ABSTRACT
This paper presents the design of Network Coded TCP (NCTCP). NCTCP is a reliable TCP-like transport protocol that uses network coding to dramatically improve the overall performance in networks with lossy links. By sending datagrams that represent a linear combination of packets, we are able to receive data in an orderless fashion and still push data in byte-correct order to the receiver's application layer. To eliminate round off errors we generate consecutive ones linear combinations of packets which form totally unimodular matrices. NCTCP has an Additive Increase Multiplicative Decrease (AIMD) congestion control mechanism that uses Round Trip Times to increase the performance on wireless networks without hurting the performance on wired networks. Unlike TCP, NCTCP does not use a congestion window and can be implemented in user space and hence does not need any changes to the kernel. We test our protocol using ns-2 simulator using several performance measurements, namely, throughput, goodput, efficiency and receiver efficiency. Our simulations show that NCTCP performs better than standard TCP implementations and previously proposed network coding protocols; TCP/NC and CTCP in terms of the aforementioned performance measures. We demonstrate that NCTCP is TCP-friendly and maintains the fairness property essential for the health of the Internet.

Author Keywords
TCP; Network coding; TCP friendliness; TCP fairness

ACM Classification Keywords
I.6.1 SIMULATION AND MODELING; C.2.0 Computer Communication Network

1. INTRODUCTION
The Transmission Control Protocol is one of the most used Internet protocols because of its reliable transportation between nodes. There have been several implementations aiming at improving the performance of TCP since the evolution of RFC793 in 1981. The most important modification was the introduction of congestion control and congestion avoidance techniques using the principle of self-clocking [1]. However, TCP was designed in a time long before wireless networking came into existence, and was optimized under the assumption of wired links. Specifically, the assumption that a dropped packet signifies network congestion seriously penalizes wireless communications, where packet loss can happen because of a multitude of physical phenomena and does not necessarily mean a fully saturated network [2].

Network coding is a very simple idea. Instead of sending separate packets as in traditional networks, a linear combination of packets is sent. Using linear combinations of the packets in a block is a truly novel way to compensate for lost packets. Consider TCP in the case where multiple packets are sent. If the packets arrive in order, then TCP suffers no wasted resources. However, if the first packet was lost and the rest were received, TCP would need to request the first packet again. In extreme cases, where packets are intermittently lost, TCP would use fast-retransmit. During this period, only single packets may be sent until the receiver has filled an order data arrival without causing sequence gaps, wasting valuable network resources. Transmitting datagrams of linear combinations of packets eliminates this problem as it allows for the reconstruction of lost data at the receiving side.

In this paper we present the design of Network Coded TCP (NCTCP), which uses network coding to allow out-of-order data arrival without causing sequence gaps. We design NCTCP so to overcome the disadvantages with the previous network coding protocols. Unlike other network coding protocols, NCTCP does need any change in the kernel. It maintains end-to-end semantics feature that is useful for any transport protocol. Moreover, it avoids roundoff errors during decoding data and hence guarantees the integrity of the data. NCTCP decodes the data in the correct order while improving performance.
The rest of this paper is organized as follows: In Section 2, we describe the work most closely related to this paper. In Section 3, we present our proposed network coded protocol. We first discuss the mathematical model and how we use Bareiss’s Algorithm and Sylvester’s identity to counter the problem of roundoff error. We also propose totally unimodular matrices and the consecutive-ones property to not only eliminate roundoff errors during the elimination process but also in the final solution. In Section 4, we analyze the performance of our protocol versus its predecessors, TCP/NC and CTCP as well as Reno TCP and TCP Vegas. Finally, in Section 5 the paper concludes with the gains made by NCTCP over TCP.

2. RELATED WORK

In the literature, there are two main proposals that use network coding, namely, TCP/NC [3] and CTCP [4]. Both proposals demonstrate the improvement of TCP on wireless networks in terms of throughput and efficiency. However, there are some disadvantages with both proposals.

Sundararajan et al. [3] introduced TCP/NC as a shim between the Transport and Network layers that modified TCP packets to use linear algebra. This shim would receive packets from the Transport layer, store them in a coding window, and send a modified packet to the Network layer that represented a linear combination of the packets in the coding window. Once enough packets were received to decode the linear combinations, it would report up to the Transport layer with the decoded data.

