Network Working Group
Request for Comments: 1263

S. O'Malley
L. Peterson
University of Arizona

October 1991

TCP EXTENSIONS CONSIDERED HARMFUL

Status of this Memo

This memo provides information for the Internet community. It does not specify an Internet standard. Distribution of this document is unlimited.

Abstract

This RFC comments on recent proposals to extend TCP. It argues that the backward compatible extensions proposed in RFC's 1072 and 1185 should not be pursued, and proposes an alternative way to evolve the Internet protocol suite. Its purpose is to stimulate discussion in the Internet community.

1. Introduction

The rapid growth of the size, capacity, and complexity of the Internet has led to the need to change the existing protocol suite. For example, the maximum TCP window size is no longer sufficient to efficiently support the high capacity links currently being planned and constructed. One is then faced with the choice of either leaving the protocol alone and accepting the fact that TCP will run no faster on high capacity links than on low capacity links, or changing TCP. This is not an isolated incident. We have counted at least eight other proposed changes to TCP (some to be taken more seriously than others), and the question is not whether to change the protocol suite, but what is the most cost effective way to change it.

This RFC compares the costs and benefits of three approaches to making these changes: the creation of new protocols, backward compatible protocol extensions, and protocol evolution. The next section introduces these three approaches and enumerates the strengths and weaknesses of each. The following section describes how we believe these three approaches are best applied to the many proposed changes to TCP. Note that we have not written this RFC as an academic exercise. It is our intent to argue against acceptance of the various TCP extensions, most notably RFC's 1072 and 1185 [4,5], by describing a more palatable alternative.

2. Creation vs. Extension vs. Evolution

2.1. Protocol Creation

Protocol creation involves the design, implementation, standardization, and distribution of an entirely new protocol. In this context, there are two basic reasons for creating a new protocol. The first is to replace an old protocol that is so outdated that it can no longer be effectively extended to perform its original function. The second is to add a new protocol because users are making demands upon the original protocol that were not envisioned by the designer and cannot be efficiently handled in terms of the original protocol. For example, TCP was designed as a reliable byte-stream protocol but is commonly used as both a reliable record-stream protocol and a reliable request-reply protocol due to the lack of such protocols in the Internet protocol suite. The performance demands placed upon a byte-stream protocol in the new Internet environment makes it difficult to extend TCP to meet these new application demands.

The advantage of creating a new protocol is the ability to start with a clean sheet of paper when attempting to solve a complex network problem. The designer, free from the constraints of an existing protocol, can take maximum advantage of modern network research in the basic algorithms needed to solve the problem. Even more importantly, the implementor is free to steal from a large number of existing academic protocols that have been developed over the years. In some cases, if truly new functionality is desired, creating a new protocol is the only viable approach.

The most obvious disadvantage of this approach is the high cost of standardizing and distributing an entirely new protocol. Second, there is the issue of making the new protocol reliable. Since new protocols have not undergone years of network stress testing, they often contain bugs which require backward compatible fixes, and hence, the designer is back where he or she started. A third disadvantage of introducing new protocols is that they generally have new interfaces which require significant effort on the part of the Internet community to use. This alone is often enough to kill a new protocol.

Finally, there is a subtle problem introduced by the very freedom provided by this approach. Specifically, being able to introduce a new protocol often results in protocols that go far beyond the basic needs of the situation. New protocols resemble Senate appropriations bills; they tend to accumulate many amendments that have nothing to do with the original problem. A good example of this phenomena is the attempt to standardize VMTP [1] as the Internet RPC protocol. While

VMTP was a large protocol to begin with, the closer it got to standardization the more features were added until it essentially collapsed under its own weight. As we argue below, new protocols should initially be minimal, and then evolve as the situation dictates.

2.2. Backward Compatible Extensions

In a backward compatible extension, the protocol is modified in such a fashion that the new version of the protocol can transparently inter-operate with existing versions of the protocol. This generally implies no changes to the protocol's header. TCP slow start [3] is an example of such a change. In a slightly more relaxed version of backward compatibility, no changes are made to the fixed part of a protocol's header. Instead, either some fields are added to the variable length options field found at the end of the header, or existing header fields are overloaded (i.e., used for multiple purposes). However, we can find no real advantage to this technique over simply changing the protocol.

