A QUEST FOR HIGH-PERFORMANCE PEER-TO-PEER LIVE MULTIMEDIA STREAMING

BY

MEA WANG

A THESIS SUBMITTED IN CONFORMITY WITH THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY (Ph.D.), DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING, IN THE UNIVERSITY OF TORONTO.

COPYRIGHT © 2008 BY MEA WANG. ALL RIGHTS RESERVED.
Abstract

Demands for multimedia content, one form of digital content, are continuously increasing at a phenomenal pace, as video features are commonly available on personal devices, such as iPod, cell phone, laptop, PDA, and Blackberry. The streaming service poses unique bandwidth and delay challenges to application designers. The size of a typical video content is usually orders of magnitude larger than that of any other type of content, resulting in high demands for bandwidth contribution from the content providers. Even more challenging, the content must be delivered to end hosts in real time to maintain smooth playback, i.e., the content must be transmitted at a satisfactory rate. In this thesis, we present our research towards a high-quality peer-to-peer live streaming system that utilizes network coding, a novel technique that permits coding at every peer, which has proven benefits in file dissemination applications. To ensure the practicality of our work, it is our imperative objective to conduct all experiments under realistic settings.
Special dedication to my mother, Julie

In memory of Speedy
Acknowledgments

First and foremost, I would like to express my profound gratitude and appreciation to my advisor, Professor Baochun Li, for his continued guidance and support in the past six years. He has offered me insightful suggestions not only on academic research, but also on teaching, interpersonal skills, leadership skills, and career choices.

I would like to send my great appreciation to my thesis external appraiser, Professor Ekram Hossain, and the committee members, Professor Alberto Leon-Garcia, Professor Shahrokh Valaee, Professor David Lie, and Professor Eyal de Lara, for their patient and valuable comments on this thesis.

I reserve my heartfelt thanks for my friends and colleagues in the iQua research group. Thank you for being parts of this amazing journey from the very beginning of my research career. To my best friends, Ella Huang and Peter Zhang, thank you so much for your supports — friendship forever. To Ella, the four years that we shared were quite pleasant and memorable.

A special “Thank You” from the bottom of my heart to my wonderful mother for her unconditional love and support throughout my life, and for the sacrifices she made to provide opportunities for me. This thesis is dedicated in memory of my lovely dog, Speedy, who had been with me on every special occasion in the past 10 years. He holds an irreplaceable spot in my memory and heart forever. I will take good care of his friend, Snoopy. Hugs and kisses ...
Contents

Abstract

Acknowledgments

List of Tables

List of Figures

1 Introduction 1

2 Related Work 7

2.1 P2P Streaming Systems 7

2.2 Network Coding 12

2.3 Practical Network Coding 16

3 Crystal: An Emulation Framework for Peer-to-Peer Streaming Systems 21

3.1 Existing Frameworks 23

3.2 Design Objectives 25

3.3 Architecture and Design 26
| 5.1.3  | Segment Scheduling                            | 81 |
| 5.1.4  | Streaming Protocol                            | 84 |
| 5.1.5  | Network Coding Plug-in                      | 86 |
| 5.2    | Implementation                                | 90 |
| 5.3    | Evaluation of Network Coding in P2P Streaming | 94 |
| 5.3.1  | Performance of Network Coding                | 95 |
| 5.3.2  | Scalability                                   | 97 |
| 5.3.3  | Tuning Density and Aggressiveness             | 99 |
| 5.3.4  | Tuning Experimental Parameters                | 103|
| 5.3.5  | When Bandwidth Demand Meets the Supply        | 107|
| 5.3.6  | Effects of Peer Dynamics                      | 109|
| 5.4    | Summary                                       | 114|

6  \( R^2 \): A High-Quality Peer-to-Peer Streaming System 115

| 6.1    | The Design of \( R^2 \)                       | 116 |
| 6.1.1  | Design Objectives                             | 117|
| 6.1.2  | The Essence of \( R^2 \): Random Push with Random Network Coding | 118|
| 6.1.3  | Timely Feedback from Downstream Peers         | 121|
| 6.1.4  | Random Selection of Downstream Peers          | 127|
| 6.1.5  | Design Objectives Revisited                   | 127|
| 6.2    | Implementation                                | 128|
| 6.3    | Performance Evaluation                        | 130|
| 6.3.1  | Scalability                                   | 133|
| 6.3.2  | Buffering Levels                              | 135|
| 6.3.3  | Initial Buffering Delays                      | 136|
6.3.4 Effects of Random Network Coding .................................. 139
6.3.5 Tuning the Random Segment Selection Algorithm ................. 141
6.3.6 When Bandwidth Demand Meets the Supply .......................... 142
6.3.7 Effects of Peer Dynamics .............................................. 145
6.4 Summary ............................................................................ 147

7 Concluding Remarks ............................................................... 149
  7.1 Contributions ...................................................................... 149
    7.1.1 The P2P Prototyping and Emulation Framework ................. 149
    7.1.2 Practical Network Coding ............................................ 150
    7.1.3 P2P Streaming with Network Coding ............................... 151
  7.2 Future Work ....................................................................... 151
    7.2.1 Streaming Algorithms .................................................. 151
    7.2.2 Empirical Studies under Practical Settings ...................... 153

Bibliography ............................................................................... 155
## List of Tables

3.1 A summary of pros and cons of the real-world deployment, simulation, and emulation ........................................... 22  
3.2 The main body of the network thread .................................................. 29  
3.3 The main body of the engine thread .................................................. 30  
3.4 Emulation of bandwidth limits using `select()` .................................... 35  
3.5 The event scheduler ........................................................................ 38  
3.6 An example of the events file ............................................................... 39  
3.7 The event types ............................................................................... 39  
3.8 An example of the topology file ............................................................ 41  
3.9 The API of Crystal ........................................................................... 44  
3.10 The skeleton of the relaying protocol using Crystal ............................... 45  
4.1 Multiplication of two bytes on GF(256) in Lava ........................................ 55  
5.1 The segment scheduling algorithm ....................................................... 83  
5.2 The skeleton of Lava using Crystal ...................................................... 93  
6.1 Communication overhead brought by the buffer map in both Lava and $R^2$ 125  
6.2 The skeleton of $R^2$ using Crystal ...................................................... 131
6.3 The average time (in seconds) taken to fill the priority region in sessions involving different numbers of peers ........................................ 138
6.4 The time (in seconds) takes to fill the priority region when tuning the streaming rate .......................................................... 145
List of Figures

1.1 An example of P2P live multimedia streaming . . . . . . . . . . . . . . 3
2.1 An example of push-based P2P streaming systems . . . . . . . . . . . . 8
2.2 An example of pull-based P2P streaming systems . . . . . . . . . . . . 11
2.3 Network coding improves session throughput: well-known examples . . 14
2.4 An example of network coding in P2P file dissemination applications . . 19
3.1 The Crystal architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.2 The design of the network layer . . . . . . . . . . . . . . . . . . . . . . . 29
3.3 The design of the engine layer . . . . . . . . . . . . . . . . . . . . . . . . 31
3.4 Bandwidth emulation in Crystal . . . . . . . . . . . . . . . . . . . . . . . 34
3.5 The pseudo-source in Crystal . . . . . . . . . . . . . . . . . . . . . . . . 37
3.6 An example of the playback buffer . . . . . . . . . . . . . . . . . . . . . . 42
3.7 A sample experimental setup with chains of peers served by a server . . 46
3.8 The accumulative streaming rate from all peers on a single server . . . 47
3.9 The average streaming rate on each peer from a single server, with stan-
dard deviation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
3.10 The CPU usage as the per-link upload bandwidth increases on a single 
server. The number of peers is fixed to 10 ........................................... 49
3.11 The CPU usage as the number of peer increases on a single server. Per 
link bandwidth is fixed to 50 KB/sec .................................................. 49
3.12 The average streaming rate on each peer from 10 servers, with standard 
deviation ................................................................. 50
3.13 The maximum sustainable streaming rate as the message size increases . 51
4.1 An application of network coding in P2P mesh networks ................. 54
4.2 Coding rate when tuning the block size and number of blocks ........ 57
4.3 Performance of randomized network coding .............................. 59
4.4 Average transmission time and the percentage of peers that have success- 
fully decoded after receiving 100 coded blocks, without linear dependence 
checks ................................................................. 62
4.5 An example where linearly dependent blocks may be received by a peer . 64
4.6 Performance of linear dependence checks .............................. 65
4.7 Average transmission times and the number of blocks required for success- 
ful decoding, as a percentage of the number of original blocks, with linear 
dependence checks ................................................................. 67
4.8 An example of progressive decoding with Gauss-Jordan elimination . 69
4.9 The performance of network coding library with progressive decoding . 71
5.1 An example of network coding in P2P streaming systems ................. 78
5.2 Segment scheduling in Lava .................................................. 82
5.3 An example of the pull-based streaming protocol ........................ 85
5.4 Another example of the pull-based streaming protocol ................. 86
5.5 Network coding in Lava ........................................ 87
5.6 An example of the pull-based streaming protocol with network coding 89
5.7 Another example of the pull-based streaming protocol with network coding 89
5.8 Lava in Crystal .................................................. 91
5.9 Coding performance for a session with a streaming rate of 64 KB/sec . . 96
5.10 Scalability in terms of network size .............................. 98
5.11 The transmission and playback status in a 64 KB/sec streaming session on 264 peers ........................................ 100
5.12 Effects of the aggressiveness in a 64 KB/sec streaming session with 264 peers ........................................ 102
5.13 Average playback skips and bandwidth redundancy when tuning density (aggressiveness set to 25%) .......................... 103
5.14 Effects of the playback buffer size and the initial buffering delay in a 64 KB/sec streaming session with 264 peer .... 105
5.15 Effects of the segment size in a 64 KB/sec streaming session with 264 peers ........................................ 106
5.16 Effects of the bandwidth supply and demand in a 64 KB/sec streaming session with 264 peers ............................ 108
5.17 The PDFs of the peer join rate distributions ...................... 110
5.18 Effects of the peer join rate in a 64 KB/sec streaming session with 264 peers .............................. 111
5.19 The PDFs of the peer lifetime distributions ...................... 112
5.20 Effects of the peer lifetime length in a 64 KB/sec streaming session with 264 peers ........................................ 113
6.1 Selecting a segment for transmission in $R^2$ .................... 120
6.2 The playback buffer in $R^2$ ...................................... 121
6.3 Bandwidth waste due to propagation delay of the buffer map ........... 123
6.4 An example of the streaming protocol in $R^2$ ............................. 124
6.5 An illustrative comparison between a traditional P2P streaming protocol and $R^2$ ................................................................. 126
6.6 $R^2$ in Crystal ................................................................. 129
6.7 Average playback and bandwidth status on each peer in a 64 KB/sec streaming session deployed to P2P networks of various sizes, ranging from 88 to 792 ................................................................. 134
6.8 Uplink bandwidth consumption on the dedicated streaming source during a 64 KB/sec streaming session deployed to P2P networks of various sizes, ranging from 88 to 792 ................................................................. 135
6.9 Average buffering level during the sessions and on each peer in a 64 KB/sec streaming session deployed to networks that consist of 88 and 792 peers . 136
6.10 The average playback skips and buffering level during the session when tuning the initial buffering delay, in a 64 KB/sec streaming session with 800 peers ................................................................. 137
6.11 Average playback skips when tuning the length of the priority region, in a 64 KB/sec streaming session with 800 peers ................................................................. 139
6.12 The average playback skips and buffering levels during a 64 KB/sec streaming session, with a fixed number of blocks in each segment ................ 140
6.13 Average playback skips and buffering levels during a 64 KB/sec streaming session with a fixed block size ........................................ 141
6.14 The PDFs of the probability distributions used in segment selection ... 142
6.15 The effects of different probability distributions in the segment selection
algorithm, in a 64 KB/sec streaming session .................................. 143
6.16 Effects of the balance between bandwidth supply and demand with 800
peers ........................................................................... 144
6.17 The PDFs of the peer lifetime distributions ................................. 145
6.18 $R^2$ is more resilient to peer dynamics as compared to network coding and

Vanilla, in a 64 KB/sec streaming session involving 800 peers .......... 146
Chapter 1

Introduction

In 1994, ABC’s World News Now was the first television show to be broadcast over the Internet. Internet radio company AudioNet started the first continuous live webcasts in January, 1998. Later in September 1999, a regional telecommunications operator in the U.K. became one of the first companies in the world to launch an IPTV over DSL broadband interactive TV service, also known as Peer-to-Peer (P2P) multimedia streaming. As the broadband service has become available to more than 200 million households worldwide since 2005, demands for P2P streaming are growing at a phenomenal pace. Software, like PPLive [1], PPlite [2], PPStream [3], Coolstreaming [4], UUSee [5], QQLive [6], SopCast [7], TV Ants [8], and many more, allows users to watch streaming media (TV channels) based on BitTorrent-like [9] P2P technologies. The streaming service poses unique bandwidth and delay challenges to application designers. The size of a typical video content is usually orders of magnitude larger than that of any other type of content, resulting in high demands for bandwidth contribution from the content providers. Even more challenging, the content must be delivered to end hosts in real time to maintain smooth playback, i.e., the content must be transmitted at a satisfactory rate.
P2P networks formed by end hosts on the Internet have recently emerged to replace the traditional client-server approach to multimedia streaming. The end hosts are commonly referred to as peers. As participating peers contribute their upload bandwidth capacities to serve other peers, the load on dedicated streaming servers is significantly reduced. Thus, each peer can potentially be a server for the content it has received. Fig. 1.1 illustrates an example of P2P live multimedia streaming, in which three events are concurrently streamed across the Internet. Each event corresponds to a single session, also referred to as a channel in end-host applications. One or more servers, known as the streaming source, are dedicated to serve each session. Peers across the Internet participate in the sessions according to their interests. Without loss of generality, in this thesis, we consider a single live P2P session with one streaming source (usually under the administrative control of a service provider), and a large number of peers. Peers arrive at and depart from a session in unpredictable ways. The streaming content is coded into a constant bit rate, usually in the range of 38 – 50 KB/sec in real-world streaming applications.

In a typical P2P multimedia streaming system, peers must communicate with each other in order to avoid receiving redundant content at each peer. This is commonly referred to as data reconciliation, and results in excessive communication overhead and lack of flexibility to network dynamics. Bandwidth loss due to such overhead leads to low session throughput and prolongs the end-to-end delay from the source to peers. The problem is more noticeable as the number of peers grows, especially in the “flash crowd” scenario. In such testing situations, certain receivers of the data streams may experience deteriorating quality, reflected in playback interruptions and streaming rate degradations.

In the past few years, a few state-of-the-art algorithms have been proposed towards
Figure 1.1: An example of P2P live multimedia streaming
efficient multimedia streaming, namely ConCast [10], SpreadIt [11], NICE [12], CoopNet [13], ZigZag [14], PeerStreaming [15], CollectCast [16], CoolStreaming [4], ChunkSpread [17, 18], Chainsaw [19], and PRIME [20]. The successful deployments of PPLive [1] and CoolStreaming [4] have demonstrated the feasibility of P2P multimedia streaming over the Internet. The essential advantage of live P2P streaming is to dramatically increase the number of peers a streaming session may sustain with several dedicated streaming sources. As one of the most significant benefits, P2P streaming enjoys the salient advantage of scalability in live sessions, where upload capacities on streaming sources are no longer the bottleneck.

To date, the quality of such service is still not comparable to that offered by cable or satellite networks. Due to the inefficient use of bandwidth, end users often experience long waiting during channel surfing, pauses during playback, and connection losses. These problems are often manifested in wireless networks or networks consisting of mainly peers equipped with DSL/Cable-modems, since each peer in the network has limited bandwidth and may arbitrarily join or leave the network. In this thesis, we seek to improve the quality of multimedia streaming in practical P2P networks. Orthogonal to the design of video codec [21, 22, 23] that compresses the content with no or little loss in fidelity, we focus on the interaction among peers in the network — communication protocols and content exchange. The objective is to achieve high streaming quality while keeping the communication overhead low and content exchange efficient.

One of the key enabling technologies in this thesis is network coding. It has been originally proposed in information theory [24, 25, 26], and has since emerged as one of the most promising information theoretic approaches for improving performance in P2P networks. The upshot of network coding is to allow coding at intermediate nodes in
information flows. It has been shown that random linear codes using Galois Fields are sufficient to implement network coding in a practical network setting [27]. Avalanche [28, 29] has demonstrated — using both simulation studies and realistic experiments — that network coding may improve the overall performance of P2P content distribution. The intuition is that, with network coding, all pieces of information are treated equally, without the need to identify and distribute the “rarest piece” first. Given the proven advantages of network coding in P2P content distribution systems, we are determined to take full advantages of it in our quest for high-quality multimedia streaming systems.

In the quest for a high-performance peer-to-peer live streaming system, we first designed and implemented Crystal, an emulation framework, in preparation for analyzing the performance of various streaming systems in emulated peer-to-peer networks. We then examined existing coding libraries and implemented network coding operations in the most efficient way. With supports from Crystal and the network coding library, we developed two streaming systems to conduct a systematic study on the attributes of network coding in multimedia streaming systems. Finally, we proposed $R^2$, a brand new peer-to-peer streaming algorithm, that takes full advantages of network coding and offers significant performance improvements in terms of server load, initial buffering delay, communication overhead, playback smoothness, and resilience to peer dynamics. The main contributions of this thesis include practical applications based on network coding and the Crystal emulation framework for peer-to-peer algorithm development and evaluation. While the focus of my research is practical network coding, the research results encompass the areas of middleware architectures, content delivery networks, and multimedia networks.

The remainder of this thesis is organized as follows. In Chapter 2, we review the
existing P2P multimedia streaming systems. Chapter 3 presents Crystal, our emulation framework for real-world P2P multimedia streaming applications. Chapter 4 incorporates network coding operations as a plug-in library into the Crystal framework. Chapter 5 presents our systematic study of the advantages and tradeoffs of network coding in a typical P2P streaming system. Chapter 6 proposes a new streaming protocol that takes full advantage of network coding. Chapter 7 concludes this thesis with a recap of the contributions and an outlook on the future work.
Chapter 2

Related Work

2.1 P2P Streaming Systems

The most traditional approach to multimedia streaming is established through a client-server setup, in which the streaming content is broadcast to all participating peers. Among the existing literatures [30, 31, 16, 32] on client-server streaming systems, PROMISE [32] is an outstanding example that provides flexibility and robustness to network topology variations, by maintaining a set of backup servers in addition to active ones. In PROMISE, a centralized algorithm implemented on each client performs an exhaustive search among all servers to select the best set of active ones based on bandwidth and loss rate of the corresponding P2P links. This algorithm also computes the streaming rate offered by each active server. Since clients make independent decisions, certain network segments and servers might become overloaded, resulting in high loss rates. The affected clients are required to apply the server selection algorithm again to select a different set of active servers. Despite its simplicity, this approach is not scalable because of the limited uploading bandwidth and computing power at the dedicated streaming servers.
To alleviate the bandwidth demand on the source, a number of push-based, also known as tree-based, approaches are stemmed from the philosophy of IP multicast. In such a paradigm, peers are organized into one or more multicast trees rooted at the source. The source decomposes the original content into a set of small data pieces and “push” them to descendants among the trees. Due to bandwidth limitations, usually a subset of the data pieces can be transmitted between peers; hence, a receiver rarely receives the entire content from the same parent. A simple example of such push-based approach is depicted in Fig. 2.1, in which the streaming source splits the content into two sub-streams — streams 1 and 2. The participating peers are organized into two multicast trees, one for each stream. The streams are pushed from the source to all peers along the corresponding trees.

Figure 2.1: An example of push-based P2P streaming systems

To better utilize the incoming bandwidth of a peer in multicast trees, CoopNet [13] encodes the content into separate streams, such that every subset of them is decodable,
that is, every incoming link of the receiver carries innovative data pieces. Instead of using the coding method, SplitStream [33] and Mutualcast [34] impose certain structures among the multicast trees and distribute disjoint portions of the content along each tree. Despite the advantages of push-based approaches, constructing and maintaining a well-organized distribution tree burdens peers and links with heavy control overhead, especially in a dynamic environment, as demonstrated by SpreadIt [11]. To relieve the burden of control overhead in the multicast trees to a certain degree, NICE [12] and ZIGZAG [14] manage peers as multi-layer hierarchical clusters, i.e., a tree of clusters. Although the push-based approaches lead to short delays in distributing the content, it is not generally employed in real-world streaming applications, mainly due to the complexity and difficulty in maintaining the structured network topology, especially under the presence of peer dynamics.

In sharp contrast, the pull-based, also known as mesh-based, approaches do not enforce any rigid structure among the peers. Instead, connections are established dynamically based on the content availability at each peer. The streaming content is presented as a series of segments, each representing a short duration of playback. The content exchange in this approach is best described as a “data-driven” or “swarming” style of multicast. In data swarming, each peer advertises to its neighbors the segment availability in its buffer, and the neighbors explicitly request the segments as needed. The primary advantages here are simplicity in maintaining peer connectivities and robustness in dynamic networks. Nonetheless, the delay in delivering the live content to each participating peer is inevitably increased, ascribed to the periodical exchange of segment availability among the peers.

Fig. 2.2 presents an instance of the pull-based P2P streaming systems, in which the
source has two segments for streaming, and each segment corresponds to 1 second of the playback. In this synthetic network, each peer, except the source, has upload capacity of $1.5S$, where $S$ is the size of a segment. The source has an upload capacity of $1.5S$ on each outgoing connection, with a maximum of 3 connections. Without loss of generality, we assume no end-to-end delay on each link. Initially, five peers join the live streaming session. Since the source can handle at most three peers at a time, only three out of five peers are connected to the source. At time 0, source starts to transmit one segment on each of its outgoing links. After 0.66 second, these three peers have completely received one segment each and start to help the source to serve other participating peers. Five more peers join the session after 1.32 seconds into the session, they pull segments from any peers that have useful segments and sufficient bandwidth to them. After 1.98 seconds into the session, 17 segments have been sent around the network, i.e., the system throughput is 8.58 segments per second.

Bullet [35], for example, constructs a mesh on top of a multicast tree to improve overall bandwidth utilization. The underlying tree structure provides an environment for periodic dissemination of peer identifiers and content availability. CoolStreaming [4] completely abandons the tree structure and employs a gossiping protocol for peer discovery. A peer in CoolStreaming [4] maintains not only a list of neighboring peers, but also a summary of available content on its neighbors. Based on these information, segments are scheduled to be streamed from the appropriate neighbors, while striving to meet the playback time. The performance of pull-based approaches highly depends on the choice of peer discovery protocols and data swarming strategies. Hybrid designs [36, 17, 18, 20] are proposed to combine the better resilience to dynamics from pull-based approaches, and the better delay and stability from push-based approaches. Essentially,
2.1. P2P STREAMING SYSTEMS

Figure 2.2: An example of pull-based P2P streaming systems
the connections are initialized based on the content availability at each peer, and then portions of the original content are pushed onto each connection.

