Reliable streaming protocol for lossy networks

Mathias Brulatout, Hicham Khalife, Vania Conan, Jérémie Leguay, Emmanuel Lochin, Jérôme Lacan

▶ To cite this version:

Mathias Brulatout, Hicham Khalife, Vania Conan, Jérémie Leguay, Emmanuel Lochin, et al.. Reliable streaming protocol for lossy networks. 2015 International Wireless Communications and Mobile Computing Conference (IWCMC), Aug 2015, Dubrovnik, Croatia. pp.1486-1491, 10.1109/IWCMC.2015.7289302 . hal-02546018

HAL Id: hal-02546018
https://hal.science/hal-02546018

Submitted on 17 Apr 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Open Archive TOULOUSE Archive Ouverte (OATAO)

OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible.

This is an author-deposited version published in : http://oatao.univ-toulouse.fr/
Eprints ID : 13912

To cite this version : Brulatout, Mathias and Khalifé, Hicham and Conan, Vania and Leguay, Jérémie and Lochin, Emmanuel and Lacan, Jérôme Reliable streaming protocol for lossy networks. (In Press: 2015) In: The International Wireless Communications & Mobile Computing Conference, 24 August 2015 - 27 August 2015 (Dubrovnik, Croatia).

Any correspondance concerning this service should be sent to the repository administrator: staff-oatao@listes-diff.inp-toulouse.fr

Reliable Streaming Protocol for Lossy Networks

Mathias Brulatout, Hicham Khalifé,
Vania Conan
Thales Communications & Security
4 rue des Louvresses
92230 Gennevilliers, France
firstname.name@thalesgroup.com

Jérémie Leguay
Huawei Technologies Co. Ltd
Mathematical and Algorithmic Sciences lab
French Research Center
jeremie.leguay@huawei.com

Emmanuel Lochin, Jérôme Lacan
ISAE Toulouse
10 avenue Édouard Belin
31400 Toulouse, France
firstname.name@isae.fr

***Index Terms*—Reliable transport, linear on-the-fly coding scheme, lossy link.**

***Abstract*—This paper introduces REST, a reliable streaming protocol for lossy networks. REST ensures full reliability while recovering losses as soon as possible thanks to the proactive injection of redundancy packets encoded following an on-the-fly scheme. It dynamically adapts the sending of codes depending on the estimation of the packet error rate with periodic acknowledgments to limit feedback dependency and protocol overhead. Results show that data are smoothly delivered to the receiving application with minimum overhead when errors are uniform. For systems with limited processing capacity, we propose to use a bounded encoding window to deliver data more uniformly while limiting decoding matrices size. We study the performance of REST under different network conditions and highlight the underlying trade-offs behind each system parameter. We show that an optimal acknowledgement frequency can be estimated to minimize overhead while meeting system requirements in terms of delivery delay and computational power.**

I. INTRODUCTION

The physical layer of today's wireless communication systems such as LTE/LTE-A and Wi-Fi focuses on lossless connectivity which are achieved by accurate network planning and link budget allocation. This paper considers wireless networks where the channel is still difficult to estimate in a stable and timely manner, leading to lossy network links. Such harsh conditions can be found in public safety and military systems, preventing accurate and stable channel estimation due to long propagation delays, fast mobility and high interference.

In this context, this paper targets a particular type of streaming applications which require a full reliable service. Such applications cannot tolerate any losses and should consume data in-order as soon as possible. An example of such application in surveillance systems is the streaming of audio or video flows that should be consumed in real-time by operators but also recorded in a lossless manner for later use. Our goal is then to design a reliable streaming protocol which recovers losses as soon as possible. As lossy networks often suffer from scare resources and sometimes long delays, we also seek to limit protocol overhead and processing complexity.

To ensure a full reliable streaming service, feedback from the receiver to the source are necessary to organize the recovery of losses. However, they introduce high overhead if all packets are acknowledged, like in TCP [2], and they can also slow down the error recovery process in case of high round trip times (RTT), as the connection may stall if no feedback is received [4]. To provide full reliability in lossy networks, there is one coded based reliability mechanism which proposes to integrate the use of an on-the-fly code (RLNC [1]) in TCP. The resulting proposal, named CTCP [3], mitigates the retransmission of a full data block by sending redundancy packets. In this paper and in the context of reliable streaming over lossy networks, CTCP is not appropriate as it generates too much overhead from acknowledging every single data packet, depends too much on RTT estimation and does not deliver data smoothly to the application as it follows standard TCP behaviour.

