QUALCOMM®

Improving Content Delivery Solutions

RaptorQ™ Technical Overview

Overview of RaptorQ

Raptor is a forward error correction (FEC) technology implemented in software that provides application-layer protection against network packet loss. RaptorQ is the most flexible and powerful product in the Raptor Technology line, pioneered by Digital Fountain. The RaptorQ encoder and decoder software libraries allow streaming and file delivery services to recover data lost in transit and completely reconstruct it, without using a backchannel. Raptor Technology has a 10-year proven track record of successfully enabling solutions to achieve the high quality-of-service (QoS) content providers and end users demand.

The RaptorQ encoder and decoder software libraries can be used by applications as follows.

- A sender application generates encoded data from source data using the RaptorQ encoder.

- The encoded data is sent over a network to receiver applications.

- Some of the encoded data may be lost before it arrives at a receiver application.

- The receiver application decodes the data using the RaptorQ decoder. As long as enough encoded data arrives at the receiver application, error-free decoding is achieved, independent of the pattern of loss.

The RaptorQ encoding and decoding libraries offer the following key properties:

- Exceptionally fast encoding and decoding – linear-time encoding and decoding, enabling deployment in even the most CPU constrained environments.

- Exceptional loss recovery properties – efficiently and completely recovers the original source data from reception of any combination of encoded data essentially equal in size to the source data, independent of which encoded data arrives and which encoded data is lost.

- Flexibility to operate on a wide range of source data sizes and produce as much encoded data as necessary—like a water fountain that produces an endless supply of water, any of which can be used to completely fill a glass, RaptorQ is a fountain code (a "digital fountain") that can efficiently generate a potentially unlimited amount of encoded data from the source data, any of which is useful for the reconstruction of the data.

RaptorQ encodes and decodes a block of source data, called a source block, which is partitioned into equal-size pieces of data, called source symbols. The source block size is configured by the application that incorporates the RaptorQ software library based on the application requirements. The RaptorQ encoder generates repair symbols from the source symbols of a source block, where the repair symbols are the same size as the source symbols and the encoded symbols that can be sent consist of the combination of the source symbols and the repair symbols.

Typically, each encoded symbol is sent in an individual packet together with a 32-bit header, called the FEC Payload ID consisting of an 8-bit source block number and a 24-bit encoded symbol identifier (ESI) that allows the receiver to identify the encoded symbol carried in the packet.

The RaptorQ software library supports from 1 to 56,403 source symbols per source block. The number of repair symbols that can be generated is huge, i.e., many more than is needed by almost all applications: There can be up to 2^{24} encoded symbols per source block.

The recovery properties of the RaptorQ decoder are exceptional. If there are K source symbols in a source block, then the RaptorQ decoder can recover the source block with probability greater than:

- 99% from reception of K encoded symbols
- 99.99% from reception of K+1 encoded symbols
- 99.9999% from reception of K+2 encoded symbols

These recovery probabilities hold across the entire range of possible numbers of source symbols, source symbol sizes, and loss probabilities of sent encoded symbols, e.g., 10% loss of sent encoded symbols, 30% loss of encoded symbols, 50% loss of encoded symbols, 70% loss of encoded symbols, 90% loss of encoded symbols.

## Algorithmic Ingredients of RaptorQ Encoding and Decoding

The RaptorQ encoding and decoding algorithms are fully specified in *IETF RMT RaptorQ*. Some of the main algorithmic ingredients of RaptorQ are a well-designed combination of the following.

LT code

The LT code provides a very simple XOR-based encoding and decoding method that is extremely fast and effective. Each encoded symbol is computed as the exclusive-or (XOR) of a neighbor set of d source symbols. The value of d for an encoded symbol is chosen from a probability distribution called the degree distribution. The d neighbors of an encoded symbol are uniformly and randomly chosen from among the source symbols. This encoding process provides the fountain-like properties described above: because encoded symbols are generated independently of one another, as many encoded symbols as desired can be generated efficiently.

