Sphinx: A Transport Protocol for High-Speed and Lossy Mobile Networks

Junfeng Li[1], Dan Li[1], Wenfei Wu[1], K. K. Ramakrishnan[2], Jinkun Geng[1], Fei Gui[3], Fanzhao Wang[4], Kai Zheng[4]
[1]Tsinghua University, [2]University of California, Riverside, [3]University of XiangTan, [4]Huawei Technologies

***Abstract*—Modern mobile wireless networks have been demonstrated to be high-speed but lossy, while mobile applications have more strict requirements including reliability, goodput guarantee, bandwidth efficiency, and computation efficiency. Such a complicated combination of requirements and conditions in networks pushes the pressure to transport layer protocol design. We analyze and argue that few of existing network transport layer solutions are able to handle all these requirements. We design and implement `Sphinx` to satisfy the four requirements in high-speed and lossy networks. `Sphinx` has (1) a proactive coding-based method named semi-random LT codes for loss recovery, which estimates packet loss rate and adjusts the redundancy level accordingly, (2) a reactive retransmission method named Instantaneous Compensation Mechanism (ICM) for loss retransmission, which compensates the lost packets once actual loss exceeds the estimation, and (3) a parallel coding architecture, which leverages multi-core, shared memory and kernel-bypass DPDK. Prototype and evaluation show that `Sphinx` outperforms TCP and other coding solutions significantly in microbenchmarks across all four requirements, and improves the performance of applications such as video streaming and block data transfer.**

***Index Terms*—Transport protocol, Forward error correction, Proactive and reactive hybrid protocol, Lossy mobile networks**

I. Introduction

Recent progress in mobile radio access networks focuses on improving network bandwidth (from about 50Kbps in 2G to more than 10Gbps in 5G) and coverage (e.g., small cell) [1]–[3]. One design choice to boost bandwidth is to apply higher-frequency (shorter wavelength) channels, which are able to carry more information. However, the lossy nature of a wireless network is still preserved, and this lossy nature is even exacerbated by the pursuit of bandwidth since using a short wavelength in communication makes the channel more vulnerable to noisy and complicated environments [4]. Thus, *mobile applications face a high-speed but lossy wireless network*.

Mobile applications atop these mobile networks are flourishing and inspired by the mobile Internet economy. These diverse applications tend to propose requirements on various dimensions to the underlying networks. We summarize and list these requirements as (1) reliability (for precise information delivery), (2) goodput guarantee (for data transmission completion time), (3) bandwidth efficiency (for cost saving in mobile data plan), and (4) computational efficiency (for applicability in mobile devices). All these requirements should be satisfied by the network transport layer between the applications and the mobile hardware NICs. And *all these requirements should be achieved in high-speed and lossy wireless environments.*

978-1-7281-1025-7/19/$31.00 © 2019 IEEE

However, few of the existing transport layer protocols were designed for such a complicated combination of application requirements and network environments. UDP is excluded due to its non-guarantee of reliability. Reliable transport protocols are categorized into two classes — reactive approaches and proactive ones. TCP stands for the reactive solutions, which guarantee reliability by detecting packet loss and retransmitting it. We argue this reactive approach does not suit our target scenario because the detection and retransmission cost one extra round trip time (RTT) and congestion window is also falsely adjusted, which impairs the goodput guarantee.

Proactive solutions in the other class usually provide extra redundancy together with the original packets so that even if a loss happens the receiver can still recover the content delivered. This redundancy is usually implemented by forward error correction (FEC) [5]. However, this family is still insufficient in our complicated application scenario (i.e., four application requirements and two network properties). First, some FEC schemes are computationally expensive (e.g., Reed-Solomon codes (RS codes) [6] in Loss-Tolerant TCP (LT-TCP) [7]), which limits their deployment in mobile devices. Second, and more importantly, few of FEC approaches can adapt to the actual network loss condition to adjust the redundancy level (i.e., they either use static coding parameters [6] or send by the best effort in a rateless way [8]), which violates the bandwidth efficiency requirement.

Having analyzed the insufficiency of existing solutions, we propose a new transport protocol — `Sphinx` — targeting at the requirements of *reliability, goodput guarantee, bandwidth efficiency, and computation efficiency* in *high-speed and lossy wireless networks*. `Sphinx` is a combination of both proactive and reactive ideas, and it preserves the advantages of both sides.

`Sphinx`'s proactive loss protection adopts improved LT codes, which first estimate the packet loss rate (PLR) and then adjust the redundancy level accordingly. One-RTT loss detection time is saved if packet loss happens. And `Sphinx` also conducts simple bitwise exclusive or (XOR) operation on randomly selected symbols to make it friendly to less powerful mobile devices. As this coding approach combines prediction and random coding, we name it *semi-random LT codes*.

`Sphinx` also has reactive loss protection in case that

the actual PLR exceeds the estimated PLR and the receiver side cannot decode the data. This reactive loss protection is provided by acknowledging arrived data blocks/symbols and retransmitting lost ones. We also learn from selective ACK (SACK [9]) to provide precise loss information about blocks and symbols. Thus block-level SACK (Block-SACK) and symbol-level SACK (Symbol-SACK) are designed; the former releases sender's resource and avoids the retransmission of needless redundant packets; the latter gives instantaneous feedback of loss to reduce the waiting and detecting time. And we name this reactive loss protection mechanism *instantaneous compensation mechanism (ICM)*.