This paper aims at filling this gap by presenting REST (Reliable Streaming for Lossy Networks), a reliable streaming protocol which ensures full reliability while repairing losses as soon as possible thanks to the proactive injection of redundancy based on an on-the-fly coding scheme. It dynamically adapts the sending of redundancy codes depending on the estimation of the packet error rate (PER) using an Exponentially Weighted Moving Average (EWMA) with periodic acknowledgments which limits feedback dependency and protocol overhead. REST builds upon the on-the-fly coding scheme of Tetrys [6] to perform reliable streaming.

For uniform errors, results show that REST leads to a smooth data delivery with minimum redundancy overhead. However, for bursty errors, the size of decoding matrices still varies in an unpredicted way. For systems with limited processing capacity, we thus propose to use a bounded encoding window to deliver data more uniformly to the application while limiting decoding matrices size. We evaluate our solution under different network conditions and study the underlying trade-offs behind each system parameter such as the frequency of acknowledgements (f_{ACK}) and the encoding window size on the protocol overhead and the data delivery speed to application. We show that an optimal f_{ACK} can be found depending on network conditions and system requirements.

This paper is structured as follows. Sec. II presents the state of the art on reliability mechanisms and compares on-the-fly coding versus block coding. Sec. III details the design of REST and Sec. IV presents its performance evaluation. Sec. V concludes this paper and discusses future work.

II. Related Work

Two mechanisms have been initially proposed to ensure reliability. A first one is based on retransmission and another one based on redundancy. The former, a reactive scheme, also known as Automatic Repeat reQuest (ARQ), ensures reliability with retransmissions. This implies a feedback from the receiver notifying a lost packet. While this is quite a simple mechanism, it is also RTT-dependant, which can be a problem in long delay and lossy networks. Hence, it is not adapted to streaming applications.

The latter, a proactive scheme known as Application-Layer Forward Error Correction (AL-FEC), introduces redundancy at the sender side. Since it is known that there will be lost packets, the sender can send more data to overcome losses. The choice of which data to send is crucial, and there is a theoretical limit, where n lost packets can be recovered using n redundancy packets. These codes use a block approach, where some redundancy is added to each fixed-size block of data. Decoding is no longer possible when you lose more than the added redundancy. On the opposite, if only few packets are lost, some of the repair packets become useless.

A solution to this problem, known as Hybrid FEC-ARQ (or H-ARQ) mechanism, is to use receiver's feedback to send additional repair packets or to adjust the redundancy level of the FEC to the observed packet loss rate. However, when using such mechanisms to achieve full reliability, large RTT might lead to very long delays to efficiently recover a packet.

On the contrary, the principle of on-the-fly codes (such as Tetrys [6] or RLNC [3], [1]) is to build repair packets from a source packets set (the coding window) which is updated with the receiver's feedback (as seen in Fig. 1). We use a parameter k which is defined as the number of data packets sent between two repair packets. Although initially RLNC was proposed in a non-systematic version while Tetrys was in a systematic one, both codes can be enabled indifferently in both modes. The main difference leads in the way the coding coefficients are computed: RLNC uses a pseudo-random generator while Tetrys uses a deterministic function based on the sequence numbers. The receiver's feedback update is done in a way that any source packet sent is included in the coding window as long as the sender does not receive its acknowledgement. The method used by the sender to generate a repair packet is simply a random linear combination over a finite field $\mathbb{F}_q$ of the data source packets belonging to the coding window. Then, the receiver performs the inverse linear combinations to recover the missing source packets from the received data and repair packets. In this approach, there is no fixed partitioning of the data stream as would be required with block codes.

In the following, we detail REST which borrows this concept of on-the-fly code to proactively inject repair packets.

III. Reliable Streaming for Lossy Networks

We present our solution, called REST, a light-weight and adaptive reliable streaming protocol. Like Tetrys [6], REST sends on-the-fly codes at an inline redundancy rate k to proactively recover packet losses and smooth data delivery.