Figure 1 illustrates a toy example of LT encoding: $x_1, x_2, x_3, x_4, x_5, x_6$ depict source symbols and $y_1, y_2, y_3, y_4, y_5, y_6, y_7$ depict encoded symbols generated from the source symbols, where for example y_1 is of degree 3 and has neighbors x_3, x_5, x_6, whereas y_4 is of degree 1 and has neighbor x_3.

$$
\begin{aligned}
y_1 &= x_3 + x_5 + x_6 \\
y_2 &= x_1 + x_2 + x_3 \\
y_3 &= x_1 + x_4 \\
y_4 &= x_3 \\
y_5 &= x_1 + x_6 \\
y_6 &= x_3 + x_6 \\
y_7 &= x_2 + x_4 + x_5 + x_6
\end{aligned}
$$
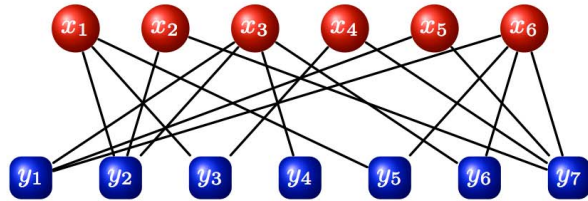
Figure 1

Decoding consists of repeating the following until all source symbols have been recovered, starting with received encoded symbols: if there is an encoded symbol with exactly one neighboring unrecovered source symbol then set the source symbol value to that of the encoded symbol (thus recovering the source symbol) and XOR the value of the source symbol into all the other encoded symbols that have that source symbol as a neighbor. This process is sometimes called belief-propagation decoding, and is a restricted version of Gaussian elimination decoding.

LT decoding applied to Figure 1 works as follows:

$x_3 = y_4$; XOR the value of x_3 into y_1, y_2, y_6;

$x_6 = y_6$; XOR the value of x_6 into y_1, y_5, y_7;

$x_5 = y_1$; XOR the value of x_5 into y_7;

$x_1 = y_5$; XOR the value of x_1 into y_2, y_3;

$x_2 = y_2$; XOR the value of x_2 into y_7;
$x_4 = y_3$.

In this example there is an unneeded encoded symbol, i.e., y_7 could also have been used to recover x_4 in the last step.

Although belief-propagation decoding is more efficient than general Gaussian elimination decoding, belief-propagation decoding can fail when Gaussian elimination decoding would succeed, and this is why the degree distribution design is crucial. The degree distribution has the following property: the probability of choosing $d = 1$ is small; for values of d between 2 and K, the probability of choosing an encoded symbol with d neighbors is approximately equal to $\dfrac{1}{d \times (d-1)}$. Thus, the average number of

neighbors of an encoded symbol is proportional to $\displaystyle\sum_{d=2}^{K} \frac{d}{d \cdot (d-1)} = \sum_{d=2}^{K} \frac{1}{d-1} \approx \ln(K)$.

This degree distribution ensures that belief-propagation decoding recovers a source block of *K* source symbols from slightly more than *K* received encoded symbols with high probability.

For both encoding and decoding, there is at most one symbol-XOR operation per encoded symbol neighbor, and thus the average number of symbol-XOR operations per generated encoded symbol is proportional to ln(*K*), and the average number of symbol-XOR operations to recover the *K* source symbols from slightly more than *K* encoded symbols is proportional to *K*·ln(*K*).

Pre-coding

Although an LT code is fast, it is not linear time. The reason for this is that the recovery of the last few source symbols using LT decoding uses very high-degree encoded symbols. The idea behind pre-coding is to relax the recovery problem: employ a light-weight pre-coding to the source symbols to generate a small fraction of additional redundant symbols. The combination of the source symbols and the redundant symbols, called the intermediate symbols, has the property that all of the intermediate symbols can be efficiently recovered once most of the intermediate symbols are known. This recovery process uses the built-in redundancy between the source symbols and redundant symbols defined by the pre-coding.