Backward compatible extensions are widely used to modify protocols because there is no need to synchronize the distribution of the new version of the protocol. The new version is essentially allowed to diffuse through the Internet at its own pace, and at least in theory, the Internet will continue to function as before. Thus, the explicit distribution costs are limited. Backward compatible extensions also avoid the bureaucratic costs of standardizing a new protocol. TCP is still TCP and the approval cost of a modification to an existing protocol is much less than that of a new protocol. Finally, the very difficulty of making such changes tends to restrict the changes to the minimal set needed to solve the current problem. Thus, it is rare to see unneeded changes made when using this technique.

Unfortunately, this approach has several drawbacks. First, the time to distribute the new version of the protocol to all hosts can be quite long (forever in fact). This leaves the network in a heterogeneous state for long periods of time. If there is the slightest incompatibly between old and new versions, chaos can result. Thus, the implicit cost of this type of distribution can be quite high. Second, designing a backward compatible change to a new protocol is extremely difficult, and the implementations "tend toward complexity and ugliness" [5]. The need for backward compatibility ensures that no code can every really be eliminated from the protocol, and since such vestigial code is rarely executed, it is often wrong. Finally, most protocols have limits, based upon the design decisions of it inventors, that simply cannot be side-stepped in this fashion.

2.3. Protocol Evolution

Protocol evolution is an approach to protocol change that attempts to escape the limits of backward compatibility without incurring all of the costs of creating new protocols. The basic idea is for the protocol designer to take an existing protocol that requires modification and make the desired changes without maintaining backward compatibility. This drastically simplifies the job of the protocol designer. For example, the limited TCP window size could be fixed by changing the definition of the window size in the header from 16-bits to 32-bits, and re-compiling the protocol. The effect of backward compatibility would be ensured by simply keeping both the new and old version of the protocol running until most machines use the new version. Since the change is small and invisible to the user interface, it is a trivial problem to dynamically select the correct TCP version at runtime. How this is done is discussed in the next section.

Protocol evolution has several advantages. First, it is by far the simplest type of modification to make to a protocol, and hence, the modifications can be made faster and are less likely to contain bugs. There is no need to worry about the effects of the change on all previous versions of the protocol. Also, most of the protocol is carried over into the new version unchanged, thus avoiding the design and debugging cost of creating an entirely new protocol. Second, there is no artificial limit to the amount of change that can be made to a protocol, and as a consequence, its useful lifetime can be extended indefinitely. In a series of evolutionary steps, it is possible to make fairly radical changes to a protocol without upsetting the Internet community greatly. Specifically, it is possible to both add new features and remove features that are no longer required for the current environment. Thus, the protocol is not condemned to grow without bound. Finally, by keeping the old version of the protocol around, backward compatibility is guaranteed. The old code will work as well as it ever did.

Assuming the infrastructure described in the following subsection, the only real disadvantage of protocol evolution is the amount of memory required to run several versions of the same protocol. Fortunately, memory is not the scarcest resource in modern workstations (it may, however, be at a premium in the BSD kernel and its derivatives). Since old versions may rarely if ever be executed, the old versions can be swapped out to disk with little performance loss. Finally, since this cost is explicit, there is a huge incentive to eliminate old protocol versions from the network.

2.4. Infrastructure Support for Protocol Evolution

The effective use of protocol evolution implies that each protocol is considered a vector of implementations which share the same top level interface, and perhaps not much else. TCP[0] is the current implementation of TCP and exists to provide backward compatibility with all existing machines. TCP[1] is a version of TCP that is optimized for high-speed networks. TCP[0] is always present; TCP[1] may or may not be. Treating TCP as a vector of protocols requires only three changes to the way protocols are designed and implemented.

First, each version of TCP is assigned a unique id, but this id is not given as an IP protocol number. (This is because IP's protocol number field is only 8 bits long and could easily be exhausted.) The "obvious" solution to this limitation is to increase IP's protocol number field to 32 bits. In this case, however, the obvious solution is wrong, not because of the difficultly of changing IP, but simply because there is a better approach. The best way to deal with this problem is to increase the IP protocol number field to 32 bits and move it to the very end of the IP header (i.e., the first four bytes of the TCP header). A backward compatible modification would be made to IP such that for all packets with a special protocol number, say 77, IP would look into the four bytes following its header for its de-multiplexing information. On systems which do not support a modified IP, an actual protocol 77 would be used to perform the de-multiplexing to the correct TCP version.