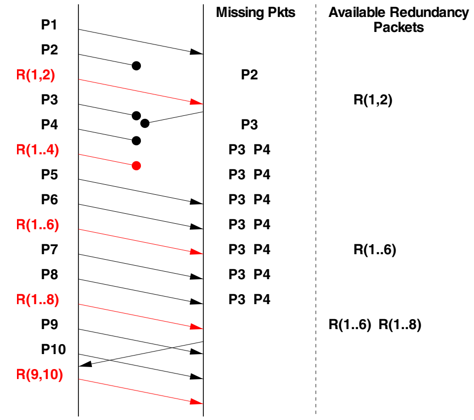


Fig. 1. Example of Tetrys with $k = 2$.

However, it aims at providing full reliability while ensuring that data are delivered continuously to the application and minimizing overhead. It can be implemented as a type II H-ARQ (incremental redundancy), a UDP transport proxy or a reliable tunnel (like RBSCP [5]). REST is made out of the three following mechanisms:

- A full reliability mechanism which ensures that all losses are recovered and that the session terminates when there is no more data to send;
- An adaptation of the inline redundancy based on a continuous packet loss rate estimation with a light-weight acknowledgment mechanism;
- A mobile encoding window to limit the impact of bursts on the data delivery speed to application and bound the decoding complexity at the receiver.

The three mechanisms are described in the three following sections. Table I lists the different notations used in this paper.

A. Full reliability

REST provides a reliability layer working in all kinds of environments, independently from the PER and the inline redundancy ratio. After sending the entire data stream once, the sender continues to send linear combination of the remaining non-acknowledged packets until they are acknowledged by the receiver to ensure reliability. Each sent packet contains a F flag that the sender sets to one to indicate the end of the flow.

TABLE I. Notations

Parameter	Meaning
Px	Data packet of index x.
$R(x, y)$	Repair packet coding packet from Px to Py.
R	Sender's inline redundancy ratio.
p	Channel's packet error rate.
k	Number of data packets between two repair packets.
W	Sender's encoding window maximum size.
f_{ack}	Receiver's acknowledgement frequency.
α	Receiver's EWMA smoothing factor.
W_{EWMA}	Receiver's EWMA window size.

Once the receiver makes the final decoding, it tells the sender using the F flag of the ACK messages to close the transport session (as seen on Fig. 2). This ensures a reliable transfer, without knowing the missing packets in the receiver's buffer and with a light-weight feedback mechanism.

B. Adaptive redundancy ratio to smooth data delivery

REST aims at delivering continuous flows of data in a reliable fashion. While lossy networks have varying behaviours, a fixed inline redundancy ratio is not appropriate. Adding more inline redundancy than necessary increases the overhead, while an insufficient inline redundancy induces potentially that huge matrices must be inverted at the receiver side, consuming massive computing resources. Adaptive inline redundancy is compulsory to smooth data delivery to the application while minimizing overhead. Depending on the channel PER p, the required inline redundancy can be estimated using Eq. 1a. The sender sends N packets with $R * N$ redundancy packets, both subject to a PER p, R being the redundancy ratio. This amount of received data should be greater than N. The value k (estimated in Eq. 1b) represents the amount of data packets between two redundancy packets.

$$(N + R \times N)(1 - p) \geq N \Leftrightarrow R \geq \frac{1}{1-p} - 1 \tag{1a}$$

$$k = \left\lfloor \frac{1}{R} \right\rfloor \tag{1b}$$

Using this equation and knowing the PER, the sender can adapt its redundancy ratio. The PER estimation is computed by the receiver and sent back to the sender in ACK messages. We use an Exponentially Weighted Moving Average (EWMA) characterized by its smoothing factor α and a time window W_{EWMA}. In our simulation, we compute this average using a time window W_{EWMA} of three seconds and a smoothing factor $\alpha = 0.8$ to give more weight to newer values, hence to have a faster reaction to varying error rates (such as bursts).

We use a light-weight periodic feedback mechanism using a constant frequency f_{ACK}. Unlike CTCP, whose receiver sends ACK for every packet, we use simple cumulative and periodic feedback reducing channel's occupation. A study of REST behaviour with a varying f_{ack} is presented in Sec. IV-E.