Finally, we design a parallel architecture to accelerate `Sphinx` by leveraging kernel-bypass technique (DPDK [10]), shared memory and multi-core in end hosts.

We implement the prototype and evaluate `Sphinx` in an emulated high-speed and lossy network. Microbenchmark shows that `Sphinx` satisfies the four requirements significantly better than existing TCP (CUBIC [11]) and coding solutions, and each of the three design points in `Sphinx` does benefit its performance profiling. We also show two cases, i.e., data block transmission and video streaming, to demonstrate that `Sphinx` can improve application's performance significantly (e.g., $10\times$ faster than TCP in block transmission).

`Sphinx` makes the following contributions:

1) Semi-random LT codes, a proactive coding scheme that estimates network loss rate and adjusts the redundancy level accordingly for loss recovery.
2) Instantaneous Compensation Mechanism (ICM), a reactive retransmission method that compensates the lost packets promptly once actual loss exceeds the estimation.
3) Parallel coding architecture, which improves scalability and performance on multi-core platforms.

II. Background

A. Network Requirements in Mobile Networks

Network Conditions in Modern Mobile Networks. Mobile radio access network is evolving to the 5th generation (5G) now, and a significant change is that it tends to use communication channels in high frequency (short wavelength). This progress makes network bandwidth increase significantly because the high-frequency channel is able to carry more information (e.g., from less than 1Mbps in 2G to potentially exceeding 10Gbps in 5G) [1].

However, a side effect is that the communication channel becomes vulnerable due to short wavelength. Wireless networks have the nature to be lossy, and this is amplified with the wavelength being smaller. The wireless signal transmission faces the problem of fading (the signal strength decreases with distance), obstacles (the signal is reflected), and interference (a signal overlaps with itself or some other signals). Moreover, mobile devices would not likely to be static in the space, and the motion itself would disturb the data transmission [4]. In summary, in modern mobile (radio) networks, applications are facing a high-bandwidth but lossy network environment.

Network Requirements from Applications. In the meanwhile, the flourish of mobile application eco-system leads to a huge amount of diverse mobile applications in users' mobile devices. Different applications would have different network requirements due to their application scenarios, and we summarize and list these requirements below:

1) **Reliability.** Applications such as cloud storage and video-on-demand may require the network transfer to be reliable, and no contents in the whole piece of data should be lost during the transmission.
2) **Goodput Guarantee.** Application payload (i.e., the original content) should be delivered quickly enough. This is particularly useful for applications with time constraints (e.g., instant messaging, live video streaming).
3) **Bandwidth Efficiency.** As in mobile networks, the mobile users pay for their consumed bandwidth. Thus, the mobile network bandwidth had better be efficiently utilized to transmit meaningful payload.
4) **Computation Efficiency.** In practical scenarios, the user mobile devices usually are not as strong as servers, which requires the computation in network stack to be within the computational capability of the hardware.

Retrospecting existing solutions (listed in the next section), few solutions can satisfy all these requirements in current mobile network conditions.

B. Related Works and State of the Art

While there are solutions proposed to solve one or several requirements above, few of them are sufficient to satisfy all the requirements above in complicated network conditions, such as modern or next-generation mobile networks.

The reliability requirement excludes connection-less solutions such as UDP, and the remaining reliable transport protocols are categorized into reactive protocols and proactive ones. The reactive protocols detect packet loss and retransmit them, and the proactive protocols transmit packets with extra redundancy so that if a loss happens, the lost packet can be recovered. We discuss existing solutions in detail below.

Reactive TCP-based Solutions. There exist a lot of TCP variants now, such as NewReno [12], SACK [9], Vegas [13], CUBIC [11], but TCP's reliability design targets packet loss caused by network congestion, not corruptions caused by low channel quality. TCP relies on acknowledgment to confirm previously transmitted data packets arriving at the receiver side, and retransmits lost packets. We argue that TCP has two insufficiencies: first, the detect-and-retransmit scheme costs at least one RTT to feedback the loss signal to the sender, which is a waste to channel bandwidth [16]; second, a lost packet would cause TCP's congestion control to slow down the sender's rate; but the packet loss may be caused by unqualified channel instead of congestion, and thus, the congestion window shrinking is unnecessary [17]. These two disadvantages cause TCP not able to guarantee the flow throughput (as well as goodput) in the lossy wireless networks.

Proactive Coding-based Solutions. There are two typical kinds of coding-based transport protocols. One is based on

TABLE I
IMPROVEMENTS OF SPHINX COMPARED WITH EXISTING SOLUTIONS

	reliability	goodput guarantee	bandwidth efficiency	computation efficiency
UDP	✗	✓	✓	✓
TCP variants [9], [11]–[13]	✓	✗	✓	✓
RS Codes [6] (LT-TCP [7])	✓	✓	✗	✗
Fountain Codes [8] (LT Codes [14], Raptor codes [15])	✓	✓	✗	✓
Sphinx	✓	✓	✓	✓

fixed-rate codes, such as Reed-Solomon codes (RS codes) [6]. RS codes make combinations of n packets to m encoded packets ($m > n$, and bandwidth efficiency is n/m), and thus, it is resistant to a loss rate of at most $(m-n)/m$. Loss-Tolerant TCP (LT-TCP) [7] is a TCP-based implementation with RS codes. LT-TCP provides extra retransmission once the actual loss rate exceeds $(m-n)/m$ to guarantee the receiver is able to decode all packets. Referring to the four requirements above, RS-code based solutions have static parameters (i.e., m and n), which may not be adaptive to the actual channel quality; that is, once the channel quality varies (or improved), there may be more redundant packets than needed. Thus, it is not bandwidth efficient. In addition, RS codes' encoding and decoding are based on the Galois Field Arithmetic, which requires special hardware or too intensive software computation for mobile devices [18]. Previous version of QUIC [19] implements a simple FEC mechanism (i.e., XOR) with a fixed code rate, whose static redundancy is also hard to adapt to varying network conditions [20].