A toy example of pre-coding is illustrated in Figure 2: $x_1, x_2, x_3, x_4, x_5, x_6$ depict source symbols, z_1, z_2 depict pre-coding symbols added to the source symbols to form the intermediate symbols, $0, 0$ depict constraint symbols that indicate the relationships between the source and pre-coding symbols, i.e., they constrain the XOR sum of their neighbors to be zero, and $y_1, y_2, y_3, y_4, y_5, y_6, y_7$ depict encoded symbols generated from the intermediate source symbols. The top-right portion of Figure 2 shows the relationship between the intermediate symbols and the encoded symbols, and as can be seen source symbol x_2 is not a neighbor of any encoded symbol and cannot be directly recovered by LT decoding alone. In the bottom-right portion of Figure 2, the constraint symbols are also shown, and the source symbol x_2 is a neighbor of a constraint symbol and can be potentially recovered.
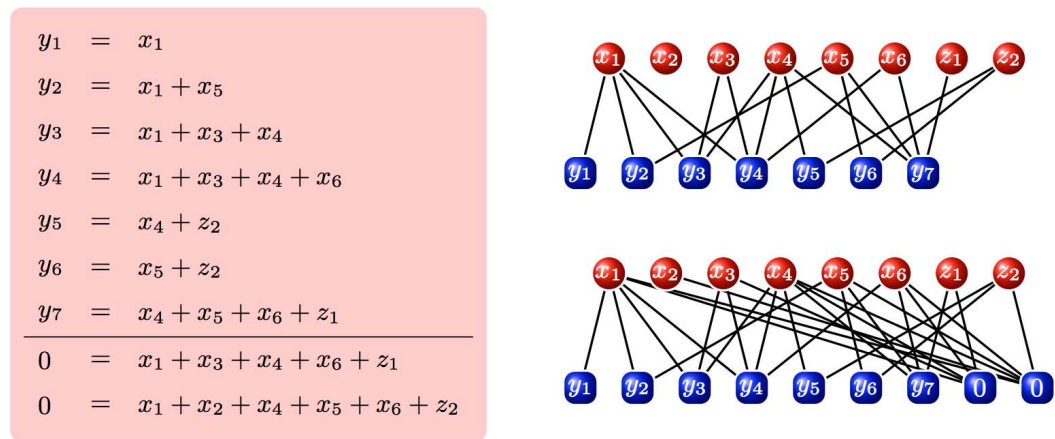

$$
\begin{aligned}
y_1 &= x_1 \\
y_2 &= x_1 + x_5 \\
y_3 &= x_1 + x_3 + x_4 \\
y_4 &= x_1 + x_3 + x_4 + x_6 \\
y_5 &= x_4 + z_2 \\
y_6 &= x_5 + z_2 \\
y_7 &= x_4 + x_5 + x_6 + z_1 \\
\hline
0 &= x_1 + x_3 + x_4 + x_6 + z_1 \\
0 &= x_1 + x_2 + x_4 + x_5 + x_6 + z_2
\end{aligned}
$$

Figure 2

A two-stage pre-coding algorithm is used for RaptorQ. The first pre-coding stage uses an LDPC code (low-density parity check code) to generate redundant symbols from the source symbols of the source block. The LDPC code generates most of the redundant symbols of the overall pre-coding, and the encoding and decoding times are linear in the source block size. The second pre-coding stage uses an HDPC code (high-density parity check code) to generate a small number of additional redundant symbols, and the HDPC code is designed to enable encoding and decoding times that are linear in the source block size.

LT encoding can be applied to the intermediate symbols to generate encoded symbols, and then LT decoding can be applied to encoded symbols that have been received to recover the intermediate symbols. The advantage is that instead of having to recover all of the intermediate symbols with LT decoding, only a large fraction of the intermediate symbols need to be recovered, and then the built-in redundancy amongst the intermediate symbols can be used to recover the remaining intermediate symbols. Because of this, very high-degree encoded symbols no longer need to be used in the degree distribution, and the average degree of the degree distribution used for LT encoding can be reduced from a number proportional to the logarithm of K to a constant.

Because of this property, the overall time to generate a block of encoded symbols (that is the combination of the source symbols and generated repair symbols) is linear in the size of the block; this is because the encoding time to generate the intermediate symbols is linear in the source block size, and because the average time for generating each repair symbol is linear in the symbol size.

Similarly, the overall time to recover a source block is linear in the size of the source block; this is because the decoding time to recover most of the intermediate symbols from the received encoded symbols using LT decoding is linear in the source block size, and because the time to decode the remaining intermediate symbols from the recovered intermediate symbols using LDPC and HDPC decoding is linear in the source block size.

Inactivation decoding

Inactivation decoding is an intertwined combination of belief-propagation decoding and Gaussian elimination decoding, and provides the low complexity of belief-propagation with the decoding guarantee of Gaussian elimination.

