ΑΣΦΑΛΗΣ ΔΙΚΤΥΑΚΗ ΜΕΤΑΦΟΡΑ ΓΙΑ ΑΞΙΟΠΙΣΤΗ ΑΠΟΘΗΚΕΥΣΗ ΡΟΩΝ ΔΕΔΟΜΕΝΩΝ

Η
ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

Υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης
του Τμήματος Πληροφορικής
Εξεταστική Επιτροπή

από τον

Μελισσόβα Δημητρη

ως μέρος των Υποχρεώσεων

για τη λήψη

του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ ΣΤΑ ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ

Φεβρουάριος 2010

DEDICATION

To my family, friends and colleagues.

ACKNOWLEDGEMENTS

I would like to extend my deepest expressions of gratitude to all the people that have assisted me in the completion of this thesis.

I am most grateful to my supervisor, Prof. Stergios Anastasiadis for his guidance throughout the course of my studies and for having given me the opportunity to work on an interesting subject, such as this one.

I need to thank my parents, friends and family for their unabated support. Had it not been for their encouragement, patience and understanding, I fear that I would have been unable to step up to the challenges of life itself.

Words can scarcely describe my feelings towards my fellow colleagues at the University of Ioannina's Systems Research Group (SRG), or otherwise. If I had to make any exceptions to that rule, then I would have to personally thank (in no particular order) Andromachi Hatzieleftheriou, George Margaritis and Nikos Papanikos for essentially becoming the driving force behind my motivation at times.

I would like to thank a benefactor of Ioannina city, Christodolos Efthimiou, and the managing committee of his bequest for having elected to qualify me as their scholar and supporting my financially throughout the course of both my undergraduate and my postgraduate studies.

Finally, it should be noted that all work presented in this thesis was partly supported by the INTERSAFE project with approval number 303090/YD7631 of INTERREG IIIA Greece-Albania 2000-20006 neighbouring program co-funded by the Greek State and the European Union.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Dimitri Melissovas, MSc, Computer Science Department, University of Ioannina, Greece. February, 2010. Secure network transport protocol for reliable stream storage.

Thesis Supervisor: Stergios V. Anastasiadis.

Real-time storage of stream data is emerging as a critical component in modern computing infrastructure used for continuous monitoring purposes. Due to the sensitive nature of this kind of information, it is reasonable to assume that no single party can be trusted with it, as we run the risk of leaking it to an adversary.

Instead, much like the paradigm established by other decentralized storage systems, we use erasure-coding to disperse information geographically among nodes owned by numerous administrative entities. This upholds that single node failures or leaks do not compromise the reliability and confidentiality of a system as a whole. This may force adversaries into shifting their attention towards the network link, wherein data is more easily accessible while it is still in transit.

We propose an end-to-end protocol for remote storage of real-time stream data that combines reliability with confidentiality and integrity. In our approach we have examined a wide range of scenarios ranging from transient network failures, node crashes and attacker intervention. Therefore, the type of acknowledgement that we are after is that of a disk-commit. The primary goal for our protocol is to minimize the delay between when new data becomes available on a sender, and when it is committed on a sufficient number of storage nodes, which guarantees that it will be recoverable during playback.

We have strictly avoided batching any sort of operation on the senders. Furthermore, we have used parallel computing techniques to better utilize their resources and alleviate the computation overhead introduced by the cryptographic primitives involved in securing the transport.

Moreover, we have built on protocol on top of UDP in order to avoid the delays associated with packet loss. TCP and other protocols that seek to preserve packet ordering will simply stall already received data, until their preceding packets have arrived, for example. We have built our timeout and retransmission mechanisms for our protocol based on existing versions and extensions of them currently employed by TCP. This has allowed us to carry out some further optimizations, such as the ability to determine whether retransmitting specific timed-out packets would increase the availability of data on our system.

Senders use a simple messaging protocol to have storage nodes record new stream data on specific file offsets. The receiver uses the transport protocol to acknowledge when data has been safely committed to disk. Due to the semantics of the messaging protocol, storage nodes can process its messages in any specific order, and yet produce an identical result, in a seekable form.

Finally, we have the effects of our design choices on the efficiency of our system. Our storage nodes have exhibited good traits in their performance and can scale well to support higher bitrate streams without affecting their write latency significantly. In addition to that, in its current form, our retransmission policy reduces the effects of packet loss on the number of retransmissions and packet duplicates.

ΕΚΤΕΤΑΜΕΝΗ ΠΕΡΙΛΗΨΗ

Δημήτρης Μελισσόβας του Βασιλείου και της Παρασκευής. MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Φεβρουάριος 2010. Ασφαλής Δικτυακή Μεταφορά για Αξιόπιστη Αποθήκευση Ροών Δεδομένων.

Επιβλέπων: Στέργιος Αναστασιάδης.

Η αποθήκευση ροών δεδομένων σε πραγματικό χρόνο αποτελεί βασική υπηρεσία των σύγχρονων υπολογιστικών συστημάτων, κυρίως σε περιπτώσεις εφαρμογών παρακολούθησης. Λόγω της ευαίσθητης φύσης αυτού του τύπου της πληροφορίας, δε μπορούμε να την εμπιστευτούμε εξ' ολοκήρου σε μία αρχή, καθώς υπάρχει μεγάλη πιθανότητα να διαρρέυσει.

Για αυτό, ακολουθούμε το μοντέλο που έχει καθιερωθεί από ήδη υπάρχοντα αποκεντρικοποιημένα συστήματα αποθήκευσης και χρησιμοποιούμε κωδικοποίηση πλεονασμού για να διανείμουμε το σύνολο της πληροφορίας σε ένα σύνολο κόμβων. Κάθε ενας απο αυτούς τους κόμβους βρίσκεται σε διαφορετική τοποθεσία και ελέγχεται από διαφορετική διαχειριστική αρχή. Με αυτό συνεπάγεται ότι τόσο οι φυσικές καταστροφές, όσο και οι διαρροές πληροφορίας από μεμονωμένους κόμβους δε θέτουν σε κίνδυνο την αξιοπιστία και την ασφάλεια του συστήματος συνολικά. Για αυτό, είναι λογικό κάποιος να υποθέσει ότι ένας κακόβουλος χρήστης μάλλον θα επικεντρωθεί στο να προσπαθήσει να υποκλέψει ή και να αλλοιώσει την πληροφορία όσο αυτή διασχίζει το δίκτυο.

Προτείνουμε ένα πρωτόκολλο απομακρυσμένης αποθήκευσης ροών πραγματικού χρόνου από το ένα άκρο ως το άλλο, το οποίο συνδυάζει και παρέχει αξιοπιστία, εμπιστευτικότητα και ακεραιότητα. Κατά τη διάρκεια της μεταφοράς και

αποθήκευσης μπορούν να συμβούν σφάλματα που οφείλονται σε λάθη του δικτύου, αποτυχίες κόμβων και την κακόβουλη παρέμβαση ενός αντιπάλου. Οπότε, το μόνο είδος επιβεβαίωσης που μπορούμε να εμπιστευτούμε είναι αυτό της μόνιμης αποθήκευσης σε δίσκο. Πρωταρχικός μας στόχος στο πρωτόκολλο είναι να ελαχιστοποιήσουμε την καθυστέριση μεταξύ της εισόδου δεδομένων στους αποστολείς και την αποθήκευση σε ένα ικανοποιητικό αριθμό κόμβων αποθήκευσης, το οποίο και μας εγγυάται ότι θα μπορέσουμε να τα ανακτήσουμε.

Οι αποστολείς δεν περιμένουν να συλλέξουν παραπάνω δεδομένα, αλλα μεταδίδουν τα δεδομένα τους άμεσα. Επιπλέον έχουμε χρησιμοποιήσει τεχνικές παράλληλου υπολογισμού για να πετύχουμε καλύτερη αξιοποίηση των διαθέσιμων πόρων και να μειώσουμε την καθυστέριση που προκαλεί η κρυπτογράφηση.

Επιπλέον, έχουμε στηρίξει το πρωτόκολλο μας πάνω στο UDP, έτσι ώστε να αποφύγουμε τις καθυστερίση που θα μπορούσε να εισάγει μία απώλεια πακέτων. Πρωτόκολλα που διατηρούν εγγυήσεις ορθής σειράς παράδοσης πακέτων, όπως το TCP, καθυστερούν την παράδοση ετεροχρονισμένων πακέτων, μέχρι να λάβουν αυτά που προηγούνται. Έχουμε δημιουργήσει τον μηχανισμό επανεκπομπών του πρωτοκόλλου μας, βασιζόμενοι πάνω σε ήδη υπάρχουσα δουλειά που έχει γίνει για το TCP. Αυτό μας επέτρεψε να προβούμε σε περαιτέρω βελτιστοποιήσεις, όπως να επιτρέψουμε στο μηχανισμό να αποφανθεί εάν η επανεκπομπή ενός πακέτου θα έχει πραγματικά οφέλη για το σύστημα.

Οι αποστολείς χρησιμοποιούν ένα απλό πρωτόκολλο μηνυμάτων για να δίνουν εντολές αποθήκευσης σε συγκεκριμένες μετατοπίσεις πάνω σε ένα αρχείο. Ο παραλήπτης καταγράφει τα δεδομένα στο δίσκο και χρησιμοποιεί το πρωτόκολλο μεταφοράς για να επιβεβαιώσει την αποθήκευση. Λόγω της σημασιολογίας του πρωτοκόλλου αυτού, οι αποστολείς μπορούν να επεξεργαστούν τα μηνύματα του με οποιαδήποτε σειρά, και να πάρουν το ίδιο ακριβώς αποτέλεσμα: ένα αρχείο, στο οποίο μπορούν να αναζητήσουν οποιοδήποτε κομάτι δεδομένων με βάση κάποια μετατόπιση.

Μελετήσαμε πειραματικά την αποδοτικότητα του συστήματος μας. Οι κόμβοι αποθήκευσης μπορούν να υποστηρίζουν μεγαλύτερες ροές δεδομένων χωρίς να επηρρεάζεται ιδιαίτερα η καθυστέρηση εγγραφής τους. Επιπλέον, η παρούσα τακτική αναμεταδόσεων που χρησιμοποιούμε μετριάζει την επιρροή της απώλειας πακέτων πάνω στα ποσοστά αναμεταδόσεων και διπλότυπων.

CHAPTER 1. INTRODUCTION AND GOALS

1.1. Introduction

The state of the art in remote data storage advocates the use of erasure-coding techniques as opposed to the employment of a mere replication scheme because the application of erasure-coding algorithm yields significant benefits, such as a decrease in storage space requirements and network traffic.

Erasure coding algorithms take a set of data blocks as their input and use them to produce an additional set of coding – also referred to as parity or redundancy - blocks as their output. Both sets of blocks are then usually dispersed and recorded on a set of erasure-coded devices; disks, sites or other types, depending on the context. Provided that the number of erasures – or mechanical failures in the broader sense – does not exceed a certain threshold, we are able to recover the originally stored information. In order to avoid confusing them with other elements of this thesis, we will refer to these blocks as *shares* for the remainder of this document. Therefore, a set of data shares produces another set of coding shares.

If one were to recover the information from within the erasure-coded disks, they would have to acquire access to a sufficient number of such shares, as defined by the dispersal algorithm used. Following that, they would be capable of reconstructing the original full set of data, without any loss of information.

We believe that this particular property of erasure-coding algorithms may find good use in becoming a building block for ensuring confidentiality in a remote storage system environment. Following that notion, erasure-coding could be employed not only to improve the availability guarantees of any system at hand, but it can also be used as an effective measure against a single site leaking its portion of data to an adversary. Provided that every single site enforces its own security policy it would be nearly impossible for an attacker, insider or outsider, to compromise the security of the entire system and intercept the information stored within. Instead, it is reasonable to assume that the attacker would be forced to shift their line of focus towards capturing the data as it traverses a potentially insecure network, while it is on its way towards the storage sites.

We have examined the incorporation of that particular principle into a system which could be used to accommodate the handling of sensitive information, such as a monitoring system. In our approach, we are taking two types of sites into consideration. We define *source nodes* (or senders) of our system as the nodes responsible for streaming the captured media securely over the network. Similarly, we define *drain nodes* (also referred to as receivers and storage sites) to be the storage nodes of our system, which we assign to receive an erasure-coded portion of each serviced stream, and record it locally on their storage media.

In the real world scenario of a monitoring system, the source nodes could either be the sensors themselves, or some dedicated proxies which aggregate the individual streams of their designated set of sensors and relay the data they capture to the rest of the system. Similarly, the real world interpretation of drain nodes could easily be that of a group of organizations handling the sensitive nature of the data stream, as assigned by a higher administrative entity. The application of an erasure-coding algorithm in that context would ensure that the confidentiality of the system is not going to be compromised, even in the event that a single organization chooses to voluntarily yield its share of data to an adversary.

The design of our system is governed by several factors, such as capturing streams in real-time and providing state-of-the-art security guarantees despite the presence of

adversaries who strive to undermine the goals and the operational efficiency of our system.

Much of the work on this thesis follows the design and implementation of a transport protocol which we have tailored to fit the needs of our system by building it from scratch and using UDP as a basis. Moreover, we have pushed our modifications along with the respective bookkeeping procedures towards the receiving nodes, so as to enhance the performance and reliability of our system, overall. The outcome of our endeavor is a system that is capable of storing streams while mostly maintaining UNIX-file semantics; the stream capture routine resembles that of an append-only file, whereas it is possible to seek and read bytes from any specific location of it, during playback.

We shall begin by detailing the various goals and assumptions we have set for our system. Afterwards, we will provide insight to the theoretical background which we deem indispensible in terms of understanding our work, such as erasure-coding algorithms, cryptography as well as examples of practical applications of those primitives. Following that, we are going to detail the design and implementation of our transport protocol, step by step. In our narrative, we will 1) share the alternatives to the various design choices that have made, 2) compare our protocol to existing variations of TCP in regards to its reliability and efficiency, and 3) make a complete reference to any noteworthy occasion of borrowing security primitives from other existing protocols, such as SRTP [1], in order to accommodate our needs. Finally, we are going to 4) describe how we store data on individual drain nodes while abiding to effective information bookkeeping which is tailored towards the erasure-coded scheme.

1.2. Goals and observations

As mentioned previously, our main focus lies on designing a transport protocol which is secure and reliable, while maintaining its efficiency. Whereas, at first glance, our requirements may seem to be rather overwhelming, we make certain observations

regarding the nature of streams which have proved to be significant aides in improving the performance of our system.

We aim to design a secure and practical transport protocol, which makes the best use of the available computational and networking resources in order to ensure that the largest possible amount of information successfully reaches the storage medium in a hostile environment. We plan to do so by keeping the overhead to a minimum with the common case in mind, wherein no adversary chooses to interfere with our system. Additionally, we aim to prevent attackers from forcefully degrading the performance of our system, or any other of its guarantees.

In our analysis, we assume that an input packet has been successfully stored - or committed in the broader sense of transactional systems – once sufficient shares of it have reached the physical disks of the storage sites of our system, so that a reconstruction of its original form is possible. We have engaged in several further steps, which inevitably lead us to disqualify TCP from the basis of our transport protocol. Instead, we chose to build our protocol from scratch, borrowing ideas from TCP whenever we deemed it appropriate.

Our primary observation is that we consider a block of data to be much safer after being received by our drain nodes than simply reside in the memory of the originating source node. To that extend, our system is much better off pushing data to the network channel, as soon as becomes available, instead of stalling it so that it may be accumulated into batches. The latter could potentially improve the perceived throughput and network utilization in terms of packet header overhead, especially on the sender side, while the former is a best-effort approach towards storing data on non-volatile storage. Due to the nature of real-time streams, we should be favoring the former.

The second element in our list of observations stems from a property that all streams are subject to; streams are large *immutable* objects. The implications of this property for our system is that it has no need for versioning its objects' (i.e. streams) blocks. As a consequence, an in-order delivery of packets scheme is not required, unlike

typical file-storage systems for example, wherein proper handling of file updates and their ordering is crucial. As it will become clearer later on, applying erasure-coding algorithms carelessly on successive packets can disrupt this property under certain circumstances. We describe both the issue and an effective workaround of it on the information dispersal chapter of this thesis.

Between UDP and TCP, UDP is the only protocol of the transport layer stack which permits the application to read any packet which it has received, regardless of whether its preceding packets have arrived. This property enables the UDP to make packets available on the opposite end of a communication channel as soon as it is possible, to the point that we no longer considered TCP as a viable candidate for building the protocol upon. However, we will be referencing TCP constantly throughout this document, so as to establish a better field of comparison for our design choices.

Additionally, the use of erasure coding enables us to be more flexible whenever it comes to determining when any block of data has been committed successfully. We credit this idea to the Federated Array of Bricks (FAB). FAB is an object-based decentralized storage system wherein an object is considered to be committed once *[k + m / 2]* (upper limit) nodes agree on a specific version for it. This agreement is called an *m-quorum*, since it ensures that *k* out of any *[k + m / 2]* replies from the set of storage nodes are going to refer to the same version of the object in question, which means that the object can be reconstructed and used. While this setting reduces the fault tolerance guarantees of the system to *m / 2* node failures, it helps preserve its availability; as long as *[k + m / 2]* nodes are available, the sender node can carry on pushing objects unobstructed to the receiver nodes. Despite it being a misnomer, we are going to refer to the *[k + m / 2]* quantity itself as an *m-quorum*, for the sake of brevity.

Our system uses the *m-quorum* idea in order to advance its sending window accordingly. The main difference, however, is the fact that we used a finer granularity to define objects in our particular context. Instead of treating streams as different objects, we decided to treat the packets of each streams as different objects. Since streams are immutable, their packets are immutable as well which leads to those

objects being independent from one another. Our sender nodes, therefore, stop retransmitting those packets for which they have received *m-quorum* number of different disk-write confirmations. In effect, we use our available network resources to retransmit information in order to get adequate shares of it stored.

Finally, it is also useful to note that we are dealing with a one-way transmission scheme of stream packets to their respective destinations (i.e. drain nodes). This allows us to focus our design on reducing the processing latency at the sender-side. Additionally, it enables us to batch operations together on the drain nodes, such as disk writes and acknowledgement packet preparation, so as to improve the throughput of our system. The latter would additionally disable an adversary from clogging the performance of our system down with bursts or replayed messages.

CHAPTER 2. BACKGROUND

2.1 Erasure Coding

2.2 Message security

In this chapter, we cover the topic of erasure coding message security in the context of its practical application. The actual theoretical background behind those areas is rather extensive. We do, however, believe that our brief introduction covers these areas sufficiently for the purposes of understanding this thesis.

2.1. Erasure Coding

The main incentive for storing files on erasure-coded devices (sites or disks, depending on the context) is the ability to achieve fault-tolerance, while greatly reducing the storage space requirements when comparing them to a replication scheme. If one were to design a system that is capable of withstanding m device failures, without loss of data through replication they would have to procure $(m + 1)$ devices, each holding a full copy of the stored data. Thus, the ability to store d bytes of data through using the replication scheme would cost $(d * (m + 1))$ bytes worth of physical storage capacity to their respective owner.

An erasure-coding scheme is mainly affected by a pair of additional parameters, namely k, which stands for the desired number of data devices, and m, which defines the number of coding devices. An erasure-coding algorithm can be used to help withstand m erasures and withstand up to $m / 2$ errors. If we rule errors out, a system designer needs to procure $n = k + m$ devices, m of which can fail without compromising data integrity. Storing d bytes of data on the system now involves

splitting it evenly into k equal-sized data blocks and producing the coding blocks. The resulting blocks are then dispersed among the available devices, each committing *d / k* bytes worth of disk capacity to storing its share. Thus, applying an erasure-coding scheme to data only uses $n * d / k = (k + m) * d / k = d * (1 + m/k)$ bytes of physical storage in order to be able to withstand *m* device failures, which is considerably less to what is required by a replication scheme.

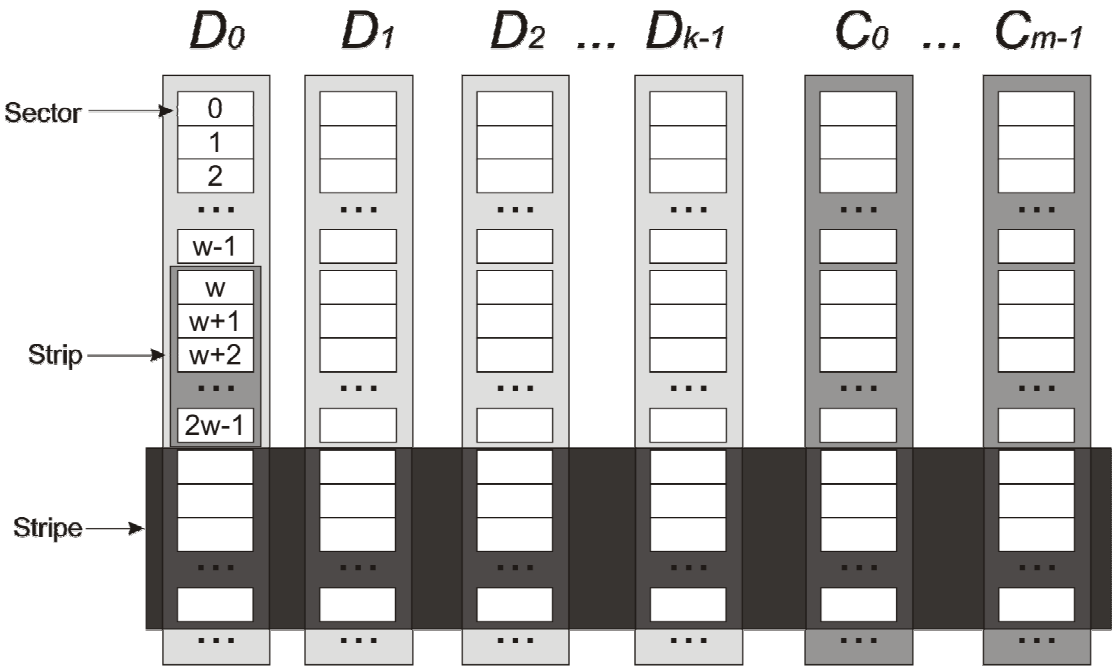

Figure 2.1 Depiction of strips, stripes and sectors on a storage system with erasure-coding. There are *k* data disks and *m* coding disks. A strip consists of w sectors, and a stripe consists of $n = k + m$ strips.