Despite the simplicity and performance improvements of these work, they exhibit the common problem of data reconciliation. Departing from the push versus pull communication paradigm, this thesis examines P2P multimedia streaming from a new perspective, where network coding plays an important role. With network coding, operations are performed in Galois Fields, which preserves the size of the original data, i.e., no additional bandwidth is required [25, 26]. Gkantsidis et al. [28, 29] have shown that network coding is beneficial in file sharing applications. It significantly improves the resilience to peer dynamics and the downloading throughput, compared to when network coding is not used. Given these improvements in file sharing applications, it is a natural research direction to apply network coding in P2P multimedia streaming applications, since they share many common characteristics.

2.2 Network Coding

Network coding has been originally proposed in information theory in 2000 [24], and has since received extensive research attention. The essence of network coding is a paradigm shift to allow coding at intermediate nodes between the source and the receivers in one or multiple communication sessions, assuming that communication links are free of errors using a lower-layer protocol, such as ARQ-based transport-layer protocols (e.g., TCP). The fundamental assumption of error-free links has relieved the research on network coding from addressing the challenges of interference, which often lead to the most difficult problems in the field of network information theory [26].

The fundamental insight of network coding is that information to be transmitted
from the source in a session can be inferred, or decoded, by the intended receivers, and does not have to be transmitted verbatim. It also focuses on the coding capabilities of intermediate nodes. With the ability to code at relay nodes in a session, we may forward, replicate and code information flows, as opposed to traditional commodity flows, where only forwarding is allowed. In recent research literatures on network coding, it is a well known result that network coding — by using linear codes only — may achieve better network throughput in some of the network topologies [37].

The benefit of network coding with respect to improving throughput in directed graphs can be best illustrated in the “Butterfly” example, as shown in Fig. 2.3(a). In this example, each link in the topology has unit capacity, and the source $S$ seeks to maximize its session throughput to both $R_1$ and $R_2$ in a multicast session. Without network coding, optimal throughput can be achieved by solving the problem of steiner tree packing, which leads to a flow rate of 1.5. With network coding, as shown in Fig. 2.3(a), node $X$ is able to code its input messages $a$ and $b$ into $a+b$, with the $+$ operation defined in a finite field, leading to an effective end-to-end throughput of 2. In wireless networking scenarios requiring wireless information exchange, network coding is also shown to be helpful [38]. In Fig. 2.3(b), if $R_1$ sends $a$ to $R_2$, while $R_2$ sends $b$ to $R_1$, it requires four units (e.g., time slots) of transmission without coding. With network coding and the wireless broadcast advantage, the intermediate node $S$ may simply broadcast $a+b$ to both $R_1$ and $R_2$, making it feasible to complete the exchange in three time units.

In contrast with source erasure codes, network coding applies coding at intermediate relay nodes throughout the network, while source erasure code applies coding operations at the terminal nodes (sender and receivers) only. Source erasure codes are a class of error-correction codes that can help recover errors or losses in source data blocks from
2.2. NETWORK CODING

Figure 2.3: Network coding improves session throughput: well-known examples

redundant encoded check blocks. The classic Reed-Solomon code [39] was invented over half a century ago and has seen ubiquitous applications in data storage and communication [40]. The main idea behind Reed-Solomon code is to view the source data blocks as coefficients of a polynomial of a certain degree $k$, and then generate $n > k$ samples, in the form of encoded blocks, out of them. The coefficients (source data blocks) are over-determined due to over-sampling, and any $k$ out of the $n$ encoded blocks can be used to recover them using Lagrange interpolation. Ideally, we wish the check blocks to have the same size as the source blocks. This is achieved by viewing the source blocks as numbers within a certain finite field, and by applying the (linear) coding operations over that field. Since numbers in the same finite field have fixed-length representation, data overflow is no longer a problem.

The major drawback of Reed-Solomon code is its high time-complexity in encoding and decoding, especially if one wishes to have a software implementation. The root of
the problem is that Reed-Solomon code has a dense encoding matrix over a large field. In view of this, Tornado code [41, 42] was introduced. Tornado codes are designed to operate over the binary field only, \textit{i.e.,} its only coding operation is bit-wise exclusive-or (XOR, $\oplus$). It also has a sparse encoding matrix. As a result, Tornado codes enjoy much faster encoding and decoding over Reed-Solomon codes. The price that it pays to achieve this, is that the coding scheme is no longer optimal, in that slightly more than $k$ blocks need to be collected in order to decode and recover the $k$ source blocks.

Tornado codes usually encodes $k$ source blocks into $n = ck$ encoded blocks, for some constant small integer $c$. This constant is referred to as the stretch factor or the rate of the coding scheme. In many networking applications, we wish to have the source continuously transmit out data blocks, with the property that any $k$ of them can be decoded to obtain the source data. Such a source can be viewed as a digital fountain, and codes that enable this property are called fountain codes [43, 44]. Essentially, we need to remove the rate limit we have in Tornado codes. For that reason fountain codes are also called rateless codes [45].

Three classes of well-known rateless codes are Raptor codes [46], LT codes [47], and Online codes [48]. They are all rateless erasure codes that employ XOR operations only, they both employ a two-stage encoding scheme, and they are both near-optimal. The two stage encoding scheme consists of an outer encoding phase and an inner encoding phase. During the outer phase, auxiliary blocks are generated and are appended to the set of original source blocks. Then the outer phase is applied to the output of the inner phase, during which each encoded block is XORed from $d$ randomly chosen blocks. Specific variants of rateless codes differ in how they generate a random number $d$ for each output block. The decoding may be performed conversely in a two stage fashion, or may be done
2.3. Practical Network Coding

To practically implement network coding, one needs to address the challenges of computing coding coefficients to be used by each of the intermediate nodes in the session, so that the coded messages at the receivers are guaranteed to be decodable. This process is usually referred to as code assignment. Although deterministic code assignment algorithms have been proposed and shown to be polynomial time computable [52], they require extensive exchanges of control messages, which may not be feasible in dynamic P2P networks. As an alternative, Ho et al. [27] proposed the concept of randomized network coding, in which a node transmits on each outgoing link a linear combination of incoming messages, with independently and randomly chosen coding coefficients over some finite field.

Since the landmark paper on randomized network coding by Ho et al. [53], there has been a gradual shift in research focus in the area of network coding, from theoretical studies on achievable flow rates and code assignment algorithms [54, 55], to more practical
studies on applying network coding in a practical setting [56, 57, 58, 59, 60, 51]. Such a shift of focus has been marked by the work titled “Practical Network Coding” [56], in which the authors have concluded that randomized network coding can be designed to be “robust to random packet loss, delay, as well as any changes in network topology and capacity.” It was shown that sessions with randomized network coding can achieve “close to the theoretically optimal performance.” The highly visible Avalanche project by Microsoft Research [28] has further illustrated that randomized network coding can be used for bulk content distribution, in competition with BitTorrent [9, 61], one of the most successful P2P content distribution protocols at the time of this writing. The work has made the claim that “the performance benefits provided by network coding in terms of throughput can be more than 2-3 times better compared to transmitting uncoded blocks.” In this sense, one may conclude that network coding can indeed be practically implemented, and does offer significant advantages as compared to BitTorrent [62, 63, 64].

Another recent work from the Avalanche project at Microsoft Research has sought to demonstrate the feasibility of network coding with a real-world implementation in C# [29]. In this work, a P2P content distribution application with network coding has been used to establish a P2P session among 100 clients, spreading across the Internet. With its experiments in a session lasting around 38 hours to distribute a 4.3GB file, the paper has concluded that “network coding incurs little overhead, both in terms of CPU and I/O, and it results in smooth and fast downloads.” In particular, with respect to the computational overhead of network coding, the conclusion was drawn from the observation that each of the clients only consumes about 20% – 40% of its CPU throughout the session. This work has not compared the performance of its network coding implementation with any BitTorrent-like content distribution protocols without using coding. Compared to its
upbeat and optimistic views, our experiences in Chapter 4 echo some of its observations on CPU overhead, but with more cautious overall conclusions.

Let’s consider a simple example of network coding in file dissemination applications. A file is divided into 8 original blocks, illustrated as 8 white squares on the source, in Fig. 2.4. The encoding operation is performed among all blocks. Each encoded block, shown as small squares in the darker shade in Fig. 2.4, is a linear combination of the original blocks. In order to make an insightful comparison with the case in Fig. 2.2, we assume the file size is $2^S$, where $S$ is the size of a segment in Fig. 2.2. Similarly, each peer, except the source, has upload capacity of $1.5S$, and the source has upload capacity of $1.5S$ on each outgoing connection, with up to 3 connections.

Initially, five peers join the downloading session. Since the source can handle at most three peers at a time, only three out of five peers are connected to the source. At time 0, source starts to transmit coded blocks on each of its outgoing links. After 0.5 second, these three peers have completely received three coded blocks and start to help the source to serve other participating peers. As we will show in Chapter 4, a peer can start data swarming as soon as one coded block is received. Here, we assume that there is a small delay in content availability exchange and peer discovery. Five more peers join the session after 1 second into the session, they pull coded blocks from any peers that have sufficient bandwidth to them. Meanwhile, two out of the first three peers have completely downloaded the file, they stop receiving from the source so that the newly joined peers can share the spare bandwidth from the source. After 1.5 seconds into the session, 54 blocks (i.e., 13.5 segments of size $S$) have been sent into the network, resulting in higher system throughput — 9 segments per second.

From this example, we observe that network coding allows transmitting content in a
Figure 2.4: An example of network coding in P2P file dissemination applications
finer granularity, which leads to faster information propagation. This is only feasible with network coding, since all coded blocks are equally important. To swarm uncoded blocks among peers, we must design the protocol to exchange block-level segment availability — updating buffer status more frequently. Moreover, when a peer departs from the network, only a few small blocks are lost at its neighboring peers. These peers can quickly switch to other peers, since coded blocks on any peer are all equally useful. Given the efficient data swarming and better resilience to peer dynamics in file dissemination applications, we are interested in studying the feasibility of network coding in P2P streaming systems.
Chapter 3

Crystal: An Emulation Framework for Peer-to-Peer Streaming Systems

Since one of the objectives in this thesis is to assess the practicality of network coding and our proposed streaming protocol under real-world settings, we must develop our systems and conduct all experiments in a practical environment. Ideally, the best route to evaluate new protocols is to actually implement and deploy them across the Internet, on real peers with home broadband connections. We note that such an approach, while being realistic, may not offer sufficient scientific evidence from which conclusions are drawn. First, due to the highly dynamic nature of peers in P2P networks, experimental results in this setting may not be analyzed and diagnosed as tuning parameters or designs. It is also difficult for other researchers to independently reproduce and verify the results. Second, without a dedicated commercial launch, it is hard to include a large number of peers with DSL/cable Internet connections since this group of peers are even more dynamic than institutional users. Third, it may be difficult to collect the logistics and statistics measured at each peer. If centralized logging servers are used, it is usually not scalable to a large number of
peers in the session. Finally, CPU and bandwidth — the most important resources that lead to the advantage of the P2P architecture — are heterogeneous and highly dynamic, as the availability of CPU cycles and bandwidth is subject to the fluctuating load of concurrent tasks on the same host. Some existing evaluations of P2P protocols made use of excellent experimental testbeds such as PlanetLab [65] and Netbed [66], which are examples that suffer from drawbacks of peer dynamics, non-DSL peers, and lack of scalability and supports for streaming applications.

Due to the complication in implementation and deployment under practical network settings, most academic researchers resort to simulation studies, which do not always appropriately and accurately reflect the reality of a large number of bandwidth-limited peers behind home broadband connections, with their arrival and departure dynamics, and actual traffic being relayed and transmitted. In P2P streaming, the complexity of protocols — in the form of computational requirements on peers and sophisticated message exchanges — has completely ruled out the possibility of using simulations.

Table 3.1: A summary of pros and cons of the real-world deployment, simulation, and emulation

<table>
<thead>
<tr>
<th></th>
<th>Real-world deployment</th>
<th>Simulation</th>
<th>Emulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real traffic</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Real time computation</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Control over CPU and memory</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Reproducible results</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Emulation has recently received major attention as processing units are becoming increasingly faster at cheaper cost. A network emulator usually involves real-time computation and real traffic. It gives system developers a better sense on how algorithms will perform in reality. Table 3.1 compares the pros and cons of real-world deployment, simulation, and emulation. Given the cluster of servers at our deposit, an emulator can
offer us the advantages of both real-world deployment and simulation. The only concern is the complexity and accuracy in the implementation of the emulator. For these reasons, we designed and developed Crystal, an emulation framework for real-world P2P multimedia streaming systems. In order to collect insightful and convincing performance results, we must include a significant number of largely uninteresting, yet mandatory, elements, such as bootstrapping protocols, efficient message forwarding mechanisms, timed and periodic event schedulers, TCP and UDP network socket programming, multi-threaded programming, exception handling of failures and disconnections, as well as facilities to control, troubleshoot, and measure the performance metrics. We designed Crystal to be a flexible framework for developing, testing, and troubleshooting new P2P streaming systems in a controlled server-cluster environment. With Crystal, we seek to offer ease of use, rapid experimental turnaround, scalability in terms of number of emulated peers, as well as the capability of emulating realistic P2P environments.

3.1 Existing Frameworks

The Flux OSKit project [67] modularizes common OS components such that they are reusable, to facilitate rapid development of experimental OS kernels. Similarly, Crystal provides a reusable set of components, organized in the Crystal engine and libraries, to facilitate rapid prototyping and evaluation of P2P protocols in server clusters.

There exist previous work on using virtual machines, such as VMWare [68], Xen [69], User-Mode Linux [70]. The main objective was to support the deployment of full-fledged applications over a virtual network (e.g., vBET [71]), or in emulation testbeds and environments to test network protocols in a virtualized and sandbox environment (e.g., Netbed [66, 72] and ModelNet [73]). In particular, ModelNet [73] has introduced a set of
ModelNet core nodes that serve as virtualized kernel-level packet switches with emulated bandwidth, latency and loss rates. Crystal has similar objectives, but is designed to be simpler to deploy, more scalable on a single physical cluster node, and much easier to develop with. Although Crystal supports emulating hundreds of peers on a single physical cluster node, its implementation are achieved at the user level beyond the abstraction of sockets. To deploy and test a new protocol using Crystal, a P2P researcher does not need to configure the system with multiple virtual machines, or patching and recompiling the kernel. We strive to make the Crystal platform neutral. At the time of this writing, Crystal is portable across major UNIX variants (Linux, FreeBSD and Mac OS X), as well as Microsoft Windows.

MACEDON [74] and MACE [75] feature new languages to describe the behavior of a protocol in distributed systems, from which actual code can be generated using a code generator. As a result, MACEDON allows protocol designers to focus their attention on the semantics of the protocol itself, and less on tedious implementation details. Crystal, however, is based on a drastically different design philosophy. MACEDON attempts to minimize the lines of code to be developed by the protocol developer, by using a new language to specify the characteristics of a specific category of P2P protocols, including DHT search and end system multicast. In contrast, Crystal seeks to maximize the freedom and flexibility of designing and implementing new protocols with C++, especially allowing complex computational tasks (such as coding) to be performed as part of the new P2P protocols. Crystal and MACEDON reflect different tradeoffs between having the fewest lines of code and allowing maximum flexibility.
3.2 Design Objectives

To ensure the practicality and correctness of our emulation framework, Crystal, we captured characteristics of typical real-world P2P multimedia streaming systems.

▷ Any two peers have the potential to establish a connection over the Internet. Each connection is associated with the available bandwidth and end-to-end delay.

▷ Peer arrivals and departures in a particular session need to be emulated, which leads to the challenges of maintaining up-to-date dynamic network connections.

▷ Actual network traffic and realistic streaming servers need to be supported to emulate practical streaming sessions. Hence, a potentially large number of TCP connections and UDP flows need to be efficiently managed by each peer.

▷ In existing systems, a dedicated tracking server is employed to bootstrap each participating peer. It provides peers with the initial information of the P2P network upon their arrival to the streaming session.

▷ A peer maintains a playback buffer that consists of segments to be played in the immediate future. The buffer refreshes itself by removing obsolete content to make room for future content.

▷ Each system has its own streaming protocol to facilitate peer discovery, buffer status update, and content exchange.

It is our imperative objective to make Crystal meeting common characteristics of general P2P streaming systems listed above.
3.3 Architecture and Design

*Crystal* features the following highlights. First, *Crystal* provides a set of common elements required in any P2P streaming system, including multi-threading, message switching, timed and periodic event scheduling, network socket programming, and exception handling. These elements are organized into three layers: network, engine, and algorithm. Second, *Crystal* is custom-tailored for server clusters. As shown in Fig. 3.1, each instance of a *Crystal* stack corresponds to an emulated peer. A server can easily accommodate from one to hundreds of emulated peers, depending on available physical resources such as CPU and memory. Finally, any two emulated peers can establish TCP or UDP connections. *Crystal* addresses the lack of reality in simulation and the lack of controllability in real-world deployments, and is capable to *emulate* any peer upload and download capacities, end-to-end delays, as well as peer arrivals and departures. This framework crystallizes the past two years of our work towards implementing a framework for P2P network emulation, in 10279 LOC (lines of code including comments, not scripts).

The design of the engine is based on our previous work on *iOverlay* [76], a lightweight middleware framework for developing overlay applications over the Internet. The design of *iOverlay* employed a “thread-per-connection” concurrency model, using blocking socket operations and forking a new thread for each TCP connection. For each emulated peer, the number of threads is the same as the number of active connections, leading to excessive overhead of thread context switching. Though this model may be appropriate when building overlay applications on actual end hosts, it is not the most scalable way to build emulated peers in a server cluster with limited CPU resources.

In contrast, *Crystal* employs two threads for each peer, the *network thread* and the
engine thread. Similar to event-driven designs by Pai et al. [77] and Welsh et al. [78], this approach buys in the extra complexity of event-driven implementation in order to achieve higher performance. With less number of threads, the threading overhead is significantly decreased, which makes Crystal more scalable than the alternative VMM-based solutions when emulating a large number of peers on a single server.

The network thread, also referred to as network in Fig. 3.1, is responsible for handling queues of new incoming and outgoing messages, emulating bandwidth and delay, monitoring socket status, as well as detecting arrivals and departures of peers. The engine thread, including both engine and algorithm in Fig. 3.1, is mainly responsible for processing incoming messages, performing protocol-specific logic, emulating peer arrivals and departures, as well as handling all periodic or timed events. The engine and the network naturally form a consumer-producer relationship with respect to messages. When an upstream peer sends a message to peer $p$, the network first detects the incoming traffic and
receives the message into the corresponding queue. The engine then takes the message and passes it to the appropriate message handler, implemented in either the engine or the algorithm. In this case, the network is the producer, and the engine is the consumer. To send a message from peer $p$, the message is usually created by the algorithm, and is then queued into the network via the engine. The engine does not do any processing in this case, except looking up for the appropriate queue in the network. The network layer then sends messages to the downstream peer. Herein, the network is the consumer, whereas the engine is the producer. Overall, the network thread is dedicated to manage network-level events, while the engine thread processes and produces messages. This design leads to very fast responses to socket events, and subsequently makes emulating high-throughput peers possible.

### 3.3.1 The Network

The network thread provides low-level network I/O services in Crystal. It handles basic sockets-level tasks related to new incoming connections, exception handling related to broken connections, as well as the actual communication (send and receive operations) for all active connections. The network thread supports both connection-oriented stream sockets (TCP) and connectionless datagram sockets (UDP). As implied in Fig. 3.2, each TCP connection from an upstream peer or to a downstream peer is associated with a queue, implemented as a circular buffer of messages. UDP traffic has its own dedicated incoming and outgoing queues: we maintain one pair for UDP data messages, and another pair for UDP control messages.

As shown in Table 3.2, the main body of the network thread consists of a loop running for a peer’s lifetime. In each iteration, the network thread first prepares a list of “active
sockets”, including the server sockets, sockets from all “ready outgoing connections” and sockets from all “ready incoming connections”. The readiness of a connection is computed based on the per-link bandwidth limits of the corresponding TCP connections, and on the per-peer bandwidth limits. We postpone our discussion of bandwidth emulation to Sec. 3.4.1.

Table 3.2: The main body of the network thread

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>while peer is alive</td>
</tr>
<tr>
<td>2</td>
<td>add TCP listening socket and UDP sockets to <code>activeSockets_in</code></td>
</tr>
<tr>
<td>3</td>
<td>for every socket <code>sock_i</code> of a ready incoming connection <code>i</code></td>
</tr>
<tr>
<td>4</td>
<td>add <code>sock_i</code> to <code>activeSockets_in</code></td>
</tr>
<tr>
<td>5</td>
<td>for every socket <code>sock_j</code> of a ready outgoing connection <code>j</code></td>
</tr>
<tr>
<td>6</td>
<td>add <code>sock_j</code> to <code>activeSockets_out</code></td>
</tr>
<tr>
<td>7</td>
<td><code>select(activeSockets_in, activeSockets_out, timeout)</code></td>
</tr>
<tr>
<td>8</td>
<td>for every ready socket <code>sock_i</code> ∈ <code>activeSockets_in</code></td>
</tr>
<tr>
<td>9</td>
<td>receive message <code>m</code> from <code>sock_i</code></td>
</tr>
<tr>
<td>10</td>
<td>for every ready socket <code>sock_j</code> ∈ <code>activeSockets_out</code></td>
</tr>
<tr>
<td>11</td>
<td>send message <code>m</code> from <code>sock_j</code></td>
</tr>
<tr>
<td>12</td>
<td>update connection statistics</td>
</tr>
</tbody>
</table>

All incoming and outgoing network traffic are monitored by a single `select()` call with a specific timeout value. The `select()` call releases when one or more sockets from the active list become ready for I/O operations, or when the prescribed timeout expires.
3.3. ARCHITECTURE AND DESIGN

The network thread then proceeds to process these sockets. All TCP I/O operations are non-blocking, and the send or receive operation of a message in a queue does not necessarily finish in one iteration of the network loop.

3.3.2 The Engine

Fig. 3.3 shows that the engine consists of four parts: (1) the message handler for processing incoming messages; (2) the event timer for scheduling timed or periodic events; (3) the logger for facilitating a thread-safe logging system; and (4) the API for algorithm implementation. The engine’s main responsibility is retrieving messages from the incoming queues of the network, and routing them to the appropriate message handlers. Some messages are handled by the engine itself, and the remainders are destined to the algorithm. In each iteration of the engine thread, as in Table 3.3, at most one incoming message from each incoming queue is processed (completely or partially), and subsequently removed after being completely processed.

