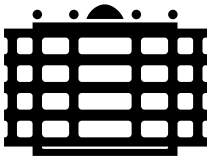


CHEMNITZ UNIVERSITY
OF TECHNOLOGY

Department of Computer Science

Chair of Computer Architecture

Diploma Thesis

Application Specific Optimization of Ethernet Cluster Communication

Matthias Treydte

Chemnitz, March 29, 2007

Supervisor : Prof. Dr. Wolfgang Rehm
Advisor : Dipl.-Inf. Torsten Höfler

Abstract

In this diploma thesis, the implementation of a networking protocol, which is adopted from a previous work, is refined to a state where it is ready for production use. The protocol is tailored for the specific needs in a cluster environment which uses standard ethernet as the interconnection hardware. Thereby focuses is on developing and implementing a flow control which is very lightweigt in terms of processing overhead and copes the challenges of such a tightly coupled network gracefully. Second, the Log(G)P models for parallel computation are examined and it is shown that they do not give a usable definition of the overhead or latency parameters for protocols whose overhead is $\mathcal{O}$(message size)–instead, the P-LogP model is used to characterize the protocol and for comparisons to the TCP/IP protocol suite. Last, it is demonstrated that the protocol gives usable results with real-world scientific applications.

Contents

1 Introduction

1.1 Motivation

In a previous work [Rei06] developed the foundation of a new network protocol, which is potentially beneficial for running parallel scientific applications in cluster environments. This protocol showed promising results in terms of latency reduction compared to TCP/IP, but it lacked any provisions dealing with flow control or network contention, it could not deal with these situations, and did not make any progress with data transmission. This clearly shows, that these facilities must not be omitted.

Therefore, the simplifications a solution tailored specifically for a cluster environment may exploit, possibly lead to flow control scheme which can maintain the benefit of the previous work. Such an solution can possibly speed up the execution parallel applications while retaining the main benefits of using standard ethernet hardwardware in a cluster, which are the unrivalled price and the ubiquitousity.

1.2 Task Formulation

The objective of this diploma thesis is to get the results of a previous work [Rei06] ready for the use with scientific applications while retaining its benefits of low CPU overhead and latency [HRM+06]. One main problem is ESP's bad performance in situations of network congestion. Therefore a method for flow control which fits the needs of a cluster environment shall be developed and implemented. A benchmark uncovering different network congestion situations has to be implemented to show the results of the optimization. Additionally, the CPU overhead has to be assessed and compared with other implmementations (e.g. TCP/IP). Finally, some selected applications shall be chosen to show the benefits of the new approach compared to other techniques.

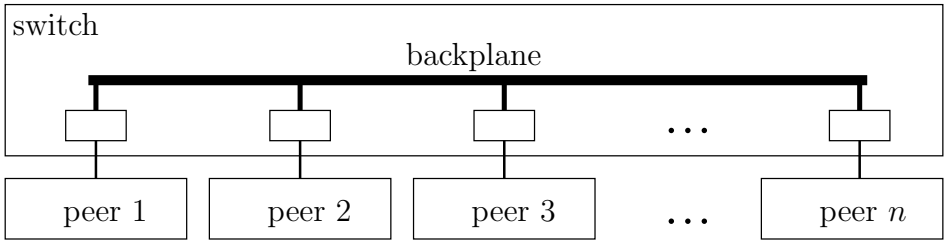


Figure 1: Topology of a cluster environment. Each of the peers is directly and equally connected to its port on the switch. The ports on their part are connected to the switch's backplane. Thick lines are part of data transmission paths, and line width is proportional to available bandwidth.

2 Congestion Avoidance

2.1 Different Types of Congestion

There are two kinds of congestion which can occur in a computer network. These are receiver congestion and network congestion, which will be explained in more detail below, observing the specifics of a cluster environment.

2.1.1 Receiver Congestion

Receiver congestion occurs if the recipient of data is unable to process it at the same speed the data arrives at it's network interface. The networking protocol stores the data received into socket buffers which are a stopover before the user-space application which initiated the communication is ready to consume the data received by calling `recvmsg` or an equivalent. If this application does expensive computation on the data or experiences shortage in I/O performance, the networking protocol may run out of socket buffers. When there are no more socket buffers available it will be forced to drop incoming packets, which are lost then.

Receiver congestion is very easy to detect by the software implementing the network protocol because it knows when it runs out of socket buffers for storing the packets coming in from the network. More specifically it even knows exactly how many buffer space it has in spare at any time.

2.1.2 Network Congestion

Basically, network congestion means that the network can not bear the amount of data currently transmitted between communication hosts [Flo00]. Network congestion can happen wherever in a network a store-and-forward approch is used to mediate between a section N_1 offering a bandwidth C_1 and a section N_2 offering bandwidth C_2 with $C_2 > C_1$. Store-and-forward means that there is a buffer where the packets for N_2 are stored at rate C_1, before they are forwarded to their destination.

As long as this buffer is not empty, the data is sent out to N_2 at rate C_2 and finally removed from the buffer. A lower bound on the size S_{buff} of this buffer for a network with a maximum packet size of $p_{\max}$ bytes is given by the equation:

$$S_{\text{buff}} \geq \frac{C_1}{C_2} \cdot p_{\max} \tag{1}$$

A typical size for this buffer in ethernet switches is around 128 kB. Whenever this buffer is full and an additional packet arrives from N_1 this packet has to be dropped and is lost. That is what commonly is referred to as network congestion.

In a cluster environment this can happen for data leaving the backplane and entering the per-peer port buffer[1], but it becomes apparent only if the bandwidth actually *utilized* in N_1 is greater than C_2. Because all peers are connected to the switch with equal upstream bandwidth, two or more peers have to send data to a common third peer at full speed to generate congestion. Then data for the receiving peer comes in at a rate several times higher than it can be forwarded to it's destination. The per-port buffer of the switch will fill up, and eventually the switch will be forced to start dropping packets.

Recapulating, in a typical cluster environment, where all peers are interconnected by one or more switches using a backplane capable of offering full bisection bandwidth, the chances for network congestion to occure are rather limited to certain well-known situations.

2.2 Preventing Network Congestion

Network congestion can be prevented by limiting the rate at which data is emitted into the network by the involved hosts. The natural way of doing so would be to obey a limit of the form

$$\frac{\text{data}}{t} \leq C_{\text{limit}}$$

Unfortunately, this leads the problem that t is surprisingly hard to measure. It is virtually impossible to have clocks with sufficient low granularity under Linux these days, as with any general purpose (i.e. non-realtime) operating system. A typical granularity at which tasks can be scheduled in these environments is 10 ms or worse. A gigabit ethernet device would have sent out about 1.28 MBytes of data in this time frame, which is far too much as a base for implementing a reasonable flow control.

Observing the fact that transmission speed at physical layer in the OSI reference model [DZ83] is constant for ethernet networking devices (as for almost every networking device) leads to the insight, that limiting the amount of data which is "in flight" on the network is sufficient for preventing network congestion. The term in flight refers to data which has left the networking device of the sending peer but has not yet arrived

[1] Actually today's switches know another mode of operation called cut-through switching. Rather than storing and forwarding full packets the switch only waits for the packet's header to arrive, so it knows the destination address. As soon as the responsible port is known, data is forwarded there. This mode offers lower latency but it is prone to collisions. It is disabled when collisions become too frequent, e.g. more than one peer tries to send data to a single receiving peer.

completely at the networking device of the receiving peer. Being pessimistic it has to be assumed that this data is currently stored in a buffer threatened by overflow as outlined in section 2.1.2.

2.3 Simplifications in a Cluster Environment

Typically, congestion avoidance on a large-scale network as the internet is a task at the mercy of many unknown quantities. The available bandwidth may change at any time, new routes may appear, previosly used routes may disappear [Flo01]. In order to develop a congestion avoidance scheme suitable for clusters, the following simplifications will be exploited in section 3.2:

- The bandwidth available between two peers chosen at random from the set of peers in the cluster is the same for any two peers.
- This bandwidth available between two peers is constant over time, given those two peers do not communicate with other peers.
- The bandwidth available to a group of peers is completely independent of the bandwidth consumed by any other group of peers as long as there is no communication between these two groups. This is a direct conclusion of the claim for a network offering full bisection bandwidth.
- There is only one route between two peers at any time. (Even if physically loops exist in the topology, techniques like the spanning-tree protocol [STP97] are exerted by the switches to logically eliminate them.) This means that packets are neither duplicated nor reordered.

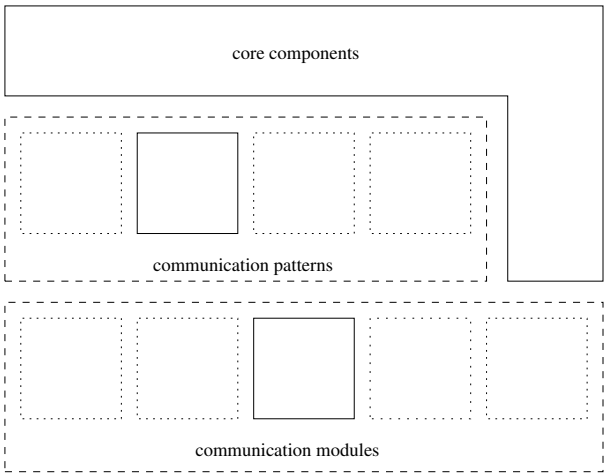


Figure 2: The structure of Netgauge. A component being on top of another means that it makes use of services supplied by the lower component. Among the modules and patterns only one may be chosen for a actual run of the benchmark.

2.4 Measuring Congestion Performance

2.4.1 Introducing Netgauge

To show the benefit of this work, a benchmark had to be developed to measure the performance of the ESP protocol specifically in the presence of network congestion. As a starting point for this benchmark, "Netgauge" was chosen, a network performance analysis tool developed here at the chair of computer architecture.

Netgauge had to be extended to be capable of provoking network congestion in a cluster environment, because its behavior, when given a set of peers to measure bandwidth between, was to split them in two equal-sized groups at random, assigning each peer in one group a partner in the other group and average out the network performance between each of these pairs, thus measuring bisection bandwidth. This scheme of operation does not generate any network congestion in a cluster environment, as explained in section 2.1.2.

Therefore Netgauge had to be reworked so it can use different communication patterns. Netgauge now is organized as a framework supporting the development of two kinds of components: transport modules and communication patterns, where every communication pattern can use any transport module for doing the actual data transmission. The architecture is shown in figure 2.

2.4.2 Transport Modules

Transport modules are responsible for transmitting the data over the various the networks and protocols which Netgauge supports. This is accomplished by having a interface common to all modules in terms of function prototypes, which are to be implemented.

An important aspect of the transport modules is the abstraction from the addressing scheme which the underlying protocol uses. The rest of the program identifies the hosts involved in the current benchmark by the means of integer numbers ranging within $[0,\ n-1]$, where n is the number of hosts involved in the current run of the benchmark. For example, the transport modules for the TCP/IP and ESP protocols internally use look-up tables to map the given integeres to the file descriptors, which are used for the actual communication.

The communication module is responsible for establishing connections between the peers involved in the benchmark in a way allowing every peer to send data to every other peer. Because socket connections are bidirectional by default, for the TCP/IP and the ESP protocols it is sufficient if peer i makes connections to peers $[i+1,\ n-1]$, with $i \in \{1, 2, \ldots, n-2\}$.

The most important functions used for communication, which every module has to implement are:

- `int sendto(int dst, void *buffer, int size)`
  This function is used for sending data to another peer. It takes the peer to send data to, a pointer to a buffer containing the data to transmit and as the last parameter the amount of data to send to the other peer. The returned value gives the number of bytes actually sent out with the function call, or be an error code.

- `int recvfrom(int src, void *buffer, int size)`
  This is for receiving data from another peer. It's complementary to the `sendto` function and takes the peer to receive data from, a pointer where the data should be stored at and the amount of data to receive. Similar to the `sendto()` function, the number bytes actually received is returned.

- `int select(int count, int *clients, unsigned long timeout)`
  This mimics the POSIX `select(...)` function, but with a simpler interface. The purpose of this function is for being able to multiplex the communication between a receiver and several senders. It is given an array of senders from which incoming messages are expected and returns one of them, from which some data has already arrived.

These functions are to be filled in a `struct` as function pointers, together with some auxillary information. The implementation of the communication pattern which shall be benchmarked the makes use of them to perform the desired data transmissions. The lifecycle of a communication module is as follows:

1. The module gets chance to parse its supported command line options if it has any, via the module-supplied `getopt()` function.

2. The `init()` function is called. Here the module may check its prerequisites (e.g. if a network protocol is available) and set it up. If this function returns non-zero (indicating a problem) the modules' life ends here (especially the `shutdown()` function will not be called).

3. Otherwise, a call to `setup_channels()` will instruct the module to set up channels between all peers. If this function returns non-zero (indicating an error) the next function called will be `shutdown()`, being the last in the modules' lifecycle.

4. If setting up the channels succeeds the main program will do calls to the data transmission functions in an unpredictable manner, controlled by the communication pattern currently used.

5. When the communication pattern is finished with its data transmission phase, the main program will call `shutdown()` to give the module a chance to free all occupied resources.

The functions `getopt()`, `init()`, `setup_channels()` and `shutdown()` are optional. This means that if such a function is not set by the module (the function pointer is NULL), no function call will be made and the main program will go on as if the function call was made and succeeded.

2.4.3 Communication Patterns

Communication patterns ("patterns") implement the actual workings of a specific benchmark. They control when a peer sends data to another peer, when to receive data and which times and throughputs to measure. To achieve this, patterns can make use of the functionality the rest of Netgauge provides.

In particular this means usage of the communication module which was selected for data transmission, as well as supporting features for gathering the statistical information for the benchmark provided by Netgauge.

2.4.4 A Communication Pattern to provoke Network Congestion

Having the new design of Netgauge as a foundation, it was an easy task to implement a pattern which can be used for measurements on network congestion, especially in a cluster environment. As already mentioned, the effect of network congestion in a cluster is that the per-port buffer(s) on the switch cannot hold the data coming in for the host on the specific port and thus some packets have to be dropped. This situation shall be provoked by the communication pattern developed here.

To achieve this, the following scheme of operation was chosed for this communication pattern: From the set of peers taking part in the current run of the benchmark, one is chosen at random to act as the server, and all other peers become clients to this server. Communication happens only between the server and the clients, there is no inter-client communication.

The algoritm the server executes is fairly simple. It basically consumes all data it receives from the clients, and when the full message from one client is received, a single byte message is sent back to the specific client. The scheme of sending the full message only towards the server, but only a single byte message back was chosen because on the way back to the clients, the data transmission is not impaired by congestion effects, which

would obscure the true magnitude of network congestion effects. The only complexity on the server side arises for the need to multiplex the communication using the `select()` function the module provides. The pseudo code for the communication pattern is given in figure 3. It is worth to mention that the sockets have to be set to non-blocking operation, as otherwise the server would wait for all of the data from the client which gets the first packet to the server in line 7, instead of serving the other clients in between. Each of these algorithms is executed in a loop several times, and the measurements are averaged out to compensate for temporal fluctuations.

2.4.5 Results and Comparison

By having the possibility to implement custom communication patters, Netgauge's fields of application multiplied. Because a communication pattern is independent from the networking protocol and even the hardware used, it's a great tool for comparing different interconnection techniques. And reciprocally, the transport module being independent from the communication pattern makes it easy to measure different aspects of networking performance while having to implement a transport module only once.

An important aspect when making changes to a benchmark tool is the comparability of the metered results. To validate the results of the new Netgauge, several test runs were performed, once with the old version of Netgauge and once with the new one. For the new implementation the communication pattern "one-one" was created, which mimics the behavior of the old implementation. Some of the results of these test runs are shown in figure 4, and the results for the one-many pattern are shown in section 6.5.2.

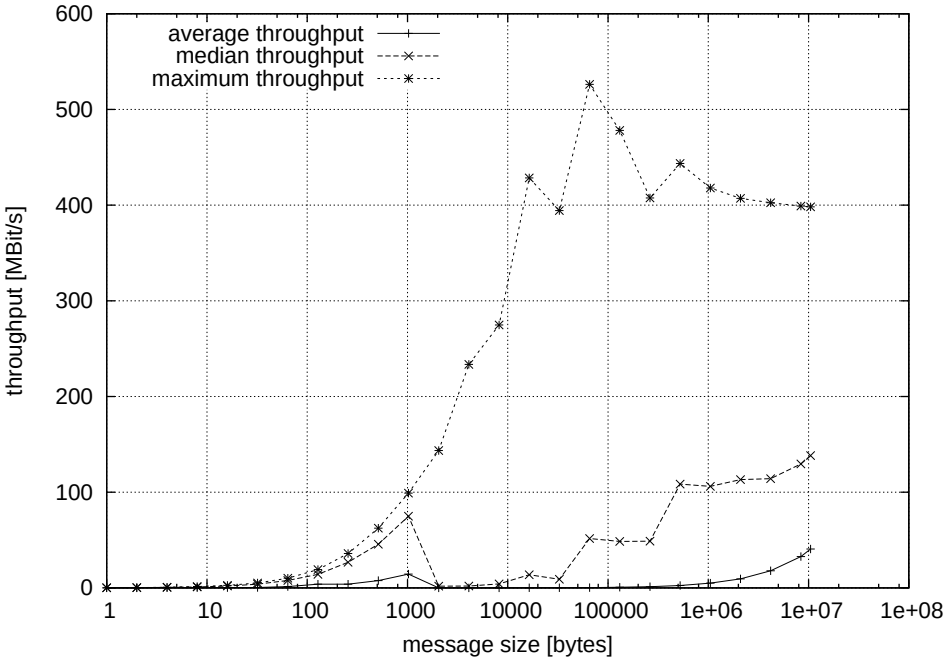
Listing 1: Server side

```
while (!<enough data received from all clients>)
{
  /* select() from the clients */
  client = select(<set of clients>);

  /* receive client's data */
  size = recvfrom(client, <buffer to recv to>,
     <data size of current run>);

  <record how much data was received from the client>

  if (<this client is done>) {
    /* ping back */
    sendto(client, <buffer>, 1);
  }
}
```

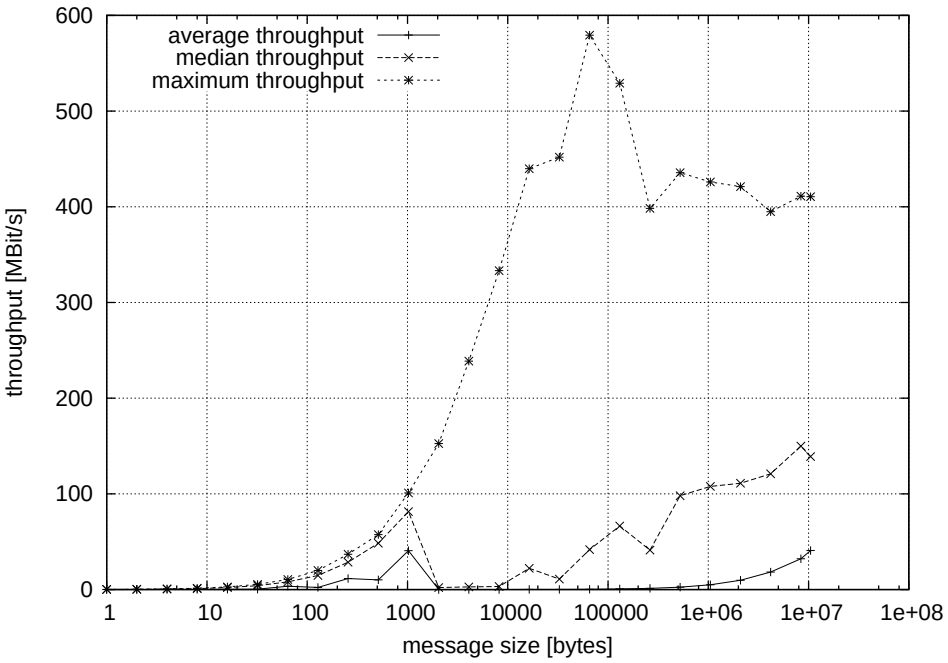
Listing 2: Client side

```
<take before-starting time>

/* send data to server */
sendto(<server>, <buffer>, <data size of current run>);

/* wait for ping */
recvfrom(<server>, <buffer>, 1);

<take after-receiving time>
<compute and store throughput>
```

Figure 3: Pseudo code of the one-many communication pattern as developed for Netgauge to generate a network congestion situation.