C. Mobile encoding window to limit decoding complexity

In order to master the computing resources at the receiver even in chaotic situations (bursty channels), decoding complexity must be bounded in some embedded systems. This can be done using a mobile encoding window of size W. Compared to the classic sliding window with a potentially infinite size, the mobile window grows until reaching its maximum size. Each repair packet is coding a maximum number of packets. When the encoding window is full, we continue to send multiple repair packets coding the same set of packets (with different coefficients). Note that, while the encoding window can be blocked for some time in case of important burst, we still continue to send new data packets in order to smooth future packet delivery at receiver side.

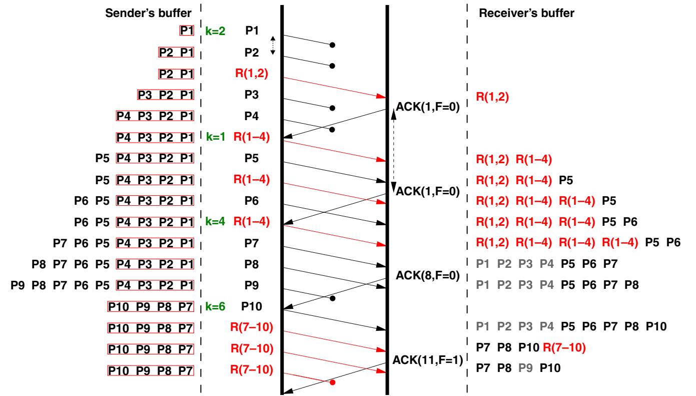


Fig. 2. A complete data exchange. Data packets are in black, repair packets are in red and recovered packets in grey. Encoding window of size $W = 4$ is the red frame in the sender's buffer.

D. A complete example

Consider the data exchange illustrated in Fig. 2 where the sender sends data at a Constant Bit Rate (CBR) and the receiver sends acknowledgements at a frequency f_{ack}. For an illustration purpose, we choose a very small encoding window size of four packets ($W = 4$). At first, the sender sends a repair packet for every two packets of data sent. Hence, $k = 2$. Receiver stores every useful repair packets until it has enough to decode (when n repair packets together coding n lost packets are received). Then, an ACK sent by the receiver notifies the sender of an increasing error rate, resulting in a higher inline redundancy ratio ($k = 1$, see III-B). Having a maximum size and having grown to its full size (see III-C), the encoding window will not change until a decoding is notified. This allows a fast decoding after receiving the repair packet $R(1, 4)$ three times. Before the last $R(1, 4)$ emission, the redundancy ratio is changed ($k = 4$) since more packets are received by the receiver. At the end of the emission, the sender still has not received the final ACK (with a F flag set to 1), thus it enters in its reliable mode (see III-A) and sends repair packets only, until the receiver sends the expected final ACK.

IV. Performance Evaluation

This section presents the performance evaluation of REST.

A. Simulation setup

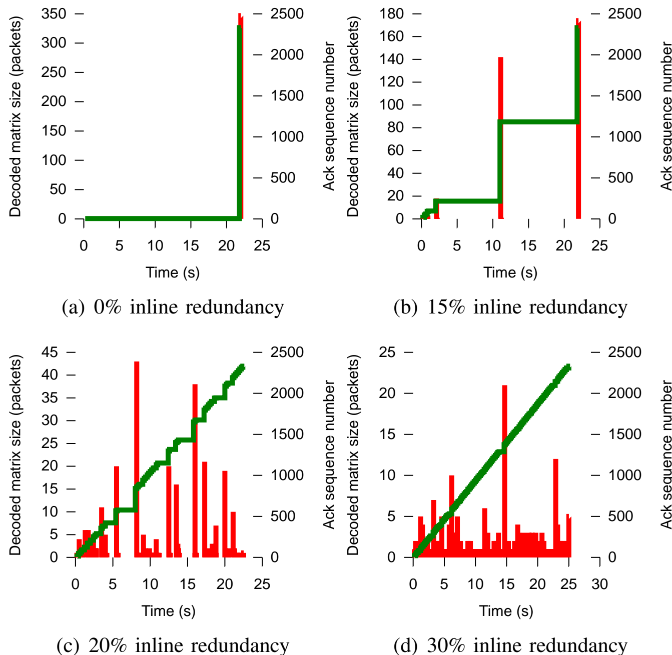
We used NS-3 to simulate the transmission of a 1MB file over a lossy network. Table II presents the different parameters used. Various packet error rate models are used to simulate different network behaviours. Note that the decoding process is simulated. As an abstraction, we suppose that all repair packets are linearly independent from each other. Hence, when receiving exactly n repair packets covering n losses, decoding is possible. This is almost always true when the coding coefficients are chosen in a sufficiently large field.