In a first phase the inactivation decoding process seeks out the intermediate symbols that could be solved using belief-propagation (but doesn't solve them, because the value may depend on those of other intermediate symbols that the belief propagation has ignored). Whenever belief-propagation gets stuck, an intermediate symbol is put aside (inactivated), which thereafter belief propagation will ignore so that belief-propagation can continue. In a second phase, Gaussian elimination is used on a typically dense set of equations to solve for the inactivated intermediate symbols. In a third phase, belief-propagation is used in combination with the values of the inactivated intermediate symbols to fully recover all the intermediate symbols.

A toy example of inactivation decoding is illustrated in the series of figures: Figure 3, Figure 4, Figure 5, and Figure 6. Figure 3 shows an example of a system of equations to be solved: $x_1, x_2, x_3, x_4, x_5, x_6, x_7$ depicts the unknown intermediate symbols, $y_1, y_2, y_3, y_4, y_5, y_6, y_7$ depicts the combination of known encoded and constraint symbols. Figure 4 shows the same system of equations in matrix form. In phase 1, belief-

propagation is applied to intermediate symbols in the order x_3, x_6, x_5, x_2, x_1, and during the process x_7 and x_4 are inactivated, resulting in the system of equations shown in Figure 5. Figure 6 shows the system of equations used to solve for x_7 and x_4 in phase 2. Phase 3 is similar to phase 1, except that the solved values of x_7 and x_4 are substituted into the equations instead of inactivated.
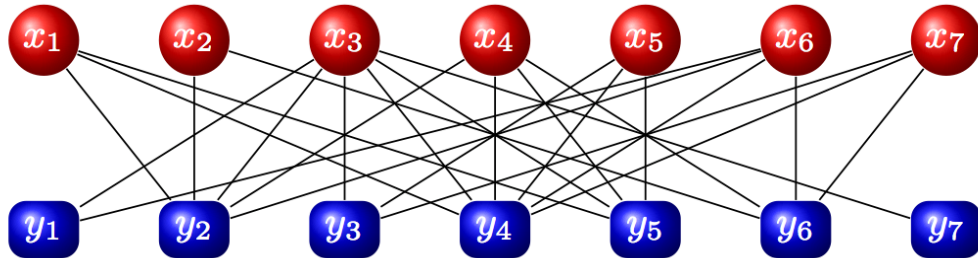

Figure 3

$$
\begin{pmatrix}
0 & 0 & 1 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0
\end{pmatrix}
\cdot
\begin{pmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
x_6 \\
x_7
\end{pmatrix}
=
\begin{pmatrix}
y_1 \\
y_2 \\
y_3 \\
y_4 \\
y_5 \\
y_6 \\
y_7
\end{pmatrix}
$$

Figure 4

$$
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0
\end{pmatrix}
\cdot
\begin{pmatrix}
x_3 \\ x_6 \\ x_5 \\ x_2 \\ x_1 \\ x_7 \\ x_4
\end{pmatrix}
=
\begin{pmatrix}
\hat{y}_7 \\ \hat{y}_1 \\ \hat{y}_3 \\ \hat{y}_6 \\ \hat{y}_4 \\ \hat{y}_2 \\ \hat{y}_5
\end{pmatrix}
$$

Figure 5

$$
\begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix}
\cdot
\begin{pmatrix}
x_7 \\ x_4
\end{pmatrix}
=
\begin{pmatrix}
\hat{y}_2 \\ \hat{y}_5
\end{pmatrix}
$$

Figure 6

Inactivation decoding is guaranteed to recover the source block if Gaussian elimination would recover the source block, and the advantage is that inactivation decoding is much faster than Gaussian elimination. The first and third phases of inactivation decoding use belief-propagation decoding, and thus their running times are linear in the source block size. The second phase involves inverting a dense $M \times M$ matrix, and then solving for the inactivated intermediate symbols using the inverse, where M is the number of inactivated intermediate symbols. The degree distribution is designed so that M is at most proportional to $\sqrt{K}$, while at the same time maximizing the probability that decoding is possible. The second phase matrix inversion incurs a number of bit operations proportional to M^3, but this is dwarfed by the number of symbol operations to solve for the inactivated intermediate symbols, which is proportional to $M^2 = K$. Thus, the overall running time of inactivation decoding is linear in the source block size.