Another type is based on rateless codes, such as fountain codes [8]. Fountain codes employ a simple XOR-based operation for coding with high computation efficiency. The sender repeats to send (randomly) encoded packets to the receiver until the receiver acknowledges that it can decode the currently transmitted chunk. LTTP [21] adopts LT codes [14], one kind of fountain codes, to achieve reliable UDP-based protocol. However, the transmission of fountain codes tends to saturate all available bandwidth with encoded packets until the acknowledgment arrives (at least one RTT later), which is hard to be bandwidth efficient.

Application Layer Solutions. There are also application layer protocols, which are built atop the transport layer; examples are Real-Time Messaging Protocol (RTMP) [22] and Dynamic Adaptive Streaming over HTTP (DASH) [23] for video streaming. These protocols focus on adjusting the data block properties (e.g., chunk size, bit rate) instead of guarantee the transmission qualities. Currently, they are built atop TCP or UDP and consequently inherit their weakness (i.e., no goodput guarantee on TCP, and no reliability on UDP).

The categorization and analysis can be summarized as Table I, and we can see that in the context of high-bandwidth but lossy wireless network, most of existing transport solutions are not able to serve the current diversifying network applications. And our goal in this paper is to design a new network transport protocol that can provide reliable, high goodput, efficient bandwidth utilization, and mobile device computationally efficient data transmission to mobile applications in high-bandwidth but lossy wireless networks.

III. SPHINX DESIGN

We combine the advantages of proactive and reactive transport protocols, and design Sphinx targeting the new requirements. We elaborate the details of Sphinx in §III-B and §III-C. Finally, if supported by devices, Sphinx can further leverage parallel processing to accelerate Sphinx.

A. Overview

Intuition. Sphinx combines the advantages of both proactive and reactive approaches. As summarized in Table I, proactive approaches do not have good bandwidth efficiency. The essential reason is because the *level of redundancy* is fixed in existing designs. We propose to make *the level of redundancy optimistically adaptive to the network conditions* (i.e., the instantaneous loss rate). (§III-B)

While the estimated redundancy level in Sphinx would only guarantee packet recovery with the best effort, there are still chances where the actual packet loss number exceeds the estimation. In this case, the receiver would not be able to decode the received packets. Thus, Sphinx adds a reactive extension. The receiver sends selective acknowledgments for each symbol and block so that the sender can timely infer the unrecovered symbols/blocks and retransmit them. (§III-C)

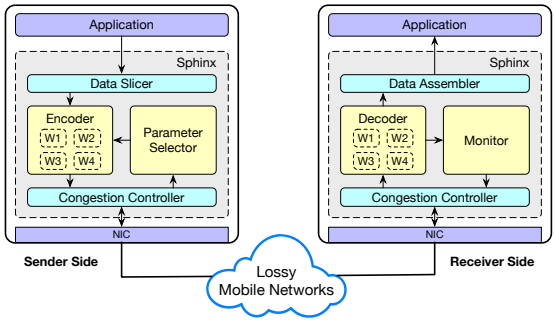

Fig. 1. Sphinx overview.

Architecture. Fig. 1 shows the architecture of Sphinx. It works as a layer between applications and NICs. On the sender side, the *Data Slicer* divides the data into several blocks, each of which is encoded by the *Encoder*. While the encoding process requires parameters, which are obtained from statistics collected by *Congestion Controller* and computed by *Parameter Selector*. *Congestion Controller* also has another function to control the sending rate.

On the receiver side, the *Decoder* recovers data blocks from encoded packets, and then, the *Data Assembler* combines and delivers them to the application layer. The *Monitor* passes the

decoding status to the *Congestion Controller*, and the latter sends feedbacks accordingly.

In addition, the coding workloads can be distributed to multiple *Workers (e.g., W1 ∼ W4)* among several cores on both sender and receiver sides.

B. Sphinx Coding (Semi-Random LT Codes)

1) Overview: One insufficiency of coding approaches is that their redundancy level fails to adapt to the actual network loss conditions. Thus, Sphinx's coding process leverages runtime measured packet loss information to guide to generate suitable redundant packets, which both guarantees the lost packet recovery and reduces bandwidth waste to the best extent. As this approach is a combination of estimation of packet loss rate (PLR) and random selection of packets for coding, we name it *semi-random LT codes*.

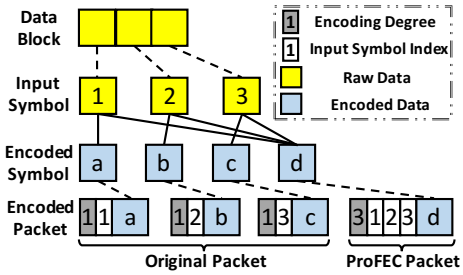

Fig. 2. An example of the coding and sending process of Sphinx.