Table 3.3: The main body of the engine thread

```
1   while peer is alive
2       sleep until at least one incoming queue becomes non-empty
        or the next event has expired
3       for every non-empty incoming queue $q_i$
4           get the head-of-the-line message $m$ from $q_i$
5           if $m$ is destined to the engine
6               self.process($m$)
7           else
8               algorithm.process($m$)
9       if $m$ is fully processed
10          dequeue message $m$ from queue $q_i$
11       for every event $e_j$ in the event list that has expired
12          call $e_j$’s registered callback function
```
The message handler, \texttt{process()}, processes the messages and returns one of the two statuses: \textit{consumed} and \textit{hold}. The \textit{consumed} status implies that the message has been processed normally and can be discarded by the engine after it has been processed by the engine thread. Upon receiving such a status, the engine dequeues the message from the corresponding incoming queue, if the message has been consumed and properly enqueued to designated outgoing queue(s) specified by the algorithm. The \textit{hold} status implies that the message has been examined, but it should be held for further processing. Upon receiving this status, the engine keeps the message in the incoming queue, so that the algorithm can process this message again in the next iteration.

The engine provides supports for timed and periodic events by maintaining an event list. Each event in the list is associated with a timeout value and a callback function. The events are sorted according to their timeout values. To add an event, the algorithm or the engine simply inserts the event to the list. The engine periodically examines the
list and triggers events. When an event is due, the engine invokes the callback function and removes the event from the list. In the case of a periodical event, the event is inserted back to the list according to its next timeout after executing the callback function.

### 3.3.3 The Algorithm

The algorithm in the engine thread is where new P2P streaming protocols are to be developed. Crystal supports a wide range of multimedia streaming algorithms, from basic client-server setup to any sophisticated ones. To minimize development time, the algorithm includes a collection of basic elements for multimedia streaming, including playback buffers and simple protocols. The collection grows as new protocols are designed and implemented by system designers. The algorithm is usually implemented as an instance of an application-specific C++ class, within the engine thread. The application-specific class is derived from the `iAlgorithm` class, which defines the Crystal API between the engine and the algorithm. We will present the API in Sec. 3.4.6. There is a two-way communication between the engine and the algorithm. On one hand, the engine routes messages from all upstream peers to the algorithm for processing. On the other hand, the algorithm sends messages, via the engine and the network, to the appropriate downstream peers.

When the incoming queues are filled with unprocessed messages and the outgoing queues are empty, the network thread is idling. Otherwise, the engine should be busy switching messages at the rate of receiving. When an algorithm involves lengthy operations, it can block the engine thread from processing incoming queues and timed events. For this reason, we allow an algorithm to launch its own private thread(s). Similar to any other multi-threaded application with potential access to a set of shared data by
multiple threads, each algorithm thread has to use proper synchronization construct to prevent potential race conditions. Crystal provides high-level synchronization constructs in its library, which can be easily employed. While allowing algorithm-specific threads adds to the complexity of the algorithm development, it gives developers the flexibility in designing efficient and sophisticated algorithms. Obviously, algorithm threads should be used with discretion.

3.4 Design Objectives Revisited

There are no limitations in the Crystal design that preclude emulating more than one peer on each server; in fact, such a way of running emulated peers is encouraged. Emulated peers do not need to periodically contact a central server for logistics or authentication. All logs are written to local file systems, and are then collected and analyzed by scripts after each experiment. By minimizing the footprint of each emulated peer, we are able to run hundreds of peers on one server. Let us now revisit the original design objectives, and note how Crystal fulfills these requirements.

3.4.1 Emulating Bandwidth and Delay

In a realistic P2P network, peers are connected to the Internet using home (e.g., cable/DSL) broadband connections. Crystal supports the emulation of peer upload and download bandwidth limits, peer total bandwidth limits, as well as per-connection bandwidth limits for TCP connections. More than one limits can be enforced concurrently on a peer. In implementing such bandwidth limits, our objective is to minimize the demand for CPU cycles. The engine incurs lighter CPU load as the bandwidth limit decreases,
allowing more bandwidth-emulated peers on each server.

Figure 3.4: Bandwidth emulation in Crystal

The basic idea, as shown in Fig. 3.4, is to use a timer in the network thread to limit peer upload and download bandwidth, and an individual timer to enforce per-connection bandwidth limits. The timers are implemented through the `select()` call that is already used for monitoring active sockets in the network. The timeout value of the `select()` call is tuned dynamically according to the network traffic and available bandwidth, and is critical to the correctness and scalability of the bandwidth/delay emulation. For per-connection upload (download) bandwidth emulation, the message queue is delayed for \( n/b \) seconds after sending (receiving) a message, where \( n \) is the number of bytes in the message, and \( b \) is the bandwidth limit of this connection. The descriptor associated with this connection is not added to `fd_set` until \( n/b \) seconds later. For peer upload (download) bandwidth emulation, the timeout of each `select()` call is calculated as \( N/B \), where \( N \) is the number of bytes sent (received) since the last `select()` call, and \( B \) is the bandwidth limit of this peer. In other words, we dynamically determine the set of “ready” TCP queues, which are allowed to transmit at the current time. By dynamically changing the membership of `fd_set` based on predefined bandwidth limits and the current time, multiple bandwidth limits can be enforced simultaneously with little overhead.
3.4. DESIGN OBJECTIVES REVISITED

Table 3.4: Emulation of bandwidth limits using select()

<table>
<thead>
<tr>
<th>Each TCP connection, if bandwidth is limited, has a \texttt{timeForNextMsg} parameter indicating the earliest time that the peer should process the next message from this connection. It is initialized to the current time for any new connections, \textit{i.e.}, \texttt{timeForNextMsg = virtualTime = getCurrentTime();}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 while peer is alive</td>
</tr>
<tr>
<td>2 \textbf{sleep} until at least one connection becomes eligible for sending or receiving</td>
</tr>
<tr>
<td>3 add TCP listening socket and UDP sockets to \texttt{activeSockets\textsubscript{in}}</td>
</tr>
<tr>
<td>4 \textbf{if} \texttt{virtualTime} \textless= \texttt{getCurrentTime()}</td>
</tr>
<tr>
<td>5 \textbf{for} every socket \texttt{sock\textsubscript{i}} of an incoming connection \texttt{i}</td>
</tr>
<tr>
<td>6 \textbf{with} \texttt{timeForNextMsg} \textless= \texttt{getCurrentTime()}</td>
</tr>
<tr>
<td>7 add \texttt{sock\textsubscript{i}} to \texttt{activeSockets\textsubscript{in}}</td>
</tr>
<tr>
<td>8 \textbf{for} every socket \texttt{sock\textsubscript{j}} of an outgoing connection \texttt{j}</td>
</tr>
<tr>
<td>9 \textbf{with} \texttt{timeForNextMsg} \textless= \texttt{getCurrentTime()}</td>
</tr>
<tr>
<td>10 add \texttt{sock\textsubscript{j}} to \texttt{activeSockets\textsubscript{out}}</td>
</tr>
<tr>
<td>11 \textbf{select} (\texttt{activeSockets\textsubscript{in}}, \texttt{activeSockets\textsubscript{out}}, 10usec)</td>
</tr>
<tr>
<td>12 \textbf{for} every ready socket \texttt{sock\textsubscript{i}} \texttt{\in} \texttt{activeSockets\textsubscript{in}}</td>
</tr>
<tr>
<td>13 receive message \texttt{m} from \texttt{sock\textsubscript{i}}</td>
</tr>
<tr>
<td>14 update \texttt{timeForNextMsg} for this connection to</td>
</tr>
<tr>
<td>15 \texttt{timeForNextMsg} = \texttt{timeForNextMsg} + \texttt{n/ b}, where \texttt{n} is the size of the message \texttt{m} and \texttt{b} is the bandwidth limit for this connection.</td>
</tr>
<tr>
<td>16 \textbf{for} every ready socket \texttt{sock\textsubscript{j}} \texttt{\in} \texttt{activeSockets\textsubscript{out}}</td>
</tr>
<tr>
<td>17 send message \texttt{m} from \texttt{sock\textsubscript{j}}</td>
</tr>
<tr>
<td>18 update \texttt{timeForNextMsg} for this connection to</td>
</tr>
<tr>
<td>19 \texttt{timeForNextMsg} = \texttt{timeForNextMsg} + \texttt{n/ b}, where \texttt{n} is the size of the message \texttt{m} and \texttt{b} is the bandwidth limit for this connection.</td>
</tr>
<tr>
<td>20 update connection statistics</td>
</tr>
<tr>
<td>21 update \texttt{virtualTime} for this peer to \texttt{virtualTime} = \texttt{virtualTime} + \texttt{N/ B}, where \texttt{N} is the total number of bytes processed, and \texttt{B} is the peer outgoing bandwidth limit</td>
</tr>
<tr>
<td>22 \texttt{timeout} = \texttt{min(virtualTime, {timeForNextMsg})}, where \texttt{{timeForNextMsg}} consists of the timeout values from all connections.</td>
</tr>
<tr>
<td>23 \textbf{select} (0, 0, \texttt{timeout})</td>
</tr>
</tbody>
</table>
In more details, Table 3.4, modified based on Table 3.2, shows a skeleton of the bandwidth emulation algorithm we have implemented. Our experiences with bandwidth emulation in Crystal are very positive: CPU load decreases steadily as the bandwidth limit becomes lower, thus allowing more peers to be hosted by one server. Without any traffic, there is no load on CPU. Without any bandwidth limit, Crystal achieves the same TCP throughput as any other file transfer applications in the cluster (e.g., FTP).

To implement delays on a connection, we timestamp the creation time of each message, and add an additional step after line 15 in Table 3.4. In this step, \( \text{timeForNextMsg} \) is updated to \( \text{timeForNextMsg} + (d - d') \), where \( d \) is the link delay and \( d' \) is the difference between current time and the creation time of the message. The basic idea here is to have each message delayed for \( d \) microseconds, starting from its creation time.

### 3.4.2 Real Streaming Server and Real Traffic

Crystal provides a testing data source to facilitate the emulation of the streaming server in a live session. The pseudo-source is able to produce data from a regular file, the standard input, or a stream of randomly generated bytes. As shown in Fig. 3.5, the data are stored in a dedicated message queue that is treated by the engine in the same way as other message queues. The source distributes the content via TCP or UDP connections, according to specified streaming rate. This design leads to minimal changes in the implementation of the streaming server, i.e., the streaming server shares the same architecture as a regular peer, with a special data generator and a source message queue. Though a testing data source is provided, algorithm designers may still choose to implement their own data sources.
3.4. DESIGN OBJECTIVES REVISITED

3.4.3 Emulating Peer Dynamics

In a realistic P2P network, peers may present a significant level of dynamics by joining and departing at any time. To emulate peer dynamics, we have implemented a log-driven facility. Upon the startup of each experiment, each peer parses the events file for all events that are associated with its IP address, and registers them in the event timer managed by the engine. The use of such an event-driven facility relaxes the necessity to contact centralized servers, which may lead to a considerable amount of TCP traffic that may affect the precision of experiments. Table 3.5 shows a sketch of the event scheduler. Line 2 is very important to prevent busy looping, in cases where no events are left in the queue.

The events file specifies the time that the event should occur in an experiment and the type of the event. Each experiment is enclosed by the experiment name and the keyword end, as shown in Table 3.6. More experiments can be specified in the events file, by using different names for each experiment. Typical events include the birth and die times of a peer in the network, as well as the join and leave times of a peer in a session. There are two events that are dedicated to the streaming server: deploy and terminate. The deploy
Table 3.5: The event scheduler

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>while</strong> peer is alive</td>
</tr>
<tr>
<td>2</td>
<td><strong>if</strong> the event queue is empty</td>
</tr>
<tr>
<td>3</td>
<td>Let <em>delay</em> be a small time unit that is greater than zero.</td>
</tr>
<tr>
<td>4</td>
<td><strong>else</strong></td>
</tr>
<tr>
<td>5</td>
<td>Let <em>delay</em> be the difference between the current time and the next event in the queue.</td>
</tr>
<tr>
<td>6</td>
<td><strong>nanosleep</strong>(delay).</td>
</tr>
<tr>
<td>7</td>
<td>Trigger all events that are due before or at current time, by calling the appropriate event handlers.</td>
</tr>
</tbody>
</table>

Table 3.5: The event scheduler

Assuming all events has been compiled into the event queue, sorted by the event time in ascending order.

The event specifies the streaming rate in bytes/second and the time that the server should start to produce streaming content, whereas the terminate event specifies the time when the source stops producing data. We also allow optional parameters associated with each event. For example, to specify the arrival of a peer in a session, the event in the log has parameters such as the session identifier, streaming server and streaming rate, as well as the arrival time. Table 3.7 summarizes these six event types and their format in the events file.

Table 3.6 is a simple events file. The first parameter of each event is the relative time since the start time of the experiment, at which this event should be scheduled to occur. The event type is always followed by the IP and port number of the peer that this event is associated with. This events file contains all six types of events discussed so far, each with its own list of parameters. The bandwidth limits and streaming rate are specified in bytes per second. We reserved $-1$ to denote unlimited bandwidth or bit rate. It is also important to note that peers are not limited to these six types of events. The event queue accepts any events introduced by the algorithm.
3.4. DESIGN OBJECTIVES REVISITED

Table 3.6: An example of the events file

```
simpletest
0,birth,10.2.1.1,9000,1435328,-1,-1,
2,join,10.2.1.1,9000,1,65536,
3,deploy,10.2.1.1,9000,1,65536,
4.38,birth,10.2.1.2,9000,78499,-1,-1,
4.38,join,10.2.1.2,9000,1,65536,
5.19,birth,10.2.1.3,9000,54494,-1,-1,
5.19,join,10.2.1.3,9000,1,65536,
121.28,leave,10.2.1.2,9000,1,
122.18,leave,10.2.1.3,9000,1,
123.28,die,10.2.1.2,9000,
124,leave,10.2.1.1,9000,1,
124,terminate,10.2.1.1,9000,1,
124.18,die,10.2.1.3,9000,
126,die,10.2.1.1,9000,
end
```

Table 3.7: The event types

<table>
<thead>
<tr>
<th>types</th>
<th>format</th>
</tr>
</thead>
<tbody>
<tr>
<td>birth</td>
<td>&lt;time&gt;,birth,&lt;IP&gt;,&lt;port&gt;,&lt;total peer bandwidth&gt;,&lt;total uplink bandwidth&gt;,&lt;total downlink bandwidth&gt;,</td>
</tr>
<tr>
<td>die</td>
<td>&lt;time&gt;,die,&lt;IP&gt;,&lt;port&gt;,</td>
</tr>
<tr>
<td>join</td>
<td>&lt;time&gt;,join,&lt;IP&gt;,&lt;port&gt;,&lt;session ID&gt;,&lt;streaming rate&gt;,</td>
</tr>
<tr>
<td>leave</td>
<td>&lt;time&gt;,leave,&lt;IP&gt;,&lt;port&gt;,&lt;session ID&gt;,</td>
</tr>
<tr>
<td>deploy</td>
<td>&lt;time&gt;,deploy,&lt;IP&gt;,&lt;port&gt;,&lt;session ID&gt;,&lt;streaming rate&gt;,</td>
</tr>
<tr>
<td>terminate</td>
<td>&lt;time&gt;,terminate,&lt;IP&gt;,&lt;port&gt;,&lt;session ID&gt;,</td>
</tr>
</tbody>
</table>
3.4. DESIGN OBJECTIVES REVISITED

The events in the log can be generated either randomly or following a certain distribution. For instance, according to work by Stutzbach et al. [79], we could generate the birth events following the Poisson distribution with a PDF $f(k; \lambda) = \frac{e^{-\lambda} \lambda^k}{k!}$, where $1/\lambda$ is the inter-arrival time. In addition, the join events may follow a Weibull distribution with PDF $f(x; k, \lambda) = \frac{k}{\lambda} (\frac{x}{\lambda})^{k-1} e^{-(x/\lambda)^k}$, where $k$ is the shape parameter and $\lambda$ is the scale parameter. The lifetime of a peer (the difference between join time and leave time) may follow either a Weibull distribution or a Log-Normal distribution with a PDF $f(x; \mu, \sigma) = \frac{1}{x \sigma \sqrt{2\pi}} e^{-\frac{(lnx-\mu)^2}{2\sigma^2}}$, where $\mu$ and $\sigma$ are the mean and standard deviation, respectively. Naturally, these distributions can be modified by algorithm developers at any time.

3.4.4 Peer Bootstrapping

Upon the creation of a peer, the peer needs to start with a set of live peers in the network. We have implemented a centralized tracker to provide this first-level bootstrap support by providing newly joined peers a random subset of existing peers. The tracker can either actively probe each peer for aliveness or passively wait for reports from each peer, depending on the configuration.

The topology file provides an alternative solution for bootstrapping peers. For each peer, this file specifies a small number of peers that are alive at the birth time of this peer. The use of this file relaxes the necessity to contact the tracker, which may lead to a considerable amount of TCP traffic. The topology file consists of two sections: links and bootstrap. The links section is simply an edge list of a complete directed graph. For each edge, there are two parameters supplied: link bandwidth in bytes per second and link delay in milliseconds. The bootstrap section specifies the bootstrap information for
3.4. DESIGN OBJECTIVES REVISITED

each peer in the network. The first IP address and port number in each line represent a peer, and the rest of the line consists of the bootstrap information for this peer.

Table 3.8: An example of the topology file

| simpletest |
| links      |
| 10.2.1.1:9000,10.2.1.2:9000,-1,7, |
| 10.2.1.2:9000,10.2.1.1:9000,-1,76, |
| 10.2.1.2:9000,10.2.1.3:9000,-1,189, |
| 10.2.1.3:9000,10.2.1.2:9000,-1,134, |
| 10.2.1.1:9000,10.2.1.3:9000,-1,12, |
| 10.2.1.3:9000,10.2.1.1:9000,-1,128, |
| bootstrap  |
| 10.2.1.2:9000,10.2.1.1:9000, |
| 10.2.1.3:9000,10.2.1.1:9000,10.2.1.2:9000, |
| end        |

3.4.5 Playback Buffer

For each streaming session, a peer maintains a playback buffer in which segments (small units of streaming content) are ordered according to their playback time. Although a streaming session can be indefinitely long, there is only limited memory space for the buffer. For memory efficiency, the playback buffer is internally implemented as a circular queue. As illustrated in Fig. 3.6, the source usually has a larger buffer filled with segments that are ready for distribution, and each peer has a buffer with a small fixed size and two pointers: front and rear indicating the first segment and the next available slot in the buffer, respectively. The rear pointer moves forward to make room for new segments, whereas the front pointer moves forward when a segment is removed after being played. The buffer is full when the rear pointer meets the front pointer. The playback buffer
registers a periodical event with the engine to emulate segment playback. The period of this event is determined by the size of each segment, \(i.e.,\) the number of seconds of the playback represented by a segment. For example, if each segment represents 1 second of the playback, \(t_2 - t_1 = t_3 - t_2 = 1\) second in Fig. 3.6.

![Figure 3.6: An example of the playback buffer](image-url)
3.4.6 Streaming Algorithm Development

One of the design objectives of Crystal is to aid rapid development of new streaming systems. As such, Crystal provides an abstraction for most mundane but necessary programming tasks. Since the engine is developed in the C++ programming language, algorithms should be developed in C++ as well. Crystal is designed so that algorithm development does not have to be concerned with thread safety, in that all algorithm specific code is executed in the engine thread. Such a design allows the designers to focus on the algorithm without worrying about the internal data structure and thread safety issues in the engine. However, it is the algorithm developer’s responsibility to handle race conditions and synchronization issues in the threads created outside Crystal.

A new algorithm should be developed as a derived class of iAlgorithm, a base skeleton class implemented by Crystal, defining the Crystal API. It includes a few member functions that the new algorithm must implement. They are listed and explained in Table 3.9. The nature of such an algorithm is reactive, i.e., it reacts to messages received by the engine from its upstream peers, by implementing process() and processData(), which is called by the engine. The algorithm calls send() to send one or more messages to downstream peers. Examples of implementing new algorithms in Crystal will be given throughout this thesis.

3.5 Performance Evaluation

For an emulation framework, nothing is more important than its scalability and usability. In this section, we present the benchmark measurements of Crystal in terms of CPU usage and maximum achievable throughput to show its scalability. We studied three
### Table 3.9: The API of Crystal

<table>
<thead>
<tr>
<th>Member Function of iAlgorithm</th>
<th>Explanations</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bootstrap()</code></td>
<td>This function needs to be implemented to bootstrap a new peer, and initialize its algorithm-specific parameters.</td>
</tr>
<tr>
<td><code>joinSession()</code></td>
<td>The engine calls these functions when a peer joins a session or leaves a session. When a session is removed by the source, <code>leaveSession()</code> is also called by the engine. A session is uniquely identified by its session identifier (a short type integer).</td>
</tr>
<tr>
<td><code>leaveSession()</code></td>
<td></td>
</tr>
<tr>
<td><code>process()</code></td>
<td>The engine calls this function when a control message is to be processed by the algorithm.</td>
</tr>
<tr>
<td><code>processData()</code></td>
<td>The engine calls this function when a data message is to be processed by the algorithm.</td>
</tr>
<tr>
<td><code>peerDeparted()</code></td>
<td>The engine calls this function when an existing upstream or downstream peer has departed. The engine encapsulates all exception handling mechanisms, and these functions are called when exceptions occur.</td>
</tr>
<tr>
<td><code>removePeer()</code></td>
<td>The algorithm calls this function to explicitly tear down the existing TCP connection to an upstream or downstream peer.</td>
</tr>
<tr>
<td><code>send()</code></td>
<td>The algorithm calls this function to send a message to a peer. A message can be sent via either TCP or UDP, specified by one of the parameters.</td>
</tr>
</tbody>
</table>

Performance benchmarks: (1) Accumulative streaming rate (MB/sec), the sum of the streaming rates achieved by all peers. These measurements indicated the system-wide achievable throughput, which is also a good indication of the TCP bottleneck. (2) Per-peer streaming rate (MB/sec), the average streaming rate from all peers in the network. This metric is a good reference for choosing the right number of peers to emulate a network of peers with given bandwidth limits. (3) CPU usage, the percentage of utilized CPU cycles, which signifies the scalability of Crystal in terms of CPU load. We resorted all experiments to our cluster of dual-CPU servers (Pentium 4 Xeon 3.6 GHz and AMD Opteron 2.4 GHz), interconnected by Gigabit Ethernet.
To evaluate the scalability of Crystal, we implemented a simple P2P relaying protocol. In this protocol, a peer simply relays incoming data messages to all neighboring peers. The skeleton of the relaying protocol is given in Table 3.10. All functions are invoked by the engine as the events are triggered or messages are received. The algorithm only needs to call one function of the engine: the \texttt{send()} function. The message handler, \texttt{process()}, switches on different message types. The \texttt{removePeer()} is called by the engine when a connection failure is detected in the network layer. The \texttt{joinSession()} and \texttt{leaveSession()} functions are callback functions for the \texttt{join} and \texttt{leave} events. The other events in the events file are internally handled by the engine. We will further demonstrate the usability and effectiveness of Crystal in Chapter 5 and Chapter 6, in which we present our implementations of a conventional and a new P2P streaming systems.

