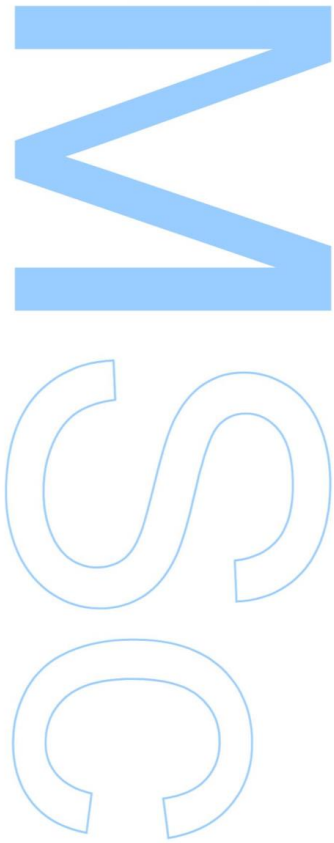
Técnicas de codificação para correção de erros em armazenamento de dados

André Filipe Viana Vila Verde
Mestrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciências de Computadores
2020

Orientador
Sérgio Armindo Lopes Crisóstomo, Professor Auxiliar
Faculdade de Ciências da Universidade do Porto
Instituto de Telecomunicações



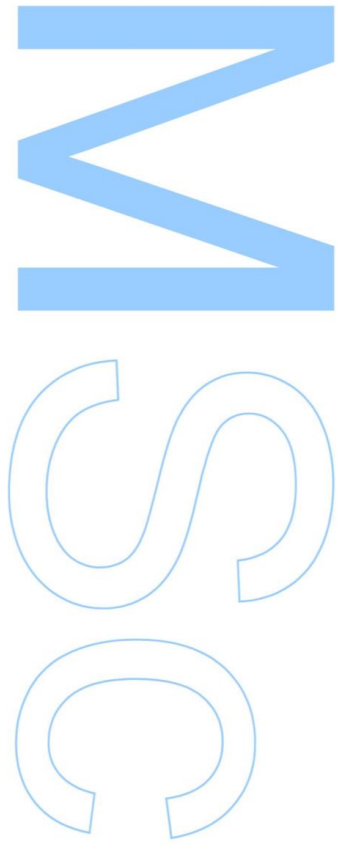
U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____



Dedicado a todas as pessoas que estiveram comigo nestes anos

Resumo

Em sistemas de armazenamento existe a necessidade de lidar com grandes quantidades de dados. Se um disco de um nó tiver dados corrompidos, este nó teria de pedir uma cópia dos dados a outro nó do sistema o que iria causar uma retransmissão de um bloco de dados. Assim surge a necessidade de desenvolver técnicas baseadas em *network coding* que, a partir, dos dados corrompidos num determinado disco sejam capazes de recuperar a informação correta sem ser necessário pedir retransmissão de dados de outros nós da rede.

O objetivo deste trabalho foi implementar as técnicas Packetized Rateless Algebraic Consistency (PRAC) e Data Aware Packetized Rateless Algebraic Consistency (DAPRAC) recorrendo a uma biblioteca de *network coding*. Após isto foi efectuada uma comparação de performance entre as duas técnicas e também entre o método de descodificação base. Estes testes de comparação permitem comparar o tempo de descodificação dos dois algoritmos baseado na probabilidade de erro em cada posição do pacote e também baseado no tamanho da geração. Outro teste efetuado foi a percentagem de sucesso de descodificação e o número de pacotes adicionais enviados para o descodificador. Destes testes obtemos as vantagens e desvantagens relativas das duas técnicas. Estes algoritmos, partindo dos dados corrompidos vão permitir recuperar os dados originais sem ser necessário pedir uma retransmissão de dados a outros nós do sistema. Usam técnicas de codificação em rede, principalmente o Random Linear Network Coding (RLNC), que permite que os dados sejam algebricamente combinados recorrendo a um conjunto de coeficientes aleatórios dando origem aos dados codificados.

O DAPRAC vem melhorar o PRAC a nível de performance. O PRAC apresenta um elevado tempo de correção dos erros quando o número de erros aumenta. O DAPRAC não só vem melhorar estes tempos, como também permite corrigir um maior número de erros em relação ao PRAC. No entanto o DAPRAC tem o problema que o tempo de correção aumentar demasiado com um aumento do tamanho da geração. Mesmo assim se o objetivo for codificar um tamanho de geração pequeno, o DAPRAC continua a ser a melhor escolha em relação ao PRAC como foi demonstrado neste trabalho.

Abstract

In storage distributed systems we have need to deal with a high data volumes. When a disk in a node can not read data because it has errors, this node would have to request data retransmission from other nodes in the system. This process would cause an overhead on the system. Therefore, there is a necessity to implement new algorithms that are able to recover the original data from corrupted data in the node, without the need of retransmissions of data from others nodes.

The goal of this work was to implement the Packetized Rateless Algebraic Consistency (PRAC) and Data Aware Packetized Rateless Algebraic Consistency (DAPRAC) techniques using a network coding library. After that we compared the performance of these techniques with the default decoding method used by the library. These performance tests allowed us to compare the decoding time of these two algorithms for the same error probability in each position of a coded packet and based on the generation size. Another test that was performed was the probability of successful decoding and the number of additional packets that had to be sent to the decoder. From these tests we get the advantages and desadvantages of these two techniques. These algorithms, starting from corrupted data, allows the recovery of the original data without having to request the retransmission of the corrupted data from other nodes in the system. These algorithms resort to network coding techniques, mainly Random Linear Network Coding (RLNC), that allows data to be algebraically combined with a set of random coefficients to produce the coded data.

The DAPRAC algorithm improves the decoding time. The PRAC algorithm has a huge correction time compared to DAPRAC and, when the number of errors in the packets is more than 4, the PRAC technique becomes impracticable. The DAPRAC not only improves the correction time but also can correct a larger number of errors. However, the correction time of DAPRAC increases a lot with the number of packets. Even so, if the goal is to send a reduced number of packets to the receiver, DAPRAC is the best choice in relation to PRAC, as we have shown in this work.

Agradecimentos

Queria agradecer ao meu orientador Sérgio Crisóstomo por me aceitar a realizar este projeto. Obrigado pelo conhecimento que partilhou comigo, os concelhos e a ajuda que foram úteis nos momentos mais complicados no desenvolvimento deste trabalho. Gostei da forma como queria as informações claras e concisas, tanto na implementação como na escrita.

Agradecer ao meus pais pelo apoio e todo as ajudas de custo e suporte que foram necessárias nestes anos de curso. A toda a família que me incentivou e se preocupou com que tudo estivesse bem.

Aos meus amigos e colegas, o meu obrigado, pelas brincadeiras, os bons momentos que passamos juntos e todo o conhecimento que partilhamos ao longo destes anos.

Uma palavra também para a Universidade do Porto na qual eu tive o prazer de fazer parte, pelas excelentes condições de infraestrutura, recursos, todo o staff e todos que participaram direta ou indiretamente nesta longa caminhada.

Aos docentes das unidades curriculares da qual eu participei, principalmente o professor Pedro Ribeiro, Pedro Brandão, Rui Prior e Sérgio Crisóstomo, pois estão relacionados às unidades curriculares da qual eu tenho mais interesse e mais me identifiquei, mas sem tirar o mérito a todos os docentes da Faculdade pois fazem um grande trabalho e tem todos o meu respeito.

Entrega o teu caminho ao Senhor, confia nele e ele tudo fará

Salmos 37:5

Conteúdo

Lista de Figuras

Lista de Blocos de Código

Lista de Equações

Acrónimos

ACR Algebraic Consistency Rule

API Application Programming Interface

CDN Content Delivery Network

CPU Central Process Unity

CRC Cyclic Redundancy Check

DAPRAC Data Aware Packetized Rateless Algebraic Consistency

ECC Error Correction Code

GF Galois Field

IoT Internet of Things

LNC Linear Network Coding

MDS Maximum Distance Separable

NAS Network Attached Storage

NORM Nack Oriented Reliable Multicast

OTT Over the Top

P2P Peer to Peer

PRAC Packetized Rateless Algebraic Consistency

QoE Quality of Experience

RAID Redundant Array of Independent Disks

RAM Random Access Memory

RLNC Random Linear Network Coding

RS Reed Solomon

RTT Round Trip Time

SNR Signal to Noise Ratio

S-PRAC Segmented Packetized Rateless Algebraic Consistency

TCP Transmission Control Protocol

V2V Vehicle to Vehicle

Capítulo 1

Introdução

Sistemas de armazenamento são constituídos por diversos nós distribuídos na rede, onde cada nó pode conter um ou mais discos que permitem guardar blocos de dados. Quando um disco da rede falha, porque os dados ficaram corrompidos ou foram apagados, esses dados ficam inacessíveis e precisam de ser recuperados. Assim os sistemas de armazenamento introduzem redundância para melhorar a fiabilidade e a tolerância a falhas caso um nó da rede falhe. A redundância pode ser feita replicando os dados pelos vários nós ou usando técnicas de erasure coding. A técnica de replicação de dados pelos vários nós da rede pode ser ineficiente, pois com o aumento do volume de dados, esta técnica vai implicar mover grandes quantidades de dados entre nós da rede. Recentemente, *network coding* tem vindo a substituir a replicação pois diminui a sobrecarga de armazenamento e apresenta uma maior eficiência na recuperação dos dados.

Com um rápido aumento do volume de dados digitais, desde dispositivos móveis até data centers de larga escala, é importante que estes sistemas consigam mitigar a ocorrência de erros e o impacto que esses erros possam causar no sistema. Surge assim a necessidade de criar novas técnicas de codificação que forneçam fiabilidade e tolerância a falhas nos sistemas de armazenamento, minimizando assim o tráfego de dados entre nós da rede. O objetivo deste trabalho é implementar o Packetized Rateless Algebraic Consistency (PRAC) e Data Aware Packetized Rateless Algebraic Consistency (DAPRAC) que são técnicas de codificação em *network coding* e fazer uma comparação de performance, baseado em várias métricas, entre estas duas técnicas implementadas. Baseado nestas métricas será possível descrever as vantagens e desvantagens de cada técnica e se realmente elas podem resolver o problema proposto.

Assim, um disco de um determinado nó do sistema sempre que encontrar dados corrompidos será capaz de detetar e corrigir esses erros a partir dos dados que tem disponível sem ser necessário uma retransmissão por parte de outro nó do sistema. Com isto será possível diminuir a sobrecarga no sistema causada pela retransmissão dos dados entre nós da rede, apenas sendo necessário colocar inicialmente uma certa quantidade de dados redundantes no disco, evitando uma maior retransmissão dos blocos de dados. Para criar estas técnicas vamos usar uma biblioteca de *network coding* chamada KODO[4], que apresenta um largo conjunto de algoritmos de *network coding*, entre eles o Random Linear Network Coding (RLNC). Esta biblioteca garante elevado

desempenho e flexibilidade podendo ser assim usada a gosto de cada utilizador. Outra biblioteca a ser usada é a Cryptopp [2], que vai permitir calcular códigos Cyclic Redundancy Check (CRC) sobre pacotes de dados que permitirá saber se os pacotes enviados contém ou não erros.

1.1 Organização

- **Capitulo 2:** este capitulo incide sobre o estado da arte e sobre as tecnologias envolvendo *network coding* e as técnicas de codificação em sistemas de armazenamento. Aqui falamos sobre *Linear Network Coding* e *Random Linear Network Coding* e também sobre os tipos de códigos usados em armazenamento.
- **Capitulo 3:** funcionamento das técnicas de codificação para correção de erros que vão ser apresentados neste trabalho, PRAC e DAPRAC, e para trabalho futuro o Segmented Packetized Rateless Algebraic Consistency (S-PRAC). Como funciona a fase de codificação dos dados, a fase de descodificação, como são detetados os erros e por fim como esses erros são corrigidos, se possível.
- **Capitulo 4:** explora um pouco mais as duas bibliotecas de C++ que foram usadas e porque elas foram escolhidas.De seguida explica as etapas importantes do código dos dois algoritmos. Quais foram as estruturas de dados usadas, quais são as funções principais e como essas funções se relacionam. Este capitulo é mais detalhado a nível de explicação de código e os requisitos a serem instalados. Por fim explica como usar o programa e quais os requisitos e a forma de instalação.
- **Capitulo 5:** fala sobre a comparação de resultados entre os dois algoritmos e o descodificador RLNC normal. Foram feitos 3 testes usando variáveis diferentes. O tempo de descodificação em relação à probabilidade de erro em cada byte dos pacotes, o tempo de descodificação em relação ao número de pacotes enviados e por fim a percentagem de sucesso em relação ao número de pacotes enviados adicionalmente.
- **Capitulo 6:** conclusão do trabalho, explicando o porque de o algoritmo DAPRAC ser mais eficiente em relação ao PRAC. É apresentado também um trabalho futuro de um novo algoritmo que vem colmatar as desvantagens do DAPRAC.

Capítulo 2

Técnicas para Detecção e Correção de Erros

Neste capítulo são apresentadas as técnicas de deteção e correção de erros bem como as diversas formas de codificar dados em sistemas de armazenamento. Na 2.1 será apresentada o que é *error detection coding* seguida de *erasure coding* na 2.2. Na 2.3 é explicada a diferença entre *Linear Network Coding* e *Random Linear Network Coding*. Por fim na 2.4 é apresentada a técnica de *storage coding* e as suas tecnologias envolventes.

2.1 Error Detection Coding

Todos os serviços de comunicação sofrem algum tipo de interferência ou ruído quando a sua informação é enviada de um emissor para um recetor. Isto porque essa informação é transmitida ao longo de um canal de comunicação em que existe um certo nível de ruído que pode alterar o conteúdo da mensagem.