There are, of course, corner-cases wherein assigning certain values to *m* means that we can use a better-suited algorithm. If $m = 1$ we could simply do without much of the above, by simply XOR-ing the *k* data blocks together in order to produce the (one) coding block. Similarly, overcoming two site failures requires us to pick *m* so that it equals 2. In that case we are better off using the RAID-6 erasure coding algorithm, which is highly optimized for this particular setting. We believe that there is no reason to avoid picking these two algorithms when setting our system up with the aforementioned parameters, provided that we are able to distinguish the currently applied dispersal profile.

Since we could also be looking into providing a higher degree of redundancy in our system, we should also pick a suitable algorithm to cover the rest of the available options. There is a wide range of erasure-coding algorithms for one to choose from. For reasons of simplicity and our intention to keep our design clean enough, we decided to pick an optimization of the much acclaimed Reed-Solomon algorithm that is better known by the name of Cauchy Reed Solomon. It shall be neatly abbreviated as CRS for the remainder of this document.

CRS operates on fixed-sized words of length w (in bits). As far as the value for this parameter is concerned, it directly affects the size of the distribution matrix (also known as a generator matrix), which may in turn affect the computational effort needed to produce the coding blocks. The main limitation involved in setting up a proper value for w is that n should be bounded by 2^w. Certain implementations of it may additionally require that w is aligned to byte or machine word values. As far as such implementations are concerned, when w belongs to {8, 16, 32}, we can consider each collection of w bits to be a byte, short word or word respectively.

Alternatively, other implementations of CRS may simply partition the storage space of a device into strips of w sectors each. In that case, any particular value may be assigned to w as long as the sector size is byte-aligned. If the sectors have been set to one byte long, then w simply refers to the coding word byte-length instead of their bit-length, as introduced previously. Finally, the collection of strips from all devices on the array that encode and decode together is called a stripe. Figure 2.1 provides a visual depiction of these elements.

The levels through which the original Reed-Solomon has been modified in order to yield the CRS algorithm are twofold. The computation of coding blocks in vanilla Reed-Solomon consists of multiplying the Generator Matrix (*Vandermonde matrix*, in this case) with the data words in order to produce the coding words. CRS is handling its own version of the Generator Matrix differently, to allow for substituting the comparably expensive vector multiplications with XOR operations on sectors of data. In the context of CRS, the Generator Matrix, which is constructed from a *Cauchy*

Matrix, is effectively a *bitmatrix* of size *wn * wk* which maps the required XOR operations directly. A sparser bitmatrix in terms of 1's means that producing the coding blocks requires fewer XOR operations and, in effect, shall reduce the entire operation time. Fortunately, the CRS algorithm has been around long enough for highly optimized implementations of it to be freely available.

The storage sites may be dispersed geographically so that data can be protected from physical disasters. As long as no more than m sites sustain damage at any given time, data can be recovered and redistributed to these sites once they become available again. In a similar notion, one could split the storage sites to a set of different administrative entities in order to prevent captured data from being leaked. Since any particular authority can be compromised by an adversary, the confidentiality guarantees in an erasure-coded setting are significantly improved; reading data requires the consent of *k* sites, which is obviously preferable to placing one's own trust on a single entity. Additionally, an adversary would have to corrupt data on *m / 2* sites simultaneously at least, in order to prevent error-correction from restoring data. Erasure-coding may therefore be employed to provide security and data longevity against equipment failures and people, alike.

2.2. Message security

Security is closely related with the weakest link property. An adversary is assumed to be clever enough to shift their attention from a seemingly secure data storage scheme to the protocol that is employed to relay information from its source nodes to the drain sites. Using an insecure protocol for the transfer is likely to prove disastrous. Provided that anyone may ever be able to detect it, such a catastrophe can manifest itself in many different ways including, but not limited to, the entire stream being compromised the adversary and disks being overwhelmed by bogus messages or, worse yet, forged data. It is, therefore, imperative for us to employ cryptography so as to both shield the contents of its messages from an eavesdropper and protect their integrity from anyone who wishes to tamper with them.

Cryptography is an invaluable aide when designing a secure system, yet, it should only be considered as a tool and nothing more when it comes to fortifying our communication channel. Relying on cryptographic primitives alone may still leave our system exposed to certain vulnerabilities, in case these primitives are not used properly. In practice, insecure protocols are often compromised by attackers abusing security holes that are inherent to their design. As we hope to establish later on, the knowledge of the encryption and/or authentication keys is not a prerequisite for an adversary succeeding to overcome the defenses of an insecure protocol. Instead, an adversary could forge bogus messages by simply replaying old ones, provided the protocol is unable to identify them as such. Otherwise, a failure to discard those packets as duplicates compromises security, because an attacker may exploit such weaknesses to manipulate the data stored by our sites. Hence, we are going to provide background information on message security as a whole in its simplest established form.

Message security does implicate the use of cryptography. More precisely, however, we are more interested in providing a combination of block ciphers modes, message authentication codes as well as the various levels of interaction with typical message header fields such as the origin, destination and sequence number fields of a packet. We will, therefore, introduce encryption and authentication separately, in their respective sections, and collectively, in the final section of this chapter, in order to demonstrate how they can be used properly. For the sake of brevity, we will refer to the act of encrypting and authenticating as security encoding.

2.2.1. Nonces

A *nonce* is a commonly used abbreviation for number used once, and is a rather prevalent concept on both encryption and authentication alike. Nonces help prevent adversaries from replaying old messages to bypass an authentication scheme and/or using a message's predictable structure to bypass encryption and decrypt its contents.
A nonce is associated with a message and a key and should never again be used to encrypt and authenticate another message with the same key. Fortunately, protocol messages are an abundant source of such nonce values, as their headers often contain

unique combinations of values to tell different sessions and packets apart. Therefore, it is safe to use a concatenation of a unique message index, as well as the source and destination to fulfill this purpose in practice.

The only perceivable drawback with this approach is that we are required to keep the value of such fields open to the attacker, as we will establish on the encryption section of this chapter. This, however, does not give away any valuable information to an adversary; they could easily deduce the plaintext value of those fields through traffic-analysis anyway, regardless of our encryption scheme.

2.2.2. Encryption

Encryption protects a message against an eavesdropper as it traverses a communication channel. In this chapter, we are going to give an introduction on basic cryptographic primitives, such as fixed-length block ciphers and how they can be used securely in order to accommodate for the encryption of an entire message, which may be of any imaginable length.

A block cipher basically is an encryption function for fix-sized blocks. The current generation of block ciphers has a block size of 128 bits. This means that for each given secret key, the block cipher acts as a mapping which relates every possible value of a plaintext block to a unique value of ciphertext block, both blocks being 128 bits long. There are a fair number of provably secure block ciphers which one could easily choose from, such as the AES.

Since block ciphers may only operate on fixed size blocks, their application on encrypting entire messages is quite limited. One could easily claim that this may be addressed by applying a block cipher on every single block that the message in question consists of. This type of message encryption is known as the electronic codebook mode ECB and was proven to be incredibly insecure and, thus, it should be avoided at all costs. This is where real block cipher modes come into play.

2.2.2.1. ECB

In order to truly appreciate what those modes have to offer us, we are first going to discuss the security deficiencies of ECB. Given a key of a specific value, *k*, a block cipher acts as a unique one-to-one mapping of any given plaintext to a resulting *ciphertext*. If we were to couple this with the fact that we are using the same block cipher on every input block, this could yield a great deal of information to an attacker, such as all the pairs of *plaintext* blocks that equal to one another. Given that several messages abide to a highly predictable structure, ECB is bound to result in leaking entire blocks of data. This is definitely far from an ideal encryption scheme. Fortunately, it is fairly simple to avoid the pitfalls associated with the use of ECB, as we can turn to block cipher modes, instead.

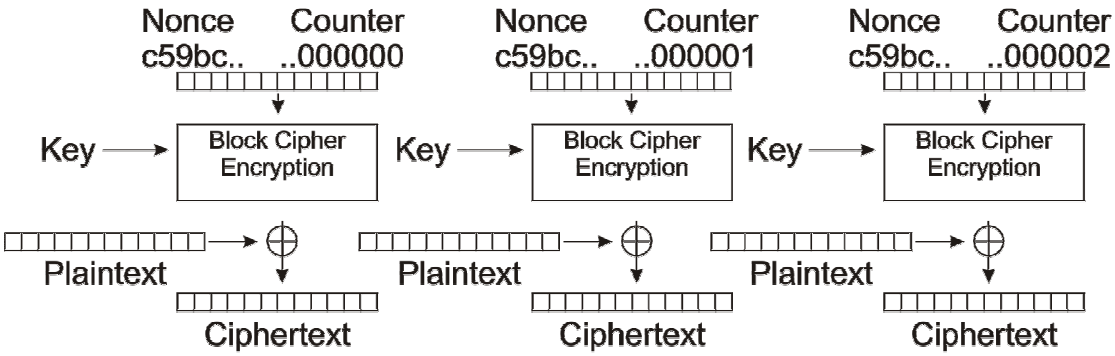

Figure 2.2 Block cipher counter mode (CTR). Each counter-generated nonce is encrypted by the underlying block cipher code to produce a new block of the keystream, each. The resulting keystream blocks are then XOR-ed directly with the plaintext blocks to produce the Ciphertext.

2.2.2.2. Block cipher modes

Whereas it is reasonable to assume that an adversary may only capture messages in the form of *ciphertext*, we should consider the possibility of someone guessing the plaintext values of certain parts of some messages. This can be accomplished through a message's predictable structure, or even through the successful injection of entire messages to the stream on the attacker's behalf. While this kind of leak is negligible the attacker may use this knowledge to achieve their ulterior motive of decrypting the entire message stream. This sequence of actions is referred to as a known plaintext attack and can be countered by using a *block cipher mode*.

Block cipher modes have one thing in common with each other. Instead of directly applying cryptographic primitives on the plaintext, modes associate every block of data with a *nonce*. A block cipher mode encrypts each of those nonce values with a given key and XORs the resulting value with the plaintext in order to produce a ciphertext block.

What sets every block cipher mode apart each other is the way they manage to compute the aforementioned streak of nonces. Those are produced in a rather systematic fashion, since nonce values need not be kept secret, and we should be using a cost-efficient means of conveying them to the receiving party, so that they may decipher the ciphertext. Thus, instead of having to relay a large amount of nonce values to the receiving counterpart, we only have to send the value used for the encryption of the very first block. That particular nonce is most commonly referred to as the Initialization Vector (IV). The manner through which the subsequent nonces are generated may, or may not take the input plaintext into account. The latter case of cipher modes is better known as stream ciphers, and the encrypted collection of nonces is referred to as a keystream.

The *block cipher counter mode* (also known as Integer Counter Mode, Figure 2.2) has a rather straightforward way of producing its stream of nonce values through an incremental counter. This mode simply encrypts consecutive counter values starting off from the initialization vector which, in turn, results to producing the entire keystream. These counter values are equal in terms of bit-length with the block size of the block cipher, and the only limitation imposed on them is that the same value may never again be used to encrypt another block, which effectively means that this value is, in fact, a nonce. This can easily be achieved by picking appropriate values for the initialization vector of each message so that the ranges of counter values used may never overlap between different messages. Despite its rather apparent operation, this mode has been proven very secure and as a consequence any vulnerability observed on it may only be attributed to that vulnerability being inherent to the block cipher used, instead.

2.2.3. Authentication

Whereas encryption is the fine art of keeping the contents of a message secret, authentication helps us certify that a message has indeed originated from the source we expected to be and that it has not been tampered with. In order to provide authentication, we use message authentication codes (MAC) in a manner which is pretty much analogous to how we use block cipher modes for encryption.

Secure hashes are the basis of message authentication codes. A hash is effectively a mapping of an arbitrarily long message m to a fixed-size output *h(m)*. There are several requirements for a hash function in order for it to be secure and, as a result, eligible for use on a message authentication code. Two of these requirements are, however, the most sought after for this kind of functions. The hash should be a *one-way function*, meaning that given a value x it is computationally hard to find a message m so that $h(m) = x$. Additionally, the hash function should be *collision-resistant*; although collisions do exist and occur when $h(m1) = h(m2)$ for two different messages, *m1* and *m2*, they can only be found through an exhaustive brute-force method.

Should we wish to have a message authenticated, we could simply run a secure hash over the concatenation of a secret key with the input message. Afterwards, we could place both the message and the resulting authentication tag on a packet and have it act as a digital signature. For example, the sender computes $h(X \,||\, m)$ and transmits $m \,||\, h(X \,||\, m)$ to the receiving end. In this case, *X* is the secret key, *m* is the message and $h(X \,||\, m)$ is the authentication tag. A receiver that knows the secret key that the message has been authenticated with, can validate its contents by running the same hash function over $X \,||\, m$ and comparing its return value to the authentication tag.

Unfortunately, the fact that real world hash functions are far off from being ideal secure hash functions would render this scheme extremely insecure, if ever used. This is part of the reason why we are holding a discussion on message authentication codes and not on secure hashes themselves. Since all well-known hash functions are iterative, an authentication protocol based exclusively off a hash function can easily become victim of length extensions or partial-message collisions. A length extension

attack enables an attacker to append an arbitrary amount of bytes to the message and update the authentication tag as they see fit and, as a result, this results in a quite successful forgery. A partial-collision, on the other hand, simply enables an attacker to substitute our message, *m1*, in the packet with the message *m2*, provided that *h(m1) = h(m2)*. This is particularly effective as their computation does not take the secret key into account, and the whole process of detecting a collision is further sped up by the birthday paradox.

The birthday paradox states that whenever 23 random people are found inside the same room, the chances that two of them have the same birthday exceed 50%. This is an alarmingly large possibility, given that there are 365 possible birthdays. In practical terms, a collision may happen much sooner than one would expect it, which is after an adversary makes roughly *sqrt(n)* guesses.

A typical message authentication code (MAC) is an algorithm which takes a message, *m*, and a secret key, *X*, as input and produces an authentication tag in a secure fashion. Since all well-known secure hash functions fall short from the ideal hash function, it is up to the MAC to eliminate those vulnerabilities and provide a secure authentication scheme. HMAC, for example, computes *f(X, m) = h(X xor a || h((X xor b) || m))* as its authentication tag, where a and b are specified constants, *m* is the message and *X* is the session key. Similar to the secure hash message authentication paradigm we previously introduced, the sender transmits *m || f(X, m)*, where *f(X, m)* is the authentication tag of the message, which the receiver uses to validate its contents.

It is useful, however, to note that we might be dealing with a devious adversary who has managed to assume control over the underlying network. They are, therefore, capable of capturing an entire stream as it traverses the network, drop packets and replay old ones as they see fit. Messages, therefore, should contain some sort of nonce so as to prevent adversaries from bypassing out authentication scheme. Typically, protocols that combine encryption with authentication run the MAC over the same fields that make up for the initialization vector of the block cipher mode. This is very similar to how network protocols detect and drop packet duplicates, with the

exception that packet index – also known as the sequence number of the packet – is authenticated as well, in order to prevent an adversary from altering its value.

2.2.4. Messaging protocol

Finally we are going to describe how encryption and authentication may be encapsulated on a secure message protocol. For this approach, we shall assume that we are dealing with a one-way communication approach and that both parties have already agreed on a symmetric secret key. The latter is usually accomplished during the initialization phase through using public key cryptography or protocols such as the Diffie-Hellman algorithm. We should also mention that an in-depth study of SRTP has been instrumental in helping us understand a great deal about applying message security. SRTP is a modification of the Real-time Transport Protocol (RTP) that is used to provide confidentiality and integrity for an RTP stream.

As discussed earlier, we need to provide the algorithm with a nonce value for each input block of our collection of messages. This burden is somewhat alleviated by the fact that we need only specify the initialization vector for the block cipher mode that we are using, and having that sent along with the encrypted message.

There is only one last precaution that we have to be wary of, which is the prevention of individual nonce values from overlapping with the nonce values selected to encrypt any other block on a message stream. Assuming that we are using the counter mode, we would have to specify the initialization vector between consecutive messages so that this scenario may never materialize. A most common workaround for this is simply left-shifting the IV sufficiently, so that the maximum allowed message length can be accommodated. The IV values of SRTP are shifted 16 bits to the left, since its messages are not expected to exceed 2^{16} (128-bit) blocks, which amounts to 2^{20} bytes in length.

Things are slightly more complicated, however, in the context of providing authentication to our message stream. Whilst the sender only has to run a MAC over

the entire packet, thus digitally signing its payload, source, destination and unique sequence number, it is up to the receiving end to discover and drop duplicate or even forged packets.

The receiving end, therefore, needs to repeat the following procedure for each message that it captures. First, it should check a packet against a replay list in order to determine its duplicity. Afterwards, they should examine the integrity of a message by running a MAC over it and comparing the outcome with the authentication tag it bears. Should the message be, indeed, authentic, then we should update the replay list and forward the message to the application layer. Fortunately, transport protocols, such as TCP, do keep track of such lists for the purpose of dropping duplicate packets. Relying on TCP alone to solve a problem of that stature, however, is ineffective; its internal state is updated before the message can even be checked for authenticity in the application layer, thus, leading to any subsequent authentic messages being dropped by the transport layer as duplicates.

As a final note, it is useful to observe that since the sequence number field of a packet will wrap-around itself, eventually, most protocols keep track of an additional counter called a rollover counter (ROC). The ROC serves as an extension to the sequence number which is carried by most packets. It, thus, serves the twofold purpose of allowing for more packets to be encrypted by using the same key, without violating the nonce property and reducing packet overhead by being removing these bits from the header. In the context of encryption, the value of ROC is indeed taken into account when constructing the IV. Additionally, while authenticating the message, the MAC processes the value of ROC, leading to a message stream which is still protected against replay attacks, even after its sequence number has wrapped around itself.

CHAPTER 3. SYSTEM ARCHITECTURE

We envision a transport protocol as part of a system that is both secure and reliable in capturing real-time streams. Drain nodes store information on their local storage media. On the other hand, the source nodes keep track of dispersing the information that their streams carry.

We are splitting the protocol's presentation in this chapter into distinct sections so as to help better explain its functionality. We start with an overview of the system, its nomenclature and a brief summary of the flow of data between the source nodes and the drain nodes. Then, we present our full reasoning for disqualifying TCP as our protocol of choice for the fourth layer and why we turned to UDP instead. Following that, we discuss our information dispersal scheme, its message format and how drain nodes process them. Finally, we conclude by discussing certain packet transmission issues and how we counter them.

3.1. Protocol overview and nomenclature

Our transfer model is push-oriented. A source node initializes the session by reading the CRS (Cauchy Reed Solomon) parameters in addition to the node mapping, which remains fixed. The source node contacts the drain nodes, begins streaming shares of

data to them and uses the drain node acknowledgements to decide which packets it should retransmit. A pull-based model, wherein drain nodes issue requests for the parts of a stream share that they are still missing would not be an efficient alternative. Since information only gradually becomes available on the source nodes, this would incur the overhead of a round-trip time before streaming new data; a source node should first notify the drain node about the arrival of a new packet and then wait for its response.

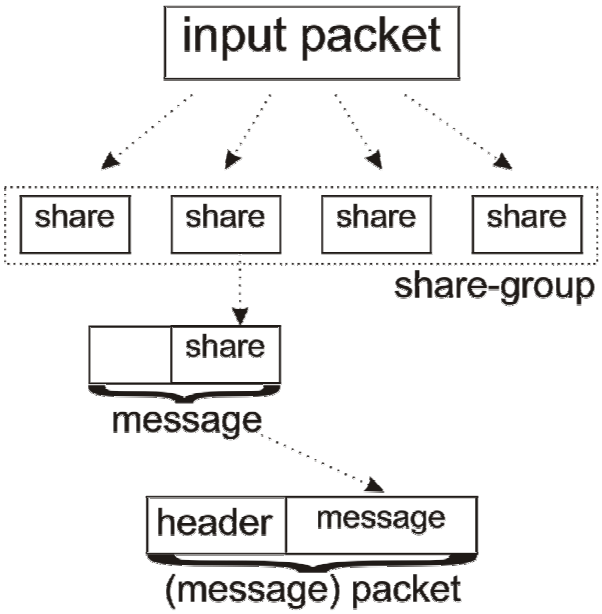

Figure 3.1 Data transformation in the system. An *input packet* is dispersed into *n shares* which all belong to the same *share-group*. A *message* carries one share along with its storage-level related information that the drain node requires to process it. Finally, a *packet* carries a message and an accompanying header which conveys transport-level related information.

We define that source nodes receive input from the streams they serve in variably-sized *input packets*. These packets undergo a series of transformations, which can be seen on Figure 3.1. The source node initially splits the input packet into *k data shares* and, then, applies CRS to produce the *m coding shares*, similarly to how strips are defined. Such shares, *k* of which can be decoded together to produce the original *input packet* are called *peer shares* and belong to the same *share-group*. The source node encapsulates each share into a different *message*, which a drain node can process individually to store its payload. In order to guarantee confidentiality and data

integrity during transfer, the sender prepends a small header to the message to form a *packet* which it subsequently encrypts and authenticates. Note that we use the terms *packets* and *messages* to describe the outgoing traffic of a source node exclusively, whereas the term *input packet* always refers to new stream data.

The fact that CRS is applied on equally-sized strips implies the use of a padding scheme. The padding itself is temporary, and usually involves appending zero values to the last data strip. Although it is not necessary to transmit or store any of the padding bytes, their presence, alone, can complicate our design considerably, as we explain on the information dispersal section.

In our current implementation, we assume that both ends of each connection pair between a source node and a drain node have established a secret key. Any specific secret key should only ever be used once to encrypt and authenticate the same pair, otherwise this would violate the nonce property of the cryptographic primitives that we are using. While this requirement seems to be rather unrealistic for our system, there are plenty of initialization protocols which use public-key cryptography to establish unique keys between both ends of communication. Our ultimate goal is to have a source node authenticate itself to the drain nodes prior to being allowed to store any shares on them. Additionally, it should be possible to revive a previously expired session by having the communicating ends exchange their state mutually. As a reminder, the cost of any initialization algorithm is going to be amortized throughout the session, mainly due to the bulk of data transfer involved. Our only concern is that the initialization algorithm should be secure.