Table 3.10: The skeleton of the relaying protocol using Crystal

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{bootstrap(Msg * m)}</td>
<td>Contact the tracker or parse the \texttt{topology} file for an initial set of peers</td>
</tr>
<tr>
<td>\texttt{process(Msg * m)}</td>
<td>switch \texttt{m-&gt;type()})&lt;br&gt;case \texttt{join}:&lt;br&gt; Add the peer who sent this message to the downstream peer list&lt;br&gt;case \texttt{data}:&lt;br&gt; for each peer \texttt{p} in the downstream peer list&lt;br&gt; send\texttt{(m, p)}&lt;br&gt;case \texttt{default}:&lt;br&gt; Unknown message type.&lt;br&gt; return consumed</td>
</tr>
<tr>
<td>\texttt{removePeer(Msg * m)}</td>
<td>Remove a peer from the downstream peer list</td>
</tr>
<tr>
<td>\texttt{joinSession(Msg * m)}</td>
<td>Send a \texttt{join} message to the initial neighboring peers</td>
</tr>
<tr>
<td>\texttt{leaveSession(Msg * m)}</td>
<td>Leave a session by disconnecting all downstream and upstream peers.</td>
</tr>
</tbody>
</table>
The streaming server and the tracker are hosted on a separate machine from the rest of the peers, to ensure no CPU interference. We enforced no bit rate on the source, \textit{i.e.}, the source generates data as fast as the CPU and TCP connections allow. The tracker is configured to bootstraps peers in a way so that the peers form multiple chains of up to 10 peers that are stemmed from the streaming source. A sample setup is shown in Fig. 3.7.

![Figure 3.7: A sample experimental setup with chains of peers served by a server](image)

We examined the maximum achievable streaming rate as the number of emulated peers increases on a single cluster server. In this experiment, we enforced no bandwidth limits or link delays so that the engine switches messages as fast as possible, \textit{i.e.}, both CPUs on each server are 100\% saturated. The message payload is set to 1 KB. As shown in Fig. 3.8, the accumulative streaming rate gradually decreases as the number of peers grows. Despite the decrease, Crystal is able to achieve more than 12 MB/sec streaming rate before reaching the TCP bottleneck. Fig. 3.9 further shows that per-peer streaming rate quickly converges to 50 KB/sec, based on which we predicted that a single server can support at most 200 - 300 peers at this rate.
3.5. PERFORMANCE EVALUATION

Figure 3.8: The accumulative streaming rate from all peers on a single server

Figure 3.9: The average streaming rate on each peer from a single server, with standard deviation
The above observation turned our attention to the CPU performance when tuning the per-link upload bandwidth limit and the number of peers. In this experiment, we first fixed the number of peers to 10 and increased the per-link upload bandwidth limit from 50 KB/sec to 1.7 MB/sec. As the bandwidth limit grows, the engine needs to switch more messages every second, i.e., the CPU spends more time on message switching. As illustrated in Fig. 3.10, the CPU load decreases steadily as the bandwidth limit lowers, thus allowing more peers to be hosted on one server. Without traffic, there is no load on CPU, and without any bandwidth limits, Crystal achieves the same TCP throughput as any other application. The CPU utilization grows slowly when the bandwidth limit is less than 600 KB/sec, and then linearly increases as the bandwidth limit grows. In real-world P2P networks, most peers have DSL-like connections with less than 300 KB/sec upload and download bandwidth. The results shown in Fig. 3.10 indicate that Crystal can easily support a few dozens of DSL-like peers on a single server with a light CPU footprint.

We then fixed the bandwidth limit to 50 KB/sec, a typical DSL-like peer upload bandwidth, and increased the number of peers from 10 to 250. As shown in Fig. 3.11, the CPU load decreases steadily as the number peers decreases. We observed a noticeable throughput drop on the peers that were far away from the streaming server when the network consisted of more than 200 peers. This means that the message switch in the engine could not keep up with the arrival rate of the incoming messages, which confirms the observation from Fig. 3.9 that a single server can support at most 200 peers under this setting.

To further show the scalability of Crystal on multiple servers, we scaled the network size across 10 different cluster servers. Again, we enforced no bandwidth limits or link
3.5. PERFORMANCE EVALUATION

Figure 3.10: The CPU usage as the per-link upload bandwidth increases on a single server. The number of peers is fixed to 10.

Figure 3.11: The CPU usage as the number of peer increases on a single server. Per link bandwidth is fixed to 50 KB/sec.
3.5. PERFORMANCE EVALUATION

delays, and the message size is still set to 1 KB. The average per-peer streaming rate in Fig. 3.12 follows the same pattern as in Fig. 3.9. However, for the same network size, Crystal offers higher streaming rate on 10 servers than it does on a single server. Moreover, the streaming rate decreases at a slower rate in Fig. 3.12. Hence, Crystal scales better when it is deployed on multiple servers.

![Average streaming rate on each peer from 10 servers, with standard deviation](image)

**Figure 3.12:** The average streaming rate on each peer from 10 servers, with standard deviation

The number of messages to be switched by the engine is determined by not only the bandwidth, but also the message size. We conducted another set of experiments to determine the maximum achievable streaming rate as we varied the size of messages from 1 KB to 40 KB. The same experiment is repeated in three different networks consisting of 10, 100, and 200 peers. Fig. 3.13 shows that the accumulative streaming rate grows almost linearly as the message size increases. Thus, to emulate a network with higher bandwidth limits, one can either increase the number of servers or increase the message size.
3.6 Summary

Crystal is designed to address the challenges of prototyping, testing, and evaluating new P2P streaming systems, in a more realistic scenario than simulations, yet a more controllable and scalable environment than real-world experiments. Our implementation crystallizes about two years of research and development, with about ten thousand lines of code in C++, and is now ready for release in an open-source form to the research community. It includes the engine and the network at the core to handle networking and message switching functions, a clearly defined and simple API between the engine and the algorithm. Throughout this thesis, we will use Crystal to develop P2P streaming systems and to conduct empirical studies. With evidence of more than 200 emulated peers on one cluster server, Crystal is indeed a turn-key solution to prototype and evaluate new P2P streaming protocols in large-scale P2P sessions, and in record time.
Chapter 4

High-Performance Network Coding Library

In addition to the emulation framework, we also need network coding supports to assess its practicality, and eventually develop a new streaming protocol that takes the advantages of network coding. In this chapter, we present the details of our high-performance network coding library, with experimental benchmark results.

4.1 Design and Implementation

Our network coding library adopts the concept of randomized network coding [27, 56, 28, 53]. The original content on the source is divided into $n$ blocks $[b_1, b_2, \ldots, b_n]$, each block $b_i$ has a fixed number of bytes, $k$, referred to as the block size. At the time of encoding for downstream peer $p$, a peer (including the source) randomly and independently chooses a set of coding coefficients $[c_{p1}^p, c_{p2}^p, \ldots, c_{pm}^p]$ in GF(256) for the downstream peer $p$, where $m \leq n$. It then randomly chooses $m$ blocks — $[b_{p1}^p, b_{p2}^p, \ldots, b_{pm}^p]$ — out of all the blocks it
has received so far or all the original blocks if it is a source of the session. The ratio \( m/n \) is referred to as density, and a low ratio leads to sparse decoding matrices. One encoded block \( x \) of \( k \) bytes is produced by:

\[
x = \sum_{i=1}^{m} c_i^p \cdot b_i^p
\]  \hspace{1cm} (4.1)

In the example shown in Fig. 4.1, there are five original blocks to be encoded, and each block has exactly 1 byte of data. The encoding process first converts the data to their corresponding ASCII values, in the range of \([0, 255]\). Each byte is then multiplied with the randomly generated coding coefficients in \( \text{GF}(256) \). The sum of these products yields one encoded block.

The coding coefficients used to encode original blocks to \( x \) are typically embedded in the header of the coded block for transmission. We thus need a total of \( n \) coefficients, leading to an overhead of \( n \) bytes per coded block. These \( n \) coding coefficients to be embedded can easily be computed by multiplying \([c_1^p, \ldots, c_m^p]\) with the \( m \times n \) matrix of coding coefficients of the incoming blocks \([b_1^p, b_2^p, \ldots, b_m^p]\). We note that an overhead of \( n \) bytes may still be considered substantial when \( n \) is large (e.g., 128) and the block size \( k \) is small (e.g., 1 KB). If \( m/n = 1 \) and the seed has all \( n \) blocks when producing a coded block, we just need to embed the random seed used to produce the series of random coefficients with a known pseudo-random number generator. This effectively reduces the overhead to just 4 bytes, regardless the values of \( n \) and \( k \).

As the session proceeds, a peer accumulates coded blocks into its playback buffer, and encodes new coded blocks to serve its downstream peers. In order to reduce the delay introduced by waiting for new coded blocks, the peer produces a new coded block upon receiving \( \alpha \cdot n \) coded blocks \((0 < \alpha \leq 1)\), in which the tunable parameter \( \alpha \) is referred to
Figure 4.1: An application of network coding in P2P mesh networks
as the *aggressiveness*. A smaller \( \alpha \) implies that downstream peers can be served sooner. In other words, a peer is more “aggressive” in becoming a seed.

As soon as a peer has received a total of \( n \) coded blocks \( \mathbf{x} = [x_1, x_2, \ldots, x_n] \), it starts the decoding process. To decode, it first forms a \( n \times n \) matrix \( \mathbf{A} \), using the \( n \) coding coefficients embedded in each of the \( n \) coded blocks it has received. If \( m/n = 1 \), the random coefficients can be reproduced by using the seed embedded in each block. Each row in \( \mathbf{A} \) corresponds to \( n \) coded coefficients of one coded block. It then recovers the original blocks \( \mathbf{b} = [b_1, b_2, \ldots, b_n] \) by:

\[
\mathbf{b} = \mathbf{A}^{-1} \mathbf{x}^T \tag{4.2}
\]

In this equation, it first needs to compute the inverse of \( \mathbf{A} \), using Gaussian elimination. It then needs to multiply \( \mathbf{A}^{-1} \) and \( \mathbf{x}^T \), which takes \( n^2 \cdot k \) multiplications of two bytes in \( \text{GF}(256) \). Since addition in \( \text{GF}(256) \) is simply an XOR operation, it is important to optimize the implementation of multiplication in \( \text{GF}(256) \), in order to optimize the performance of coding. The code for this operation is listed in Table 4.1, which only requires three memory reads and one addition.

**Table 4.1: Multiplication of two bytes on \( \text{GF}(256) \) in Lava**

```c
int gf_single_multiply(int x, int y)
{
    if (x == 0 || y == 0) return 0;
    return J_TO_B[B_TO_J[x] + B_TO_J[y]];
}
```
4.2 Performance of the Network Coding Library

To guarantee the most optimized binary, our network coding library is implemented in C++, and compiled with full optimization (-O3). Though all our experiments are performed in our server cluster, it can be readily compiled on other UNIX variants as well. This section presents the benchmark results of the network coding library.

4.2.1 Coding Performance

The first question that one would naturally ask is: “What is the performance of randomized network coding?” To answer this question, we established a single connection between one sender and one receiver, each hosted by a dedicated CPU (Pentium IV Xeon 3.6GHz). Fig. 4.2(a) and in Fig. 4.2(b) show the rate of the encoding process and the rate of the decoding process in bytes per second, respectively. In these figures, the $x$ axis shows the number of blocks used, and bars with different gray scale represent different block sizes. We varied the block size from 64 bytes to 1 MB, and varied the number of blocks from 10 to 1500. There are a number of important observations we derived from these results.

First, the absolute values of both encoding and decoding rates are better than our expectation, especially when the number of blocks is smaller than 100. As we have shown, when there are only 10 blocks, the coding rate exceeds 10 MB/sec! Even with 1000 blocks, we still observed a coding rate of more than 40 KB/sec, which is about the same bit rate as a typical multimedia stream.

Second, the coding rate rapidly decreases as the number of blocks increases, but it does not vary significantly as the block size varies. This observation justifies the use of a small number of blocks (such as 100). Indeed, the P2P experiments in Avalanche
4.2. PERFORMANCE OF THE NETWORK CODING LIBRARY

![Diagram](image)

(a) Encoding rate

(b) Decoding rate

Figure 4.2: Coding rate when tuning the block size and number of blocks
divide 4.3 GB files into groups (also referred to as generations [56]), and uses 80 blocks per group. Each block in Avalanche has approximately 2.3 MB\(^1\). Our results reflect that, with 80 blocks and 2.3 MB per block, the coding rate is between 1 and 2 MB/sec. Since most peers are on slower connections, the coding thread does not have the data readily available to code at its full bandwidth. This explains why CPU load is around 20% in Avalanche experiments.

Third, as the block size varies, there is a “sweet spot” for maximizing the coding rate. This optimal block size shifts upwards as the number of blocks increases, but is mostly around 2 KB – 32 KB. When the block is too small, only a few bytes is produced after each encoding or decoding operation. Intuitively, bigger blocks would offer higher coding rate. However, when the block size reaches more than 32 KB, the CPU is close to 100% saturated, leading to a decrease in coding rate. This justifies the use of small block sizes less than 32 KB. Indeed, even BitTorrent uses 16 KB as its block size.

Finally, in each configuration, the encoding and the corresponding decoding rate are practically the same. At first glance, this seems to be hardly surprising, as both of them have been shown in Sec. 4.1 to be a multiplication of a matrix and a vector, in GF(256). However, we noted that it involves matrix inversion using Gaussian elimination, which has a complexity of \(O(n^3)\). Shouldn’t decoding be much less efficient than encoding?

To obtain further insights, Fig. 4.3(a) shows the time to generate encoding matrices, encode, inverse a matrix and decode, normalized to microseconds per byte. Again, we observed that the coding performance degrades significantly as the number of blocks increase from 100 to 500, regardless of the block size. The time taken to perform matrix inversion is negligible as the block size \(k\) increases beyond 2 KB. Larger blocks are actually

\(^1\)The number of blocks per generation has not been shown in [29], but has instead been obtained via personal email communication.
4.2. PERFORMANCE OF THE NETWORK CODING LIBRARY

(a) Computational time per byte in various configurations of block size and number of blocks

(b) Time needed to transmit a 13.8 MB file in an one-to-one session, with the average transmitting time around one second, and matrix inversion around half a second

Figure 4.3: Performance of randomized network coding
more reasonable, as the header overhead to carry $n$ coding coefficients in a coded block would be too significant if the block size is smaller than 2 KB.

What if we wish to transmit a fixed-size file from the sender to the receiver? Fig. 4.3(b) shows the time it takes to transmitting a 13.8 MB file using network coding in a one-to-one session, over Gigabit Ethernet without bandwidth limits. It appears that it takes, on average, only a second to transmit the coded blocks and half a second to inverse the matrix $A$, both of which are negligible as compared to the encoding and decoding time. In Sec. 4.1, we noted that it takes $n^2 \cdot k$ multiplications in GF(256) to multiply $A^{-1}$ and $x^T$. If the number of bytes in the to-be-distributed data, i.e., $n \cdot k$, remains fixed, then the number of multiplications on GF(256) that a peer needs to perform (after computing $A^{-1}$) scales linearly with $n$. This precisely matches the results shown in Fig. 4.3(b).

### 4.2.2 Density and Aggressiveness

Now that we have an idea about coding performance, we deployed randomized network coding in more realistic network topologies, and compared its performance to naive broadcast. Upon receiving a block of data, each peer simply forwards it to all of its downstream neighbors. All incoming duplicated blocks will simply be discarded. We used naive broadcast as the baseline benchmark to evaluate the practicality of network coding, with the mentality that if it is challenging for network coding to compete with naive broadcast, it will have greater difficulties competing with a well-tuned protocol.

To experiment with more realistic topologies, we constructed bandwidth-emulated P2P topologies with around 80 peers, each with a dedicated CPU in the server cluster. We implemented a staged construction of such topologies, in which peers are added in

---

2The upper bound of 80 peers is due to the fact that we only possessed 40 dual-CPU servers in our cluster, and that we dedicated one CPU to each peer.
batches, and the tracker bootstraps new peers with at most 10 randomly selected peers as the upstream neighbors of the new peer. The topologies are designed to contain 30% Ethernet peers and 70% DSL peers. The uplink and downlink bandwidth limits of an Ethernet peer are set at 1 MB/sec, and a DSL peer has an uplink bandwidth limit of 50 – 70 KB/sec, and a downlink bandwidth limit of 200 – 300 KB/sec. These topologies are designed to emulate real-world P2P networks, and are referred to as mixed topologies hereafter. We were mostly concerned with the average transmission time, which is measured as the time period from the starting point of the session (before encoding starts in network coding sessions) to the point that a peer finished decoding (or recovering) all the original blocks. We used 100 blocks of size 32 KB each, as they have been shown to offer superb performance in our previous experiments.

To optimize the performance of a P2P session using network coding, we needed to tune its operational parameters. One such parameter, first introduced in Sec. 4.1, is density $m/n$, in which $m$ is the number of blocks a peer randomly selects to encode a new coded block for its downstream peers, and $n$ is the total number of blocks in the system. In Fig. 4.4(a), we varied the density from 100% to 2%, and measured the average transmission time in a mixed topology of 80 peers.

From Fig. 4.4(a), we observed that as density decreases, the average transmission time steadily decreases. Indeed, this observation meets our intuition, as when density decreases, each peer has fewer blocks to encode, thus leading to a higher coding rate, and smaller computational overhead. That said, when we tuned the density to lower than 6%, we noted that a substantial number of peers is not able to successfully decode after receiving 100 coded blocks. This is not surprising, as the coding matrix $A$ in Eqn. 4.2 would be too sparse to be full rank if the density is too low. It is indeed
4.2. PERFORMANCE OF THE NETWORK CODING LIBRARY

(a) Tuning density from 100% to 2%, as compared to naive broadcast, with aggressiveness as 100%

(b) Tuning aggressiveness from 100% to 20%, as compared to naive broadcast, with density as 6%

Figure 4.4: Average transmission time and the percentage of peers that have successfully decoded after receiving 100 coded blocks, without linear dependence checks
surprising, however, when we compared network coding with naive broadcast. Naive broadcast, being the worst possible non-coding protocol, actually enjoys a better average transmission time than coding at any density using network coding! We postpone our discussion until Fig. 4.7(a).

As our confidence is not yet shattered, we tried to find remedies to improve the performance of network coding. When reviewing our implementation of network coding, it appears that each peer only starts to produce and serve coded blocks to its downstream peers when it has buffered $n$ coded blocks itself. This is not efficient at all, since it can easily start to produce coded blocks much earlier. We then tuned the aggressiveness from 100\% down to 20\%, in the hope that it can lead to shorter transmission times, as each peer is now more eager to serve coded blocks. We conducted our experiment with the same configuration as the density one, but with density set at 6\%, the best we have observed. Results in Fig. 4.4(b) indeed show a trend of slowly decreasing transmission times, which may not be as promising as we expected. We were surprised to see that, even with aggressiveness set to as high as 96\%, there exist a few peers who fail to decode after receiving 100 blocks, which implies that they have received linearly dependent blocks. All previous work (e.g., [53]), however, theoretically maintained that the blocks should be linearly independent with high probability.

Going to one extreme, we ran a simple test with 10 peers and aggressiveness set to 1\%, i.e., a peer sends a new coded block to its downstream peer whenever it receives a new coded block from its upstream peers. Fig. 4.5 shows a simplified version of what we observed from the experiment traces. As the source, peer 1 sends two linearly independent blocks $a$ and $b$ to peers 2 and 3, respectively. Peer 2 immediately sends $a$ to its own downstream peers 4 and 5, scaled with two random coding coefficients, say $5a$ and $3a$.  

4.2. PERFORMANCE OF THE NETWORK CODING LIBRARY
Peer 3 also forwards $2b$ to peer 4 around the same time. If peer 4 receives $5a$ first, it will immediately send $k \cdot 5a$ to peer 5, which is linearly dependent with $3a$ that peer 5 has already received from peer 2. If peer 4 receives $2b$ first, it will first send $k_1 \cdot 2b$ to peer 5, and then another coded block $k_2 \cdot 5a + k_3 \cdot 2b$ upon receiving $5a$ from peer 2 (where $k_1$, $k_2$ and $k_3$ are randomly chosen coefficients). Naturally, the three coded blocks that peer 5 has just received are not linearly independent from one another.

The above example confirms the informal intuition that, when peers are too “eager” to serve other peers with new coded blocks, there are not enough linearly independent blocks to go around the system. Our solution to this challenge is to perform linear dependence checks at each peer: when a coded block is received at a peer, we attempted to run it through a dependence check routine. It is discarded if it is linearly dependent with any of the previously received blocks. As shown in Fig. 4.6(a), the computational overhead of such linear dependence checks increases as the number of buffered blocks increases, which implies that such linear dependence checks is only practical if each peer buffers a small number of blocks. As shown in Fig. 4.6(b) when comparing the transmission time of a 13.8 MB file in an one-to-one session, it is indeed the case that the transmission
4.2. PERFORMANCE OF THE NETWORK CODING LIBRARY

(a) Execution time of linear dependence checks

(b) Performing linear dependence checks when transmitting 13.8 MB file

Figure 4.6: Performance of linear dependence checks
time is practically the same with linear dependence checks turned on and off, when the number of blocks is smaller than 300. This observation is further verified when we re-ran our density tuning experiment with linear dependence checks activated. Fig. 4.7(a) shows that the average transmission times are unchanged when compared to Fig. 4.4(a), without the checks. This confirms, in a more realistic topology, that the overhead of linear dependence checks is quite negligible if the number of blocks is small.

We ran the aggressiveness tuning experiment again, this time with linear dependence checks. Fig. 4.7(b) shows that the average transmission times slowly decrease as the aggressiveness lowers. However, we noted that as aggressiveness reaches a certain critical point (around 66%), the number of blocks required to decode takes a sharp turn upwards, which leads to longer, rather than shorter, transmission times. This corresponds to the same point in Fig. 4.4(b) without checks, as the number of peers that fail to decode increases dramatically around 63% – 66%. Even with 66% aggressiveness, it can only offer a 25% advantage as compared to when aggressiveness is 100%. Nonetheless, the brighter side of the story is that, with aggressiveness at 66% and density at 6%, the performance of network coding with linear dependence checks approaches that of naive broadcast (but still inferior).