Um dos principais fatores que introduzem erros na mensagem é o ruído. O ruído é descrito como uma flutuação aleatória no sinal. Assumindo que este ruído é causado de forma aleatória e natural, é necessário criar uma forma de limitar ou evitar que os dados não seja corrompidos. Um sugestão para este problema foi sugerida por *Shannon* em 1948, em que a informação que passa num canal de comunicação possa conter um número baixo de erros desde que não exceda a capacidade do canal. O teorema de *Shannon* é expresso na seguinte formula: $C = B*log_2(1+S/N)$, onde B corresponde à largura de banda do canal, S/N é o Signal to Noise Ratio (SNR) no recetor e C a capacidade do canal expresso em bits por segundo. Esta foi uma descoberta notável no ramo das comunicações que serviu de base para a criação de códigos de deteção e correção de erros [22].

O utilizador deseja que os erros da mensagem sejam reduzidos a um nível muito baixo de forma a que a informação a receber seja útil. Assim foram desenvolvidas técnicas de controlo de erros de forma a detetar a presença de erros quando a informação chega ao utilizador. A ideia

principal é adicionar redundância à mensagem de forma a permitir detetar se ocorreu um erro durante o processo de transmissão da mensagem.

2.1.1 Simple parity check

A *simple parity check* é um mecanismo simples de deteção de erros. Nesta técnica, um bit de paridade é adicionado no final da mensagem. Se o número de bits 1 for ímpar, é adicionado o bit de paridade 1. Caso contrario, caso o número de bits 1 for par é adicionado o bit de paridade 0. O recetor calcula o bit de paridade da mensagem recebida e compara com o bit de paridade recebido. A desvantagem desta técnica é que só permite detetar padrões de erro com um número ímpar de troca de bits[7].

2.1.2 Checksum

Nesta técnica, os dados são divididos em k segmentos de m bits. O emissor faz a soma desses segmentos usando a aritmética de complemento para 1. O resultado final da soma é complementado para formar o *checksum* e adicionado aos segmentos para enviar para o recetor. O recetor soma todos os segmentos de dados recebidos usando a aritmética de complemento para 1, para obter a soma. Se todos os bits do complemento da soma forem 0 os dados são aceites, caso contrário são rejeitados [7].

2.1.3 Cyclic Redundancy Check (CRC)

O CRC é um código de deteção de erros que permite usado em redes de comunicação e sistemas de armazenamento que permite detetar alterações nos dados. A técnica CRC envolve uma divisão binária dos bits dos dados através de um divisor polinomial. Este divisor polinomial já está predeterminado pelo emissor e recetor antes de começar o processo de transmissão dos dados. O emissor divide os dados pelo divisor polinomial num processo de divisão binária. O resto resultante da divisão, chamado CRC, é adicionado aos dados a enviar de forma a que o resultado enviado seja exatamente divisível pelo polinómio. O recetor divide o resultado recebido pelo mesmo polinómio. Se o resultado da divisão for zero, então os dados são interpretados como corretos, caso contrário serão rejeitados [7].

2.2 Erasure Coding

Erasure coding transforma um conjunto de dados de tamanho k em um conjunto de dados maior de tamanho n de modo a que seja possível reconstruir os dados originais com qualquer subconjunto de n blocos dos dados codificados . O objetivo do *erasure coding* é permitir que os dados corrompidos em algum disco sejam reconstruídos usando informação desses dados que

está armazenada em algum outro local do sistema de armazenamento. A principal desvantagem dos erasure codes é que podem consumir muito Central Process Unity (CPU) pois os dados redundantes precisam de ser processados, o que se pode traduzir numa maior latência [1].

Vamos assumir que o nosso sistema é composto por n discos. Vamos particionar em k discos para dados originais e $m = n - k$ para dados codificados. Os dados codificados serão usados para recuperar os dados originais em caso de algum disco falhar. Na fase de codificação os k discos são usados para calcular os dados codificados que vão estar nos m discos. Quando até m discos falham, o conteúdo pode ser recuperado dos discos restantes. Este processo de falha chama-se *erasure* e ocorre quando o conteúdo de um disco fica corrompido e não pode ser lido. O conteúdo guardado nos m discos correspondem a combinações lineares derivadas do conteúdo contido nos k discos. Este processo é semelhante ao processo que ocorre em *network coding* pois os pacotes codificados correspondem a combinações lineares dos pacotes originais e em caso de perdas é possível recuperar a informação original. [25].

Erasure coding pode ser útil para sistemas que usam grandes quantidades de dados e precisam de tolerar falhas como sistemas de armazenamento em *array*, *data grids*, aplicações de armazenamento distribuída e armazenamento de ficheiros em disco. Muitos produtos comerciais de fornecimento de serviços de *cloud* também já usam erasure coding [3].

2.3 Network coding

Network coding é uma nova técnica que permite melhorar o *throughput* e a aumentar o grau de robustez da rede. Nas rede comuns, quando uma fonte envia um conjunto de dados, esses dados podem passar por um conjunto de nós intermédios até chegar do nó destino. Os nós intermédios apenas retransmitem esses pacotes para os nós seguintes. A retransmissão separada destes pacotes é pouco eficiente, como veremos seguidamente. Utilizando *network coding*, os nós em vez de retransmitirem os pacotes originais, criam e transmitem combinações lineares dos diferentes pacotes que receberam [12].

Um bom exemplo para ilustrar as vantagens de usar *network coding* é o exemplo de uma rede *butterfly* [23]. Na 2.1 temos um exemplo de *bottleneck* onde o emissor s pretende transmitir pacotes $p1$ e $p2$ para os destinos $d1$ e $d2$. Assumindo a capacidade das ligações igual uma forma de enviar seria ilustrado na 2.1a). Aqui existe um *bottleneck* entre $r3$ e $r4$. Se $r3$ transmitir o pacote $p1$, o nó $d1$ não vai receber o pacote $p2$. Por outro lado se $r3$ transmitir $p2$ então o nó $d2$ só vai receber o $p2$. Na 2.1b) o nó $r3$ aplica *network coding* e envia uma combinação de $p1$ e $p2$. Assim o nó $d1$ pode subtrair ao pacote $p3$ o pacote $p1$ para obter $p2$. Do outro lado o nó $d2$ pode subtrair ao pacote $p3$ o pacote $p2$ para obter $p1$. A técnica de *network coding* permite enviar mais do que um pacote em simultâneo aumentando a capacidade de envio da rede.

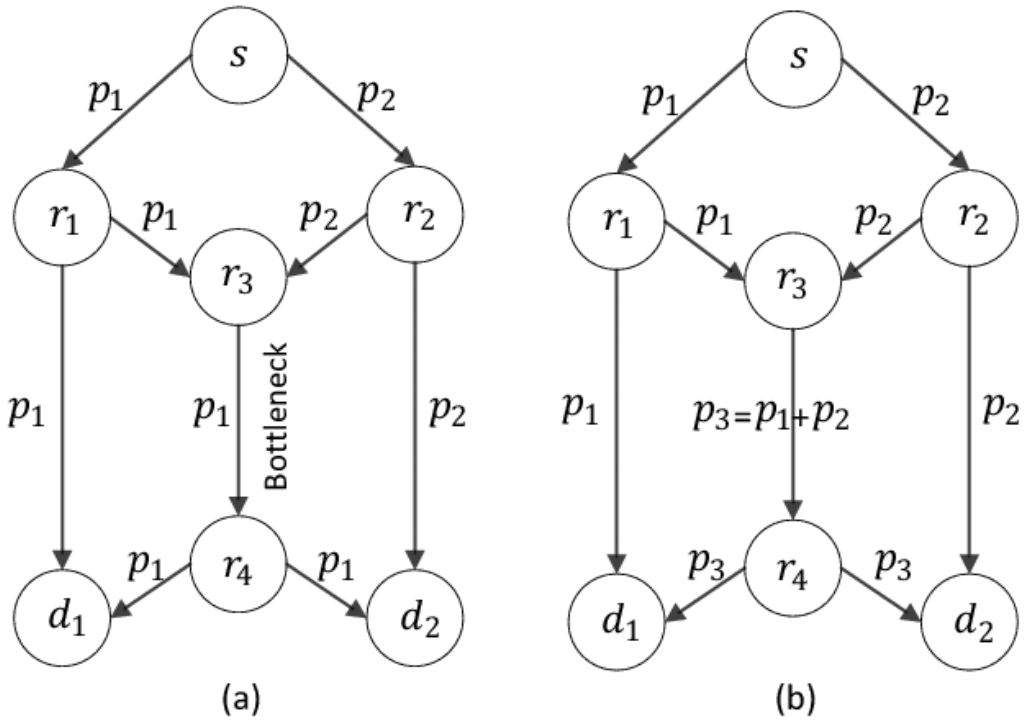


Figura 2.1: Butterfly exemplo [23]

2.3.1 Linear Network Coding

Linear Network Coding (LNC) foi a primeira implementação prática de *network coding*. Nesta técnica, na fase de codificação, um conjunto de pacotes originais são linearmente combinados de forma a criar os pacotes codificados. Os pacotes codificados vão ter o mesmo tamanho e conter informação relevante dos pacotes originais. No processo de descodificação o recetor tem de receber, pelo menos, um número de pacotes igual ao número de pacotes originais e também os coeficientes usados pelo emissor na altura da codificação/recodificação, para conseguir descodificar corretamente os dados [14].

2.3.2 Random Linear Network Coding

Random Linear Network Coding é uma variante de *network coding* que se baseia na combinação linear de pacotes recorrendo a coeficientes selecionados aleatoriamente. Estas combinações permitem usar álgebra para combinar os dados de múltiplos pacotes em apenas um pacote. Este método de codificação difere do *linear network coding* nos coeficientes de codificação. Aqui estes coeficientes são selecionados aleatoriamente de um corpo finito. Isto permite que os pacotes possam ser recodificados sem serem primeiro descodificados [18]. A figura 2.2 ilustra a fase onde 3 pacotes são codificados. Primeiro é gerado um coeficiente aleatório para cada pacote não codificado. O emissor multiplica os coeficientes pelos pacotes originais em cada posição em todos os pacotes criados de forma a obter o pacote final. Cada símbolo do pacote codificado é uma

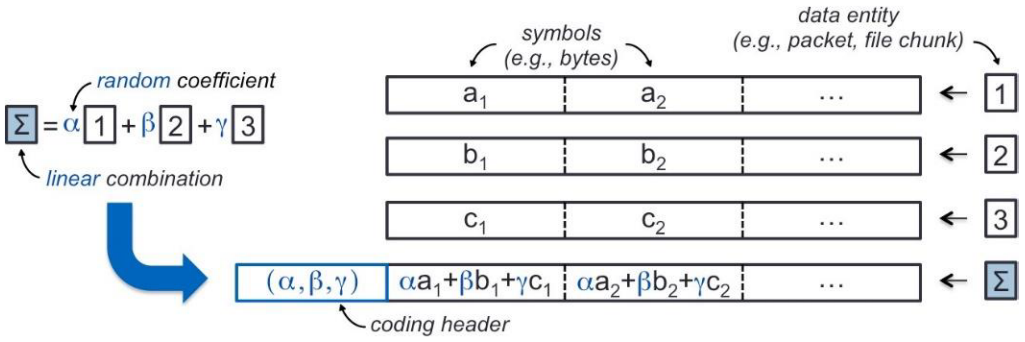


Figura 2.2: Método de codificação usando Random Linear Network Coding (RLNC) [17]

combinação linear dos símbolos correspondentes aos pacotes originais. O RLNC atribui a cada pacote uma variável para ser usada nas operações de álgebra. O resultado é $X = C * P$, onde X é a matriz dos pacotes codificados, C é a matriz dos coeficientes e P a matriz dos pacotes originais. Na fase de descodificação, o recetor deve saber os coeficientes usados na codificação, e para isso esses coeficientes são enviados no cabeçalho do pacote codificado. O recetor tem de resolver o conjunto de equações $P = C^{-1} * X$ para obter os pacotes originais. O descodificador precisa de receber tantos pacotes codificados quanto o número de variáveis existentes [17].

O RLNC também traz melhorias de performance quando comparado a outros códigos erasure e códigos Reed Solomon (RS). Isto porque RLNC usa coeficientes aleatórios que são enviados dentro dos pacotes. Assim é possível codificar vários pacotes de uma forma desestruturada, dando a este protocolo versatilidade a responder a falhas. [20]

2.3.2.1 Recodificação em RLNC

A recodificação é uma característica única em RLNC [19]. Com isto os pacotes codificados podem ser recombinados sem ser necessário uma descodificação. Isto porque os pacotes codificados contem os coeficientes de natureza aleatória. A recodificação permite que os nós intermédios combinem pacotes codificados para criar novos pacotes codificados que são combinações dos pacotes usados na primeira fase da codificação. A figura 2.3 (A) ilustra os pacotes originais de cor completa a serem codificados e depois recodificados com outros pacotes já codificados.

O processo de recodificação envolve a combinação linear de pacotes codificados e a geração de novos coeficientes aleatórios. É um processo flexível em que os dados podem ser codificados à medida que se tornam disponíveis como ilustrado na figura 2.3 (B), onde a recombinação pode codificar um pacote já codificado anteriormente com um pacote original que se encontrava ausente.

As figuras 2.4 e 2.5 ilustram o processo de recodificação em redes *multi-hop* e redes *mesh*. Na figura 2.4, 4 pacotes são enviados através de uma rede *multi-hop*. Nesta rede a recodificação permite que os nós intermédios compensar as falhas dinâmicas localmente sem ser necessário voltar ao nó de origem.