For Sphinx coding, data is diced into blocks, and each block is divided into input symbols and transmitted individually. In a block transmission, there are two kinds of packets — the *original packets* are generated by encapsulating input symbols directly and the *ProFEC packets* (i.e., Proactive Forward Error Correction packets) are encoded packets computed from input symbols. As an example shown in Fig. 2, the data block is sliced into three input symbols. Three original packets and one ProFEC packet are generated. In each packet, there is a metadata field, including the encoding degree (i.e., the number of input symbols encoded) and the input symbol index (i.e., the index of all input symbols). To make the coding operation friendly to mobile devices, Sphinx uses simple XOR to encode packets.

2) Adaptive Encoding: Algorithm and Parameter Calculation: The algorithm of Sphinx encoding has two parameters — the encoding degree of each ProFEC packet (D_h) and the total number of ProFEC packets for a block (N_{pro}). For a block after sending all its original packets, Sphinx generates N_{pro} ProFEC packets. For each ProFEC packet, Sphinx randomly samples D_h input symbols and encodes (XOR) them into a ProFEC packet.

The D_h and N_{pro} are computed based on the PLR measured in past transmission. Assume the PLR is measured as p_{avg} with standard deviation δ in the past period, the two parameters' calculation is detailed as follows:

(1) Degree of ProFEC Packets D_h. A ProFEC packet with encoding degree D_h is encoded from D_h input symbols. It would be decodable if less than one input symbol is lost. Thus,

$$D_h \times (p_{avg} + \delta) \leq 1.$$

And we have

$$D_h = \left\lfloor \frac{1}{p_{avg} + \delta} \right\rfloor \tag{1}$$

Note that we choose $(p_{avg} + \delta)$ to denote the PLR, which overestimates the packet loss and further increases the confidence to decode packets. If PLR follows a normal distribution, $[p_{avg} - \delta, p_{avg} + \delta]$ would cover 68.2% of PLR possibilities.

(2) Number of ProFEC Packets N_{pro}. Assume one block is diced into n input symbols, and the expected number of lost original packets is $q = n \cdot p_{avg}$. In other words, q input symbols could be lost in the transmission.

2a) Sample Input Symbols to Cover Lost Symbols. For encoding process, we need to sample k input symbols (with replacement) to cover all lost symbols at least once so that the receiver can recover all lost symbols. Among the k samples, the probability where l of them are among the lost q symbols is (C represents Combination numbers):

$$C_k^l \cdot \frac{q^l \cdot (n-q)^{k-l}}{n^k} \tag{2}$$

For a certain l, the probability where each of q lost symbols is chosen for at least once is (S represents Stirling numbers) [24]:

$$\frac{q! \cdot S(l,q)}{q^l} \tag{3}$$

Thus, combining Equation 2 and 3, the probability where the k samples contain all q lost input symbols is:

$$\begin{aligned} P(k) &= \sum_{l=q}^{k} C_k^l \cdot \frac{q^l \cdot (n-q)^{k-l}}{n^k} \cdot \frac{q! \cdot S(l,q)}{q^l} \\ &= \sum_{l=q}^{k} C_k^l \cdot \frac{(n-q)^{k-l} \cdot q! \cdot S(l,q)}{n^k} \end{aligned} \tag{4}$$

We set the probability of coverage $P(k)$ with a high value (i.e., $P(K) > T$), and iterate k to find its minimum value.

2b) Encode Samples to ProFEC Packets. Finally, the all k samples would be encoded to N_{pro} ProFEC packets. Considering the PLR, we set

$$N_{pro} \times D_h \times (1 - p_{avg}) = k,$$

and get

$$N_{pro} = \left\lceil (\frac{1}{1 - p_{avg}}) \cdot \frac{k}{D_h} \right\rceil \tag{5}$$

3) Decoding: The Sphinx decoding algorithm is the same as the decoding algorithm of LT codes [14].

C. Sphinx Transmission Control (ICM and Others)

The Sphinx coding mechanism makes a tradeoff between bandwidth efficiency and payload decodability (related to goodput) with best effort, but symbol recovery failure is still possible when the actual PLR exceeds the estimated one. Thus, Sphinx introduces loss recovery to compensate the network loss. In addition, as a transport protocol, Sphinx also need to handle other transmission control issues such as congestion control (§III-C2).

1) Instantaneous Compensation Mechanism (ICM) for Loss Recovery: As data is encoded, transmitted, and decoded in the granularity of block, we considered two loss recovery options — block-level feedback or symbol-level feedback. Block-level feedback could notify the sender about the decoding status so that retransmission can be precise, but its coarse granularity makes the message of a lost block sent by its next successful block, which introduces non-trivial loss detection time (i.e., block_size/throughput). Symbol-level feedback can deliver the packet loss information quickly to the sender, but it may cause unnecessary retransmissions. The whole data transmission has redundancy (i.e., ProFEC packets) to recover a block, and some lost packets do not need retransmission.

Considering the advantages of both block-level and symbol-level feedbacks, as well as the light-weight to send feedback packets, we propose to *apply both kinds of feedbacks*. We also learn from selective ACK (SACK [9]) to provide precise loss information about blocks and symbols. Thus block-level SACK (Block-SACK) and symbol-level SACK (Symbol-SACK) are designed for precise retransmission. As this approach is an Instantaneous Compensation Mechanism for loss recovery, we name it *ICM*. The logic is elaborated as below:

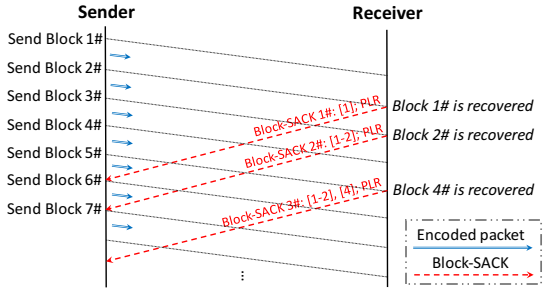

Fig. 3. A part of Block-SACKs and the workflow.