Larger finite fields

For all the constructions described above, all of the symbol operations are XOR operations, i.e., operations over the Galois field GF(2). There is a fundamental limitation on the recovery properties of any such code: the best that any such code can achieve is recovery from reception of $K + h$ encoded symbols with probability approximately equal to $1 - \dfrac{1}{2^{h+1}}$. A clever combination of the constructions above essentially achieves this bound, but in many practical situations a better recovery guarantee is desirable.

The way to overcome this limitation is to use operations over larger finite fields, where for example a code using symbol operations over GF(256) instead of over GF(2) has the potential of achieving recovery from reception of $K + h$ encoded symbols with probability approximately $1 - \dfrac{1}{256^{h+1}}$. Possible recovery properties are shown in Figure 7, for different possible finite fields GF(q). Each different q-value line shows the decode failure probability that could possibly be achieved using GF(q) as a function of the overhead h.
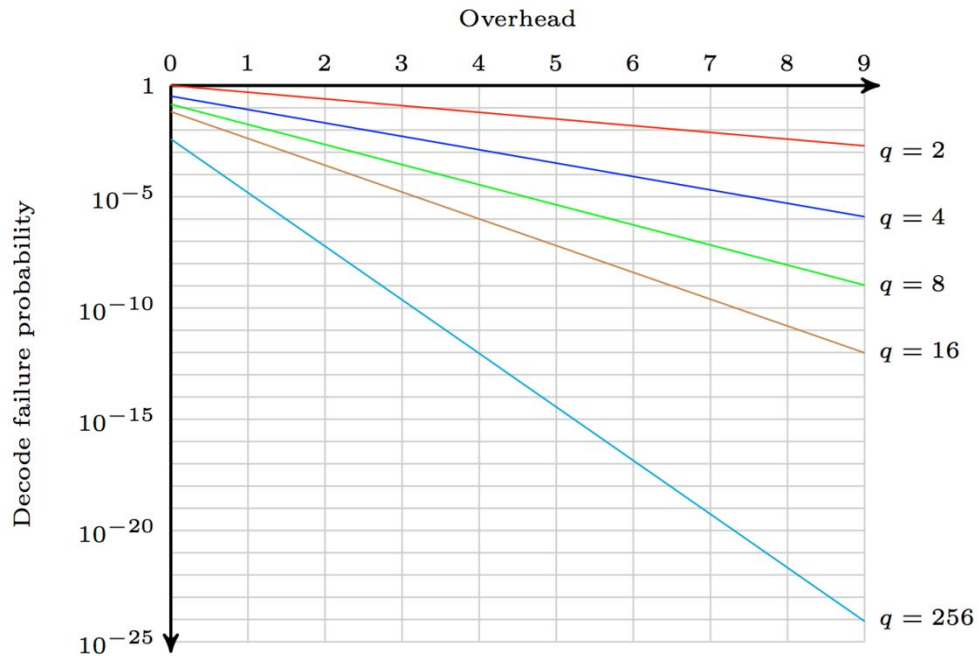
Figure 7

The disadvantage of symbol operations over larger finite fields is that they are much more computationally expensive than simple XOR operations. The key then is to use larger finite fields, but only a little bit, i.e., most of the symbol operations should be XOR operations, and only a tiny minority should be symbol operations over a larger finite field. Using larger finite fields in this way provides the low-complexity of XOR-based symbol operations with the decoding guarantee of larger finite fields.

The HDPC symbols are computed using symbols in GF(256) for RaptorQ, and the remainder of the symbol operations use GF(2), i.e., simple XOR operations. The GF(256) parts of the matrix are kept isolated during encoding and decoding, so that the vast majority of the symbol operations are over GF(2), and only a small minority are over GF(256).

Permanent inactivation

Permanent inactivation is an interesting extension of the LT code and of inactivation decoding that dramatically improves recovery properties while still maintaining linear time encoding and decoding.

Permanent inactivation for RaptorQ works as follows. Approximately $\sqrt{K}$ of the intermediate symbols are declared to be permanently inactive, and these are called the PI symbols, and the remaining majority of the intermediate symbols are called the LT symbols. The PI symbols and LT symbols are treated differently in the encoding and decoding algorithms.