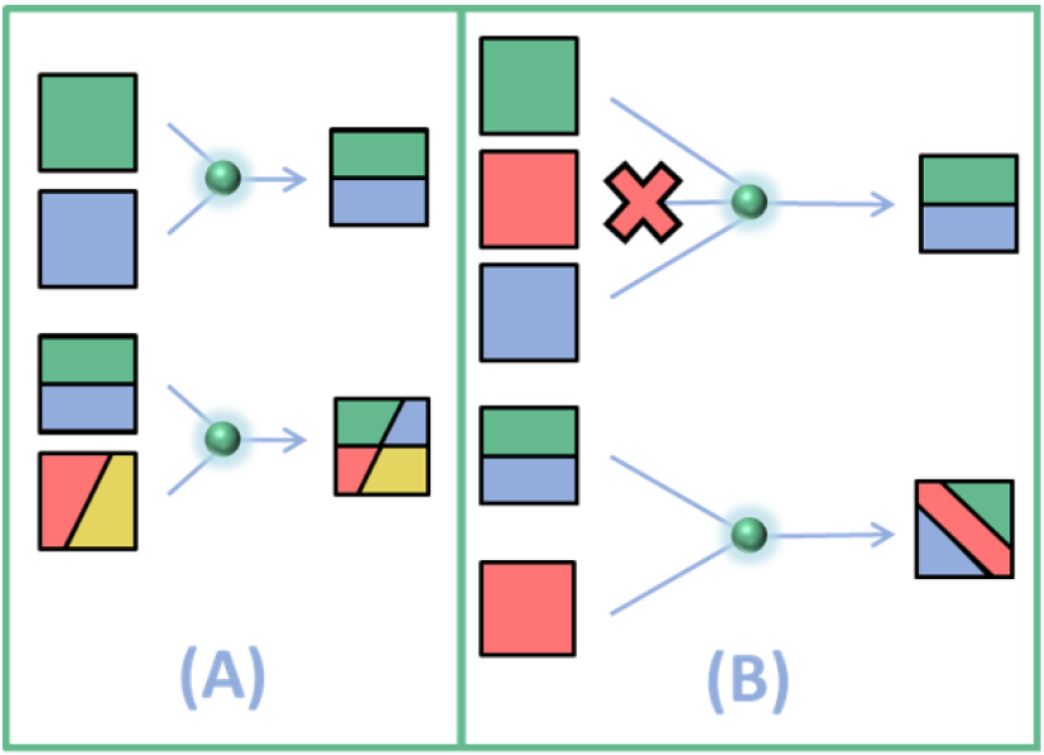


Figura 2.3: Codificação e recodificação usando RLNC [19]

Na figura 2.5 o emissor envia 4 pacotes através de uma rede *mesh* com dois caminhos diferentes. Neste caso é possível enviar um maior número de pacotes redundantes pelas diferentes ligações. Mesmo assim a quantidade de redundância a enviar depende do ruído presente na ligação, da capacidade da ligação e também da capacidade do dispositivo que recebe os dados. Este exemplo ilustra como a capacidade dos diferentes caminhos pode ser combinado dinâmicamente.

O processo de recodificação permite pela primeira vez codificar através de redes *mesh* e em sistemas de armazenamento distribuído. A recodificação em RLNC também minimiza o número total de pacotes transmitidos por uma rede *mesh* ou sistema de armazenamento. Os ganhos de energia assim gerados vão ser superiores aos requisitos de energia necessários para o processo de codificação.

2.3.3 RLNC em armazenamento estático

Em distribuição de armazenamento estático, o conteúdo não é alterado mas apenas enviado para os utilizadores ou nós que vão ler os dados. As vantagens do RLNC em armazenamento estático são:

- **Sistemas de armazenamento robusto:** A capacidade de aceder a dados de vários nós em simultâneo permite assim um aumento da velocidade e entrega de volumes de dados.
- **Menor espaço para armazenamento:** é necessário menos espaço de armazenamento

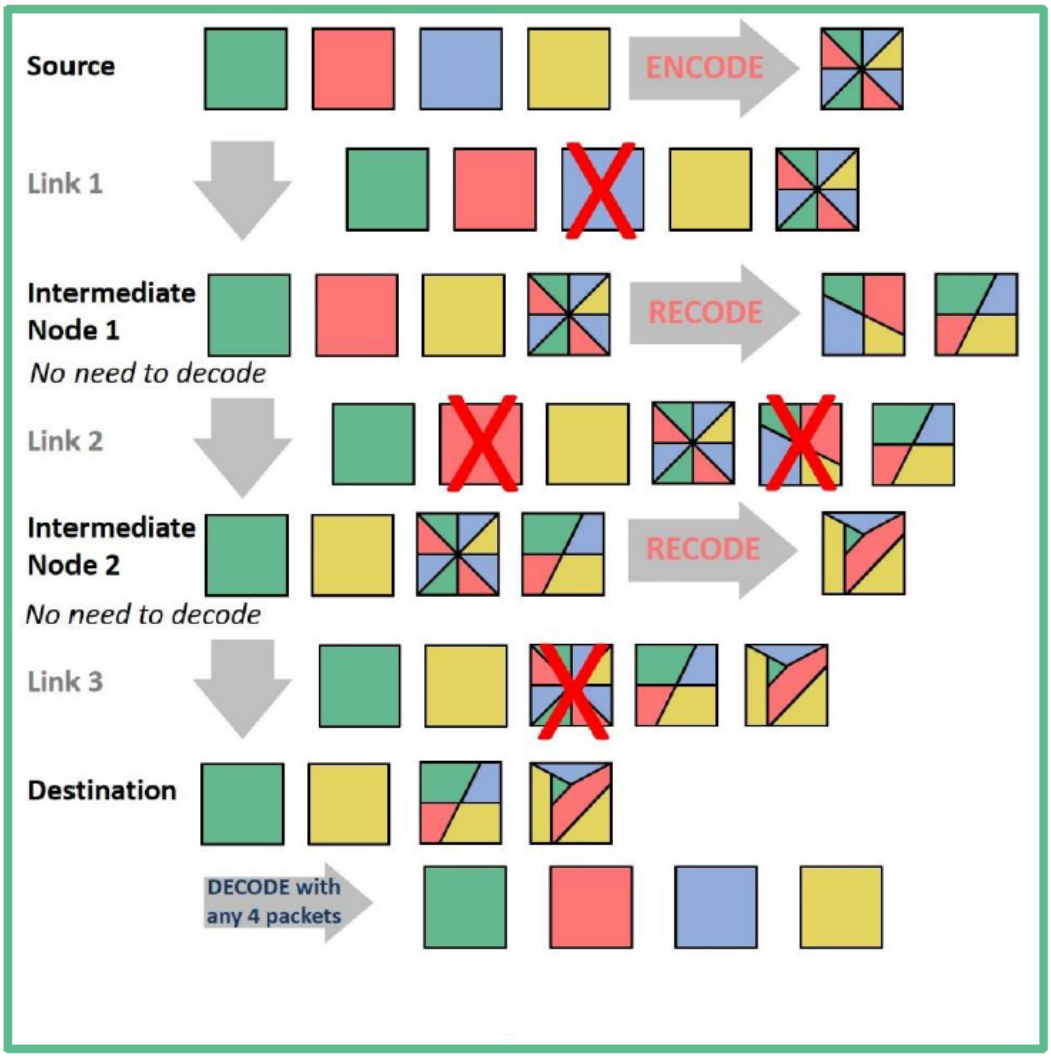


Figura 2.4: Recodificação em redes *multi-hop* [19]

para enviar os mesmos dados de forma viável, reduzindo assim os custos de armazenamento dos utilizadores. Em *data centers* esta propriedade tem a vantagem de produzir maior disponibilidade e capacidade.

- **Largura de banda eficiente:** os dados são reparados de forma mais eficiente sem ser necessário uma grande quantidade de retransmissões. [20]

2.3.4 RLNC em armazenamento dinâmico

Em armazenamento dinâmico o conteúdo pode ser usado e alterado por vários utilizadores e sistemas em paralelo. O uso do RLNC em armazenamento dinâmico vai permitir o uso de novas tecnologias de streaming e distribuição de conteúdo. A flexibilidade do RLNC é adequada para aumentar a fiabilidade, escalabilidade e Quality of Experience (QoE) em ambientes dinâmicos, levando a poderosas tecnologias de streaming e distribuição de conteúdo. As vantagens do RLNC

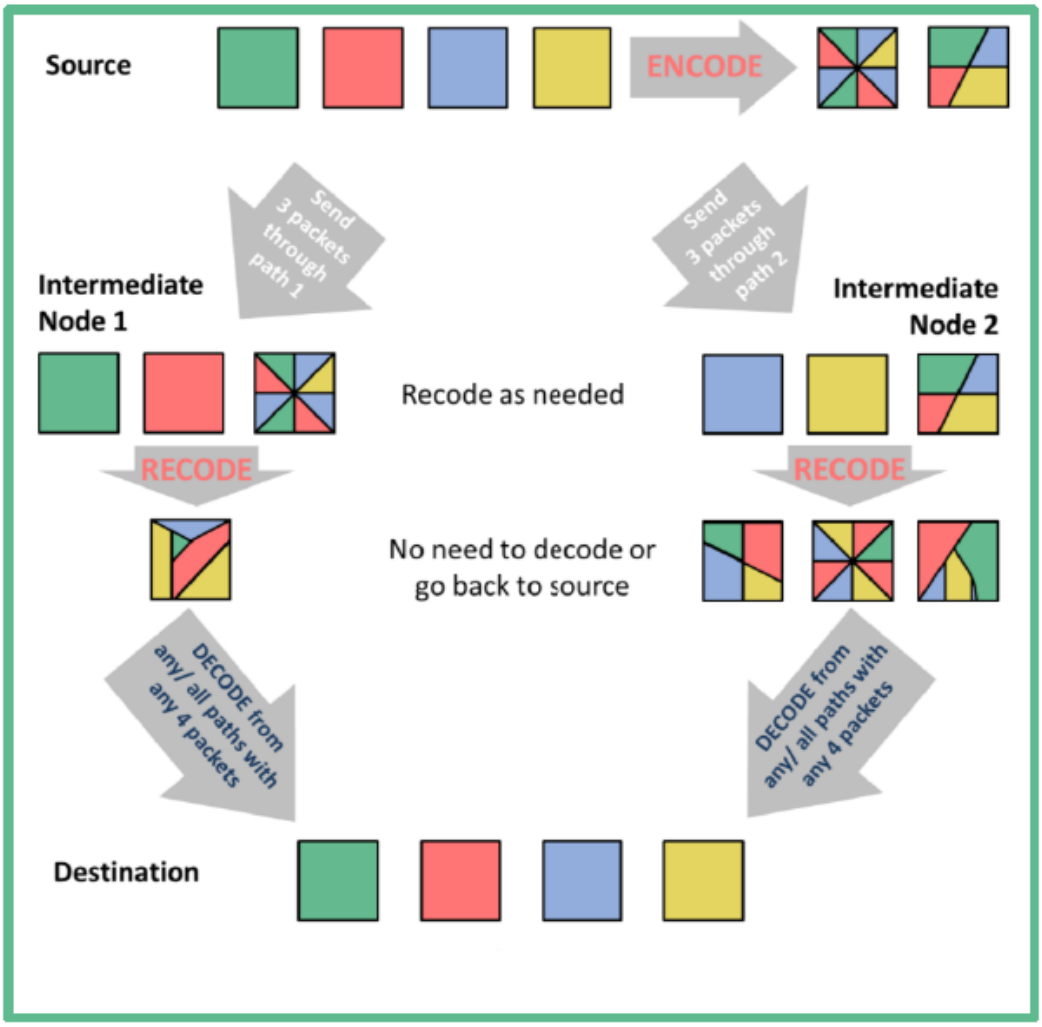


Figura 2.5: Recodificação em redes *mesh* [19]

em armazenamento dinâmico são:

- **Transporte mais fiável:** *Network coding* oferece melhorias na fiabilidade onde exista uma fraca ligação à rede. O RLNC tem a capacidade de codificar *on-the-fly*. Isto é possível porque a codificação pode começar quando chegam dois ou mais pacotes. Quando um pacote está codificado em RLNC ele pode dar origem a múltiplos pacotes, evitando assim que certos pacotes específicos não necessitem de ser enviados.
- **Uso da rede mais eficiente:** Os ganhos de largura de banda e tempo de reparação são bem mais significantes no armazenamento dinâmico, especialmente quando os nós são frequentemente actualizados, alterados e reconfigurados. [20]

2.3.5 Casos de uso do RLNC

- **Redes Multicast:** Com o aumento do número de sistemas recetores torna-se difícil gerir todos os feedbacks dos recetores individuais. Com isto surgiram protocolos de multicast complexos como o Nack Oriented Reliable Multicast (NORM) que podem adicionar ainda mais sobrecarga na rede. Assim o uso do RLNC nestes sistemas pode reduzir significativamente a necessidade de feedback e torna-se uma solução para o aumento da QoE.

- **Comunicação por Satélite:** O setor das comunicações por satélite está sempre em crescimento e em uso de novas tecnologias. O RLNC é introduzido nestas comunicações pois permite gerar dados codificados em tempo real e a latência, complexidade e fiabilidade podem ser ajustadas dinamicamente para satisfazer as condições e requisitos da rede. O RLNC permite uma margem maior de trocas e optimizações comparado com as soluções usadas normalmente e é adequado para ambientes com alto Round Trip Time (RTT).