(a) Old implementation of Netgauge



(b) Pattern "one-one" from the new implementation

Figure 4: This plot shows the comparison results for the old and the new implementation of Netgauge, using the TCP/IP protocol on cluster 1.

3 Flow Control in a Cluster Environment

3.1 Evaluation of Flow Control Schemes for TCP/IP

The key component of flow control for TCP is the size w of the congestion window used. This value determines the amount of data the sending host might push into the network without having received an acknowledgement and thus is considered "in flight". There are a few occasions when this parameter is adjusted, which are [Jac88]:

- The receipt of an ACK: $w \leftarrow w + \frac{a}{w}$
- The detection of packet loss: $w \leftarrow w - bw$
- The receipt of an ACK during slow-start: $w \leftarrow w + \frac{w}{c}$

As a result of the need of compatibility among the different implementations of TCP/IP, many aspects of the different flow control schemes outlined here can be expressed in terms of concrete values for the parameters a, b and c, which defined in units of the maximum segment size (MSS) of the link used.

The primary goal of an congestion avoidance algorithm is to choose a window size w such that the network in between the communicating hosts does not become congested, neither the receiving host becomes overloaded and has to drop packets because it runs out of receive buffer space. TCP uses two values, which both give a upper bound on the size of the congestion window:

- The send buffer size of the sending host and
- the receive buffer size of the receiving host.

The congestion window for TCP is always less than or equal to these values, as shown in figure 5. Because the sending host can not guess the available buffer space on the receiver's side, this value is communicated to the sender using a TCP header field with every ACK packet[2].

3.1.1 TCP Reno

Reno is the base flow control algorithm for most modern TCP/IP implementations. It consists of the following key features: slow start, fast retransmit and fast recovery.

For TCP Reno the parameters for the equations in 3.1 are $a = 1$, $b = 0.5$ and $c = 1$. The initial size of the send window was chosen to be equal to the maximum segment size of the connection used. In other words, TCP is allowed to initially send one (maximum sized) packet and then waits for the incoming acknowledgement(s). The congestion avoidance subsequently decides if it is valid to increase the size of the send window.

Slow start is the phase where Reno tries to detect the bandwidth capacity available to the communicating hosts. During this phase, the send window size is doubled for every

[2] Although this field is *always* part of the TCP header, it may only be evaluated if the ACK flag is set on the received packet.

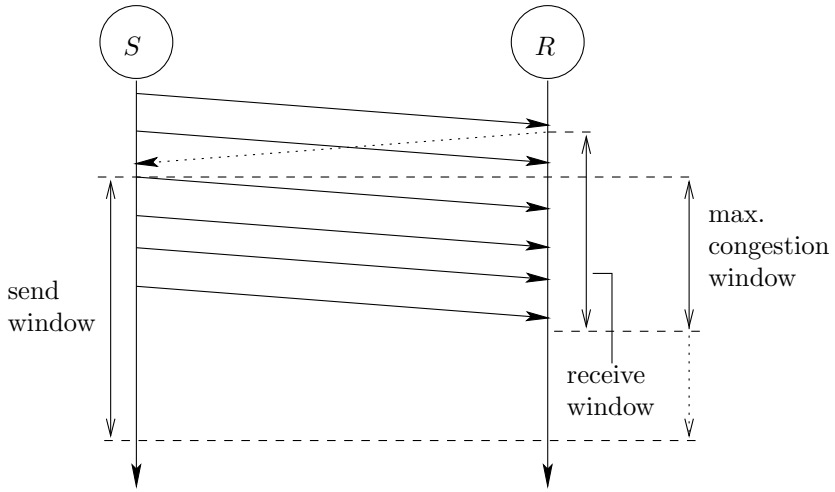


Figure 5: The dependence of the congestion window from the sending window and the receiving window. This figure illustrates how the maximum congestion window size for TCP is limited by the send and the receive window sizes at the time S receives the ACK (dotted arrow) from R.

incoming acknowledgement, until the expected acknowledgements for the packets sent begin to stay away. This event is as an indication for packet loss, which leads to the slow start phase being finished. Then the congestion congestion window is halved and the regular data transmission phase begins.

In this second phase the the growth of the congestion window is decelerated, though constantly present. Because of this, the congestion window will eventually reach a size where the link is congested again, and packets get lost. When this packet loss is detected, the congestion window is halved again, and so on... Because of this scheme of operation, TCP Reno oscillates around the optimal size of the send window, instead of using it.

Fast retransmit and fast recovery as defined in [Ste97] are an extension to the original Reno flow control, which may be used to detect the loss of packets faster. Following this extension, the receiver informs the sender about a intermediate packet being lost by acknowledging the last packet successfully received in-sequence packet again, for every out-of-order packet arriving. The sender notices these duplicate ACKs, and after receiving the third of them he transmits the lost packet again, and thus avoids having to wait for a timeout to finish. Fast recovery avoids halving the send window upon the detected packet loss. This is accomplished by observing that each of the incoming duplicate ACKs indicates a packet was successfully received, though out of order. So, for each of these duplicate ACKs the send window is raised again.

One of the main drawbacks of Reno when used in high-performance networks is its peculiarity to interpret every packet loss as the result of network congestion. Under this view it is a straight consequence to shrink the congestion window whenever packet loss is detected. Reno *needs* packet loss to detect the optimal congestion window.

3.1.2 High Speed TCP

High Speed TCP [FRS02] is an approach to improve the bandwidth utilization of TCP mainly by modifying the TCP response function [Jac88] so that packet loss is tolerated by the congestion avoidance to a certain level, without backing up the window size. This is accomplished by turning the determining parameters a and b into functions of the current size of the send window w, which are found to be

$$\begin{aligned} a(w) &= \frac{w^2 \cdot 2.0 \cdot b(w) \cdot P}{2.0 - b(w)} \\ b(w) &= (B - 0.5)\frac{\log(w) - \log(W)}{\log(W_1) - \log(W)} + 0.5 \\ a(w) &= 1 \text{ for } w \leq W \\ b(w) &= 0.5 \text{ for } w \leq W \\ B &= b(W_1) \end{aligned}$$

this allows to tune the shape of the response function (which determines the window size) to be even more aggressive than the original response function, once the window size w has grown over W by choosing appropriate values for W_1 and P, where W_1 is the desired size of the send window for a packets loss rate of P.

The benefit of this modification is that High Speed TCP has a better tolerance for packet loss, and accepts a certain loss rate as being unavoidable in a computer network. Still, High Speed TCP oscillates around the optimal congestion window size, as Reno does.

3.1.3 TCP Vegas

TCP Vegas was developed by Lawrence S. Brakmo et al. [BMP94] and introduces several changes to the sender side of a TCP connection. These changes lead to the remarkable result of improving the throughput by 40–70 % and lowering the amount of retransmissions to one-half to one-fifth compared to Reno. A brief overview of how this is achieved shall be given here.

Vegas uses a new mechanism to decide when to retransmit, which uses the receipt of an ACK in certain situations as a trigger to check if a timeout should happen. This allows neccessary retransmissions to be sent out earlier. Secondly, Vegas uses a modified window sizing when congestion is detected. Basically, if the detected packet loss concerns only packets which were sent out *before* the last shrink of the congestion window, this is not taken as an indication to shrink the congestion window again. This is so because the old window might have been too large, but it's still possible that the new window is sized appropriately. Vegas also incorporates a spike-suppression facility which aims to more evenly distribute the emitted data packets over time. Lastly, Vegas eliminates Reno's need for packet loss to estimate the possible throughput (and thus the congestion window size w). Instead, this is done by comparing the actual througput to the expected

throughput, which is:

$$\text{Expected throughput} = \frac{w}{\text{baseRTT}}$$

If the actual throughput is smaller than the expected throughput, the window size is increased and vice versa. Additionally there is a threshold value used to prevent the window size from oscillating.

The original Vegas implementation had some issues when rerouting leads to a new route with a larger propagation delay, as this could not be detected and thus Vegas' value for baseRTT was not updated. These have been solved recently, though [SJA03]. All these advantages make it desireable for Vegas to become the default TCP implementation on the internet soon.

3.1.4 Conclusion

The basic approach of all flow control schemes mentioned in this section is to constantly try to go faster and to back up whenever packet loss is detected. This is passable thing to do for large-scale networks like the internet where bandwidth may change all the time.

In a cluster environment, several of the challenes TCP tries to fight with this strategy simply don't exist or are limited to well known situations. Problems arise especially if multiple senders exist, which try to send to a single receiver. In this situation, if one of the senders uses a send window size which is too large and causes congestion, it is well possible that the result of this is that a packet of *another* sender gets lost. Consequently, the sender whose packet was lost will back up. Additionally, the sender that caused the congestion will keep increasing its congestion window. This situation can be seen as a set of dependent control loops (one loop for each sender) which "fight" for the available bandwidth, and it can take considerable time for this system to reach equilibrium.

3.2 Design of a Flow Control Scheme suitable for Clusters

The most important goals for the flow control scheme to be developed are:

- Reliable data transfer
- Reach high speeds without requiring unrealisticly low packet loss rates
- Preserve ESP's low latency and processing overhead
- Keep the header as small as possible for maximum efficiency

As depicted in section 2.3, several assumptions about the available bandwidth can be made in a cluster environment, which do not hold true for large-scale networks. The flow control scheme developed in this section tries to exploit this knowledge, trying to gain better results than TCP does, especially aiming to reduce packet loss significantly in multiple-sender situations.

Foremost, the bandwidth is constant over time. This elimiantes the need to probe for it over and over again by trying to send "some more" data as TCP does. Having a

constant value being the optimal size of the congestion window w (chosen large enought to be capable to saturate the link), and using it whenever possible seems like a good starting point for a flow control scheme in a cluster. It's got the potential to keep up with TCP/IP flow control schemes, no matter how well the heuristic controlling the probing process may be chosen. There is no congestion-implied packet loss if the number of packets in flight with a common destination host, P_{cd} is limited so that

$$P_{\text{cd}} \leq \left\lceil \frac{S_{\text{buff}}}{p_{\max}} \right\rceil \quad , \tag{2}$$

with S_{buff} being the per-port buffer size on the switch and $p_{\max}$ being the maximum packet size of the ethernet.

3.2.1 Basic Principles

The constitutive idea to loose the fighting for bandwidth which TCP holds in a multiple-sender situation is to unravel the time-division multiplexing (TDM) which inevitably occures on wire, when two or more hosts concurrently send data to the same destination host. The idea is to give each of the competitive senders a time slice it may use to send out some packets carrying data.

The task of managing the TDM is assigned to the receiving host of the multiple-sender constellation. This host is naturally involved in each of the competing transmissions, and therefore has the ability to detect (and manage) a multiple-sender situation. The fundamental principles used to achieve this aim are the following:

1. The window size w is a constant value which is common to all hosts within the ethernet's broadcast domain (i.e. the cluster). It must be chosen large enough to allow saturating the link.

2. A sending host may emit data into the network as long as the congestion window allows it. The sender does not make any assumptions about the round-trip time. Especially it does not do retransmissions[3] unless it is asked to do so. Incoming acknowledgements may open the congestion window, allowing some more data to be emitted.

3. A receiving host takes care not to send out too many acknowledgements to the sending host(s) to prevent the per-port buffer on the switch from overflowing. When the receiver detects packet loss it must request a retransmission of the lost data from the sender.

3.2.2 Handling of Receiver Congestion

By following the rules defined in section 3.2.1, developing a scheme for handling receiver congestion is very straightforward: As mentioned in section 2.1.1 the receiving host can

[3] An exception to this rule is explained in 3.2.4.

easily detect receiver congestion. This is done by comparing the amount of memory currently used for the receive buffers to the fixed quota which is set on this value.

If the quota does not allow for storing another burst of w full sized packets, the sender does not acknowledge the receipt of data to the sending host. As a result the sender will assume the data is still in flight, and by obeying the congestion window size it will stop sending new data. This gives the application on the receiver side the chance to consume the data currently in the receive buffer, thereby making room for new data. Once there is sufficient space free, the receiver subsequently acknowledges the receipt of the data, which opens the congestion window on the sender side and allows the transfer of additional data.

There are two drawbacks which arise from this way of handling receiver congestion: First, the utilization of the receive buffer is not optimal, as in average when entering receiver congestion $\lceil w/2 \rceil \cdot p_{\max}$ bytes will remain unused there. It is worth to mention that this memory is not wasted, it just is not used though the user set (respectively the protocol default) value for the socket's receive buffer size would allow to do so. This memory is never allocated, so choosing a slightly larger quota for the receive buffer fully compensates for this effect.

On the sender side the situation is a little worse. The sender has to store every sent packet into the send queue and may drop the packets there only if the receipt of them is acknowledged. By not sending an ACK frame the receiving host leaves the sender uncertain about the successfull transmission of the data, which will therefore have to remain on queue. Indeed, this prevents some memory from being deallocated which could have been freed if the sender would have known about the successfull transmission.

3.2.3 Handling Retransmissions

Packet loss is a problem even the ideal congestion avoidance algorithm (in terms of never forcing a unit using a store-and-forward scheme to drop a packet, neither overcharging the processing capacity of the receiver) will have to cope with, as the ethernet with a packet corruption rate of 0 still remains to be developed.

As proposed, the usual retransmission scheme which TCP utilizes shall not be used here. Because the sender does not make any assumptions about the round-trip time, it doesn't even know for when to schedule a retransmission timer anyway.

Therefore, instead of letting the sender decide when to do a retransmission by using a timout value or by guessing multiple ACKs mean some data was lost as practised by the fast-retransmit algorithm [Ste97] of the TCP/IP protocol, a new flag packets may carry is introduced. This new flag is called the retransmission request (RRQ) - flag. A RRQ - flagged frame must always have the ACK flag set as well. The acknowledging aspect of this frame tells the sender up to which packet the data was received successfully; and the RRQ instructs to retransmit as many frames, counting from the first still unacknowledged frame, as the size of the send window allows.

There are two occasions which result in a RRQ to be sent. One is the receipt of a frame with a packet sequence number which is higher than the next expected sequence number. The sequence number of the next expected data frame is recorded with the

socket and monotone increased[4] by one for every data frame successfully enqueued onto the receive queue. As there are no loops in the network, packet reordering does not have to be taken into account here, and a jump in packet sequence numbers really means that the packets inbetween have been lost. This is sufficient for detecting most packet losses, but not each.

The situation when this does not work is when a contiguous number of frames off the end of the bulk transmission is lost, including the worst case where every single frame sent as response to the last ACK was lost. This situation is handled by a timeout, which is given by the round-trip time of the network.

The receiver also has to know when the sender is willing to send more data (and thus when to have a RRQ timeout scheduled), and contrary when the sender is finished with the current transfer and this timer should be stopped. This problem is handled by two more flags, as explained in the next section.

3.2.4 Requesting Transmission Slots

The problem that still remains residual is how the receiver can decide if there is still data to be expected from the sender. Just having an open connection in no way means that there currently is data to be exchanged. Its perfectly legal to set up a connection between two peers and close it afterwards without exchanging a single byte of payload over this connection. But the receiver needs to to know for which sockets to have the RRQ timer running. If the timer would be trivially running for any socket residing in a connected state, the protocol would not scale well in presence of many connections.

The solution chosen to overcome this was to introduce two more flags. These are called "start of transmission" (TXS) and "transmission finished" (TXF). The sender has to set the TXS flag on the first packet of a message to be transmitted and the TXF flag on the last packet of this message. Messages of a length less than or equal to the maximum segment size of the network, which fit into a single packet, should should have both, the TXS and the TXF flag set. By evaluating these flags the receiver can easily decide when to start or stop the retransmission request timer.

There is one pitfall about this way of handling the scheduling of the RRQ timer, which becomes apparent if the packet carrying the TXS flag is lost. As the receiver in this case never gets to know that there is data pending to be transmitted, it will not schedule the retransmission request timer, resulting in the transmission to linger forever. To overcome this the sender keeps retransmitting the packet carrying the TXS flag until the successfull receipt is acknowledged.

