Big Data, Transmission Errors, and the Internet

Susmit Shannigrahi
Computer Science Department
Tennessee Tech
sshannigrahi@tntech.edu

Craig Partridge
Computer Science Department
Colorado State University
craig.partridge@colostate.edu

***Abstract*—A cursory look at the Internet protocol stack shows error checking capability almost at every layer, and yet, a slowly growing set of results show that a surprising fraction of big data transfers over TCP/IP are failing. As we have dug into this problem, we have come to realize that nobody is paying much attention to the causes of transmission errors in the Internet. Rather, they have typically resorted to file-level retransmissions. Given the exponential growth in data sizes, this approach is not sustainable. Furthermore, while there has been considerable progress in understanding error codes and how to choose or create error codes that offer sturdy error protection, the Internet has not made use of this new science. We propose a set of new ideas that look at paths forward to reduce error rates and better protect big data. We also propose a new file transfer protocol that efficiently handles errors and minimizes retransmissions.**

***Index Terms*—big data, error checking, file transfer protocol**

I. Introduction

The advent of "big data" has led to an explosion in large datasets. These datasets, either in their entirety or in part, are regularly transferred over networks. These data transfers use the Internet protocol suite, referred to as TCP/IP.

The error check mechanisms in the typical Internet stack, from Ethernet or WiFi up through IP and TCP, were all designed in an era where a 1 MB file transfer was considered big. A small but growing number of studies have found modern (1GB or larger) transfers failing at alarming rates. When they encounter failures, current file transfer protocols discard what has been transferred and retransmit the entire file. In the soon-approaching Exabyte era, throwing away partially good data and starting over will soon be infeasible. Scarier is that back-of-the-envelope calculations suggest that we are fast reaching the point where errors will slip through, and users will be using bad data without knowing it.

At this moment, what we know is that there is a reliability problem, but we neither fully understand its extent and nor know what errors processes are responsible for the problem. The existing tools used to detect the failures unfortunately do not identify the source of the failures. Yet finding and understanding the source of the failures is essential to make big file transfers (which are an increasing share of the Internet and data center traffic) more reliable.

This paper briefly summarizes the studies that point to problems in large data transfer and their implications. We then discuss few possible paths forward, including a new file transfer protocol that minimizes file-level retransmissions by creating smaller segments and performing error detection and selective retransmission of the corrupted segments.

II. Puzzling Errors

A handful of recent studies and some informal reports from major data centers suggest there is a serious problem transferring large (c. 1GB and larger) data files in the Internet.

A study analyzing server logs at Lawrence Livermore National Laboratory (LLNL) [1] showed that the majority of large file transfers failed. Of 18.5 million file transfer requests, approximately 13 million requests failed to transfer a file. We performed a similar internal study of the logs of the German Climate Computing Center (DRKZ), with similar results.

Another study of four years of files transferred on the Energy Science Network (ESnet) found that 1 in every 121 file transfers (.8%) delivered a file whose file checksum did not match that of the original file [2]. Other studies point to similar problems at different scales [3] [4]. Our discussions with researchers in large cloud providers point to similar anecdotal evidence. Errors have been seen in data center networks where a large number of file transfers fail on specific paths.

These numbers are disturbing. If we use a typical 32-bit integrity check on each file, then a back-of-the-envelope calculation suggests about 1 in every 520 billion file transfer could result in an inaccurate file that slips past the integrity check. While 520 billion may sound large, we looked at a recent CDN log from a large cloud provider where users requested 10 million files in a 2-hour period. These files were requested from *one* HTTP server at *one* point of presence (translating to approximately 44 Billion files per year per server). Given this large number of files transferred every year, we are already at the risk of transferring incorrect data in critical scientific and measurement applications.