B. Reliable version with fixed inline redundancy

The goal of this simulation is to understand the underlying trade-offs in terms of overhead, decoding complexity and

TABLE II. Simulation parameters.

Parameter	Value
Link rate	10Mbps
RTT	300ms
Payload size	450bytes
CBR packet interval	0.008s
ACK interval ($1/f_{ACK}$)	0.2s



(a) 0% inline redundancy

(b) 15% inline redundancy

(c) 20% inline redundancy

(d) 30% inline redundancy

Fig. 3. Reliable version with a uniform PER of 15%. Left y-axis : Decoded matrix size (red). Right y-axis : ACK sequence number (green).

delivery time to the application. We define the overhead as the ratio of the sum of useless coded packets and acknowledgements, over the total amount of sent packets.

Fig. 3 shows how REST behaves on a 15% uniform PER link, but with different static inline redundancy ratios, respectively 0%, 15%, 20% and 30%. When the redundancy is smaller or equal than the loss rate as in Fig. 3(a) and 3(b), many missing packets are recovered at the end of the transmission, ending in a single and very costly decoding process. However, transferring time is minimum. Fig. 3(c) shows the case with 20% redundancy where multiple decoding of various sizes happen during transfer. The application receives a continuous data flow, despite some useless recovery packets received leading to a very small overhead. However, the application can receive a continuous data flow. Fig. 3(d) shows that for a high redundancy of 30%, each lost packet is quickly recovered, through very simple decoding operations. Overhead is huge and application-layer data rate is uniform.

Table III shows the total streaming duration for 1MB and the mean packet delay for the four different runs presented in Fig. 3. The packet delay is the difference between the reception and the emission of a data packet. If a packet is not lost, delay is equal to the link delay. In case of a loss, the packet delay is equal to the sum between the link delay and the time until the packet is decoded at receiver. One can see that streaming duration remains the same when the injected redundancy is less or equal than the error rate. Beyond the error rate threshold, streaming duration increases as more and more useless redundancy packets are sent. On the opposite, the packet delay decreases (mean and standard deviation) as the inline redundancy injected grows. Notice that the mean packet delay has a lower bound, which is the link delay ($RTT/2$).

TABLE III. Delivery times for the Reliable version.

Redundancy ratio	Streaming duration	Packet delay (mean)	Packet delay (std dev)
0%	21.96s	2.00s	4.91s
15%	21.96s	0.89s	2.12s
20%	22.53s	0.23s	0.32s
30%	25.01s	0.16s	0.06s

These results clearly show how the reliability and the on-the-fly decoding features work whereas a lack of adaptability from the sender side and a varying application-layer rate are obvious weaknesses to this solution. When it comes to smoothing application-layer rate (minimizing the standard deviation of packet delay), a simple solution would be to always overcome losses with a huge inline redundancy ratio hence generating a huge overhead. These results motivate the adaptive redundancy injection proposed in next section.

C. Proactive injection of adaptive redundancy

Using the redundancy adaptation presented in Sec. III-B, we achieve a uniform application-layer rate with uniform PER and almost no overhead except for a handful of packets (between 0.5 and 1% of useless sent packets). Fig. 4(a)(c) and (e) present the acknowledged sequence numbers and the size of decoded matrices over time. Fig. 4(b)(d) and (f) present the estimated error rates and the real-time inline redundancy ratio k. An infinite coding window is considered here.

As lossy networks have varying behaviours, we considered more varying packet loss distribution such as a sinusoidal one (although not very representative of any channel, it models simple variations varying from 0% to 20% with a 10% mean error rate) and a more realistic bursty model (a burst of one to 20 packets happens randomly after a loss which has 1% chance of happening). Results for these two distributions are presented in Fig. 4. For implementation purpose, we assume that an infinite k ratio is equal to 100.