Every stream input packet arrival triggers off a short chain of events on the source nodes. In order to accommodate a secure and reliable transfer, we have taken a wide range of possible scenarios into consideration which include, but are not limited to, network and node failures. Whereas the network failures are transitive in nature, node failures may render a collection of nodes inaccessible for an extended period of time. Keeping a record of outgoing traffic and the data associated with it increases the processing overhead required on the source nodes. However, it is absolutely necessary to do so, since it is the only effective measure against certain common issues, such as

lost packets and node failures. A rather straightforward means to implement this scheme is keeping a copy of the outgoing packets themselves on the sender-side until it receives disk-write confirmations from a set of remote drain nodes. Additionally, this approach allows us to skip repeating any encryption-related computations, when source nodes have to retransmit a packet.

Drain nodes maintain two separate lists for tracking duplicate packets and generating their write confirmations. The *reception list* contains the packet indexes of all the packets that have been received and authenticated. The *confirmation list* contains the indexes of the packets that have been committed to disk and, as such, it is a subset of the reception list by definition. Upon receiving a new packet, a drain node authenticates its contents and looks its index up on both lists, which leads to one of the following three events.

1) A packet which has been received but not confirmed indicates a duplicate message which the receiver can safely ignore.
2) A packet index which has been confirmed implies that a confirmation packet has been lost on its way back to the source node and needs to be retransmitted.
3) The packet index belongs to neither list, and as such, the drain node updates its reception list, decrypts it and forwards it to the storage layer for further processing.

We can further reduce the load on the drain nodes by merging the first two types of events together. A drain node only needs to look an index up on the reception list and determine whether to accept a packet or drop it and replay the most recent confirmation packet. Maintaining such lists can be quite efficient, as it is possible to store ranges of sequence numbers as opposed to committing entire packets into memory.

Another interesting property that allows us to be more conservative in designing drain nodes is owed to erasure-coding. The odds of several ($\geq$m) receiving nodes failing simultaneously are pretty low. Therefore, we have the opportunity to batch operations together in order to increase the throughput of our system and limit the effects of attacker interference. As far as network traffic is concerned, new confirmation packets

are produced periodically. We are doing this in order to prevent write confirmations from overwhelming the source nodes when drain nodes operate under the effects of a bursty packet arrival pattern, which may be adversary-induced. We have made similar improvements when for committing data to disk; a drain node combines several consecutive messages referring to the same stream and executes a single sequential disk write.

An additional reason which led us to put the creation of acknowledgement packets on a periodic timer is the fact that they, too, need to be authenticated, much like the message packets. Disk confirmations are very crucial to the reliability of the system as a whole and, therefore, protecting them against forgery necessary. The authentication tag increases the overhead of an acknowledgment packet overall. We can, however, greatly offset that by merging all outgoing confirmations to a source node into a single confirmation packet.

In order to prevent an attacker from replaying acknowledgement packets, all ack packets carry a sequence number of their own. Each of these packets is heavily in line with TCP-SACK [6], which we explain on section 3.4.3, and conveys the latest state of a drain node to its fullest, thus making any preceding acknowledgements redundant. That being said, a source node may skip processing several acknowledgements by only examining the one bearing the greatest sequence number, at any given moment.

Finally, our protocol employs a retransmission mechanism on the source nodes to make up for packet loss. Senders maintain a running estimate of the retransmission timeout (RTO) by using measuring the round-trip time (RTT) of messages and their respective acknowledgements. Using a RTO estimate that is smaller than the actual RTT would lead to transmitting more packets than what is necessary. On the other hand, using a larger estimate may trigger the loss of data whenever a source node crashes before managing to retransmit a packet following a transient network failure.

3.2. TCP' shortcomings

We have a specialized setting for our system in mind, which we believe that TCP would be highly inadequate if applied to. Whereas it offers a long list of remarkable features which make up for a reliable transfer, we believe that there is a sufficiently large room for improvement so as to make it better serve our purposes. Since modifying TCP itself to include these modifications is out of the question, we have fleshed out our protocol from scratch by using UDP. Before we begin discussing the disadvantages of using TCP, however, we briefly introduce its features.

3.2.1. Introduction to TCP

TCP easily becomes the first choice that comes to mind, when picking a reliable transfer protocol. It is an end-to-end protocol which features a powerful transmission mechanism that handles acknowledgements provides congestion and flow control and maintains packet order. TCP handles its input in the form of a bytestream which increases in size as new data becomes available. The bytestream is broken up into separate segments which are subsequently released into the network in the form of packets.

The receiver maintains a bounded receiving window which it uses to collect data before forwarding them to the application layer. It reads the sequence number of a TCP packet header, which determines the bytestream offset of the payload and copies new data to a receiving buffer if it does not exceed the window. The buffer is described by a specific socket field called una_seqno which identifies the lowest bytestream offset that the receiver has yet to acquire. Every time this field manages to advance, new data becomes available to any subsequent recv() calls to that socket.

Following that, the receiver uses the acknowledgement sequence number field of the TCP header to notify the sender. Instead of producing separate packets for this, TCP simply piggybacks its acknowledgements along with any other data it needs to transmit. The sequence number is the una_seqno field of the socket, which signals that the receiver has acquired all parts of the bytestream prior to that offset. In addition to that, the TCP-SACK option enables a packet to carry additional

acknowledged ranges back to the sender. This helps prevent any subsequent transmissions from sending duplicate portions of the bytestream past the una_seqno mark, that have already been received. Finally, the response from the receiver also advertises the new size for its receiving window, which, in turn, is used to limit how much data the sender is allowed to transmit during the following network round trip.

The sender digests the receiver's response and makes a new calculation of the network's round trip time, which it uses to update a running estimate of the retransmission timeout. Following that, it reawakens itself on every round trip time interval to check whether it needs to send new data, or retransmit old one. TCP automatically assumes that any packet loss is due to either traffic congestion or bad RTO estimate. Therefore, it shrinks its sending window and doubles the RTO estimate for the following transmission so as to avoid making things any worse off for the network.

3.2.2. Potential improvements

The reliability of a system depends highly on its capability to swiftly commit data to non-volatile storage. Therefore, we would rather have our protocol dispatching its packets to the receiving end with the lowest possible latency introduced. Since we are dealing with storing immutable data streams on an erasure-coded environment, which is highly resilient to errors on its own accord, we believe that we can go a long way improving the performance of our system, particularly in terms of one-way delays.

TCP is a general purpose protocol, which is particularly useful when one wishes to perform bulk data transfer across different nodes. We, however, believe that some of its features, such as in-order packet delivery, would do more to harm the reliability of our transfer than increase it. We have grouped our concerns into three discrete categories, which we analyze in the following sections.

Despite the shortcomings that we point out in this section, TCP bears a remarkably lengthy list of features that we would like to encapsulate in our protocol. Since our

main line of focus is to design a robust retransmission algorithm for our protocol, we have borrowed several TCP features, such as selective acknowledgements and RTO estimations among others, which we have tailored to fit the needs of our system.

3.2.2.1. Flexibility concerns

TCP is far too inflexible when it comes to implementing a fair set of our design goals effectively. Our system communicates information that has little need for versioning and, as such, the semantics of in-order delivery would only manage to decrease its performance instead of providing any sizeable benefits to the rest of the system. A stream packet is still useful, even its preceding ones have yet to arrive. To that end, we believe that our transfer protocol has no real need for a receiving window, in the sense that it is defined for TCP.

The receiving window that TCP uses is bounded by the memory that the receiver can allocate for the bytestream portions past the una_seqno byte of the receiving buffer. In order to guarantee in-order delivery, TCP makes new data available to the application layer only whenever the una_seqno field progresses. Since an exceedingly large window would only use up the receiver's memory resources, it makes sense that it is only sufficiently large enough to cover the bandwidth * delay product of the network path.

This kind of approach, however, yields two rather significant disadvantages. First, assuming that we wish to accommodate high bit-rate streams on high delay network paths, we may end up spending a large portion of the memory pool in order to make provisions for the receiving buffers.

Instead, if we simply stored those packets as soon as they arrived, regardless of their ordering, we would have no use for specifying such receiving buffers. Since this approach costs us no memory whatsoever, we can easily extend the size of our receiving window, as long as the newest packet does carry an index that wraps around the first unacknowledged sequence number. Second, assuming that packets can get lost along the network path, a single source node failure can lead to considerable loss

of information, which could have easily been avoided, otherwise. The loss of a single segment along a network path makes a receiver unable to read past the una_seqno offset of its incoming bytestream. If we are dealing with high bit-rate streams, it is likely that large sums of information can get trapped inside the receiving buffer. We consider this to be a major setback in the reliability of our system, even more so since our data storage semantics are pretty straightforward, and do not require any further coordination from the source node to complete, unlike other storage systems.

Finally, it would have been impossible for us to implement handling m-quorum events properly by using TCP. Storing every message that belongs to a share-group is not required, as its information can easily become available through decoding the rest of its peers together. To that extend, we should drop a packet off the retransmission queue when either a drain node confirms storing it, directly, or several of its peers in the share-group have already been confirmed. Implementing such a feature by using TCP would have been hard.

3.2.2.2. Semantics concerns

TCP exposes a very user-friendly interface to a developer, when it comes to designing the application layer for both communicating ends. However, it is highly inadequate for satisfying the semantics we have in mind for our system due to the fact that the API offers a different type of control over it, from what we actually require.

As far as senders are concerned, the reliability guarantees of TCP are limited to network delivery. Furthermore, since it is not possible to reclaim any data from the kernel's sending buffer, even if the application finds out that a receiving node has crashed, nearly every node failure leads to loss of data. One workaround to this issue is to have drain nodes use TCP to transmit confirmation messages back to the senders once they commit their data to disk. Similarly, source nodes should use a bookkeeping structure of their own to keep track of uncommitted messages, in addition to the one that TCP maintains for them on the kernel space. This solution would, however, defeat the purpose of our protocol operating efficiency, as the source nodes would be better off without having to update two different structures for every

confirmation they receive. Instead we can simply merge those two layers together by having drain nodes put off acknowledging anything until they commit data to disk.

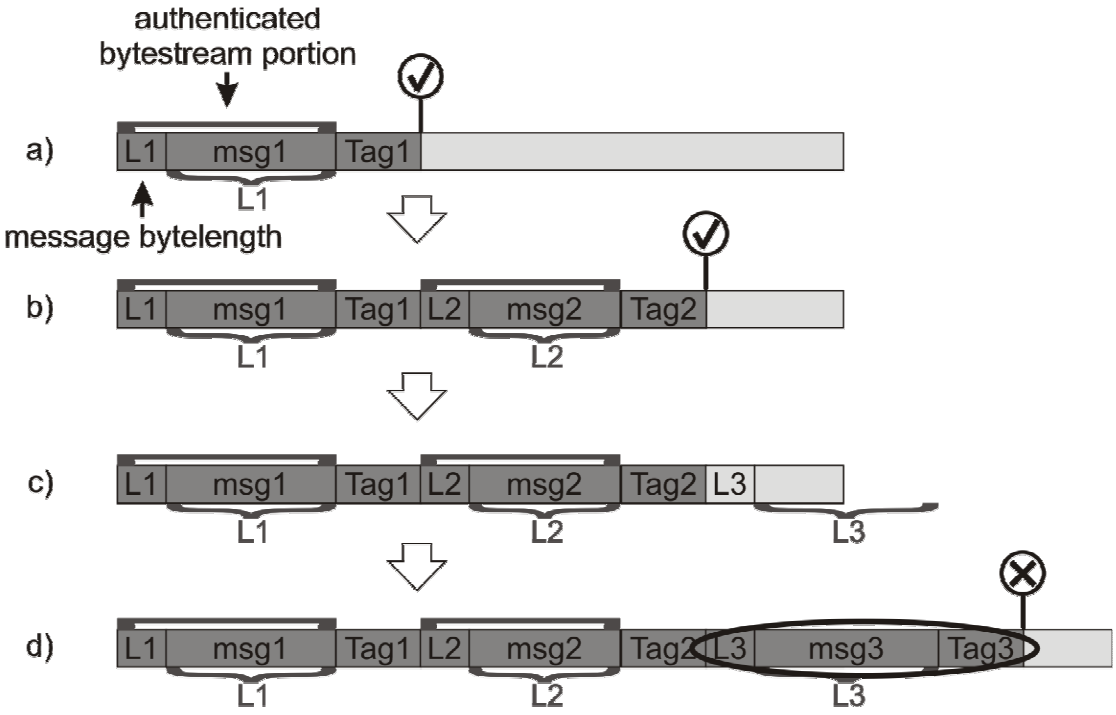

Figure 3.2 A successful TCP injection. Braces ([) visualize the packet length, L_i, whereas brackets ({) indicate the portion of the bytestream that Tag_i authenticates. When the forged packet arrives, the receiver cannot trust any value on the red-coloured portion of the bytestream, including the supposed packet length of the message, L_3. Therefore, it has no other option but to terminate the session.

3.2.2.3. Security concerns

TCP is a stateful protocol, which can lead to certain undesired circumstances when we put authentication into perspective. If we tried to make source nodes introduce authentication to data before they insert them into their outgoing bytestream, then this would leave our protocol vulnerable to a breed of DOS attacks, which we will introduce further along this section. Otherwise, if we were to use IPSec [4] to secure outgoing traffic, then it is likely that our attempts would probably fail; a third-layer protocol cannot possibly retain the same high-level perception of message authentication that the application requires, and is able to provide. This observation happens to be just another facet of what is collectively known as the Horton Principle; “Authenticate what is being meant, not what is being said".

For the purpose of making our concerns clear enough, let us first describe how a successful TCP injection can be achieved. An adversary manages to take over an intermediate router along a connection path and, as a result, they can now control every aspect of the network traffic on both directions. The simplest way they can manage to inject forged data to TCP's bytestream is by altering the payload of a packet. The receiver would then read the sequence number field of the packet and copy the forged data to its respective offset on the receiving buffer. Let us examine how we would deal with TCP injection by using either an application layer authentication approach or IPSec respectively.

Application layer authentication approach: An application-layer oriented authentication scheme can never recover from a successful TCP injection. The receiver is still able to run a MAC over a message, verify that its payload has been forged and drop it altogether. Alas, due to the fact that the socket's internal state cannot revert to its original value, any subsequent retransmissions will have no effect on restoring the bytestream to its uncorrupted form. What makes matters even worse, however, is the way that we encapsulate consecutive messages into the TCP's bytestream.

The extent of damage caused to the bytestream by a single TCP injection can be far greater than the size of its forged portion, whenever variable-sized messages come into play. A sender typically has to choose between two different methods for separating individual message in the bytestream; they can either prepend a bytelength value to each message, or insert fixed tokens between message-and-tag pairs. A successful TCP injection can, however, corrupt the contents of any follow-up messages by altering the value of those delimiters. Security protocols that use TCP as a basis, such as SSH, have to resort to terminating a running session and starting a new one whenever they discover a forged message. This would, however, enable adversaries to mount Denial of Service (DOS) attacks on our system and cripple its performance. Thus, an attacker which makes a single successful injection into the TCP stream brings the data transfer to a halt and forces the communicating ends into

reinitializing the session, which involves the use of public key cryptography, and may span several round-trip times in duration.

An example of the effects of a successful TCP injection attack on the messages separated by bytelength values can be seen on Figure 3.2. Each message consists of a bytelength value (L_i field), its payload (unmarked portion) and an authentication tag (Tag_i field) which is used to authenticate both the bytelength and the payload. Each bytelength value declares the length of the payload and, consequently, the location of the authentication tag of the message. The receiver has already received two messages which it has validated successfully, but has yet to receive sufficient bytes from the third message so as to be able to process it. When the new packet arrives, the receiver runs a MAC over the third message and finds out the MAC-produced value and the authentication tag differ. This means that any byte that belongs to the authenticated portion of the third message's bytestream could have been tampered with by an attacker. If the attacker has only altered the payload or the authentication tag of the message, then the receiver can still trust the bytelength value and move over to the next packet in the bytestream. Otherwise, the receiver cannot trust the bytelength value and should terminate the session. Unfortunately, however, the receiver is unable to distinguish the aforementioned scenarios, and should always do the latter.

Matters are only slightly better when a fixed-length message approach is involved. Such a scheme would, however, require source nodes to delay data during transfer, or use a padding scheme, so that they can always produce full-sized messages. A receiver can safely ignore any forged messages that it detects and still carry on reading from the same bytestream without having to start a session anew. TCP's acknowledgement mechanism would, however, mean nothing; we would still require drain nodes to notify source nodes about the reception of *authentic* messages.

IPSec: The only real counter to TCP injection requires focusing our efforts on the third layer, much like IPSec does. Its security credentials do not differ from what can be expected from typical message security protocols. The only exception to that is that it is applied directly on the IP layer, thus, effectively protecting every single packet segment. Furthermore, commodity hardware, such as routers, can run IPSec to secure

both ends of a communication channel while using an insecure network infrastructure. While the security guarantees of IPSec are acceptable, we believe that there is much room for improvement in terms of performance and fail-stop semantics.

IPSec is a protocol which is applied directly to the third-layer and can integrate seamlessly with both UDP and TCP as its overlying transport protocol. As a result, its algorithms make no distinction between new packets and those that have been transmitted before. If TCP decides to retransmit a segment that has timed out, IPSec will produce a new IPSec packet for that segment, with a fresh IPSec header sequence number. The packets for both segments should pass the authenticity checks of IPSec on the receiving side and get relayed to the TCP above. TCP would, in turn, discover that both packets decode to the same segment and drop one of them as duplicate. It is, therefore, evident that retransmitting the same packet requires rerunning the same cryptographic primitives over an identical set of data. We, instead, opted for having our protocol only ever encrypt and authenticate a packet once per session, as we expect packets losses to take place quite frequently, due to both network failures and potential attacker interference.

Fail-stop semantics govern how much damage a single authentication layer failure can cause to our system. While an adversary cannot always guess the right authentication tag to include with their messages, it is reasonable to assume that they will always manage to brute-force a single message past the authentication layer. A system exhibits good fail-stop semantics if it can isolate the damage caused.

Every message authentication protocol, however, has its own limitations, and that is when fail-stop semantics come into play. Regardless of however long we opt for the authentication tag of our messages to be, a persistent attacker may still be lucky enough to get a single message past the authentication layer. Ideally, we would like to isolate the damage caused in our communications to that single message. Alas, using IPSec to encrypt and authenticate a TCP communication would bear nearly the same outcome as the bytestream injection scenarios we described before. The only difference in that the application layer would never be able to tell the forgery apart

and terminate the session, since it is not expected to have an authentication mechanism of its own.

Contrary to IPSec, we combine the structures that prevent replay attacks, detect duplicates and produce acknowledgements into a single entity which we use to provide reliability and integrity. Our transmission semantics govern that a retransmitted message remains identical in content to its first transmission and, therefore, utilizes the same mechanism that guards against replay attacks. As a consequence, source nodes are not required to secure their outgoing traffic in the event of packet loss. For the purpose of providing storage-level acknowledgements and preventing replay attacks, drain nodes use the reception and confirmation lists. These lists are not independent from one another; the confirmation list is a subset of the reception list and has only been introduced so that we can streamline the operations on the drain node by helping decouple its reception and data storage mechanisms, as we discuss on the implementation chapter of this thesis.

3.3. Information dispersal

Source nodes capture real-time streams, one packet at a time, and disperse their contents to the drain nodes as soon as new information becomes available. They split each input packet into k data shares and use CRS to produce m additional coding shares. Afterwards, they encapsulate each share into a separate message and dispatch them to their respective destinations. Keeping the information dispersal scheme orthogonal to the packet retransmission protocol is desirable. We should, however, design message handling and preparation in a manner that we can take full advantage of what the protocol has to offer.

The remote storage scheme for our drain nodes should, therefore, be guided by these principles.

1) Source nodes are expected to receive stream packets and produce their shares in a sequential manner. This, however, does not apply to the drain nodes as shares may get delivered out of order, or never at all. Therefore, drain nodes should be able to

process and store each message individually and without having to wait for their predecessors to arrive.

2) A stream's bytestream should be seekable in its dispersed form without having to pull the entire structure from disk.
3) Although we are mainly interested in optimizing the performance of stream capturing, our on-disk structure should not penalize stream playback and data reconstruction too much.

We are presenting our information dispersal scheme by splitting it into three different subsections. First, we explain how source nodes split packets into shares and why that is important for our system. Afterwards, we describe how we use messages to carry each individual share. Finally, we present the bookkeeping information structure that drain nodes use, and how it handles incoming messages.

3.3.1. Splitting scheme

Erasure coding algorithms require a minimal form of padding so that they can split their input data into k equal-sized data strips. Applying erasure coding algorithms on files - or other objects whose size and data are known to the algorithm a priori – is hardly an issue; we need only apply the padding scheme to the final block of the object.

Contrary to that, however, this is not the case for our system. Data only becomes available gradually in the form of packets and it is impossible to predict when and whether a capturing session will conclude. Therefore, it is apparently that applying a padding scheme is going to become the norm rather than the exception when it comes to splitting packets. What makes matters worse is that erasure-coding algorithms have not been designed with appending data to an already existing object in mind. Before we offer our solution, we are going to present a very basic approach to this issue and the pitfalls associated with it.