In the encoding algorithm, an encoded symbol is computed as the XOR of two temporary symbols, where LT encoding is applied to the LT symbols to generate one temporary symbol, and where PI encoding is applied to the PI symbols to generate the other temporary symbol. The PI encoding process is a simple version of the LT encoding process, where two or three of the PI symbols are chosen randomly and XORed together.

A toy example of permanent inactivation encoding is illustrated in Figure 8: the intermediate symbols are partitioned into the LT symbols $x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9$ and the PI symbols y_1, y_2, y_3, y_4, y_5. For each encoded symbol z, a set of neighbors from the LT symbols is chosen according to a degree distribution Ω, a set of neighbors from the PI symbols is chosen according to a degree distribution Π, and z is the XOR of all of these neighbors.
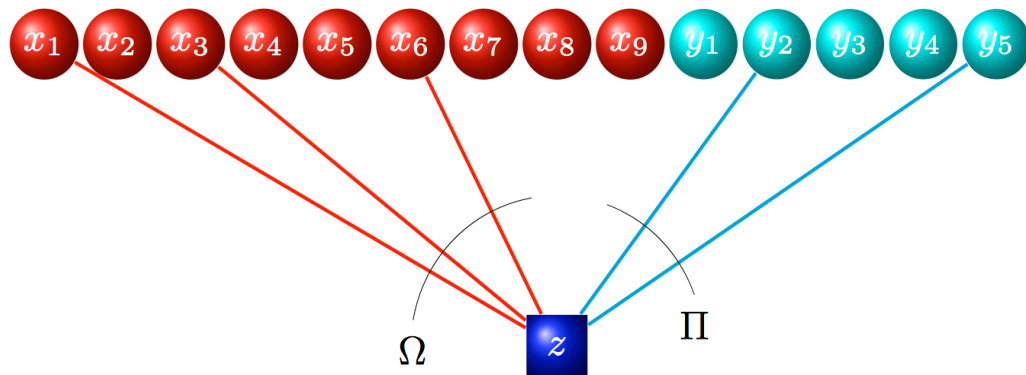

Figure 8

In the decoding algorithm, the PI symbols are inactivated at the start, and then inactivation decoding is applied as described previously, but operating only on the LT symbols in the first and third phases, and the PI symbols and any other intermediate symbols inactivated in the first phase are recovered in the second phase. When K encoded symbols are used to decode the intermediate symbols, because there are approximately $\sqrt{K}$ more encoded symbols than the number of LT symbols, the additional number of inactivations is minimized, and the probability that the encoded symbols successfully recover the LT symbols is very high.

Systematic construction

For many practical reasons, systematic codes are preferable, i.e., codes where the source symbols are amongst the encoded symbols and the recovery properties of the code do not depend on which fraction of the received symbols corresponds to source symbols and which fraction corresponds to repair symbols. All of the components of the RaptorQ code described up to this point have been non-systematic, i.e., the original source symbols are not among the encoded symbols. It is not very difficult to show that with any of these constructions, the naïve idea of simply adding the source symbols to the encoded symbols and sending the source symbols does not have the desired recovery properties. In particular, reception of some of the source symbols does not always help in combination with reception of some of the other encoded symbols to recover the remaining source symbols, and many more than K received symbols may be needed to recover the remaining source symbols.

The systematic construction provides a simple but counter-intuitive way to use a non-systematic code R to construct a systematic code S: Suppose encoder R, based on a source block with K source symbols, produces an intermediate block $Z = z_1, z_2, ..., z_n$, and from Z generates a sequence of encoded symbols $\varphi_1, \varphi_2, ..., \varphi_K, \varphi_{K+1}, \varphi_{K+2}, ...$. Let $x_1, x_2, ..., x_K$ be the values of the source symbols for S. Encoder S consists of:

1. Setting $\varphi_1 = x_1, \ \varphi_2 = x_2, ..., \varphi_K = x_K$

2. Using decoder R to recover Z from $\varphi_1, \varphi_2, ..., \varphi_K$

3. Using encoder R to produce the repair symbols $\varphi_{K+1}, \varphi_{K+2}, ...$ from Z

Then, the encoded symbols for S are the source symbols $x_1, x_2, ..., x_K$ and the repair symbols $\varphi_{K+1}, \varphi_{K+2}, ...$.