Block-SACK. Each block is indexed by a block sequence number which can be regarded as its identifier. When the receiver successfully decodes a block, a Block-SACK would be sent back to the sender. The Block-SACK contains the latest PLR and the indexes of decoded blocks. When a sender gets a Block-SACK which specifies a missing block, the sender would re-send the block immediately. Fig. 3 shows Block-SACK's format and workflow.

The most important reason of introducing Block-SACK is that it allows the sender side to release all resources (memory, timers, etc.) of recovered blocks and continue to send.

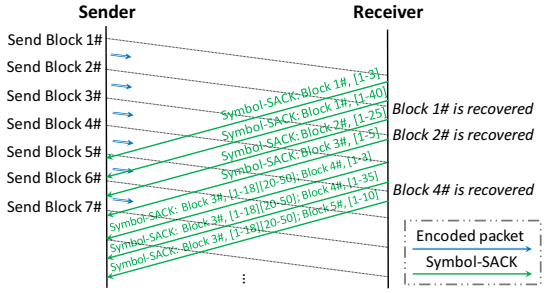

Fig. 4. A part of Symbol-SACKs and the workflow.

Symbol-SACK. `Sphinx` identifies each input symbol with its index. Symbol-SACK is sent for the successful receiving of both original packets and ProFEC packets. In the Symbol-SACK, the indexes of recovered input symbols in the most recent block and previous unrecovered blocks are attached. Fig. 4 shows Symbol-SACK's format and workflow.

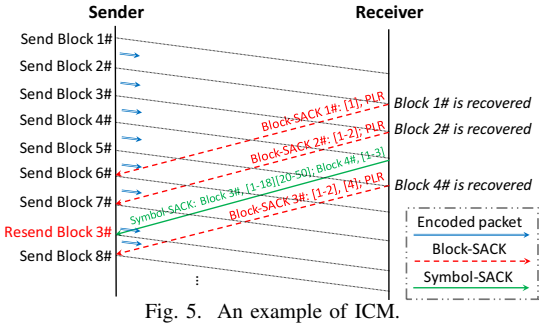

Fig. 5. An example of ICM.

When the Symbol-SACK contains the recovery status of more than two blocks and the latest block has more than three input symbols recovered, indicating that at least one block could not be recovered by the receiver, the sender will retransmit the unrecovered block immediately. To reduce redundancy as well as latency, the sender only retransmits part of original packets for missing input symbols and their corresponding FEC packets (i.e., reactive FEC packets, a.k.a., ReFEC packets). ReFEC packets are generated in a similar way as ProFEC packets. Fig. 5 shows an example where a Symbol-SACK notifies a previously missing block 3#, and the sender retransmits that block instantaneously, sooner than it receives the Block-SACK 3#.

2) Other Designs: **Feedback Loss.** Block-SACK and Symbol-SACK may be lost as well. However, the sender can infer from other feedbacks it receives. If a Block-SACK is lost, the sender can infer recovered block indexes from Symbol-SACKs. If a Symbol-SACK is missing, the sender could gather the recovery status from its successor Symbol-SACKs. Since the amount of Symbol-SACKs is large, the loss of some Symbol-SACKs has limited impact.

Continuous Sending. Unlike some existing solutions [21], when `Sphinx` completes sending one block, it does not wait for its acknowledgment but goes on to send the next one, reducing the latency, until receiving the feedback to stimulate retransmission.

Rate Control. `Sphinx` uses a congestion window to control sending rate. The window would buffer all data blocks to send and in flight, and release a block once it is acknowledged. The window is dynamically adjusted in the additive increase multiplicative decrease (AIMD) mode like TCP, i.e., the window is increased by one every RTT and decreased to a half when a loss happens.

A challenge is to distinguish the packet loss caused by congestion or low channel quality. The former requires the sending rate to slow down (i.e., decrease window); while the latter does not. In our current design, we use Explicit Congestion Notification (ECN) in the network to notify congestion, and packet loss to indicate channel low quality. But if the network does not support ECN, we would fall back to use packet loss as the signal of congestion.

D. Parallel Coding Architecture

We implement Sphinx as a layer between NIC and applications using DPDK [10]. As shown in Fig. 6, we further design a parallel coding architecture to accelerate Sphinx by leveraging shared memory and multi-core in end hosts. DPDK is supported in mobile devices [25]; the shared memory is a memory management technique that applies to mobile devices, and multi-core is applicable in a subset of mobile devices. Especially when one end of Sphinx is deployed on physical servers (e.g., video servers), the parallel architecture would boost Sphinx significantly.

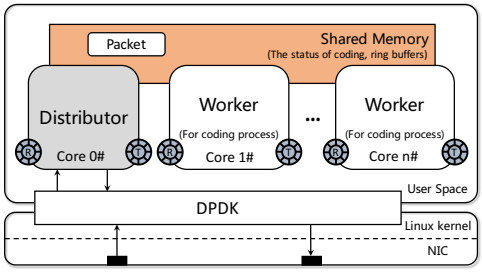

Fig. 6. The parallel coding architecture of Sphinx.