While the gains from TCP/NC were significant, it was not without its hurdles. Most problematic was the very shim layer TCP/NC was designed as. On most systems, this requires a direct modification of the OS kernel. This shim also forced a design that intentionally fools the Transport layer, rewriting packets and ACKs to account for its own information. Hence it does not maintain the end-to-end semantics of TCP. Additionally, since the coding window can potentially contain the entire data stream, the decoding matrix can quickly grow to thousands of rows and columns and become inefficient to solve.

Kim et al. [4] introduced CTCP to improve on TCP/NC. It is a modification of UDP to incorporate TCP’s congestion control and reliability while taking advantage of network coding. Since it exists as its own protocol — and not as a shim between network layers. This has the added benefit of the protocol being completely in control, as opposed to intentionally modifying information being sent to and from TCP. CTCP does not require modifying the kernel. Kim et al. [4] has implemented CTCP in user space. It also overcomes the issue of a large decoding matrix by segmenting the coding window into multiple matrices of a set size.

However, CTCP suffers from its own problems. First, it assumes packet losses spanning the entire connection must be corrected while transmitting every block, leading to a large number of needlessly retransmitted packets. Additionally, it only tracks the number of received datagrams from the block currently being decoded, which often causes retransmits of entire blocks that are not the current block. Lastly, Gaussian elimination is used when solving the linear combinations. As is well known, Gaussian elimination can suffer from severe roundoff errors [5] unless rational numbers are used [6], which introduces its own performance hit [7]. For matrices with a large dimension, this error can further compound during every iteration. While the newer computer architectures allow for greater precision of floating point numbers, it is still possible for this roundoff to affect the decoded data. Since a core feature of TCP is errorless data transmissions, it is unacceptable for a replacement protocol to not guarantee the integrity of its data.

3. NETWORK CODED TCP (NCTCP)

In this section, we present our design of the Network Coded TCP protocol. We first explain its mathematical model then we explain the details of the NCTCP sender and receiver design.

While Bareiss’s Algorithm eliminates the possibility of roundoff errors, it only guarantees integers in the matrix during the elimination process. That means the final solution of any row in the augmented matrix may still be a fraction, and may still suffer roundoff. To solve this, we introduce totally unimodular matrices. Let A be an n x n matrix. Then A is said to be totally unimodular if every square submatrix has a determinant of -1, 0, or 1. If we then append an integer vector to create our augmented matrix, it is guaranteed to have an integer solution [9].

To take advantage of this property, all that is needed is the ability to generate random rows such that the combined matrix is totally unimodular. Using the consecutive-ones property, we can do just that. Given a binary matrix, it is a consecutive-ones matrix if all the 1s on a row appear consecutively. Further, such a matrix is totally unimodular [10]. Given that a consecutive-ones matrix depends solely on the position of the 1s in a given row, we may disregard the total matrix and generate individual rows as needed. We may then store this generated row using only two integers, one for the column of the first 1 and one for the number of consecutive 1s.

Given that A requires at least n linearly independent rows to solve, we must now consider the probability that a randomly generated consecutive-ones row is
linearly dependent. Since there are $n$ columns, there are $n$ possible starting columns, $i$, and there may be up to $n - i + 1$ consecutive-ones. This implies that there are $\frac{n(n+1)}{n}$ unique rows that may be generated.

Through brute force testing, it is possible to determine the probability of any randomly generated $m \times n$ matrix will be a certain rank.

### 3.1. Sending Datagrams

Like CTCP, NCTCP stores data in packets of a fixed length, which are then divided into blocks. When performing the initial handshake, the number of blocks must be conveyed to the receiver since the sender may send a datagram representing a linear combination of the packets in any allocated block.

NCTCP, though, generate consecutive-ones linear combinations which form totally unimodular matrices. A modified version of Bareiss Algorithm is then used to efficiently decode datagrams. The data stream that is being sent is first divided into packets of fixed length, which are then grouped into blocks of $blksize$ packets. Our calculations showed that larger $n$ values lead to a reduced likelihood of a full rank matrix, we suggest limiting $blksize$ to less than 5.

The sender includes the block number that the datagram belongs to, a sequence number that represents that this is the $i$th datagram sent, and a seed for generating the coefficients with a pseudorandom number generator, to the header of every datagram.