A. Possible Causes

The simplest explanation that explains all the reported cases of frequent errors is that something in the network path (a router, a firewall, a switch, a transmission link, etc.) is generating errors that the TCP checksum fails to detect. For instance, the Globus file transfers are done via FTP and have no in-progress data checks above TCP. Sometimes, such transfers are also done over UDP for performance, and UDP's checksum is optional. If the Globus transfer incorporates bad data it is detected when the received file's cryptographic hash is found to be wrong. Conversely, the LLNL transfers are

TABLE I
ERROR CHECKS AT DIFFERENT LAYERS

Layer	Error Checking	Typical Error Cause
Transport	16-bit Internet checksum on the entire packet (UDP checksums are optional)	Hardware error in router/firewall data path; error in firewall packet rewriting logic and/or failure to preserve end-to-end checksum; errors in host network adapters or their drivers
IP	16-bit Internet checksum on packet header	Error in rewriting headers in firewalls
Link	32-bit IEEE CRC of entire frame (only in some Link layers)	RF interference; poor RF receivers/filters

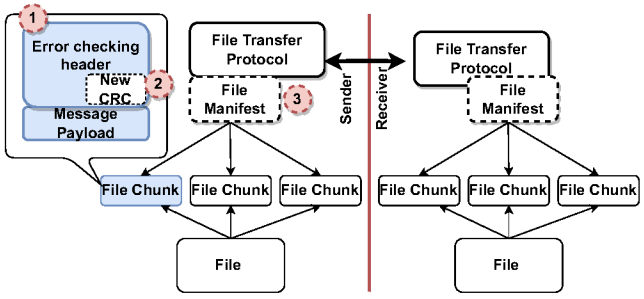

Fig. 1. Proposed File Transfer Protocol. (1) A new error-checking mechanism, (2) Error-checking capability and robustness, and (3) Performance.

done over SCP/SSH and so there is a security layer that will immediately detect errors TCP fails to detect and abort the transfer, which is the pattern the LLNL transfers exhibit.

This explanation assumes that there is a single source of errors. If we admit to multiple possibilities, then disk errors become a serious risk (and our understanding is the Globus community believes such changes may cause these errors).

B. Error Checking in the TCP Stack

For the rest of this paper we will assume the problem lies in errors slipping past TCP's checksum. This explanation has merit since it explains all the previously mentioned error studies. Furthermore, it has been known for over two decades that the TCP checksum is a poor error check [5], [6].

Table I shows the different forms of error checking that take place in TCP and the layers below it in the network stack. The table also lists the most likely causes of errors. A central message of Table I is that error checking in the Internet depends heavily on two error checks developed in the 1970s, for 1970s networks. The Internet checksum was chosen for its ease of computation on the 16-bit computers of the day and has limited error detection capabilities. [5]–[9]. Joseph Hammond developed IEEE CRC-32 based on studies of RF devices in the 1970s [10]. It is an excellent error check but is vulnerable to longer burst errors which have become more likely as IEEE 802 packet sizes have increased, as shown by [11] et al. and corroborated by an internal study.

III. PATHS FORWARD

In this section, we outline possible paths forward. Figure 1 shows the high-level overview that integrates the four different components needed to create a robust and performant file transfer protocol. The directions indicated by the numbers provide individual functions that form the foundation of a next-generation file transfer protocol. Direction 1 proposes a new message header that allows us to identify and measure the errors that might go past regular error checks. Direction 2 provides better error-checking functions through the introduction of new CRC algorithms. Thrusts 3, along with file segmentation, provides a performant file transfer protocol.

A. Error Measuring Messages

To determine if the TCP checksum is missing errors and if so, which errors are being missed, we need to know what kinds of errors are happening in the Internet and whether TCP's checksum is protecting bits against them. No one has looked at this problem in over 20 years [6], and the methods used then (capturing TCP traces and looking for errors) are hard to use at scale as it requires (oft-refused) permission to examine actual traffic on various networks.

Rather, we take another well-known approach, send known data between endpoints, and see what TCP errors we can observe. We take advantage of the fact that measurements from edge nodes scale as n^2, so a modest number of measurement points can reveal quite a bit about the Internet [12].

Specifically, we do an instrumented file transfer, using a customized file transfer protocol that breaks files into messages containing multiple error checks. The protocol opens a TCP connection between two endpoints and sends a sequence of large files. Each file is broken into messages which are encoded using Consistent Overhead Byte Stuffing (COBS). COBS is a highly efficient byte-stuffing algorithm that also ensures swift recovery from transmission errors that overwrite either the COBS encoding or the message delimiters [13]. Using COBS allows us to catch multiple errors in one test.