Second, a version control protocol, called VTCP, is used to select the appropriate version of TCP for a particular connection. VTCP is an example of a virtual protocol as introduced in [2]. Application programs access the various versions of TCP through VTCP. When a TCP connection is opened to a specific machine, VTCP checks its local cache to determine the highest common version shared by the two machines. If the target machine is in the cache, it opens that version of TCP and returns the connection to the protocol above and does not effect performance. If the target machine is not found in the cache, VTCP sends a UDP packet to the other machine asking what versions of TCP that machine supports. If it receives a response, it uses that information to select a version and puts the information in the cache. If no reply is forthcoming, it assumes that the other machine does not support VTCP and attempts to open a TCP[0] connection. VTCP's cache is flushed occasionally to ensure that its information is current.

Note that this is only one possible way for VTCP to decide the right version of TCP to use. Another possibility is for VTCP to learn the right version for a particular host when it resolves the host's name. That is, version information could be stored in the Domain Name

System. It is also possible that VTCP might take the performance characteristics of the network into consideration when selecting a version; TCP[0] may in fact turn out to be the correct choice for a low-bandwidth network.

Third, because our proposal would lead to a more dynamically changing network architecture, a mechanism for distributing new versions will need to be developed. This is clearly the hardest requirement of the infrastructure, but we believe that it can be addressed in stages. More importantly, we believe this problem can be addressed after the decision has been made to go the protocol evolution route. In the short term, we are considering only a single new version of TCP---TCP[1]. This version can be distributed in the same ad hoc way, and at exactly the same cost, as the backward compatible changes suggested in RFC's 1072 and 1185.

In the medium term, we envision the IAB approving new versions of TCP every year or so. Given this scenario, a simple distribution mechanism can be designed based on software distribution mechanisms that have be developed for other environments; e.g., Unix RDIST and Mach SUP. Such a mechanism need not be available on all hosts. Instead, hosts will be divided into two sets, those that can quickly be updated with new protocols and those that cannot. High performance machines that can use high performance networks will need the most current version of TCP as soon as it is available, thus they have incentive to change. Old machines which are too slow to drive a high capacity lines can be ignored, and probably should be ignored.

In the long term, we envision protocols being designed on an application by application basis, without the need for central approval. In such a world, a common protocol implementation environment---a protocol backplane---is the right way to go. Given such a backplane, protocols can be automatically installed over the network. While we claim to know how to build such an environment, such a discussion is beyond the scope of this paper.

2.5. Remarks

Each of these three methods has its advantages. When used in combination, the result is better protocols at a lower overall cost. Backward compatible changes are best reserved for changes that do not affect the protocol's header, and do not require that the instance running on the other end of the connection also be changed. Protocol evolution should be the primary way of dealing with header fields that are no longer large enough, or when one algorithm is substituted directly for another. New protocols should be written to off load unexpected user demands on existing protocols, or better yet, to catch them before they start.

There are also synergistic effects. First, since we know it is possible to evolve a newly created protocol once it has been put in place, the pressure to add unnecessary features should be reduced. Second, the ability to create new protocols removes the pressure to overextend a given protocol. Finally, the ability to evolve a protocol removes the pressure to maintain backward compatibility where it is really not possible.

3. TCP Extensions: A Case Study

This section examines the effects of using our proposed methodology to implement changes to TCP. We will begin by analyzing the backward compatible extensions defined in RFC's 1072 and 1185, and proposing a set of much simpler evolutionary modifications. We also analyze several more problematical extensions to TCP, such as Transactional TCP. Finally, we point our some areas of TCP which may require changes in the future.

The evolutionary modification to TCP that we propose includes all of the functionality described in RFC's 1072 and 1185, but does not preserve the header format. At the risk of being misunderstood as believing backward compatibility is a good idea, we also show how our proposed changes to TCP can be folded into a backward compatible implementation of TCP. We do this as a courtesy for those readers that cannot accept the possibility of multiple versions of TCP.

3.1. RFC's 1072 and 1185

3.1.1. Round Trip Timing

In RFC 1072, a new ECHO option is proposed that allows each TCP packet to carry a timestamp in its header. This timestamp is used to keep a more accurate estimate of the RTT (round trip time) used to decide when to re-transmit segments. In the original TCP algorithm, the sender manually times a small number of sends. The resulting algorithm was quite complex and does not produce an accurate enough RTT for high capacity networks. The inclusion of a timestamp in every header both simplifies the code needed to calculate the RTT and improves the accuracy and robustness of the algorithm.