Since reordering the rows of an augmented matrix doesn’t affect the final solution, NCTCP’s datagrams are orderless. So if any packet is lost, the next received packet is accepted as if it were the lost packet. This eliminates sequence gaps completely and the necessity for fast-retransmits.

When NCTCP sender is allowed to send a datagram — whether it is allowed to is described in Section 3.3 — it first selects the block $blkno$ to send it from using Algorithm 1, which is a modified version of the block scheduling algorithm originally used in CTCP [4].

Next, it must determine the starting column $cstart$ and length $clength$ of the consecutive-ones used to create the linear combination of packets. Finally, it gives a unique identifier $seqno$ to the linear combination. It must then transmit the $blkno$, $cstart$, $clength$, and $seqno$ in the header of the datagram, with the payload being the linear combination of the packets in $blkno$.

---

**Algorithm 1 Block Scheduling**

1: $onfly[numblks] \leftarrow 0$
2: $sent \leftarrow \text{false}$
3: for $seqno \leftarrow seqno\.una, seqno\.nxt$ do
4:     if $p = 0$ or current time $< Time(seqno) + 1.5RTT$ then
5:         $onfly[BlockNo(seqno)] \leftarrow onfly[BlockNo(seqno)] + 1$
6:     end if
7: end for
8: for $blkno \leftarrow currblk, currblk + numblks$ do
9:     if $onfly[blkno] + currdof[blkno] < blksize$ then
10:        $sent \leftarrow \text{true}$
11:        Send datagram from $blkno$ with sequence number $seqno\.nxt$
12:        $seqno\.nxt \leftarrow seqno\.nxt + 1$
13:    end if
14: end for
15: if $sent = \text{false}$ then
16:    $numblks \leftarrow numblks + 1$
17:    push 0 onto $currdof$
18: end if

---

In Algorithm 1, one can notice the use of the packet loss probability $p$ and the $onfly$ array. The $onfly$ array is set to the number of datagrams the Sender expects are still in transit for each block. In determining this number, the algorithm checks if the packet loss probability is zero and, if not, whether the datagram was sent within the last 1.5RTT. When the packet loss probability is zero, we can be confident that this datagram will eventually be received and count it as $onfly$. It is also counted as $onfly$ if it was not sent in an unreasonably long time.

The algorithm will cause the Sender to send a datagram from $currblk$ only if it has not already transmitted enough to give the Receiver a full rank block. If it shouldn’t send one from $currblk$, it will instead loop until it finds a rank deficient block. Notice the use of the $currdof$ — discussed in section Section 3.3 — array. This optimizes the number of datagrams
sent with the assumption that the Receiver only needs blksize - currdof more datagrams if it has already received currdof linearly independent datagrams. The final conditional in the algorithm may expand numblks — and allocate new blocks from which the Receiver may receive data based on network conditions. If the network allows the Sender to send a datagram, but it can’t find a block that is rank deficient, it interprets this as the network being undersaturated and dynamically increases numblks so that it may send more datagrams.

When generating the consecutive-ones linear combination, we first attempt to send unencoded datagrams. This is captured by checking the number of already sent rows: when the Sender has sent zero rows, the linear combination is given only a single 1 in the first column. When the Sender has sent one row, the linear combination is given a single 1 in the second column. This continues until it has sent blksize rows, in which case it will send a linear combination with 1s in every column. This sequence has been chosen because a single lost datagram can be seamlessly recovered. After that, all rows are randomly generated, given that the generated combination of cstart and clenath has not already been sent. In the event that all the unique linear combinations have been generated, the already sent rows is cleared, so that any combination can be resent.

3.2. Receiving Datagrams
The NCTCP Receiver must allocate numblks matrices of dimension blksize during the initial handshake. Note, however, the Receiver may receive a datagram that is more than numblks blocks ahead depending on if the Sender dynamically increases numblks. Next, it must keep track of the block number it is currently trying to decode, ack_currblk. When a datagram is received, the Receiver starts by determining the rank ack_blkdof of the matrix blkno after adding the datagram to it. If during the elimination process, a row becomes a null row, then the datagram is not linearly independent and should be discarded. If adding this datagram causes ack_currblk to become full rank, then the receiver will have modified ack_currblk into a diagonalized matrix that can be used to solve for the original stream data. When ack_currblk is successfully decoded, ack_currblk is incremented to the next block. If this block is also full rank, then it is decoded and the process repeated.