One can see that two decoding of the same size occur in the descending sinus phase (one decoding for each sinus wave. See Fig. 4(c) and Fig. 4(d)). The same behaviour can be observed (with different matrix sizes) with the burst error model. When a set of burst increases the measured error rate, decoding is not possible until a return to a lower error value.

Adaptive redundancy shows great improvements in terms of overhead and matrices size reduction. However, in realistic conditions such as bursty PER distribution, matrices size re-

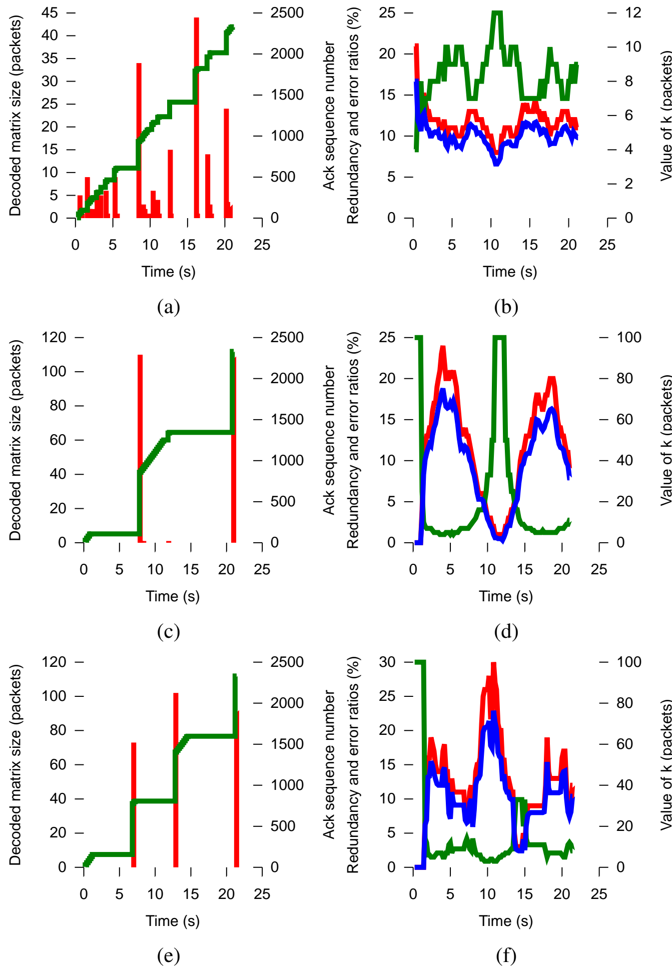


Fig. 4. Adaptive version with a uniform(a)(b), sinus(c)(d) and bursty(e)(f) PER. (a)(c) and (e) : Decoded matrix size (red) and ACK sequence number (green). (b)(d) and (f) : Left y-axis (in %) Estimated error rate (blue), Inline redundancy (red). Right y-axis k value (green).

mains potentially large, leading to a non-uniform application-layer rate.

D. *Mobile window with adaptive redundancy*

As in section IV-C, error model is based on a burst model (a burst of one to 20 packets happens randomly after a loss which has 1% chance of happening). Results in Fig. 5 represent simulations using an encoding window maximum size $W = 100$. In theory, the size of decoded matrices can be up to 100 packets (in case of a huge burst of 100 consecutive losses). In reality, their sizes evolve around $W * p$.

This solution achieves a more uniform rate at the application layer than the adaptive one in more realistic conditions; i.e. bursty networks. There is a slight overhead though, induced by useless repair packets sent. This is due to the fact that, when a block is decoded, feedback takes some time to reach its destination. Meanwhile, the sender still sends repair packets from the same window. This overhead depends on the encoding window size, the ACK frequency and the RTT (see Section IV-E for a thorough study). Even with a high RTT and a bursty PER, we provide a uniform application layer rate with minimum overhead when encoding window size is well chosen. Indeed, a small window combined with a high RTT will smooth the application-layer rate, but at a substantial overhead. Many useless repair packets will be sent by the sender over the network, while waiting for an incoming ACK which takes longer than usual to reach its destination, resulting in a significant overhead.