In our message format, data is wrapped by a header, a trailer, and a parity section containing both horizontal and vertical odd parity. The header and the trailer mirror each other and contain the message length, a 64-bit CRC, a message identifier and information on how to extract the parity information.

This additional information allows us to identify a range of damaged messages quickly. Due to the nature of COBS encoding, errors that damage or insert message delimiters cause partial or concatenated messages. Partial messages can be detected due to mismatching lengths and sometimes can be reassembled. Concatenated messages can be detected due to mismatched message identifiers and message lengths, and by using the trailer of the first message in the concatenation and the trailer in the last message of the concatenation, it may be possible to extract messages.

In messages where the payload is affected, the CRC points out that an error occurred during transmission. The parity bits likely point out the location of the error (up to a certain number of bit errors) in the message and the affected bits.

We will initially collect data by transferring large representative files between test machines to collect error statistics. Once the software is tested and debugged, we will deploy the software on various testbeds (e.g., NSF's FABRIC testbed) and inside different network providers' infrastructure. We will collect the errors from the traffic flowing through these networks and store them at a central location. These logs will allow us to determine the kinds of errors the networks are allowing to get through.

B. Better Error Checks

We know more about error checks such as CRCs and cryptographic hashes than we did when the Internet's error checks were designed. One possibility is to replace some of the Internet's error checks with more modern alternatives. In this section, we discuss what has been learned and how it might be leveraged.

A useful metric for an error check is how much strength adding an additional bit to the check imparts to error checking. In general, one expects an error check of length $x+1$ to catch at least twice as many errors as an error check of length x. Another way to think of the issue is that given a check function $c()$ giving a result of x bits and a packet p and a version $\hat{p}$ generated by corrupting p with some random error process, we expect that $c(p) = c(\hat{p})$ with a chance of less than (or at worst equal to) 1 in 2^x.

Multiple studies have shown that the Internet checksum fails this metric, often badly. It fails to detect transpositions of data and insertions (or deletions) of 16-bit zeros. In certain cases of combining portions of multiple packets, the 16-bit TCP checksum was shown to behave like a 10-bit sum [5].

The useful discovery of recent years is that CRCs are better than we realized. For any useful packet size (up to 248MB in length), there exist 32-bit CRCs that are robust to errors of up to 4 bits (so a Hamming distance of 4), including errors that include the CRC (i.e., the CRC is changed by the error) [14]–[16]. CRCs are also invulnerable to any contiguous bit error that is shorter than the bits in the CRC. The implication is, in many environments, a large fraction of errors will be deterministically detected by a 32-bit CRC. An internal study of WiFi errors has shown that about 40% of errors would always be caught by CRC-32.For errors not caught deterministically, CRCs are guaranteed to miss random errors with probability 1 in 2^x.

Furthermore, the community has gotten better at computing CRCs in software (see, for instance, Finkel's parallel computation of CRC-64 and Engdahl's table-free implementation of CRC-16) so that the cost of computing a CRC in software is close to that of a checksum, and far far less than the cost of a cryptographic hash [17], [18].

Cryptographic hashes, which are designed to thwart attempts by an adversary to alter a text, are mediocre error checks. They have a consistent 1 in 2^x chance of catching any error. This makes them far inferior to CRCs, which can be chosen to deterministically detect likely errors.

We observe that if the goal is to protect data from errors (rather than an adversary's attack), the preferred error check is clearly a CRC, and the cost of computing a CRC is modest enough that there is no reason to prefer a checksum such as the Internet's. A secondary observation is that if one revived the now deprecated TCP option [19] to negotiate a checksum, one could make TCP far more robust using a 32-bit or 64-bit CRC at a little performance cost.

C. Robust File Transfer

In addition to trying to learn about the current sources of errors in the Internet, it seems prudent to think about how to create file transfer protocols that are more robust to errors. We want a protocol that does not fail partway through a transfer when a bit of bad data gets past TCP and recovers if the delivered file fails validation.