Doing this has the added benefit of trivially determining the rank of any block. Once the algorithm has been run, it prunes off any null rows. The number of remaining rows is then the rank of the block.

Regardless of whether the Receiver succeeds in decoding a block, it must reply with an ACK. This ACK must contain the ack_currblk, ack_seqno, and ack_blkdof. This differs from CTCP [4], which only returns the rank of ack_currblk, and allows the Sender to keep track of the state of the Receiver. Replying to every datagram is an idea originally proposed in [12] to untie ACKs from specific data and instead use it for network estimation as described in Section 3.3.

The Modified Bareiss Elimination we proposed takes m iterations to finish. This makes the algorithm perform in O(m³). Since we recommend that the blksize be less than 5, the time complexity of this algorithm is negligible.

In cases where the pivot element is zero, full pivoting may be used to swap a nonzero value into the pivot position. This is especially important when dealing with consecutive-ones matrices, because there is a high likelihood of zeros in the pivot elements. Full pivoting has been selected over partial pivoting since the matrix does not necessarily have multiple rows to swap, but may still have a column that is nonzero. We propose a modified version of the traditional full pivoting algorithm. Since Bareiss’s Algorithm is inherently stable, we need not find the maximum value in the matrix while pivoting, only the first nonzero value.

3.3. Receiving ACKs
NCTCP uses a modified TCP Additive Increase Multiplicative Decrease (AIMP) — designed for use in CTCP [4]. First, it does not use a congestion window to limit the number of unacknowledged packets in transit. Instead the concept of tokens is used, where a token is used in sending a datagram. The slow start phase behaves identically to traditional TCP, with every ACK increasing the number of tokens by 1. If an ACK arrives later than a preset round trip maximum RTM, the algorithm resets to slow start mode with an initial tokens value. When in congestion avoidance phase, an ACK can cause tokens to change in one of two ways. When the ACK is in response to the next datagram — i.e. this ACK is in response to the datagram that was sent directly after the last acked datagram — tokens is increased by its reciprocal. When the ACK is in response to an unexpected datagram — the expected datagram has been lost — tokens is scaled by a factor of RTTmin / RTT. This second branch is the most important, and causes tokens to adapt to actual changes in network performance. If the network is being undersaturated — a loss is not a true indication of network congestion — then RTT will be near RTTmin and tokens will be scaled back by a factor near 1. When the network is being fully saturated — losses are true indications of network congestion — then RTT will be nearly double RTTmin and tokens will be halved, meaning the update behaves akin to standard TCP’s multiplicative decrease.
Additionally upon the receipt of an ACK, the Sender must update the parameters used to estimate the network saturation as well as the internal state. If the ACK reports that the Receiver has successfully decoded currblk, it will free all the blocks until ack_currblk and shift off the ranks for the now freed blocks from currdof. Finally, the packet loss probability $p$ must be updated. The equation for updating $p$ uses a smoothing factor $\mu$, which causes $p$ to update on an exponential smoothing curve [4]. It should be noted that losses can be $\geq 0$, with larger values causing a more drastic change in $p$.

$$p \leftarrow p(1 - u)^{\text{losses}+1} + (1 - (1 - u)^{\text{losses}}$$

4. PERFORMANCE MEASUREMENTS

We analyze the performance of NCTCP using network simulator 2 (ns-2) [12]. We used four performance metrics, namely, throughput, goodput, efficiency and receiver efficiency as defined in RFC 6349 [14]. We compare the performance of NCTCP to the performance of TCP Reno, TCP Vegas and two network coding protocols, TCP/NC and CTCP.

We first analyze the performance in a wireless environment where packet loss is prominent. Each simulation sends a 16 MB file from a source to a sink, which are connected via a router with a 20 packet queue. All connections are 1 Mb/s and have a 100 ms propagation delay, and the connection between the router and sink has a variably set packet loss probability. Each packet sent has a size of 1000 bytes. All protocols are given a maximum window size of 100 packets. As we recommend in Section 3.2, a blksize of 3 was chosen for NCTCP. The TCP/NC redundancy factor is set to the reciprocal of $(1 - p)$, plus .05, as is suggested by Sundararajan et al. [3]. This redundancy was chosen to avoid large decoding matrices.