- **Armazenamento Distribuído:** O RLNC tem o recurso da recodificação que permite que um nó possa codificar a partir de outros pacotes codificados sem ter de descodificar primeiro. Isto permite aos nós de armazenamento e caches intermédias criem redundância adicional e de forma descentralizada, reduzindo a velocidades de transporte armazenamento, energia e reparo da unidade.

  As soluções apresentadas pelo RLNC podem ser usadas em sistemas de armazenamento distribuído, unidades de cache e sistemas de *cloud* híbridos. Com isto as soluções do RLNC tem um papel importante nas redes Peer to Peer (P2P), Content Delivery Network (CDN) e serviços de *cloud*.

- ***Mesh Networking* e Internet of Things (IoT):** Como falado em cima as tecnologias que usam RLNC podem ser combinadas para criar redes *mesh* mais sofisticadas. Primeiro, os protocolos *mesh* melhorados pelo RLNC permitem que os nós armazenem pacotes ouvidos e os codifiquem para criar nova redundância. Segundo, o RLNC permite codificar em tempo real. Os novos protocolos que usam RLNC são capazes de reduzir a latência da rede e melhorar a resiliência das redes *mesh*.

  O RLNC demonstrou um forte potencial em redes *mesh* e comunicações *multipath* e está a ser estudado e testado em diferentes áreas tecnológicas, como 5G, Over the Top (OTT), Vehicle to Vehicle (V2V) e redes *time-sensitive* pelas suas topologias confiáveis em redes *mesh* de baixa latência e operações *multipath* [13].

2.4 Storage Coding

Um sistema de armazenamento distribuído é uma rede de nós de armazenamento onde informação redundante de ficheiros de dados são distribuídas pelos vários nós na rede para fornecer fiabilidade em caso de falhas ou perdas num dos nós. Existem diversos tipos de aplicações, como *data*

centers, sistemas *peer-to-peer* e sistemas de armazenamento via *wireless*. O armazenamento de dados em vários nós usa *erasure coding* e requer menos redundância do que simplesmente fazer uma replicação completa dos dados, permitindo assim reduzir os custos de armazenamento e o tempo de copiar os dados de uma nó para outro. Os nós da rede podem falhar por serem desligados durante a manutenção, falhas de energia ou porque o seu tempo de vida chegou ao fim. Assim os sistemas de armazenamento tem de garantir um armazenamento confiável dos volumes de dados por longos períodos de tempo mesmo que alguns nós da rede ficam inacessíveis para ler o seu conteúdo. Para garantir esta fiabilidade começou por ser introduzida a replicação dos dados. A replicação corresponde à cópia dos dados originais. Os dados ficam disponíveis até uma cópia existir na rede de nós. No entanto esta técnica não é vantajosa pois quando existem grandes quantidades de dados para distribuir pelos vários nós este processo torna-se ineficiente a nível de custo de armazenamento e custos operacionais como espaço necessário para conter o hardware, custos de energia e manutenção.

Assim os sistemas de armazenamento usam meios mais eficientes que consomem menos largura de banda e armazenamento para mover grandes quantidades de dados na rede quando um nó falha. Os sistemas utilizam erasure codes como *Reed Solomon* que apresentam uma otimização de armazenamento dado que são códigos Maximum Distance Separable (MDS). Outro meio usado pelos sistemas de armazenamento é o Redundant Array of Independent Disks (RAID), que usa códigos *Reed Solomon* para recuperar os dados quando múltiplos nós falham em simultâneo [16].

- **Códigos MDS** : Num código MDS (n, k), um ficheiro de tamanho M pacotes de um corpo finito F_q é dividido em k fragmentos de tamanho M/k, codificado e armazenado em n nós. O ficheiro original pode ser recuperado de um conjunto de k fragmentos. Assim o sistema pode tolerar a falha de $n - k$ nós e o total de informação armazenada é nM/k [21].

2.4.1 Reed Solomon

O código *Reed-Solomon* é um código de correção de erros que pertence à classe dos códigos de blocos. Por ser um código de blocos, os dados a serem enviados são divididos em blocos separados de dados. Ao final de cada bloco é adicionada uma informação de paridade de forma a formar uma *codeword*. Um código *Reed-Solomon* pode ser descrito como RS (n, k) com pacotes de m bits, onde n é o tamanho completo da *codeword*, k representa o número de pacotes da mensagem original e $n - k = 2t$ representa o número de pacotes de paridade, ilustrado na figura 2.6. Assim o emissor pega em k pacotes de dados de m bits cada, adiciona $n - k$ pacotes de paridade e produz uma *codeword* de n pacotes. O descodificador pode corrigir até t pacotes que contém erros numa *codeword*, onde $2t = n - k$ [11] .

Como o código *Reed-Solomon* tem uma distancia mínima $d_{min} = n - k + 1$, este código é óptimo no sentido de a distancia mínima ser o valor máximo para um tamanho de código linear (n, k). Assim este código é chamado código MDS. Um código *Reed-Solomon* é limitado por um Galois Field (GF)(2^n). Quanto maior for o *Galois Field*, maior vai ser o código, mas em

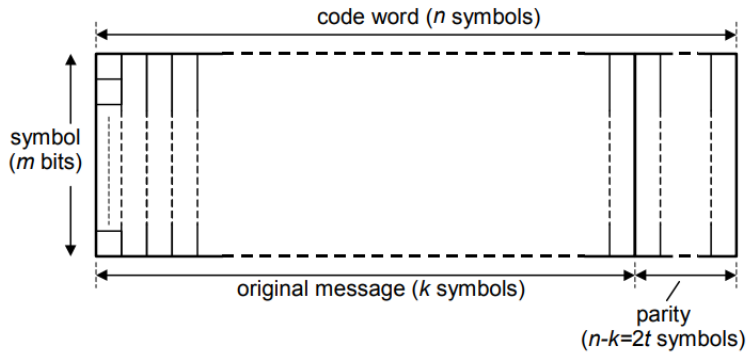


Figura 2.6: Descrição do código Reed-Solomon [11]

contrapartida maior vai ser o tempo e a complexidade das operações. Assim muitos sistema de armazenamento distribuídos usam este código pois tem otimização de armazenamento (dado que é um código MDS) e tem uma aplicação genérica pois é possível construir códigos RS com n e k arbitrários [16].

2.4.2 RAID

O sistema RAID é uma forma de armazenar os mesmos dados em diferentes discos de forma a proteger esses dados em caso de ocorrer uma falha em algum dos discos. Normalmente este sistema é usado em dispositivos Network Attached Storage (NAS) e servidores usados em sistemas de armazenamento na *cloud*, *datacenters* e em uso doméstico. O RAID usa dois ou mais discos de forma a melhorar a performance e fornecer um certo nível de tolerância a falhas, pois caso um disco falhe o outro continua operacional sem o utilizador se aperceber dessa falha [5]. Este sistema aparece no sistema operativo como um único disco lógico e usa as técnicas de *disk mirroring* ou *disk striping*. *Mirroring* faz uma copia dos dados por diferentes discos. *Striping* particiona cada unidade de armazenamento em intervalos de um setor (512 bytes) até vários megabytes. As *stripes* de todos os discos são intercaladas e endereçadas em ordem [24].

Existem vários níveis de RAID que podem ser configurados pelo utilizador para os fins que ele pretender. Os diferentes níveis de RAID são configurados dependendo do número de discos que existem no sistema, quão critico os dados de um disco são em caso de falha e qual é o nível de performance desejada para o sistema de armazenamento [6].

- **RAID 0** : O RAID 0 usa a técnica de *striping* onde os dados são segmentados e distribuídos por vários discos. Nesta configuração não existe redundância o que torna não tolerante a falhas, ou seja, caso um disco falha os dados são perdidos. A vantagem desta configuração é que existe alta performance tanto na leitura como na escrita do disco.
- **RAID 1** : O RAID 1 usa a técnica de *mirroring* , onde é necessário dois discos que contem a exacta duplicação dos dados. Na pratica é como se existisse apenas um disco, sendo que os restantes são uma copia do primeiro. Assim caso um disco falhe toda a informação esta

contida nos outros discos. O RAID 1 garante redundância ao contrário do RAID 0. A performance pode ser um ponto fraco dado que é necessário escrever a mesma informação em todos os discos.

- **RAID 2** : Semelhante ao RAID 0 só que a informação distribuída pelos vários discos é feita ao nível do bit recorrendo a um processo de Error Correction Code (ECC). Apresenta baixa performance mas alto nível de segurança da informação guardada. Este modelo já não é usado pois os discos já tem um sistema de deteção e correção de erros.
- **RAID 3** : O RAID 3 usa *striping* e necessita de mais um disco para guardar informação de paridade. A recuperação dos dados em caso de falha de um dos discos é realizada ao fazer o XOR do disco de paridade com os outros discos.
- **RAID 4** : Este nível usa grandes *stripes* o que permite ler registos de qualquer disco. Isto permite seja usado operações de *I/O* em simultâneo. No entanto apresenta uma performance de escrita lenta dado que qualquer operação de escrita exige que o disco de paridade seja atualizado.
- **RAID 5** : Esta nível é similar ao RAID 4 mas supera alguns dos problemas apresentados nesse nível. As informações de paridade são distribuídas ao longo de todos os discos ao invés de serem guardadas apenas num único disco, oferecendo assim mais desempenho que o RAID 4. Este nível não é recomendado para sistemas com gravação intensiva devido ao impacto no desempenho associado à atualização da informação de paridade.
- **RAID 6** : O RAID 6 é semelhante ao RAID 5 mas inclui o dobro dos bits de paridade garantindo a integridade dos dados caso ocorra a falha de dois discos. Este nível tem um maior necessidade de armazenamento e apresenta maior tempo na fase de escrita.
- **RAID 10** : Este nível corresponde a uma junção do RAID 1 com o RAID 0. Os dados são espelhados (RAID 1) garantindo redundância e distribuídos (RAID 0) para melhorar a performance. Assim metade podem falhar em simultâneo desde que não falhem os dois discos de um espelho.
- **RAID 01** : Este nível é uma combinação dos níveis RAID 0 e RAID 1 onde os dados são distribuídos pelos discos para melhorar a performance e utilizam outros discos para duplicar os dados. São necessários pelo menos 4 discos para criar um RAID deste nível.

2.5 Resumo

Neste capitulo foram explicadas várias técnicas de deteção de erros foram na Sec. 2.1. A técnica de *erausre coding* foi explicada na Sec. 2.2. De seguida na Sec. 2.3 foram apresentadas as técnicas de *Linear Netowk Coding* e *Random Linear Network Coding* bem como os casos de uso do *Random Linear Network Coding*. Por fim na Sec. 2.4 foram apresentadas várias técnicas de codificação para armazenamento.

Capítulo 3

Técnicas de codificação baseadas em PRAC

Neste capítulo é descrito o funcionamento das três técnicas de codificação para correção de erros em sistemas de armazenamento. Duas delas, o Packetized Rateless Algebraic Consistency (PRAC) e o Data Aware Packetized Rateless Algebraic Consistency (DAPRAC), foram implementadas neste trabalho, enquanto que o Segmented Packetized Rateless Algebraic Consistency (S-PRAC) ficou proposto para o trabalho futuro. Cada técnica tem associada uma fase de codificação e uma fase de descodificação e dentro de cada fase tem os algoritmos para codificar e descodificar os dados. Todas estas fases são explicadas neste capítulo. O PRAC é apresentado na secção 3.1, de seguida o DAPRAC na secção 3.2 e por fim o S-PRAC na secção 3.3.

3.1 PRAC

No artigo [9], é proposto um novo esquema de recuperação parcial de pacotes, chamado PRAC. Este esquema permite identificar e recuperar segmentos de pacotes corrompidos sem recorrer a retransmissões dos pacotes corrompidos. A maioria dos sistemas descarta os pacotes independentemente do número de erros que eles contêm. No entanto descartar pacotes e pedir a sua retransmissão pode não ser uma solução óptima e poderá adicionar sobrecarga na rede pois os pacotes ainda contêm uma certa informação útil. Esta informação útil será usada para recuperar os dados que estão corrompidos.

No PRAC os dados transmitidos são codificados usando Random Linear Network Coding (RLNC) o que faz com que os dados a enviar sejam combinados recorrendo a conjunto de coeficientes aleatórios dando origem a combinações lineares dos pacotes originais. Os dados corretos são aproveitados dos dados corrompidos usando verificações Algebraic Consistency Rule (ACR). Verificações ACR exploram a propriedade de códigos lineares (n, k), onde apenas k de n pacotes são suficientes para descodificar os dados iniciais e os outros $(n - k)$ podem ser expressos como combinações lineares deles. O processo de recuperação é feito de forma iterativa

usando um algoritmo de procura otimizado, verificações ACR e atualizações Cyclic Redundancy Check (CRC) .

3.1.1 Fase de Codificação

A fase de codificação é executada sobre um grupo de pacotes k , conhecido também como tamanho da geração. Seja P_i, onde $i = 1, 2, ..., k$, os pacotes originais para transmissão consistindo em l blocos. Estes blocos são chamados pacotes, tem tamanho q bits e pertencem a um corpo finito $F(2^q)$. Usando notação de matriz, os pacotes originais podem ser expressos como uma matriz $P = k * l$. Na fase de codificação os k pacotes originais dão origem a m pacotes codificados pela multiplicação da matriz sobre operações do corpo finito : $P' = C * P$, onde C é a matriz dos coeficientes gerados aleatoriamente sobre o corpo finito e P' é a matriz com os pacotes codificados. Cada pacote tem associado um conjunto de coeficientes que tem de ser enviados com o pacote, pode ser no cabeçalho, ou então tem de ser gerados no recetor em sincronização com o emissor.

