# Scheduled Transfer Protocol on Linux

Pekka Pietikäinen
Pekka.Pietikainen@cern.ch

**Abstract**

Scheduled Transfer Protocol (STP[1]) is a new ANSI specified connection-oriented data transfer protocol. In STP small control messages are used to allocate buffers on the remote host before any data transfer. This reduces the workload of the receiver considerably and makes hardware acceleration relatively simple to implement. Applications for the protocol include network attached storage (SCSI running over STP). The low-latency aspects of STP also make it a very attractive protocol for for clustering.

In this paper, we describe problems seen in gigabit networking today and how how STP overcomes these problems. We also describe the Linux implementation of STP, which supports zero-copy transmits using the Linux 2.4 zero-copy framework and receives using a modified Alteon Tigon-II firmware. Finally, we compare the performance of STP and TCP.

## 1 Introduction

In the last few years, gigabit networking has become possible using commodity hardware, standard PCs and Gigabit Ethernet (GigE). However, utilizing the full network bandwidth has proved to be a complicated problem.

The main two problems seen in gigabit networking seen today are unnecessary memory copies and interrupts.

Excessive memory copies are a significant problem as network speeds approach the speed of main memory. In current PCs, such is the case as the memory speeds are in the range of several hundred MB/s. The theoretical maximum speed of GigE is 125MB/s

Most modern TCP implementations perform one copy from user-space to kernel buffers, and a DMA from the kernel to the NIC. With sufficiently advanced hardware that supports scatter-gather DMA and hardware checksums, it is possible to avoid the copy entirely and significantly lower CPU utilization.

On the receiver, things are more complicated. Since the packets need to be demultiplexed to the correct applications, avoiding the copy is nearly impossible. Performing the demultiplex operation in hardware is possible, although the large amount of state information required be TCP would require performing the entire protocol in hardware, which is generally considered a bad idea.

If the packet payload is a multiple of the page size i.e. 4k or 8k, it is also possible to place the data into a separate page and use the MMU to remap the page into the correct place in the virtual address space of the application. The page remapping operations have a cost, however, which does not necessarily show in the microbenchmarks which are used to measure network performance.

Interrupts are typically generated for each packed received or transmitted. For low-speed networks such as 10Mbps Ethernet this is not a significant problem, since the amount of interrupts is still only a few thousand per second even with small packets. On Gigabit Ethernet using the standard 1500 byte packets an interrupt per packet would cause nearly 90000 interrupts per second. With smaller packets the problem is even worse.

The reason interrupts cause a problem is that hardware interrupts usually have priority over everything else running on the same system. When an interrupt is raised, the machine has to save its current state, run the interrupt handler, and then continue with previous task.

[1]The abbreviation ST is also commonly used e.g. in the ANSI standard, but the usage is discouraged since the same abbreviation is also used by IETF to refer to the Internet Stream Protocol defined in RFC 1819.

If the interrupt rate is high enough, the OS may become live-locked by being so busy processing the interrupts coming from the NIC that it has very little time to handle anything else.

There are some ways of minimizing the amount of interrupts generated, which is usually referred to as interrupt mitigation or interrupt coalescing. One approach is having the driver poll the card for several new packets after each interrupt. Another possibility is is making the adapter wait for a specified amount of time for more packets to arrive before interrupting the host CPU. The interrupt service routine must then handle all the packets in the queue every time it is called. The down-side of both approaches is that it increases latency for all applications, even ones that are latency-critical.

Another way of reducing the amount of interrupts is reducing the amount of packets by increasing their sizes. Unfortunately the Ethernet standard limits the maximum size of a frame to 1500 bytes, which was adequate for the hardware and networks used 20 years ago, but is insufficient for modern networks. An extension called jumbo frames allows the use of frames of up to 9k, which is supported by most new NICs, but only a few switches. An added benefit of jumbo frames is that they are large enough to accommodate entire memory pages, which can used to optimize implementations.

Scheduled Transfer Protocol is a new ANSI specified connection-oriented data transfer protocol which was designed to make it possible for protocol processing to be placed partially placed into intelligent hardware, making it possible to accomplish true zero-copy transmits and receives with an extremely low interrupt rate.

## 2 Scheduled Transfer Protocol

STP was originally designed to be the network protocol to be used with the GSN (Gigabyte System Network), a 6.4Gbps successor to the aging HIPPI network. As the standardization effort for GSN continued, it was noticed that STP would also be useful on other networks such as Gigabit Ethernet, and thus STP was separated into a separate standard.

The basic design principle of STP is that as much work as possible should be performed by the transmitter, and the receiver only needs to verify the incoming packet and place the data into the correct buffer. Before any data transfer happens, small control messages are transmitted to pre-allocate buffers at the receiver before the data movement begins. The data can then be directly moved from the network into host memory.