A similar problem arises for the packet carrying the TXF flag. When this packet has successfully arrived at the receiver he will send an ACK frame to the sender. This final ACK can even be sent without taking care of network or receiver congestion, as it won't trigger any new data to be send. It just allows the sender to clear it's send queue. But if this final ACK is lost this cleanup will never happen. That's why a retransmission

[4] In fact there is a wrap-around. So adding 1 to $2^{16} - 1$ leads to the next sequence number expected being 0.

timer is started on sender side which keeps sending this TXF flagged packet until the acknowledgement for it is received.

3.2.5 Deciding when to send an Acknowledgement

The receiver has to send out acknowledgements to the sender to open its congestion window and thus allowing more data packets to be sent out. Additionally the sender may remove all acknowledged packets from its write queue to allow the sending application to put more data in there, if it still some part of the message pending.

Obviously there is some elbowroom on the rate r_{ACK} at which acknowledgements are sent. The lower bound is to send an ACK for each and every data frame received. This would work like expected but adds significant processing overhead [Rei06] on the sending as well as on the receiving side, which seems unneccesary. The upper bound is given by the window size w. If r_{ACK} was greater than w this would in practice mean no ACKs are sent at all. Any progress in data transmission would solely rely on the RRQ timer to kick in and send out the ACK, resulting in a very jerky data transmission.

The upper bound has to be lowered even further by respecting the time it takes for the ACK to be constructed, sent out, transferred over the network and finally being received and processed by the sender. It's desirable this time overlaps with the time it takes for the last frames of data within the current congestion window to be emitted by the sending host. This gives a new upper bound of

$$r_{\mathrm{ACK}} \leq w - \left\lceil \frac{T_{\mathrm{SAT}}}{2} \right\rceil \quad , \tag{3}$$

where T_{sat} is the minimum sending rate needed to saturate the link, measured in MSS-sized packets per round-trip time (PPR). In practice this value should be lowered even further to keep the ethernet device sending in case there is a small hiccup (e.g. because the kernel decides to swap some pages out or similar) preventing the ACK from being processed immediately. In general, it's advisable to choose r_{ACK} as large as possible (respecting the above limitations) in order to minimize the processing overhead.

3.2.6 Handling multiple Senders

The task of handling situations when more than one host wants to send data to a common receiving host is handled completely by the receiver. As mentioned in section 3.2.1, the receiver has to take care not to emit too many acknowledgements to prevent network as well as receiver congestion. The other aim that should be achieved is to keep the link saturated.

When there is only a single sender these requirements are fullfilled by simply choosing w large enough while sending out ACKs just in time (see equation (3)), so the sender keeps emitting packets without a rest until the full message is transferred. In the threat of receiver congestion the receiver may simply stop sending ACKs and the sender will calm down until the next ACK opens the congestion window again. This scheme gives a nice yet simple way to avoid congestion by exploiting the simplifications a cluster

environment allows, but it still is focused on a single connection (i.e. a pair of connected sockets).

In the presence of multiple senders, their values for w add up and its conceivable that if n being the number of senders, $n \cdot w$, which is the upper bound on the number of packets in flight with a common destination host P_{cd}, may violate the rule of equation (2). Broadening the view to all sockets currently receiving data on the current host leads to a surprisingly simple way to get around this. All that is needed is an instance which holds back the odd acknowledgements to prevent too many senders from emitting packets addressed to the common receiver.

All this instance has to know is the number of currently active incoming transmissions n_{recv}, which can be easily incremented when a TXS flagged packet is received and decremented upon the receipt of a TXF flagged packet. Now whenever a socket feels it should send out an ACK because r_{ACK} packets have been received, this ACK is given to this mediating instance, which has to take care to have held back at most $n_{\text{recv}} - 1$ ACKs at any time.[5]

In order to be fair to the sending hosts its advisable to use a first-in first-out scheme on the ACKs held back. But this can be changed to any order, for example to give preference to sockets with a higher priority in further extensions, if desired. Figure 6 illustrates the exchange of packets and acknowledgements for a situation with two senders and one receiver.

3.2.7 Additional Differences to TCP

The meaning of the sequence numbers used by TCP and ESP differs slightly as a result of exploiting another simplification a cluster environment offers: While the sequence numbers of TCP in fact count bytes, for ESP they simply count packets of an arbitrary size, which is limited by the maximum segment size possible on the network.

TCP's need for the finer granularity of the sequence numbers arises from the possibility of fragmentation [Ins81]. For TCP, a n-byte packet with a sequence number of s could at worst be fragmented into n 1-byte packets by any intermediate router. Each of these 1-byte packets still needs an individual sequence number in the range $[s,\ s + n - 1]$ to allow the reconstruction of the original message in case these packets arrive out of order at the receiving host. Therefore, after a n-byte packet with sequence number s has been sent out, the next packet must carry the sequence number $s + n$ to avoid collisions with the sequence numbers. As fragmentation does not occure in a switched ethernet (which resides in layer 2 according to the OSI reference model [DZ83]), just numbering the packets is sufficient for ESP.

The measure of the window size needs to be a multiple of the measure of the sequence numbers because the send window is always defined *relative* to the current position in the data stream and there is no such thing like fractional sequence numbers. For TCP the measure of the window size was originally chosen to be 1 byte. Additionally, the free space in the receive buffer, which the receiver communicates to the sender in every

[5] The number of ACKs held back may be less than $n_{\text{recv}} - 1$ because of sockets already having incremented the receiver count but not yet received R_{ACK} packets.

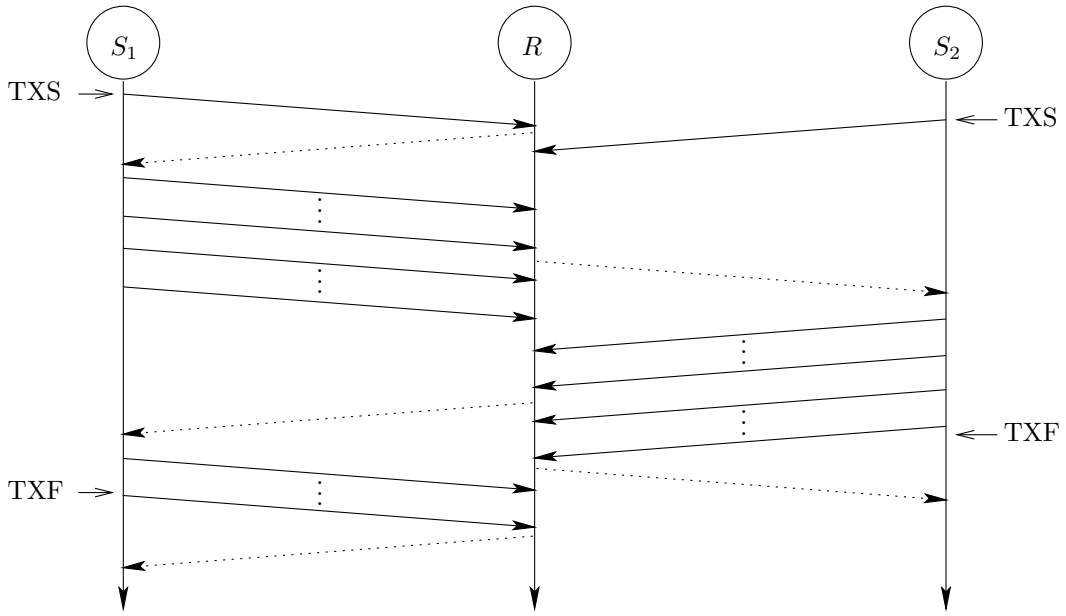


Figure 6: Schematic diagramm of a multiple-sender transmission where the receiver services the senders in a first-in first-out fashion. S_i are sending hosts, R is the receiver. Dotted lines are acknowledgements for received data, solid lines indicate data transmission.

acknowledgement, is limited to 16 bits. This gives an upper bound on the size of the congestion window of only $2^{16} = 65$ kbyte. Additionally, the maximum window size must be less than or equal to the maximum sequence number possible. This is so to avoid two packets with the same sequence number can be on the fly at the same time, because these packets would be indistinguishable and might be mixed up by the receiver, if packet reordering occures.

Because the sender may emit data frames only if the receipt of previous frames is acknowledged, the throughput of a connection is limited by $w_{\max}/\mathrm{RTT}$ (with $w_{\max}$ being the maximum congestion window size possible and RTT being the round-trip time). Therefore a small maximum window size directly influences the maximum possible throughput, especially on long-delay links.

This limitation became apparent very soon and was lifted by the TCP window scale option as defined in [JBB92]. The window scale option uses a previously unused part of the TCP header to arrange an agreement on a scaling factor (being a power of two), which is applied to the window size. This predefinition is determined during connection handshake, and allows TCP to use window sizes up to $2^{30} = 1$ Gbyte by trading co-domain for granularity. This leaves plenty margin to saturate a link with both, a high throughput and long round-trip times.

For ESP, the measure of the window size was chosen to be equal to the maximum segment size of the ethernet it runs atop. By assuming a MTU of 1500 bytes this gives 1489 bytes (1500 bytes - 11 bytes for the ESP header). ESP also uses a 16-bit field in its header for sequencing, so that the maximum amount of data in flight is limited by $2^{16} \cdot 1489$ bytes ≈ 93 Mbytes, which is enough to saturate even very fast links, especially because of the short round-trip times experienced in a cluster environment.

Another simplification compared to TCP concerns the push-flag (PSH). The purpose of this flag is to discourage intermediate routers from collecting several small packets and conflating them into a bigger one. While this behavior is mostly favourable, it might not be applicable under certain circumstances. For example, dialog-based applications like shell access or web browsing would show unacceptable response times. These applications typically send out a small message which acts as a request (e.g. the HTTP GET request as defined in [FIG+99]) and then wait for a response. It does not make sense for an intermediate router to wait for additional packets from the client to append to the packet containing the request, as the client on his part is already waiting for the server's response. Therefore, a facility is needed to flag these packets for immediate delivery, which is the PSH flag. Because ethernet does not know the concept of segmentation (and conversely reassembling), there is no need for a push- or similar flag in ESP.

3.2.8 Conclusion

By being dispensed from the need for compatibility to existing flow control schemes, as required for the modifications to TCP mentioned in section 3.1, a new scheme of flow control could be developed which is truly different. It possibly has the potential to utilize the the bandwidth available in a cluster environment very well, and it is free of heuristics trying to meter values which are known a priori in such a tightly coupled

network.

In addition, the proposed flow control scheme needs very little per-packet header space. Only three flags were introduced, what means only three bits are needed for the purposes of congestion avoidance and flow control. As the flags field of the original ESP implementation still had some bits in spare, the size of the header in fact did not grow at all. With a header size of 11 bytes for ESP and a MTU of 1500 Bytes[6], the header occupies only 0.7 % of a packet, while TCP/IP uses 2.7 % (20 bytes for the IP header plus 20 bytes for the TCP header).

[6] If all stations within a broadcast domain in an ethernet support this protocol extension, jumbo frames may be used. Jumbo frames are typically sized 9 – 16 kB. However, choosing a frame size larger than 12 kB does not seem advisable because of the 32 bit CRC used by ethernet looses it's expressiveness there. This is especially true for ESP as it does not perform additional checksumming.

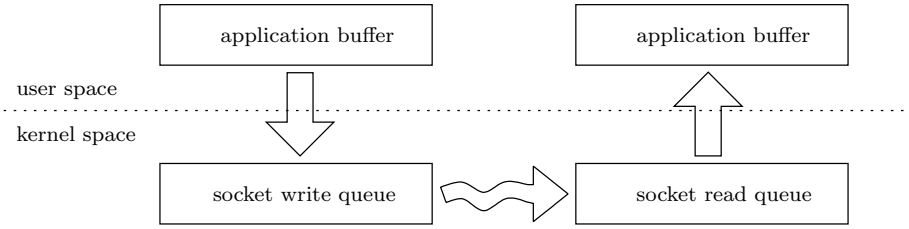


Figure 7: Stations of data flow from the sending to the receiving process. The curved arrow indicates the data transmission over the network.

4 Implementation of the Flow Control for ESP

The flow control scheme developed in section 3.2 had to be implemented on top of the existing implementation of the ethernet streaming protocol (ESP) by [Rei06]. ESPs implementation provides applications the usual socket programming interface, just like TCP. This means every data transmission is started by an application making a call to `sendmsg` or its related functions and ends with the receiving application's call to `recvmsg` (or its related).

As ESP is a streaming protocol providing reliable data transmission, the first step to be made before data can be transmitted is to set up a virtual channel (called "connection" in the following) between the two sockets. This is accomplished using a 3-way handshake quite similar to what TCP does and explained in detail in [Rei06]. Calling the functions `sendmsg` or `recvmsg` is sensible only for sockets which have entered the connected state.

As shown in figure 7, there are several stations the data has to pass before it's finally written to the buffer space provided by the receiving process:

1. First the message has to be assembled by the sending process. While this sounds trivial, this step can contain several non-trivial operations. For example, byte-order conversions could be necessary or the data might be padded out with dummy bytes to allow aligned (and thus faster) access to individual pieces, if the architecture profits by (or even requires) doing so. There is no point in transmitting these dummy bytes over the network, so they should not be included with the final message. These tasks of compositing the final message to be sent are often performed by an intermediate piece of software like the various implementations of the MPI (message passing library; [Pac96]) standard[7].

2. After the message has been composed and is ready for transmission, it's handed to the protocol layer which resides within kernel-space. This is where the ESP protocol steps in. The message is split into MSS-sized packets, equipped with a transport header, stored on the write queue and possibly the first packets are already sent out over the network. This step is explained in detail in section 4.2.

[7] At the time of this writing, only the OpenMPI [GWS05, Rei06] implementation has support for the ESP protocol layer.

3. Next step is the actual transmission of the assembled packets over the network, including retransmissions if necessary. Here is the field of activity for the flow control and congestion avoidance algorithms.

4. The packets arriving at the networking device of the receiving host are handed out to the protocol handler registered with the kernel for the specific ethernet packet type. This handler decides if a socket exists which might be interested in the incoming packet by looking at the source and destination port numbers and adresses [Rei06]. If it finds one, the packet is given to this socket for processing. Packets containing useful payload are stored on the socket-private receive queue until they are fetched by the receiving process.

5. Finally, the receiving process will do one or more call(s) to the `recvmsg()` function. Here the socket buffers are taken off the receive queue and copied over to user-space. This step is repeated until the full message is stored in the application-provided buffer. Possibly the data has to be unpacked by intermediate layers to be ready for further processing by the actual application, as mentioned in step 1.

The steps 2 – 5 are carried out by (or involve activities from) the protocol layer, that is ESP. The practical aspects of these tasks, including some optimizations applied, are to be explained in this chapter. For easier reading there is a general survey of the state information which is maintained by the ESP protocol layer in section 4.5 on page 33 ff. This should be used as a reference, if the meaning of some of the identifier names is not obvious to the reader.

4.1 Design of the ESP Output Engine

The output engine is responsible for actually emitting packets into the network. Two kinds of packets are to be differentiated here: packets carrying payload ("data packets") and packets used for status updates only ("empty packets").

Data packets are only created upon user request (see 4.2) and are always stored on the write queue until they are acknowledget by the receiver. They are sent out whenever the congestion avoidance strategy decides that it is valid to do so. Events which may trigger the sending of data, roughly ordered by the probability of their occurrence, are:

- The receipt of an acknowledgement (ACK).
- The user calls the `sendmsg` handler.
- The receipt of a retransmission request (RRQ).
- The expiration of the retransmission timer.

Empty packets (packets with a length of 0 recorded in the ESP header) are used frequently for the exchange of state information, but they never enter the write queue. These packets instead are constructed and emitted on the fly. If a retransmission of

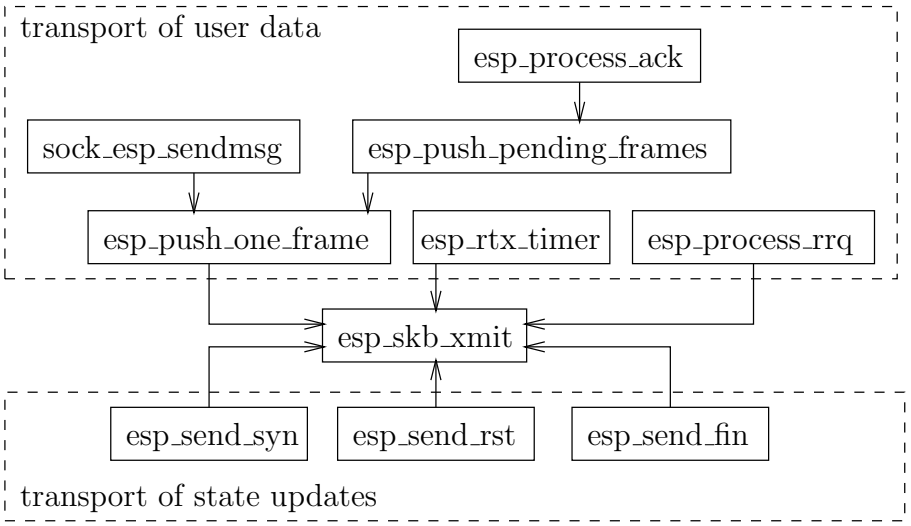


Figure 8: Structure of the ESP output engine. An arrow stands for a possible function call. The only function which actually emits packets to the network is esp_skb_xmit.

such an packet is necessary, it's simply constructed and emitted again. This procedure does introduce some negligible overhead, but it helps to simplify and streamline the implementation a lot. As these packets are very small[8], it is not neccessary to limit the emission of these packets by the congestion avoidance. If there was a noteworthy share of bandwidth used by these empty packets, it would rather indicate a bug in the implementation.

Figure 8 gives an overview of the various function calling paths leading to a packet being emitted into the network, where the upper frame contains functions which emit packets taken off the write queue, and the lower frame deals with empty (i.e. flags-only) packets. The emission of acknowledgements is not included in this figure, because it uses a special-cased implementation as explained in section 4.4. This is, because if a socket wants to send out an ACK, it might actually trigger the sending of an ACK which was enqued by another socket, as outlined in section 3.2.6. The `esp_skb_xmit()` function can not handle this situation, the a special implementation was needed.

4.2 Receiving Data from User-space

There is a single point where data is handed from user-space to the ESP protocol layer for transmission, which is implemented by the function `sock_esp_sendmsg()`. This function had to be extended in several ways in order to implement the proposed flow control scheme.

[8] Depending on what to count, the size of these frames is 39 bytes (storage space required when buffered) or 64 bytes (the minimum ethernet frame duration, which includes padding).