3.1.2 Fase de Descodificação

Quando um novo pacote chega no recetor ele é marcado como válido se o seu CRC não detetar erros, ou então, é marcado como parcial caso apresente erros. Quando o número de pacotes recebidos é superior ao tamanho de geração (k) a fase de recuperação é iniciada. Este processo é realizado ao longo das colunas da matriz P' de forma sequencial. Os erros nos pacotes são detetados usando verificações ACR e os erros são corrigidos por um processo iterativo usando um algoritmo otimizado de pesquisa e verificações ACR.

3.1.2.1 Verificações ACR

PRAC identifica a informação correta dentro dos pacotes parciais usando a vantagem da propriedade dos códigos lineares (n, k) , onde um conjunto k de n pacotes válidos podem ser usados para descodificar os dados iniciais e os restantes pacotes $(n - k)$ são uma combinação linear deles. Assumindo que o tamanho da geração é k e foram recebidos x pacotes, onde $(x > k)$, incluindo parciais e válidos. Os passos das verificações ACR são 4 e são os seguintes (3.1):

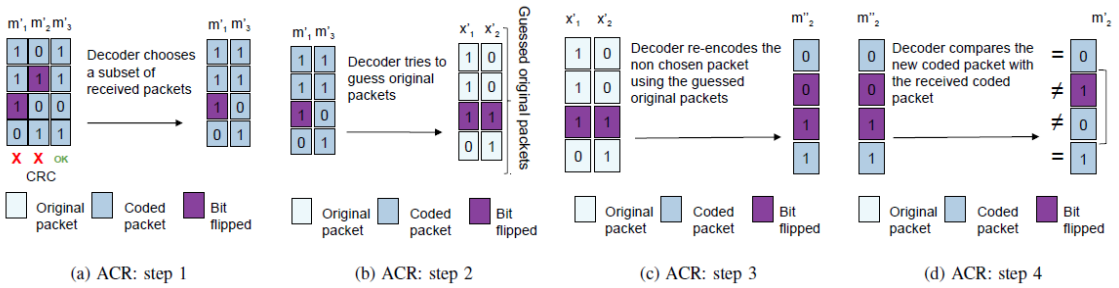


Figura 3.1: Passos do algoritmo ACR [15]

No primeiro, são escolhidos um conjunto de pacotes codificados, válidos e inválidos, ($m_1^{'}$ e $m_3^{'}$). No passo 2 o recetor descodifica os pacotes escolhidos ($m_1^{'}$ e $m_3^{'}$) usando RLNC para obter os pacotes originais ($x_1^{'}$ e $x_2^{'}$). De seguida, no passo 3, o recetor recodifica o pacote codificado não escolhido inicialmente ($m_2^{''}$) recorrendo aos pacotes descodificados no passo 2. Por fim, no passo 4, é feita uma comparação entre o pacote recodificado ($m_2^{''}$) e o pacote codificado que não foi escolhido inicialmente ($m_2^{'}$). O recetor estima a localização dos erros caso os pacotes comparados não sejam iguais. Como o recetor não sabe a localização exata dos bits corruptos, ele aplica várias vezes as verificações ACR. O processo de correção é aplicado depois de cada verificação ACR ser efetuada, onde é feita uma procura de força bruta para encontrar o bit correto de forma a que o CRC interno esteja válido. Por fim a fase de descodificação é realizada quando a fase 1 esteja terminada, ou seja, quando o pacote codificado já não contiver erros. Na fase 2 é feita a descodificação usando RLNC para obter os pacotes originais sem erros.

3.2 DAPRAC

No projeto [10], foi desenvolvido um novo algoritmo implementado em paralelo com o PRAC. O tempo de descodificação no PRAC pode aumentar muito caso os pacotes cheguem ao recetor com demasiado erros. Isto porque o PRAC usa força bruta nas combinações possíveis de pacotes para corrigir os erros detetados nos pacotes. Com isto, caso exista um aumento do número de erros vai haver um aumento no tempo de descodificação podendo até nem ser possível corrigir. O novo algoritmo chamado DAPRAC, pode diminuir o tempo de descodificação do PRAC em duas ordens de magnitude sobre certas características de erros. A ideia principal passa por introduzir um código CRC extra computado sobre os dados do pacote original. Esta informação adicional pode ser usada para corrigir mais rápido os erros pois reduz o espaço de pesquisa de pacotes.

3.2.1 Fase de Codificação

Na figura 3.2 está ilustrado a fase de codificação do algoritmo DAPRAC. Em cada pacote original é adicionado um código CRC que é computado sobre os dados do pacote original. A restante parte da codificação é a mesma que a PRAC, onde os pacotes originais são multiplicados pelos vetores de coeficientes aleatórios de forma a produzir os pacotes codificados que são combinações lineares dos pacotes originais. Os pacotes codificados, identificados como m_i, vão ser enviados pelo canal de comunicação e vários dos seus bits podem sofrer alterações. O recetor vai receber os pacotes codificados m_i^* que podem conter erros devido ás alterações que sofreram no canal de comunicação.

3.2.2 Fase de Descodificação

Quando o recetor receber $g+1$ pacotes, onde g é o tamanho da geração, ele pode começar a fase de deteção de erros. Esta fase ilustrada na figura 3.3, consiste em descodificar várias vezes

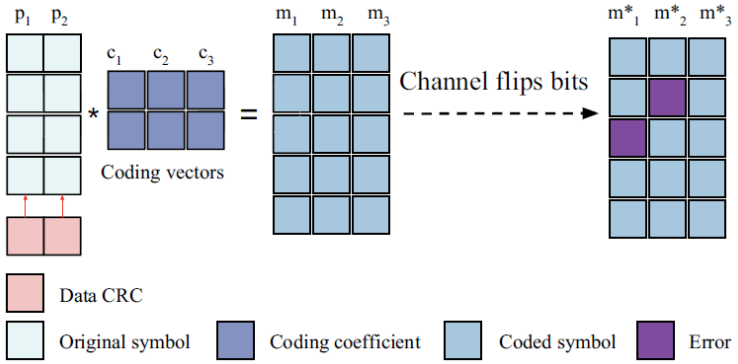


Figura 3.2: Método de codificação do DAPRAC [10]

os pacotes usando todas as combinações possíveis de g pacotes dos $g+1$ pacotes codificados recebidos. Estes pacotes são chamados pacotes pré-descodificados e cada pacote p_i^{xy} é o pacote pré-descodificado i usando a combinação dos pacotes recebidos m_x^* e m_y^* com $x \neq y$.

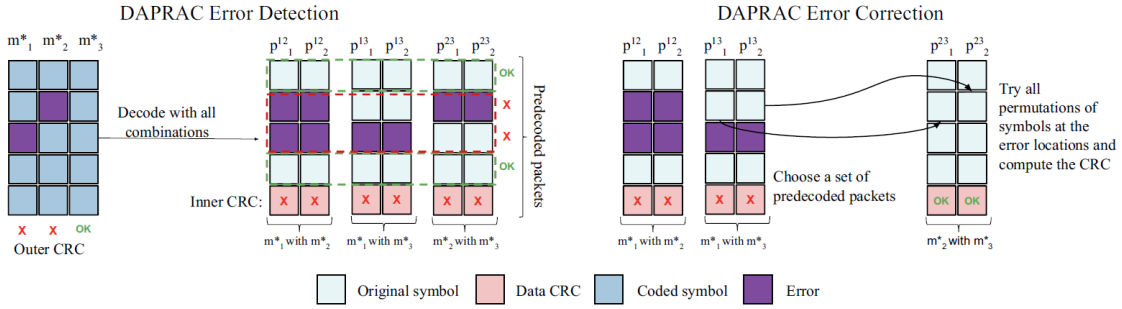


Figura 3.3: Algoritmo de descodificação do DAPRAC [10]

A localização dos erros é detetada encontrando diferenças de símbolos nos pacotes pré-descodificados. Se os símbolos numa certa posição em todos os pacotes p_i^{xy} com i fixo forem iguais, então não existem erros nessa posição. Caso contrario, os erros encontram-se nesses locais.

Por fim, na fase de correção dos dados na figura 3.3, o recetor escolhe aleatoriamente um conjunto de pacotes p_i^{xy} com x e y fixos. Para cada localização de erros ele permuta os símbolos do pacote escolhido pelos símbolos da mesma localização dos outros pacotes pré-descodificados e recalcula o CRC interno até este estar correto. Quando o CRC estiver correto o recetor tem o pacote descodificado corretamente.

A vantagem do DAPRAC sobre o PRAC está no processo de correcção de erros, pois em vez de usar força bruta para calcular as combinações de símbolos nas posições com erros, o DAPRAC faz permutações de um número reduzido de símbolos até encontrar o correto. A desvantagem deste algoritmo é que não pode haver erros na mesma posição em todos os pacotes recebidos e o CRC interno também não pode conter erros.

3.3 S-PRAC

No mais recente artigo de 2019 [15], é proposto um novo esquema para recuperar pacotes parciais em canais wireless com elevado ruído, chamado S-PRAC. Nos esquemas PRAC e DAPRAC falados anteriormente, a performance tende a diminuir quando o número de erros aumenta em canais de comunicação com ruído, o que faz com que o tempo na fase de recuperação tenda a aumentar. O S-PRAC veio melhorar o tempo de detecção e correcção de erros apresentados nos esquemas anteriores e é eficiente e rápido em canais que apresentam condições de muito ruído.

3.3.1 Fase de Codificação

A fase de codificação é semelhante aos esquemas falados anteriormente, onde o emissor tem k pacotes originais para enviar M_1,M_2,...,M_k e cria n pacotes codificados X_1,X_2,...,X_n onde n>k. Cada pacote codificado está relacionado com uma serie de coeficientes aleatórios G_i. A matriz de codificação é definida por $X = G * M$, onde X é a matriz dos pacotes codificados, G é a matriz dos coeficientes aleatórios e M a matriz dos pacotes originais. A segmentação dos pacotes apresentada na figura 3.4 é efetuada depois de os pacotes estarem codificados. O emissor divide cada pacote codificado em segmentos de tamanho igual e no final de cada segmento adiciona um código CRC-8 interno. No final adiciona um código CRC-32 externo no final de cada pacote segmentado. Com o aumento do número de segmentos faz com que o número de CRC internos seja maior mas isto vai fazer com que o tempo de descodificação seja significativamente menor.

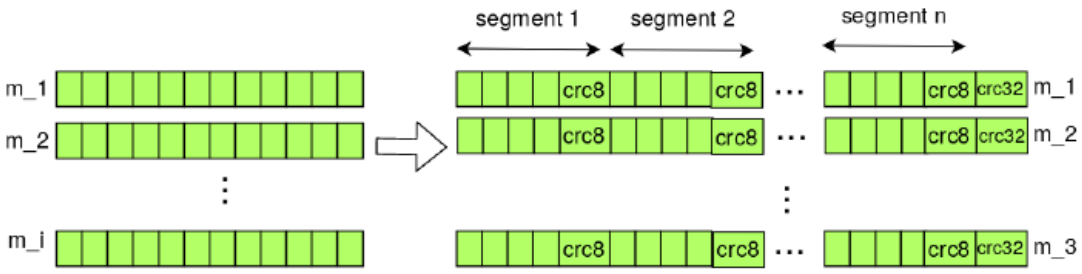


Figura 3.4: Processo de segmentação do S-PRAC [15]

3.3.2 Fase de Descodificação

O processo de descodificação é constituído por duas fases: a primeira é a fase de estimação e correção de erros e a segunda a fase de descodificação. O recetor ao receber os pacotes codificados analisa o seu CRC externo e marca os pacotes como válidos ou inválidos. Os pacotes ficam num buffer a aguardar que comece a primeira fase, que começa quando o recetor receber $g+1$ pacotes, onde g é o número de pacotes (tamanho da geração). A primeira fase começa com verificações

ACR, como ilustrado na figura 3.1. Esta fase é constituída por quatro passos. No primeiro, são escolhidos um conjunto de pacotes válidos e inválidos ($m_1^{'}$ e $m_3^{'}$). No passo 2 o recetor descodifica os pacotes escolhidos ($m_1^{'}$ e $m_3^{'}$) usando RLNC para obter os pacotes originais ($x_1^{'}$ e $x_2^{'}$). De seguida, no passo 3, o recetor recodifica o pacote não escolhido inicialmente ($m_2^{''}$) recorrendo aos pacotes descodificados no passo 2. Por fim, no passo 4, é feita uma comparação entre o pacote recodificado ($m_2^{''}$) e o pacote que não foi escolhido inicialmente ($m_2^{'}$). O recetor estima a localização dos erros caso os pacotes comparados não sejam iguais. Como o recetor não sabe a localização exata dos bits corruptos, ele aplica várias vezes as verificações ACR. O processo de correção é aplicado depois de cada verificação ACR ser efetuada, onde é feita uma procura de força bruta para encontrar o bit correto de forma a que o CRC interno esteja válido. Por fim a fase de descodificação é realizada quando a fase 1 esteja terminada, ou seja, quando o pacote codificado já não contiver erros. Na fase 2 é feita a descodificação usando RLNC para obter os pacotes originais sem erros. A segmentação reduz assim o tempo de correcção, pois limita o número de permutações necessárias no processo de correcção.

3.4 Resumo

Neste capítulo apresentados duas técnicas implementadas neste trabalho para codificação de dados em armazenamento. O PRAC (Sec. 3.1) foi a primeira a ser implementada e de seguida o DAPRAC (Sec. 3.2). Por fim, foi apresentada a técnica que ficou proposta para trabalho futuro S-PRAC (Sec. 3.3). Nestas técnicas foram apresentadas as fases de codificação e descodificação dos dados bem como os algoritmos usados para detetar e corrigir os erros introduzidos nos dados.