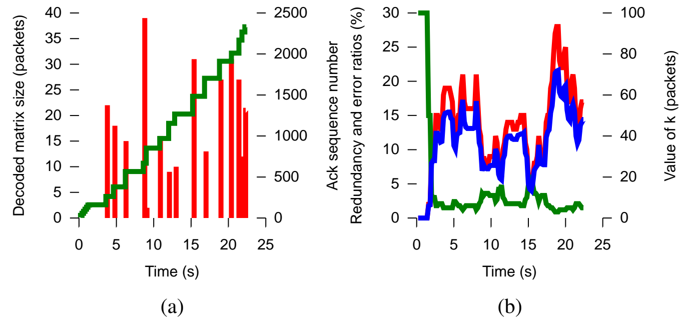
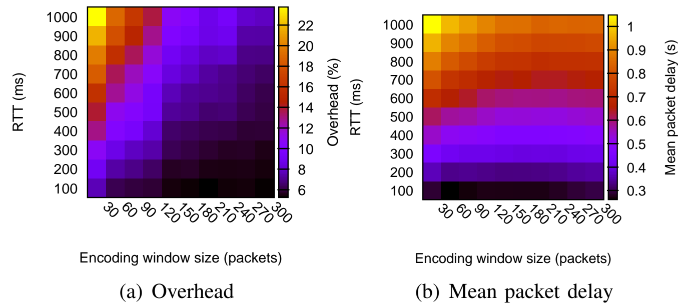


Fig. 5. Mobile Adaptive version with a bursty PER. (a) : Decoded matrix size (red) and ACK sequence number (green). (b) : Left y-axis (in %) Estimated error rate (blue), Inline redundancy (red). Right y-axis k value (green).



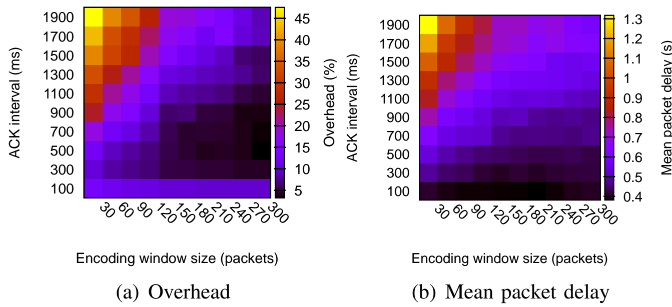
(a) Overhead (b) Mean packet delay

Fig. 6. Overhead(a) and mean packet delay(b) depending on RTT and encoding window size.

Using multiple network conditions, we study the effect of the RTT on the encoding window size. Fig. 6(a) shows that a low RTT and a large encoding window limit the overhead. A threshold can be seen beyond which, a small encoding window combined with a higher RTT induce a large overhead. This is due to the fact that many redundancy packets encoding the same data packet range are sent while the receiver's feedback takes too much time to reach the sender. Fig. 6(b) represents the mean packet delay with varying RTT and encoding window size. For a given RTT, say 500ms, it shows a minimum packet delay for sufficiently large encoding windows, 120 in this case because the maximum window size is never reached.

E. *Feedback and loss dependence*

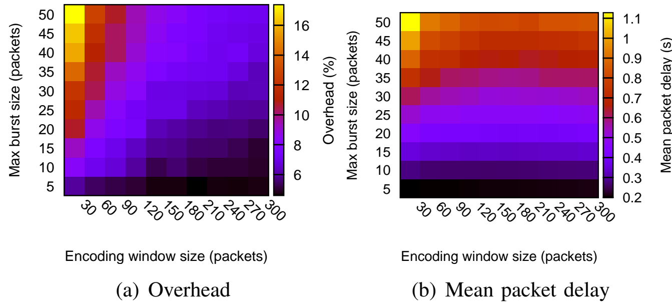
In this section, we study the impact of the ACK frequency and the error rate on the overhead and the mean packet delay. In the simulations, we set a moderate RTT of 300ms and use different f_{ACK} from 100ms to 1s. Error model is set to the most representative one, which is the burst model, with a default mean error rate of 10%.



(a) Overhead

(b) Mean packet delay

Fig. 7. Overhead(a) and mean packet delay(b) depending on f_{ACK} and encoding window size with RTT=300ms.