Parallel Workload Processing. There are a distributor and multiple workers, and each of them is dedicated to one core. Since the coding processes of different blocks are independent of each other, each block is assigned to a fixed worker, which avoids concurrent access conflicts. The distributor sends/receives packets and assigns packets to workers according to their block ID (a metadata field in the header). Thus, coding is processed in parallel at block level.

Shared State Management. The distributor tries to balance the workload among workers. It reads the progress of each worker and assigns new blocks to the least-loaded worker. The progress states are also stored in the shared memory. Only the worker is able to update its own states, and the distributor can only read.

Zero-copy Packet Movement. After a packet is received, it is stored in shared memory, and only the pointer to the packet is transferred between the distributor and workers through packet descriptor in a ring buffer [26]–[29]. Thus, heavy memory copy around distributor and workers is eliminated.

IV. Evaluation

We evaluate the microbenchmarks of Sphinx in terms of its bandwidth efficiency, goodput guarantee, and computation efficiency. We then show its performance metrics for applications, including data block transmission time and video streaming buffering. Results show that Sphinx outperforms various existing solutions (TCP, other coding solutions) obviously, and serves applications' requirements very well.

A. Experiment Settings

We set up our testbed to emulate a network topology. Two physical servers are directly connected by a link. Five pairs of sender and receiver are set up with senders on one server and receivers on another. We use TC netem to emulate a high-speed and lossy wireless link. In all experiments, we start traffic from senders to their corresponding receivers. Each server is equipped with Intel Core CPU i7-5930k @ 3.5GHz (6 cores), Intel 82599ES 10G NIC, and 16GB memory. Ubuntu 16.04 and DPDK 17.05 are installed.

B. Microbenchmarks

In this section, the traffic pattern is 500MB data transmission between each pair, and each data transmission is chunked into 5000 blocks if a coding-based approach is applied. The data transmission is performed by the best effort of both the sender and the receiver side, i.e., either the CPU of one side or the link capacity is saturated to 100%.

When measuring coding specific metrics, we measure and compare (all or some of) (1) semi-random LT codes in Sphinx, (2) traditional LT codes, (3) RS codes, and (4) FFT-RS codes [30]. When measuring protocol specific metrics, we measure and compare (all or some of) (1) basic Sphinx, which is a single-core implementation of §III, (2) Sphinx with traditional LT codes, which replaces semi-random LT codes in §III-B with traditional LT codes, (3) Sphinx with RS codes, which replaces semi-random LT codes with RS codes, (4) Sphinx without ICM, which removes ICM in §III-C, (5) multi-core Sphinx, which implements §III-D, and (6) original TCP (default CUBIC).

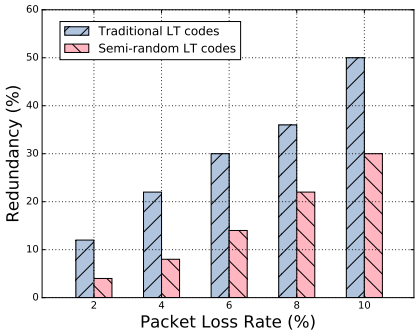

Fig. 7. The redundancy of different versions of LT codes.

Bandwidth Efficiency. The coding redundancy is quantified as the ratio of extra size introduced by encoded data and the original data size. In RS codes, this is a tunable parameter (i.e., $(m-n)/n$ in §II). Sphinx and traditional LT codes (rateless fountain codes) are two coding approaches which adapt the PLR automatically, and the result is shown in Fig. 7. We observe that the redundancy increases with the PLR since more lost packets need more redundancy to recover. And also semi-random LT codes in Sphinx introduce only about 50% redundancy of traditional LT codes (e.g., 14% vs. 30% with 6% PLR).

Computation Efficiency. We vary the length of input symbols and compare the average per-block coding time for all four coding methods. In Fig. 8, we observe that coding time increases with the length of input symbols, and semi-random LT codes outperform the RS codes and traditional LT codes significantly. For example, for symbol length of 1000 bytes, encoding a block using semi-random LT codes takes about 2ms, but it takes about 15ms for traditional LT codes and 18ms for RS codes. There are three reasons: Semi-random

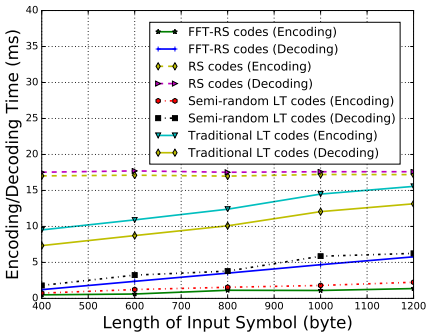

Fig. 8. The encoding/decoding time of different FEC codes with different input symbol length.

LT codes (1) do not encode original packets; (2) predict a minimum redundancy level; (3) use simple XOR operation for coding. FFT-RS codes have performance comparable with semi-random LT codes, but the application of FFT-RS codes is constrained by the requirement to perform FFT operation on symbols, which does not always hold.