The new algorithm as proposed in RFC 1072 does not appear to have any serious problems. However, the authors of RFC 1072 go to great lengths in an attempt to keep this modification backward compatible with the previous version of TCP. They place an ECHO option in the

SYN segment and state, "It is likely that most implementations will properly ignore any options in the SYN segment that they do not understand, so new initial options should not cause problems" [4]. This statement does not exactly inspire confidence, and we consider the addition of an optional field to any protocol to be a de-facto, if not a de-jure, example of an evolutionary change. Optional fields simply attempt to hide the basic incompatibility inside the protocol, it does not eliminate it. Therefore, since we are making an evolutionary change anyway, the only modification to the proposed algorithm is to move the fields into the header proper. Thus, each header will contain 32-bit echo and echo reply fields. Two fields are needed to handle bi-directional data streams.

3.1.2. Window Size and Sequence Number Space

Long Fat Networks (LFN's), networks which contain very high capacity lines with very high latency, introduce the possibility that the number of bits in transit (the bandwidth-delay product) could exceed the TCP window size, thus making TCP the limiting factor in network performance. Worse yet, the time it takes the sequence numbers to wrap around could be reduced to a point below the MSL (maximum segment lifetime), introducing the possibility of old packets being mistakenly accepted as new.

RFC 1072 extends the window size through the use of an implicit constant scaling factor. The window size in the TCP header is multiplied by this factor to get the true window size. This algorithm has three problems. First, one must prove that at all times the implicit scaling factor used by the sender is the same as the receiver. The proposed algorithm appears to do so, but the complexity of the algorithm creates the opportunity for poor implementations to affect the correctness of TCP. Second, the use of a scaling factor complicates the TCP implementation in general, and can have serious effects on other parts of the protocol.

A final problem is what we characterize as the "quantum window sizing" problem. Assuming that the scaling factors will be powers of two, the algorithm right shifts the receiver's window before sending it. This effectively rounds the window size down to the nearest multiple of the scaling factor. For large scaling factors, say 64k, this implies that window values are all multiples of 64k and the minimum window size is 64k; advertising a smaller window is impossible. While this is not necessarily a problem (and it seems to be an extreme solution to the silly window syndrome) what effect this will have on the performance of high-speed network links is anyone's guess. We can imagine this extension leading to future papers entitled "A Quantum Mechanical Approach to Network Performance".

RFC 1185 is an attempt to get around the problem of the window wrapping too quickly without explicitly increasing the sequence number space. Instead, the RFC proposes to use the timestamp used in the ECHO option to weed out old duplicate messages. The algorithm presented in RFC 1185 is complex and has been shown to be seriously flawed at a recent End-to-End Research Group meeting. Attempts are currently underway to fix the algorithm presented in the RFC. We believe that this is a serious mistake.

We see two problems with this approach on a very fundamental level. First, we believe that making TCP depend on accurate clocks for correctness to be a mistake. The Internet community has NO experience with transport protocols that depend on clocks for correctness. Second, the proposal uses two distinct schemes to deal with old duplicate packets: the sliding window algorithm takes care of "new" old packets (packets from the current sequence number epoch) and the timestamp algorithm deals with "old" old packets (packets from previous sequence number epochs). It is hard enough getting one of these schemes to work much less to get two to work and ensure that they do not interfere with one another.

In RFC 1185, the statement is made that "An obvious fix for the problem of cycling the sequence number space is to increase the size of the TCP sequence number field." Using protocol evolution, the obvious fix is also the correct one. The window size can be increased to 32 bits by simply changing a short to a long in the definition of the TCP header. At the same time, the sequence number and acknowledgment fields can be increased to 64 bits. This change is the minimum complexity modification to get the job done and requires little or no analysis to be shown to work correctly.

On machines that do not support 64-bit integers, increasing the sequence number size is not as trivial as increasing the window size. However, it is identical in cost to the modification proposed in RFC 1185; the high order bits can be thought of as an optimal clock that ticks only when it has to. Also, because we are not dealing with real time, the problems with unreliable system clocks is avoided. On machines that support 64-bit integers, the original TCP code may be reused. Since only very high performance machines can hope to drive a communications network at the rates this modification is designed to support, and the new generation of RISC microprocessors (e.g., MIPS R4000 and PA-RISC) do support 64-bit integers, the assumption of 64-bit arithmetic may be more of an advantage than a liability.

3.1.3. Selective Retransmission

Another problem with TCP's support for LFN's is that the sliding window algorithm used by TCP does not support any form of selective acknowledgment. Thus, if a segment is lost, the total amount of data that must be re-transmitted is some constant times the bandwidth-delay product, despite the fact that most of the segments have in fact arrived at the receiver. RFC 1072 proposes to extend TCP to allow the receiver to return partial acknowledgments to the sender in the hope that the sender will use that information to avoid unnecessary re-transmissions.