4.3 Progressive Decoding and Performance Tuning

Since the delivery of a segment in a P2P live streaming session is time sensitive, we wish to reduce the computational time required by the network coding library as much as possible. We note that a peer does not have to wait for all \( n \) linearly independent coded blocks before starting to decode a segment. In fact, it can start decoding as soon as the first coded block is received, and then progressively decode each of the new coded blocks,
4.3. PROGRESSIVE DECODING AND PERFORMANCE TUNING

Figure 4.7: Average transmission times and the number of blocks required for successful decoding, as a percentage of the number of original blocks, with linear dependence checks.
as they are received over the network. In this process, the decoding time overlaps with
the time required to receive the coded block, and thus hidden from the tally of overhead
caued by encoding and decoding processes. To realize such a progressive decoding
process, we employ Gauss-Jordan elimination, rather than Gaussian elimination, in the
decoding process. Gauss-Jordan elimination is a variant of Gaussian elimination, that
transforms a matrix to its reduced row-echelon form (RREF), in which each row contains
only zeros until the first nonzero element, which must be 1.

A peer starts to progressively decode the received content, as soon as it receives the
first coded block. As each new coded block is received, its coefficients are added to the
coding coefficient matrix \( A \). A pass of Gauss-Jordan elimination is then performed on
this matrix, with identical operations performed on the data portion of the block. When
a total of \( n \) coded blocks \( x = [x_1, x_2, \ldots, x_n] \) have been received, the matrix is reduced
to an identity matrix, the data portion of each block becomes the original block, without
the needs for additional decoding. The use of progressive decoding enjoys an additional
benefit: if a peer has received a coded block that is linearly dependent on existing blocks,
the elimination process will lead to a row of all zeros, so that this coded block can be
immediately discarded. No explicit linear dependence checks are required either during
or at the end of the transmission, and the matrix \( A \) in Eqn. 4.2 is always full rank at
the end of the process. Though Gauss-Jordan elimination usually leads to numerical
instability, it does not affect network coding since we operate in the Galois Field.

We illustrate progressive decoding in Fig. 4.8, with 3 blocks consisting one byte of data
each. The coding coefficients and the actual data (in their ASCII values) are separated by
a vertical bar. For each newly received coded block, Gauss-Jordan elimination is applied
to the coding coefficient matrix \( A \) on the left. The same operation is carried out in \( x \) on
4.3. PROGRESSIVE DECODING AND PERFORMANCE TUNING

Figure 4.8: An example of progressive decoding with Gauss-Jordan elimination
the right. Each iteration partially decodes the data by reducing $A$ to the Row Reduced Echelon Form (RREF). Once the last block is received, the segment is recovered with one more pass of the Guass-Jordan elimination.

To further accelerate the network coding operations, Shojania et al. [80] employed a hardware accelerator with SSE2 [81] and Altivec [82] SIMD vector instructions by replacing the conventional fast way of Galois Field multiplication through logarithm and exponential tables with loop-based multiplication. SIMD acceleration resulted in encoding speedups of around 8 and decoding speedups of around 5 for a typical range of coding parameters. Adding threads to the picture resulted in a near-linear speedup over the SIMD-accelerated implementation. Achieved coding bandwidth like 43 MB/sec with 64 blocks of 32 KB each suffices to saturate upload bandwidths of even super nodes in many applications. The details of this work can be found in [80].

Now, it is natural to ask what is the raw performance of network coding, with and without our progressive decoding? Furthermore, what is the maximum sustainable streaming rate of this optimized network coding library? With Crystal, we established a single streaming connection between one source and one receiving peer, each hosted by a dedicated server, interconnected by Gigabit Ethernet, without imposing bandwidth limits. We tested network coding in live streams with an average duration of 125 seconds. Assuming each segment represents 1 second of the playback, there are 125 segments. When tuning network coding, we set both density and aggressiveness to 100%, since we wished to evaluate the raw coding performance. We varied the block size from 128 bytes to 256 KB, and showed the average of results from all 125 segments. For each block size, we increased the streaming rate until the CPU is 100% saturated to find the maximum sustainable streaming rate.
### 4.3. PROGRESSIVE DECODING AND PERFORMANCE TUNING

(a) The encoding rate and maximum sustained streaming rates with different numbers of blocks. We attempted streaming rates up to 8 MB/sec.

<table>
<thead>
<tr>
<th>Size</th>
<th>Encoding Rate</th>
<th>Maximum Streaming Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>128B</td>
<td>0.50</td>
<td>8.00</td>
</tr>
<tr>
<td>512B</td>
<td>1.00</td>
<td>8.00</td>
</tr>
<tr>
<td>1KB</td>
<td>1.50</td>
<td>5.00</td>
</tr>
<tr>
<td>2KB</td>
<td>2.00</td>
<td>4.00</td>
</tr>
<tr>
<td>4KB</td>
<td>2.50</td>
<td>2.75</td>
</tr>
<tr>
<td>8KB</td>
<td>3.00</td>
<td>2.00</td>
</tr>
<tr>
<td>16KB</td>
<td>3.50</td>
<td>1.38</td>
</tr>
<tr>
<td>32KB</td>
<td>4.00</td>
<td>0.94</td>
</tr>
<tr>
<td>64KB</td>
<td>4.50</td>
<td>0.58</td>
</tr>
<tr>
<td>128KB</td>
<td>5.00</td>
<td>0.34</td>
</tr>
<tr>
<td>256KB</td>
<td>5.50</td>
<td>0.10</td>
</tr>
</tbody>
</table>

(b) The effects of progressive decoding: the time required to receive all blocks of a segment (including encoding and progressive decoding times), and the time used to recover the original blocks after the last coded block is received.

Figure 4.9: The performance of network coding library with progressive decoding
Fig. 4.9(a) shows our results of evaluating the coding performance, with respect to the encoding and decoding rate, as well as the maximum sustainable streaming rate. From these results, we observed that, thanks to the SSE2 accelerated implementation, the absolute coding performance is quite impressive, especially when there are fewer blocks. When there are only 32 blocks, the encoding rate exceeds 15 MB/sec on one CPU! On the flip side, we also observed that both encoding rate and decoding bandwidth rapidly decrease as the number of blocks per segment linearly increases. Network coding can support a wide range of streaming rates, from 100 KB/sec to more than 8 MB/sec, which are more than sufficient to accommodate typical streaming rates in real-world P2P streaming systems.

With SSE2-accelerated network coding operations, we noted that the decoding rate decreases faster than the encoding rate as the number of blocks increases. This is mostly due to the fact that the computational overhead of Gauss-Jordan elimination may not be as easily accelerated with SSE2 as straightforward vector multiplications. This phenomenon also makes the decoding process the bottleneck in the streaming process, i.e., the maximum sustainable streaming rate is limited by the decoding rate, as indicated in Fig. 4.9(a).

To illustrate the advantage of progressive decoding using Gauss-Jordan elimination, we ran the same experiment using the original implementation of the decoder in Sec. 4.1. In this experiment, we measured the time required to completely receive a segment (transmission time), and the time spent by the decoding process to recover the original blocks after all blocks have been received (recovery time). The transmission time includes the encoding time on the source.

In Fig. 4.9(b), the bar on the left of each setting represents the results from using
conventional decoding, and the bar on the right corresponds to progressive decoding. We noted that the recovery time of conventional decoding is longer than the transmission time in most cases. In fact, the conventional decoding process consumes a remarkable amount of CPU such that most of the segments can not be played according to their deadlines. Progressive decoding significantly reduces the time required to completely receive and recover a segment, with one exception. In the (128B, 784) setting, progressive decoding has a longer transmission time because the computational overhead of Gauss-Jordan elimination dominates the transmission time when the number of blocks is large. In general, the decoding time spent after receiving the last coded block is negligible. With progressive decoding, decoding times are almost completely concealed within the time required to receive the segment.

4.4 Summary

This chapter presents our highly-optimized network coding library. We have the following key observations in network coding performance. First, we are content with the raw one-to-one coding bandwidth when the number of blocks is small and when the block size is appropriate, but have noted the presence of linearly dependent blocks when we seek to tune density and aggressiveness to speed it up in more realistic topologies. Second, though encoding takes little computational overhead if we use a low density (such as 6%), decoding still requires a full matrix multiplication. Third, due to its requirement to buffer a certain number of coded blocks before it can serve new coded blocks, the latencies of such block accumulation are not negligible. Since it is not possible to use a very small aggressiveness value due to linear dependence constraints, such buffering latencies cannot be completely eliminated. We do wish to note, however, that the computational costs
introduced by network coding are very low in typical media streaming rates, especially when progressive decoding using Gauss-Jordan elimination is implemented.
Chapter 5

Lava: Network Coding in Live Peer-to-Peer Streaming

Avalanche [28, 29] has demonstrated — using both simulation studies and realistic experiments — that network coding may improve the overall performance of P2P content distribution, in networks consisting of up to about a hundred peers. The intuition that supports such a claim is that, with network coding, all blocks are treated equally, without the need to distribute the “rarest block” first, or to find them in the “end game” of a downloading process. While these are noteworthy observations, we note that content distribution applications deal with elastic traffic: one wishes to minimize downloading times, but there are no required lower bounds with respect to the instantaneous rate of a live session.

The requirements of P2P live multimedia streaming applications, however, have marked a significant departure from traditional applications of elastic content distribution. The most critical requirement is that the streaming rate has to be maintained for smooth playback. Each live streaming session may involve a live media stream with
a specific streaming rate, such as 800 Kbps for a typical Standard-Definition stream, generated with a modern codec such as H.264. The challenge of streaming is that the demand for bandwidth at the streaming rate (which is very similar to CBR traffic) must be satisfied at all peers, while additional bandwidth is, in general, not required.

Existing successes with P2P streaming, such as PPLive [1] and CoolStreaming [4], have demonstrated that P2P live streaming is not only feasible, but also practical at a large scale. Observing the recent success of using network coding in wireless mesh networks [50] and P2P content distribution [28], we naturally ask the following interesting question: Will network coding help in P2P live streaming? In this chapter, we endeavor to explore the benefits and tradeoffs of applying network coding in P2P live streaming, using Crystal with real traffic and our highly optimized network coding library.

Without loss of generality, we can consider a P2P streaming session as a P2P dissemination session with two additional constraints: (1) the content (segments) only becomes available as time goes by; and (2) each segment must be delivered before its playback time. In other words, segments are transmitted in a roughly sequential order, and the transmission of each segment is the same as disseminating a small file. We illustrate the concept of network coding in P2P streaming systems by revisiting the running example from Chapter 2, in which the source has two segments for streaming. Each segment corresponds to 1 second of the playback and is further divided into 4 blocks, i.e. 8 blocks in total as in Fig. 5.1. The encoding and decoding operations are performed within a segment only, i.e., each encoded block is a linear combination of the original blocks from the same segment. In this synthetic network, each peer, except the source, has upload capacity of 1.5S, where S is the size of a segment. The source upload capacity of 1.5S on each outgoing connection, with up to 3 connections.
Initially, five peers join the downloading session. Since the source can handle at most three peers at a time, only three out of five peers are connected to the source. At time 0, source starts to transmit coded blocks of different segments on each of its outgoing links. After 0.5 sec, these three peers have completely received three coded blocks and start to help the source to serve other participating peers. Five more peers join the session after 1 second into the session, they pull coded blocks from any peers that have sufficient bandwidth to them. Meanwhile, two out of the first three peers have completely downloaded the file, they stop receiving from the source so that the newly joined peers can share the spare bandwidth from the source. After 1.5 secs into the session, 54 blocks (i.e., 13.5 segments of size $S$) have been sent into the network. This example shows that with careful design, we can bring the benefits of network coding from file dissemination applications to P2P streaming systems.

Chapter 4 has investigated the practicality of randomized network coding, from the perspective of coding complexity. We have shown that the performance of network coding is acceptable when one uses a moderate number of blocks, in the order of less than 1000. We continue to explore the practicality of network coding in this chapter, but in the setting of P2P live streaming, rather than the raw coding throughput. We strive to faithfully report our results from an unbiased point of view, as well as our empirical observations and insights from analyzing logs of a large number of hands-on experiences. Our experimental results show that network coding makes it possible to perform streaming with a finer granularity, which reduces the redundancy of bandwidth usage, improves resilience to network dynamics, and is most instrumental when the bandwidth supply barely meets the streaming demand.
Figure 5.1: An example of network coding in P2P streaming systems
5.1 The Design of Lava

We designed Lava to answer the question whether network coding is helpful in P2P multimedia streaming. Our objective with Lava is to implement the best possible pull-based data-driven P2P live streaming protocol, achieving performance that matches real-world protocols (e.g., PPLive [1] and CoolStreaming [4]). This is necessary since we do not wish to make incorrect conclusions simply due to inferior performance when network coding is not used. To do so, we must first determine the standard protocol employed in existing streaming systems. Since none of the existing systems are open-source, we did our best to implement a pull-based protocol (henceforth codenamed Vanilla), based on the description of CoolStreaming in [4] and our understanding of PPLive [1]. Our emulation results in Sec. 5.3 confirm that the playback quality of Lava (without network coding) closely matches what we experienced in PPLive. As an intentional design decision to guarantee fairness in our comparison studies, network coding is implemented as a plugin component in Vanilla, such that both applications share identical protocols, configuration parameters, and design choices. In this section, we present the details of the streaming protocol as well as the application of network coding.

5.1.1 Playback Buffer

As in any practical P2P streaming protocol, a peer in Lava maintains a playback buffer in which segments are ordered according to their playback times. Segments are uniquely identified by a series of increasing sequence numbers, i.e., the first segment has sequence number 1, the second segment has sequence number 2, and so on. Segments are inserted into the buffer as they are received from upstream peers. The streaming algorithm registers a periodically playback event with the engine to emulate the playback motion
5.1. THE DESIGN OF LAVA

in streaming applications. A segment is removed from the buffer after being played, regardless of whether or not it is completely received. In the unfortunate event that a segment is not successfully received in time, it is skipped during playback. The number of playback skips is a good indicating factor for quantitatively evaluating the quality of the streaming playback.

To accommodate data swarming, peers periodically exchange information on segment availability, which is commonly referred to as buffer maps. The buffer map is presented in a bit vector format, in which each bit represents a segment in the playback buffer. The value of 1 indicates that the corresponding segment is available for transmission, and 0 otherwise. For example, if a playback buffer of size 12 has segments 4, 5, 6, 8, 9, 12, 13, its buffer map in bit-vector format should be [00000010 11101100 1100000], i.e., [4 236 96] in byte format. The first byte is the sequence number of the first segment in the buffer map. If the buffer size is not divisible by 8, we pad 0-bits at the end of the buffer map. According to the buffer map, those peers that have a particular segment available for transmission are referred to as the seeds of this segment. The seeds for all missing segments of peer are collectively referred as the upstream peers of this peer.

5.1.2 Synchronized Playback

A possible drawback of synchronized playback is that, the time between the occurrence of a live event in the media stream and its playback is the same across the board on all peers for the entire session. Though seemingly harmful, this may even be an advantage when live interaction is involved (such as live voting with SMS): all peers will view the same content at the same time, such that interactive behavior starts to occur at the same time as well. This design choice implies that each segment should be delivered within $\delta$
seconds from the source to the entire network in order to achieve synchronized playback at each peer. Hence, the playback time on each peer should be $\delta$ seconds after the playback point on the source. We refer to $\delta$ as the time lag of the stream, and define $t + \delta$ as the playback point in the network, where $t$ is the source playback point. The actual value of $\delta$ is determined by the available bandwidth in the network and the diameter of the network topology, assuming peers blocked by the NAT and firewalls contribute no bandwidth. In a network with higher bandwidth supply, peers with higher upload capacity adopts more downstream peers, resulting in smaller network diameter.

To prevent segment skips during playback and to better accommodate data swarming, there must be sufficient segments available in the playback buffer at all time. For this reason, a peer does not start playback immediately after joining the session. Instead, a newly joined peer retrieves segments that are $\epsilon$ seconds after the current playback point in the network. This time difference is referred to as the initial buffering delay, denoted by $\epsilon$. Regardless of the status of the playback buffer, the peer starts playback in exactly $\epsilon$ seconds. During such an initial buffering process, the download capacity of the new peer is usually saturated, and the playback buffer fills up rapidly. The design of synchronized playback leads to at least $\epsilon$ seconds worth of content in the playback buffer at all time, as long as the average downloading rate exceeds or meets the streaming rate.

5.1.3 Segment Scheduling

During smooth playback, all segments should be readily available before their playback times. For each segment that is due for playback, the peer schedules a new segment for transmission by sending a request to an arbitrarily selected seed. Assume each segment represents 1 second of playback, then a peer requests a new segment every second. If a
5.1. THE DESIGN OF LAVA

requested segment is not received within the expected time — the per-segment timeout period, either due to peer departures or insufficient bandwidth, Lava requests the segment again (possibly from a different seed).

To minimize playback skips and to fully utilize download capacity, we introduce the low and standard buffering watermarks, which mark the alert mode and normal playback mode, respectively. Segments in the buffer are grouped into batches, as shown in Fig. 5.2, and each batch consists of a set of consecutive segments. To ensure that early segments receive higher priority in transmission, the peer begins with the first batch in the buffer, and moves to the next batch only if all segments in the previous batch have been scheduled. Segments from the same batch are requested in an arbitrary order, to promote bandwidth sharing among peers. The pseudo-code of the scheduling algorithm is given in Table 5.1. Line 1 through line 4 is executed after each playback, i.e., a new segment is requested after each playback.

![Segment scheduling in Lava](image)

Figure 5.2: Segment scheduling in Lava

A peer enters the alert mode if there are missing segments below the low buffering watermark. The throughput must have been below the streaming rate earlier in the session. In this case, for each missing segment entering the alert zone, one additional
Table 5.1: The segment scheduling algorithm

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>if bucket is empty</td>
</tr>
<tr>
<td>2</td>
<td>fill the bucket with the next batch of missing segments</td>
</tr>
<tr>
<td>3</td>
<td>if buffering level below the high watermark</td>
</tr>
<tr>
<td>4</td>
<td>randomly select a segment from the bucket and request it</td>
</tr>
<tr>
<td>5</td>
<td>if the segment before the low buffering watermark is missing</td>
</tr>
<tr>
<td>6</td>
<td>randomly select a segment from the bucket and request it</td>
</tr>
</tbody>
</table>

segment is scheduled for transmission when scheduling the next segment for transmission, in effort to make up the earlier loss in throughput. Line 5 to line 6 in Table 5.1 ensures that only one additional request is issued for each missing segment in the alert zone, by checking only the segment that is right before the low buffering watermark. Naturally, the peer returns to its normal mode after it reaches the low buffering watermark again.

Let each segment represents 1 second of the playback. In the worst case, when all segments in the alert zone are incomplete at time \( t \), two segments are scheduled for transmission every second. There will be at most \( 2 \times \tau \) outstanding requests at time \( t + \tau \), where \( \tau \) is the number of segments in the alert zone. If none of the segments in the alert zone is completely received in the next \( \tau \) seconds, the peer will have \( 4 \times \tau \) segments in transmission. If such scenario continues, the number of outstanding requests increases at the rate of 2 per second. Lava limits the buffer size to impose an upper bound for this number and to avoid having excessive number of requests sent to the seeds. Furthermore, to prevent overloading the seeds, Lava also limits the number of concurrent TCP connections on each peer. With appropriate settings of the buffer size and initial buffering delay, as verified in Sec. 5.3, the downloading rate of a peer is seldom slower than the streaming rate. Hence, the number of consecutively available segments in the buffer is always more than the number of segments received during the initial buffering delay.
5.1.4 Streaming Protocol

In any P2P network, a peer can join into or depart from a session at any time. In Lava, we design the tracker to bootstrap each peer with a maximum of 5 existing peers, on which the number of available segments in the buffer exceeds a certain threshold. To join a session, peer \( p \) sends a join request to existing peers. The peers that acknowledge the request with an accept message become the upstream peers of \( p \), and insert \( p \) into their downstream peer list. Up to this point, peer \( p \) is engaged into a live session, and will potentially become an upstream peer of other peers in the network. An engaged peer performs one of the following five tasks: (1) periodically sending its buffer status, referred to as buffer map (see Sec. 5.1.1), to downstream peers, (2) performing playback in the buffer, (3) requesting a segment, (4) serving a segment, (5) canceling a segment request. The sequence of these actions is best illustrated with examples.

**Example 1:** In Fig. 5.3, peer 1 has one upstream peer (peer 3) and one downstream peer (peer 2). As an upstream peer of peer 1, peer 3 periodically sends its buffer map to downstream peers. When a segment is due for playback, the peer removes it from the playback buffer to make room for future segments. After each playback, the peer schedules another missing segment in the buffer for transmission by sending a request to one upstream peer, e.g. peer 2 → peer 1. Upon receiving the request, peer 1 sends the segment to peer 2. In networks with sufficient bandwidth and acceptable end-to-end delay, the protocol operates as in Fig. 5.3, *i.e.*, all segments are delivered in time.

In reality, a P2P network consists of a large number of peers with limited upload bandwidth. Moreover, a peer can be potentially overloaded with excessive segment requests, due to the dynamic nature of P2P networks. In either case, a segment may not be delivered in time. To improve the resilience to bandwidth variations and peer dynamics,
we employed a timeout mechanism for monitoring the delivery status of each segment. The timeout value is the estimated time required to send a request and to completely receive a segment. For instance, if each segment represents 1 second of the playback, and the end-to-end delay of a request is less than 0.1 seconds, we can set the timeout value to 1.1 seconds. A peer cancels a request in 1.1 seconds after sending the request, and may request the same segment from another peer.

Example 2: In Fig. 5.4, we assume that peer 1 has a long list of incoming requests. It takes peer 1 longer than 1.1 seconds to start transmitting the first segment requested by peer 2. Instead of waiting, peer 2 actively cancels this request, and schedules the transmission with another peer. In the case where a peer (e.g., peer 3) is not overloaded with requests, it immediately starts transmitting segment i upon the reception of the request from peer 1. The timeout on peer 1 is triggered since the transmission took
longer than expected. Peer 1 then requests segment \( i \) from peer 4. As soon as the transmission of segment \( i \) is completed, peer 1 cancels all outstanding requests related to this segment. However, this timeout mechanism does not guarantee zero bandwidth waste. For example, in Fig. 5.4, segment \( j \) is due for playback shortly after peer 2 sends a request for segment \( j \) to peer 1. Although a cancellation notice is sent immediately after playback, segment \( j \) is still transmitted and is discarded. We will show how network coding can reduce such bandwidth waste in Sec. 5.1.5. We omit buffer map messages in Fig. 5.4 for clarity.

![Diagram](image)

Figure 5.4: Another example of the pull-based streaming protocol

### 5.1.5 Network Coding Plug-in

Network coding (both encoding and decoding processes) is brought into Lava as a plugin component, such that it shares identical protocol design and parameter settings with
Vanilla. When using network coding, segments are further divided into blocks and then coded. Peers exchange coded blocks of much smaller size instead of raw segments. As illustrated in Fig. 5.1, it is intuitive to conceive at least one important benefit of using network coding in a live P2P streaming session: a peer may download coded blocks of a segment from multiple seeds at the same time, without requiring any protocol to coordinate these seeds. Each seed simply starts to serve coded blocks of the segment upon receiving a request. When a peer completely decodes the segment, it informs all its seeds by sending each one of them a short message to terminate the transmission of this segment. Without using coding, a peer will have to explicitly request particular blocks from a particular seed, which requires more frequent buffer map exchange.