Goodput Guarantee. We vary the PLR and RTT of the link, and measure the goodput of all six protocol schemes. For space reason, we only list the situations where RTT is 10ms (Fig. 9), all other scenarios show the same trend. The goodput metric is the y-axis in the figures, named Packet (after decoding) Per Second (PPS). We get three observations: (1) when there is no packet loss, TCP shows the best goodput, because it does not compute redundant packets, saving CPU and bandwidth. (2) When packet loss is introduced, coding-based schemes outperform TCP. The reason is that TCP mistakes the packet loss caused by low channel quality for that caused by congestion, and falsely reduces congestion window (as well as sending rate). While coding-based `Sphinx` always sends by the best effort without reducing sending window. (3) All the three designs, i.e., semi-random LT codes (basic `Sphinx` vs. `Sphinx` with RS codes/traditional LT codes), ICM (basic `Sphinx` vs. `Sphinx` without ICM), and parallel coding architecture (basic `Sphinx` vs. multi-core `Sphinx`), contribute to the goodput improvement. Note that the reasons of improvement are different — semi-random LT codes reduce CPU cost; ICM reduces loss detection time; and parallel coding architecture leverages more CPU resource.

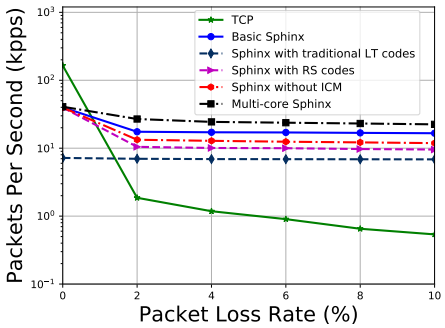

Fig. 9. Packets per second (PPS) with 10ms RTT.

C. Case Study: Data Block Transmission

Data block transmission time is a metric that directly affects the application (e.g., instant messenger, searching) performance. We still vary the PLR and RTT, and measure per-block completion time, which is named "Average Payload Unit Completion Time" (APUCT) in the y-axis of Fig. 10. The conclusion is similar to that of goodput. When the network has no loss, TCP is the most efficient; with packet loss in the network, coding-based approaches perform better. `Sphinx` has a more efficient coding design than RS codes and traditional LT codes in terms of bandwidth and CPU efficiency, and its ICM further reduces failure detection time. Multi-core parallelism is a suitable acceleration technique for `Sphinx`. As an example of `Sphinx` with all these merits, when RTT is 10ms and PLR is 6%, APUCT is about 4ms for multi-core `Sphinx` which is 25 times less than TCP (>100ms).

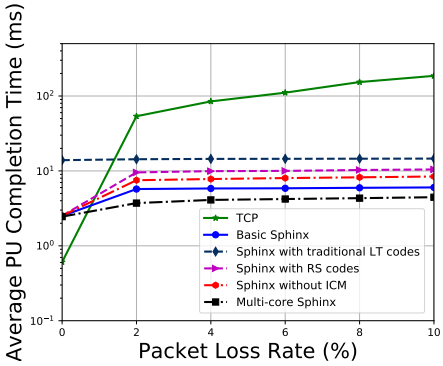

Fig. 10. Average payload unit completion time (APUCT) with 10ms RTT.

D. Case Study: Video Streaming

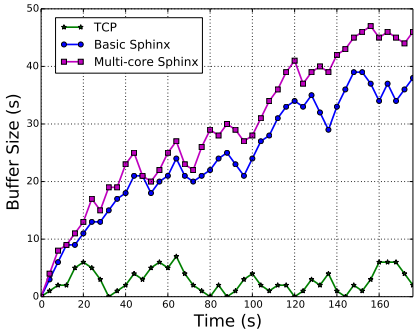

Fig. 11. The buffer size over time.

We show how `Sphinx` benefits a practical application — video streaming. We replay a video stream trace from D-LiTE [31] between a pair of sender and receiver, and the PLR and RTT of network is set to 2% and 10ms. The sender still sends with the best effort. The receiver maintains a buffer to cache received video content, and it also drains the buffer at a constant rate (the video bitrate). We measure the buffer size (i.e., buffered content length in the unit of second). Buffer accumulation indicates the transport is more efficient to fetch contents, and buffer dropping to zero means there is nothing to play at that instantaneous time and video lagging would appear, impairing user's quality of experience (QoE).

The experiment result is shown in Fig. 11. We can obviously see that basic Sphinx and multi-core Sphinx accumulate video buffers, which may provide good QoE. But TCP cannot react correctly to the lossy link, and the buffer occasionally drops to zero, causing lagging in video play. Multi-core Sphinx outperforms the basic Sphinx, but not significantly, since the amount of video data from the application layer is the bottleneck for multi-core Sphinx experiment. Given that the video data is transferred at full speed, the performance gain of multi-core Sphinx is expected to be more distinct than that of basic Sphinx.

V. Conclusion

Targeting at the requirements of reliability, goodput guarantee, bandwidth efficiency, and computation efficiency in current high-speed and lossy mobile wireless networks, we design Sphinx, which combines a proactive coding-based method with a reactive loss retransmission method. The proactive method is named semi-random LT codes, which measure and predict the network loss rate first, and then adjust the redundancy level (parameters) to use less extra bandwidth to guarantee the best loss recovery. The reactive method is named Instantaneous Compensation Mechanism (ICM), which has both Block-SACK to timely inform the sender of successful decoding and avoid unnecessary lost packet retransmission, and Symbol-SACK to provide instantaneous loss information without block-level long-time waiting. We finally design a parallel coding architecture to accelerate Sphinx by leveraging multi-core, shared memory and kernel-bypass DPDK. Our prototype and evaluation demonstrate the feasibility of Sphinx and benefits to applications compared with existing TCP (CUBIC) and coding solutions.