It has been our experience on predictable local area networks that the performance of partial re-transmission strategies is highly non-obvious, and it generally requires more than one iteration to find a decent algorithm. It is therefore not surprising that the algorithm proposed in RFC 1072 has some problems. The proposed TCP extension allows the receiver to include a short list of received fragments with every ACK. The idea being that when the receiver sends back a normal ACK, it checks its queue of segments that have been received out of order and sends the relative sequence numbers of contiguous blocks of segments back to the sender. The sender then uses this information to re-transmit the segments transmitted but not listed in the ACK.

As specified, this algorithm has two related problems: (1) it ignores the relative frequencies of delivered and dropped packets, and (2) the list provided in the option field is probably too short to do much good on networks with large bandwidth-delay products. In every model of high bandwidth networks that we have seen, the packet loss rate is very low, and thus, the ratio of dropped packets to delivered packets is very low. An algorithm that returns ACKs as proposed is simply going to have to send more information than one in which the receiver returns NAKs.

This problem is compounded by the short size of the TCP option field (44 bytes). In theory, since we are only worried about high bandwidth networks, returning ACKs instead of NAKs is not really a problem; the bandwidth is available to send any information that's needed. The problem comes when trying to compress the ACK information into the 44 bytes allowed. The proposed extensions effectively compresses the ACK information by allowing the receiver to ACK byte ranges rather than segments, and scaling the relative sequence numbers of the re-transmitted segments. This makes it much more difficult for the sender to tell which segments should be re-transmitted, and complicates the re-transmission code. More importantly, one should never compress small amounts of data being sent over a high bandwidth network; it trades a scarce resource for an abundant resource. On low bandwidth networks, selective retransmission is not needed and the SACK option should be disabled.

We propose two solutions to this problem. First, the receiver can examine its list of out-of-order packets and guess which segments have been dropped, and NAK those segments back to the sender. The number of NAKs should be low enough that one per TCP packet should be sufficient. Note that the receiver has just as much information as the sender about what packets should be retransmitted, and in any case, the NAKs are simply suggestions which have no effect on correctness.

Our second proposed modification is to increase the offset field in the TCP header from 4 bits to 16 bits. This allows 64k-bytes of TCP header, which allows us to radically simplify the selective re-transmission algorithm proposed in RFC 1072. The receiver can now simply send a list of 64-bit sequence numbers for the out-of-order segments to the sender. The sender can then use this information to do a partial retransmission without needing an ouji board to translate ACKs into segments. With the new header size, it may be faster for the receiver to send a large list than to attempt to aggregate segments into larger blocks.

3.1.4. Header Modifications

The modifications proposed above drastically change the size and structure of the TCP header. This makes it a good time to re-think the structure of the proposed TCP header. The primary goal of the current TCP header is to save bits in the output stream. When TCP was developed, a high bandwidth network was 56kbps, and the key use for TCP was terminal I/O. In both situations, minimal header size was important. Unfortunately, while the network has drastically increased in performance and the usage pattern of the network is now vastly different, most protocol designers still consider saving a few bits in the header to be worth almost any price. Our basic goal is different: to improve performance by eliminating the need to extract information packed into odd length bit fields in the header. Below is our first cut at such a modification.

The protocol id field is there to make further evolutionary modifications to TCP easier. This field basically subsumes the protocol number field contained in the IP header with a version number. Each distinct TCP version has a different protocol id and this field ensures that the right code is looking at the right header. The offset field has been increased to 16 bits to support the larger header size required, and to simplify header processing. The code field has been extended to 16 bits to support more options.