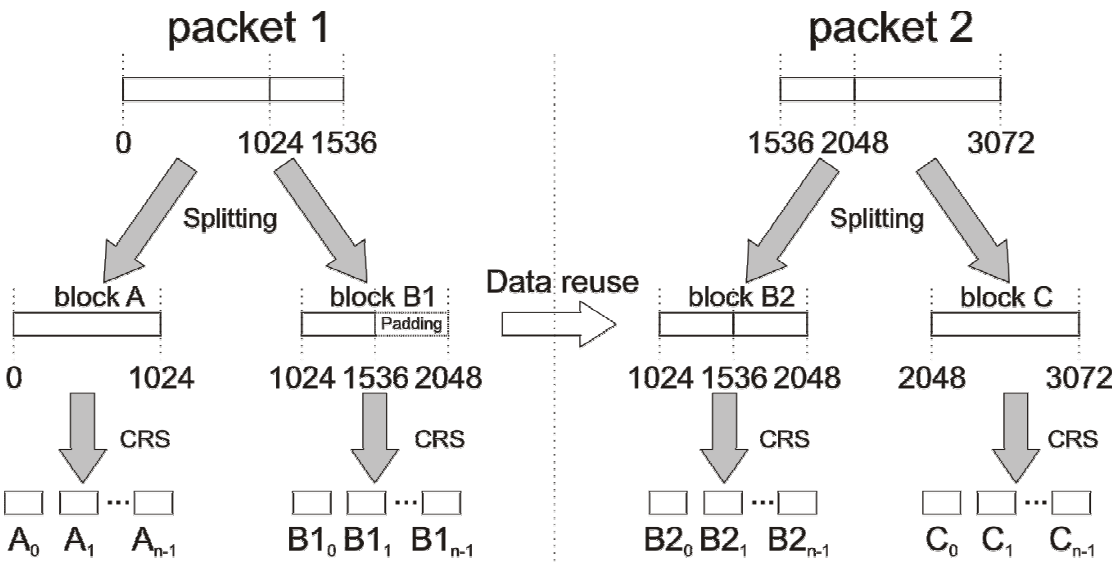

Figure 3.3 The basic approach to packet splitting. The A_i, $B1_i$ $B2_i$ and C_i shares can be used to reconstruct the [0, 1023], [1024, 1535], [1024, 2047] and [2048, 3071] portions of the bytestream respectively. Since our on-disk structure has to be seekable, drain nodes cannot retain both the $B1_i$ and the $B2_i$ version of their share at the same time. However, this can be the source of major trouble to the reliability of our system when only half of the nodes decide to overwrite their $B1_i$ share with a $B2_i$ one. If that is the case, the system will be unable to reconstruct either the [1024, 1535] or the [1024, 2047] portion of the bytestream since it lacks *k* shares of the same version.

3.3.1.1. A basic approach

A rather apparent solution to this would be splitting packets into fixed-sized blocks that can be split into *k* equal-sized data shares, and then encoding them separately by using CRS. Suppose that the blocksize has been set to 1024 bytes and a source node receives two input packets, each being 1536 bytes long. The node processes the first packet, splitting it into two blocks with the following byteranges: [0, 1023] (block A) and [1024, 1535] (block B1). Afterwards, it pads the second block with zeroes, encodes them into shares ($B1_1$, $B1_2$ … $B1_{n-1}$) by using CRS and dispatches them to the drain nodes. Additionally, the source node should retain the second block into memory for processing it further, when the second packet arrives. The source node uses the contents of the second packet to substitute the padding zeros of the second block and produce a new version of it, [1024, 2047] (block B2). Once the second block has filled up, it uses the remaining contents of the packet to produce the third block, [2048, 3071] (block C). The whole process is illustrated on Figure 3.3.

In total, the source node produces 4 different blocks (A, B1, B2, C) - two of which (B1 and B2) are different versions of the same block – in order to split the input packets' byterange [0, 3071] successfully. Each of those blocks is encoded separately by using CRS, producing n equal-sized shares which are sent to the drain nodes. Ideally, our drain nodes should only maintain the shares produced by the latest version of each input block, so that data can be easily seekable. This is pretty straightforward to achieve by having drain nodes overwrite locally stored data block shares with their updated versions, when they receive any.

This, however, introduces versioning into our system, which effectively renders our data mutable, and can cause serious trouble if left unattended. Suppose that $k = 4$, $m = 2$ and all 6 nodes have already received and acknowledged their share of block B1. The source node produces block B2, encodes it, starts dispatching a share each to the drain nodes and crashes only after it has managed to send off the 3 first shares of it. These 3 drain nodes update their block share version to B2 by overwriting the B1 version with it. Effectively, our system has now lost the ability to reproduce either block B1 or block B2, since neither version has k drain nodes maintaining it. That is to say, although all nodes have received and confirmed storing a share of block B1, all information pertaining to it has been wiped out of our system.

Alternatively, we could redesign our system so that we can accommodate non-immutable data blocks to it. This would, naturally, incur an additional overhead, as source nodes need not only dispatch data shares to the drain nodes and wait for their acknowledgements. They should also coordinate drain nodes and notify them to commit their block shares to disk and overwrite their former versions, once they have received enough acknowledgements. While the former approach is highly unreliable when it comes to already committed data, the latter limits our system's ability to store new data fast enough.

Instead, we avoided both of these pitfalls by using a different splitting scheme on our packets. The idea is rather simple and it defines how we should populate the k data shares of a stripe whenever we have to disperse a new input packet. As a result,

source nodes update their data and coding shares in incremental manner, which is similar to how information becomes available to them.

3.3.1.2. A structured approach

The smallest possible quantum of operations on CRS is called a stripe. The data portion of a stripe is *w* * *k* bytes long and it pretty much defines how much padding is needed on any input block. Although the size of blocks may vary considerably, a padding scheme is applied to them so that their size becomes a multiple of the data portion of a stripe. Supposing that the padded input block contains *p* data stripes, each data strip – or share block in our context – is going to be *p* words long. Given *k* data strips, the CRS algorithm produces *m* coding streams. The *i-th* word of a coding strip is produced by encoding the *i-th* word of every data strip, according to the transformation bitmatrix.

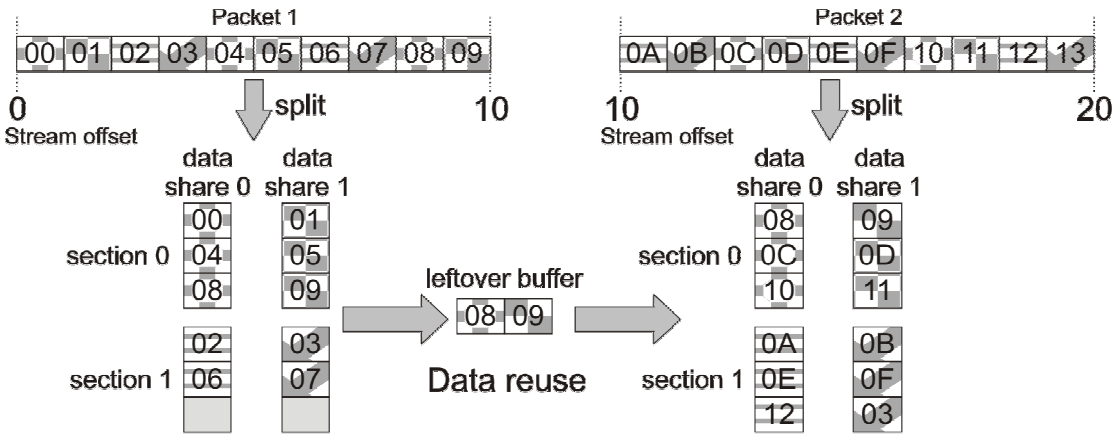

Figure 3.4 The structured approach to splitting two 10-byte input packets. The splitting algorithm retains any data it has previously encoded with padding zeroes in the leftover buffer for the following run.

We can, therefore, tweak the basic fixed-size block approach we described earlier to make it more reliable. Instead of using any arbitrary value for the block size, we picked *w* * *k*, since this defines the quantum of operations. We also extended the notion of the leftover byte buffer that we introduced earlier. Much like the previous

approach, the leftover buffer retains the final block of the previous input packet, before the sender pads it with zeros. The subsequent packet fills the contents of the leftover buffer first, before carrying on to create additional blocks. To conclude its operations, our splitting algorithm checks whether it has to pad the final block with zeroes and retain it in the leftover buffer for its following run. Figure 3.4 illustrates how our algorithm would handle splitting two consecutive 10-byte long packets of a 20-byte long bytestream into data shares. The erasure coding parameters of the system are ($k = 2$, $m = 1$, $w = 2$) and the contents of each byte equals that of its offset, as can be seen in hexadecimal form.

We refer to the number of zeros our splitting algorithm uses for padding a message as its *padding number*, which we use to version coding shares among other things. The padding number describes the last block of a coding share and can take any value between 0 (if no padding zeros were introduced) and $w * k - 1$ (if the last block only contained a single byte of data). A drain node can easily determine that the coding share that bears the lowest padding number field is more recent than another coding share that refers to the same offset of the input bytestream.

Unlike the basic approach, however, we do not need to version data block shares. Since input packets are immutable, their derived data shares are immutable too, by definition. Consequently, source nodes do not have to resend any of the data that the leftover buffer contains, when producing their next data shares. However, since we can only combine coding words with whole data words to retrieve data, we prefer to retransmit that data anyway; this would increase the size of each resulting data message by a mere $w - 1$ bytes at most. This does not change the fact that data drain nodes do not overwrite any previously stored information whatsoever. Therefore, they are excluded from the worst-case scenario analysis we have previously described.

The real benefit gained from using the padding number to version coding shares is the added flexibility when it comes to combining different shares to reconstruct lost data. During playback, a reader node that has access to all the data and coding files of a stream can reconstruct it by decoding its data, word by word, according to the transformation bitmatrix that we used to disperse them. The reader has to find k

different words referring to the same version in order to reconstruct up to *m* missing words for one specific bytestream offset. Since data words have no actual version, the reader can downgrade them by reversing the splitting scheme and temporarily substituting the data of a word with virtual padding zeros. Additionally the reader can upgrade or downgrade the version of a coding word if the data bytestream contains zeros after or prior to the padding number mark, respectively.

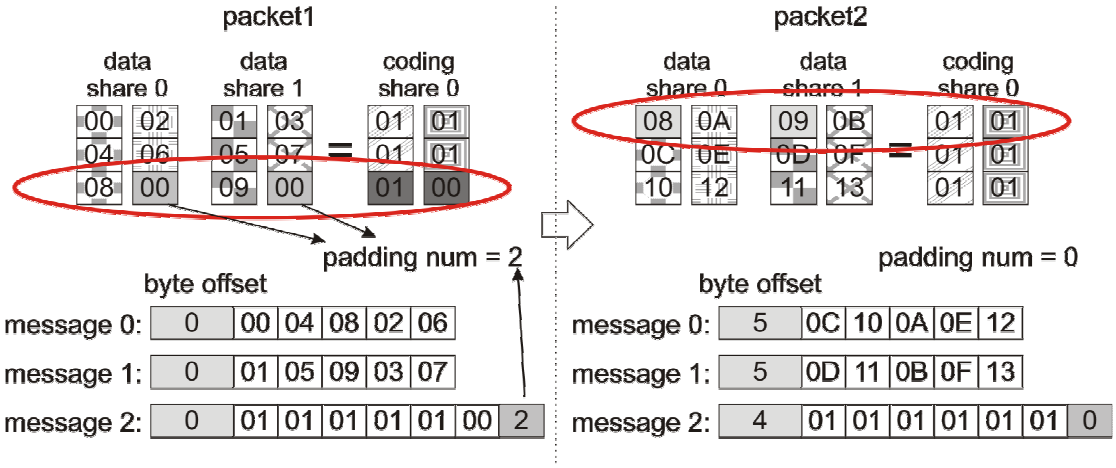

Figure 3.5 Creating messages from two 10-byte input packets. Since the first set of shares, to the left, were encoded through using two padding zeroes, its coding shares bear a padding num of two, which is recorded on the message. The following coding message has to overwrite any coding data that was produced through using padding zeroes and, therefore, carries a byte offset field value of four, as opposed to the byte offset of five that the data shares carry.

3.3.2. Message format

Source nodes use messages to pass information to the storage layer mechanism of the drain nodes. As we previously stated, a drain node can miss out on certain shares. Therefore, each message should contain sufficient information so that nodes can process and store them immediately. These messages contain the data of a share-block, a value that describes its byte offset position within the locally stored stream file of a drain node and the padding number of its last word. The words of a share-block are organized into a collection of *w* sections of size *p* each (roughly), as opposed to *p* words of size *w*, and do not contain any padding zeros. Figure 3.5 carries on the splitting example that we previously introduced and shows the remaining data transformations required to create new messages.

Our system transfers these messages by using packets, which we fully describe in the following sections. The reason we are not using the byte offset field of the messages as a sequence number to resolve packet ordering, encryption and authentication is because two consecutive coding shares often have overlapping boundaries. Since we expect our packets to be more than *w* * *k* bytes long, it is unlikely that two messages will carry identical offset field values. However, the fact that the boundaries of two consecutive shares can overlap makes the offset field value unsuitable for use on widely-adopted block cipher modes such as the counter mode.

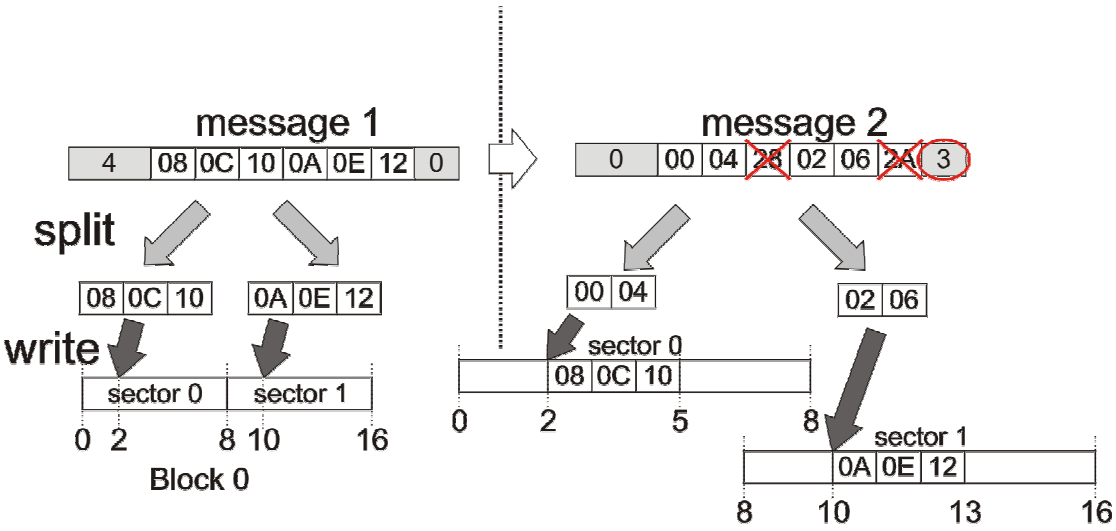

Figure 3.6 Message storage and resolution of a range collision on a drain node. The 28 and 2A byte-values of the second message have a padding number of three, which qualifies them as less recent than the already stored 08 and 0A byte-values, which have a padding num of zero. As a result, the former byte-values are ruled out during the range collision resolution phase and do not overwrite the latter byte-values.

3.3.3. Bookkeeping structure

Drain nodes capture their share each stream on a different file. They receive stream share messages, process the byte offset and padding number fields and record their information on the appropriate location in the file.

The file itself consists of fixed-size storage blocks in order to accommodate for faster reconstruction during playback. Instead of recording the words of a stream sequentially, we use storage blocks. We have separated each storage block into *w*

sectors to record the respective sections that each message consists of. A drain node identifies the sections of a message share, locates the starting storage block - according to the byte offset field – and distributes its contents to their respective sectors. The reason why reconstruction works faster is owed to the way high performance implementations of CRS are designed. These run on a pre-computed schedule that they derive from the transformation bitmatrix. Since XOR is the basic form of operation in CRS, they avoid running an iteration of the same schedule for each word by XOR-ing entire sections of data together.

At first glance, it might appear that this scheme of spreading the contents of a message across w different sectors of a file, instead of writing them down contiguously, might be crippling our write performance. This is mainly owed to the fact that writing data by using one single seek is always more efficient than doing w ones. However, there is a catch: the way that filesystems operate when it comes to updating file blocks. They have to fetch the corresponding block into memory, alter its contents and write it back down, again, possibly on a new location to guarantee atomicity. If the size of the file's storage blocks matches or is a divisor of the filesystem's block size, then the system will be required to carry out the same amount of disk operations.

Drain nodes also use a metadata structure in order to identify the valid sections of a data file and handle coding message versioning. This metadata structure contains a list of all valid bytestream ranges within a file. Every element of that list also carries a padding number field for the range which describes the version of its last word. Moreover, the padding number field values for the rest of the words falling within a range are assumed to be zero by definition. Drain nodes, therefore, consult their metadata structure to decide which parts of the data file need to be overwritten with new information, when it comes to resolving overlapping boundaries. Figure 3.6 describes an example of two messages that arrive in an out-of-order fashion to their coding drain node. Just like the previous examples, $w = 2$. Since the second message carries a padding number value of 3, the bytes carrying the values 28 and 2A will not overwrite the bytes 08 and 0A, since the latter bear a padding number value of 0.

3.4. Packet transmission issues

Source nodes are expected to have a high load of incoming stream packets that they need to disperse, since they are acting as proxies to the rest of the system. They produce *n* different messages for each input packet they receive, which they have to encrypt and authenticate. Since these operations can be costly in terms of CPU, they maintain the encrypted and authenticated versions of these packets into memory, for as long it is necessary. Therefore, if a packet times out, a source node saves its CPU time to process new packets by not processing old ones anew.

This particular direction, however, limits our choices for selecting a suitable packet header format. Since we are using a MAC to authenticate our packets, both their payload and their header should remain unchanged between different retransmissions. As a consequence, we cannot include certain fields such as transmission timestamps, which are often employed by typical retransmission algorithms. Nevertheless, we believe that this also limits the amount of damage an attacker can inflict to our system, since blocking drain node acknowledgements does not increase load on the source nodes.

In the remainder of this section we introduce certain issues regarding the communications of our protocol and how we address them.

3.4.1. RTO estimation

TCP maintains estimates of both the mean value and the typical variance of the round-trip time (RTT), which they update with each measured packet. We cannot simply use the mean value as the retransmission timeout (RTO) estimate, however, as that would lead to having half packets retransmitted unnecessarily. Instead, the value of the typical variance is used to provide an acceptable upper limit.

TCP tracks one datagram per connection at a time, by recording its transmission timestamp locally. The receiver cannot use the datagram's acknowledgement to update the RTO estimate, if it has been retransmitted. This causes a problem if the delay grows faster than the algorithm can adapt, since the receiver will drop all

samples. Karn's algorithm addresses this issue by applying an exponential back-off and doubling the RTO estimate whenever a loss occurs.

There is, however, another approach which helps eliminate the acknowledgement ambiguity that applies to retransmitted packets. Jacobson's algorithm is able to track multiple packets at a time, by simply recording the transmission timestamp directly on the packet's header and updating it, whenever it gets retransmitted. The receiver simply echoes the timestamp in its acknowledgement, and the sender can use all samples to calculate the estimate.

As we explained previously, we do not wish to interact with the contents of a packet, since we want to avoid rerunning the same cryptographic primitives over a packet, between consecutive retransmissions. Therefore, we used a variation of the Karn's algorithm. Our protocol tracks the transmission timestamp of every packet on their respective packet nodes, as opposed to one single packet per round trip. Even if packet loss does occur, our algorithm has a better chance of recovering, without doubling its RTO estimate needlessly.

3.4.2. Replay lists

Packets include a separate sequence number field of their own, which does not depend upon the byte offset field of a message. Unlike source nodes, which retain whole packets into their packet repository, drain nodes only maintain two lists for each stream. The *reception list* is, effectively our protocol's replay list, and is used to track duplicate packets. The *confirmation list*, on the other hand, is used by the authentication mechanism of the drain nodes to construct disk-write confirmations. Both lists are similar in form and, therefore, we only describe the reception one.

The reception list marks packet arrival by grouping consecutive ranges of packet sequence numbers together. Since sequence numbers will eventually wrap-around, it also maintains the ROC which the reception mechanism uses to authenticate incoming packets. As we previously described, drain nodes use the reception list to decide

whether to accept new packets or drop the duplicate ones and replay their most recent acknowledgement packet.

At this point, it is useful to compare our replay list structure to the bitmap that IPSec uses to detect duplicate packets. The bitmap structure is a very compact structure which uses a single bit for each packet to describe whether it has arrived. It is roughly 128 bits long and is associated with a single packet index that identifies the first packet referred to by the bitmap. Therefore, the bitmap can only track packets with indexes within the range of [index, index + 128), where index is the packet index identifier.

Although this bitmap structure is highly efficient at first glance, we are not using it for a number of reasons.

1) IPSec uses this structure to track real-time packets and, therefore, it can advance its packet index identifier sparingly. If a packet is lost, TCP will retransmit it and IPSec will assign a new packet index to that. We, obviously, do not want that, since this would require us to re-authenticate packets during retransmission.
2) We have to authenticate packets through their indexes, since this is the only field which remains unique throughout a session, since it increases monotonically. Reassigning packet indexes would have made that impractical, as it would be required to track each separate packet index that a share has used in order to process an acknowledgement.
3) Additionally, we risk losing the flexibility of our receiving window. As long as we do not reassign indexes, a single lost packet would effectively prevent a drain node from accepting packets that are 128 slots behind it. This would, in turn, negate one of the major advantages of our protocol over TCP, which is that of maintaining an infinitely large receiving window without allocating any memory for it.
4) IPSec can only detect duplicate IPSec packets with this structure. TCP maintains a similar structure to ours to detect and drop any duplicate segments which belong to different IPSec packets. The only difference, of course, is that that its structure is bounded by TCP's receiving window, while ours is not.

3.4.3. Selective acknowledgements

Drain nodes confirm successful disk-writes by using selective acknowledgements, which are similar in form and functionality to the TCP-SACK option. TCP uses collective acknowledgements by default, wherein the Acknowledgement Sequence number field of the TCP header acknowledges the arrival of all data in the bytestream up to that specific index. Instead, if the SACK option is enabled, packets also carry up to three additional ranges of acknowledged sequence numbers, which can help improve network utilization.

In addition to that, our system benefits even more from using selective acknowledgements to confirm its disk writes. A source node combines the selective acknowledgement packets from all drain nodes to pinpoint m-quorum events much more accurately than collective acknowledgements. Since drain nodes do not transmit any data of their own, we can further increase the payload of our acknowledgement packets to include 10 ranges for every stream, which we take directly from the confirmation list. Finally, we have further offset the authentication tag's overhead by grouping all acknowledgements which a drain nodes has to send to the same source node, together.