Figure 5.5: Network coding in Lava

The architecture of the network coding implementation in Lava is summarized in Fig. 5.5. Every time a new coded block is received, a peer performs progressive decoding by applying Gauss-Jordan elimination to the new coded block and all previously received blocks of this segment. If the new coded block does not lead to a row of 0’s, it is added to the segment in the playback buffer. As the decoding process progresses, intermediate outcomes of Gauss-Jordan elimination are stored in the playback buffer, until the entire segment is completely decoded. An important note we wish to make is that, depending
on the settings of the aggressiveness, the encoding process uses either the intermediate or fully decoded blocks from the playback buffer, and progresses *concurrently* with the decoding process. This guarantees that the encoding progress may start before the decoding process is complete. We also tune the number of blocks per segment so that the encoding process is much faster than the upload link capacity, and would not become the bottleneck of the streaming session.

We further illustrate the concept of network coding and its advantages by revisiting the two examples in Sec. 5.1.4. In the following examples, each segment is further divided into 3 blocks. After sending a request, a peer $p$ sends a `cancel` message when either the timeout is triggered or 3 linearly independent coded blocks are received. The upstream peers of $p$ keeps sending coded blocks until a `cancel` message is received.

**Example 3:** (example 1 revisit) As shown in Fig. 5.6, a peer can still request a segment from a single upstream peer, *e.g.*, peer 1 requests a segment from peer 3. The maximum number of coded blocks sent by an upstream peer is capped by the number of blocks in a segment, which is 3 in this example, since any additional block will be linearly dependent with the previous blocks. The advantage of network coding is that it takes less time to completely receive a segment by having multiple peers collaboratively serve a segment. For example, peer 2 requests a segment from both peers 1 and 3. Upon receiving this request, peers 1 and 3 start sending coded blocks of the segment until a `cancel` message is received. The tradeoff is that the propagation delay of the `cancel` message may still lead to bandwidth waste. After peer 2 cancels the segment with peers 1 and 3, two more coded blocks are received during the transmission of the `cancel` message. We omit buffer map messages in Fig. 5.6 for clarity.

**Example 4:** (example 2 revisit) In Fig. 5.7, peer 2 requests a segment from peers
5.1. THE DESIGN OF LAVA

Figure 5.6: An example of the pull-based streaming protocol with network coding

Figure 5.7: Another example of the pull-based streaming protocol with network coding
5.2. IMPLEMENTATION

1 and 3. Peer 1 has a long list of incoming requests. Unlike the case in Example 2 (Fig. 5.4), peer 3 serves all 3 coded blocks before the timeout on peer 2 triggers. For the cases where the transmission took longer than expected, the protocol behaves the same, with and without network coding. Finally, we see another benefit of network coding in the transmission of segment $j$. Since the cancellation notice of segment $j$ is sent shortly after the request, only two coded blocks are sent from peer 1 to peer 2. Upon receiving the cancel message, peer 1 immediately stops further transmission. As a result, peer 2 discards only 2 blocks of segment $j$ instead of the entire segment. In this case, we save $1/3$ of the bandwidth, as compared to Example 2 (Fig. 5.4). We omit buffer map messages in Fig. 5.7 for clarity.

5.2 Implementation

To conduct a reality check on the practicality of network coding in real-world P2P streaming systems, we must address the following design challenges:

- Actual network traffic needs to be involved to emulate practical streaming sessions. Hence, a potentially large number of TCP connections and UDP flows need to be efficiently managed by each peer.

- To avoid miscalculations and incorrect conclusions due to an inferior implementation of randomized network coding, a highly optimized implementation with maximum performance is a must, with attention to details.

- Peer arrivals and departures in a particular session need to be emulated, which leads to the challenges of maintaining up-to-date peer lists and handling dynamic network connections.
5.2. IMPLEMENTATION

- Since we wish to maximize the number of peers to be emulated on each cluster server, the processing and memory footprints of our implementation must be minimized.

- Network coding should be evaluated in the same context as a conventional P2P streaming protocol, with identical parameter settings and protocol design.

In fact, the first four challenges mentioned above are addressed by Crystal. This allows us to focus on the design and implementation of the P2P streaming protocols, as well as the incorporation of network coding. Fig. 5.8 presents the architecture of Lava, in which all components, except the streaming protocol, are provided by Crystal. The dash lines illustrate the data flow in Lava.

![Figure 5.8: Lava in Crystal](image)

For incoming messages:

1. The network layer receives the message into the appropriate incoming queue.
2. The engine takes the message and routes it to the algorithm (streaming protocol).

3. *(Network coding only)* The algorithm employs the progressive decoder to decode the message.

4. The message is stored into the playback buffer.

To serve downstream peers:

5. The algorithm locates the requested segment in the playback buffer.

6. *(Network coding only)* The algorithm employs the encoder to produce an encoded block.

7. The message is enqueued into the appropriate outgoing queue in the network layer via the engine.

8. The network layer sends the message to the corresponding downstream peer.

We configured the tracker in Crystal to bootstrap each newly joined peer with 5 randomly selected peers in the network. The tracker does not interact with the peer after bootstrapping them. Peers employ a gossip-based protocol to discover existence of other peers in the network. In short, each peer maintains a list of peers that it has already discovered or received from the tracker. Periodically, a peer selectively sends a subset of the list to peers it has discovered. Eventually, each peer in the network has an almost complete list of existing peers.

Crystal allows us to focus only on the protocol implementation instead of the mundane tasks of socket and multi-threaded programming. The pseudo-code is given in Table 5.2. Each function is invoked by the engine as the events are triggered or messages are received.
5.2. IMPLEMENTATION

Table 5.2: The skeleton of Lava using Crystal

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bootstrap(Msg * m)</td>
<td>Contact the tracker or parse the topology file for an initial set of peers. Register a periodic timer for playback with callback playback(). Register a periodic timer for buffer map exchange with callback sendBuffermap().</td>
</tr>
<tr>
<td>process(Msg * m)</td>
<td></td>
</tr>
</tbody>
</table>
  - switch m->type() |
  - case join: Add the peer who sent this message to the downstream peer list. Send accept to the new downstream peer. |
  - case accept: Receive the acknowledgment from upstream peers for the join request. |
  - case buffermap: Update the buffer status of the corresponding upstream peers. |
  - case requestSeg: Process or enqueue the request from a downstream peer. |
  - case cancelSeg: Remove the corresponding request from the request queue. |
  - case data: Insert the received segment (or coded block) to the playback buffer. |
  - case default: Unknown message type. |
  - return consumed |
| removePeer(Msg * m) | Remove a peer from the downstream peer list. |
| joinSession(Msg * m) | Send a join message to the initial neighboring peers. |
| leaveSession(Msg * m) | Leave a session by disconnect all downstream and upstream peers. |
| sendBuffermap() | For every downstream peer p, send(buffermap, p). |
| playback() | Remove the segments that is due for playback from the playback buffer. |
5.3 EVALUATION OF NETWORK CODING IN P2P STREAMING

The algorithm only needs to call one function of the engine: the `send()` function. Similar to the simple relaying protocol in Chapter 3, the message handler, `process()`, switches on different message types. The `removePeer()` is called by the engine when a connection failure is detected in the network layer. The `joinSession()` and `leaveSession()` are callback functions for the join and leave events. The other events in the events file are automatically handled by the engine. Lava introduces two additional events: `playback` and `buffermap`. In `bootstrap()`, the two periodical timers are registered with the engine.

5.3 Evaluation of Network Coding in P2P Streaming

With Lava, we are now ready to perform an empirical “reality check” of network coding in P2P live streaming. The focus of our study is on the practicality, performance and overhead of random network coding, as compared to Vanilla, a standard P2P streaming protocol without using network coding. The ultimate objective of this study is to answer the question: Is network coding helpful in P2P live streaming?

We deployed streaming sessions in a cluster of 44 dual-CPU servers. In all experiments, the uplink bandwidth on the dedicated streaming server of the session is constrained to 1 MB/sec. In reality, a P2P network usually consists of a certain number of peers with upload capacity more than 1 MB/sec. For a more challenging scenario, we emulated all peer connections as DSL uplinks, uniformly distributed between 80 and 100 KB/sec. We used a streaming bit rate of 64 KB/sec, which must be satisfied at all peers during their streaming playback. In our experiments, unless otherwise specified, each segment represents 1 second of the playback, and is divided into 32 blocks, offering a satisfactory encoding and decoding bandwidth with our implementation of randomized network coding (around 19 MB/second). We set the buffer size to 30 seconds, the lag
time to 30 seconds, the initial buffering delay to 20 seconds, and the batch size to 10 seconds. Each streaming session lasts for 10 minutes.

To compare the performance, we evaluated several important metrics: (1) Playback skips: measured as the percentage of segments skipped during playback. A segment is skipped during playback if it is still not completely received at the time of playback. (2) Bandwidth redundancy: measured as the percentage of discarded segments, due to obsolescence or linear dependence, over all completely received segments. (3) Buffering levels: measured as the percentage of completely received segments in the playback buffer on each peer during a live session. For all measurements, we took the average from all peers in the network.

5.3.1 Performance of Network Coding

While we have previously shown, in Chapter 4, the performance of network coding when CPU is intentionally 100% saturated, we would like to focus on the performance of network coding in a realistic streaming session with a fixed streaming rate. For this purpose, we deployed a streaming session with a typical streaming rate of 64 KB/sec between two peers. Real-world P2P applications, such as PPLive, usually use streaming bit rates of less than 50 KB/sec. Fig. 5.9(a) and Fig. 5.9(b) show that such a typical streaming rate is sustainable regardless of the number of blocks in each segment, with negligible decoding times after a segment is completely received. Empirically, we also observed (not explicitly shown in the figure) that the CPU usage increases from 4% to 15% as the number of blocks increases. Beyond benefits of negligible CPU usage, the other advantage of using a smaller number of blocks, 32 in this case, is that the overhead for carrying the coding coefficients in each coded block is smaller. For instance, the coding coefficient overhead is 100% when using 256 blocks, where as it is only 1.5% when
5.3. EVALUATION OF NETWORK CODING IN P2P STREAMING

(a) Encoding and decoding bandwidth

(b) Encoding and decoding overhead

Figure 5.9: Coding performance for a session with a streaming rate of 64 KB/sec
5.3. EVALUATION OF NETWORK CODING IN P2P STREAMING

using 32 blocks.

5.3.2 Scalability

We now compare Vanilla and network coding when the number of peers in the network scales up. In this experiment, we added one peer on each server at a time, until all 44 servers are fully saturated. A 64 KB/sec streaming session is deployed in the network for 10 minutes. Fig. 5.10(a) shows that, in networks consisting of 572 peers or less, network coding is approximately as scalable as Vanilla in terms of playback quality. The CPU usage of both algorithms grows linearly with respect to the network size. Due to computational complexity, when the network size reaches 572 (12 peers on each server), network coding consumes more than 85% of resources on each CPU, and its performance degrades significantly. Nevertheless, we argue that such a limitation is not applicable in reality, since each peer runs only one coding instance, which consumes less than 10% of the CPU. Fig. 5.10(a) shows that the bandwidth redundancy introduced by Vanilla is approximately 20% of the total network traffic, in networks with more than 264 peers. Hence, network coding is more scalable in term of bandwidth redundancy.

As an important metric to gain insights on the playback quality, we examined the fluctuations in average buffering levels over the course of a streaming session. Fig. 5.10(b) compares the average buffering levels of network coding and Vanilla in three representative networks, from which we drew two key observations. First, the buffering level ramps up quickly and remains stable when using network coding, while Vanilla maintains a much lower level with a slight variation over time. Second, the buffering levels of both algorithms decrease as the network size increases; however, the change in the case of network coding is not significant.
5.3. EVALUATION OF NETWORK CODING IN P2P STREAMING

Figure 5.10: Scalability in terms of network size
It is interesting to note that network coding does not offer better playback quality than Vanilla, while maintaining a remarkably high and stable buffering level, regardless of the network size. To trace the cause of such phenomenon, we timed the transmission of each segment in a 10-minute session, and took the average from 264 peers. During data streaming, when network coding is employed, peers are exchanging coded blocks instead of segments. On one extreme, if a peer discovered only one seed for a particular segment, coded blocks are sequentially produced, leading to longer delay in transmission. On the other extreme, if a peer has more than 32 seeds for a segment, the encoding process is evenly distributed across 32 seeds, i.e. each seed encodes at most one block. The average transmission time depicted in Fig. 5.11(a) verified this conjecture. It takes network coding longer time to receive earlier segments, since fewer seeds are known by a newly joined peer. As the session progresses, more seeds are discovered; hence, the transmission time is significantly reduced.

The conclusion drawn from Fig. 5.11(a) is further verified in Fig. 5.11(b), which illustrates the percentage of peers in the network that successfully played each segment in the 10-minute session. We noted that network coding has more skips in the first 60 seconds of a session. However, all peers experience perfectly smooth playback after 70 seconds into the session. The skips in the first 60 seconds is referred as the initial skips.

### 5.3.3 Tuning Density and Aggressiveness

Theoretically, a lower coding density leads to a smaller number of blocks being coded, which reduces the coding complexity. In addition, a lower aggressiveness setting leads to more “supply” of coded blocks. That said, if peers become too aggressive and start producing new coded blocks too soon, it may not have a sufficient number of original
5.3. EVALUATION OF NETWORK CODING IN P2P STREAMING

Figure 5.11: The transmission and playback status in a 64 KB/sec streaming session on 264 peers

(a) The average transmission time of each segment in a 10-minute session

(b) Percentage of peers successfully played each segment in a 10-minute session
blocks in its playback buffer, leading to a potential of linearly dependent blocks being produced. Since the transmission of such linearly dependent blocks consumes bandwidth, they lead to redundancy in terms of bandwidth usage. Bandwidth redundancy may also be caused by blocks that are received later than the per-segment timeout or after the playback time, due to busy seeds or lack of bandwidth.

We are interested in the effects of tuning density and aggressiveness parameters in network coding, compared to Vanilla. For this purpose, we established a streaming session on 264 peers. We first varied the aggressiveness, and then vary the density. In Fig. 5.12(a), although the percentage of playback skips remains insignificant and almost unchanged when tuning the aggressiveness, the bandwidth redundancy is minimized when the aggressiveness is 0.25. Furthermore, Fig. 5.12(b) shows that the average buffering level grows faster and remains at a higher level when aggressiveness is 0.25. We continued the density experiment with the aggressiveness as 0.25. As shown in Fig. 5.13, both the percentage of playback skips and bandwidth redundancy remain insignificant and almost unchanged as well when tuning the density.

Overall, variations in both aggressiveness and density do not significantly affect the playback quality. The percentage of linearly dependent blocks discarded at the peers is insignificant, less than 0.2% of the network traffic. Though these results may seem counter-intuitive, we believe that it is primarily due to the nature of live streaming playback, in that there does not exist sufficient room in each segment for these coding parameter settings to take effect. Since tuning these parameters does not materially affect streaming quality, and the best playback and buffering level is achieved when the aggressiveness and density are 0.25 and 0.75, respectively, we used this setting in the remaining experiments.
5.3. Evaluation of Network Coding in P2P Streaming

(a) Average playback skips and bandwidth redundancy when tuning aggressiveness (density set to 100%)

(b) The average buffering level when tuning aggressiveness (density set to 100%)

Figure 5.12: Effects of the aggressiveness in a 64 KB/sec streaming session with 264 peers
5.3. EVALUATION OF NETWORK CODING IN P2P STREAMING

5.3.4 Tuning Experimental Parameters

We conducted extensive experiments to find the best possible performance of network coding in live P2P multimedia streaming. From these experiments, we discovered that network coding is very sensitive to parameter tuning. This section presents a selective set of the experimental results.

We first tuned the buffer size from 30 seconds to 60 seconds. In the steady state, the playback buffer is filled up to the standard buffering watermark, the first 20 seconds in our case; and the algorithm schedules transmissions at the rate of 1 segment per second. Hence, the buffer should be at least large enough to hold one segment in addition to the standard buffering watermark. When using a large buffer, a peer may have too many outstanding requests. Thus, the earlier segments receive less attention and are more likely to miss their times. Fig. 5.14(a) shows that a large buffer does not necessarily
offer better playback quality and buffering level. Although the buffering levels ramps up faster in larger buffers, they still converge to the same level when the session enters the steady state.

We then fixed the buffer size to 30 seconds, and adjusted the initial buffering delay from 10 seconds to 20 seconds. Intuitively, the longer initial buffering delay should offer better playback quality and higher buffering level, since it allows the buffer to accumulate more segments before the playback starts. As expected, fig. 5.14(b) shows that the longer delay does improve the playback quality and the buffering level.

With respect to the segment size, we varied the time represented by a segment from 1 second to 6 seconds. With fixed buffer size, the buffer holds fewer segments as segments become larger, leading to less randomness in scheduling segments for transmission. In the first experiment, we set the block size to 2048 bytes, and increased the number of blocks within a segment from 32 to 196. The transmission overhead, message header and coding coefficients, grow proportionally with respect to the number of blocks in a segment. Fig. 5.15(a) shows that both algorithms cannot maintain a satisfactory playback quality when the segment is longer than 2 seconds. In the second experiment, we fixed the number of blocks in a segment at 32, and increased the block size from 2 KB to 12 KB as the segment size increases. Fig. 5.15(a) shows that network coding offers much better playback than the previous experiment; thus, it is less sensitive to the block size than it does to the number of blocks.

As indicated Fig. 5.15(b), Vanilla cannot effectively utilize the bandwidth as the segment size increases, resulting in dramatic performance deterioration in playback quality. Since network coding makes it possible to perform data streaming in a finer granularity, the bandwidth redundancy is significantly less severe.
5.3. EVALUATION OF NETWORK CODING IN P2P STREAMING

Figure 5.14: Effects of the playback buffer size and the initial buffering delay in a 64 KB/sec streaming session with 264 peer
5.3. EVALUATION OF NETWORK CODING IN P2P STREAMING

(a) The average playback skips during the session, when tuning the segment size

(b) The average bandwidth redundancy during the session, when tuning the segment size

Figure 5.15: Effects of the segment size in a 64 KB/sec streaming session with 264 peers
5.3.5 When Bandwidth Demand Meets the Supply

In our previous experiment, bandwidth supply outstrips demand since the peer upload bandwidth is higher than the streaming rate. It may appear that network coding does not lead to improved performance when compared to Vanilla. The question naturally becomes: Is this the case when the supply-demand relationship of bandwidth changes? In a network consisting of 264 peers, with all DSL-connections except the source, the average bandwidth supply in the network is 93 KB/sec for each peer. We ran a set of experiments to compare network coding and Vanilla, with three different streaming rates: 64 KB/sec to represent the case where supply outstrips demand, 73 KB/sec to represent an approximate match between supply and demand, as well as 75 KB per second, when the demand exceeds the supply of bandwidth. When the streaming rate is 75 KB/sec, the average bandwidth demand by Vanilla is more than 93 KB/sec from each peer, including the protocol messages and the 20% redundant traffic.

From Fig. 5.16(a), we observed that network coding performs significantly better than Vanilla when there is a close match between supply and demand. When supply outstrips demand or vice versa, there does not exist a significant difference between the two. We also observed that, when bandwidth supply outstrips or meets the demand, network coding is able to consistently maintain a buffering level at the standard buffering watermark, while Vanilla is striving to maintain the buffering level above the low buffering watermark. To further show the benefits of network coding, we reduced the upload capacity on the streaming source, from 1 MB/sec to 250 KB/sec. Fig. 5.16(b) illustrates the same pattern as in Fig. 5.16(a) as the source bandwidth supply drops. The moral of the story is that, in comparison to Vanilla, the streaming quality of network coding excels in the challenging situation when the supply of upload bandwidth barely meets
5.3. EVALUATION OF NETWORK CODING IN P2P STREAMING

(a) The average playback skips and buffering level during the sessions with different streaming rates

(b) The average playback skips and buffering level during the sessions with different upload capacity on the source

Figure 5.16: Effects of the bandwidth supply and demand in a 64 KB/sec streaming session with 264 peers
the streaming demand.

5.3.6 Effects of Peer Dynamics

To investigate the effects of network coding in the case of dynamic peer arrivals and departures, we used Perl scripts to generate peer join and departure events in the events file. Based on Stutzbach et al. [79], both interarrival times of peer join events and peer lifetimes can be modeled as a Weibull distribution \((k, \lambda)\), with a PDF \(f(x; k, \lambda) = \frac{k}{\lambda} (\frac{x}{\lambda})^{k-1} e^{-\left(\frac{x}{\lambda}\right)^k}\), under various settings of the shape parameter \(k\) and scale parameter \(\lambda\).

The first case we examined is the performance under different peer join rates. According to Stutzbach et al. [79], peer interarrival time in a file downloading session follows a Weibull distribution with \(k = 0.79, 0.53, \) or 0.62. We set \(k\) to the average value of the three, 0.65. We varied the mean value \(\lambda\) from 80 to 200 seconds so that the join rate reduces and network becomes more dynamic over time. When \(\lambda = 80\), it is similar to the flash-crowd scenario, in which more than 90% of the peers join the session in the first 60 seconds. However, Stutzbach et al. [79] only studied the case of file downloading, where peers usually stay in the session until the completion of the download. In a streaming session, peers may join and leave at any time, and may not necessarily follow the same distribution. For this reason, we repeated the experiment with \(k = 2\), in which the peer interarrival time is approaching normal distribution with different means. For clarity, the plot of each join distribution is shown in Fig. 5.17. The majority of the peers join the session in the first 60 seconds when \(k = 0.65\), whereas peers join the session at a much more slower rate when \(k = 2\). Hence, the network is more dynamic in the later case.

As discussed in Sec. 5.3.2, the number of seeds has direct impact on the performance of network coding. In contrast to the flash-crowd scenario, more seeds with good buffering level are available when peers slowly join the network. For this reason, network
5.3. EVALUATION OF NETWORK CODING IN P2P STREAMING

Weibull (80, 0.65)
Weibull (120, 0.65)
Weibull (200, 0.65)

Figure 5.17: The PDFs of the peer join rate distributions

coding outperforms Vanilla in terms of overall playback quality in Fig. 5.18(a). Moreover, Fig. 5.18(a) also indicates that network coding achieves perfect playback after 60 seconds into the session. Fig. 5.18(b) shows that the buffering level of Vanilla decreases as more peers joining the network, whereas that of network coding remains the same for all peer join rates.