The source port and destination port are unchanged. The size of both the sequence number and ACK fields have been increased to 64 bits. The open window field has been increased to 32 bits. The checksum and urgent data pointer fields are unchanged. The echo and echo reply fields are added. The option field remains but can be much larger than in the old TCP. All headers are padded out to 32 bit boundaries. Note that these changes increase the minimum header size from 24 bytes (actually 36 bytes if the ECHO and ECHO reply options defined in RFC 1072 are included on every packet) to 48 bytes. The maximum header size has been increased to the maximum segment size. We do not believe that the the increased header size will have a measurable effect on protocol performance.

```
    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                        Protocol ID                            |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |              Offset           |              Code             |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |              Source           |              Dest             |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                              Seq                              |
   |                                                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                              Ack                              |
   |                                                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                            Window                             |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |             Checksum          |             Urgent            |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                             Echo                              |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                          Echo Reply                           |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |  Options                                      |     Pad       |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

3.1.5. Backward Compatibility

The most likely objection to the proposed TCP extension is that it is not backward compatible with the current version of TCP, and most importantly, TCP's header. In this section we will present three versions of the proposed extension with increasing degrees of backward compatibility. The final version will combine the same degree of backward compatibility found in the protocol described in

RFC's 1072/1185, with the much simpler semantics described in this RFC.

We believe that the best way to preserve backward compatibility is to leave all of TCP alone and support the transparent use of a new protocol when and where it is needed. The basic scheme is the one described in section 2.4. Those machines and operating systems that need to support high speed connections should implement some general protocol infrastructure that allows them to rapidly evolve protocols. Machines that do not require such service simply keep using the existing version of TCP. A virtual protocol is used to manage the use of multiple TCP versions.

This approach has several advantages. First, it guarantees backward compatibility with ALL existing TCP versions because such implementations will never see strange packets with new options. Second, it supports further modification of TCP with little additional costs. Finally, since our version of TCP will more closely resemble the existing TCP protocol than that proposed in RFC's 1072/1185, the cost of maintaining two simple protocols will probably be lower than maintaining one complex protocol. (Note that with high probability you still have to maintain two versions of TCP in any case.) The only additional cost is the memory required for keeping around two copies of TCP.

For those that insist that the only efficient way to implement TCP modifications is in a single monolithic protocol, or those that believe that the space requirements of two protocols would be too great, we simply migrate the virtual protocol into TCP. TCP is modified so that when opening a connection, the sender uses the TCP VERSION option attached to the SYN packet to request using the new version. The receiver responds with a TCP VERSION ACK in the SYN ACK packet, after which point, the new header format described in Section 3.1.4 is used. Thus, there is only one version of TCP, but that version supports multiple header formats. The complexity of such a protocol would be no worse than the protocol described in RFC 1072/1185. It does, however, make it more difficult to make additional changes to TCP.

Finally, for those that believe that the preservation of the TCP's header format has any intrinsic value (e.g., for those that don't want to re-program their ethernet monitors), a header compatible version of our proposal is possible. One simply takes all of the additional information contained in the header given in Section 3.1.4 and places it into a single optional field. Thus, one could define a new TCP option which consists of the top 32 bits of the sequence and ack fields, the echo and echo_reply fields, and the top 16 bits of the window field. This modification makes it more difficult to take advantage of machines with 64-bit address spaces, but at a minimum will be just as easy to process as the protocol described in RFC 1072/1185. The only restriction is that the size of the header option field is still limited to 44 bytes, and thus, selective retransmission using NAKs rather than ACKs will probably be required.

The key observation is that one should make a protocol extension correct and simple before trying to make it backward compatible. As far as we can tell, the only advantages possessed by the protocol described in RFC 1072/1185 is that its typical header, size including options, is 8 to 10 bytes shorter. The price for this "advantage" is a protocol of such complexity that it may prove impossible for normal humans to implement. Trying to maintain backward compatibility at every stage of the protocol design process is a serious mistake.

3.2. TCP Over Extension

