Since the layers of the OSI were developed to be independent of one another, they contain a few header redundancies. Looking at transport layer protocols which were built primarily to be robust, we can also observe some unnecessary overhead in processing algorithms. The following articles are aimed at optimizing TCP and UDP.

As networks grew faster, the protocols they carry did not speed up proportionally. TCP is a very popular transport protocol which carries the functions of detecting and recovering lost/corrupted packets, flow control and multiplexing. It sits above the Network Layer which is connectionless and deals with host addressing and packet fragmentation/reassembly.

In order to optimize TCP, we need to look at the most common case and make it as efficient as possible, which is data transfer once the connection has been negotiated. There are three general stages to the TCP processing of incoming packets. First, finding the local state information by looking up the TCB depends on the details of the data structure and the assumed number of connections. Secondly, TCP checksum, implemented end-to-end, is computed after the packet is actually in main memory to provides a level of protection that is very valuable. However, computing this checksum using the CPU rather than some outboard chip may be a considerable burden on the protocol. Finally the processing of the packet’s performance is dependent on the code implemented. TCP puts most of its complexity in the sending end of the connection. This complexity is not caused by the actual sending of the data, but by processing the control information about that data in an incoming acknowledgment packet. We can observe that the next TCP segment received is highly likely to be destined for the same application as the last segment, so we can cash the last PCB used to avoid doing PCB lookup to find out who is to receive the segment.

Authors agree that protocol processing is not the real source of the processing overhead. Most of the delay is caused by the OS, since packet processing requires an interrupt, allocate buffer and initiate devices, which are all tasks performed by the OS. To copy data in memory requires two memory cycles, read and write. In other words, the bandwidth of the memory must be twice the achieved rate of the copy. Checksum computation has only one memory operation, since the data is being read but not written. In this implementation of TCP, receiving a packet thus requires four memory cycles per word, one for the input DMA, one for the checksum, and two for the copy. This is a far more important limit than the TCP processing limits computed above. The overhead of TCP would intrude into the limitations of the memory. So if the operating system and the memory overhead are the real limits, an optimization might be to avoid these by moving the processing outboard from the processor onto a special controller.

It is possible to save on sending complexities because of redundancies in the IP and TCP headers. One writer proposes an implementation where the entire state of the output side, including the unsent data, is stored as a preformatted output packet. This reduced the cost of sending a packet to a few lines of code, because all there is left to do is append the packets with the missing ID and checksum fields. We can also achieve a speedup at the receiving end by pre-computing what values should be found in the next incoming packet header for the connection. If the packets arrive in order, a few simple comparisons suffice to complete header processing.

A buffer layer can easily grow in complexity to swamp the protocol itself because it is the part of the code in which there is the greatest demand for varieties of service and flexibility. The problem of the buffer layer is made worse by the fact that the protocol descriptions do not admit that such a layer exists since it is not a part of the IS0 reference model.

Since most TCP optimizations have focused on prediction algorithm using information that TCP must store about the connection state to predict the next segment that will arrive and optimize the processing of the expected segment, one could ask if some of those enhancements that work for state full protocols like TCP also work for stateless protocols like UDP.

We can also optimize UDP by compressing its header in order to reduce the overhead of carrying small packets of size comparable to their header. In order to do so, we have to cross the protocol layer boundaries to eliminate redundancies. In TCP, since half of the header remains constant over the conversation, once a full packet have been transmitted, subsequent packets on the same communication can be stripped of this redundant information.

To achieve UDP packet compression, in the case presented for RTP applications, we can observe redundancies between layers, as well as constant changing fields. For example the total length, packet ID, and header check-sum fields are all present in duplicate between network layers and the information they convey can be merged. The only non constant one is the packet ID, but since it changes constantly, we can forward a first full header, notifying the receiver of the rate of change of the different fields in the header, then strip subsequent headers of these redundant fields. In order to implement this, we will have to maintain a session between the sender and the receiver in order to record the state of the conversation. This session will include the source and destination’s IP, UDP port number, RTP SSRC field and first order difference of ID fields, as well as initial value for these fields. With this session information, the receiver will be able to reconstruct the original UDP packets.

There are also other enhancements possible to make in UDP processing, not related to shrinking packet headers. By focusing on reducing the per-packet overhead and the improving data handling, in particular to combine checksum and copy loops to save on overhead. Initially, the copy loop had one read and one write for each word of data, and the checksum had to read the data again, for a total of three accesses. The combined loop reads each word into a register, adds the word to the running checksum, and then writes the word, thus giving two accesses plus an addition. The latter enhancements, even though they were a lot more difficult to implement, proved to be the most efficient.

This became obvious from the observation that a majority of the kernel’s time was spent simply handling interrupts and managing changes in processor priority to protect critical code regions. So reducing the number of data copies required for processing was a great performance enhancement. A way to do this is to combine the checksum and copy loops. Reducing the number of memory accesses also helped with OS overhead, which is required to perform interrupt driven context changes to process incoming packets.

We can also use the implementation of ‘one-behind-cashes’ to exploit locality. This enhancement exploits the fact that there is a good chance that the next datagram received will be destined for the same socket as the previous datagram received. Keeping the cache makes it possible to avoid a more expensive search of all the UDP PCBs.

One last UPD performance enhancement was to use replace expensive general purpose code with code tuned to the particular protocol. For example, computing the IP header checksum in-line rather than in a general purpose subroutine, one author was able to see significant performance enhancements.