The first part of this function tests if the preconditions for sending data on the given socket are met, namely if the socket really is in a connected state, the shutdown of the connection has not already been announced to the receiver, and finally if the device this socket operates on has not been shut down inbetween. If all of these conditions are met, it is valid to send out data on this socket. As explained in section 3.2.4, any transmission has to be announced to the receiver by the means of a TXS-flagged packet to be sent as the very first of any transmission. To ensure this, the next step is to test the per-socket variable `trans_announced`.

If the value stored there is non-zero, the current call to `sock_esp_sendmsg()` is part of an already announced transmission, which was interrupted by a signal sent to the user-space process or the lack of write queue space and a subsequent timeout while waiting for more write space via the kernel function `sk_stream_wait_memory()`. In either case, the transmission is already announced and the `sendmsg()` handler can just go on enqueing data onto the write queue.

But if the value is zero, a new transmission is about to start, and there are two ways to deal with this situation. The first way is to minute that the next packet to be constructed will have to carry the TXS flag, and to record this packet's sequence number as the new value for `announce_seq`. This procedure initiates a new transmission.

Still, there is a more elegant solution, which can be used if the socket's `send_head` is not NULL: As already mentioned, the send head always points to the next socket buffer to be sent out *for the first time*. If this variable actually points to a socket buffer, no matter what the position of this specific buffer is on the write queue, it is sure there is data on the queue which has not been sent out before. Consequently, because `trans_announced` was zero, the last packet on the write queue will have the TXF flag set, which ought to indicate the end of transmission to the receiver. Now it is possible to just clear the TXF flag on this last packet. Doing so effectively merges these two consecutive transmissions into a single, larger one. This procedure has two positive effects: it lowers the processing overhead on the receiver side and allows the sender to stick with using the full congestion window, skipping the waiting for the first ACK, which would be required upon transmission announcement oherwise.

The rest of the function is mainly a big loop which slices the application-provided buffer into MSS-sized packets and enqueues them to the tail of the write queue. Thereby it sets the `send_head` if it was NULL and takes care to set the TXF flag on the last packet, as well as clearing `trans_announced` in this case. Additionally, in every pass a call to `esp_push_one_frame()` is made. This aims for minimizing the end-to-end latency by overlapping the receipt of data from userspace and sending the data out over the network.

Additionally, the `sendmsg()` handler function has been reworked so it can gather the data to output from multiple buffers. This feature, exposed to user-space by the `writev()` ("write a vector") function, is the primary way the Open MPI library makes use of the ESP protocol. Having a `sendmsg()` which is capable of doinng vector I/O can significantly reduce the number user- kernel-space transitions needed to carry out certain collective MPI operations, namely from the scatter / gather group of operations.

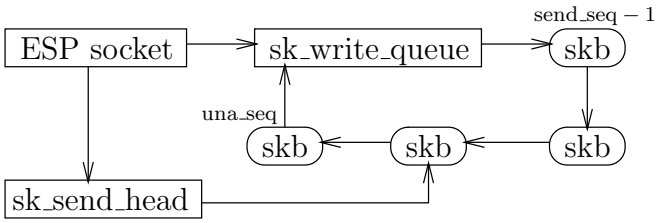


Figure 9: The ESP write queue. Arrows indicate pointers; for the write queue, the pointers to the previous element are omitted for clarity. The last SKB on the queue is the oldest, whose `pkt_seq` equals `una_seq`; the send head points to the next packet to be emitted for the first time.

4.3 Sending Data to the Receiver

As can be seen in figure 8, the regular calling stack which leads to data packets being sent out to the receiver always leads through the function `esp_push_one_frame()`, which shall be explained here along with its supporting functions.

All data being sent out comes off the write queue, whose structure is shown in figure 9. The write queue is organized as a double linked list, with the list head (pointers to the first and the last socket buffer on the queue) stored in the struct `sk_write_queue`, along with some auxiliary information used to maintain the list and provide the ability to use it from concurrently running contexts safely by providing a locking mechanism.

The first operation this function performs is to check if the socket's `sk_send_head` pointer is not NULL. If it is NULL, the function returns immediately because there is no *new* data on queue to be sent. Otherwise, the congestion avoidance is asked if it is valid to send out the packet found.

This is done by calling `esp_cwnd_test()`, which performs the following: First it checks if the last TXS-flagged packet was already acknowledged by the receiver. If this is true (because `una_seq` is after `announce_seq`), this qualifies for using the full congestion window. In this the case the packet may be emitted if its sequence number is before `una_seq` plus `esp_burst_length`. Otherwise the socket is currently waiting for the first ACK from the receiver, and only packets with a sequence number up to (including) `announce_seq` plus `esp_initial_ack_burst_length` may be emitted. If the congestion avoidance finally decides that its valid to emit the given packet, it is handed to the `esp_skb_xmit()` function and emitted as explained in [Rei06].

A special case this function takes care of is the following: Suppose the application uses non-blocking I/O and tries to send a message which exceeds the size of the send buffer. In this case the `sendmsg()` handler will copy over as much of the message to the send buffer as possible and return to user-space. As the message transmission goes on, some buffer space will become free again, which the application *might* fill up. But if it decides not to do so (which is a perfectly valid thing to do), eventually the last packet in the send buffer will be sent out, which will not have the TXF flag set. Because of this, the receiver will not know the sending socket ran out of data and the retransmission request

timer will try to recover from an apparent packet loss, which did not occur. To avoid this, the `esp_push_one_frame()` function will set the TXF flag on this very last packet found in the send buffer prior to its emission.

4.4 Managing the ACK queue

The ACK queue is the data structure where odd[9] ACKs can be stored until it's time for them to be transmitted to the sending host, where they will open the send window and allow for more data to be transfered. The occasions at which an ACK has to be sent out are:

- Received data was successfully inserted into the receive queue.
- Packet loss was detected because of a jump in packet sequence numbers.
- A TXS-flagged packet was received.
- A memory pressure situation was recently lifted because the application read some of the data which was buffered there.

4.4.1 Enqueuing an ACK

The task of enqueuing an ACK onto the receive queue is performed by the function `esp_enqueue_ack()`. This function takes two parameters: The socket which wants to enqueue an ACK and a parameter to indicate additional flags (apart from the mandatory ACK flag), which the socket wishes to be set on the packet that will be constructed.

The first action this function performs is to check if the device the given socket operates on is still up, as there is no point in enqueuing an ACK for a shut down networking device. If this check passes, the packet carrying the ACK (and maybe additional flags which were handed in as the mentioned function parameter) is constructed by calling `esp_skb_alloc_sk()` which returns a socket buffer (SKB) containing the needed network and protocol layer headers, but no data section. This socket is now charged to the socket, which is important to avoid memory being leaked and for the later identification of the originating socket of this ACK as well. Additionally, the socket's `ack_seq` is set to the current value of `recv_seq` to account for this ACK being sent out sooner or later.

The function `esp_enqueue_ack()` is called from numerous places without prior checking the value of `esp_recv_count`. Because of this, there is chance that the enqueueing of the ACK in fact is not necessary at all, as there is currently only one socket receiving data. This is checked here, and if there is only one receiver, the ACK will be sent out immediately on the fast path, which skips manipulating the ACK queue.

If this optimization is not possible because of a multiple-sender situation, the next step is to grab a lock on the ACK queue to allow its safe manipulation. Now the just created packet is added to the tail of the ACK queue, thus making it grow. This growing is desired whenever the size of the queue is less than the current value of `esp_recv_count`,

[9] Odd in the sense that sending them out would be at the risk of causing network congestion.

as outlined in section 3.2.6. If this is the case the function is done for now, releases its lock on the ACK queue and returns.

If the size of queue is greater than or equal to the count of sending hosts, it is time to send out an acknowledgment that was enqueued by another socket. Therefore the packet at the head is dequeued and the lock on the queue is released. Now the function holds a reference to a SKB which contains an ACK for some other socket waiting for data. The last thing to do prior to sending out this ACK is to start the originating socket's retransmission request timer, in order to give this socket a chance to notice if anything goes wrong, either with the transport of this ACK or the data frames the sender will emit in return when he processes this ACK.

The strategy of always anqueuing at the tail of the ACK queue and always dequeuing from the head has two positive effects: First it allows for $O(1)$ access, which is important as access to this queue happens very often, and performance has to be taken in consideration to avoid introducing a new bottleneck. Secondly, it ensures that the available bandwidth is evenly distributed among the competitive sockets.

4.5 State Information used by the Flow Control

There is several state information which has to be maintained by the ESP protocol layer. Some of it is private to the individual socket allocated, some is shared across all sockets.

4.5.1 Per-Socket Information

The per-socket information is stored either within the `struct sock` as defined by the Linux kernel, or in its ESP-specific extension `struct esp_opt`. These two stuctures are always allocated and used together and shall not be distinguished here.

- `sk_send_head`: A pointer to the next packet on the write queue to be sent out if congestion avoidance allows it. If this pointer is NULL, all packets on the write queue have been sent out at least once.
- `send_seq`: The sequence number to be used for the next data packet constructed; this also is the sequence number of the packet at head of the send queue plus one.
- `recv_seq`: The sequence number of the next data packet expected from the sender.
- `ack_seq`: The least sequence number to acknowledge next. All packets with a sequence number less than `ack_seq` have been acknowledged at least once.
- `una_seq`: The sequence number of the first sent but not yet acknowledged packet; therefore the sequence number of the packet at the tail of the write queue.
- `trans_announced`: Non-zero if there is a currently running transmission from this socket announced to the receiver, zero otherwise.
- `recv_announced`: Non-zero if the sender has announced a transmission to this socket, zero otherwise.

- `announce_seq`: The sequence number of the last data packet sent carrying the TXS flag; if this packet is already acknowledged, the full congestion window may be used.
- `rcv_mem_pressure`: Non-zero if this socket is currently short of buffer space for the receive queue (and thus under "memory pressure"); zero otherwise.

4.5.2 Global Information

The global information is shared between all ESP sockets, and its storage space is allocated upon loading of the ESP kernel module. Special care has to be taken to either only use atomic operations on these structures or to use the locking mechanisms as provided by the Linux kernel to guard critical sections.

- `esp_recv_count`: The number of sockets which have a incoming transmission running, as detected by having received a TXS flagged packet.
- `esp_ack_queue`: The queue where the acknowledgements for the received data may be parked if necessary to avoid congesting the network.
- `esp_burst_length`: The congestion window used during bulk transfers.
- `esp_initial_ack_burst_length`: The congestion window used while waiting for the first ACK from the receiver.
- `esp_packets_to_ack`: The least number of successfully enqueued data packets needed to trigger the sending of an ACK.

5 Targeting Production Readyness

5.1 Debugging Strategies

Like almost every non-trivial program, the ESP kernel module prone to programming errors. A list of the bugs which were found and could be fixed using either of the presented debugging techniques can be found in the appendix of this document on page 65.

5.1.1 Debugging with KGDB

KGDB [KGD] is a source-level debugger for the Linux kernel. It allows to use the GNU debugger (GDB)[10] to debug the Linux kernel as if it was a regular program, including breakpoints, stepping through the kernel code, watching the contents variables and with support for multithreading. Because suspending kernel code execution for analysis causes any user-space applications to be halted as well, it is indispensable to have two machines in order to use KGDB: One as the testing machine, where the kernel to debug is running and another which is used as the development machine, where the kernel code execution can be monitored by the means of the GDB program.

KGDB is distributed as a kernel patch, which must be applied to the kernel source tree before the kernel is compiled. These modifications add some functionality to the kernel, which is necessary for the debuggung process:

- The GDB stub, which is the heart of the debugger. This is the part that handles requests coming from GDB on the development machine. It has control over all processors in the testing machine when the kernel running on it is inside the debugger.
- Modifications to the kernel fault handlers – instead of doing a kernel panic as outlined in section 5.1.2, these modifications to the fault handlers allow kernel developers to analyze unexpected faults by giving the control over the machine to the GDB stub.
- Communication – there are two versions of this component. They both have the purpose of establishing a connection between the development and the testing machine. One version can use a serial line to connect these two machines, the other can work over ethernet by using UDP/IP frames for message exchange. It is necessary to have a implementation of this functionality seperate from the one the Linux kernel already offers in order to keep the side effects of debugging as small as possible. This component is also responsible for handling control break requests sent by the GDB on the development machine.

In this work, KGDB could be used to track down several bugs which involved connection establishment and shutdown. While having a full-fledged debugger at hand to

[10] In order to be able to analyze dynamically loaded kernel modules, a modified version of GDB has to be used.

```
Unable to handle kernel NULL pointer dereference at virtual
   address 0000008
*pde = 00000000
Oops: 0000
CPU:      0
EIP:      0010:[<c026cb16>]
EFLAGS:   00210213
eax: 00000000    ebx: c6155c6c  ecx: 00000038   edx: 00000000
esi: c672f000    edi: c672f07c  ebp: 00000004   esp: c6155b0c
ds:  0018         es: 0018       ss: 0018
Process netgauge (pid: 2293, stackpage=c6155000)
Stack: c672f000 c672f07c 00000000 00000038 00000060 00000000
       c6d7d2a0 c6c79018 00000001 c6155c6c 00000000 c6d7d2a0
       c017eb4f c6155c6c 00000000 00000098 c017fc44 c672f000
       00000084 00001020 00001000 c7129028 00000038 00000069
Call Trace: [<c017eb4f>] [<c017fc44>] [<c0180115>]
            [<c018a1c8>] [<c017bb3a>] [<c018738f>]
            [<c0177a13>] [<d0871044>] [<c0178274>]
            [<c0142e36>] [<c013c75f>] [<c013c7f8>]
            [<c0108f77>] [<c010002b>]
Code: 8b 40 14 ff d0 89 c2 8b 06 83
      c4 10 01 c2 89 16 8b 83 8c 01
Kernel panic: Aiee, killing interrupt handler!
In interrupt handler - not syncing
```

Figure 10: An example of a Linux kernel Oops message which caused a kernel panic. This information is printed out to the console in the event of a detected error condition inside the kernel.

analyze kernel execution is a very valueable thing in itself, its applicability for debugging a networking protocol implementation is limited. These limitations arise from the fact that it is not possible to synchronize the debugging of two or more machines. Because of this, while one machine is suspended for debugging, the other machine experiences several timeouts and eventually assumes the connection is dead. Additionally some problems did not show up at all when using the kernel versions for which KGDB is available (new versions of KGDB are released for chosen kernel versions only, and with considerable delay). Therefore, another way to debug the ESP kernel module had to be used in addition to KGDB, as described in the next section.

5.1.2 Analyzing Kernel Oops Messages

Kernel Oops messages are a mechanism in the linux kernel which aims for printing out some vital information whenever the kernel encounters an error condition. This

information may be used by a developer to track down and fix the problem. An example for such a message is shown in figure 10. Whenever a Oops occures, the causing kernel component is killed instantly, together with any userspace processes currently doing system calls to this component. This is done without releasing any locks or cleaning up half-modified data structures, so a machince with an Oopsed kernel should be rebootet as soon as possible to avoid further problems, which are to be expected. Additionally, if killing the causing component implies killing a vital part of the kernel like an interrupt handler, the system is halted completely. The information contained within an Oops message is as follows:

- Line 1: An error message briefly decribing what happened. In the example it was attempted to dereference a NULL pointer. The low, but non-zero address is an indication that there was an attempt to read from a member variable of a `struct`, and the address of this `struct` was assumed to be NULL.
- Line 4: The value of a counter which is incremented for each Oops the kernel produces. It is important to observe, that only the first of these message contains reliable information.
- Line 6: The code segment (0010) and the value of the extended instruction pointer (EIP). This unambiguously identifies the faulty instruction.
- Lines 7 - 10: The values of the program status and control register, the general purpose registers and more segment registers.
- Lines 12 - 15: The last values stored on the stack. These are parameters to half-run function calls and return addresses.
- Lines 16 - 20: The call trace. These are the addresses of the entry points of the functions which were executed when the error condition occured.[11]

While this information on its own is not very usefull, additional user-space apllications exist which can be used to track down the cause of the problem. Because the value of the EIP gives the function base address plus the instruction offset, it can be used to identify the kernel function where the crash occured. The "System.map" file belonging to the kernel allows to look up the corresponding symbol (function-) name, along with its base adress. And knowing the base address of the function gives the offset within this function by subtracting it from the EIP. Now it is possible to inspect the true cause of the problem by examining the specific instructions of the failed function. Fortunately, the process described above can be performed by the "ksymoops" user-space program, which takes a kernel Oops message as input and automatically extracts all usable information.

[11] The in-kernel symbol resolver ("kksymoops") has the ability to translate these adresses into function names and offsets within these functions. Unfortunately, it was not found to always give reliable results for the ESP module.

5.2 Creating an Interface for User Settings

The ESP protocol has a view parameters which may be tuned for optimal performance. While there are default values for each of this parameter defined by the means of compile-time constants, it is desireable to give the system administrator a way to modify these parameters without doing a unload module, recompile, load module cycle every time.

Setting kernel parameters always involves passing some data from user-space to kernel-space. The most generic way to do this would be to create a new system call, write a user-space library exhibiting the capabilities of this call and finally to create a application which uses this library to allow retrieving or setting the parameters needed. This is just the way iptables (which is the user-space part) and netfilter (which is the part running in kernel-space) are implemented [NF].

Fortunately, this is not necessary if the preferences to set have a structure as simple as just a few integer values, which is the case for the ESP protocol. To have a consistent interface for accessing such parameters, the "sysctl" interface was introduced with the 4.4BSD version of Unix and ported to Linux as of kernel version 1.3.57.

5.2.1 The Sysctl Interface

The sysctl interface consists of a single function[12] implemented in the standard C library, libc. This function transports the parameter to set from user- to kernel-space and vice versa. This definition of this fuction is:

```
int sysctl(int *name, u_int namelen,
           void *oldp, size_t *oldlenp,
           void *newp, size_t newlen);
```

The first two parameters tell the `sysctl` function, which kernel parameter shall be accessed by the means of an array of integer values and the length of this array. All sysctl parameters are organized in an out-tree, where the nodes and leaves are identified by integer numbers. The array `name` gives a path through this tree; the root node is implicitly given. For the sake of unambiguousness of the `name` parameter, it is obviously neccessary that the children of any node in the tree have associated unique numbers. While this requirement is easy to stick with for the interior nodes of an sub-tree added to this hierarchy, special care has to be taken about the root of this sub-tree as outlined in section 5.2.2.