Expressing this as a set of protocol requirements, we seek a file transfer protocol that: (a) is robust to TCP failing to detect errors (where by robust we mean continues along a path to completing the transfer); (b) is robust to receiving bad file data; (c) is robust to attacks by adversaries; and (d) delivers files efficiently (where by efficiently we mean with a minimum of retransmitted data and in a timely fashion).

The first three requirements are interconnected. If a protocol is to be robust to TCP's failure to detect errors, it must either provide a mechanism to validate the received file or a layer above TCP that catches errors in data in flight before data is delivered to the file's receiver (or both). One good way to detect errors in flight above TCP is to add a security layer that signs individual chunks of data and validates them. A security layer also provides a measure of protection against adversaries.

The requirement to deliver the file efficiently makes the protocol more complicated. The requirement to deliver the file, in spite of errors being found, means that it has to be possible to restart at least part of the file transfer.

There are two possible scenarios due to an error. The first possibility is the TCP connection breaks mid-transfer as a result of the error (or the security layer shuts the connection down on error) and the file transfer must be restarted with a new TCP connection. Another possibility is it is found that the received file is incorrect and the file needs to be fixed.

In either scenario, to deliver the file efficiently, the file transfer protocol must be able to deal with a partial file at the receiver and only request those parts that are missing or in error. Transmitting substantial but already-received chunks of a gigabit file is clearly wasteful and will be even more wasteful as petabyte and exabyte files become more common.

Generally, when network bandwidth is modest (low megabits), the best approach to delivering partial files or fixing files with errors is to exchange checksums over small parts of the file and transmitting the minimum amount of data necessary to repair or complete the file. (A slightly more sophisticated version is to use the checksums to find identical

chunks in the already received portions of the file and use those to fill in the errors or gaps [20].) In today's network, bandwidth is substantial, and the design decisions instead point to a "when in doubt, send the data" approach.

We are currently pursuing an approach based on SSH and assuming the presence of a long-term security association (e.g., via *ssh-agent*), so a new SSH connection can be invoked without user participation. Whenever an SSH connection breaks (e.g., because TCP has passed bad data to SSH), we start a new one and restart the file transfer at an appropriate point in the file. Files are also validated with a hash once transferred and repaired if in error. The challenge of figuring out the size of the unit of recovery (and, indeed, whether to determine it dynamically) remains open.

D. File Manifests

One consequence of errors in file transfers is that scientists, who want to be sure the data they copy is correct, avoid replicated copies. Rather they make sure to copy their data (sometimes more than once) from the site where the data originated, the data publisher. Clearly, this behavior scales poorly, and we would like to increase confidence in the ability to use replicated copies for data transfer.

One possibility is to use a manifest. Before the actual transfer begins, the server and the client exchange information including chunk sizes, additional metadata, and a CRC of individual chunks. The data publisher cryptographically signs this manifest. By creating a manifest of file chunks and providing individual CRCs, we are making an important improvement to current file transfer protocols. By making the CRCs available at the beginning of the transfer and signing the manifest, we can detach the file transfer protocol from a particular source. The same file located at two locations should have the same CRCs for the individual file chunks. This enables downloading files in parallel from multiple sources without compromising the file's integrity. This protocol assumes that the data or the manifest may not change during transmission. If such changes occur, the protocol must periodically check if the checksum of a fetched chunk matches the checksum in the manifest.

IV. Conclusions

Recent studies have found a disturbingly high number of errors in large data transfers. This study presents a few hypotheses about why these errors might be happening. While several studies have investigated disk and memory errors, the last large-scale study on Internet error was performed 20 years ago. This work describes how we are creating an infrastructure to address this gap systematically and plan to identify and measure errors, and perform better error checks. We propose a robust file transfer protocol that not only works around the errors but also provides novel properties such as a checksum manifest and selective retransmission of content chunks that benefits the users and reduce resource usage in the network.

Acknowledgments

This work has been supported by National Science Foundation Awards 2019163, 2126148, and 2019012.

References

[1] S. Shannigrahi, C. Fan, and C. Papadopoulos, "Request aggregation, caching, and forwarding strategies for improving large climate data distribution with ndn: a case study," in *Proc. 4th ACM Conf. Information-Centric Networking*, 2017, pp. 54–65.