We then switched our attention to the lifetime of each peer. In this experiment, the join events follow the Weibull distribution (80, 0.65). According to Stutzbach et al. [79], the lifetime of a peer in a file downloading session follows a Weibull distribution with $k = 0.34, 0.38, \text{ or } 0.59$. We set $k$ to the average value of the three, 0.43. We varied the mean value $\lambda$ from 500 to 300 seconds so that average lifetime of a peer becomes shorter, leading to a higher churn rate. For the same reason, we repeated the experiment with $k = 2$. For clarity, the plot of each join distribution is illustrated in Fig. 5.19.

Although network coding offers slight worse overall playback quality than Vanilla, it
5.3. EVALUATION OF NETWORK CODING IN P2P STREAMING

(a) The average playback skips with and without the initial skips during the session, when tuning peer join rate

(b) The average buffering level during the session, when tuning peer join rate

Figure 5.18: Effects of the peer join rate in a 64 KB/sec streaming session with 264 peers
5.3. EVALUATION OF NETWORK CODING IN P2P STREAMING

Figure 5.19: The PDFs of the peer lifetime distributions

still achieves perfect playback after 60 seconds into the session, as shown in Fig. 5.20(a). Vanilla is not able to maintain its performance in networks with higher churns, when $k = 2$. In Fig. 5.20(b), the buffering level of Vanilla is stabilized only after majority of the peers joined the session. As peers departing from the session, the ratio between bandwidth supply and demand increases. In this case, Vanilla managed to bring the buffering level up to the standard watermark, and network coding is able to maintain the same buffering level, regardless the network churns.

To conclude, we made the following important observations in our empirical studies. First, network coding offers the same playback quality as Vanilla does, with up to 24% less network traffic. Second, the aggressiveness and density settings do not materially change the performance of network coding. Third, the buffer size, initial buffering delay, and segment size may have a significant impact on the performance of network coding. Fourth, network coding requires sufficient number of seeds in order to achieve its optimal
5.3. EVALUATION OF NETWORK CODING IN P2P STREAMING

(a) The average playback skips with and without the initial skips during the session, when tuning peer lifetime length

(b) The average buffering level during the session, when tuning peer lifetime length

Figure 5.20: Effects of the peer lifetime length in a 64 KB/sec streaming session with 264 peers
performance. Finally, despite the initial skips due to lack of seeds, network coding is able to maintain perfectly smooth playback and stable buffering level, with the presence of peer dynamics.

5.4 Summary

The objective of this chapter is to evaluate the potential and tradeoffs of applying network coding in P2P live streaming sessions, using an experimental testbed in a server cluster, with emulated peer upload capacities and peer dynamics. To achieve a fair comparison between using and not using network coding, we include a pull-based P2P live streaming protocol in Lava. As a result of our empirical studies, we believe that network coding has demonstrated its advantages in P2P live streaming when peers are volatile and dynamic with respect to their arrivals and departures, i.e., the churn rate of the network is high, and in the case that the supply of upload bandwidth barely exceeds the bandwidth demand in the session. However, we have also observed that network coding requires precise parameter tuning to achieve its optimal performance, which could be specific to the scale of P2P networks. We do wish to note that although we only studied the performance of network coding in one streaming protocol, we believe that any modification to the streaming system will not materially change our conclusions. Certain changes might reduce the performance difference between network coding and Vanilla, while others might result in more obvious performance difference. To the best of our knowledge, there has been no existing work that has systematically studied the practicality of using network coding in P2P live streaming applications, especially using a realistic testbed that involves actual network traffic and peer dynamics.
Chapter 6

$R^2$: A High-Quality Peer-to-Peer Streaming System

In Chapter 5, we have found that network coding may offer some advantages when peers are volatile and dynamic with respect to their arrivals and departures. While Lava has focused on a fair comparison study without any changes of the P2P streaming protocol, we believe that the advantages of network coding have not been fully explored with a traditional pull-based protocol. Convinced that random network coding is beneficial, we are determined to redesign the P2P streaming protocol to take full advantage of random network coding by revisiting the entire spectrum of design choices, and by introducing randomizing elements into the algorithm. We present $R^2$, our new streaming algorithm designed from scratch to incorporate random network coding with a randomized push algorithm. Though the first and most important requirement of $R^2$ is to achieve perfect playback, $R^2$ is nevertheless designed to improve the overall performance, in terms of initial buffering delays, resilience to peer dynamics, as well as reduced bandwidth costs on dedicated streaming servers. Unlike the rigid and more “static” push design using
6.1. THE DESIGN OF $R^2$

predetermined trees in [36], a peer in $R^2$ randomly chooses a segment to push whenever a coded block (of a very small size) is to be sent. Peers in $R^2$ proactively send segments that are missing from their downstream peers — no pull is ever required — and there is never a need to switch between pull and push mechanisms on-the-fly.

The idea of random push is partly inspired by Chunked Codes [83]. With Chunked Codes, the message to be communicated is logically partitioned into disjoint “chunks” of contiguous symbols. To encode at the source node, it randomly and uniformly chooses a chunk, and performs a dense linear combination of input symbols from this chunk. The intermediate node, again, randomly and uniformly chooses a chunk, and then performs a dense linear combination of so-far received coded symbols within this chunk. Decoding at the receiver is performed as a regular Gaussian elimination to solve the symbol diagonal matrix. Chunked Codes are designed for the network erasure channel, which is applicable to the P2P streaming case, due to peer dynamics. However, the design of Chunked Codes has been purely analytical and specifically excluded supports for the streaming case, due to its strict timing requirements. With $R^2$, we focus on the design of a practical streaming solution based on random push of coded blocks of each segment, as well as an experimental evaluation of its effectiveness. Since $R^2$ only encodes within a particular segment, it enjoys similar advantages in terms of coding complexity as Chunked Codes do.

6.1 The Design of $R^2$

In this section, we present $R^2$, our attempt at a complete redesign of the live P2P streaming protocol, to take full advantages of random network coding.
6.1. Design Objectives

Traditional pull-based P2P streaming protocols are simple to implement (possibly within a few thousand lines of Python code), and are robust to peer arrivals and departures. Current-generation real-world protocols, such as PPLive [1], PPlite [2], PPStream [3], Coolstreaming [4], UUSee [5], QQLive [6], SopCast [7], TV Ants [8], and many more, have used such pull-based strategies, and have been able to serve tens of thousands of users in a session with acceptable playback quality. However, we believe that the user experience in P2P streaming sessions can be further enhanced by considering the following quality metrics that are beyond the basic playback quality:

- **Shorter initial buffering delays**: Since the initial buffering delay must be experienced by a user when switching to a new channel, a shorter delay dramatically improves the user experience.

- **Reduced streaming source bandwidth costs**: Since peer upload capacities may not be sufficient to sustain the entire streaming session for all participating peers, dedicated streaming servers provide additional supply of bandwidth. Since operational costs of these dedicated servers depend on the bandwidth consumed, we believe that it is critical to minimize server bandwidth costs. Even if monetary operational costs are not a concern, minimizing the source bandwidth usage allows sufficient unused capacity to cater to the demand spike in “flash crowd” scenarios.

- **Better resilience to extremely volatile peers**: When peers join and leave in an extremely volatile fashion, we wish to maintain smooth playback as much as possible.

- **Smother playback when bandwidth supply barely exceeds the demand**: With an increasing number of peers and a fixed number of dedicated streaming servers, the
saturation point will eventually be reached, where the bandwidth supply barely exceeds the demand. We wish to maintain smooth playback as much as possible in such challenging situations. This is especially the case in unpopular sessions, when a very small number of servers (usually just one or two) and a few hundred peers are engaged. The playback quality in current pull-based protocols is usually unsatisfactory in such cases, due to the lack of seeds and unpredictable bandwidth among peers.

6.1.2 The Essence of $R^2$: Random Push with Random Network Coding

**Random Network Coding.** Random network coding serves as the cornerstone of $R^2$, and is instrumental towards most of the advantages of $R^2$. In traditional P2P streaming protocols, the live stream to be served is divided into segments, such that they can be better exchanged among peers. In $R^2$, each segment is further divided into $n$ blocks $[b_1, b_2, \ldots, b_n]$, each $b_i$ has a fixed number of bytes $k$ (referred to as the block size). If the segment duration $t$ (for example, 4 seconds) and the streaming rate $r$ are predetermined, the block size $k$ can be computed as $\frac{rt}{n}$.

When network coding is used on a seed (a serving peer or the streaming source), we do not propose to code across different segments. This is similar to the approach taken by Chunked Codes, which only codes within a chunk. This is primarily for the purpose of reducing the number of blocks to code, leading to much reduced coding complexity of dense linear codes, as has been well established in the Chunked Codes. $R^2$ shares the same design and implementation of random network coding with Lava.

**Random Push.** When a seed encodes for a downstream peer $p$, as a critical aspect
of the algorithm design, we need to address the question: *Which segment should the seed select in which to code and send a coded block?* Trivially, segments that $p$ has already received should be excluded, and the decision would be made among the remaining segments that $p$ has not completely received so far. In a traditional pull-based protocol, such a decision is made on the downstream peer $p$, and explicit requests are made to the seed. The seed then honors the request by sending the segment. To better take advantage of random network coding in $R^2$, we instead use *random push*, in which the seed *randomly chooses* a segment whenever it is ready to produce one coded block, among all remaining segments that $p$ has not completely received. The coded block is then sent to $p$ without the need for any requests. Since all coded blocks are equally useful, all seeds of $p$ cooperatively serve the missing segments on $p$, without any explicit communication.

Now, the question is how does a seed randomize such a segment selection process for each outgoing coded block? The answer to this question constitutes an important design choice in $R^2$. Naturally, an important concern at the downstream peer is that it should expedite the downloading process of “urgent” but missing segments, *i.e.*, those missing segments that are close to their playback time. This range of urgent segments may be $\tau$ seconds after the current playback point, and is referred to as the *priority region*, as shown in Fig. 6.1. Since there are no explicit requests made by the downstream peer (no pull required), seeds should give strict priority to the segments within the priority region.

Let $A$ and $B$ be the bit-vectors representing the buffer status on the downstream peer and the local host, respectively. The simple bitwise operation $(A \oplus B) \cap B$ determines the set of segments that are available in local playback buffer, but not in the playback buffer on the downstream peer. This operation is applied to the priority and normal regions separately, as in Fig. 6.1.
In our randomized segment selection, we stipulate that a seed should randomize within the priority region using an uniform distribution, whenever segments in this region are still missing on the downstream peer. From the viewpoint of a downstream peer, as playback progresses, if a few missing segments eventually fall into the priority region, their urgency guarantees that all of its seeds will serve these segments. If the downstream peer has sufficient download bandwidth, it should be able to completely receive these missing segments before playback.

If there are no missing segments in the priority region at the downstream peer, the seed will choose missing segments from outside of the priority region. Such a randomized choice is subject to a certain probability distribution with PDF that gives preference to segments that are earlier in playback time. In our experiments (Sec. 6.3), we used a Weibull distribution with a PDF \( f(x; k, \lambda) = \frac{k}{\lambda}(\frac{x}{\lambda})^{k-1}e^{-\left(\frac{x}{\lambda}\right)^k} \), such that different shapes of the PDF may be obtained by simply tuning the shape parameter \( k \) and scale parameter \( \lambda \). An alternative distribution is acceptable as well, as long as it prefers earlier segments.

Without loss of generality, we assume that the priority region is the same as the
initial buffering delay. When a new peer joins, the priority region of its buffer is empty. During the initial buffering delay, all seeds of the new peer start to serve segments within the priority region. Akin to a “flash crowd” scenario, such a phenomenon in $R^2$ easily saturates the download bandwidth of the new peer. If the download throughput exceeds the streaming rate, $R^2$ guarantees smooth playback during the priority region. In practice, this ensures that $R^2$ does not need to employ an exceeding large initial buffering delay (in the order of minutes as in PPLive [1], for example), and can use one as small as $10 - 20$ seconds.

To summarize, as long as the average receiving rate at a peer exceeds the streaming rate over time with small variations, the priority region of the playback buffer should always be filled. The peer concurrently transmits segments in the rest of the playback buffer, where earlier segments take precedence over the later ones. The dark shade in Fig. 6.2 indicates the receiving status of each segment in the playback buffer on a typical peer.

![Figure 6.2: The playback buffer in $R^2$.](image)

### 6.1.3 Timely Feedback from Downstream Peers

One outstanding but important question from the previous discussion is: *How does the seed obtain precise knowledge of the missing segments on its downstream peers at any*
6.1. THE DESIGN OF $R^2$

**time?** In traditional pull-based protocols, a *buffer map* is exchanged among peers periodically, which is a bitmap that represents segment availability in the playback buffer. The period of such an exchange cannot be too short, as a typical playback buffer in traditional pull-based protocols usually contains hundreds of segments. We perform the following back-of-the-envelope calculation: With 480 segments, a buffer map needs 60 bytes. With dozens (if not hundreds) of neighboring peers, if we exchange buffer maps every second, it amounts to 6 KB/sec on-the-wire overhead from exchanging buffer maps alone (out of around 40 KB/sec streaming bit rate)! For this reason, most real-world protocols exchange buffer maps less frequently.

The information in a buffer map can be up to $\lambda + \epsilon$ seconds delayed, where $\lambda$ is the exchange period and $\epsilon$ is the transmission time. Even with the level of overhead in the unrealistic case of exchanging every second, a seed may still be sending segments that are no longer missing on the downstream peers. In the traditional pull-based protocol, such delayed knowledge is less of a concern. Since the seed will not send a segment until it receives an explicit request, such delayed knowledge only leads to delayed requests. In $R^2$, this delayed knowledge of missing segments is no less than catastrophic: it will lead to coded blocks, that are no longer useful, being sent to and discarded by a downstream peer. For instance, in Fig. 6.3, peers 1 and 3 are collaboratively serving coded blocks of a segment to peer 2. Before the next buffer map is sent and processed, peer 2 received 8 coded blocks of this segment. Assuming the segment has 4 original blocks, 4 out of 8 coded blocks are discarded, resulting in 50% bandwidth waste.

The design of $R^2$ stipulates that buffer maps are exchanged at a much higher frequency to reduce the bandwidth waste illustrated in Fig. 6.3. As a matter of fact, the buffer maps are no longer sent *periodically*. Instead, a downstream peer sends its buffer map
6.1. THE DESIGN OF $R^2$

Whenever the buffer status changes — when it has played back a segment, or when it has completed the downloading of a segment. Whenever possible, the buffer map is embedded in outgoing coded blocks. Otherwise, it is separately sent to the upstream peers. With such a design, $R^2$ guarantees that the delay of obtaining precise buffer maps from downstream peers is never higher than the network transmission delay on the P2P links, which is in the range between a few to a few hundred milliseconds. We further note that, as an arbitrary pair of peers will be likely to serve as seeds for each other, such explicit transmission of buffer maps may rarely be needed.

The buffer maps are also used as a signal for seeds to stop a segment transmission, once the segment has been completely received (likely with the assistance from other seeds). Since buffer maps are sent in the most timely fashion, such a “negative” signal is received as soon as the network allows. The sequence of these actions is best illustrated with an example.

**Example 1:** A segment represents 4 seconds of the playback, and each segment is further divided into 16 blocks. In Fig. 6.4, peer 2 has two upstream peers (1 and 3). We assume that all three peers have roughly synchronized playback time, indicated
by the dashed lines. The playback occurs every second, and a buffer map is sent after each playback or after a segment is completely received and decoded. Peers 1 and 3 collaboratively serve coded blocks to peer 2. When 16 linearly encoded blocks are received by peer 2, another buffer map is sent.

Figure 6.4: An example of the streaming protocol in $R^2$

Compared with Lava, $R^2$ can serve the streaming content at a faster rate, even if the serving rates from the two upstream peers fluctuate over time. If the accumulative serving rate from all upstream peers does not meet the streaming rate, peer 2 can selectively abort the current transmission and contact more upstream peers by sending additional buffer maps.

In fact, there is nothing in the design of $R^2$ that prevents a downstream peer from sending the negative signal even before it has completely received a segment, in order to prematurely stop a subset of the seeds from sending this segment, usually when the transmission of this segment is almost completed. Such a premature braking algorithm
may be designed to favor seeds with better bandwidth to complete the download, and stop those seeds with lesser inter-peer bandwidth. The design of such algorithms may be quite elaborate, gradually stopping more seeds based on estimates of the downloading process. In our experiments, we do not include this feature, since we wish to focus on the raw performance of the random push algorithm.

How does $R^2$ manage the excessive overhead of exchanging buffer maps, then? Let us revisit the example discussed earlier, in which a playback buffer has 480 segments representing 160 seconds of the playback — around 15 KB per segment with a streaming rate of 45 KB/second. $R^2$, instead, divides the buffer into 40 segments of length 4 seconds each, and further divides each 180 KB segment into 180 blocks of size 1 KB each. This leads to just 5 bytes to represent each buffer in the bit-vector format, which can be easily embedded in a 1 KB coded block with only 0.4% overhead, when required. Moreover, a segment is removed from the buffer every 4 seconds, and a segment is completely received every 4 seconds in a steady state. Hence, a peer sends at most 2 buffer maps to each neighboring peer every 4 seconds on average. It amounts to approximately 250 bytes/sec on-the-wire overhead to exchange buffer maps among dozens of peers, a significant improvement in comparison to the 6 KB/sec overhead offered by a traditional protocol. Table 6.1 summarizes this comparison.

Table 6.1: Communication overhead brought by the buffer map in both Lava and $R^2$

<table>
<thead>
<tr>
<th></th>
<th>Lava</th>
<th>$R^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block size</td>
<td>—</td>
<td>1 KB</td>
</tr>
<tr>
<td>Segment size</td>
<td>0.3 sec, 15 KB</td>
<td>4 secs, 180 KB</td>
</tr>
<tr>
<td>Buffer size</td>
<td>480 segments</td>
<td>40 segments</td>
</tr>
<tr>
<td>Size of buffer map</td>
<td>60 bytes</td>
<td>5 bytes</td>
</tr>
<tr>
<td>Overhead</td>
<td>6 KB/sec</td>
<td>250 bytes/sec</td>
</tr>
</tbody>
</table>
Finally, why is $R^2$ able to use much larger segments? In traditional pull-based protocols, we observe that a missing segment on a downstream peer can only be served by one seed at a time (with the possibility of switching to a different seed if the transmission fails due to low bandwidth or peer dynamics). With random push coupled with random network coding, a segment can be served by multiple seeds, as each seed uses its randomized selection algorithm to select a segment to send coded blocks. We refer to such a phenomenon as *perfect collaboration*, since seeds collaborate with each other without any protocol messages. Such an excellent property is due to the fundamental characteristic of dense random linear codes, in that any coded block is as good as any other, regardless of the seed that produces them. The sharp contrast between a traditional P2P streaming protocol and $R^2$ is shown in Fig. 6.5. While traditional P2P streaming protocols have smaller segments, and each segment is served by one seed, $R^2$ can afford to have larger segments (that are further divided into blocks), and each segment is served by multiple seeds.

![Diagram](a) Traditional pull-based live P2P streaming.  (b) $R^2$.

Figure 6.5: An illustrative comparison between a traditional P2P streaming protocol and $R^2$
6.1.4 Random Selection of Downstream Peers

To make sure that coded blocks from one segment are not “spread too thin” in all the peers, a seed only sends a segment to a limited number of downstream peers at any given time, subject to an upper bound. To select such limit, the seed can randomly select from all its downstream peers, or select those that have historically had the highest flow rate with the seed. The maximum number of downstream receivers should be linearly related to the upload capacity of a seed: the lower the upload capacity, the smaller number of active downstream receivers it should maintain. This design choice in $R^2$ places “emphasis” on a small number of participating peers for a particular segment, which accelerates the rate of initial propagation of a segment that has just been made available on the streaming source. For such a segment, as the number of peers who have already received it exceeds a threshold, the remaining peers will be able to download smoothly — leading to an exponential propagation behavior.

When a seed randomly chooses downstream peers for a segment, each segment should have different downstream peers for each set of randomly generated coding coefficients. This randomizes the data dissemination process since a seed serves different segments to different sets of peers. The randomized selection of both downstream peers and segments (for a particular peer) in $R^2$ is perfectly resilient to peer departures and network losses.

6.1.5 Design Objectives Revisited

Let us now revisit the original design objectives that we have outlined, and note how $R^2$ fulfills these requirements.

- **Shorter initial buffering delays:** Peers in $R^2$ enjoy shorter initial buffering delays, as smooth playback is guaranteed if sufficient seeds are used to saturate the peer
download capacity. This is due to our design of synchronized playback, as well as perfect collaboration among seeds.

- **Reduced streaming source bandwidth costs:** With network coding, every coded block being transmitted is equally useful to the participating peers. With multiple seeds serving a segment, and without any overhead incurred by explicit requests, the probability of saturating both peer upload and peer download bandwidth capacities is much higher than in traditional pull-based protocols. Both of these factors contribute to reducing the source bandwidth costs, since peers are able to serve more useful bits to one another.

- **Better resilience to peer dynamics:** In $R^2$, resilience to peer dynamics has been significantly improved. Since multiple seeds are used to serve each segment, the departure of one or a few of them does not constitute a challenge.

- **Smooth playback when bandwidth supply is tight:** Since $R^2$ utilizes bandwidth as efficiently as possible with UDP-based transmission of coded blocks under flow control, and since $R^2$ gives strict priority to urgent missing segments, playback quality will be much improved as compared to traditional pull-based protocols, especially in challenging scenarios when overall bandwidth supply barely exceeds the overall bandwidth demand in the entire session.

6.2 Implementation

Based on the Lava design, the implementation of $R^2$ is as simple as replacing the streaming protocol, as shown in Fig. 6.6. This allows us to focus on the design of $R^2$. Again, the dash lines in Fig. 6.6 illustrate the data flow in $R^2$, which is the same as in Lava.
For incoming messages:

1. The network layer receives the message into the appropriate incoming queue.

2. The engine takes the message and routes it to the algorithm (streaming protocol).

3. The algorithm employs the progressive decoder to decode the message.

4. The message is stored into the playback buffer.

To serve downstream peers:

5. For each downstream peer, the random push algorithm selects a segments from the playback buffer.

6. The algorithm employs the encoder to produce an encoded block of the selected segment.

Figure 6.6: $R^2$ in Crystal
7. The message is enqueued into the appropriate outgoing queue in the network layer via the engine.

8. The network layer sends the message to the corresponding downstream peer.

The skeleton of \( R^2 \) implementation is given in Table 6.2. All functions are invoked by the engine as the events are triggered or messages are received. The algorithm only needs to call one function of the engine: the \texttt{send()} function. Similar to the simple relaying protocol in Chapter 3, the message handler, \texttt{process()}, switches on different message types. In contrast with Lava, \( R^2 \) employs less message types (no \textit{request} or \textit{cancel}), and introduces only one additional event, \textit{playback}, during bootstrap.