The next two parameters are for retrieving the old value of the parameter and storing it at the memory pointed to by `oldp`. Finally, the last two parameters are for setting a new value. If one of these operations is not desired, its corresponding parameters may be set to zero. The sysctl interface itself does not make any assumptions about the structure of the data passed between user- and kernel-space, as the data is given by a `void`-typed pointer. The convention about the data transferred is implmented only in the specific kernel module and the calling user-space application.

[12] Under BSD, two more sysctl-related functions exist. These are `sysctlbyname` and `sysctlnametomib` and allow for accessing the sysctl interface via human-readable names instead of an array of intergers.

5.2.2 Adding a Sysctl Interface to ESP

In order to exhibit any internal settings via the sysctl interface, an array of the `struct ctl_table` has to be filled in. Each entry stands for either an interior node, which allows to logically group the parameters; these groups can be used for access permissions as well–or it stands for a leaf node representing an actual value which may be read or set. The most interesting fields to be initialized in the `struct ctl_table` are:

- `ctl_name`, which is the mentioned unique identification number,
- `mode`, the access permissions in classical Unix notation,
- `data`, a pointer the the destination of the supplied data in kernel-space and
- `procname`, a human-readable name of the parameter (why this is supported under Linux despite the absence of the `sysctlbyname()` and `sysctlnametomib()` functions is explained in section 5.2.3), as well as
- `proc_handler` and `strategy`, which are both function pointers

The `proc_handler()` function implements a Linux specific extension of the sysctl interface. Under Linux, the complete tree of sysctl parameters known to the system is mirrored in the directory `sys/` of the procfs [Mou01] virtual filesystem. There, every interior node of the sysctl tree is represented by a directory entry and the leaf nodes are represented by files, which may be read from or written to like any other file in the filesystem. This requires converting the parameters to be accessed from/to a string representation and some handshaking to allow serial access to these strings, as required by the file access API. These tasks are carried out by the `proc_handler()` function.

The `strategy()` function pointer may be set to be zero, which will cause the kernel to call a default implementation of this function. This default implementation will just perform some minimal validity checks based on the size parameters of the `sysctl()` function and copy the data from user-space to where `data` points in kernel-space. This behavior is fine for all parameters of the ESP protocol, with the only exception being the round-trip time.

The round-trip time is special because this value determines a timeout. In the ESP protocol implementation, a timeout is realized by scheduling a timer. This is accomplished by making a call to the `__mod_timer()` or a similar kernel function. All these functions expect the timeout to be given in "jiffies". Jiffies is a variable inside the Linux kernel which keeps increasing forever at a fixed rate as the result of a hardware interrupt. It is the basic packet of time in the Linux kernel, and the rate of this hardware interrupt is given by the kernel's compile-time constant `HZ`. Its value depends on the architecture, and on some architectures it can even be modified during kernel configuration.

In conclusion, it it desireable that the person who wants to set up the ESP protocol on a machine does not have to know about these details. Therefore the time measure for the round trip time was chosen to be μs and is automatically converted upon getting and setting of this parameter by special-cased implementations of the `proc_handler()` and

`strategy()` functions. These functions take the desired timeout value in microseconds and set ESP's internal variable to the next full jiffie, thus rounding up the given value. The method of rounding up was chosen, because having the RRQ timer to kick in a little too late has only minor effect on overall performance as shown in 6.1.1. On the other hand, if an RRQ is sent too early, it will cause a bunch of packets to be transmitted again at no avail, which would degrade performance badly.

The preferred place where the ESP options should show up in the sysctl tree is `CTL_NET/CTL_ESP`. But with the current implementation of the sysctl interface in Linux it is not possible to attach new children to the interior nodes of the sysctl without patching the kernel source, which does not seem to be worth the hassle. Therefore, all ESP options are grouped under the `CTL_ESP` node, which is a direct child of the sysctl root node. The numerical value of the `CTL_ESP` constant is defined in the `af_enet.h` header file and was set to a value that is currently unused by the rest of the Linux kernel. The values reserved for Linux core components can be found in the file `linux/sysctl.h` of the kernel source code.

5.2.3 Using the Sysctl Interface

Under Linux, there are three ways to read and set the parameters exhibited through the ESP sysctl interface:

1. Using the `sysctl()` function call. To use this function call, it is necessary to know the `ctl_name` constants which were used by the kernel module upon registration. These are defined in the `af_enet.h` header file which comes with the ESP kernel module and has to be included by every application which wants to use this protocol.

2. Accessing the files under the `/proc/sys/esp/` virtual file system. This allows for quick testing of parameters by just using console commands like `cat` and `echo`. Having to give these directories and files senseful names is the reason why the `procname` entry in the registration struct is needed.

3. Using the `/sbin/sysctl` program. This program is also capable of reading the parameters to set from a file, which is used by most Linux distributions to set the parameters specified in the `/etc/sysctl.conf` file at boot time.

A full list of all parameters ESP offers through the sysctl interface is shown in table 1, along with a short description of meaning of the individual settings.

Constant	procfs Entry Name	Description
CTL_ESP	esp/	The root node of the sysctl subtree for ESP.
CTL_BURST_LENGTH	burst_length	The window size w used during bulk transfers. The is the number of packets the protocol may have in flight when it knows the TXS was received successfully.
CTL_INITIAL_ACK_BURST_LENGTH	initial_ack_burst_length	The window size when waiting for the first ACK which acknowledges the TXS frame has been received. This resembles the initial window size of the TCP protocol.
CTL_PACKETS_TO_ACK	packets_to_ack	The number of data frames needed to trigger the sending of an ACK. The detection of packet loss and the receipt of an TXS cause the immediate sending of an ACK, independent of this setting.
CTL_SEND_BUFF	send_buff_size	The size of the send buffer to be used. Only affects sockets allocated after setting a new value.
CTL_RECV_BUFF	recv_buff_size	The size of the receive buffer to be used. Only affects sockets allocated after setting a new value.
CTL_ROUND_TRIP_TIME	round_trip_time	The round trip time assumed for the connection. This value is to be given in μs and is automatically rounded up to the next full jiffie[a].

Table 1: Parameters of the ESP protocol exhibited through the sysctl interface.

[a] See 5.2.2 for an detailed explanation.

6 Analysis and Evaluation

6.1 Comparison of the proposed Flow Control Scheme to TCP

For the calculations here an 10 Gbps link, 1500 byte packets and a round-trip time of $R = 10\,\text{ms}$ are assumed. To saturate such a link, a sending rate of $T_{\text{SAT}} \approx 8.3$ packets per round trip time (PPR) is required. The probability of a packet being lost through packet corruption is assumed to be $p = 5 \cdot 10^{-4}$, which is a real-world value metered by letting the BSD *ping* command run with MSS-sized packets overnight on an otherwise unused link.

6.1.1 Handling of Packet Loss

An important aspect of ESP concerning the handling of packet loss the probability that the RRQ timer has to kick in to recover. Because in a reasonable fast ethernet with latency values of a few microseconds, waiting for a timer which can be scheduled with a granularity in the order of 10 ms implies a big penalty. Therefore it is desireable that packet loss is detected by the interrupt-triggered packet receiving function of the protocol, instead of waiting for a timer which would take magnitudes longer.

As stated before, the RRQ timer is necessary only to handle the cases where a consecutive number of frames from the end of the transmission is lost. Assume p is the probability for a frame to get lost and w is the window size used, then the probability for the RRQ timer having to kick in is

$$\begin{aligned} p_{\text{rrq}}(p,w) &= \sum_{i=0}^{w-1}(1-p)^i \cdot p^{w-i} \\ &= p^w \sum_{i=0}^{w-1}\left(\frac{1-p}{p}\right)^i \\ &= p^w \frac{\left(\frac{1-p}{p}\right)^w - 1}{\frac{1-p}{p} - 1} \\ &= \frac{p(1-p)^w - p^{w+1}}{1-2p} \end{aligned}$$

Figure 11 shows three plots of p_{rrq} for different values of p. As can be seen, this function has the friendly behavior of converging to 0 the faster, the greater the value of p is.

6.1.2 Ramp up Times

The ramp up time means the time it takes for the protocol to reach a state where the congestion window w is large enough to allow for saturating the link. Because of TCP's nature to double the size of the congestion window during slow-start for every ACK

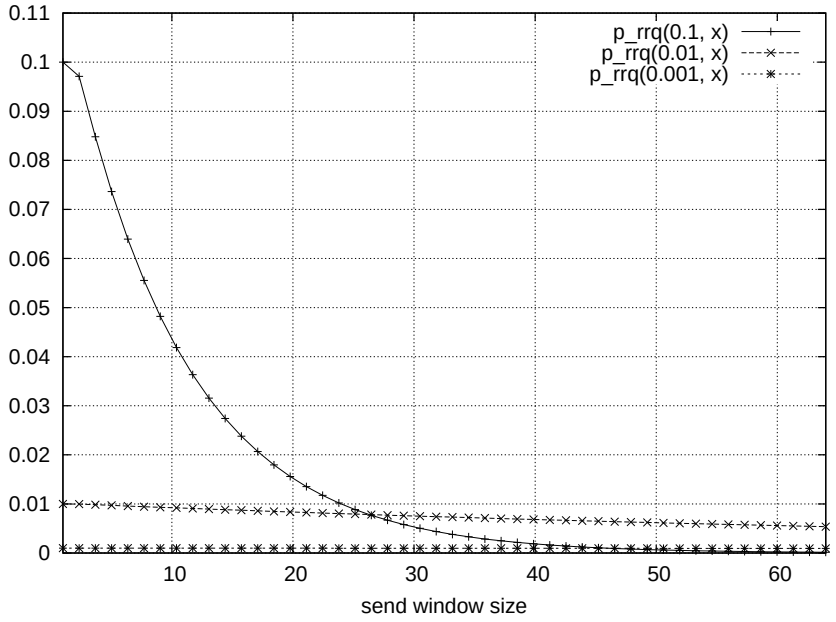


Figure 11: The Probability the RRQ timer is needed to recover from packet loss as a function of the window size (which is x here) and the probability of a single packet being lost.

received[13], effectivly showing an exponential growth rate and making the term "slow start" quite misleading, it takes

$$t(T) = R \log_2(T)$$

seconds to reach a sending rate of T packets per round-trip time. Assuming a RTT of $R = 10\,\text{ms}$, this gives about $R \times \lceil \log_2(8.3) \rceil \approx 0.04\,\text{s}$ for TCP/IP to saturate the link.

Because the proposed flow control scheme reaches the full sending rate immediately after receiving the first ACK, it takes only one round-trip time to saturate the link, which is $R = 0.01$ seconds. This looks quite promising, but is in no way to be confused with throughput. This is just the time it takes for the congestion window to reach the size T_{sat} – just enough to saturate the link. The actual transmission time is not taken into consideration here.

6.1.3 Fairness Considerations

Because it would be unadequate to assume the complete infrastructure[14] of a cluster installation to be reworked in order to use ESP as the transport protocol, it is important for ESP to cooperate with other flow control schemes which share the same link.

[13]This is true for all TCP variants presented in section 3.1.

[14]This includes the network file system, shell access, health monitoring, etc...

In this context, "fairness" is equivalent to the occupied buffer space on the switch's per-port buffer of an overcharged host. This is so because the switch will work off this buffer in a first-in first-out manner, at the rate which the link between this buffer and the destination host permits. When this happens, the ratio of packets a protocol has stored in this buffer equals the ratio of packets forwarded to the destination host, and thus gives the share of bandwidth the specific protocol engrosses [MAW98].

Let S_{Buff} be the size of the buffer, then an estimation on the amount of buffer space (measured in MTU-sized packets) occupied by the ESP protocol is given by

$$b_{\text{esp}} \approx k, \;\; k \leq \frac{w}{2}$$

where w is the window size used by ESP. Because of ESP's nature to share the congestion window among all active transmissions, this holds true even in the presence of multiple senders. Knowing this, and observing TCP Reno's behavior of increasing the congestion window until packet loss occurs, the number of packets it has on queue ranges within $[0, \;\frac{S_{\text{Buff}}}{\text{MTU}} - k]$. Assuming the true number of packets TCP has on queue is uniformly distributed on the interval $[0, \;\frac{S_{\text{Buff}}}{\text{MTU}} - k]$, an approximation of the queue size for TCP is given by

$$b_{\text{tcp}} \approx \frac{S_{\text{Buff}}}{2 \cdot \text{MTU}} - \frac{k}{2}$$

Using real-world numbers (a MTU of 1500 bytes, a buffer size of $S_{\text{Buff}} = 128\,\text{kB}$ and ESP's congestion window $w = 10$) gives $b_{\text{esp}} \approx 5$ and $b_{\text{tcp}} \approx 45$. This means TCP traffic will take 90 % of the bandwidth while ESP contents with 10 %. This is so because of Reno's aggressive congestion avoidance and can be overcome by switching to a TCP with a more sophisticated (and less aggressive) congestion avoidance, like TCP Vegas is.

Vegas is much more friendly in terms of buffer occupation for a saturated link, as its strategy is to have at least α and at most β packets stored on the bottleneck's buffer [BMP94], where α and β are empirically chosen values of 2 and 4, respectively. Because of this, the buffer utilization of Vegas is given by

$$b_{\text{vegas}} \approx i, \;\; i \leq \beta$$

Using the same numbers as above this leads to ESP getting 56 % of the bandwidth and Vegas using 44 %, which seems *much* more reasonable. Because of this and Vegas' other advantages as presented in section 3.1.3, it would be a good choice for a cluster environment, especially if ESP is to be used there as well.

6.2 Searching the ESP Parameter Space

The basic idea when designing the flow control scheme for ESP was to dispense it from metering values which are known a priori in a cluster environment. Nevertheless, in order to give the flow control a chance to show its potential, good values for the window size w and the acknowledgement delay r_{ACK} have to be found.

To find such a good set $(w, \;r_{\text{ACK}})$ of parameters, another communication pattern (called "esp-params") for the Netgauge benchmark tool was developed. This pattern

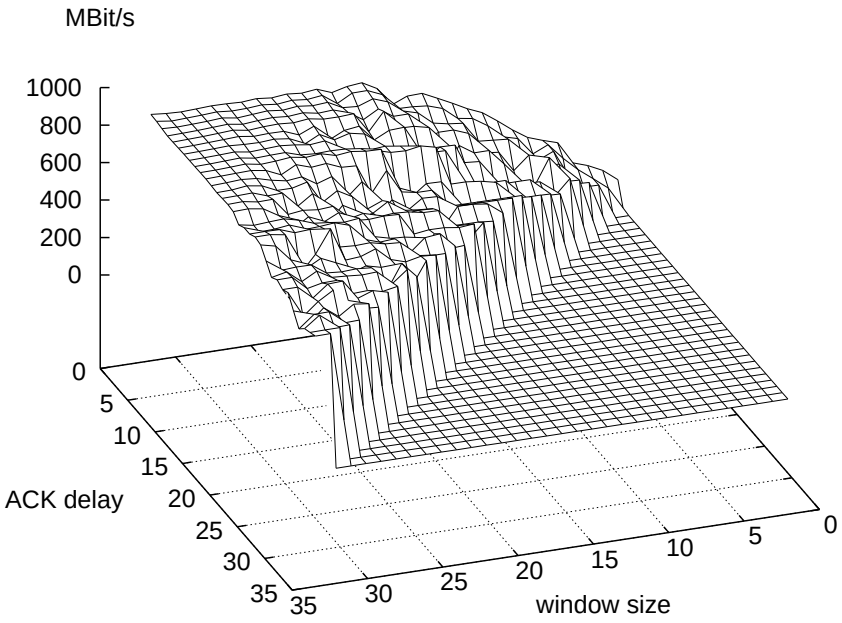


Figure 12: Plot of the ESP parameter space. This figure shows how the achieved throughput for the ESP protocol depends on the windows size and the delay which is set on the emission of acknowledgements.

mainly consists of two nested loops which enumerate the values for w and r_{ACK} within a given range. These parameters are set on the kernel module using the sysctl interface as explained in section 5.2.1. To keep the running time of this benchmark acceptable, the points in parameter space with $w < r_{\mathrm{ACK}}$ are omitted, as there any progress in data transmission would rely on the retransmission request timeout, resulting in a very bad utilization of the bandwidth. For the remaining points, several test transmissions are done, of which the peak throughput is returned.

Figure 12 shows a plot of the metered throughput, with w and r_{ACK} both ranging in $[1,\ 32]$. As can be seen from this plot, the points with $w \geq 16 \wedge r_{\mathrm{ACK}} \leq w - 8$ form a plateau where the link is saturated. Any of these parameter pairs may be chosen to achieve good performance, but perferring smaller window sizes seems advisable to allow the flow control to adapt faster to changes with bandwidth limiting conditions.

The least window size resulting in a saturated link was found to be 16, which gives $T_{\mathrm{SAT}} = 16$ as well. Equation (3) on page 22 holds true for all items measured–because of this, it seems possible to reduce the running time of search process to be $\mathcal{O}(n)$ instead of $\mathcal{O}(n^2)$, with n being the count of window sizes to be tested. But since this search process is needed only upon initial deployment of the ESP protocol, and its completion time is under 5 minutes[15] already, this approach was not followed. Additionally, in the current form the benchmark has the ability to expose anomalies, either with the network

[15]When running on a reasonable well working gigabit ethernet link.

or with the protocol itself, which might be missed if this optimization was applied.

6.3 Assessing the Overhead

In general, it can be said that the overhead of a network protocol is the load that it causes on a communicating host in order to transact a message transmission. While this is far from an exact definition, it gives an idea of what is tried to assess here. The induced load has many aspects, amongst which some are:

- The CPU time directly consumed by the protocol to set up a communication and prepare the message for transmission.
- The utilization of the bus system between the main memory, CPU and the networking interface.
- The utlilization of main memory bandwidth.
- The trashing of the CPU's caches which cause expensive cache misses for subsequent computations.
- The cost of task switches and user-kernel-space transitions.

Considering the complexity of current computers and their communication infrastructure, this list could be extended easily, hardly ever coming to an end. Because of this, a *manageable* model of parallel computation, which includes the overhead, has to introduce drastic simplifications. Several models of parallel computation have been developed in the past [CKP+93, AISS95, KBG00].