[2] Z. Liu, R. Kettimuthu, I. Foster, and N. S. V. Rao, "Cross-geography scientific data transferring trends and behavior," in *Proc. 27th Intl. Symp. High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: ACM, 2018, pp. 267–278. [Online]. Available: http://doi.acm.org/10.1145/3208040.3208053

[3] R. Kettimuthu, Z. Liu, D. Wheeler, I. Foster, K. Heitmann, and F. Cappello, "Transferring a petabyte in a day," *Future Generation Computer Systems*, vol. 88, pp. 191–198, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X18302280

[4] E. Arslan and A. Alhussen, "A low-overhead integrity verification for big data transfers," in *2018 IEEE Intl. Conf. Big Data (Big Data)*, 2018, pp. 4227–4236.

[5] J. Stone, M. Greenwald, C. Partridge, and J. Hughes, "Performance of checksums and crcs over real data," *IEEE/ACM Trans. on Networking*, vol. 6, no. 5, pp. 529–543, Oct 1998.

[6] J. Stone and C. Partridge, "When the crc and tcp checksum disagree," in *Proc. Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 309–319. [Online]. Available: https://doi.org/10.1145/347059.347561

[7] J. Postel, "Transmission control protocol," Internet Requests for Comments, RFC Editor, STD 7, September 1981, http://www.rfc-editor.org/rfc/rfc793.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc793.txt

[8] R. Braden, D. Borman, C. Partridge, and W. W. Plummer, "Computing the internet checksum," Internet Requests for Comments, RFC Editor, RFC 1071, September 1988, http://www.rfc-editor.org/rfc/rfc1071.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1071.txt

[9] A. Rijsinghani, "Computation of the internet checksum via incremental update," Internet Requests for Comments, RFC Editor, RFC 1624, May 1994.

[10] J. L. J. Hammond, J. Brown, and S. Liu, "Development of a transmission error model and an error control model," Georgia Institute of Technology, Tech. Rep., 1975.

[11] B. Han, L. Ji, S. Lee, B. Bhattacharjee, and R. R. Miller, "Are all bits equal? experimental study of ieee 802.11 communication bit errors," *IEEE/ACM Trans. Netw.*, vol. 20, no. 6, p. 1695–1706, Dec. 2012. [Online]. Available: https://doi.org/10.1109/TNET.2012.2225842

[12] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis, "An architecture for large scale internet measurement," *IEEE Communications Magazine*, vol. 36, no. 8, pp. 48–54, 1998.

[13] S. Cheshire and M. Baker, "Consistent overhead byte stuffing," in *Proc. ACM SIGCOMM '97*, ser. SIGCOMM '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 209–220. [Online]. Available: https://doi.org/10.1145/263105.263168

[14] P. Koopman, "32-bit cyclic redundancy codes for internet applications," in *Proc. Intl. Conf. Dependable Systems and Networks*, 2002, pp. 459–468.

[15] P. Koopman and T. Chakravarty, "Cyclic redundancy code (crc) polynomial selection for embedded networks," in *Intl. Conf. Dependable Systems and Networks, 2004*, 2004, pp. 145–154.

[16] J. Ray and P. Koopman, "Efficient high hamming distance crcs for embedded networks," in *Intl. Conf. Dependable Systems and Networks (DSN'06)*, 2006, pp. 3–12.

[17] H. Finkel, "CRC64.h in include x2013; hpcrc64 — trac.alcf.anl.gov," https://trac.alcf.anl.gov/projects/hpcrc64/browser/include/CRC64.h?rev=f9112258851e9d448ab1d138c87252ee5ddc6773, [Accessed 21-Apr-2023].

[18] J. R. Engdahl and D. Chung, "Fast parallel crc implementation in software," in *2014 14th Intl. Conf. Control, Automation and Systems (ICCAS 2014)*, 2014, pp. 546–550.

[19] J. Zweig and C. Partridge, "TCP alternate checksum options," RFC 1145, Feb. 1990. [Online]. Available: https://www.rfc-editor.org/info/rfc1145

[20] A. Tridgell and P. Mackerras, "The rsync algorithm," The Australian National University, Tech. Rep. TR-CS-96-05, 1996.