Acknowledgment

We thank reviewers for their valuable and insightful comments. The work was supported by the National Key Research and Development Program of China under Grant 2018YFB1800100, 2018YFB1800800, 2018YFB1800500, the Research and Development Program in Key Areas of Guangdong Province (Grant No.2018B010113001), and the National Natural Science Foundation of China under Grant No. 61432002, No. 61772305, No.61672499. Dan Li is the corresponding author of this paper.

References

[1] S. Kavanagh. (2019) How fast is 5G? [Online]. Available: https://5g.co.uk/guides/how-fast-is-5g/

[2] S. Li and et al., "5G Internet of Things: A survey," *Journal of Industrial Information Integration*, vol. 10, pp. 1–9, 2018.

[3] J. Li, D. Li, Y. Yu, and et al., "Towards full virtualization of SDN infrastructure," *Computer Networks*, vol. 143, pp. 1–14, 2018.

[4] L. Wei and et al., "Key elements to enable millimeter wave communications for 5G wireless systems," *IEEE Wireless Communications*, vol. 21, no. 6, pp. 136–143, 2014.

[5] M. Hussain and A. Hameed, "Adaptive video-aware forward error correction code allocation for reliable video transmission," *Signal, Image and Video Processing*, vol. 12, no. 1, pp. 161–169, 2018.

[6] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[7] O. Tickoo and et al., "LT-TCP: End-to-end framework to improve TCP performance over networks with lossy channels," in *International Workshop on Quality of Service*. Springer, 2005, pp. 81–93.

[8] D. J. MacKay, "Fountain codes," *IEE Proceedings-Communications*, vol. 152, no. 6, pp. 1062–1068, 2005.

[9] M. Mathis and et al., "TCP selective acknowledgment options," Tech. Rep., 1996.

[10] Intel, "Intel DPDK: Data Plane Development Kit," http://dpdk.org, 2019.

[11] S. Ha and et al., "CUBIC: a new TCP-friendly high-speed TCP variant," *ACM SIGOPS operating systems review*, vol. 42, no. 5, pp. 64–74, 2008.

[12] S. Floyd and et al., "The NewReno modification to TCP's fast recovery algorithm," Tech. Rep., 2004.

[13] L. S. Brakmo and L. L. Peterson, "TCP Vegas: End to end congestion avoidance on a global Internet," *IEEE Journal on selected Areas in communications*, vol. 13, no. 8, pp. 1465–1480, 1995.

[14] M. Luby, "LT codes," in *The 43rd Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 2002, pp. 271–280.

[15] A. Shokrollahi, "Raptor codes," *IEEE transactions on information theory*, vol. 52, no. 6, pp. 2551–2567, 2006.

[16] T. Lakshman and U. Madhow, "The performance of TCP/IP for networks with high bandwidth-delay products and random loss," *IEEE/ACM Transactions on Networking (ToN)*, vol. 5, no. 3, pp. 336–350, 1997.

[17] K. Xu and et al., "Improving TCP performance in integrated wireless communications networks," *Computer Networks*, vol. 47, no. 2, pp. 219–237, 2005.

[18] (2019) An introduction to Reed-Solomon codes: principles, architecture and implementation. [Online]. Available: https://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html

[19] A. Langley and et al., "The QUIC transport protocol: Design and Internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2017, pp. 183–196.

[20] A. Hussein and et al., "SDN for QUIC: An Enhanced Architecture with Improved Connection Establishment," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC)*. ACM, 2018, pp. 2136–2139.

[21] C. Jiang and et al., "LTTP: an LT-code based transport protocol for many-to-one communication in data centers," *IEEE Journal on Selected areas in Communications*, vol. 32, no. 1, pp. 52–64, 2014.

[22] (2019) Real-Time Messaging Protocol (RTMP) Specification. [Online]. Available: https://www.adobe.com/devnet/rtmp.html

[23] ISO/IEC 23009-1:2014. (2019) Information technology – Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats. [Online]. Available: https://www.iso.org/standard/65274.html

[24] F. Qi and et al., "A diagonal recurrence relation for the Stirling numbers of the first kind," *Applicable Analysis and Discrete Mathematics*, vol. 12, no. 1, pp. 153–165, 2018.

[25] J. Pak and K. Park, "A High-Performance Implementation of an IoT System Using DPDK," *Applied Sciences*, vol. 8, no. 4, p. 550, 2018.

[26] J. Hwang and et al., "NetVM: high performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, 2015.

[27] W. Zhang and et al., "OpenNetVM: a platform for high performance network service chains," in *Proceedings of the 2016 workshop on Hot topics in Middleboxes and Network Function Virtualization*. ACM, 2016, pp. 26–31.

[28] J. Li, D. Li, Y. Huang, and et al., "Quick NAT: High performance NAT system on commodity platforms," in *2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. IEEE, 2017, pp. 1–2.

[29] Y. Huang, J. Geng, D. Lin, and et al., "LOS: A high performance and compatible user-level network operating system," in *Proceedings of the First Asia-Pacific Workshop on Networking*. ACM, 2017, pp. 50–56.

[30] S.-J. Lin and et al., "Novel polynomial basis and its application to reed-solomon erasure codes," in *2014 IEEE 55th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2014, pp. 316–325.

[31] J. J. Quinlan and et al., "D-LiTE: A platform for evaluating DASH performance over a simulated LTE network," in *2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. IEEE, 2016, pp. 1–2.