6.3.1 The LogGP model

The LogGP model as developed by [AISS95] is aimed at estimating the running time of parallel algorithms accurately. This is accomplished by decomposing the particular nature of a communication into five parameters L, o, g, G and P, which are assumed to be constant for one run of an algorithm. The meaning of the particular parameters is depicted in figure 13. It extends the LogP model [CKP+93] so it can represent the transmission of large messages by introducing the new parameter G, which is the minimum gap between the receipt of two symbols[16] which are part of a single, long mesaage.

The LogGP model could already show its usefulness in several applications; for example the proof of optimality for a solution to the scatter-problem in the original LogGP paper [AISS95] or the estimation of the running time of several non-trivial algoritms [SSV99, Bri02].

Because LogGP tries to be very generic in order to be applicable to many different kinds of networks. Hence, the complexity of a data transmission has to be mapped to only a few parameters. This mapping is not trivial, and care has to be taken not to

[16] LogGP does not imply any particular size of a symbol. For measurements, coosing a symbol to be a single byte seems reasonable.

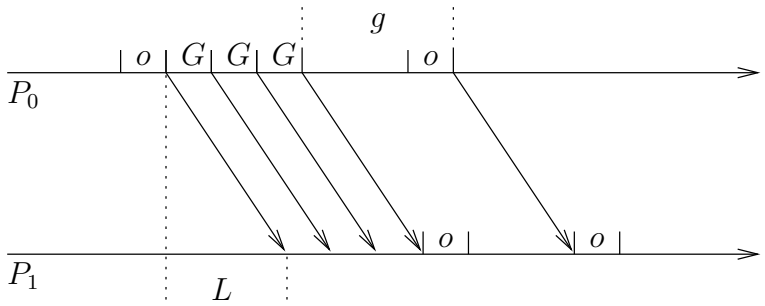


Figure 13: Depiction of the transmission of two messages from P_0 to P_1 as seen by the LogGP model. The first message consists of 4 symbols and has a overall transmission time of $2o + L + 3G$. The second message consists of a single symbol which is sent out with a delay of g after the last symbol of the first message was sent; it's transmission time is $2o + L$. For short (i.e. single-symbol) messages, LogGP is equal to the LogP [CKP+93] model.

object the parameter's definition. Some aspects of data transmission are not covered by the LogGP model at all, for example the addidional CPU overhead which is caused by many protocols for handling acknowledgements. These acknowledgements exchanged during message transmission, and their processing obviously causes some overhead on the communicating hosts – still, LogGP assumes all overhead to be aggregated at the beginning of message transmission (on the sending host) and the end of message transmission (for the receiving host). The transmission of a single message as posed by the LogGP model can be subdivided into the following steps:

1. The process willing to send a message calls relevant `sendmsg` function. This is the point in time from where on the following computations are accounted to the first o of a message transmission as defined by LogGP. In many cases, the `sendmsg` function will have a gesture similar to the one in figure 15.

2. At some point in time, the CPU will finish it's work on the first part of the message and hand it out to networking infrastructure. From now on, the delivery of this part of the message is in the network's hands and the LogGP model states that this is the point where the first o ends and L begins. The characteristic of this event is very dependent on the network used. It might be when the first packet is posted to the network interface card's DMA buffer for transmission on a modern ethernet, or when a micro-controller has sent out the first byte over I^2C for embedded systems. It is worth to note that there is no indication that this event coincides with the return from the `sendmsg` function. In fact, in effort to minimize the end-to-end latency of the network, most implementations will interleave the preparation of packets and their transmission.

3. The CPU has finished preparing the message for transmission. Now the `sendmsg`

function returns and the calling process can go on with it's computations. While this event is not relevant in the LogGP model, its still of interest for an attempt to assess the model's parameters for an existing network, because it is the first event that can be detected by a sending process after it's call to the `sendmsg` function.

4. The first part of the message arrives at the destination host. Here is where L ends, and the remainder of the message will arrive at a rate of G^{-1} symbols per second. While this event could be detected by the networking subsystem, it is not visible to the sending process.[17]

5. Finally, the full message arrived at the destination host. It still has to be prepared to be accessible for the receiving application, which is accounted to another o by the LogGP model. This final processing of the message by the networking subsystem is finished, when the `recvmsg` function returns.

The only parameter of the LogGP model which has not been used in the above walk-through is g. This is because it becomes apparent only when multiple messages are sent. This parameter defines the minimum spacing of two successive data transmissions and overlaps with o. By definition, $o \leq g$.

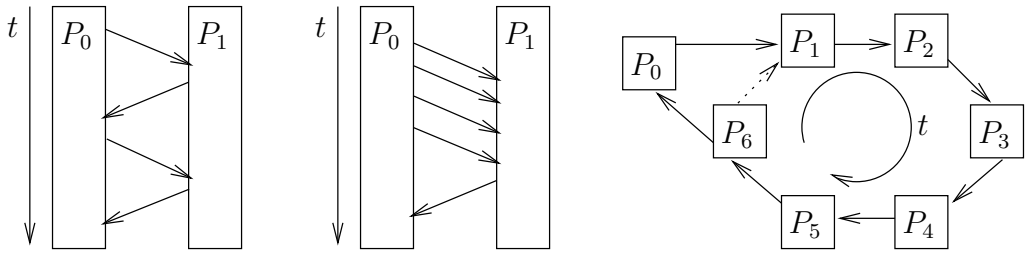
6.3.2 Existing Approaches to assess LogGP Parameters

The LogGP model can be used on its own to compare algorithms or to give upper bounds on the execution time and other theoretical applications. For these purposes, it sufficient to assume the models parameters to be known and to use them as constants; but for actually predicting the running time of an algorithm on a given infrastructure, concrete values for the model's parameters are needed.

Several strategies to assess these have been developed in the past, whereat the scheme of operation for most of them can be classified into two groups, which are shown both on the left-hand side of figure 14. One common aspect of all schemes is the need to eliminate caching effects, which is accomplished by requiring a full round-trip of the data that is sent out for measurements.

Another aspect these measurement methods have in common is that they use a modified version of the LogGP model. These modifications have two purposes: Foremost, it allows for finding a mapping from the measured times to the model at all. For most architectures, the model's parameters can not be measured directly. Therefore, the basic idea is to send several messages through the communication infrastructure in a carefully chosen way and to record the completion times for these data transmission. In a second step, the data transmissions done are expressed in the terms of the model, which gives the right-hand sides of an equation system; and the measured times give the left-hand sides of these equations. Finally, the equation system is solved for the model's parameters. In other words, a mapping is needed to express the measurements in terms of the model in a way which allows to finally find the sought-after parameters.

[17] Depending on message size and network latency it's possible for this event to happen before the event above, e.g. the first part of the message is received before the last part is sent out.



(a) Ping-Pong (b) Ping-Ping (c) Ring

Figure 14: Different strategies to assess the LogGP parameters. The actual time measurements are always done by a single peer, which is P_0 in these figures.

Secondly, the derived model in most cases allows to express the characteristics of the measured network(s) more accurately. For example, in [BBC+03, CLMY95] there is a differentiation between the send overhead o_s and the receive overhead o_r, which are assumed to be equal by the original model. The parameterized LogP (P-LogP) model as developed by [KBG00] introduces more drastic changes. It turns the originally constant values of g, o_s and o_r into functions of the message size s.

An attempt to assess the parameters of the LogGP model was done by [HLR07]. Therein, a sophisticated scheme of operation is used to expose the parameters of the model exactly. The scheme used is called the parameterized round-trip time (PRTT), which is an derivate of the ping-ping scheme as shown in figure 14(b). It allows to assess the parameters without causing high network load by introducing an artificial delay between the messages sent. Though the measurement is tailored for assessing the model's parameters exactly as defined in the original presentation, the overhead o was found to be $\mathcal{O}(s)$ (with s being the size of the message), which objects the definition of o that LogGP gives.

The reason for this behavior can be uncovered by having a look at a typical sendmsg handler function. This function is part of the networking protocol used and it is the point where the responsibility for the data to transmit is passed from the application to the underlying protocol. The basic task this function has to perform is the same for most packet-oriented network protocols, and its pseudo code is shown in figure 15.

When trying to measure the send overhead as defined by the LogGP model, the exact point where the time has to be taken is hidden[18] somewhere in line 7. The inner workings of this function are hidden in kernel-space and thus are invisible to an user-space application. Because of this, without instrumentalizing the sendmsg handler, it is only possible to measure the time somewhere after line 23. This gives an indispensable measurement error if not handeled properly when designing the benchmark used to assess the model's parameters. The additional time measured will be called x in the following;

[18] Using non-blocking I/O does not help here either, because the loop beginning in line 3 would be executed several times anyway. To minimize the measurement error x, s should be chosen as small as possible.

```
int sendmsg(void* data, int len, int shouldblock) {
  int retval = 0;
  while (1)
  {
    <get MSS bytes off the data buffer>
    <equip this packet with header information>
    <send it out>

    retval += <size of packet>

    if (<all data sent>) goto out;
    if (!<send buffer is full>) continue;
    else if (shouldblock) {
      <let the calling process sleep until
       send buffer space becomes free>
    } else {
      goto out;
    }
  }

out:
  <free occupied resources and locks>
  return retval;
}
```

Figure 15: Pseudo code of a much simplified sendmsg handler function.

obviously, $x > 0$ (at *least* returning from the sendmsg handler to the calling function). A inconsistency between measured times and expected behavior was already discovered in [BBC+03].

This error overlaps with g and it is not known in advance which of them is greater. Here, some care has to be taken to get things right: Because LogGP defines $o \leq g$, flooding the network reveals the parameter g, which is the reciprocal of the rate at which small (e.g. 1 byte) messages can be pushed into the network until saturation is reached. But it is not known if the limiting factor of the send-rate is the overhead o or some component of the networking infrastructure between the communication hosts. The case where $o = g$ would mean that the CPU is the bottleneck in the communication path, which can be seen as an indication for a configuration error or a bad choice of components. Because of this, for most cases it can be assumed that $o < g$. To meet this observation, $\mathcal{X} = \max\{x, g - o\}$ is introduced, which stands for the time the `sendmsg` handler returns *after* the overhead (in the sense of the LogGP model) has ended.

Every scheme for assessing the parameters L, o, g can be seen as a path of a message through one or more processes P_i, where each of these processes can do a few operations and/or measure the time. Because in a typical cluster environment there is no common time source of sufficient accuracy, all time measurements have to be carried out by a single process P_0. Every time measurement done can be seen as

$$t = \sum_{i=0}^{m} \mathrm{OP}^{n_i}, \;\; n_i \in \{1, 2, 3\},$$

where m is the number of operations performed and

$$\mathrm{OP}^k = \begin{cases} o + \mathcal{X}, & k = 1 \text{ (the cost of a single send operation)}^{19}, \\ 2o + L, & k = 2 \text{ (the cost of sending, transmitting and receiving a message)}, \\ d, & k = 3 \text{ (waiting some period of time)}. \end{cases}$$

6.3.3 Assessing the overhead with a linear system of equations

The original aim of this work was to assess the overhead o as defined by the LogGP model. Previous attempts to measure the LogGP parameters either used a modified version of the model or had to live with contradictions between the measurement results and the model's definition. Because of this, there is still need for a scheme of operation which exposes the characteristics of a given network in terms of concrete values for the LogGP model's parameters. In order to find suche a scheme, a novel communication pattern (in the context of LogGP parameter measurement) was examined. This scheme sends one or more messages through a ring-like topology, as shown in figure 14(c).

Several variations of the original scheme were examined. One of them is shown in figure 16 and shall be explained here in a little more detail, to show the difficulties of

[19] Here is the key difference to previous attempts to assess LogGP parameters: to respect that the `sendmsg()` handler returns some time x *after* the the first symbol of the message was sent out.

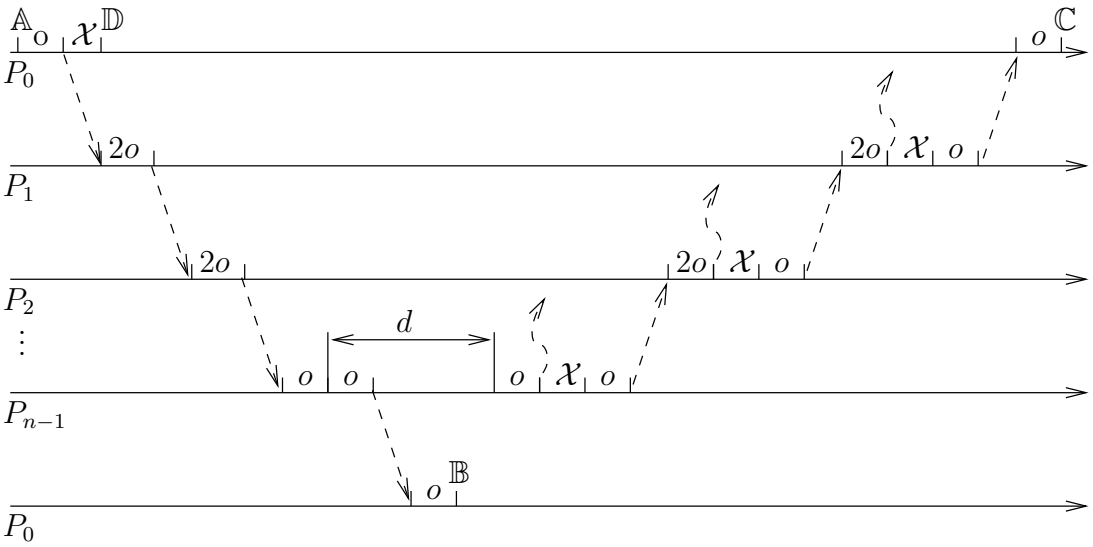


Figure 16: Schematic diagramm of an attempt to assess the overhead o as defined by the LogGP model. Observe the first and the last horizontal line *both* represent P_0. The curved arrows indicate messages which are sent out only to introduce additional overhead, their receiver is not shown for clarity. The term "$2o$" always stands for two consecutive o's which were caused by a receive and an immediately following send operation. $\mathbb{A} \ldots \mathbb{D}$ are the events where the time is measured.

this approach. As mentioned above, due to the lack of a common time source, all time measurements are performed by P_0. The first time this process minutes is the starting time $\mathbb{A}$ of the benchmark, which is immediately followed by a sending operation, which passes a message to P_1. After this send operation completes, the time $\mathbb{D}$ is taken. P_1 on its part forwards the message to P_2 and so forth, until it eventually arrives at P_{n-1}. This process duplicates the received message. The first copy is forwarded to P_0, which takes the time $\mathbb{B}$ when the receipt of this message is finished. The second copy of this message is sent back to P_{n-2} after some delay d, which must be chosen to be larger than $o + \mathcal{X}$. Subsequently, this message is forwarded backwards through the ring, until it arrives with P_0 at time $\mathbb{C}$. To make the measurement more robust against temporal fluctiations when assessing $\mathcal{X}$, this backward directed round through the ring is delayed by sending additional messages between the receipt and forwarding of the original message.

None of the times measured exhibits any of LogGP's parameters directly[20]. Because of this, it is necessary to set up a system of equations which can potentially be solved for the unknown values. For the example shown in figure 16, these equations are as follows:

$$\begin{aligned}
t_{\mathbb{A}\to\mathbb{B}} &= 2no + nL \\
t_{\mathbb{A}\to\mathbb{C}} &= 5(n-1)o + 2(n-1)L + (n-1)\mathcal{X} + d \\
t_{\mathbb{A}\to\mathbb{D}} &= o + \mathcal{X} \\
&\iff \\
t_{\mathbb{A}\to\mathbb{B}} &= 2no + nL && (4)\\
t_{\mathbb{A}\to\mathbb{C}} - (n-1)\cdot t_{\mathbb{A}\to\mathbb{D}} - d &= 4(n-1)o + 2(n-1)L && (5)\\
t_{\mathbb{A}\to\mathbb{D}} &= o + \mathcal{X}
\end{aligned}$$

Obviously, the right-hand sides of equations (4) and (5) are lineary dependent, which forbids to find a unique solution for the system. Similar problems occured with all attempts to assess the overhead o (along with the model's other parameters), and by reasoning the universally valid from special cases, this leads to:

Theorem. *It is impossible to find a general scheme to indirectly assess the overhead o as defined by the Log(G)P models.*

Proof. The resulting equations representing the measured times t will *always* be of the form

$$t = \underbrace{i \cdot (2o + L)}_{\text{send and receive}} + \underbrace{j \cdot (o + \mathcal{X})}_{\text{send}} + \underbrace{d}_{\text{wait}}, \quad i, j \geq 0, \quad d = \text{const.}$$

Remember, d is already known. Therefore, this can be rewritten to

$$\tilde{t} = o \cdot (2i + j) + L \cdot i + \mathcal{X} \cdot j \ ,$$

[20] An exception is the parameter G, the inter-symbol gap, which is the reciprocal of the bandwidth. It can be measured trivially by transmitting a sufficient large message, so that the influence of the other parameters becomes negligible.

and brought into matrix form:

$$\begin{pmatrix} 2i+j & i & j \end{pmatrix} \cdot \begin{pmatrix} o \\ L \\ \mathcal{X} \end{pmatrix} = \begin{pmatrix} \tilde{t} \end{pmatrix} \tag{6}$$

Being an underdetermined system, the matrix vector equation (6), although being solvable for every right-hand side $\tilde{t}$, does not have a unique solution (o, L, $\mathcal{X}$) for any given data i, j and $\tilde{t}$. □

The direct implication of this result is, that a general user-space application is not capable of measuring the LogGP parameters o and L, and thus can not predict its running time accurately using this model. But this stand-alone prediction of the communication and computation times is a main motivation for using models of parallel computation, as it allows to develop algorithms which adapt themselfs so they can utilize the communication performance of a given infrastructure optimally.

Still, there are interconnection techniques where the measurement error when ignoring the time x will be negligible small. A prominent example for such a network is Infiniband when using remote DMA (RDMA) with already pinned[21] memory. In this case, the equivalent of the `sendmsg()` handler function will only instruct the host channel adapter (HCA) to perform the data transport, which is a constant time operation [Pfi01].

For special purposes it is indeed possible to modify the underlying network protocol or hardware driver to provide the needed times, but this approach does not seem very elegant, neither will the effort be justified for many applications, as this approach would result in a very expensive deployment. Because of these shortcomings of the LogGP model, another model of parallel computation was examined, as presented in the next section.