3.4.4. Advance pointers

One of the features of our protocol is its ability to handle m-quorum events effectively. Source nodes do not retransmit messages that they have received sufficient write confirmations for. Drain nodes maintain replay lists which may contain multiple sequence number ranges to describe a stream. Since they have to look indexes up for every packet they receive, these lists can easily become bottlenecks for CPU, if their number of ranges is allowed to grow indefinitely. Therefore, source nodes have to notify drain nodes about any m-quorum events and, particularly, which packets they will not be retransmitting.

The packet header includes an advance pointer field, used by source nodes to communicate this kind of information to the drain nodes. An advance pointer is a sequence number which certifies that a source node will not be transmitting packets

bearing sequence number fields which are less than it. A drain node can use this pointer to patch its reception and confirmation lists and make them more compact. Just like any timestamp-related information, drain nodes only populate the packet header fields once, before they run the MAC over them. Any subsequent retransmissions of the same packet will still carry the same advance pointer field value.

Using the advance pointer field may lead to incorrectly classifying certain packets as duplicates on the drain nodes. This type of action, however, prevents attackers from abusing m-quorum events and force replay lists to allocate more ranges in order decrease the performance of a drain node. Additionally, the only messages that this scenario applies to are the ones that have had several of their peer-shares stored successfully on their respective drain nodes. Therefore, our system maintains its fault tolerance guarantees to $m / 2$ node failures.

CHAPTER 4. IMPLEMENTATION

In this chapter, we present the implementation of source nodes and drain nodes for our system. We have split our presentation into three sections, two of them describing a type of node each, while the first section contains elements which are common to both.

4.1. Common elements

The common elements of the source node and drain node implementation include the packet format that nodes use and the security encoding protocol.

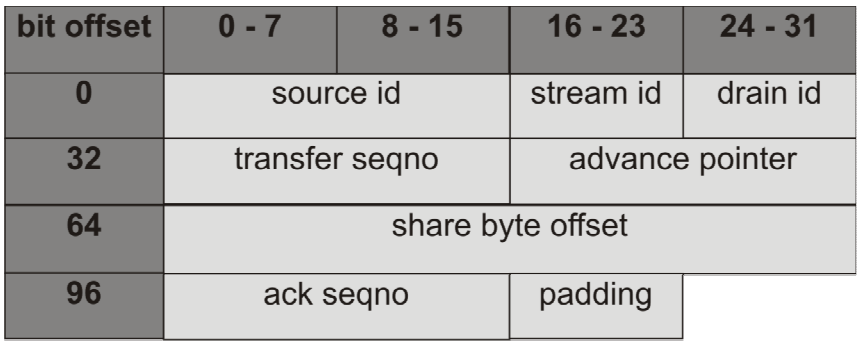

Figure 4.1 Message packet header.

4.1.1. Packets

A packet header contains three different types of fields.

1) Source, stream and drain id fields help establish both ends of communication to effectively prevent replay attacks between different sections. The source id field can carry any arbitrary, albeit unique, value. On the other hand, the stream id field may never be set to zero, and the drain field should be assigned a value inside the [0, n – 1] range.
2) Transfer, advance and ack sequence number fields help maintain source and drain node synchronization and restore message ordering. The ack_seqno field, in particular, echoes the most recent acknowledgement sequence number field that a source node has received.
3) The byte offset and padding number fields help determine the byte offset and version of a share respectively, as introduced in the message format section of this document. We do not require a length field to describe the length of a share, since drain nodes can derive it from UDP's packet length field.

Secure messaging protocols use source and destination identification values when creating message initialization values. We assume that a higher administrative authority is able to assign unique id values to each of the source nodes. Source nodes can, in turn, keep track of the streams they service and assign id values to them. Additionally, they maintain a drain node mapping per stream, wherein the drain node id value is assumed to be identical to that of their erasure-coding mapping. Therefore the concatenation of source id, drain id and stream id field values is always unique. We expect that a credential management authority is able to provide this mapping to both communicating ends during the initialization phase of our protocol.

Since every IV value that source nodes produce is unique in our entire system, attackers are unable to carry known plaintexts attacks out to deduce the content of any stream. Typical open-source SRTP media server implementations use a fixed symmetric master key to handle stream encryption and authentication. Instead, a great deal of their security depends on not picking the same synchronization source identifier value (SSRC) for two different sessions. Since our system consists of several such source nodes, the odds of such a collision taking place are far greater.

Therefore, we believe that our approach of assigning fixed unique id values to nodes and streams and generating new master keys randomly is preferable.

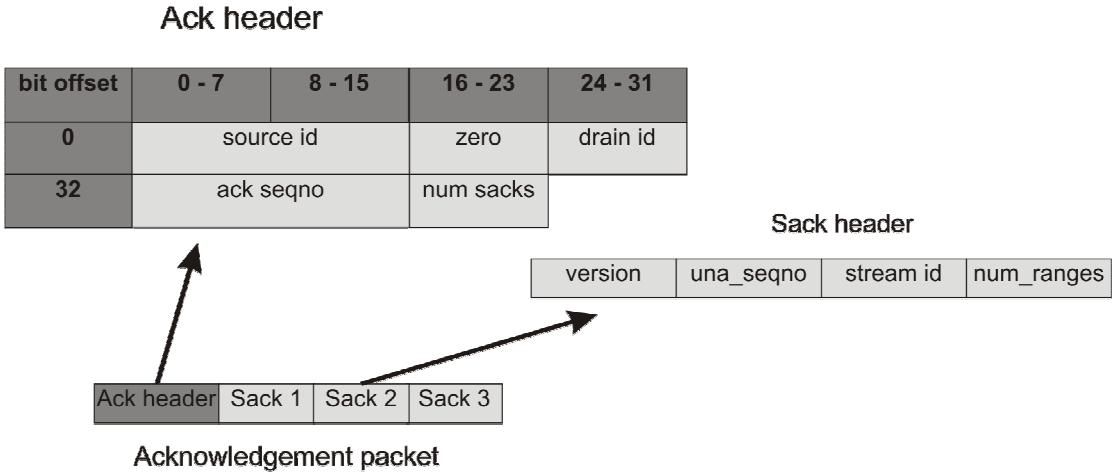

Figure 4.2 Acknowledgement and sack header format. Each acknowledgement packet can contain several sacks, and each sack can contain up to 10 acknowledged sequence number ranges.

Acknowledgement packets are a collection of write confirmations addressed to the same source node. Source and drain id fields are identical to their packet header counterparts. We set the stream id field to zero in order to prevent attackers from replaying acknowledgement packets and use them as authenticated messages. Drain nodes arrange stream specific acknowledgements into *sacks*, which also carry a version number of their own. Sacks are, effectively, an on-packet copy of the first 10 elements of the confirmation list, wherein una_seqno declares the first unacknowledged packet sequence number.

As a reminder, both packets and acknowledgements carry a 10-byte long authentication tag, which is located directly after their payload.

4.1.2. Security encoding protocol

The libSRTP library is an implementation of the SRTP protocol, which we are using to handle packet encryption and authentication.

We have modified certain aspects of the libSRTP implementation to increase its performance and make it more suitable for our system. More specifically, we have

altered the libSRTP internal structure so that it can handle the message packet and acknowledgement headers of our protocol instead of the SRTP ones. Therefore, it forms its initialization vectors by concatenating the ROC along with the packet sequence number, drain node id, source node id and stream id fields of the header.
Furthermore, we have disabled the library's internal replay list structure, since we are using the reception list to carry out the same task for the drain nodes and we do not want anything else meddling with it. As a result of our modifications to how we construct initialization vectors, source nodes can pick the same key to encrypt all communication directed to the same drain node without compromising security.

4.2. Source node implementation

Source nodes act as a proxy for our system and disperse packets to the drain nodes. Since they can fail at any time during their operation, it is obvious that we should attempt to get data across to the drain nodes as soon as possible, where they will be better protected by non-volatile storage and the splitting scheme. The design of our protocol enables source nodes to skip a significant portion of the computations involved during retransmission. We have focused our source node implementation in minimizing the packet preparation overhead.

Source nodes consist of three different mechanisms which carry out their operations.

1) The packet preparation mechanism features a processing pipeline to handle packet dispersal and security encoding. Moreover, we have parallelized its costliest element, security encoding, in a manner that does not interfere with the rest of the system and, therefore, requires minimal locking.
2) The acknowledgement handling mechanism validates acknowledgement packets and updates a local cache on the source node which describes the latest state of a drain node.
3) A timeout-handling mechanism awakens periodically to perform maintenance operations on the packet repository. It digests the information conveyed by the most recent acknowledgement and retransmits any packets that have had their timeout expire.

Prior to presenting these mechanisms, however, we describe the way that source nodes handle their streams and the internal structures they employ to accommodate their functionality.

4.2.1. Stream and connection handling

Source nodes track the state of their active connections through the *drain control blocks*. These blocks only contain any drain node information that is common among all streams that the source node serves. This type of information can be partitioned into three categories.

1) A UDP socket which remains connected to the ip address of the drain node.
2) Various statistics, such as the running RTO estimate.
3) Cryptography-related information, such as the encryption and authentication keys, which the source node uses to communicate its stream shares with that drain node.

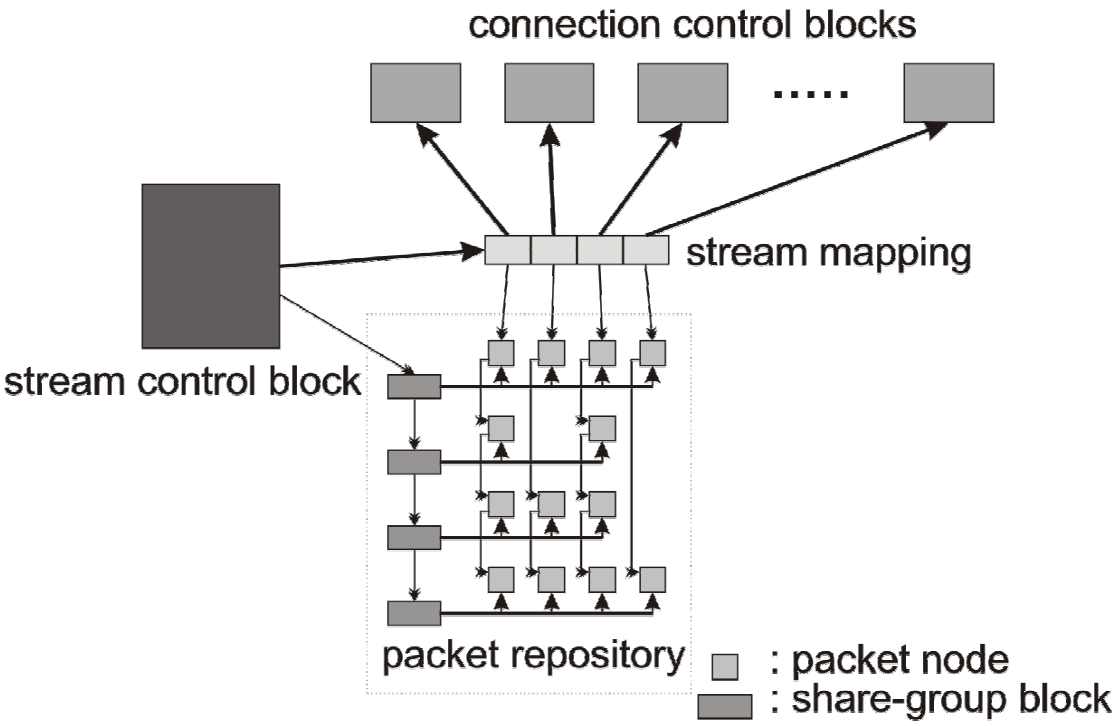

Figure 4.3 Stream and connection handling on the source nodes. Share-group blocks point to the packet nodes that carry shares which belong to the same share-group. This makes detecting and handling *m-quorum* events much easier for the source nodes. Each stream also has its own unique mapping which determines which drain node ought to be the recipient of whichever share.

Stream control blocks, on the other hand, are used to maintain any stream-specific information. They carry the drain node mapping of a stream, which is to remain static throughout the entire session. This mapping contains the last sequence number used, a copy of the most recent acknowledgement sack as well as some other sender-related information. Stream control blocks also carry information that is vital to our splitting scheme, such as the leftover buffer and the next byte offset values to be used. In addition to that, they maintain a packet repository of unacknowledged packets and two access locks; an insertion lock for the repository and an acknowledgement lock for all sacks pertaining to the stream.

Another valid alternative would have been to merge all stream packet repositories together, into a single entity. We, however, prefer maintaining separate repositories on a per stream basis, since it would allow us to extend code more easily. We can, for example, introduce a retransmission priority policy for streams or use different keys to encrypt and authenticate different streams, without much effort.

The packet repository contains *packet nodes* which are grouped together into *share-group blocks*, according to the groups their messages belong to. A share-group block contains a state field to describe whether the group runs on a *'normal'* state or an *'m-quorum event'* one, ready for deletion. Similarly, source nodes can tag packet nodes as being *'ready'* or *'under construction'*.

4.2.2. Packet preparation mechanism

The packet preparation mechanism consists of a sequence of algorithms which take a single stream packet as their input and produce *n* encrypted and authenticated messages. We are using two libraries to take care of the erasure coding and the cryptography parts of this sequence. We introduced libSRTP, which implements cryptography for our system, previously. The most efficient open source implementation of the Cauchy Reed Solomon is included in the Jerasure library, according to a very recent performance survey [10].

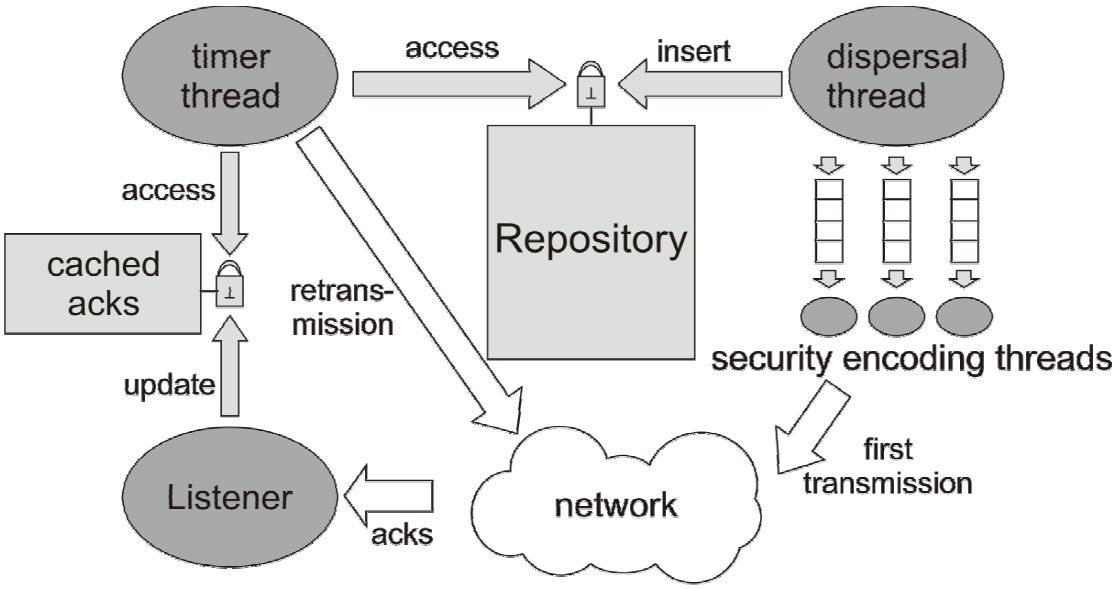

Figure 4.4 Overview of the source node architecture. The security encoding worker threads are bound to no other lock other than that of their work-queue through which they receive new tasks from the dispersal thread. The listener processes new acks as they arrive, while the timer thread uses the information provided by the cached acks to perform its maintenance operations on the repository.

The cost of message security encoding dominates the entire packet preparation process. Therefore, we have isolated this stage and split its workload evenly to a set of dedicated worker-threads. The packet preparation mechanism contains a two-step pipeline. During the dispersal step, the mechanism's function produces and inserts new shares in the packet repository. During the security encoding step, its threads encrypt and authenticate the contents of these shares before releasing them to the network. We believe that splitting this pipeline further would yield no sizeable benefits.

The calling application uses a stream handle to insert a new portion of the bytestream into our system. The packet preparation mechanism allocates a new share group block, which starts out under the 'normal' state, and *n* packet nodes, which start out under the 'under construction' state. It acquires the insertion lock to the control block of the stream, inserts the new nodes into the repository and releases it. Afterwards, it uses our splitting scheme and the Jerasure library to populate the packet nodes with data. Once it has concluded this part of its operation, the mechanism inserts a pointer to the share-group block in a work-queue to notify the *security encoding threads*.

Following that, the packet preparation mechanism returns to the calling application as the *security encoding threads* execute their workload in the background. These threads maintain a task cache in order to reduce the frequency of their accesses to the work-queue. Each of them acquires the lock to its own work-queue, moves it to its task cache, relinquishes its work-queue node resources to the packet preparation mechanism and releases the access lock. These threads only access the work-queue again when they deplete the contents of that cache.

LibSRTP's internal structure has prevented us from using any sort of load-balancing algorithm to distribute the workload equally among the security encoding threads. Instead, the function that carries out the splitting issues notifications on a per share-group block basis on every worker-queue. Each security encoding thread determines its own portion of the workload by using its thread-id value. It updates the advance pointer field of the packet and invokes the libSRTP encoding function. Since this is also the final stage of the packet preparation pipeline, they also record the creation timestamp of the message, set the state of the packet node to 'ready' and transmit the packet itself. It is also likely that a share-group enters the 'm-quorum event' state before the security encoding threads have a chance to encrypt all of its packets. In that case, the thread simply marks the packet node for deletion and carries on to its next task waiting on the work-queue.

4.2.3. Acknowledgement handling mechanism

A single listener thread waits on a UDP socket to handle incoming acknowledgements. We have not parallelized this mechanism any further because we believe there is little need to. Contrary to their message counterparts, acknowledgement packets are smaller, not encrypted and only periodically produced by the drain nodes. On the other hand, a source node may have to produce message packets at an arbitrary rate.

Whenever the listener receives a new acknowledgement it looks up its corresponding connection block. If the acknowledgement is well formed and is validated by a MAC,

the thread compares it to the most recently cached one for the same connection. Following that, it moves on to digest each of its sacks separately for each stream, requesting the acknowledgement lock before updating the local cache. The reason why this thread does not interfere with the packet repository directly is because we did not want to introduce any additional delays on the packet preparation mechanism due to lock contention. Additionally, the listener can process a great number of acknowledgement packets and provide the most up-to-date state for each drain node before the timeout handling mechanism even has to alter the contents of the repository at all.

4.2.4. Timeout handling mechanism

The timeout handling mechanism is the only other mechanism, apart from packet preparation, that directly interferes with the packet repository. It deletes any acknowledged packets, determines m-quorum events and retransmits any packets that have had their timeout expire. It is run by a single timer thread which awakens at 200ms intervals to perform its operations. Although we do not believe that this is necessary, this mechanism can be easily parallelized to offset its load on additional threads. Our main requirement of the timeout handling mechanism is that it concludes its operations swiftly enough so that it does not delay with the packet preparation mechanism, which shares one insertion lock per stream for the packet repository.

Once awakened, the thread checks on each available stream control block, processing them separately. It acquires the acknowledgement lock of the stream and merges the most recently known drain node confirmation state with the stream's packet repository to produce its results.

1) The timer thread ignores any packets nodes under the 'under construction' state.
2) It deletes packets which have been newly acknowledged, updating the RTO estimate and setting the state of a share-group block to 'm-quorum event' if the confirmation threshold has been exceeded. The timer thread will remove the remaining 'm-quorum event' packet node peers when executing its pass for their respective connection, and whenever their 'ready' state is triggered by the security encoding threads.

3) Otherwise, it retransmits packets that have had their time since last transmission exceed the RTO estimate for the running connection.

The thread acquires the insertion lock once, only if it needs to remove a packet node. It releases both the acknowledgement and the insertion lock when it concludes its operations on a stream control block.

4.3. Drain node implementation

Drain nodes receive packets which may originate from any of the source nodes available in the system. They identify the source of a packet, decrypt and authenticate its contents and then forward it to their data storage layer. Once they finish recording a message and update the respective stream's bookkeeping information structure, they send a write-confirmation out.

Contrary to the source nodes, we believe that our system can benefit greatly from batching certain operations together on the drain nodes, for a number of reasons.

1) Drain nodes form up into an erasure-coded set of devices for our system. Any single node-failure is unlikely to cause any loss of data, since several other nodes are bound to receive their own share of data.
2) We have to take disk accesses into account, since they are the definitive bottleneck in most data storage systems. A single sequential write is much more efficient than several smaller writes, as random disk seeks dominate disk bandwidth and negatively affect a system's performance.
3) Drain nodes should regulate the flow of operations, since source nodes make no batching of their own. In doing so, our system would be able to respond better to sudden bursts of packet arrival, which may be caused by an adversary. Therefore, drain nodes only update their state and produce write confirmations periodically.

Although it is not possible to group certain kinds of operations together (packet validation, for example), we believe that this helps increase the throughput and service capacity of the drain nodes.

Drain nodes consist of three different mechanisms, which we present separately in the following sections. The packet validation mechanism is responsible for dropping forged packets and duplicates. The data storage mechanism attempts to merge sequential writes together to record stream data on a simple file structure. The acknowledgement creation mechanism makes an on-packet copy of the drain node's state and dispatches it to the source nodes it receives data from. We are, however, starting off by presenting the internal state structure of the drain nodes which they use to handle their connections.

4.3.1. Internal state

Drain nodes track their inbound connections by allocating a *connection control block* for every source node. One such block holds the session key and a UDP socket which remains connected to the acknowledgement port of a source node. Moreover, since acknowledgements contain collective information about all streams that a source node serves, these blocks also contain information that is valuable to the acknowledgement creation mechanism. They hold the next acknowledgement sequence number to be used, a copy of the most recently produced acknowledgement, two event flags (dup_packet and new_ack) and an access lock to them. The remaining two mechanisms use these event flags to make the acknowledgement creation mechanism replay the previous acknowledgement or create a new one, respectively

Any stream specific state is organized into *stream receiver blocks*, which are linked to their respective connection control blocks. These blocks hold the receiver and confirmation lists for the stream, the storage bookkeeping information structure and the parameters of the erasure-coding profile. Finally, they also maintain a small packet buffer that the packet validation mechanism uses to pass messages to the data storage mechanism.