Since seeds play a vital role in \( R^2 \), we improved the tracker to not only bootstrap peers but also monitor the buffering level on each peer. Periodically, peers report to the tracker with their uploading and downloading throughput, as well as current buffering levels. The tracker maintains a pool of good seeds on which the buffering level exceeds a certain threshold. For any peer that has unsatisfactory buffering level or downloading throughput, the tracker sends it a few randomly selected peers from the seed pool. This is designed to help the peers to discover better seeds in the network. Peers with good seeds will eventually become a good seed for others.

### 6.3 Performance Evaluation

The focus of our experiments is to examine the effectiveness of \( R^2 \) with respect to our design objectives. For the sake of comparison, we ran Lava, including both Vanilla and network coding, with the new tracker under the same setting as \( R^2 \). A large real-world streaming session involves tens of thousands of peers, supported by dozens of high
6.3. PERFORMANCE EVALUATION

Table 6.2: The skeleton of $R^2$ using Crystal

```
bootstrap(Msg * m)
    Contact the tracker or parse the topology file for an initial set of peers
    register a periodic timer for playback with callback playback()

process(Msg * m)
    switch m->type()
        case join:
            Add the peer who sent this message to the downstream peer list
            Send accept to the new downstream peer
        case accept:
            Receive the acknowledgment from upstream peers for the join request
        case buffermap:
            Update the buffer status of the corresponding upstream peers
        case data:
            Insert the received coded block to the playback buffer
            if the received block completes a segment
                for every upstream peer p
                    send(buffermap, p)
        case default:
            Unknown message type.
            return consumed

removePeer(Msg * m)
    Remove a peer from the downstream peer list

joinSession(Msg * m)
    Send a join message to the initial neighboring peers

leaveSession(Msg * m)
    Leave a session by disconnect all downstream and upstream peers

playback()
    Remove the segments that is due for playback from the playback buffer
    for every upstream peer p
        send(buffermap, p)
```
6.3. PERFORMANCE EVALUATION

performance servers with at least 10 MB/sec upload capacity each. It is impossible for us to emulate such a large streaming population with actual traffic using only 48 clustered servers. With network coding, each emulated peer consumes roughly 5 – 7% of the CPU; hence, we were approaching 100% CPU usage with 800 peers. In order to obtain reasonable observations from such a relatively small network, we used only one streaming source with 1 MB/sec upload capacity, and limit all other peer connections to DSL grade, with upload capacities uniformly distributed between 80 and 100 KB/sec. We set the streaming rate to 64 KB/sec, a typical rate in real-world streaming systems. We believe that such a scenario may be experienced in real-world sessions that are relatively unpopular, when very few streaming sources have been deployed. In these cases, the real-world user experience has often been quite poor as well, corroborating our observations in these sets of experiments. By emulating a streaming session using such extreme settings, our results represent the baseline performance, and we expect $R^2$ to perform better in reality, especially in popular sessions.

In all experiments, unless specified otherwise, each segment represents 4 seconds of the playback, and is divided into 128 blocks, offering a satisfactory encoding and decoding bandwidth with our implementation of random linear codes (around 4 MB/sec). Each streaming session lasts for 10 minutes. For a more challenging scenario, we set the buffer size to 32 seconds, the lag time to 30 seconds, the initial buffering delay to 16 seconds, and the priority region to 8 seconds.

To evaluate the performance of $R^2$, we evaluated several important metrics: (1) *Playback skips*: measured as the percentage of segments skipped during playback. A segment is skipped during playback if it is still not completely received at the playback time. (2) *Bandwidth redundancy*: measured as the percentage of discarded segments or blocks, due
6.3. PERFORMANCE EVALUATION

to linear dependence or obsolescence, over all received segments or blocks. (3) Buffering levels on each peer during a live session over time, measured as the percentage of received blocks or segments in the playback buffer. (4) The uplink bandwidth consumption on the dedicated streaming source. All measurements are averaged over all peers in the session.

6.3.1 Scalability

We first evaluated the scalability of $R^2$, by varying the number of peers in the live streaming session from 88 to 792 peers. In this experiment, peers join the streaming session as soon as there is sufficient bandwidth supply in the session, i.e. the bandwidth demand closely matches the bandwidth supply, especially at the beginning of the session. Fig. 6.7(a) shows that $R^2$ offers steady playback quality, with less than 0.02% playback skips, whereas Vanilla and network coding have an increasing percentage of playback skips as the number of peers increases. The reduced playback quality in larger networks is due to the following two reasons: (1) All peers join the session in the first two to three minutes. The join pattern becomes a flash-crowd scenario. As shown in Sec. 5.3, Vanilla cannot efficiently handle such situation. (2) Due to computational complexity, when the network size reaches 572 (12 peers on each server), both network coding and $R^2$ consume more than 85% of resources on each CPU, and its performance degrades significantly. Nevertheless, we argue that such a limitation is not applicable in reality, since each peer runs only one coding instance, which consumes less than 10% of the CPU.

The improvements in $R^2$ is due to its effective use of bandwidth, as shown in Fig. 6.7(b), which is brought by network coding and the use of smaller blocks rather than larger segments. For the benefit of the service provider that hosts dedicated streaming sources, $R^2$ saves almost 15% of the upload bandwidth on the streaming source, as shown in Fig. 6.8.
6.3. PERFORMANCE EVALUATION

Figure 6.7: Average playback and bandwidth status on each peer in a 64 KB/sec streaming session deployed to P2P networks of various sizes, ranging from 88 to 792.

(a) Average playback skips

(b) Average bandwidth redundancy
Subject to exactly the same scenarios, however, \( R^2 \) delivers robust and convincing performance in terms of both playback quality and bandwidth utilization.

![Graph](image)

Figure 6.8: Uplink bandwidth consumption on the dedicated streaming source during a 64 KB/sec streaming session deployed to P2P networks of various sizes, ranging from 88 to 792

### 6.3.2 Buffering Levels

As an important metric to gain insights on the playback quality, we next examined the fluctuations in average buffering levels over the course of a streaming session. Fig. 6.9 compares the average buffering levels of \( R^2 \), network coding, and Vanilla in two representative sessions, from which we drew two key observations. First, the buffering level ramps up quickly and remains stable in \( R^2 \), while Vanilla maintains a much lower level with a slight variation over time. Network coding maintains a slightly higher buffering level than Vanilla does. Second, the buffering level of \( R^2 \) increases as more peers become available, while that of Vanilla and network coding decrease as the network size increases.
The satisfactory buffering level in $R^2$ also explains the perfect playback quality (represented by nearly nonexistent playback skips) in Fig. 6.7(a). As observed, the random push algorithm and the priority region design guarantee in-time and fast delivery of each segment.

![Graph showing average buffering level during sessions and on each peer in a 64 KB/sec streaming session deployed to networks that consist of 88 and 792 peers.]

Figure 6.9: Average buffering level during the sessions and on each peer in a 64 KB/sec streaming session deployed to networks that consist of 88 and 792 peers

### 6.3.3 Initial Buffering Delays

To better illustrate the advantage of $R^2$ in effective segment transmission, we present the time that all three algorithms take to fill the priority region when a peer joins a session in Table 6.3. Although network coding improves the time from 14 seconds to 12 seconds in traditional pull-based protocol, such a time still increases with the network size. In sharp contrast, $R^2$ is able to completely fill the priority region in less than 6 seconds regardless of the number of peers in the session, i.e., the upload bandwidth on the seeds are fully
6.3. PERFORMANCE EVALUATION

utilized in transmitting encoded blocks, and very few blocks are being discarded.

Knowing that it takes less time for $R^2$ to fill the priority region, we turned our attention to the impact of the initial buffering delay and the length of the priority region. Intuitively, longer initial buffering delays should lead to better playback quality and higher buffering levels. However, Fig. 6.10 shows that the effect of different initial buffering delays in $R^2$ is not as significant as it is in traditional protocols. We also observed that both Vanilla and network coding have unacceptable playback quality when the initial buffering delay is 8 seconds, which explains why they cannot support channel surfing well.

Figure 6.10: The average playback skips and buffering level during the session when tuning the initial buffering delay, in a 64 KB/sec streaming session with 800 peers

To gain a better understanding of the priority region setting, we fixed the initial buffering delay to 24 seconds, and increased the length of the priority region from 8 seconds to 20 seconds (more than half of the playback buffer). As the priority region grows, the earlier segments in the buffer become less “urgent”, leading to lower playback
Table 6.3: The average time (in seconds) taken to fill the priority region in sessions involving different numbers of peers

<table>
<thead>
<tr>
<th>session size</th>
<th>88</th>
<th>132</th>
<th>176</th>
<th>220</th>
<th>264</th>
<th>308</th>
<th>352</th>
<th>396</th>
<th>440</th>
<th>484</th>
<th>528</th>
<th>572</th>
<th>616</th>
<th>660</th>
<th>704</th>
<th>748</th>
<th>792</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R^2$</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Network coding</td>
<td>10</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>12</td>
<td>12</td>
<td>11</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Vanilla</td>
<td>10</td>
<td>11</td>
<td>11</td>
<td>12</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>13</td>
<td>13</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>13</td>
</tr>
</tbody>
</table>
quality, as shown in Fig. 6.11. However, we noted that the length of the priority region does not materially affect the performance, since $R^2$ effectively utilizes all bandwidth to maintain high buffering levels.

![Figure 6.11: Average playback skips when tuning the length of the priority region, in a 64 KB/sec streaming session with 800 peers](image)

### 6.3.4 Effects of Random Network Coding

With respect to the segment size, we performed two experiments and vary the time represented by a segment from 2 to 8 seconds. In the first experiment, we fixed the number of blocks in a segment to 128, and increase the block size from 1 KB to 4 KB as the segment size increases. When a segment is too small, more overhead ensues when transmitting small blocks. On the other hand, when a segment becomes larger (8 seconds of playback), the priority region consists of only one segment, leading to less randomized segment selection. The result in Fig. 6.12 shows lower buffering levels during the initial
buffering delay. We observed that all of the playback skips using large 8-second segments have occurred during the first 8 seconds of playback.

Figure 6.12: The average playback skips and buffering levels during a 64 KB/sec streaming session, with a fixed number of blocks in each segment

In the second experiment, we set the block size to 2048 bytes, and increased the number of blocks within a segment from 64 to 256 as the segment size increases. With a fixed playback buffer size, the number of segments included in the playback buffer increases as the segments become smaller. Therefore, we observed slightly more randomness during segment selection, leading to slightly better playback quality, as shown in Fig. 6.13. Though larger segments offer higher buffering levels in both experiments, the priority region is not filled as quickly as during the initial buffering delay. Moreover, larger segments do not necessarily offer better quality since a missing segment may result in longer skips in seconds. In our experiments, it is ideal to have 4-second segment broken into 128 blocks.
6.3. PERFORMANCE EVALUATION

6.3.5 Tuning the Random Segment Selection Algorithm

In our previous experiment, we obtained satisfactory results using Weibull(0.5, 1) — equivalent to exponential distribution — for segment selection outside of the priority region. Furthermore, we wished to gain a better understanding on the performance impact of different probability distributions used in our segment selection algorithm. In this experiment, we selected two representative parameter settings, Weibull(2, 2) and Weibull(5, 2). Both distributions are approaching the normal distribution, but with different mean values. The former gives more preference to earlier segments than the latter does. For clarity, the PDFs of our distributions are shown in Fig. 6.3.5. As shown in Fig. 6.15(a), distributions that favor the earlier segments offer better playback quality, without consuming additional upload bandwidth on the dedicated streaming source. Different segment selection distributions do not materially affect the average buffering level,
6.3. PERFORMANCE EVALUATION

Weibull (0.5, 1)
Weibull (2, 2)
Weibull (5, 2)

Figure 6.14: The PDFs of the probability distributions used in segment selection as shown in Fig. 6.15(b), since they all yield priority towards early segments to a certain degree.

6.3.6 When Bandwidth Demand Meets the Supply

In our previous experiment, bandwidth supply outstrips demand since the peer upload bandwidth is higher than the streaming rate. It may appear that $R^2$ does not lead to improved performance when compared to Vanilla and network coding. The question naturally becomes: is this the case when the supply-demand relationship of bandwidth changes? In a session consisting of 800 peers, with all DSL-grade connections except the source, the average bandwidth supply in the session is 93 KB/sec for each peer. We ran a set of experiments with four different streaming rates: 64 KB/sec to represent the case where the supply outstrips the demand, 70 and 75 KB/sec to represent an
6.3. PERFORMANCE EVALUATION

(a) Average playback skips and uplink bandwidth consumption on the source.

(b) Average buffering level during the session.

Figure 6.15: The effects of different probability distributions in the segment selection algorithm, in a 64 KB/sec streaming session
approximate match between the supply and demand, as well as 80 KB/sec, when the demand exceeds the supply of bandwidth. When the streaming rate is 80 KB/sec, the average bandwidth demand is more than 93 KB/sec from each peer, including protocol messages and redundant traffic.

![Figure 6.16](image_url)

**Figure 6.16:** Effects of the balance between bandwidth supply and demand with 800 peers

From Fig. 6.16(a), we observed that $R^2$ significantly outperforms both Vanilla and network coding when there is a close match between supply and demand, and even when the demand exceeds the supply. We also observed that, regardless of the streaming rate, $R^2$ is able to consistently maintain a buffering level around 90%, while Vanilla and network coding are striving to maintain the buffering level above the priority region. Fig. 6.16(b) illustrates the difference in buffering levels among the three protocols in a 70 KB/sec streaming session. Despite the different buffering levels, we noticed that $R^2$ offers a more rapid increase of buffering levels during the initial buffering delay, which is
confirmed in Table 6.4. As a result, $R^2$ can fill the priority region in less than 8 seconds. In sharp contrast, Vanilla cannot fill the priority region during the entire session when the streaming rate reaches 80 KB/sec.

Table 6.4: The time (in seconds) takes to fill the priority region when tuning the streaming rate

<table>
<thead>
<tr>
<th>Streaming rate</th>
<th>64 KB/sec</th>
<th>70 KB/sec</th>
<th>75 KB/sec</th>
<th>80 KB/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R^2$</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Network coding</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>Vanilla</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>—</td>
</tr>
</tbody>
</table>

6.3.7 Effects of Peer Dynamics

![Figure 6.17: The PDFs of the peer lifetime distributions](image)

To emulate volatile peers with frequent departures, we again used the Weibull distribution — Weibull($k, 2$) — to randomly generate the lifetime of participating peers. With
6.3. PERFORMANCE EVALUATION

Figure 6.18: $R^2$ is more resilient to peer dynamics as compared to network coding and Vanilla, in a 64 KB/sec streaming session involving 800 peers.
Weibull($k$, 2), we conveniently decreased the mean peer lifetime by adjusting $k$ from 500 to 300. For clarity, the plot of each distribution is shown in Fig. 6.17. The shorter the peer lifetime is, the more volatile the session becomes. Fig. 6.18(a) indicates that the playback skips in all three protocols do not vary significantly as while tuning the peer dynamics. However, as shown in Fig. 6.18(b), the buffering level is more than 90% on more than half of the $R^2$ peers, whereas all peers in Vanilla has less than half of the buffer filled during the entire streaming session. The intuition behind this phenomena is that all coded blocks are equally useful with network coding; thus, the content in the buffers at all seeds of a peer are equally important. A peer does not need to identify the blocks or segments affected by a departing seed. All seeds of a peer are able to cooperatively serve the missing segments. The performance of Vanilla with network coding is similar to Vanilla (that is not shown for clarity of the graph): since it has not been designed to implement perfect collaboration, a segment can only be served by requests, which may be negatively affected by frequent peer departures.

6.4 Summary

This chapter presents the design and performance evaluation of $R^2$, with a sole objective of redesigning the live P2P streaming protocol to take full advantage of random network coding. $R^2$ integrates the following original contributions into a coherent design. First, it employs a randomized push algorithm without the need of making explicit requests. Second, it utilizes random network coding within each segment, making it possible for multiple seeds to cooperatively serve the same downstream peer, with no messaging overhead. Similar to Chunked Codes [83], the use of dense linear codes within a segment reduces the coding complexity to a level that can be realistically implemented and used.
Third, as all seeds give strict priority to segments close to the playback point, new peers in a session enjoy a shorter initial buffering delay. Fourth, with synchronized playback, the overlap of playback buffers on participating peers is maximized, leading to much more opportunities for peers to serve one another, and to reduced bandwidth costs on dedicated streaming servers. Finally, with larger segments and much smaller buffer maps (a few bytes), seeds in $R^2$ receive feedback from downstream peers in a timely manner. As shown in our experiments, $R^2$ enjoys clear advantages that should not be overlooked or underestimated.
Chapter 7

Concluding Remarks

We have been analyzing the performance of network coding since 2004. We first designed and implemented Crystal, an emulation framework, in preparation for empirical studies of network coding in practical network applications. We then examined existing coding libraries and implemented the most efficient network coding library. Recently, we developed several streaming protocols, including conventional ones and our proposed $R^2$ protocol, using Crystal to systematically examine the attributes of network coding in both P2P streaming applications. We recap the main contributions of this thesis and describe our plans for future research in this chapter.

7.1 Contributions

7.1.1 The P2P Prototyping and Emulation Framework

Towards the objective of conducting repeatable experiments in practical yet controllable P2P systems, we designed and developed a scalable emulation framework, Crystal [84],
which combines the advantages offered by both simulators (e.g., NS-2 [85] and MATLAB [86]) and real-world deployments (e.g., PlanetLab [65]). The primary design focus of Crystal is scalability in terms of CPU and memory usage, since we wish to emulate as many peers as possible on a limited number of servers. We paid close attention to thread management and CPU/memory efficiency. The core of Crystal, implemented in C++, consumes less than 1% of the CPU time. Hence, the scalability of Crystal is entirely determined by the traffic and computation load entailed by the particular algorithm. This is important to our research since a typical P2P network involves tens of thousands, even millions, of peers.

7.1.2 Practical Network Coding

We implemented a network coding library [87, 59], including the encoding and decoding operations. To ensure the correctness and efficiency of the implementation, we conducted analytical and empirical studies. The experimental results showed that the encoding throughput and decoding throughput far exceed today’s streaming rates. We believe that the throughput can eventually sustain HDTV quality, as the CPU speed ramps up. However, we discovered that the decoding process introduces noticeable delay after all pieces of the original content are received, which could potentially lead to skips during playback. In order to eliminate such delay, we proposed progressive decoding that allows each peer to decode as each piece of the content arrives, i.e., the decoding time is concealed into the time required to receive the content. To the best of our knowledge, this was one of the first real implementations of network coding.
7.2. **FUTURE WORK**

7.1.3 **P2P Streaming with Network Coding**

While advantages of network coding have been better understood and tested in scenarios of P2P content distribution, we are curious to know whether the same benefits apply to P2P streaming. To this end, we conducted a fair evaluation [88, 89] on the feasibility and effectiveness of random network coding in P2P streaming applications, using Crystal. In this study, we discovered that network coding provides marginal improvements when there is limited aggregated network bandwidth or frequent peer arrival and departure.

We noted that the existing streaming applications are not explicitly designed to utilize network coding. We believe a complete redesign of the algorithm is necessary to take better advantage of network coding and proposed $R^2$ [90]. Unlike the rigid and more “static” pull-and-push design in [36], the original streaming content is randomly coded into blocks (a few KB) of data that are then randomly pushed into the network. Confirmed with empirical evidence, $R^2$ offers smoother playback compared to existing systems, without the need to switch between pull and push mechanisms. In addition, $R^2$ brings considerable performance improvements in terms of delay, stability, resilience to network dynamics, as well as reduced bandwidth costs on dedicated streaming servers, all of which are beyond the basic requirement of stable streaming playback.

7.2 **Future Work**

7.2.1 **Streaming Algorithms**

In this thesis, we have focused on the seed selection and content swarming aspects in a P2P streaming system. In fact, there are other important components that contribute to the system performance. For example, a peer discovery algorithm collects aliveness
of peers in the network. This algorithm is usually employed to alleviate the load on the centralized tracking server. Moreover, mechanisms for monitoring and predicting available bandwidth and delay on an overlay link also play a vital role in selecting seeds and content swarming. In our experiments, we noted that the subtle change on the tracking server significantly improved the performance of Vanilla (with and without network coding), which is evidenced by the difference between Fig. 5.10(a) in Chapter 5 and Fig. 6.7 in Chapter 6. We also observed that changes in network settings, such as upload capacity and peer dynamics, have notable impacts to the scalability and streaming quality. These observations initiate our curiosity in studying the effects of various implementation of each component of a P2P streaming system and the influence of different network settings.

We plan to develop a representative set of implementations of each component. Each implementation will be designed as a pluggable module to allow complete flexibility in combining components. For instance, with these pluggable modules, we can build a P2P streaming system with random peer discovery, rank-based seed selection, and pull-based content swarming by simply importing the appropriate modules. We refer to such a combination as a *streaming mold*, as it can represent a class of P2P streaming systems. We will further improve our network emulator to imitate more complicated, yet realistic, network settings. The objective of this research is to conduct reality checks of each implementation. We seek to answer the question: *which streaming mold and which parameter setting offer the best performance under a particular network setting?* Ultimately, we wish to propose a new streaming system that can dynamically switch among a collection of streaming molds according to the current network situation, to meet certain performance requirements.
7.2.2 Empirical Studies under Practical Settings

Although Crystal provides a platform for system emulation with good scalability, it requires the algorithm developer to have a good understanding of the internal design of Crystal and advanced skills in system development. Moreover, in real-world applications, minimum CPU and memory footprint is not always the primary concern for every peer-to-peer application. For instance, cross-platform portability and inter-application compatibility are important for commercial applications; whereas rapid prototyping and insightful evaluation results are the essential objectives of research and development. For these reasons, we plan to design and develop Lego, a new emulation framework that facilitates prototyping and provides full flexibility in component integration.

Lego is a collection of Python-based peer-to-peer application building blocks, from basic network connections to sophisticated network algorithms, with inter-compatible APIs. Although Python, an interpreted programming language, does not offer the same execution speed as C++ does, it is known for its cross-platform portability and good code maintainability. It also offers smooth integration of components developed in other programming languages, e.g., our C++-based network coding library. Most importantly, it allows us to utilize Twisted [91], a recently developed tool for advanced network programming. The asynchronous and event-based design of Twisted allows the program to concurrently process multiple network connections without using threads, which significantly reduces the coding complexity and improves the software quality. Lego will be a full-fledge peer-to-peer application toolkit, with components ranging from network connections to well-known protocols, from network emulation to logging/debugging facilities. New components will be added as new algorithms are proposed. Development in
7.2. FUTURE WORK

Lego could be as simple as plugging-in existing components or as complicated as multi-threaded programming. A Lego-based application can be easily deployed and evaluated in both emulated environments and real Internet-based infrastructures.
Bibliography