The main platform for STP is IRIX, where STP has been demonstrated to run at speeds of 790MB/s with a latency of 10 $\mu$s over a single GSN link between two Origin 3000 servers. SGI has also released an implementation based on the IRIX code for Linux. 3rd party versions are also available for Solaris and Tru64 UNIX.

In STP, data is transmitted in Transfers, which have a predetermined length. As shown in figure 1, Transfers are divided into Blocks, which are the level flow-control is performed on. Blocks are further divided into STUs (Scheduled Transfer Unit), which correspond to physical packets on the wire.

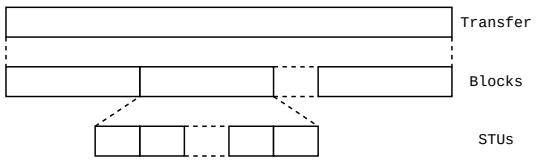


Figure 1: STP data hierarchy.

When a STP connection is established, the endpoints negotiate the maximum STU data they can accept as well as the buffer size they are using. Both ends also assign the connection an unique identifier which the other end must use for the duration of the connection.

When a host wishes to send data, it issues a Request_To_Send, which contains the size of the Transfer, the maximum Block size it can send as well as an unique identifier for the Transfer. The receiver reserves buffers for a Block and issues a Clear_To_Send for the Block. The sender then sends the data as STUs, which contain the buffer id and offset inside the buffer.

Receiving the data itself can then be implemented in hardware efficiently, as the state information for the protocol is very low. This makes zero-copy receives possible. Also it is not necessary to interrupt the host until the entire block has been received. With a Block size of 64 STUs, the interrupt rate thus drops by 98.4% (63/64).

STP is capable of striping a single Transfer across multiple network interfaces. To do so, the receiver only needs to issue the CTSs simultaneously on different physical network interfaces. Each Block will then be sent to the interface from which the CTS arrived.

For low-latency applications, the overhead involved in reserving a buffer on the remote for every Transfer is unacceptable. For these applications, STP also allows hosts to request a persistent memory region on which Remote DMA (RDMA) operations can be performed.

### 2.1 Using STP

STP supports several Upper Layer Protocols (ULP), including standard BSD sockets, SCSI and libst. The easiest way to use STP in applications is with the standard BSD socket API. A STP socket is created with

```
socket(PF_INET,SOCK_SEQPACKET,IPPROTO_STP)
```

Every write becomes a separate STP Transfer, with its own associated overhead. It is therefore necessary to use large writes (128-512k on GigE) to get good performance.

Due to the nature of STP, the receiver is required to read the entire Transfer with each read system call. If an application attempts to write more data than the receiver is expecting, the receiver will refuse the Transfer and the write() will return a "Message too long" error.

While sockets are usable for bulk data transfer, they are unable to expose the full functionality of STP, such as persistent memory operations. For this purpose, a low-level API for STP called libst has been designed, which supports OS bypass by having direct secure access to the NIC.

The SCSI over ST (SST) standard defines a method of encapsulating SCSI packets inside STP, providing a possibility for high-performance network-attached storage. STP to SCSI and Fibre Channel bridges are available commercially from some vendors.

There has also been some interest in using STP as the wire-level protocol for Intel's Virtual Interface Architecture (VIA), but no implementations exist currently.

### 2.2 STP on Linux

In May 2000, SGI released a Linux implementation of STP for Linux under GPL, which was largely based on the original IRIX code.

The basic design of the Linux implementation of STP is shown in figure 2. The STP core component is responsible for buffer management, and the various finite-state-machines required for operations such as connection set-up and tear-down. Currently, the only Upper Layer Protocol implemented is INET sockets.

To provide possibility for hardware acceleration of the protocol, the STP core advertises a stp_netdevice interface to the NICs and their device drivers to support STP. The stvd module provides a default sofware-based implementation for most of these enhancements; this enables running STP using standard, unmodified drivers.

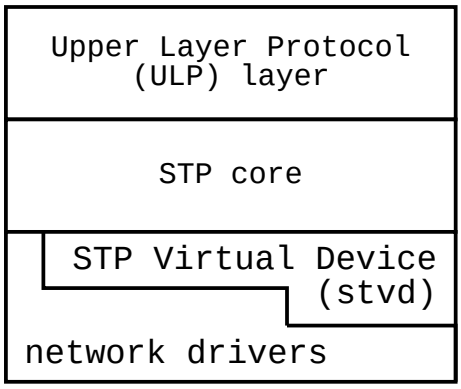


Figure 2: Structure of the Linux STP stack.

The Linux implementation includes a modified driver and firmware for the Alteon Tigon-II chip, which supports hardware acceleration of the protocol.

The driver modifications consists of about 500 lines of code, which adds support for updating the list of current buffers and connections in the NIC firmware. The firmware modifications are about 1000 lines of code, mostly in the receive path to detect incoming STP data packets, verify the headers and copy data into its final location.

