United States export control and munitions
* control restrictions.  You agree that in the event you
* seek to export this software you assume full
* responsibility for obtaining all necessary export
* licenses and approvals and for assuring compliance
* with applicable reexport restrictions.
*
*******************************************************/


/*  SCPSTP initiator
```

```
 *
 *  Opens a SCPSTP socket and source data
 *
 */

#include <stdio.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include "scps.h"

void scps_init(void);
void initiator_application(void);


/*  To turn on any of these SCPS options remove the comment
 *  blocks from that particular SCPS #define statement.
 */

/* #define SCPS_ACKDELAY      20          */ /* Sets the acknowledgment
timer
                                                  for SCPS TP to 20
milliseconds */
/* #define SCPS_SOCKBUFSIZE   65536       */ /* Sets the maximum send and
                                                     receive buffer to 65536
bytes */
/* #define SCPS_COMPRESS      1           */ /* Enables SCPS Compression */
/* #define SCPS_RATECONTROL   1000000     */ /* Sets SCPS TP rate control
to
                                                     1 million packets per
second */


extern int errno;
extern int optind;
extern char *optarg;

int    config_tp_pkt_count = 10;
int    config_tp_pkt_size = 1000;
unsigned short  config_tp_init_port = 0x8000;
unsigned short  config_tp_resp_port = 0x8fff;

unsigned int mtu_val;
```

```
struct sockaddr_in hisaddress;
int tp_sock;

#define MAX_BUF_SIZE 65536
byte outbuf[MAX_BUF_SIZE];
byte inbuf[MAX_BUF_SIZE];

unsigned long buffsize = 32768; /* Default buffer size for initiator */
int optlen;
char *peer_host;

int one = 1;
struct timeval select_time;

static int total_in_data = 0;
static tp_total_bytes;
int sched_counter;
int options = 0;
longword host;
int select_val = 0;



void
initiator_application()
{
  int cc, bytes_to_send, tp_prev_bytes;
  int push = 0;
  int errno;
  static int tp_write_cnt = 0;

  scps_init();

#ifdef SCPS_RATECONTROL
{
  int rate_control = SCPS_RATECONTROL;
  scps_setsockopt(route_sock, SCPS_ROUTE, SCPS_RATE,
    &rate_control, sizeof(rate_control));
}
#endif SCPS_RATECONTROL

  if ((tp_sock = scps_socket(AF_INET, SOCK_STREAM, 0)) < 0)
      {
        printf("Error!! Failed to create a tp_socket\n");
        exit(-1);
      }

   /* Bind to a local port */
  hisaddress.sin_port = htons(config_tp_init_port);
  bzero(&(hisaddress.sin_addr), sizeof(u_long));

  scps_bind(tp_sock, (struct sockaddr *)&hisaddress, sizeof(hisaddress));

#ifdef SCPS_SOCKBUFSIZE
{
  int sockbufsize = SCPS_SOCKBUFSIZE;

  scps_setsockopt(tp_sock, SCPS_SOCKET, SO_RCVBUF, &sockbufsize,
    sizeof(sockbufsize));
  scps_setsockopt(tp_sock, SCPS_SOCKET, SO_SNDBUF, &sockbufsize,
    sizeof(sockbufsize));
}
#endif SCPS_SOCKBUFSIZE


#ifdef SCPS_COMPRESS
    scps_setsockopt(tp_sock, PROTO_SCPSTP, SCPSTP_COMPRESS,
      &one, sizeof(one));
#endif SCPS_COMPRESS

#ifdef ACKDELAY
{
  int ackdelay = SCPS_ACK_DELAY;
  scps_setsockopt(tp_sock, PROTO_SCPSTP, SCPSTP_ACKDELAY,
    &ackdelay, sizeof(ackdelay));
  scps_setsockopt(tp_sock, PROTO_SCPSTP, SCPSTP_ACKFLOOR,
    &ackdelay, sizeof(ackdelay));
}
#endif ACKDELAY

  scps_getsockopt(tp_sock, SCPS_SOCKET, SO_RCVBUF, &buffsize, &optlen);
  printf("TP Receive Buffer size is now %u\n", buffsize);
  scps_getsockopt(tp_sock, SCPS_SOCKET, SO_SNDBUF, &buffsize, &optlen);
  printf("TP Send Buffer size is now %u\n", buffsize);
  printf("TP sending %d bytes, with %d byte writes (%d writes)\n",
    tp_total_bytes, config_tp_pkt_size, config_tp_pkt_count);
  scps_getsockopt(route_sock, SCPS_ROUTE, SCPS_RATE, &buffsize, &optlen);
  printf("TP Rate control set to %u bps\n", buffsize);
```

```
  scps_getsockopt(route_sock, SCPS_ROUTE, SCPS_MTU, &mtu_val, &optlen);
  printf("TP MTU set to %u bytes\n", mtu_val);

  /* Open a connection with a distant host/port */
  hisaddress.sin_port = htons(config_tp_resp_port);
  bcopy(&host, &(hisaddress.sin_addr), sizeof(u_long));

  scps_connect(tp_sock, (struct sockaddr *)&hisaddress,
sizeof(hisaddress));

  tp_prev_bytes = tp_total_bytes;

  while (tp_total_bytes > 0)
    {  /* data to send? */
       /*  Meter TP data out one packet at a time */
       bytes_to_send = ((tp_total_bytes > config_tp_pkt_size) ?
                    config_tp_pkt_size : tp_total_bytes);

       cc = scps_write(tp_sock, outbuf, bytes_to_send);

       if (cc > 0)
       {
            tp_write_cnt++;
            tp_total_bytes -= cc;
       }

    }

  /* No more data to send - close the connection */
  scps_close(tp_sock);
  threadExit();
}



void
scps_init()
{
  int i;

  tp_total_bytes = config_tp_pkt_count * config_tp_pkt_size;
  host = (longword) get_remote_internet_addr(peer_host);
  tp_total_bytes = config_tp_pkt_count * config_tp_pkt_size;
```

```
}



void
main(argc, argv)
  int argc;
char **argv;
{

  if (argc != 2)
    goto usage;

  peer_host = argv[optind];
  init_scheduler();
  scheduler.run_queue[0] = create_thread(tp);
  scheduler.run_queue[1] = create_thread(initiator_application);
  (void) scps_Init();
  start_threads();
  exit(0);

usage:
  fprintf(stderr,"Usage:  %s host.\n", argv [0]);
  exit(1);
}
```

Glossary

Ack	Acknowledgment
API	Application Programming Interface
BER	Bit-error rate
BETS	Best Effort Transport Service
bps	Bits per second
CPU	Central processing unit
IAB	Internet Activities Board
Mbps	Million bits per second
RFC	Request for Comments
SCMP	SCPS Control Message Protocol
SCPS	Space Communications Protocol Standards
SCPS-FP	SCPS File Protocol
SCPS-NP	SCPS Network Protocol
SCPS-SP	SCPS Security Protocol
SCPS-TP	SCPS Transport Protocol
SCPS-TWG	SCPS Technical Working Group
SIGCOMM	Special Interest Group on Communications
SNACK	Selective Negative Acknowledgment
STD	Standard
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URL	Universal Resource Locator
VJCC	Van Jacobson Congestion Control