Capítulo 4

Desenho e implementação de software

Neste capítulo são descritas as ferramentas usadas (4.1) na implementação das técnicas propostas e quais os seus casos de uso. Nas secções 4.2 e 4.3 são descritas detalhadamente os principais métodos e algoritmos usados no código do Packetized Rateless Algebraic Consistency (PRAC) e Data Aware Packetized Rateless Algebraic Consistency (DAPRAC). Por fim na secção 4.3 é explicado como executar o programa e quais os requisitos que devem ser instalados previamente para o programa funcionar.

4.1 Ferramentas usadas

Para desenvolver este projeto foram escolhidas a biblioteca de *network coding*, KODO [4], e a biblioteca de algoritmos criptográficos Cryptopp [2] ambas escritas em C++. KODO é uma biblioteca de código aberto desenvolvida em C++ e com o objetivo de ser usada para estudos práticos de algoritmos de codificação em rede. Esta biblioteca integra uma Application Programming Interface (API) que permite tirar vantagem de um grande conjunto de erasure codes desde dos mais recentes, como *Random Linear Network Code*, até a códigos tradicionais como o *Reed-Solomon*. A biblioteca foi desenvolvida para garantir desempenho, estabilidade e flexibilidade e além disso a biblioteca é continuamente testada e comparada em várias plataformas para garantir uma grande otimização. Foi por estas razões que a biblioteca KODO foi escolhida para ajudar a desenvolver este projeto. KODO está disponível sob uma licença de pesquisa e educação que permite aos investigadores implementar novos códigos e algoritmos, fazer simulações e comparar essas operações em qualquer plataforma onde exista um compilador C++.

Cryptopp é uma biblioteca open-souce de algoritmos criptográficos escrita por Wei Dai [8]. Esta biblioteca é usada em projetos de estudo/investigação, projetos não comerciais e empresas. Tem suporte para arquiteturas 32 e 64 bits num grande conjunto de sistemas operativos e plataformas. Contém um grande conjunto de algoritmos, como, por exemplo, cifras de blocos, códigos de autenticação de mensagens, funções de *hash*, criptografia de chave pública e códigos de deteção de erros. Dado os algoritmos apresentados na biblioteca Cryptopp, achamos que seria

útil tirar partido desta biblioteca, principalmente pela necessidade de usar o algoritmo Cyclic Redundancy Check (CRC) usado para a deteção de erros nos pacotes.

4.1.1 Casos de uso da biblioteca KODO

- **Comunicações com baixo atraso** : Usando algoritmos de correção de erros da biblioteca KODO, pode ser adicionada redundância no transporte dos pacotes, permitindo assim que o recetor possa recuperar rapidamente os pacotes que foram perdidos e evitar o aumento de Round Trip Time (RTT) que são introduzidos em esquemas de garantia de entrega de pacotes como é o caso do protocolo Transmission Control Protocol (TCP).
- **Projetar um protocolo de multicast fiável** : A biblioteca KODO é ideal para desenhar protocolos de multicast fiável, pois muitos dos desafios são resolvidos com mais eficiência e facilidade. O protocolo de multicast confiável da *Steinwurf* é o *Score* e é desenvolvido recorrendo ao KODO.
- **Aumentar a fiabilidade da comunicação** : Usando os códigos de correção de erros do KODO, a comunicação entre dispositivos pode ser robusta em cenários com alta probabilidade de perdas. Isto é feito ao enviar redundância codificada que é mais provável de ser útil no lado do recetor. Em cenários em que o envio de pacotes é feito através de redes de múltiplos saltos, a recodificação dos pacotes pode ser usada para obter ganhos ainda maiores.
- **Aumentar a robustez do armazenamento** : Com o KODO é possível criar sistemas de armazenamento mais robustos para a perda de dados devido a falhas em discos rígidos. Isso é feito usando os códigos de correção erasure que codificam os dados existentes e recuperam os dados originais.

4.2 Implementação PRAC

O código do programa PRAC foi fornecido pelo meu orientador pois era um trabalho que ele já estava envolvido no passado. No entanto partes desse código foram alteradas, nomeadamente a função *erros* (4.2.1) que permite introduzir erros em cada posição de um pacote de dados com uma certa probabilidade definida pelo utilizador. Todas os principais métodos e algoritmos referentes ao PRAC são explicados aqui nesta secção.

A figura 4.1 ilustra os estados do programa PRAC. Nem todos os métodos presentes no PRAC são apresentados neste fluxo, no entanto, esses métodos também serão detalhados mais à frente.

Quando o programa é compilado e executado no estado de *Start* os pacotes começam a ser codificados usando o algoritmo Random Linear Network Coding (RLNC). Isto acontece

durante o estado em que descodificador normal da biblioteca KODO não está completo (!decoder.is_complete()), ou seja, quando o rank da matriz não está completo. Durante este processo são enviados $g + n$ pacotes para o descodificador PRAC, sendo que os estes pacotes vão passar por um método que pode, eventualmente, adicionar erros com uma probabilidade estabelecida pelo utilizador. G significa o tamanho da geração, ou seja, o número de pacotes que vão ser transmitidos e n o número de pacotes a enviar adicionalmente. Isto acontece no estado *Add errors*. Esses pacotes que podem sofrer erros são depois enviados para o método *receive_entry()* bem como o pacote adicional que chega do codificador normal. O método *receive_entry()* tem como inputs o pacote codificado, os coeficientes usados na codificação desse pacote e também o CRC calculado sobre esse pacote codificado e retorna *true* quando o número de pacotes recebidos for igual a $g + 1$. De seguida o programa entra no estado *decode()*. Este método vai tentar detetar a localização dos erros introduzidos e corrigir esses erros calculando todas as permutações possíveis nessas posições. Se este processo for concluído com sucesso é retornado *true* e avança para o próximo estado *recove_decoded_data()*, caso contrário, o processo de descodificação falha. O método *recover_decoded_data()* recebe como input um vetor onde vão ser escritos os dados descodificados que depois será comparado com o vetor dos dados iniciais antes de serem codificados. Se esta comparação for igual, então os dados foram bem descodificados e são apresentados para o utilizador.

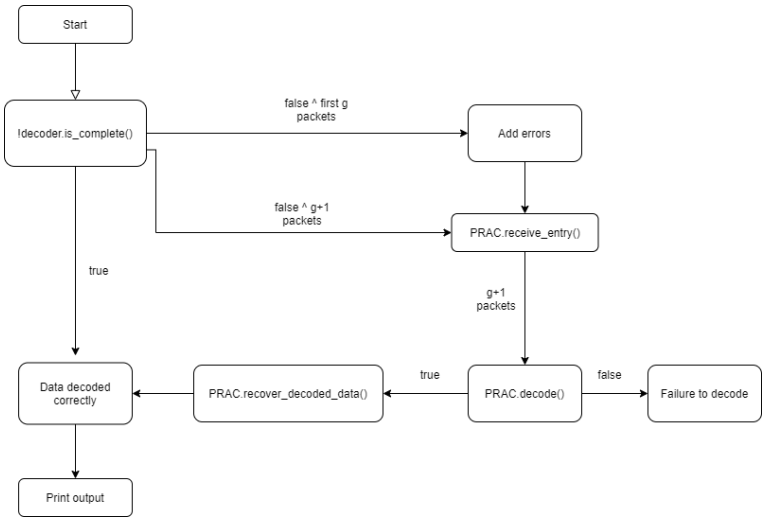


Figura 4.1: Diagrama de fluxos PRAC

Na figura 4.2 temos o diagrama de classes do PRAC. O método *printHex()* recebe como inputs o pacote codificado, o CRC desse pacote e uma variável booleana a indicar se esse pacote está corrompido ou não. este método imprime em hexadecimal esse pacote bem como o seu CRC e ainda "B"ou "G"conforme o pacote esteja corrompido ou não respetivamente. O método *status_decoder()* retorna o número de pacotes corrompidos. O método *print_status()* imprime o conteúdo dos pacotes corrompidos e dos pacotes em bom estado.

Agora será explicado mais detalhadamente todo o processo que não foi explicado anteriormente, nomeadamente o processo de criação dos codificadores e descodificadores , a criação dos buffers

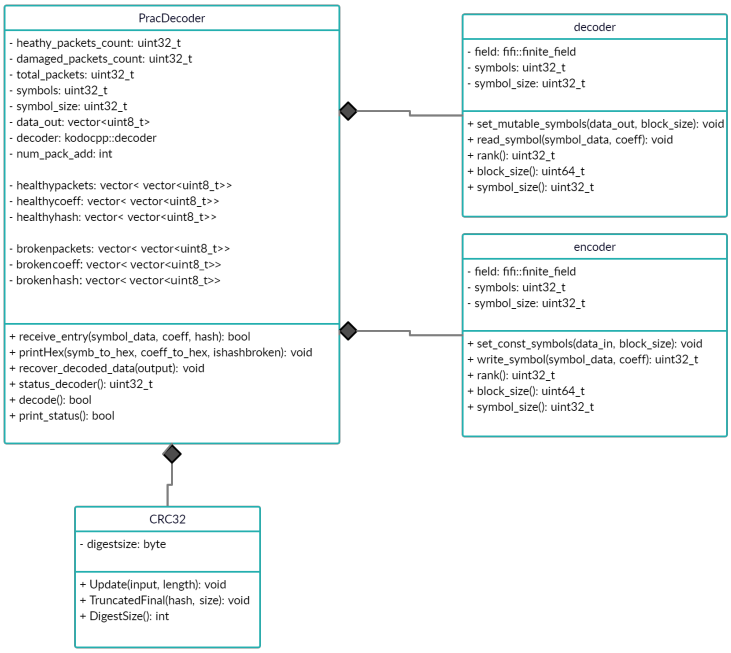


Figura 4.2: Diagrama de classes PRAC

que contém os dados, o processo de codificação e introdução de erros, bem como os métodos do PRAC de forma mais detalhada a nível de código.

No bloco de código 4.1 é apresentado a forma de criar o codificador e o descodificador. Inicialmente criamos o codificador e o descodificador onde são passados vários parâmetros. O *full_vector* corresponde ao formato do vetor de codificação, isto é, enviar o vetor de codificação completo com todos os coeficientes de codificação com cada pacote codificado. O *binary8* corresponde ao tamanho do corpo finito, sendo neste caso um Galois Field $(\mathrm{GF})(2^8)$. Os parâmetros *max_symbols* e *max_symbol_size* correspondem ao número de pacotes iniciais e ao tamanho de cada pacote, respetivamente, passados na linha de comandos.

```
1  kodocpp::encoder_factory encoder_factory(
2          kodocpp::codec::full_vector,
3          kodocpp::field::binary8,
4          max_symbols,
5          max_symbol_size);
6
7  kodocpp::encoder encoder = encoder_factory.build();
8
9  kodocpp::decoder_factory decoder_factory(
10         kodocpp::codec::full_vector,
11         kodocpp::field::binary8,
12         max_symbols,
13         max_symbol_size);
14
15 kodocpp::decoder decoder = decoder_factory.build();
```

Bloco de Código 4.1: Inicialização do codificador e descodificador

Depois de definir o codificador e o descodificador é necessário criar alguns buffers (bloco de código 4.2) onde os dados originais vão ser guardados e codificados. Todos os pacotes vão ser guardados em vetores do tipo *uint8_t*, onde cada posição do vetor vai ter tamanho 8 bits, ou seja, 1 byte. O primeiro buffer a ser criado é o *data_in* e este buffer vai conter os dados que queremos codificar. O tamanho do buffer corresponde ao tamanho do bloco, ou seja, o número de pacotes multiplicado pelo tamanho de cada pacote. No entanto este valor pode também ser obtido com a chamada do *block_size()* definido no KODO. Nesta aplicação não são codificados dados reais, logo o buffer *data_in* vai ser preenchido com bytes aleatórios criados com a função *generate()* do C++. No codificador atribuímos o buffer que ele deve codificar através da chamada *set_const_symbols()*. O buffer *data_out* vai ser associado ao descodificador e é onde o descodificador vai descodificar os dados originais. Para associar o buffer ao descodificador é chamada a função *set_mutable_symbols()*. Quando começa a codificação o codificador vai codificar pacote a pacote e enviar para o descodificador. No vetor *symbol_data* é onde está presente cada pacote codificado a ser enviado no canal de transmissão. O vetor *coeff* guarda os coeficientes, que são gerados aleatoriamente, para serem usados pelo RLNC no processo de codificação de cada pacote. Por cada pacote original é necessário um coeficiente, daí este vetor ter o tamanho do número de pacotes. Por fim, o vetor *hash* vai conter o CRC externo de cada pacote codificado e apresenta tamanho *size_crc* que na verdade corresponde a 4 bytes.

```
1  std::vector<uint8_t> data_in(encoder.block_size());
2
3  // Just for fun - fill the data with random data
4  std::generate(data_in.begin(), data_in.end(), rand);
5
6  // Assign the data buffer to the encoder so that we may start
7  // to produce encoded symbols from it
8  encoder.set_const_symbols(data_in.data(), encoder.block_size());
9
10 // Set the storage for the decoder
11 std::vector<uint8_t> data_out(decoder.block_size());
12 decoder.set_mutable_symbols(data_out.data(), decoder.block_size());
13
14
15 std::vector<uint8_t> coeff(max_symbols);
16 std::vector<uint8_t> symbol_data(max_symbol_size);
17
18 // CRC variables
19 CryptoPP::CRC32 crc;
20 std::size_t size_crc = crc.DigestSize();
21 std::vector<byte> hash(size_crc);
```

Bloco de Código 4.2: Criação dos buffers

Terminado a criação dos buffers onde serão guardados os dados e enviados no canal de transmissão podemos começar com o processo de codificação dos pacotes (bloco de código 4.3). Este processo ocorre num ciclo (*decoder.is_complete()*) que termina quando o descodificador tiver completado o processo corretamente, ou seja, quando a matriz do descodificador tiver o rank completo.