Some prelimenary code for mapping the RX/TX rings into userspace for use with libstp exist, but isn't fully functional.

### 2.3 Benchmarks

In this section, we compare the performance of STP with the Linux 2.4 zero-copy TCP.

The measurements were done between a dual 500MHz Pentium III transmitting to a dual 450MHz Pentium II with. The NIC used in both machines are 1MB DEC Gigabit Ethernet Adapters using the Alteon Tigon-II chipset. The machines were running Linux-2.4.2-pre1 with the zero-copy patch and STP 0.33.

The measurements were done with a small custom network performance tester called yantt, which was designed to be modular and easy to adapt for different kinds of tests. Yantt measures CPU usage by comparing the CPU available for a low-priority process running on the same system to the situation on an unloaded system. A result of 100CPUs are fully utilized. This approach is necessary as the standard getrusage() system call does not account for time spent in interrupt handlers, which accounts for a large amount of the total CPU use. If getrusage() is used, the CPU utilization is shown as only 4% CPU on both the sender and receiver for STP.

The MSG_TRUNC results were obtained the copy from kernel to user buffers is omitted on the receiver. The results[2] are shown in table 1.

As can be seen from the results, on these machines, TCP performance is limited by the receiver CPU.

When the receiver is not limiting performance (STP and MSG_TRUNC TCP), the performance is still far from the theoretical 125MB/s maximum. The limiting factor appears to be the 32-bit PCI bus used in these experiments.

Even when compared to TCP with MSG_TRUNC the CPU load is significantly lower, which can be attributed to the hardware acceleration mitigating nearly 99% of the interrupts.

[2]In this paper a megabyte is $2^{20}$ bytes. The 100MB/s barrier could have been easily broken by playing around with numbers, but the author chose not to.

| test | packet size | bw (MB/s) | CPU rcv | CPU snd |
|---|---|---|---|---|
| TCP write() | 9000 | 75 | 55 | 48 |
| MSG_TRUNC | 9000 | 99 | 35 | 55 |
| TCP sendfile() | 9000 | 74 | 55 | 12 |
| MSG_TRUNC | 9000 | 99 | 31 | 19 |
| STP write() | 4136 | 98 | 8 | 17 |
| TCP write() | 1500 | 52 | 55 | 50 |
| MSG_TRUNC | 1500 | 70 | 52 | 70 |
| TCP sendfile() | 1500 | 45 | 50 | 12 |
| MSG_TRUNC | 1500 | 70 | 52 | 39 |
| STP write() | 1064 | 53 | 8 | 17 |

Table 1: TCP and STP performance

With standard frames, the limitations of hardware are much more evident. The Tigon-II DMA hardware has a limit on the rate of DMAs it can handle, which is about 50000 packets/s. This especially hurts STP, because it requires the payload to be of a $2^n$ size.

It should be noted, that the measurements done above fail to account for the beneficial side effect of having the data in the cache as it is copied. To measure the effect more accurately, the benchmark software was modified to touch the data it received. The results are summarized in 2.

| test | packet size | bw (MB/s) | CPU rcv | CPU snd |
|---|---|---|---|---|
| TCP | 9000 | 49 | 62 | 22 |
| STP | 4136 | 53 | 23 | 9 |
| TCP | 1500 | 14 | 53 | 11 |
| STP | 1064 | 34 | 16 | 11 |

Table 2: Results when receiver touches the data

As can be seen from the results, performance drops significantly for both protocols. With TCP, however, the performance drop is smaller due to the data already being cached. Still, STP leaves plenty of CPU to be used for actual processing of the data. With standard frames, the TCP performance drops to nearly 100baseT speeds due to the interrupt load.

## 3 Future Challenges

There is still a lot of work to be done on STP on Linux. Although the basic implementation is com-

plete, the OS Bypass and SCSI encapsulation have not been explored.

The greatest challenge remaining is finding a suitable replacement for the aging Tigon-II, as currently the trend seems to be towards cheap non-programmable NE2000-style designs for GigE. Fortunately, some new GigE designs include the possibility for user-programmable firmware.

## 4 Acknowledgments

The author would like to thank CERN for funding his Master's thesis, which this paper is largely based on, as well as SGI for providing the initial GPL implementation of STP for Linux.

## References

[1] ANSI NCITS 337-2000, Information Technology - Scheduled Transfer (ST) (2000). American National Standards Institute, Inc., Washington DC, 112 p

[2] Project 1380-D, SCSI on Scheduled Transfer Protocol (SST) working draft revision 05 (2000, unpublished). National Committee for Information Technology Standards T.10, 35 p.

[3] Bruggencate M., Lamb G., Voellm A. et al. (2001, unpublished), Libst 2.0 design , 21 p.