(a) Overhead

(b) Mean packet delay

Fig. 8. Overhead(a) and mean packet delay(b) depending on max burst size and encoding window size with RTT=300ms.

Fig. 7(a) shows the mean overhead depending on the interval between two ACKs ($1/f_{ACK}$) and the encoding window size. A high f_{ACK} (resp. a low ACK interval) introduces a constant overhead of around 10%, whatever the encoding window size. This is due to the fact that some ACK do not reduce the encoding buffer size hence make the encoding window progress. The overhead is created at the receiver side in this case. In the opposite direction, a low f_{ACK} (resp. a high ACK interval) induces a high overhead at the sender side. Fig. 7(a) shows that an optimal value for f_{ACK} exists for a given encoding window. This is clear for encoding windows larger than 30 packets. Both Fig. 6(a) and Fig. 7(a) show that the optimal value for f_{ACK} depends on network parameters such as the RTT.

Fig. 7(b) represents the mean packet delay with varying ACK intervals and encoding window size. Results show that reducing the ACK frequency while targeting a similar mean packet delay can be achieved by significantly increasing the coding window.

Fig. 8(a) shows the mean overhead depending on the maximum burst size and the encoding window size. Burst occurring probability is still the same but with varying burst size, hence a ascending PER as the max burst size grows. The same trend is shown in this figure as on Fig. 6(a) and 7(a). A lower encoding window size increases the overhead (even more when bursts can get bigger). One can notice though, that an increasing window size reduces the effect of bursts on overhead. In other words, a bigger encoding window can absorb bigger bursts in terms of overhead while increasing packet delay.

Fig. 8(b) represents the mean packet delay with varying maximum burst size and encoding window size. It shows a minimum gain with a small burst size (30 packets or less) whatever the encoding window size is because below these two points, maximum encoding window size is never reached. Setting a high burst size with a small encoding window increases the mean packet delay way too much since the encoding window is full most of the time.

To conclude on these results, there are two options depending on the system's needs. One can set a maximum delivery delay depending on the streaming application needs and figure out the minimum encoding window size (hence the minimal computational power to comply with those restrictions) and the optimal ACK frequency. One can also set a maximum window size (when computational power is limited) and seek the optimal ACK frequency to minimize the average delay to recover packets. The optimal value will depend on network conditions (RTT, losses).

V. Conclusions and Future directions

In this paper, we presented REST, a new reliable mechanism for streaming applications in challenged networks. Based on on-the-fly codes such as Tetrys and RLNC, reliability is ensured even in case of losses on the feedback channel. With the two features that are redundancy adaptation and a mobile encoding window, we limit overhead and encoding/decoding operations complexity by reducing both sender's buffer and decoded matrices sizes. Our feedback mechanism is lightweight using a suited ACK frequency. We analyzed the impact of the acknowledgement frequency and the encoding window size on network conditions and application requirements. We showed that an optimal ACK frequency can be found to minimize overhead while meeting system requirements in terms of computational limitation and application-layer target delay.

In future work, we plan to extend REST to integrate an adaptive ACK mechanism. The goal will be to choose an optimal f_{ACK} so that the overhead is minimum. As this would maximize the size of decoding matrices and thus slow down the data delivery to application, the choice of the ACK frequency should take the data delivery speed to the application as a constraint. As also seen in this paper, this frequency should be adapted to network conditions.

References

[1] M. Médard J. K. Sundararajan, D. Shah. Arq for network coding. In

[2] Van Jacobson and Michael J. Karels. Congestion avoidance and control, 1988.

[3] MinJi Kim, Jason Cloud, Ali ParandehGheibi, Leonardo Urbina, Kerim Fouli, Douglas J. Leith, and Muriel Médard. Network Coded TCP (CTCP). CoRR, abs/1212.2291, 2012.

[4] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling tcp throughput: A simple model and its empirical validation. In Proc. of ACM SIGCOMM, 1998.

[5] L.M. Surhone, M.T. Tennoe, and S.F. Henssonow. Rate Based Satellite Control Protocol. Betascript Publishing, 2011.

[6] P. U. Tournoux, E. Lochin, J. Lacan, A. Bouabdallah, and V. Roca. On-the-fly erasure coding for real-time video applications.