O rank de uma matriz em codificação em rede, corresponde ao número de pacotes linearmente independentes que essa matriz tem. Com isto, o rank de um codificador indica quantos pacotes estão disponíveis para codificar e o rank de um descodificador indica quantos pacotes foram descodificados corretamente. Uma causa de sobrecarga no processo de codificação em rede é mesmo a criação de pacotes linearmente dependentes. Isto acontece devido à escolha de um tamanho corpo finito demasiado baixo, o que faz com que a probabilidade de escolher um coeficiente aleatório já escolhido na codificação de algum pacote anterior seja alta. Esta sobrecarga não é aconselhada dado que o descodificador vai estar a guardar na sua matriz pacotes linearmente dependentes sem necessidade, e por outro lado vão ser enviados através do canal de comunicação mais pacotes aumentando a sobrecarga da rede.

Cada iteração deste ciclo codifica um pacote usando o protocolo RLNC com coeficientes que são gerados aleatoriamente. Para gerar estes coeficientes aleatórios é usada novamente a função *generate()* do C++. O codificador vai codificar cada pacote para o vetor *symbol_data* com os coeficientes presentes no vetor *coeff*. Esta ação é feita na chamada de *write_symbol()*.

O CRC externo do pacote codificado é criado de seguida com a chamada da função *crc.Update()* da biblioteca Cryptopp. O CRC é calculado sobre o pacote codificado e também sobre os coeficientes desse pacote e posteriormente adicionado no vetor *hash*. O vetor *hash* tem tamanho de 4 bytes pois é um tamanho já definido pela biblioteca Cryptopp e que para esta aplicação definimos que se manteria este valor, podendo no entanto ser alterado se necessário. Todos os pacotes iniciais, ou seja, os *max_symbols* podem eventualmente ter erros acrescentados voluntariamente com uma certa probabilidade p definida pelo utilizador. No final de cada pacote sofrer ou não modificações é calculado novamente um CRC sobre esse mesmo pacote para definir se ele vai ser enviado para o descodificador normal ou para o descodificador PRAC. Caso os CRC calculados sejam os mesmos, isto é, não foram acrescentados erros, esse pacote é adicionado no descodificador normal pela chamada *read_symbol()*. Caso contrário o pacote é enviado para o descodificador PRAC pela função *prac_decoder.receive_entry*. Esta função recebe, tanto os pacotes sem adição de erros como os pacotes com adição de erros e vai ser explicada mais à frente. Quando a função *prac_decoder.receive_entry* receber *max_symbols* mais n, onde n corresponde ao número de pacotes que vão ser enviados adicionalmente definido pelo utilizador, o descodificador PRAC é accionado e então começa a fase de detetar e corrigir os erros introduzidos.

```
1 while (!decoder.is_complete())
2     {
3         //Fill coefficients at random
4       std::generate(coeff.begin(), coeff.end(), rand);
5
```

```
6         // Encode with specified coefficients
7       encoder.write_symbol(&symbol_data[0], &coeff[0]);
8
9       // CRC32 compute
10      crc.Update(&symbol_data[0], N);
11      crc.Update(&coeff[0], max_symbols);
12      crc.TruncatedFinal(&hash[0], size_crc);
13
14        if(n>0)
15        erros(symbol_data, coeff, hash,max_symbols,max_symbol_size,probabilidade
              );
16      n--;
17
18      // CRC32 re-compute
19      crc.Update(&symbol_data[0], N);
20      crc.Update(&coeff[0], max_symbols);
21      crc.TruncatedFinal(&hash_after[0], size_crc);
22
23      std::vector<uint8_t> tmp_symbol;
24      std::vector<uint8_t> tmp_coeff;
25      std::vector<uint8_t> tmp_hash;
26
27      tmp_symbol.reserve(symbol_data.size());
28      std::copy(symbol_data.begin(),symbol_data.end(),std::back_inserter(
            tmp_symbol));
29
30      tmp_coeff.reserve(coeff.size());
31      std::copy(coeff.begin(),coeff.end(),std::back_inserter(tmp_coeff));
32
33      tmp_hash.reserve(hash.size());
34      std::copy(hash.begin(), hash.end(),std::back_inserter(tmp_hash));
35
36      // Check if hashes are the same
37      if(hash != hash_after)
38        std::cout << "CRC is not the same" << std::endl;
39      else
40        // Decode with specified coefficients
41        decoder.read_symbol(&symbol_data[0],&coeff[0]);
42
43      // RUN PRAC decoder
44      bool stat_prac = prac_decoder.receive_entry(tmp_symbol, tmp_coeff,
            tmp_hash );
45
46      if(stat_prac == true)
47        {
48          bool success_prac = prac_decoder.decode();
49          if(success_prac == true){
50            std::cout << "PRAC Decoder: Finished" << std::endl;
51          }
52        }
53    }
```

Bloco de Código 4.3: Processo de codificação dos pacotes

4.2.1 Introdução de erros

É na função *erros* que são adicionados erros nos pacotes (Figura 4.3) . Esta função (bloco de código 4.4) recebe como input os pacotes codificados e seus coeficientes, o seu CRC e também a probabilidade de erro em cada byte definida pelo utilizador. O objetivo desta função é percorrer cada pacote codificado, e para cada posição gerar um valor aleatório entre 0 e 1. Caso este valor aleatório seja inferior à probabilidade p definida pelo utilizador, então essa posição do pacote vai sofrer alteração. Para isso, é gerado um valor aleatório pertencente ao corpo finito escolhido inicialmente, e esse valor é alterado na posição correspondente. Os coeficientes associados a cada pacote não sofrem alterações nem como o seu CRC. Caso os coeficientes sofressem alterações, o descodificador PRAC não iria conseguir descodificar os pacotes corretamente. No caso de o CRC de cada pacote sofresse alteração também não seria possível marcar o pacote como válido ou inválido, processo este, que vai ser explicado mais à frente na função *receive_entry()*.

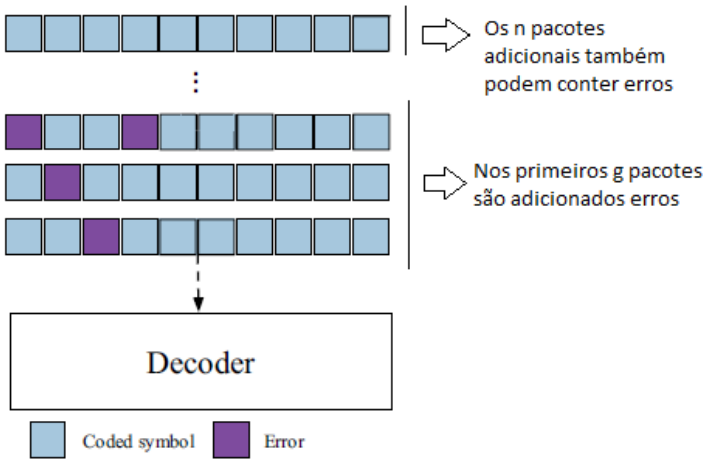


Figura 4.3: Introdução de erros [10]

```
1  void erros(std::vector<uint8_t>& codedsymbols,
2        std::vector<uint8_t>& coefficients,
3        std::vector<byte>& hash,
4        std::size_t pack_size,std::size_t sym_size,float prob)
5  {
6      uint32_t symbol_size = codedsymbols.size();
7      uint32_t size_coeff = coefficients.size();
8      std::size_t size_crc = hash.size();
9
10     assert(pack_size <= symbol_size + size_coeff + size_crc );
11     assert(symbol_size > 0);
12     assert(size_coeff > 0);
```

```
13      assert(size_crc);
14
15      uint8_t value_change;
16      uint32_t symbol_to_change;
17      float r;
18
19        for(uint32_t i=0;i<sym_size;i++){
20         r=(double)rand()/(RAND_MAX+1.0);
21
22         if(r<=prob){
23
24              value_change = rand()%256;
25              symbol_to_change = i;
26              codedsymbols[symbol_to_change] = value_change;
27              num_erros++;
28      }
29      }
30      return;
31  }
```

Bloco de Código 4.4: Função para introduzir erros

4.2.2 Método receive_entry

O método *receive_entry()* (bloco de código 4.5) tem como objetivo receber pacotes codificados e verificar se esses pacotes contém ou não erros. Os argumentos de input deste método são os pacotes codificados (*symbol_data*) juntamente com os seus coeficientes (*coeff*) e o *CRC* calculado sobre o pacote (*hash*). Inicialmente é calculado um CRC sobre o pacote que chega na função e comparado com o CRC *hash* que entra como input. Caso os CRC sejam iguais então o pacote codificado que chegou não contém erros e pode eventualmente ser adicionado a uma matriz de pacotes chamada *healthypackets* que vai conter todos os pacotes codificados sem erros. Para esse pacote ser adicionado na matriz, ele tem de ser linearmente independente e para isso, é testado se o rank do descodificador aumenta. Para saber se o rank do descodificador aumentou adicionasse o pacote codificado no descodificador normal e retornamos o valor do rank da matriz com a chamada da função do KODO *decoder.rank()*. Se o rank aumentar então o pacote é linearmente independente e pode ser adicionada na matriz juntamente com os seus coeficientes que são adicionados na matriz *healthycoeff* e o CRC que é adicionado na matriz *healthyhash*. Caso os CRC não sejam iguais, esse pacote contém erros, e então o pacote juntamente com os coeficientes e o CRC são adicionados nas matrizes *brokenpackets*, *brokencoeff* e *brokenhash*, respetivamente. A função termina de receber pacotes quando o número de pacotes que recebeu seja igual a $max\_symbols + n$, onde $max\_symbols$ corresponde ao tamanho da geração e o n o número de pacotes a enviar adicionalmente. O descodificador PRAC terá de ter armazenado, no mínimo, $max\_symbols + 1$ pacotes para começar a o processo de correção de erros.

```
1 bool PracDecoder::receive_entry(std::vector<uint8_t> & symbol_data, std::vector<
```

```
    uint8_t> & coeff, std::vector<uint8_t> & hash )
{
    CryptoPP::CRC32 crc;
   std::size_t size_crc = crc.DigestSize();
   std::vector<byte> hash_after(size_crc);

   crc.Update(&symbol_data[0], symbol_size);
   crc.Update(&coeff[0], symbols);
   crc.TruncatedFinal(&hash_after[0], size_crc);

   if(hash == hash_after)
   {
      uint32_t decoder_rank = decoder.rank();

      std::vector<uint8_t> tmp_symbol = {};
      std::vector<uint8_t> tmp_coeff = {};

      std::copy(symbol_data.begin(),symbol_data.end(),std::back_inserter(
          tmp_symbol));
      std::copy(coeff.begin(),coeff.end(),std::back_inserter(tmp_coeff));
      decoder.read_symbol(&tmp_symbol[0], &tmp_coeff[0]);

      // If increased rank, then add to list of healthy packets
      if(decoder_rank < decoder.rank())
      {
         healthypackets.push_back(symbol_data);
         healthyhash.push_back(hash);
         healthycoeff.push_back(coeff);
         healthy_packets_count ++;
         total_packets++;
      }
   }
   else
   {
      //Store as part of broken packets
      brokenpackets.push_back(symbol_data);
      brokenhash.push_back(hash);
      brokencoeff.push_back(coeff);
      damaged_packets_count++;
      total_packets++;
   }

   if(total_packets >= symbols+n)
      return true;
   else
      return false;
}
```

Bloco de Código 4.5: Função receive_entry

4.2.3 Método decode

Assumindo que o número de pacotes recebidos no descodificador PRAC seja superior ao tamanho da geração (g), ou seja, $max\_symbols + n > g$, o descodificador PRAC pode começar o processo de deteção e correção dos erros nos pacotes corrompidos.

O descodificador PRAC está dividido em duas fases. Na primeira fase são descobertas as localizações dos erros nos pacotes codificados e a segunda fase são feitas as permutações nessas posições de forma a corrigir esses erros.