6.3.4 The parameterized LogP model

Because the LogGP model turned out not to give a usable definition to assess the overhead of a network protocol, another approach had to be chosen, which is the parameterized LogP (P-LogP) model. The parameterized LogP [KBG00] model is derivate of the LogP/LogGP models, with the main differences to the LogGP model being:

- Instead of being constant, the gap g, the send overhead o_s and the receive overhead o_r are functions of the message size s.
- As a result of this, the parameter G is superfluous. The P-LogP parameter $g(s)$ can be understood as the minimum gap between two messages of size s. This implicitly gives the per-symbol gap G to be $\frac{g(s)}{s}$ (assuming the symbol size to be a single byte).

[21] The memory must be pinned in advance, as the process of pinning itself is also $\mathcal{O}(s)$.

- The latency L begins contemporary with the send overhead o_s; their times overlap. Because of this redefinition, L can be seen as the time it takes from issuing the message transmission until the first bit of the message arrives at its destination.

The first two changes mainly make this model more versatile when expressing a specific network's characteristics. But the last change is the key for being able to assess all of the model's parameters using user-space applications, as this makes it possible to create a scheme of parameter assessment, without having to instrumentalize the kernel.

6.3.5 Assessing the P-LogP parameters

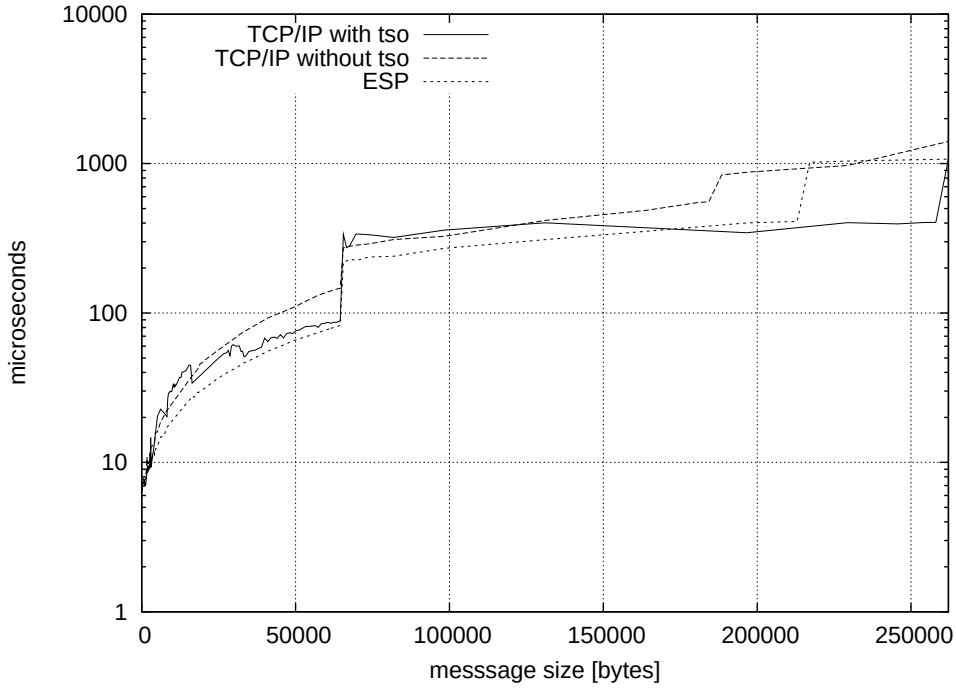
A benchmark which assesses the parameters of the P-LogP model was developed by [KBV00]. This benchmark uses a variation of the ping-ping scheme and is implemented as a parallel program which uses the MPI for communication. The benchmark was executed for the ESP protocol, for TCP with TSO enabled and for TCP with TSO disabled, and The P-LogP parameters as assessed by this benchmark are shown in figure 17. As can be seen there, the ESP outperforms both variants of TCP up to a message size of 175 kB, where TCP with TSO enabled gains the lead. The latency L was found to be 6.30 microseconds for ESP and for TCP as well. The measurements of the throughput (that gives the g parameter) is presented in section refthrough.

Because this benchmark always includes the the overhead of the message passing library, a simpler benchmark only assessing the send- and receive overhead was developed as a communication pattern for Netgauge. Its measurement results are shown in figure 18.

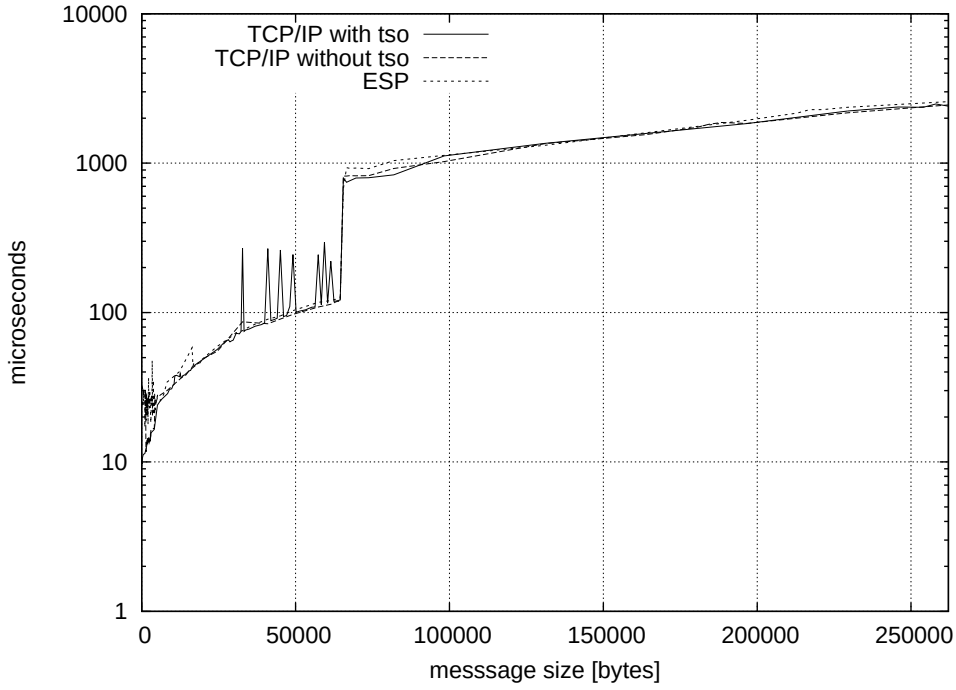
6.4 General Overhead Assessment

The P-LogP model, by representing the send- and receive overhead as a function of the message size, gives a sound base for comparing the characteristics of the ESP and the TCP procotols, as well as many other protocols. Still, it can not (and does not try to) cover every aspect that can be seen as the "overhead" a networking protocol causes to a communicating host. Most notably, it assumes the send overhead to be aggregated at the beginning of message transmission, just as the LogP model does. In fact, most of the overhead (in terms of CPU cycles occupied by the networking protocol) is caused at the beginning of message transmission, when the data is copied over from userspace, checksummed and packed into socket buffers. But, as an example, an implementation of the TCP/IP protocol is free to do the checksumming in a lazy fashion – immediately before the packet is sent out, not when it enters the send queue. In this case, depending on the size of the send buffer, the majority of the checksumming might be performed when the `sendmsg` handler function has already returned to the application. So this important part of the overhead TCP/IP causes will be invisible to the user-space application.[22]

[22] Many modern network controllers have the capability to do TCP segmentation offloading (TSO), which includes the checksumming. In this case, the CPU does not have to perform the checksumming at all.

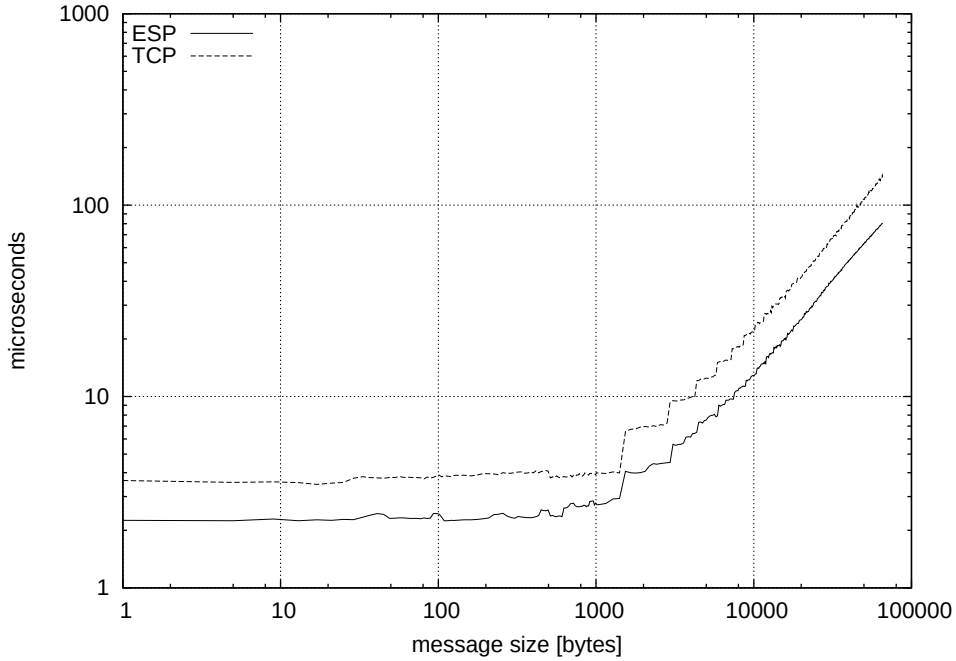


(a) send overhead

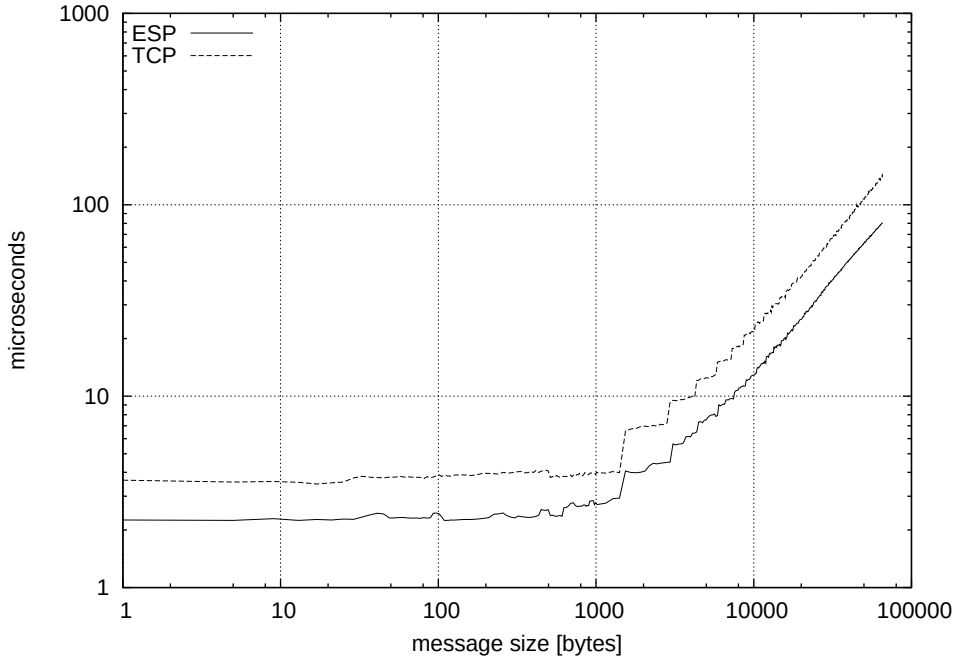


(b) receive overhead

Figure 17: Average measurement results for the overhead as defined by the P-LogP model, when using the Open MPI library as an intermediate layer.



(a) Comparison of the send overhead



(b) Comparison of the receive overhead

Figure 18: Minimum measurement results for the overhead as defined by the P-LogP model, when accessing the protocol using the socket interface. Please observe these plots use a logarithmic scaling on the y axis, so a constant difference between the opponents indeed signifies a lineary increasing edge for the ESP protocol.

A way to assess this "hidden" overhead, which occures completely in kernel-space, is to execute a computation and a data transmission concurrently. For a parallel algorithm, it is always a prime design goal to overlap computation with data transmission. Though, there is no parallel algorithm (with a practical use) known to the author which manages both: to constantly transmit data (thus to saturate the network) *and* to constantly utilize the CPU (by never having to wait for a message exchange to finish).

To simulate this behavior, another approach was chosen: A small comand-line utility was developed, which connects an ESP socket to the default file handles stdin and stdout, which the operating system opens automatically opens for every program started. The idea was to let this program read from `/dev/zero` on one host, forward the data over the ESP socket pair to another host, where the data is drained to `/dev/null`. While this could generate as much traffic as needed to saturate the link, the CPU would not be employed very much (besides processing the network protocol overhead). So the spare CPU time could be metered using any CPU benchmark desired, and comparing the performance to a comparative run of the benchmark when no data transmission is running in the background would reveal the true overhead the network transmission causes. Unfortunately, this approach gave no usable results, what is believed to be due to the extremely bad cache coherence of this scheme of operation–the data being transmitted has nothing to do with the data being computed, so every task switch between these two processes performed by the Linux kernel trashes the contents of the CPU caches.

6.5 Throughput Comparison

For all measurements presented here, the ESP module was set to use a congestion window of $w = 21$ packets and an acknowledgement delay of $r_{\mathrm{ACK}} = 10$ packets. These values promised a good performance and where found using the tool presented in section 6.2.

6.5.1 Single Sender

For the single sender throughput measurements, the one-one pattern of the Netgauge benchmark tool was used. The for TCP, the tests were executed two times, once with TCP segmentation offloading enabled and once without. As can be seen from figure 19, with TSO enabled, TCP can almost keep up with the ESP protocol, reaching a peak throughput of 833 MBit/s for a message size of 128 kB. Without TSO, the peak throughput of 756 MBit/s is reached for a message size of 65 kB. ESP reaches its best performance for a message size of 256 kB with 838 MBit/s.

6.5.2 Congestion Avoidance Stress Test

The "one-many" communication pattern for the Netgauge benchmarking tool, as developed in section 2.4.4 was used to generate the following numbers. The plot in figure 20

Still, this example shows how an implementation detail of the network protocol can greatly influence the overhead as perceived by an user-space application.

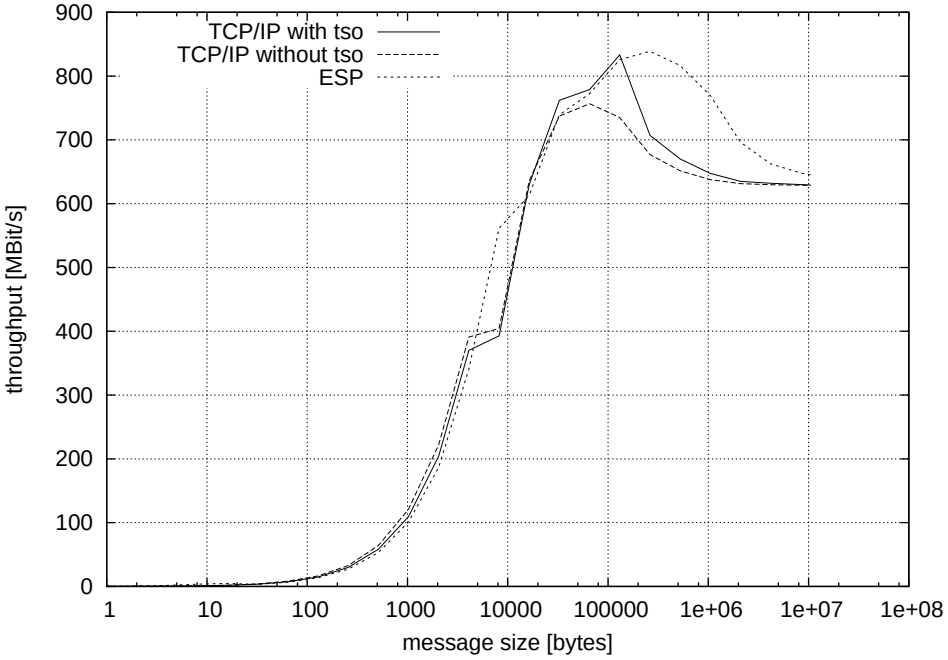


Figure 19: These plots show the achieved peak throughput on cluster 2.

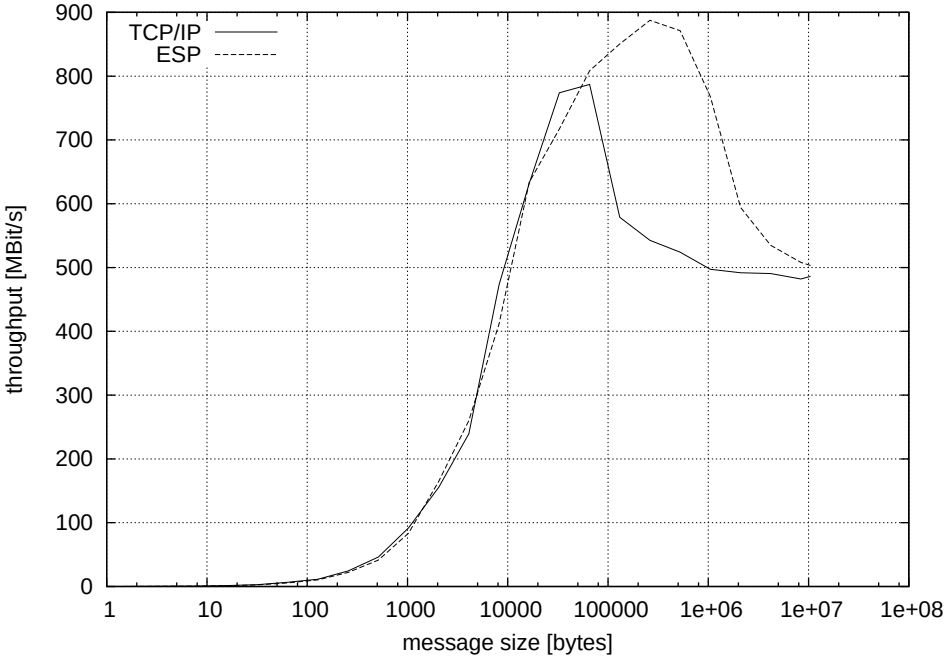


Figure 20: These plots show the achieved peak throughputs for a network congestion situation on cluster 2.

shows the median throughput of 128 runs, for a situation where two hosts send messages of increasing size to a common third host, with the throughput of both sending hosts summed–so this gives the total speed at which the messages are coming in at the receiving host. The TSO feature of the TCP/IP protocol was enabled for this test; with TSO disabled, the TCP/IP numbers go down by about 5 % for medium message sizes, as already outlined for the single-sender case in 6.5.1.