4.3.2. Packet validation mechanism

The packet validation thread listens to a user-specified port and processes a packet's header before it either forwards them to the data storage layer, or discards them. It initially looks the appropriate stream receiver block up, which we have further assisted by using a last used block cache. We reckon that the cache is going to have a high hit ratio, given that our packet retransmission scheme may send several packets off at a time.

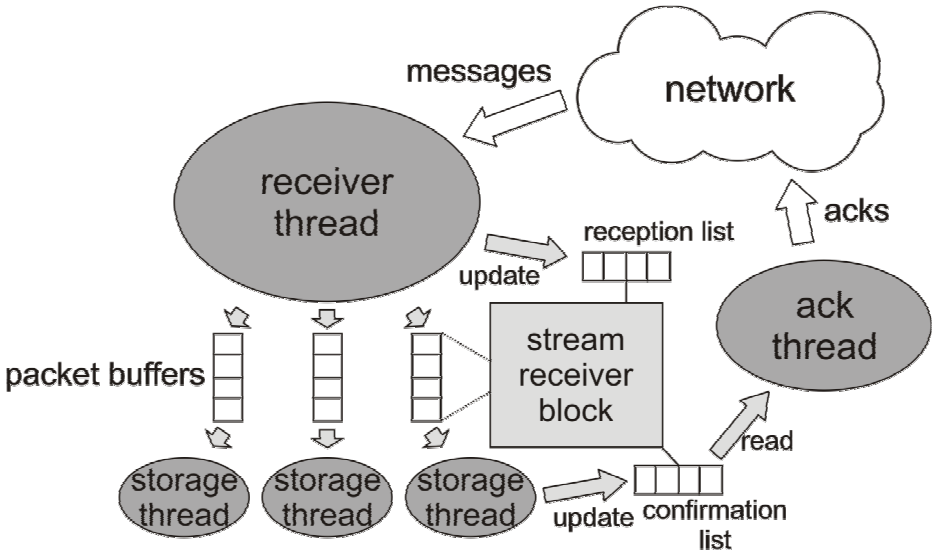

Figure 4.5 Overview of the drain node architecture. The receiver thread validates new packets and inserts them to the packet buffer of the appropriate stream receiver block. The designated storage thread empties the packet buffer, stores its shares to disk and notifies the ack thread through the confirmation list. Finally, the ack thread wakes up at set intervals and releases a new acknowledgement for the packet into the network.

The thread, then, performs certain sanity checks on the packet and looks up its index on the reception list. If the packet is a duplicate, the thread activates the dup_packet event flag and drops the packet. Otherwise, it marks the appropriate location on the reception lists for speedy insertion and runs our modified libSRTP decryption function over the packet. Much like the receiver blocks, the reception list also contains a last modified range cache, which can assist greatly during mass packet retransmission. An authentic packet causes the mechanism to use the insertion mark to update the reception list according to the sequence number and advance pointer fields of the packet.

Once a packet has been accepted, the packet validation mechanism needs to pass information down to the data storage one. It acquires the lock to the respective stream packet buffer, inserts the packet's message and releases the lock. The packet buffer is maintained in order by packet index to help detect and batch successive messages efficiently. Additionally, the mechanism passes the advance pointer field of the packet to the data storage mechanism, since the reception mechanism does not have the direct access to the confirmation list in order to update it, as that would increase lock contention.

4.3.3. Data storage mechanism

Our current data storage scheme maintains two files on a separate directory per stream in order to record their data and metadata respectively.

4.3.3.1. Operation

The mechanism uses several threads, one for each stream that the drain node services, to record their messages. Each thread receives new messages through the buffer queue of the respective stream and groups any necessary operations together to increase its throughput. It merges successive packets together on a local in-memory buffer to minimize the number of required number of disk accesses. It consults the bookkeeping information structure, resolves any byterange collisions between the new messages and the previously recorded ones before it decides which parts of each message need to be recorded. Meanwhile, it aggregates any changes directed to the confirmation list on a local buffer without locking it down.

The thread then has to update the data and metadata files in a manner that makes them maintain a consistent view of each other, even in the event of a system crash during the operation. Therefore, the thread first updates the data file and calls fsync to its file descriptor to guarantee that the data file has been updated. Afterwards it records the updated in-memory bookkeeping information structure to a temporary file in disk and calls fsync(), to guarantee that it has been updated. Afterwards, it renames the

temporary file to substitute the previously recorded on-disk metadata structure in one go. The call to rename() guarantees atomicity to the metadata file structure updates; if a system crash occurs at any point during that operation, either the new view, or the previous view will be recovered from it, and nothing else. Since the call to rename() does not guarantee that the metadata file rename has actually occurred, we also have to call fsync() to the parent folder to guarantee durability.

Following that, the drain node needs to update its confirmation list to notify the source node that the stream data has originated from. The thread acquires the access lock to the confirmation list and uses the collection of updates of its temporary list to process the confirmation list efficiently. Before it releases the lock, it notifies the acknowledgement creation mechanism of its operation by setting the new_ack event flag.

4.3.3.2. Concerns

Using a specifically-tailored filesystem to handle information storage would have arguably been a much more suitable choice for our system. Aside from any gains to performance, committing data and metadata together would yield considerably improved storage semantics. As we have already explained in the information dispersal section, coding shares are prone to some form of versioning, which we help resolve by using the padding number field. As a result, if a drain node fails to update a metadata file after having tampered with the data file of a stream, it may corrupt any already committed data. Filesystems, on the other hand, use logs to keep data and metadata changes consistent to one another. Customizing any filesystems to fulfill this role, however, would have been a very time-consuming process and lies entirely out of the scope of this thesis. Therefore, we provide a rather simple albeit realistic alternative to address this issue.

The sheer number of calls to fsync() is part of the reason why we decided to have several threads handle streams separately, instead of using one of them. Although all threads request access to the same storage medium, it is more than likely that the underlying filesystem regulates this somehow and makes the whole process much

more efficient by using its own built-in scheduling. Previous experimentation with an earlier version of our prototype featuring a single thread which handled all streams has shown that, although this scheme scales really well with higher bitrates, it scales linearly with the number of streams serviced per drain node. Supposing that a drain node is able to produce a write confirmation within 40ms from receiving a stream packet if one stream is active, then it would take roughly 4 seconds to produce acknowledgements if it handles 100 streams, and 40 seconds if the number of streams scales to 1000. Obviously this kind of scaling is far from being an ideal one.

Another point of interest regarding our storage scheme is that the storage layer handles decrypted data. While this may give rise to suspicion concerning the security of our system, we believe that providing confidentiality to on-disk data should be orthogonal to data transfer. First, as far as key management is concerned, changing the session key should not require any cooperation with the data storage layer. Otherwise, drain nodes would either have to rekey entire streams or safeguard an ever-growing log of keys, whenever a key swap occurs. Additionally, source nodes should change the way they construct their initialization vectors. Since we require that our recorded bytestream remains seekable, source nodes would be have to use the byte offset fields. However, as we explained on the splitting scheme section, coding messages often have overlapping boundaries, which prevents us from using byte offset fields with block cipher modes securely. Therefore, we advocate that the responsibility for enforcing a storage-level confidentiality policy lies entirely upon the respective administrative authorities of the drain nodes.

4.3.4. Acknowledgement creation mechanism

Drain nodes use a single thread to create their acknowledgement packets and stream them back to their respective source nodes. Since drain nodes need to authenticate their acknowledgements we have placed their creation on a timer.

The thread awakens on 200ms time intervals and reads every connection control block's dup_packet and new_ack event flags. If the new_ack flag is set, the thread

acquires the access lock for the connection and copies the confirmation lists of the appropriate streams to a new packet. It runs our modified libSRTP routine to authenticate it and increases the sequence number for the following acknowledgement. Afterwards, the thread transmits the newly produced acknowledgement, maintains a copy of it on the connection control block's local cache and releases the access lock. Therefore, handling dup_packet event flags, which might be quite common, only requires retransmitting the same previously cached acknowledgement. When everything is settled, the thread clears both event flags and moves on to the following control block.

CHAPTER 5. EVALUATION

We have evaluated our system in order to measure its performance, identify its bottlenecks and assess how its components react to varying degrees of system load. Most particularly, we wanted to certify that the packet processing delay on the source nodes remains low enough, and that the drain nodes maintain a high throughput despite the lack of batching on source nodes. Our secondary goal was to measure the effects of m-quorum events in conjunction with packet loss on the number of retransmissions and packet duplicates.

5.1. System configuration

We evaluated our system our prototype implementation using x86-based server nodes running the Debian Linux distribution on a 2.6.18 Linux kernel. We used nodes with quad-core 2.66GHz processor, 2GB RAM, and two SAS 15KRPM disks, each of 300GB storage capacity and 16MB internal buffer. All nodes are equipped with a 1Gbit Ethernet network adapter, and are connected to the same network switch. The switch in particular has a switching capacity of 96 Gbps, 64 MB of SDRAM and its throughput ranges up to 71.4 million packets per second (for 64-byte packets).

5.2. Deviations from the real-world

Despite the best of our efforts, our experimental setup differs significantly from what would be expected of a typical real-world application of our system.

First, we were unable to emulate realistic WAN-like network path conditions between source nodes and drain nodes. We initially tried using Netem to introduce packet loss and network delay in our system. However, as we started increasing stream bitrates progressively, we noticed that calls to send() were delayed for several milliseconds. In turn, this caused the each iteration of the retransmission mechanism to outlast its wake-up period and remain active permanently. We originally speculated the internal buffers that Netem uses to store packets temporarily were accountable for this delay. Despite our efforts to counter this by increasing their size limit, the problem persisted and we were forced to abandon using Netem.

In addition to that, the packet loss configuration value on Netem seemed to have no effect on the nodes' communications. Arguably, the developers, themselves, advise deploying it on the endpoints of communication in order to use that feature; Netem-induced packet loss is reported as an error to the upper layers which proceed to ignore it and resend any would-be-lost packet instantly. Instead, if we were to stream all traffic through an intermediate node that runs Netem to drop packets, then this node would become the definitive bottleneck of the system.

However, since we were very keen to measure how packet loss affects m-quorum events and packet retransmissions we had to make do with a much more simplified scheme. Both kinds of nodes use a uniform distribution to determine whether a packet would get lost each time they have to transmit one. The application skips making any call to send(), but acts as if it actually did, such as updating the number of perceived packet transmissions. Admittedly, more realistic packet loss models typically use a correlation value between successive packets to approximate that kind of behaviour. However, our system also happens to be a victim to packet loss due to traffic congestion on the network switch that we are neither able to control nor measure, which we believe that more than makes up for that.

Second, the network topology of a cluster is not a very representative portrayal of a real-world deployment of the system. Although each source node streams its packets to every drain node, traffic from all source nodes has to traverse the same network switch towards its destination.

This causes network throughput to collapse, in a similar manner to how the *incast* phenomenon affects TCP performance in cluster-based storage systems. Incast manifests itself when a node tries to retrieve blocks of data from multiple storage nodes simultaneously. This causes storage nodes to synchronize their transmission and often compete against the same resource, causing goodput performance to drop one or more orders of magnitude below the theoretical upper limit of the link.

The effects are even more severe on a LAN setting, as the minimum TCP retransmission timeout threshold, which is 200ms by default, far exceeds the round trip time of the network, which is well below 1ms. The most recently proposed counter to that issue comes from a SIGCOMM ’09 paper [16]. Among other things, the authors suggest that reducing that threshold, and using finer granularity timers on the kernel improves network utilization and goodput performance. However, we are unable to introduce this solution to our system at this point, since our RTT measurements also include data storage latency. We discuss this issue in greater detail on the Shares and Share-group acknowledgements of this chapter.

Finally, our experiments do not reflect the fact that drain nodes are expected to make use of an *external journal* to significantly increase the throughput and decrease the latency of their filesystem’s write operations. Journals resemble on-disk log files and are used to collect file metadata updates temporarily, in order to avoid disk seeks before the filesystem moves them to their final location on disk. Journals are often maintained on a separate disk, so as further these performance gains, in which case they are called external journals.

However, due to some absurd reason, not all drain nodes reacted well to the presence of a journal. During our test runs, we observed that the majority of the drain nodes exhibited a 20% reduction on the latency of their write operation, which was to be

expected. However, two drain nodes suffered from a decrease in their performance, to the point that their write latency became double of that without an external journal. Therefore, we decided to disable external journaling on all nodes.

5.3. Methodology

In our experiments we are using four source nodes streaming data to seven drain nodes on a $k = 4$, $m = 3$, $w = 4$ erasure-coding configuration. As a consequence, *m-quorum events* occur whenever 6 out of 7 nodes acknowledge their part of a share-group.

All source nodes serve the same number of streams. They produce new stream data at a fixed rate in the form of 4096-byte long input packets. Each input packet is, therefore, dispersed into seven 1024-byte long shares, which the source nodes transmit to their respective drain nodes.

In order to determine a proper stream bitrate parameter for our experiments, we have browsed the internet and have settled for three types of streams; VCD-quality equivalent streams at 1.25Mbit/s, HDTV-quality streams at 15Mbit/s and Blu-Ray Disc bitrate equivalent streams at Mbit/s. Although we do not actually expect that anyone will stream a Blu-Ray equivalent real-time stream in the near future, this can help serve as a reference to how our system would behave if we migrated from streaming several low-bitrate streams to fewer high-bitrate ones.

Therefore, each experimental configuration can be defined by three key parameters, which are:

1. The type of stream: VCD, HDTV or Blu-Ray
2. The number of streams per source node
3. Network conditions: 0% or 1% packet loss

For the remainder of this chapter we will refer to different configurations through the following abbreviation: {Stream type, number of streams per source node, loss percentage}. Therefore, {HDTV, 6, 0% Loss} refers to streaming 6 HDTV packets

per source node at a 0% network loss setting. Additionally, we have made a couple of runs after having disabled m-quorum events on our system, in order to measure their benefits. We shall explicitly refer to that setting as a no-quorum configuration, when we get to it.

Tables 5.1, 5.2 and 5.3 contain the full list of combinations we have tried between type of streams and the number of streams per source node that we have used. We have tried to carry additional experiments with a higher number of HDTV and Blu-Ray streams per source node involved. However, the system could not operate on a steady state, due to the traffic congestion that was taking place on the network switch.

The highest possible cumulative network bitrate that we have managed to stream with packet loss present comes from {HDTV, 6, 1% Loss} configuration. Although we managed to launch {Blu-Ray, 4, 0% Loss} and {HDTV, 8, 0% Loss}, their behavior was generally unstable, as network congestion was causing excessive packet loss per round trip time. Since we could not replicate the same experiments on the 1% loss configuration, we decided to discard them.

In the {HDTV, 6} setting, each source node produces 6 * 15 = 90 Mbit/sec worth of stream data, which it disperses using the $k = 4, m = 3$ erasure coding configuration. As a result, each source node releases 157.7 Mbit/sec worth of network traffic, which amounts to a 630Mbit/sec traffic traversing the network switch. Since the MTU of the network is 1500 bytes and each message carries a mere 1024 byte long payload, we are only effectively using a portion (1024 / 1500) of the available bandwidth. Therefore, the 630Mbit/sec at 1KB per share network traffic limit that we encountered is equivalent to the 922.85 Mbit/sec that we would observe from using MTU-sized shares – if possible at all.

Table 5.1 Measurement table for VCD stream type experiments. Notice that all measurements are given on a per source/drain basis. Sources produce new input packets at the bitrates and throughput provided under the Source Input tab. The actual network output and network input metrics are provided under the Source Output and Drain Input tabs respectively. (Input packet = 4KB, Share = 1KB).

	Source Input		Source Output		Drain Input	
Streams (per source)	Bitrate (Mbit/sec)	Throughput (Input packets/sec)	Bitrate (Mbit/sec)	Throughput (Shares/sec)	Bitrate (Mbit/sec)	Throughput (Shares/sec)
2	2.5	76.29	4.38	534.058	2.5	305.18
4	5	152.59	8.75	1068.12	5	610.35
8	10	305.18	17.5	2136.23	10	1220.7
16	20	610.35	35	4272.46	20	2441.41

Table 5.2 Measurement table for HDTV stream type experiments.

	Source Input		Source Output		Drain Input	
Streams (per source)	Bitrate (Mbit/sec)	Throughput (Input packets/sec)	Bitrate (Mbit/sec)	Throughput (Shares/sec)	Bitrate (Mbit/sec)	Throughput (Shares/sec)
2	30	915.53	52.5	6408.69	30	3662.11
4	60	1831.05	105	12817.38	60	7324.22
6	90	2746.58	157.5	19226.07	90	10986.33

Table 5.3 Measurement table for Blu-Ray stream type experiments

	Source Input		Source Output		Drain Input	
Streams (per source)	Bitrate (Mbit/sec)	Throughput (Input packets/sec)	Bitrate (Mbit/sec)	Throughput (Shares/sec)	Bitrate (Mbit/sec)	Throughput (Shares/sec)
1	40	1220.70	70	8544.92	40	4882.81
2	80	2441.41	140	17089.84	80	9765.63

In our experiments, we have provisioned for a 15 second warm-up phase, wherein no measurements are made while each node reaches for a steady state. After the nodes complete their warm-up phase, they enter the measurement gathering phase, which lasts for 60 seconds. Following that, nodes continue to operate for an additional 10 second long grace period to assure that everyone has left their measurement gathering phase, before they terminate.

Another important note on our experimental results is that the only action sequences that we take into account are those that start out and finish their execution during the measurement gathering phase. The typical action sequence on drain nodes consists of receiving, validating and storing packets. Similarly, we define that a new action sequence on source nodes start out when they ‘receive’ an input packet and ends when the produced share-group block is removed from the repository.

5.4. Experimental results

Our early test runs have shown that all nodes in our experimentation produced nearly identical results with each other, even between different runs with the same parameters. Therefore, we have procured our results for this section by making a single run on each configuration and producing the mean value estimates from the readings that we acquire from our node collection.

In order to better evaluate the performance of our system, we have split our findings into four different subsections. First, we examine the packet preparation mechanism. Second, we take a closer look at the reception and data storage mechanisms of the drain nodes. Afterwards, we discuss share and share-group acknowledgements on the source nodes. Finally, we assess the benefits of m-quorum events in our system.

5.4.1. Packet preparation mechanism

We have split our measurements on the packet preparation mechanism into several processing steps in order to better assess their performance and draw further conclusions regarding the efficiency of its implementation. In our experiments, source nodes use four security encoding worker threads to encrypt and authenticate their packets.

Since we had to simulate the arrival of new packets into the system, we introduced a new input thread and queue to our system. The thread reawakens at set time intervals, produces a random input packet and notifies the dispersal thread through the input queue.

Our packet preparation delay consists of the following timespans, as illustrated on Figure 5.1.

- Dispersal queuing: The packet waits for the dispersal thread on the input queue.
- Dispersal computation: The dispersal thread disperses the packet into shares and inserts them to the repository.

- Security encoding queuing: The share waits for a security encoding thread to start processing it. This also includes any delay that the dispersal thread experiences when inserting the task to the security encoding thread's work-queue.
- Security encoding computation: A security encoding thread encrypts and authenticates the share.

The greatest point of interest in these graphs is that, as the input packet arrival rates increase, so do the dispersal and security encoding queuing delays, to the point actually dwarfing the processing delays. The reason for this is that the dispersal thread has to push a new share on each security encoding work-queue upon new input packet arrival. This, in turn, causes thread synchronization to dominate the entire process.

A great portion of the blame goes to having overestimated the contribution of packet security encoding to the delay of the entire process. Granted, dispersing one input packet into seven shares is much faster than having a thread encrypt and authenticate all of them. The actual processing required to encode one of them is not as expensive as we had initially thought, however, which has limited the gains of our packet preparation scheme. Despite this setback, we believe that we should not rush to disregard the benefits of a pipeline.

First, if we find SRTP's security guarantees to be lacking, then we could easily migrate to more secure, albeit computationally expensive alternatives, which will, in turn, make a better use of our packet preparation platform.

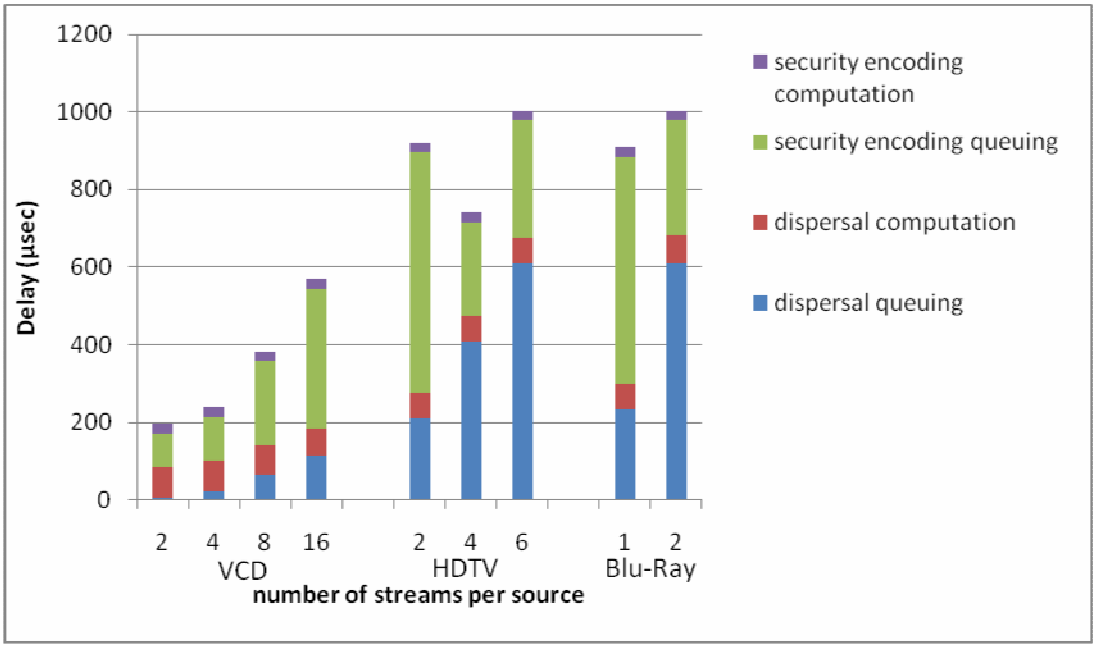

Figure 5.1 Packet preparation delay. Our packet preparation pipeline behaves well when handling up to 10Mbit/sec worth of traffic before lock contention causes the queue management to overcome the parallelization gains. We believe that we can address this issue efficiently, however.