Another potential problem with TCP that has been discussed recently, but has not yet resulted in the generation of an RFC, is the potential for TCP to grab and hold all 2**16 port numbers on a given machine. This problem is caused by short port numbers, long MSLs, and the misuse of TCP as a request-reply protocol. TCP must hold onto each port after a close until all possible messages to that port have died, about 240 seconds. Even worse, this time is not decreasing with increase network performance. With new fast hardware, it is possible for an application to open a TCP connection, send data, get a reply, and close the connection at a rate fast enough to use up all the ports in less than 240 seconds. This usage pattern is generated by people using TCP for something it was never intended to do--- guaranteeing at-most-once semantics for remote procedure calls.

The proposed solution is to embed an RPC protocol into TCP while preserving backward compatibility. This is done by piggybacking the request message on the SYN packet and the reply message on the SYN-ACK packet. This approach suffers from one key problem: it reduces the probability of a correct TCP implementation to near 0. The basic problem has nothing to do with TCP, rather it is the lack of an Internet request-reply protocol that guarantees at-most-once semantics.

We propose to solve this problem by the creation of a new protocol. This has already been attempted with VMTP, but the size and complexity of VMTP, coupled with the process currently required to standardize a new protocol doomed it from the start. Instead of solving the general problem, we propose to use Sprite RPC [7], a much simpler protocol, as a means of off-loading inappropriate users from TCP.

The basic design would attempt to preserve as much of the TCP interface as possible in order that current TCP (mis)users could be switched to Sprite RPC without requiring code modification on their part. A virtual protocol could be used to select the correct protocol TCP or Sprite RPC if it exists on the other machine. A backward compatible modification to TCP could be made which would simply prevent it from grabbing all of the ports by refusing connections. This would encourage TCP abusers to use the new protocol.

Sprite RPC, which is designed for a local area network, has two problems when extended into the Internet. First, it does not have a usefully flow control algorithm. Second, it lacks the necessary semantics to reliably tear down connections. The lack of a tear down mechanism needs to be solved, but the flow control problem could be dealt with in later iterations of the protocol as Internet blast protocols are not yet well understood; for now, we could simple limit the size of each message to 16k or 32k bytes. This might also be a good place to use a decomposed version of Sprite RPC [2], which exposes each of these features as separate protocols. This would permit the quick change of algorithms, and once the protocol had stabilized, a monolithic version could be constructed and distributed to replace the decomposed version.

In other words, the basic strategy is to introduce as simple of RPC protocol as possible today, and later evolve this protocol to address the known limitations.

3.3. Future Modifications

The header prediction algorithm should be generalized so as to be less sensitive to changes in the protocols header and algorithm. There almost seems to be as much effort to make all modifications to TCP backward compatible with header prediction as there is to make them backward compatible with TCP. The question that needs to be answered is: are there any changes we can made to TCP to make header prediction easier, including the addition of information into the header. In [6], the authors showed how one might generalize optimistic blast from VMTP to almost any protocol that performs fragmentation and reassembly. Generalizing header prediction so that it scales with TCP modification would be step in the right direction.

It is clear that an evolutionary change to increase the size of the source and destination ports in the TCP header will eventually be necessary. We also believe that TCP could be made significantly simpler and more flexible through the elimination of the pseudo-header. The solution to this problem is to simply add a length field and the IP address of the destination to the TCP header. It has also been mentioned that better and simpler TCP connection establishment algorithms would be useful. Some form of reliable record stream protocol should be developed. Performing sliding window and flow control over records rather than bytes would provide numerous opportunities for optimizations and allow TCP to return to its original purpose as a byte-stream protocol. Finally, it has become clear to us that the current Internet congestion control strategy is to use TCP for everything since it is the only protocol that supports congestion control. One of the primary reasons many "new protocols" are proposed as TCP options is that it is the only way to get at TCP's congestion control. At some point, a TCP-independent congestion control scheme must be implemented and one might then be able to remove the existing congestion control from TCP and radically simplify the protocol.

4. Discussion

One obvious side effect of the changes we propose is to increase the size of the TCP header. In some sense, this is inevitable; just about every field in the header has been pushed to its limit by the radical growth of the network. However, we have made very little effort to make the minimal changes to solve the current problem. In fact, we have tended to sacrifice header size in order to defer future changes as long as possible. The problem with this is that one of TCP's claims to fame is its efficiency at sending small one byte packets over slow networks. Increasing the size of the TCP header will inevitably result in some increase in overhead on small packets on slow networks. Clark among others have stated that they see no fundamental performance limitations that would prevent TCP from supporting very high speed networks. This is true as far as it goes; there seems to be a direct trade-off between TCP performance on high speed networks and TCP performance on slow speed networks. The dynamic range is simply too great to be optimally supported by one protocol. Hence, in keeping around the old version of TCP we have effectively split TCP into two protocols, one for high bandwidth lines and the other for low bandwidth lines.