Figure 1 presents the average throughput of each protocol versus the packet loss probability. As can be seen, all protocols achieve roughly 9Mb/s when there is a 0% packet loss. As the packet loss probability increases, both Reno and Vegas immediately drop to a third of their throughput. The protocols using network coding, on the other hand, lose throughput linearly as packet loss increases. This is further illustrated by the values of the goodput against packet loss probability in Figure 2. At packet loss probabilities of 5% and greater, NCTCP shows a 1000% increase in throughput and goodput than Reno TCP—a substantial improvement.

It should be noted that these measurements do not measure time spent decoding packets due to ns-2 using discrete time increments instead of actual system time. For instance, if a packet is received at time 1.0s and the protocol takes two seconds to process before responding with an ACK, the ACK will also be sent at time 1.0s. This gives false performance measurements to TCP/NC, which spends a significant amount of time decoding. On the other hand, NCTCP decoding time is very small, even one third of the decoding time of CTCP.

It is also worth comparing the efficiency of each protocol. That is, the percentage of bytes that were not retransmitted. This allows us to analyze how much extra work the Sender performs to transmit the file. Likewise, we may analyze the receiving efficiency of the protocol, defined as the percentage of received bytes that contain new information. This allows us to determine how much undue stress was added to the network to overcome the expected packet losses.

Figure 3 shows what is to be expected: all protocols lose efficiency as the packet loss probability increases. However, CTCP’s efficiency quickly degrades, with an efficiency of 60% at just 5% packet loss probability. This is due to CTCP ignoring the rank of all but the currently decoding block, causing a large amount of retransmits. The other protocols perform with very similar efficiencies — the largest standard deviation being 5.5%. This means that all roughly retransmit the same number of packets.

Figure 4 tells a different story, though. While the protocols must retransmit more as packet loss increases, all, except CTCP, have very stable receiving efficiencies around 90%. This means that they are retransmitting as little as possible in order for the file to be transmitted. CTCP is again the outlier, though, and quickly drops to sub-70% receiving efficiencies.
Figure 1. Average Throughput vs. Packet loss probability.

Figure 2. Average Goodput vs. Packet loss Probability.

Figure 3. Efficiency vs. Packet Loss.
We analyzed the TCP friendliness property of NCTCP. This property is very crucial for the health of the Internet. To do so, we run the simulation with two sources one is running TCP Reno and the other is running NCTCP. Both sources are connected to a router via a 1Mb/s link. The router is then connected to the sink via another 1Mb/s link, and has a 20 packet queue. The simulation starts with Reno sending a 16MB file at time 1s and NCTCP begins sending a 8MB file at time 10s. Figure 5 presents the throughput achieved by Reno and NCTCP. As can be seen, both protocols oscillate around .5Mb/s, with gains happening in either protocol when the other experiences a queue overflow.
We conclude this section by demonstrating the fairness property of NCTCP. We ran the simulation with two sources but in this scenario, both are running NCTCP. Again, both sources are connected to a router via a 1Mb/s link. The router is then connected to the sink via another 1Mb/s link, and has a 20 packet queue. The simulation starts with the first NCTCP source sending a 16MB file at time 1s and the other NCTCP source begins sending a 8MB file at time 10s. Figure 6 shows that each NCTCP source gets a fair share of the throughput when sharing the router’s bandwidth with another NCTCP source.

5. CONCLUSION
In this research, we presented the design of Network Coded TCP (NCTCP), a reliable end-to-end transport protocol. NCTCP is a TCP-like protocol that uses network coding to dramatically improve overall performance in lossy networks. By sending a linear combination of packets, we are able to receive datagrams out of order and still push data in byte-correct order to the receiver’s application layer. We decode the datagrams using an efficient technique that eliminates roundoff errors. Additionally, NCTCP uses an Additive Increase Multiplicative Decrease congestion control algorithm to determine network saturation without relying on dropped packets. Through simulations, we are able to demonstrate that NCTCP achieves 1000% increases in throughput and goodput over TCP Reno, all while maintaining 99% receiving efficiencies. We also demonstrate that NCTCP has two very important properties, namely, TCP-friendliness and fairness.

REFERENCES