.Second, while we did pay close attention to detail when implementing this mechanism, this was not the case for the input thread, which we introduced fairly recently for the sake of evaluating our system. Much like other similar parts on the drain node implementation, the dispersal thread's performance would greatly benefit from carrying several operations out at the same time. The thread can, for example, split and apply CRS on all outstanding input packets and hand them over to the security encoding threads by only requesting a single access to their work-queues.

Finally, it is reasonable to assume that the calling application can carry out some degree of batching for our protocol, especially when dealing with such enormously high bitrates. Although our messages are bounded by the MTU (Maximum Transmission Unit) constraints of the network, the dispersal thread can still split larger input packets into several smaller ones. It can then, use this knowledge to pass all of them over to the security encoding threads as a single task.

5.4.2. Reception and storage

Similar to the packet preparation mechanism, we are providing separate measurements for each step of the packet reception and storage mechanisms of the drain nodes.

The steps of the drain node reception and storage mechanism consist of the following timespans as illustrated by Figures 5.2 and 5.3.

- Accept delay: The packet reception mechanism validates and decrypts an authentic packet. Its service time remains fixed at 34 μsec and, therefore, it does not show up on either of the figures.
- Store queuing: An accepted message waits for a storage thread to handle it, inside the packet buffer of a receiver stream block.
- Data store: The storage thread finishes recording the payload of a message on the stream data file.
- Meta store: The storage thread finishes updating the stream metadata file.

The store queuing delay makes up for one third to the entire delay it takes to record messages to disk. This is no surprise to us, however, as this is how the system is supposed to behave. Each storage mechanism thread handles new stream activity by recording all messages that lie in the stream buffer. Naturally, new messages continue to mass up inside the packet buffer as the thread is busy handling the most recent activity. Therefore, we expect the mean queuing delay to be equal to half the activity handling delay, which is the sum of the Data store delay and the Meta store delay.

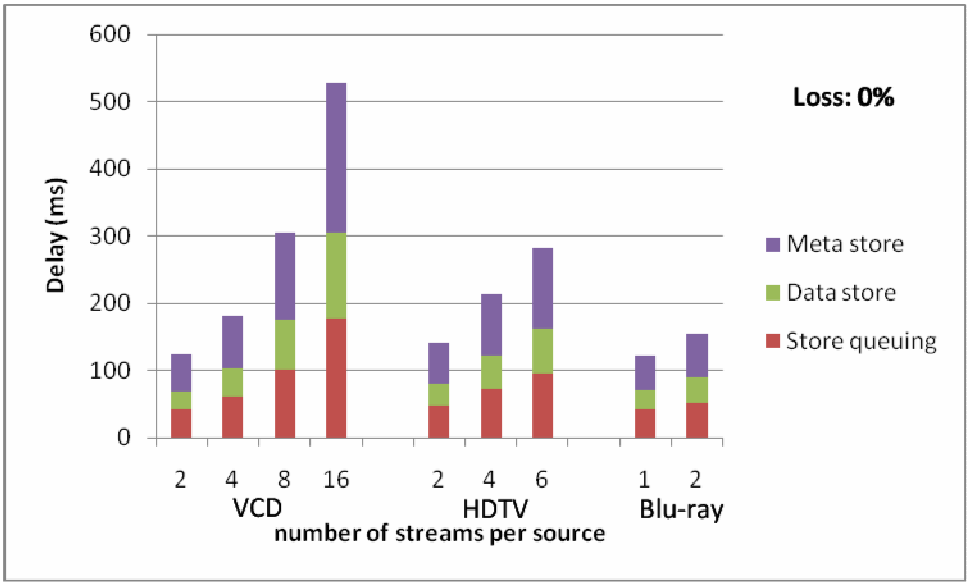

Figure 5.2 We illustrate drain node message write latency at 0% network packet loss ratio. Store queuing contributes 33% to the entire data storing delay, as we expected it to. The system scales pretty well as the stream bitrates increase. Handling a greater number of streams, however, causes the drain storage performance to degrade near-linearly. This is due to the number of files and their respective metadata that the underlying filesystem has to cope with. A working external journaling scheme would have eliminated a great portion of the metadata handling overhead.

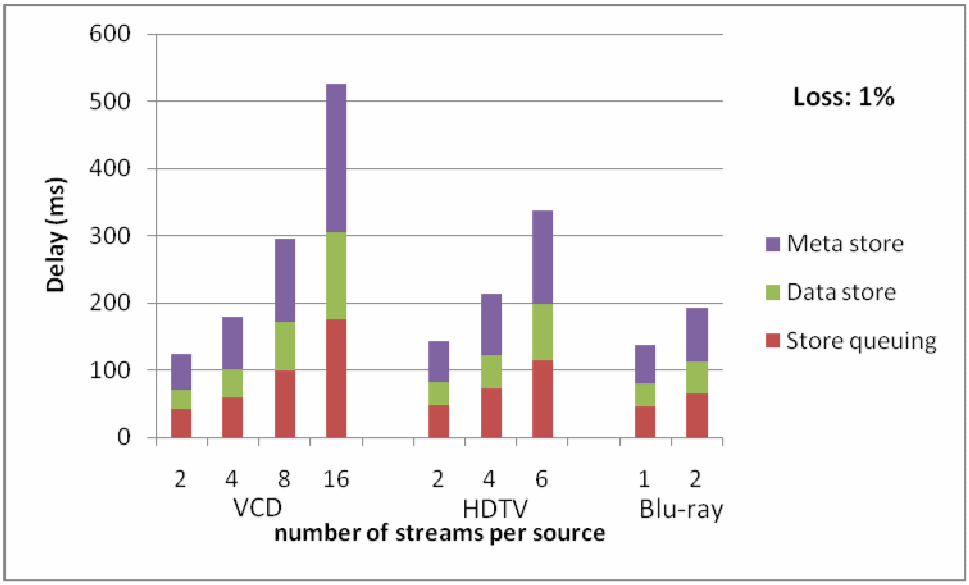

Figure 5.3 Drain node message write latency at 1% network packet loss ratio. Retransmitted packets may require drain nodes to backtrack and update previously stored file blocks to cover the missing byteranges up. This has a similar effect on the data store time to that of storing additional data. Since packet loss on higher bitrate streams causes the retransmitted stream portions to be more spread, it affects these kinds of streams the most.

The storage delay performance of a drain node is mainly affected by the number of streams that it has to handle and less so by their collective bitrate. This is owed to the fact that our file-oriented storage scheme is very heavily dependent on the filesystem's ability to update its metadata nodes. Since we did not use external journaling, the filesystem has to issue additional disk seeks to record new metadata information with each file update. The stream-number parameter affects the number of files and directories that the filesystem has to cope with, which causes the decline in performance. On the other hand, keeping the number of streams fixed and increasing their bitrate only affects data store delay slightly, since the filesystem does not, generally, have to make additional disk seeks in order to store larger batches of data.

Packet loss only marginally affects the performance of our storage scheme. Drain nodes may have to update additional file blocks while handling a new stream activity, whenever a delayed or retransmitted packet arrives. If the previous version of the file block is still available in the filesystem's cache, then it does not need to retrieve it from disk in order to update it. However, the filesystem still has to record the version somewhere on disk, which can have similar effects to recording higher bitrate streams. Naturally, this phenomenon has an even greater impact on high bitrate streams, as they affect a greater number of filesystem blocks per round trip time.

As can be observed on Figures 5.4 and 5.5, the storage throughput performance of a drain node appears to be about the same level as its designated portion of the source node packet preparation throughput for each configuration. The slight difference is owed to two facts. First, as we mentioned earlier, we only measure entire action sequences that lie within the measurement gathering phase. Second, source nodes may often choose not to retransmit lost packets, if an m-quorum event takes place, which reduces the perceived throughput on the affected drain nodes.

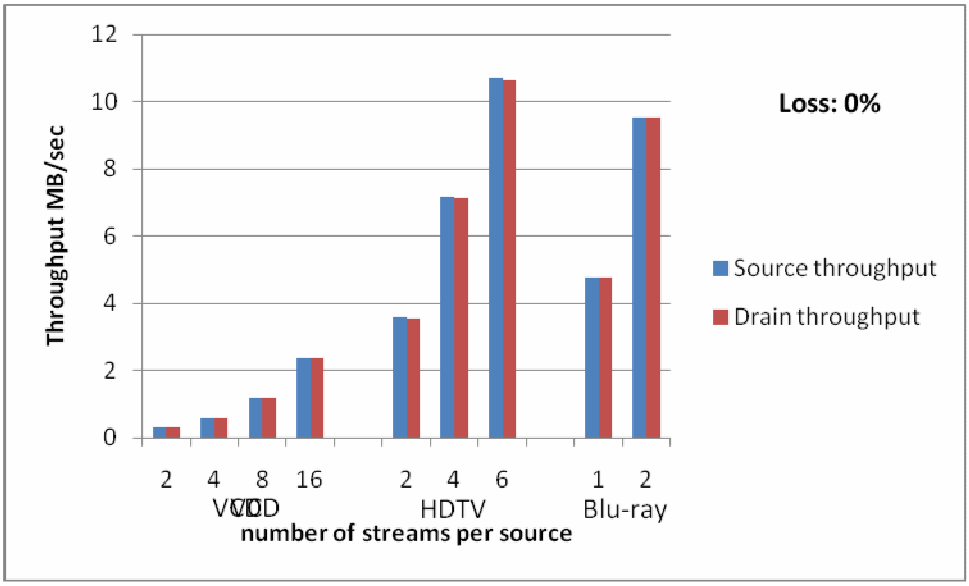

Figure 5.4 We show the drain node write throughput at 0% network packet loss ratio. The slight variation between source throughput and drain throughput is mainly owed to congestion-induced packet loss and m-quorum events. Since source nodes do not retransmit such packets, this lowers the perceived throughput on the drain nodes.

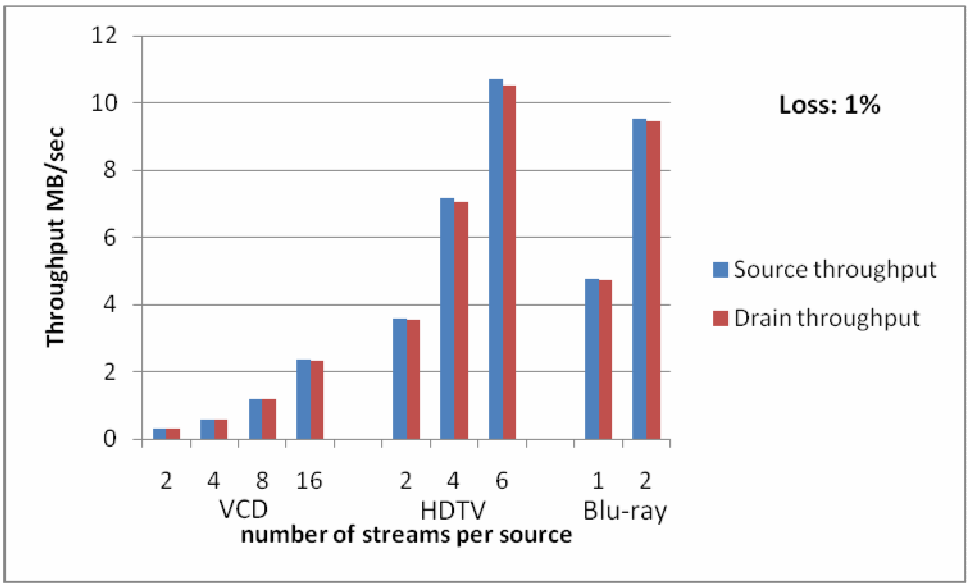

Figure 5.5 Drain node write throughput at 1% network packet loss ratio. Notice that an increased loss ratio further increases the chances of m-quorum events taking place, which further lowers the perceived throughput on the drain nodes.

.

5.4.3. Share and Share-group acknowledgements

Our acknowledgement scheme is not perfect, in the sense that RTT measurements on the source nodes also include any delays that can be attributed to the data storage mechanism. Since disk write operation latency can be quite unpredictable, our RTT estimates are bound to have a high degree of variance, which leads to producing an even higher retransmission timeout. Inevitably, this puts our implementation at a further disadvantage, since its retransmission timeout ranged between 200ms and 750ms, whereas the actual round-trip time was well below 1ms. Regardless of that, however, we were very keen on measuring the effects of m-quorum events on our retransmission scheme.

Following a packet transmission, source nodes either receive a write confirmation from the respective drain node, or a timeout takes place. Despite the fact that such acknowledgements can get lost along the network path, source nodes seem to be reluctant when it comes to retransmitting packets, as they opt out for m-quorum events much more often. The contribution of m-quorum events in deleting shares can be seen on Figure 5.6.

Although m-quorum events are the primary cause for drain nodes receiving fewer shares than they were originally intended to, declaring such events does not always have a measurable effect. For example, source nodes can declare m-quorum events if the acknowledgement from one of the drain nodes lags one tick behind the rest for any share-group in particular. If that is the case, the respective drain node has already recorded the share, and the source will simply ignore the acknowledgement, since the packet has been deleted, anyway. One can draw a comparison between Figure 5.6 and Figures 5.4 and 5.5 to measure the frequency of this effect.

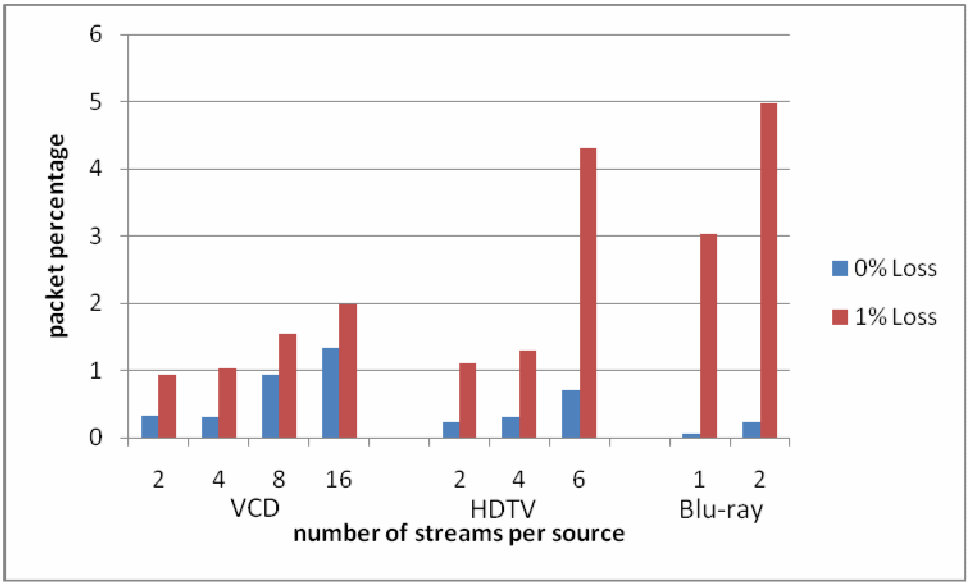

Figure 5.6 Percentage of packets removed due to m-quorum events. Although the frequency of m-quorum events is primarily affected by packet loss – due to the configuration parameters or network congestion – they can also take place when a node takes slightly longer to respond to a write request than its peers, and its acknowledgement misses the timer thread's tick.

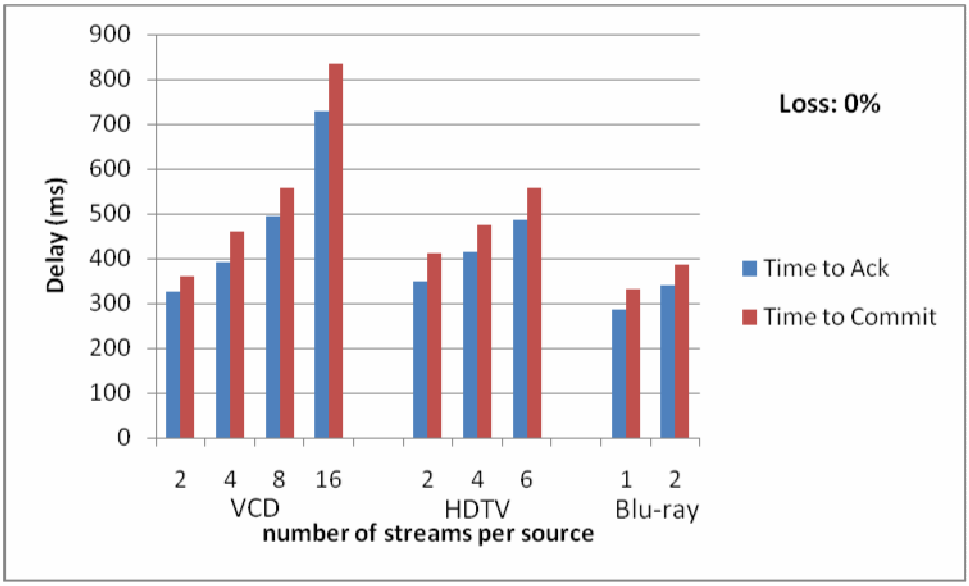

Figure 5.7 Time to Ack vs Time to Commit at 0% packet loss ratio. Time to Ack measures the mean delay between transmitting a share for the first time and the timer thread processing an acknowledgement for it. Similarly, time to commit measures the mean delay between transmitting the packets of a share-group and an m-quorum event taking place. These metrics do not reflect the actual delays of having a share or a share-group respectively safely recorded to disk. They include the time it takes to notify a source node of this action. Therefore, they are also affected by the (source node's) timer thread's and the (drain node's) ack thread's wakeup periods.

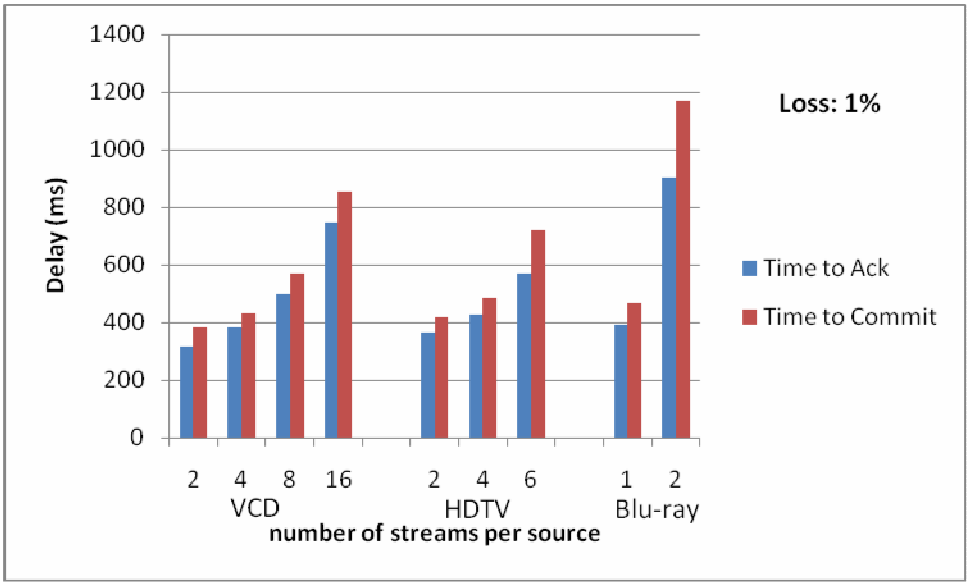

Figure 5.8 Time to Ack vs Time to Commit at 1% packet loss ratio. Naturally, packet loss affects both metrics, since retransmissions only take place after the timeout period has elapsed. This holds, as long as m-quorum events do not drop lost packets altogether.

Figures 5.7 and 5.8 describe the relation between the time that it takes a source node to receive a disk write confirmation for an individual share (time to Ack), and the time it takes to call an m-quorum event for a share-group (time to Commit). Naturally, the time to commit is influenced by the acknowledgements with the highest delay. Time to ack is affected by the write latency on the drain nodes and the wakeup timers of both the acknowledgement creation and timeout handling mechanisms, which are set to 200ms time intervals. A high congestion on the network switch may cause acknowledgments to be dropped. If that occurs, then the source node has to wait for the next one, and also, potentially retransmit the message in question.

Figures 5.9 and 5.10 depict the percentage of retransmissions that a source node carries out and the percentage of duplicates that a drain node receives, respectively. As a rule of thumb, m-quorum events help countering the effects of packet loss on both those metrics. This, however, is not always possible, as drain nodes can sometimes fail to report their acknowledged packets, if their confirmation list becomes too fragmented. We can observe such a case taking place on the {HDTV, 6, 1% Loss} and {Blu-Ray, 2, 1% Loss} experiment configurations. The system

operates under the effects of network congestion, and a random uniform 1% packet loss percentage is all it takes to tip the balance off, for those high bitrate streams. This issue has little to do with system sensitivity on any parameter whatsoever, and is simply affected by the way that we have defined the controllable portion of packet loss. Losses in the real-world are more prone to happen on a sequence of packets rather than random individual ones.