Another potential argument is that all of the changes mentioned above should be packaged together as a new version of TCP. This version could be standardized and we could all go back to the status quo of stable unchanging protocols. While to a certain extent this is inevitable---there is a backlog of necessary TCP changes because of the current logistical problems in modifying protocols---it is only begs the question. The status quo is simply unacceptably static; there will always be future changes to TCP. Evolutionary change will also result in a better and more reliable TCP. Making small changes and distributing them at regular intervals ensures that one change has actually been stabilized before the next has been made. It also presents a more balanced workload to the protocol designer; rather than designing one new protocol every 10 years he makes annual protocol extensions. It will also eventually make protocol distribution easier: the basic problem with protocol distribution now is that it is done so rarely that no one knows how to do it and there is no incentive to develop the infrastructure needed to perform the task efficiently. While the first protocol distribution is almost guaranteed to be a disaster, the problem will get easier with each additional one. Finally, such a new TCP would have the same problems as VMTP did; a radically new protocol presents a bigger target.

The violation of backward compatibility in systems as complex as the Internet is always a serious step. However, backward compatibility is a technique, not a religion. Two facts are often overlooked when backward compatibility gets out of hand. First, violating backward compatibility is always a big win when you can get away with it. One of the key advantages of RISC chips over CISC chips is simply that they were not backward compatible with anything. Thus, they were not bound by design decisions made when compilers were stupid and real men programmed in assembler. Second, one is going to have to break backward compatibility at some point anyway. Every system has some headroom limitations which result in either stagnation (IBM mainframe software) or even worse, accidental violations of backward compatibility.

Of course, the biggest problem with our approach is that it is not compatible with the existing standardization process. We hope to be able to design and distribute protocols in less time than it takes a standards committee to agree on an acceptable meeting time. This is inevitable because the basic problem with networking is the standardization process. Over the last several years, there has been a push in the research community for lightweight protocols, when in fact what is needed are lightweight standards. Also note that we have not proposed to implement some entirely new set of "superior" communications protocols, we have simply proposed a system for making necessary changes to the existing protocol suites fast enough to keep up with the underlying change in the network. In fact, the first standards organization that realizes that the primary impediment to standardization is poor logistical support will probably win.

5. Conclusions

The most important conclusion of this RFC is that protocol change happens and is currently happening at a very respectable clip. While all of the changes given as example in this document are from TCP, there are many other protocols that require modification. In a more prosaic domain, the telephone company is running out of phone numbers; they are being overrun by fax machines, modems, and cars. The underlying cause of these problems seems to be an consistent exponential increase almost all network metrics: number of hosts, bandwidth, host performance, applications, and so on, combined with an attempt to run the network with a static set of unchanging network protocols. This has been shown to be impossible and one can almost feel the pressure for protocol change building. We simply propose to explicitly deal with the changes rather keep trying to hold back the flood.

Of almost equal importance is the observation that TCP is a protocol and not a platform for implementing other protocols. Because of a lack of any alternatives, TCP has become a de-facto platform for implementing other protocols. It provides a vague standard interface with the kernel, it runs on many machines, and has a well defined distribution path. Otherwise sane people have proposed Bounded Time TCP (an unreliable byte stream protocol), Simplex TCP (which supports data in only one direction) and Multi-cast TCP (too horrible to even consider). All of these protocols probably have their uses, but not as TCP options. The fact that a large number of people are willing to use TCP as a protocol implementation platform points to the desperate need for a protocol independent platform.

Finally, we point out that in our research we have found very little difference in the actual technical work involved with the three proposed methods of protocol modification. The amount of work involved in a backward compatible change is often more than that required for an evolutionary change or the creation of a new protocol. Even the distribution costs seem to be identical. The primary cost difference between the three approaches is the cost of getting the modification approved. A protocol modification, no matter how extensive or bizarre, seems to incur much less cost and risk. It is time to stop changing the protocols to fit our current way of thinking, and start changing our way of thinking to fit the protocols.

6. References

[1] Cheriton D., "VMTP: Versatile Message Transaction Protocol", RFC 1045, Stanford University, February 1988.

[2] Hutchinson, N., Peterson, L., Abbott, M., and S. O'Malley, "RPC in the x-Kernel: Evaluating New Design Techniques", Proceedings of the 12th Symposium on Operating System Principles, Pgs. 91-101,

December 1989.

[3] Jacobson, V., "Congestion Avoidance and Control", SIGCOMM '88, August 1988.

[4] Jacobson, V., and R. Braden, "TCP Extensions for Long-Delay Paths", RFC 1072, LBL, ISI, October 1988.

[5] Jacobson, V., Braden, R., and L. Zhang, "TCP Extensions for High-Speed Paths", RFC 1185, LBL, ISI, PARC, October 1990.

[6] O'Malley, S., Abbott, M., Hutchinson, N., and L. Peterson, "A Transparent Blast Facility", Journal of Internetworking, Vol. 1, No. 2, Pgs. 57-75, December 1990.

[7] Welch, B., "The Sprite Remote Procedure Call System", UCB/CSD 86/302, University of California at Berkeley, June 1988.

7. Security Considerations

Security issues are not discussed in this memo.

8. Authors' Addresses

Larry L. Peterson
University of Arizona
Department of Computer Sciences
Tucson, AZ 85721

Phone: (602) 621-4231
EMail: llp@cs.arizona.edu

Sean O'Malley
University of Arizona
Department of Computer Sciences
Tucson, AZ 85721

Phone: 602-621-8373
EMail: sean@cs.arizona.edu