TCP/IP reaches its peak performance of 787 MBit/s for a message size of 65 kB, and backs up very rapidly beyond this point. Because of the switch's per-port buffer size of 128 kB, TCP's shortcomings with the congestion avoidance do not become apparent up to the point where this buffer can not hold the odd data for the destination host. But when the totalized message size of both sending hosts exceeds this barrier, TCP looses over 200 MBit/s, nor is it able to recover for even larger messages, converging to about 500 MBit/s for very large messages.

The congestion avoidance developed for the ESP protocol works quite well, and it even outperforms its own numbers from the single-sender case, reaching a peak performance of 887 MBit/s for a message size of 256 kB, about 13 % more than TCP/IP.

6.6 Application Benchmarks

6.6.1 Optimizing for real-world Applications

Though the low-level benchmarks looked very promising, the first runs of real-world applications gave results which were a little disappointing. When examining this issue, it was found that ESP performed rather bad compared to TCP when the link was flooded with many small messages. For ESP, this was a worst case scanario: The application calls the `sendmsg()` handler and instructs it to send out a single byte message. The handler function on its part will allocate a socket buffer and copy the message over there. Finally, the socket buffer will be enqueued onto the write queue for transmission. If this procedure is repeated rapidly, the send queue will fill up with many packets, each carrying a single byte, which is very inefficient and causes severe send overhead.

The internals of the Linux socket buffer allocation, as outlined in [Rei06] offer a way too loosen this situation: When allocating a socket buffer of a certain size, chances are good that indeed aslightly larger socket buffer is returned by the allocation function. It is possible if this was the case using the `skb_headroom()` function, which returns the storage space this socket buffer has left. So, instead of always allocating a new socket buffer, now a test is made to see if (at least a part of) the current message may be appended to the last socket buffer on the write queue, which reduced the send overhead for small message floods significantly as can be seen in figure 21.

The idea for this optimization already came up earlier during development, but back then it was not implemented on purpose. The assumption was that the MPI layer would join small messages already, which turned out to be false. The synthetic benchmarks could not discover this shortcoming of the protocol, as their scheme of operation is to alwas make sure the link is drained before the next message is sent to avoid caching effects. As this experience teaches, it might be worth to consider this issue when designing

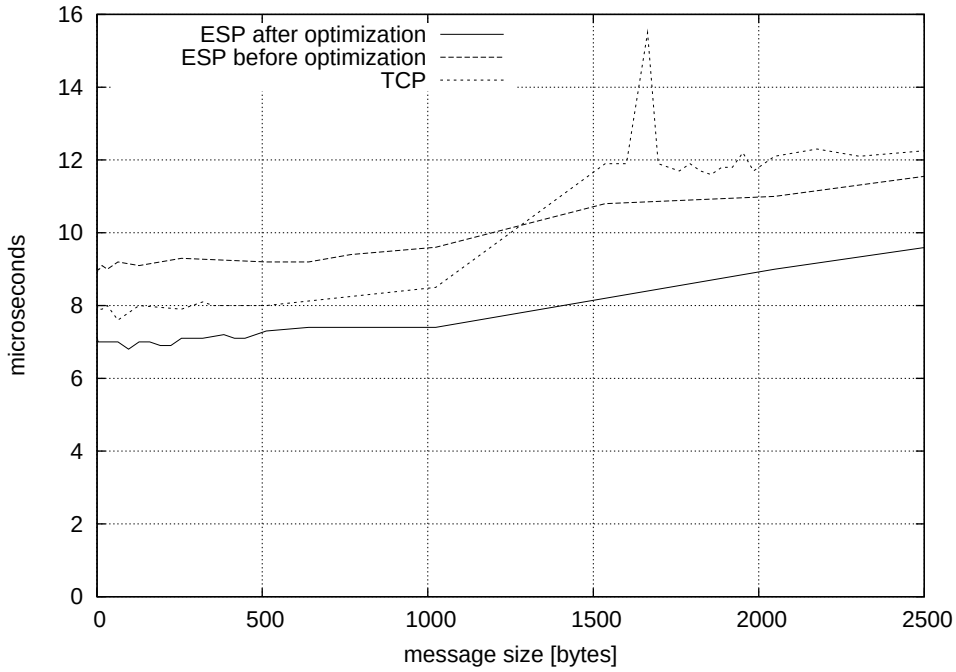


Figure 21: Result of the optimization for real-world applications. This is a plot of the send overhead and shows the benefit of being able to combine several small messages into larger packets.

Benchmark	# of nodes	ESP	TCP w/o TSO	TCP w TSO
cg	2	1.89	1.86	1.88
ep	2	**12.50**	12.53	12.62
is	2	1.75	1.72	1.64
lu	2	57.06	56.76	56.76
mg	2	2.18	2.19	2.18
bt	4	**35.37**	37.43	38.92
cg	4	1.38	1.27	1.30
ep	4	**6.32**	6.34	6.40
is	4	**1.06**	1.26	1.96
lu	4	30.04	29.87	29.66
mg	4	1.23	1.21	1.20
sp	4	**39.65**	43.44	44.33

Table 2: NAS Parallel Benchmark completion times for problem size A using 4 nodes. All values are given in seconds, and the times where ESP did better than TCP are set in bold face.

a network benchmark.

In addition to this optimization, the Open MPI BTL component for ESP developed by [Rei06] was updated to keep up with the latest developments of the component for the TCP protocol. Because both protocols are so similar in terms of the interface exposed to user-space, all optimizations found there could be ported to the ESP BTL component, namely the endpoint cache. This optimization can reduce significantly reduce the number of user- kernel-space transitions when reading many small messages off the socket's receive buffers.

6.6.2 NAS Parallel Benchmarks

The NAS Parallel Benchmarks [NPB] are a set of benchmarks which are derived from computational fluid dynamics applications, made available by the NASA Supercomputing Division. For the test runs presented here, version 3.1 of the benchmark suite was used. The suite consists of several indepentent programs. These can be compiled to operate on different problem sizes and using a different number of processors, but not every combination of problem size and processor count is possible. The results of these test runs are given in table 2.

6.6.3 ABINIT

As another benchmark to test the ESP performance for real-world applications, ABINIT [ABI] was chosen. For a short introduction to the ABINIT project, a quote from the project web site shall be given:

> ABINIT is a package whose main program allows one to find the total en-

> ergy, charge density and electronic structure of systems made of electrons and nuclei (molecules and periodic solids) within Density Functional Theory (DFT), using pseudopotentials and a planewave basis. ABINIT also includes options to optimize the geometry according to the DFT forces and stresses, or to perform molecular dynamics simulations using these forces, or to generate dynamical matrices, Born effective charges, and dielectric tensors. Excited states can be computed within the Time-Dependent Density Functional Theory (for molecules), or within Many-Body Perturbation Theory (the GW approximation).

While these words have almost no meaning to the author, the ABINIT package surely is a large-scale scientific application, which can be compiled to make use of parallel algorithms by the means of a MPI library. The parts of ABINIT for which parallel algorithms are used are:

- the treatment of k-points in reciprocal space;
- the treatment of spins, for spin-polarized collinear situations (when nsppol=2);
- the handling of bands;
- the FFTs (fast fourier transforms).

ABINIT was used to compute the SiN record on 4 nodes in parallel. When using the TCP protocol, the computations were finished in 581.7 seconds. Using ESP, only 555.8 secondes were required.

7 Summary and Conclusion

In this thesis the causes for congestion when using a switched ethernet in a cluster environment were examined, and it was investigated how different implementations of the TCP/IP deal with this situation. Additionally, the bandwidth utilization was taken into consideration.

Carrying on the gained knowledge, a benchmark tool was developed which is able to generate a network congestion situation–and can be used to do manifold measurements on different types of networks as well. By being a framework supporting the development of two types of modules (communication patterns and transport modules), the new version of Netgauge turned out to be a very versatile tool for measuring many aspects of communication hard- and software. For example, it has already been used in [HLR07], where another communication pattern was developed which aims for the assessment of networking infrastructure parameters.

In the next step, a flow control and congestion avoidance scheme was developed, which is suited specifically for the needs of a cluster environment. This scheme was designed to be very lightweight, and includes an elegant way to deal with the problem of congestion avoidance in a cluster. This flow control scheme was implemented on top

of the existing ESP protocol. Additionally, numerous program errors in the previos ESP protocol implementation could be patched, and an interface for accessing the parameters of the protocol from user-space was introduced.

The flow control scheme developed was analyzed theoretically as well as practically. In an effort to express the protocol's specifics in terms of parameters for the LogGP model of parallel computation, various existing strategies to assess the model's parameters were examined and found not to give satisfactory results. Because of their deficits, a novel approach in Log(G)P parameter assessment was done–and led to the proof that a gerneral approach to assess characteristics of a network in terms of concrete values for the Log(G)P model does not exist for user-space applications.

Finally, some scientific applications were executed which revealed that the ESP protocol, despite outperforming the TCP protocol in almost every synthetic benchmark, does not always perform better in the real world. Though, the goal of targeting a stability which is suitable for production use can be considered as to be reached. The shortcomings in performance, which some of the applications revealed are assumed to be owed to the "hidden" overhead as outlined in section 6.4. Other applications, namely the ABINIT suite, profit perceptibly from using the ESP protocol instead of TCP/IP.

Part of the hidden overhead is the processing of the acknowledgements. Because the current implemetation of the ESP protocol involves a mutex which is shared across all active sockets on a system (this mutex is used to protect the ACK queue), this is assumed to introduce additional overhead which could be lowered significantly if a future extension of the protocol was modified to use the algoritms presented in [Mic02] to manage the ACK queue. Additionally, further lower level optimizations are still possible, for example utilizing the capability to do scatter-gather I/O, which modern network interface controllers offer.

A Appendix

Corrected Bugs and Optimizations

Below is a brief summary of the bugs that could be identified and fixed during the development of this thesis. The items have been extracted from the commit logs of the version control system used, and only bugs which were not introduced by the author himself are included. Some minor optimizations, which were not mentioned in the main part of the thesis are included as well.

- Fixed the use of an unitialized address length parameter in the Open MPI BTL component.
- Fixed the dereferencing of a NULL pointer in `mca_btl_tcp_proc_remove`.
- Optimization: Two-split the acknowledgement processing: First, check if the ACK is valid and update `una_seq`, then send out pending packets, and finally remove the ack'd packets from the write queue. This gets new data on wire earlier.
- Some code expected the sequence number to increase with every packet, but only data, SYN and FIN frames eat sequence numbers.
- Several cleanups on socket close / destroy to avoid state diagramm violations in cases where the application and the remote host both want to close the connection at the same time.
- Taking care to always stop the RTX timer on socket close.
- Taking care to always clone the socket buffer prior to xmission.
- Optimization: Doing backlog processing without leaving the `recvmsg` loop by releasing and locking the socket.
- There was another severe problem with simultaneous close, which is resolved now.
- Purge receive queue on `FIN_WAIT_2` timeout. The memory was leaked previously.
- Added `set_state(ESP_CLOSE)` in `sock_esp_release` if unread data was purged to avoid state diagramm violation.
- Under certain conditions, the socket was not completely removed from the established hash table.
- Fixed the endless FIN retransmission bug. Finally, we're doing the `CLOSE_WAIT` to `LAST_ACK` transition.
- Taking care not to send out duplicate ACKs if they are not needed.
- Fixed hanging userspace application in sendmsg handler for closing / closed sockets.

- Fixed a `dev` ↔ `input_dev` mistaken bug in `esp_send_rst(...)`.
- The `ack_seq` was not always incremented when it needed to.
- Added a missing `dev_put(...)` for the case when `esp_connect(...)` fails.
- Added missing write lock for the bound hash table in `esp_sk_spawn_child(...)`.
- Fixed some kernel-version dependent debugging code.
- Fix for change of struct `packet_type` as of kernel version 2.6.14.
- A crash in the function `esp_get_port(...)` was fixed.

Configuration of the Cluster test Systems

Cluster 1

- CPU: 2 × AMD Athlon MP 1600+
- main memory: 512 MB
- network interface: SysKonnect SK-9821 V2.0 GigE Adapter (66MHz PCI)
- support for TSO: no
- operating system: CentOS 4.2 (Linux 2.6.9-22.ELsmp #6 SMP)

Cluster 2

- CPU: 2 × Intel Xeon (Woodcrest) Dual Core CPU, 2.0 GHz
- main memory: 2.0 GB
- network interface: Intel 631xESB/632xESB DPT (PCI-X)
- support for TSO: yes
- operating system: CentOS 4.2 (Linux 2.6.9-22.ELsmp #6 SMP)

References

[ABI] The abinit group. `http://www.abinit.org/`.

[AISS95] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. Loggp: incorporating long messages into the logp model–one step closer towards a realistic model for parallel computation. *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95 – 105, 1995.

[BBC+03] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick. An evaluation of current high-performance networks. Technical report, Lawrence Berkeley National Laboratory, April 2003.

[BMP94] Lawrence S. Brakmo, Sean W. O' Malley, and Larry L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. Technical report, Department of Computer Science at The University of Arizona, Tucson, February 1994.

[Bri02] Michael J. Brim. Predicting mpi-based parallel application performance on workstation clusters using loggp. Technical report, University of Wisconsin, 2002.

[CKP+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: towards a realistic model of parallel computation. *ACM SIGPLAN Notices, Volume 7, Issue 28*, pages 1 – 12, 1993.

[CLMY95] David Culler, Lok Tin Liu, Richard P. Martin, and Chad Yoshikawa. Logp performance assessment of fast network interfaces. Technical report, Computer Science Division, University of California, Berkeley, November 1995.

[DZ83] J. D. Day and H. Zimmermann. The osi reference model. *Proceedings of the IEEE*, pages 1334 – 1340, 1983.

[FIG+99] R. Fielding, UC Irvine, J. Gettys, Compaq/W3C, J. Mogul, and H. Frystyk. Rfc 2616 - hypertext transfer protocol – http/1.1. Technical report, Network Working Group, June 1999.

[Flo00] S. Floyd. Rfc 2914 - congestion control principles. Technical report, The Internet Society, 2000.

[Flo01] S. Floyd. A report on recent developments in tcp congestion control. *Communications Magazine*, pages 84–90, 2001.

[FRS02] Sally Floyd, Sylvia Ratnasamy, and Scott Shenker. Modifying tcp's congestion control for high speeds. `http://www.icir.org/floyd/papers/hstcp.pdf`, 2002.

[GWS05] R. L. Graham, T. S. Woodall, and J. M. Squyres. Open mpi: A flexible high performance mpi. Technical report, Lecture notes in computer science, 2005.

[HLR07] Torsten Hoefler, Andre Lichei, and Wolfgang Rehm. Low-overhead loggp parameter assessment for modern interconnection networks. Technical report, Chemnitz University of Technology, 2007.

[HRM+06] Torsten Hoefler, Mirko Reinhardt, Frank Mietke, Torsten Mehlan, and Wolfgang Rehm. Low overhead ethernet communication for open mpi on linux clusters. Technical report, Chemnitz Univesity of Technology, 2006.

[Ins81] Information Sciences Institute. Rfc 793 - transmission control protocol specification. Technical report, Defense Advanced Research Projects Agency, Information Processing Techniques Office, September 1981.

[Jac88] V. Jacobson. *Congestion avoidance and control.* 1988.

[JBB92] V. Jacobson, R. Braden, and D. Borman. Rfc 1323 - tcp extensions for high performance. Technical report, Network Working Group, May 1992.

[KBG00] T. Kielmann, H. E. Bal, and S. Gorlatch. Bandwidth-efficient collective communication for clustered widearea systems. *Parallel and Distributed Processing Symposium*, pages 492 – 499, 2000.

[KBV00] Thilo Kielmann, Henri E. Bal, and Kees Verstoep. Fast measurement of logp parameters for message passing platforms. *Parallel and Distributed Processing: 15 IPDPS 2000 Workshops*, page 1176, May 2000.

[KGD] The linux kernel debugger. `http://kgdb.linsyssoft.com/intro.htm`.

[MAW98] Jeonghoon Mo, Richard J. La Venkat Anantharam, and Jean Walrand. Analysis and comparison of tcp reno and vegas. Technical report, University of California at Berkeley, July 1998.

[Mic02] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73 – 82, 2002.

[Mou01] Erik Mouw. Linux kernel procfs guide. Technical report, Delft University of Technology, 2001.

[NF] The netfilter.org project. `http://www.netfilter.org/`.

[NPB] The nas parallel benchmarks. `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[Pac96] PS Pacheco. *Parallel programming with MPI.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Pfi01] G. F. Pfister. Aspects of the infiniband architecture. *IEEE International Conference on Cluster Computing. Proceedings. 20001*, pages 369–371, 2001.

[Rei06] Mirko Reinhardt. Optimizing point-to-point ethernet cluster communication. *Diploma Thesis*, 2006.

[SJA03] K.N. Srijith, L. Jacob, and A.L. Ananda. Tcp vegas-a: solving the fairness and rerouting issues of tcp vegas. Technical report, School of Computing at National University of Singapore, April 2003.

[SSV99] David Sundaram-Stukel and Mary K. Vernon. Predictive analysis of a wavefront application using loggp. *Principles and Practice of Parallel Programming*, pages 141 – 150, 1999.

[Ste97] W. Stevens. Rfc 2001 - tcp slow start, congestion avoidance, fast retransmit, and fast recovery algorithms. Technical report, Network Working Group, 1997.

[STP97] Understanding the spanning-tree protocol. `http://www.cisco.com/univercd/cc/td/doc/product/rtrmgmt/sw_ntman/cwsimain/cwsi2/cwsiug2/vlan2/stpapp.htm`, 1997.

List of Figures

Statutory Declaration

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, no other sources or auxiliary tools except those stated, referenced and acknowledged have been used.

Chemnitz, March 29, 2007

Matthias Treydte