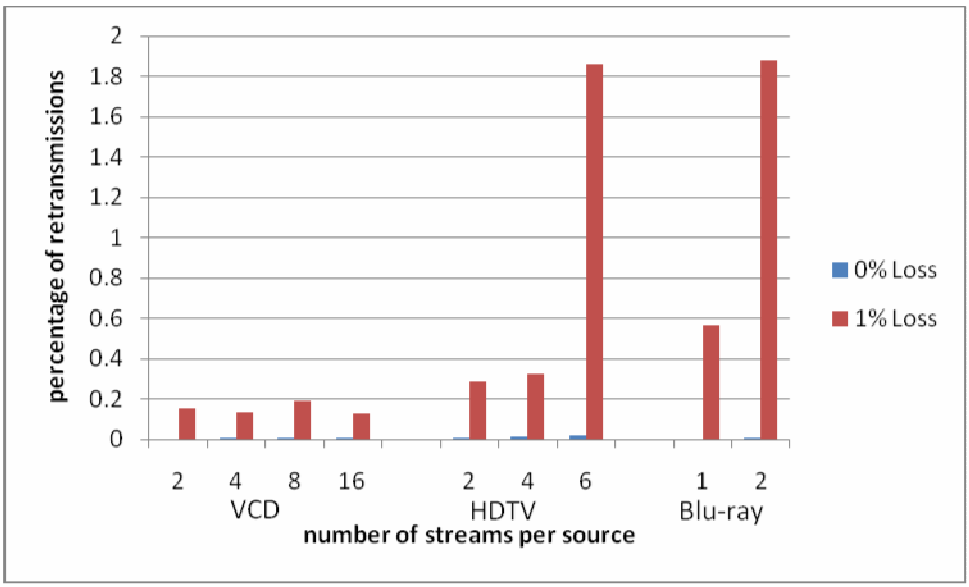

Figure 5.9 Percentage of retransmissions per message. Notice that an 1% increase in packet loss does not necessarily mean an 1% increase in the percentage of packets retransmitted, due to m-quorum events. Packet loss affects higher rate streams the most, as it has a higher chance of fragmenting their confirmation lists during one round trip. If the number of ranges on the confirmation list exceeds the maximum allowed ranges on the acknowledgement packet, this results in increased retransmission and duplicate percentages.

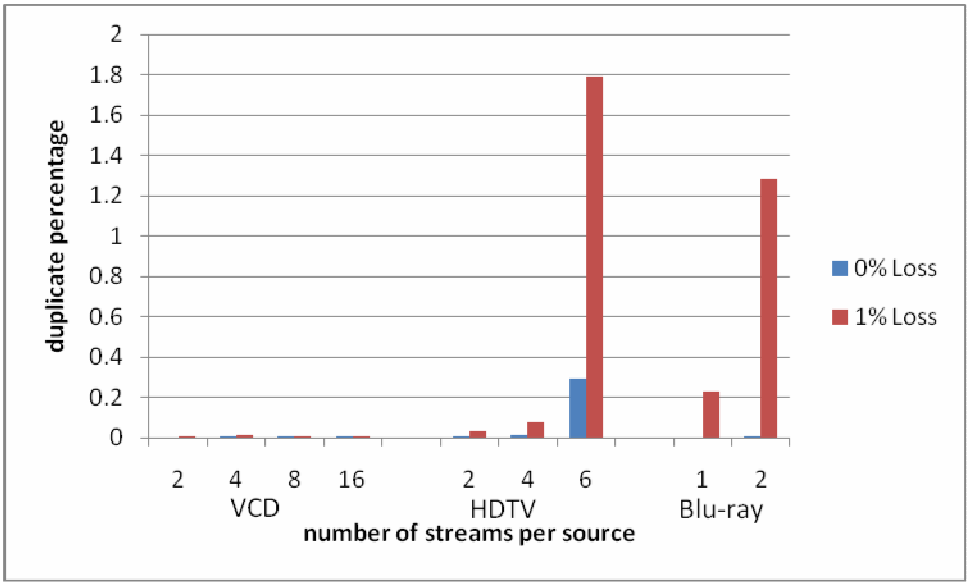

Figure 5.10 Percentage of packet duplicates received by the drain nodes.

Finally, we disabled m-quorum event handling on our system and we have re-run the {VCD, 16, 1% Loss} and {HDTV, 6, 1% Loss} configurations. Figures 5.11 and 5.12 provide a quantitative comparison of the benefits of m-quorum events on packet retransmissions and packet duplicates. The percentage of packet retransmission on the no-quorum setting immediately steps up to the 1% packet loss percentage, since every packet loss causes packet timeout and retransmission on the source nodes. This, however, is not the case for the m-quorum configuration, as it manages to retain the number of retransmissions quite low.

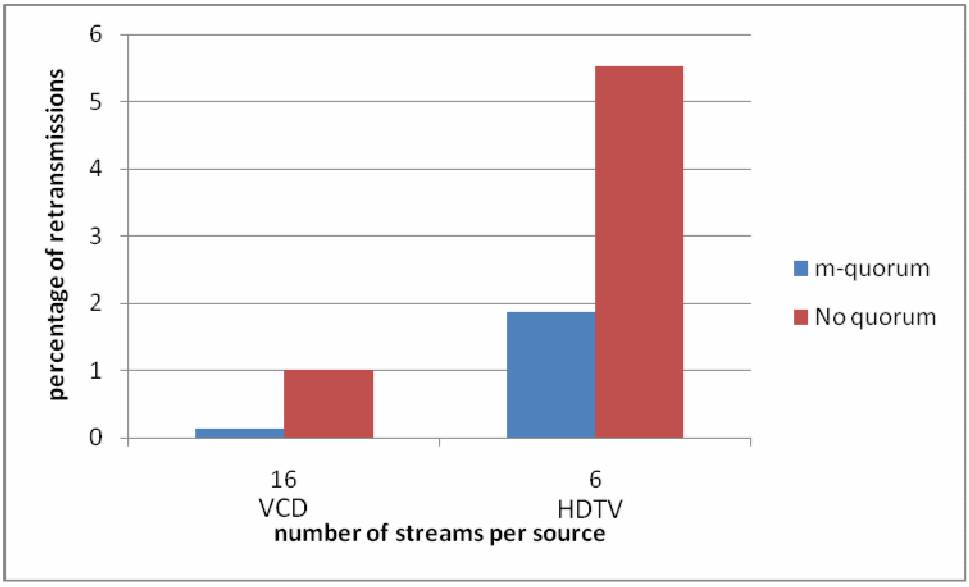

Figure 5.11 Percentage of retransmission per message, without m-quorum. The use of m-quorum events allows us to negate the effects of packet loss on both a low packet loss scenario {VCD, 16, 1% Loss} and a high loss one {HDTV, 6, 1% Loss}. In both cases, the number of retransmissions on the no-quorum setting cannot be lower than 1%, which is the configuration's loss parameter value.

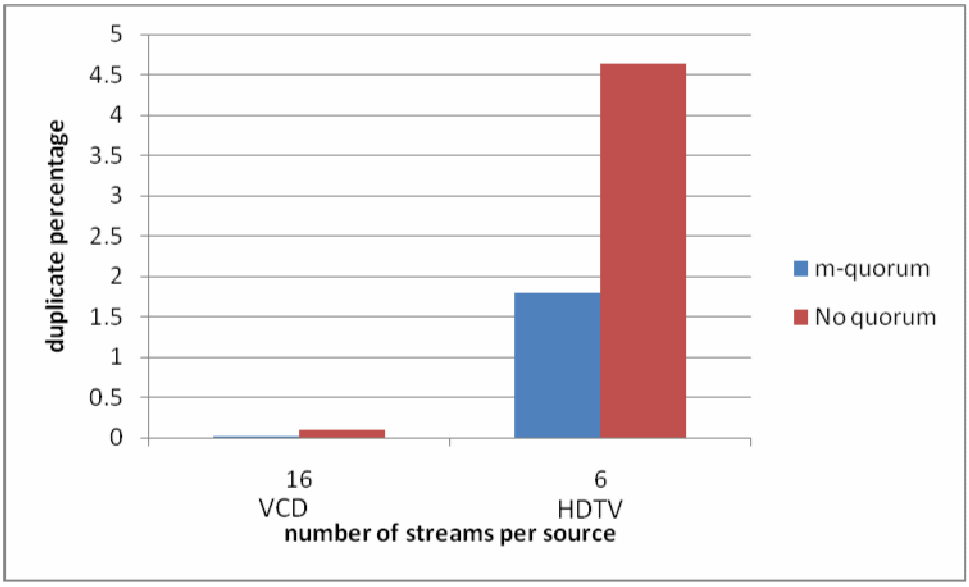

Figure 5.12 Percentage of packet duplicates received by the drain nodes, without m-quorum.

5.5. Summary

We have evaluated our design choices, and how they affect the performance of our system. Although we initially expected that drain node storage scheme would become the definitive performance bottleneck of our system, this did not happen due to packet congestion on network. Regardless of that, however, our system has exhibited good traits in its performance.

The packet preparation mechanism of the source nodes does indeed have some issues with queuing management overhead between the dispersal thread and the security encoding threads. As long as the collective stream input bitrate remains well below the 10Mbit/sec limit, our pipeline reduces packet preparation delay. We reckon that we can tweak the pipeline structure slightly to have source nodes process packets in a manner similar to how drain nodes handle new stream activity, without altering its work-conserving nature.

Drain node throughput remains high enough, despite the lack of batching on the source nodes. As our results have shown, their performance scales very well with increasingly high stream bitrates. Their write operation latency is primarily affected by the number of streams, which determines the number of files they have to access and modify at the same time.

An external journaling scheme would have been of great assistance to handling this metadata-update-heavy workload. Although this greatly boosted the performance for the majority of our drain nodes, we were forced to abandon it, since two of them reacted unusually to it. Doubtlessly, it will be very interesting to repeat our experiments once we manage to resolve this issue.

We evaluated our retransmission mechanism, when it has to perform under the effects of network packet loss, under varying degrees of load. Although we should see to that our retransmission timeout estimate is updated according to the network round trip time measurements, the use of m-quorum events has partially offset that issue. These effectively negate the effects of packet loss on the number of retransmissions, packet duplicates and time to commit share-groups.

Finally, it should be noted that the uniform packet loss distribution scheme that we were forced to fall back on does not reflect actual network conditions. This scheme limits the benefits of our selective acknowledgement protocol on higher bitrate streams, as it spreads packet loss evenly across the streams and produces multiple fragments on the drain nodes' confirmation lists. Since acknowledgements can only convey a limited number of confirmed packet index ranges, this forces the drain nodes to under-report their received packets and trigger an increase in retransmission and duplicate metric percentages, if an m-quorum event does not occur.

CHAPTER 6. RELATED WORK

We are not the first to come up with the suggestion of dispersing data to prevent it from being leaked to an adversary. As a matter of fact, there is a wide variety of object-based storage systems which focus on improving different aspects of the erasure-coded scheme each to fulfill their purposes. This particular area garners considerable attention ever since the topic of cloud storage has recently risen into prominence. The set of operations of such storage systems consists of reading, writing, data reconstruction and usually some sort of audit protocol which they use to detect data corruption.

The fundamental difference between our system and such general purpose storage systems is the fact that we deal with stream storage exclusively. This enables us to optimize our system in a degree which is far greater than that of other related systems and, thus, drastically reduce the window of opportunity during which a single node failure can cause data loss.

Since such generic storage systems have to cater for data updates, they need to handle their messages in a sequential manner. Therefore, they can safely rely on TCP to handle its in-built bytestream ordering without sacrificing the reliability of a system – in regards to committing data swiftly – as a whole. Naturally a single lost packet on a high-latency link will inevitably delay any successive ones. If the sending node were to crash at that point, then none of these messages' contents would ever become committed. Even if the designers of such systems chose to follow the same UDP-oriented approach that we have, then they would still have to delay packets on the

application layer, so as to ensure that all update requests are handled in a proper ordering.

In addition to packet ordering, such storage systems also have to address the issue of synchronizing their storage nodes to propagate updates properly. Since object reconstruction requires us to decode *k* shares of the same version together, the whole process of committing an update to an object can easily span two round trip times. During the first phase, the sender node transmits the newest shares of an object to the storage nodes, which all store a separate copy of the new share, without overwriting the old one. The sender node, then, has to wait for the write confirmations from all the storage nodes before moving to the next phase. Naturally, if some nodes are too slow at replying, then this delay would inevitably affect the entire set of nodes, and not just simply the ones causing it. Following that, the sender node broadcasts a commit message to all nodes, which signals that the system has advanced to the new version and that it is safe for storage nodes to discard their old version of that specific block.

POTSHARDS [15] uses Shamir's secret split algorithm to disperse information to its storage nodes. Once a storage node concludes a write operation on its local media, it returns the block location of the object share on its filesystem. The owner of an object collects the replies from the storage nodeses and builds an index map, which they can use to regain access to their data. Since the index itself is liable to loss, POTSHARDS uses approximate pointers to link object block shares together into a cross-storage-node chain. The presence of such block pointers enables the system to reconstruct the index map efficiently, if all storage nodes consent to that. Furthermore, an adversary cannot use their access to a storage node to request object blocks from the next storage node from the following storage node in the block chain. Since the pointer is only approximate, it is likely that the adversary requests invalid data blocks, which would in turn raise an alarm and lead to their presence becoming detected.

Safestore [5], instead, focuses on upholding data reliability on an erasure-coded set of unreliable storage service providers (SSP). The system employs an audit protocol which periodically transmits a set of challenges for random file block locations. Each of the SSPs in turn responds with a hash of the challenge message and the data of the

requested block for each challenge that it receives. Following that, the auditor picks a subset of the challenges that it has issued and requests the actual data from the SSPs in question. It reconstructs the original data, checks for data corruption, and then checks the hash responses it has collected from the previous phase to detect any lying SSPs.

The Federated Array of Bricks (FAB) introduces the m-quorum notion, which it uses to provide a greater degree of availability during its read and write operations. The original purpose of the m-quorum is to avoid any unnecessary delays which are caused by unresponsive nodes during the first phase of the storage operation. As we have described throughout this document, not only have we included a mechanism that handles m-quorum events for our system, but we have also modified our transport protocol to make a better use of it by using the advance pointer fields to maintain synchronization with the opposite end.

Finally, HAIL [2] is the most recent example of cloud-storage oriented system, which we have only had the opportunity to study as this thesis was nearing its completion. HAIL uses a clever application of MACs to detect and prevent adversaries from corrupting shares on a set of storage nodes to cause loss of data. Although the adversary is only able to control up to $m / 2$ nodes at any time, they can gradually move over to corrupt the entire set of storage nodes, as they see fit. Each storage node uses a *server code* to detect tampered data on its local share. Finally, a trusted party audits the entire collection of storage nodes together, reconstructing selected portions of data by using an *error-correction* algorithm. Obviously, the success of the system as a whole relies on the ability of the audit protocol to detect errors before the number of corrupted shares overcomes the capabilities of the *error-correction* algorithm, which is $m / 2$ errors or m erasures.

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

7.1. Conclusions

We have focused our work on building the write operation of our system, basically from scratch, which we believe it to be very well optimized with both the stream and erasure-coding setting in mind.

We are particularly pleased with the packet preparation mechanism which both provides a very unique form of versioning on the theoretical basis, and a very well performing implementation, which we shall definitely reuse. Our information splitting and dispersal scheme enables sender nodes to operate on a non-batching, work-conserving basis while preserving the ability to seek specific byteranges of stored information. We have achieved this without endangering the reliability of our system or forcing nodes to coordinate among themselves to handle share version changes, which would have undoubtedly hampered its performance.

Our information arrangement scheme enables source nodes to skip a great bulk of the computations involved in other transport protocols that aim to provide security. Instead of treating message confidentiality and integrity as different facets of data transfer, we have combined them with the other aspects of our protocol. The fact that we have opted not to separate acknowledgements from our internal replay list structure enables our protocol to make the best use of the available receiving window.

Furthermore, this arrangement offers an intuitive approach to determining which packet retransmissions would benefit the reliability of our system the most. Since erasure-coding enables us to retrieve data, even when a portion of the storage nodes have failed, source nodes should opt to retransmit share-groups that are not yet recoverable from the system. These m-quorum events help preserve that our write operation remains available for as long as only a small portion of drain nodes are inactive. In addition to that, they also limit the effects of network packet loss on retransmissions and duplicate packets, as we have demonstrated in our evaluation.

Drain nodes can process any message they receive immediately and completely irrespective of any packet reordering that may occur during transit. They do not have to maintain any re-ordered packets into memory for as long as it takes their preceding ones to arrive. Since we are no longer restricted by memory, we can specify an infinitely large receiving window, which is only bounded by sequence number wraparounds.

In addition to that, packet reordering does not result in any packet loss whatsoever, unlike other security protocols out there, such as IPSec, which employ a bitmap approach to their replay lists. In such cases, any packet reordering which causes packets to fall behind the threshold limit of the bitmap inevitably results to dropping certain packets as replayed. That is, despite the fact that this may be their first arrival on the receiving node.

Drain nodes exhibited good traits in their performance. The latency of their write operation scales very well with high bitrates and is mostly affected by the number of serviced streams. Since handling multiple streams introduces a metadata-update-heavy workload on our drain nodes, we believe that using external journaling will help counter this issue effectively.

7.2. Future work

Of course, there is still enough room for improvement which will help increase the performance of our protocol, and put it into perspective with the entire system.

Undoubtedly, future work on this part of the system should be that of defining the initialization protocol and refining the retransmission protocol, which still functions on a very basic mode of operation.

The initialization protocol should both help set up a new session between all communicating pairs, and attempt to restore a previously running session, assuming that a node had previously crashed and is rejoining it. Drain nodes should help a rejoining source node determine the most recent stream byte offset value that it has used, before it can start streaming new data to them. Similarly, a rejoining drain node should probably make the source node re-key, and possibly re-index all unacknowledged packets in its repository directed to that same drain node. If we allow ourselves to be too careless when determining drain node crashes, an adversary could find the opportunity of having source nodes re-encrypt their packets constantly. That is assuming that m-quorum events do not manage to purge these packets from the repository.

Acknowledgement packets should merge the information carried by both the reception and the confirmation list, so that source nodes can respond to network failures more efficiently. Source nodes should stop retransmitting packets the moment they acquire a *reception acknowledgement* for it. They should, however, retain it in memory until they receive a *confirmation acknowledgement* for that particular index. If the source node somehow detects that a drain node has crashed, then it will have to ignore any *reception acknowledgments* it has previously acquired, and retransmit those packets until it they become acknowledged anew. Even if a packet gets re-indexed, drain nodes can still use the bookkeeping structure to detect this kind of duplicates, in the same manner that they consult the structure to resolve overlapping boundaries between successive messages.

Such acknowledgments should also help decouple RTT estimates from the storage write time. Source nodes will be able to react to network failures much faster, which may result in getting more shares stored. Furthermore, storage write times can vary significantly between successive operations and, therefore, we should not rely on them to provide a worthwhile estimate. Additionally, assuming that the disk storage time exceeds that of a round trip time, the retransmitted packet can make it in time to the receiver block packet buffer and become committed along with its peers.

The retransmission protocol also lacks a robust congestion avoidance mechanism. Currently, we are very keen on maintaining our approach of transmitting packets as soon as the packet preparation mechanism finishes its work on them. This is owed to the fact that new share-groups do not have any of their packets committed to disk yet, and we should seek to improve it. This does not apply to packets that time out, however, which allows us to become more conservative and help reduce packet congestion on the drain nodes.

These changes should also have an impact on the frequency of m-quorum events. The packet retransmission algorithm will be using an RTO estimate which is bound to be smaller than the time it takes for confirmation acknowledgements to arrive. Therefore, packets are more likely to be retransmitted, rather than become accountable for m-quorum events. This is not something that we should worry about, since declaring m-quorum events is not a panacea. It causes the metadata structure on drain nodes to inflate and include additional ranges as the stream data file becomes more and more fragmented with gaps. Additionally, these gaps will have to fill up somehow, and that is only through the data reconstruction procedure. Therefore, we believe that we should take a more conservative approach with m-quorum events; we can use them to clear the packet repository up when a node crashes, or even help the retransmission algorithm choose which packets the system would benefit from, the most.

The next step should be that of fleshing out the remaining operations of the system. This will boil down to introducing some sort of scheduling that handles disk accesses for reading, writing and reconstructing stream data in an efficient manner. Naturally, before this can be made possible, we should probably introduce, or better yet, modify

an already existing filesystem to have it act as the cornerstone for any future endeavors in this area.

REFERENCES

[1] M. Baugher, D. McGrew, M. Nashlund, E. Carrara and K. Norrman. “The Secure Real-Time Transport Protocol (SRTP)”, RFC 3711, March 2004.

[2] K. D. Bowers, A. Juels, A. Oprea. “HAIL: A High Availability and Integrity Layer for Cloud Storage”,

[3] N. Ferguson, B. Schneier. “Practical cryptography”, Wiley, 2003.

[4] S. Kent, R. Atkinson, “Security Architecture for the Internet Protocol”, RFC 2401, November 1998.

[5] R. Kotla, L. Alvisi, M. Dahlin. “SafeStore: a durable and practical storage system”, 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference, p 1-14, June 17-22, 2007.

[6] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, “TCP Selective Acknowledgement Options”, RFC 2018, October 1996.

[7] T. K. Moon. “Error Correction Coding: Mathematical Methods and Algorithms”, Wiley-Interscience, 2005.

[8] J. S. Plank, “A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems”, Software—Practice & Experience; 27(9):995-1012, September 1997.

[9] J. S. Plank and Y. Ding, “Note: Correction to the 1997 Tutorial on Reed-Solomon Coding” Software, Practice & Experience, vol. 35, no. 2, pp 189-194, Feb. 2005.

[10] J. S. Plank, J. Luo, C. D. Schuman, L. Xu and Z. W. O’ Hearn. “A performance evaluation and examination of open-source erasure coding libraries for storage”, In 7th USENIX FAST, pages 253-265, 2009.

[11] W. R. Stevens. “UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI”, Prentice Hall, 1998.

[13] W. R. Stevens. “TCP/IP Illustrated, Volume 1: The Protocols”, Addison-Wesley, 1994.

[14] W. R. Stevens, Gary R. Wight. “TCP/IP Illustrated, Volume 2: The Implementation”, Addison-Wesley, 1995.

[15] M. W. Storer, K. M. Greenan and E. L. Miller. “POTSHARDS: Secure long-term storage without encryption”, In USENIX Annual Technical Conference, June 2007.

[16] V. Vasudevan, A. Phanisaye, H. Shah, E. Krevat, D. Andersen, G. Ganger, G. Gibson, B. Mueller. “Safe and effective Fine-grained TCP Retransmission for Datacenter Communication”, In Proc ACM SIGCOMM, Barcelona, Spain, August 2009.

SHORT VITA

Dimitri Melissovas was born in Ioannina, Greece in 1985. He was admitted to the Computer Science Department of the University of Ioannina in 2003. He received his BSc degree in Computer Science in 2007, and is currently a postgraduate student at the same department. He is a member of the Systems Research Group of the University of Ioannina ever since 2007. His main research interests pertain to the fields of Networks, Parallel Computing and Security.