In step 2, Z is constructed so that the first K encoded symbols generated by encoder R from Z are the source symbols $x_1, x_2, ..., x_K$ for S, and there is no difference between how encoder R generates the first K encoded symbols and subsequent encoded symbols from Z. From a recovery perspective, there is no differentiation between source symbols and repair symbols: as they are all just encoded symbols generated by encoder R.

The decoding algorithm for the systematic code is symmetric to the encoding algorithm. From received encoded symbols (which can be a mixture of the source symbols and the repair symbols generated by encoder S), decoder R is used to generate the intermediate symbols Z, and then encoder R generates from Z those encoded symbols among the first K that have not been received.

A toy example of the systematic code construction is illustrated in Figure 9: $Z = z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8$ depict the intermediate symbols of code R, s_1, s_2 depict constraint symbols that indicate the relationships between the intermediate symbols of code R, $x_1, x_2, x_3, x_4, x_5, x_6$ depict source symbols of code S placed in the positions of the first six encoded symbols of code R. Decoder R is used to recover Z from $x_1, x_2, x_3, x_4, x_5, x_6$ using the sequence of operations depicted in the top-left of Figure 9. The encoder R is used to generate repair symbols y_1, y_2, y_3 of code S from Z using the sequence of operations depicted in the bottom-left of Figure 9.
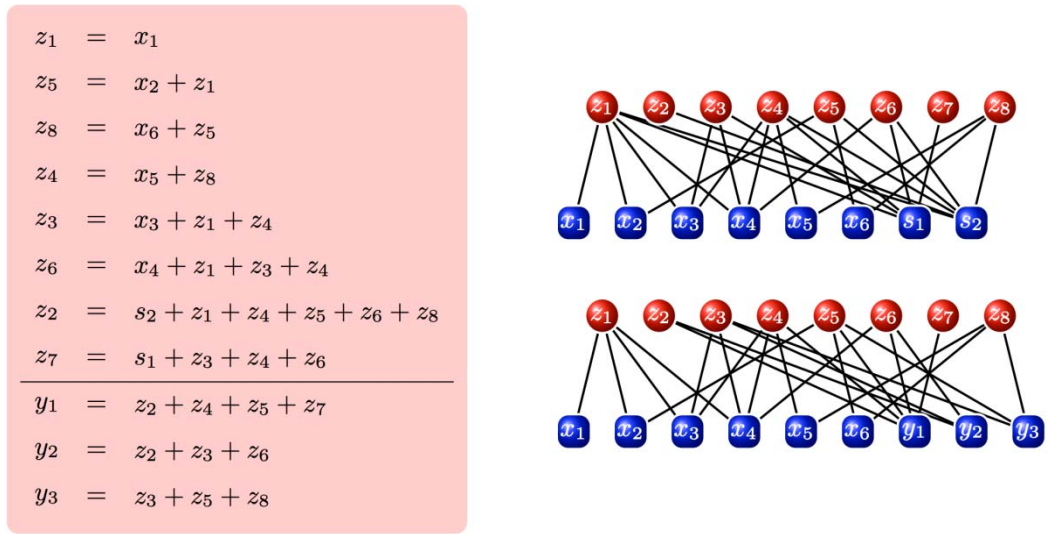
$$z_1 = x_1$$
$$z_5 = x_2 + z_1$$
$$z_8 = x_6 + z_5$$
$$z_4 = x_5 + z_8$$
$$z_3 = x_3 + z_1 + z_4$$
$$z_6 = x_4 + z_1 + z_3 + z_4$$
$$z_2 = s_2 + z_1 + z_4 + z_5 + z_6 + z_8$$
$$z_7 = s_1 + z_3 + z_4 + z_6$$
$$y_1 = z_2 + z_4 + z_5 + z_7$$
$$y_2 = z_2 + z_3 + z_6$$
$$y_3 = z_3 + z_5 + z_8$$

Figure 9

Company Background

Qualcomm develops and licenses advanced FEC technology to enhance the quality of content delivery data networks. Qualcomm's patented RaptorQ improves streaming media quality, ensures timely delivery of data, and enables the creation and development of new communications services. DF Raptor FEC technology is used today in a variety of consumer, military, and enterprise devices and applications, supporting both wired and wireless telecommunications networks.

More Information

For more information, please visit www.qualcomm.com/raptor or contact raptor-info@qualcomm.com.