4.2.3.1 Localizar erros

Nesta fase (bloco de código 4.6) começamos por criar um descodificador temporário (*tmp_decoder*) com o seu vetor de output associado (*tmp_data_out*). Primeiro inserimos apenas os pacotes que não continham erros pertencentes à matriz *healthypackets*. De seguida adicionamos os pacotes com erros até o descodificador contiver os *max_symbols* pacotes, ou seja, estiver com *full rank*. O descodificador temporário vai descodificar os dados para o buffer de output e então vai ser feita uma re-codificação para encontrar a localização dos erros. Para isso os dados do buffer de output são copiados para o vetor *data_in* onde o codificador vai voltar a ler os dados para codificar. É criado um vetor *tmp_encoding* para armazenar o pacote a ser codificado com os coeficientes específicos. Após a recodificação desse pacote é feita uma comparação, posição a posição, entre esse novo pacote e o último pacote que chegou no descodificador, o pacote $g + 1$. Sempre que o resultado da comparação seja diferente, então existe um erro nessa posição, e é adicionada a um vetor que guarda as posições com erros *tmp_broken_locations*. Desta forma temos guardadas as posições onde ocorreram a introdução de erros nos pacotes codificados.

```
1
2   //Step 1.- Let's detect locations of errors
3   //         Insert first the healthy packets first
4   //     A.--Decode data, mostly with correct packets
5   std::vector<uint8_t> tmp_pkts;
6   std::vector<uint8_t> tmp_coeff;
7   std::vector<uint8_t> tmp_hash;
8
9   for(uint32_t i = 0; i < healthy_packets_count; i++)
10  {
11      tmp_pkts = healthypackets[i];
12     tmp_coeff = healthycoeff[i];
13     tmp_decoder.read_symbol(&tmp_pkts[0], &tmp_coeff[0]);
14  }
15
16  uint32_t tmp_rank = tmp_decoder.rank();
17
18  uint32_t limit = symbols - tmp_rank;
19
20  for(uint32_t i = 0; i < limit; i++)
```

```
21      {
22          if(limit > 0)
23          {
24              tmp_pkts = brokenpackets[i];
25              tmp_coeff = brokencoeff[i];
26              tmp_decoder.read_symbol(&tmp_pkts[0], &tmp_coeff[0]);
27          }
28      }
29
30      //      B.--Re-encode to match one of the broken ones
31      std::copy(tmp_data_out.begin(),tmp_data_out.end(),std::back_inserter(data_in)
            );
32
33      encoder.set_const_symbols(data_in.data(), encoder.block_size());
34
35      // Encode with specified coefficients: only first broken packet outside range
36      std::vector<uint8_t> tmp_encoding(symbol_size);
37
38      tmp_coeff = brokencoeff[limit];
39      encoder.write_symbol(&tmp_encoding[0], &tmp_coeff[0]);
40
41      std::vector<uint8_t> tmp_broken = brokenpackets[limit];
42
43      //This one is to store the broken locations
44      std::vector<uint32_t> tmp_broken_locations = {};
45
46      for(uint32_t i = 0; i < symbol_size; i++)
47      {
48          if(tmp_encoding[i] != tmp_broken[i])
49          {
50              tmp_broken_locations.push_back(i);
51          }
52      }
53  }
```

Bloco de Código 4.6: Localização das posições com erros

4.2.3.2 Corrigir erros

Com as localizações dos erros já detetadas, é altura de corrigir essas posições nos pacotes com erros (bloco de código 4.7). No PRAC só é possível corrigir quando o número de erros é inferior ou igual a 4, pois com 5 ou mais erros iria demorar tempo indeterminado. Para isso vamos testar todos os valores possíveis para um máximo de 4 posições com erros. A variável *val_lim* vai limitar os valores possíveis em hexadecimal conforme esses valores sejam 1,2,3 ou 4 erros. Por exemplo para 1 erro teríamos *val_limit*=0x000000FF, onde 0x000000FF seria o valor hexadecimal correspondente a 255. neste caso vamos testar na posição com erro todos os valores possíveis entre 0 e 255. A mesma forma se aplica para 2,3 e 4 erros. Depois de ter definidos os limites de valores a permutar é feito um bitmask, bit a bit, entre todos os valores possíveis do *val_limit*

e o *val_limit* definido. Estes valores são alterados em todos os pacotes com erros e em cada permutação é calculado o CRC externo desse pacote de forma a saber se a permutação de valores nas posições erradas permite corrigir os erros ou não. Caso o CRC externo calculado seja igual ao CRC externo calculado previamente do pacote, então a permutação correta foi descoberta e esse pacote corrigido é adicionado ao descodificador para fazer a descodificação dos dados originais.

```
1   for(uint32_t i = 0; i < damaged_packets_count; i++)
2     {
3       uint32_t nb = tmp_broken_locations.size();
4       //for now, break if there are more than 4 broken locations
5       assert(nb < 5);
6       uint32_t val_limit = 0;
7
8       switch (nb)
9         {
10        case 1:
11         val_limit = 0x000000FF;
12         break;
13        case 2:
14         val_limit = 0x0000FFFF;
15         break;
16        case 3:
17         val_limit = 0x00FFFFFF;
18         break;
19        case 4:
20         val_limit = 0xFFFFFFFF;
21         break;
22        default:
23         std::cout << "Stop - nb is " << nb << std::endl;
24        }
25
26      std::vector<uint8_t> bpkts(4); // 4 broken bytes
27      tmp_pkts  = brokenpackets[i];
28      tmp_coeff = brokencoeff[i];
29      tmp_hash  = brokenhash[i];
30      for(uint32_t val = 0; val <= val_limit; val++)
31        {
32        uint32_t remap = val;
33        // Remap to vector of the size of 'nb'
34        remap = val & 0x000000FF;
35        bpkts[0] = (uint8_t) (remap);
36        remap = val >> 8;
37        remap = remap & 0x000000FF;
38        bpkts[1] = (uint8_t) (remap);
39        remap = val >> 16;
40        remap = remap & 0x000000FF;
41        bpkts[2] = (uint8_t) (remap);
42        remap = val >> 24;
43        remap = remap & 0x000000FF;
44        bpkts[3] = (uint8_t) (remap);
```

```
45
46          for(uint32_t j = 0; j < nb ; j++)
47            tmp_pkts[tmp_broken_locations[j]] = bpkts[j];
48
49          crc.Update(&tmp_pkts[0], symbol_size);
50          crc.Update(&tmp_coeff[0], symbols);
51          crc.TruncatedFinal(&hash_after[0], size_crc);
52
53          if(tmp_hash == hash_after)
54            {
55              decoder.read_symbol(&tmp_pkts[0], &tmp_coeff[0]);
56              break;
57            }
58          }
```

Bloco de Código 4.7: Corrigir erros PRAC

4.3 Implementação DAPRAC

Na figura 4.4 está presente o diagrama de estados do programa DAPRAC. No estado *Start* o programa é compilado e xecutado e então começa a codificação dos pacotes a serem transmitidos no método *decoder.iscomplete()* da biblioteca KODO. Todos os pacotes passam por uma função que pode adicionar erros nas posições dos pacotes com uma certa probabilidade definida pelo utilizador no estado *Add errors*, enquanto que o restante pacote adicional é enviado sem erros para o descodificador DAPRAC. O método *receive_entry()* garante que o descodificador DAPRAC recebeu $g+1$ pacotes codificados e assim pode avançar para a correção dos erros e descodificação dos pacotes. Isto acontece no método *decode()*. No método *decode()* do DAPRAC vai ser necessário calcular quantos descodificadores vamos ter bem como quais são as combinações de pacotes que vamos colocar em cada descodificador como explicado em 3.2.2. Para isso vamos usar o método *fact()* que vai calcular o fatorial de um número a ser usado em ${}^{g+1}C_n = g+1$ para calcular o número de descodificadores que serão necessários. O método *makeCombi()* retorna todas as combinações possíveis de pacotes para serem introduzidos nos diferentes descodificadores e o método *convert()* permite converter um número decimal para qualquer base passada como argumento e é usado para calcular todas as permutações possíveis que vão ser feitas nas posições com erros dos pacotes corrompidos. Caso os dados sejam corrigidos corretamente o programa avança para o estado em que usa o método *recover_decoded_data()* para descodificar os dados para um vetor de output que será usado para comparar com os dados originais que foram enviados inicialmente. Caso contrário a descodificação usando o algoritmo DAPRAC falha. Por norma a comparação do vetor de output escrito pelo DAPRAC com o vetor que continha os dados originais é igual e então os dados foram corrigidos e descodificados corretamente e são apresentados para o utilizador final.

Na figura 4.5 está ilustrado o diagrama da classe do DAPRAC. De notar que os atributos *data_outs* e *decoders* diferem do PRAC pois o algoritmo usado no processo de deteção e correção

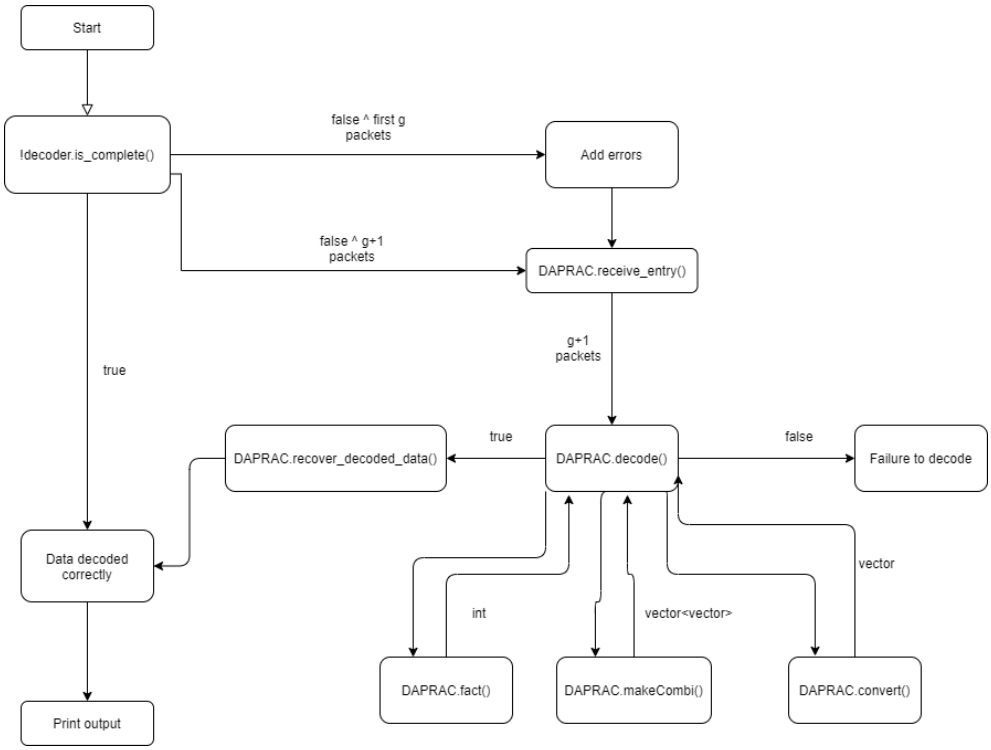


Figura 4.4: Diagrama de fluxos DAPRAC

dos erros é diferente. No DAPRAC não é usado apenas um descodificador mas sim vários, sendo este valor obtido do cálculo de ${}^{g+1}C_g$, onde g corresponde ao número de pacotes a transmitir. Assim temos um vetor de descodificadores e cada descodificador vai escrever os dados descodificados em um vetor *data_out* diferente, daí existir também um vetor de *data_outs*, onde cada *data_out* está associado a um descodificador diferente. O método *printHex()* não foi falado no fluxo anterior mas serve apenas para escrever o estado de cada pacote. Recebe como input o pacote codificado e os seus coeficientes usados no processo de codificação mais uma variável booleana a indicar se o pacote está corrompido ou não. se estiver corrompido escreve o pacote em hexadecimal seguido de "B", caso contrário escreve o pacote em hexadecimal seguido de "G".

De seguida são detalhados os métodos falados anteriormente a nível do código, bem como alguns outros processos que seria necessário explicar como é o caso da criação dos codificadores e dos descodificadores, da criação dos vetores de input e de output dos dados e do processo de codificação do programa DAPRAC que inclui a criação de CRC internos.

Tanto o codificador como o descodificador do DAPRAC vão receber argumentos diferentes dos que foram introduzidos no PRAC (bloco de código 4.8). No argumento referente ao tamanho do pacote é passado *max_symbol_size + size_crc*, isto porque antes do processo de codificação dos pacotes vai ser adicionado um CRC interno ao pacote de tamanho *size_crc*. Este CRC interno vai ser calculado sobre os dados do pacote antes de codificar.

```
1     kodocpp::encoder_factory encoder_factory(
2         kodocpp::codec::full_vector,
```

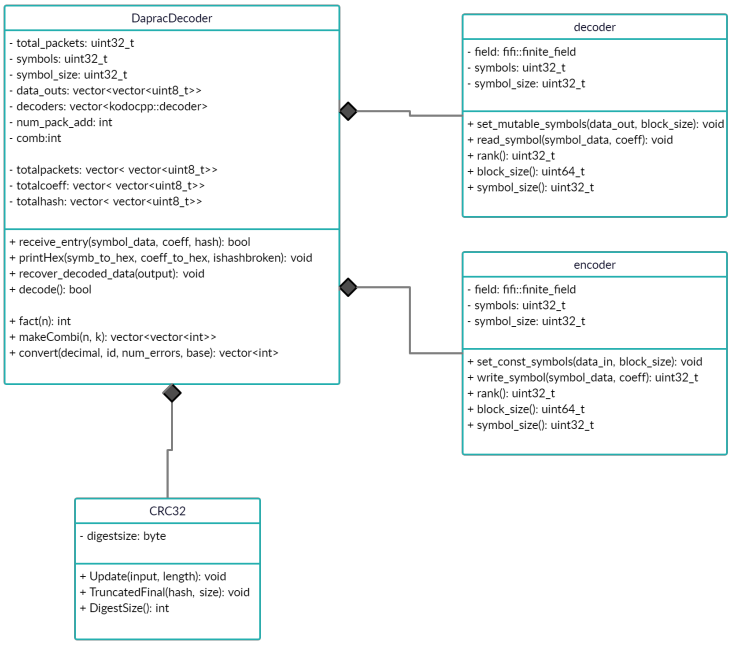


Figura 4.5: Diagrama de classes DAPRAC

```
3         kodocpp::field::binary8,
4         max_symbols,
5         max_symbol_size+size_crc);
6
7     kodocpp::encoder encoder = encoder_factory.build();
8
9     kodocpp::decoder_factory decoder_factory(
10        kodocpp::codec::full_vector,
11        kodocpp::field::binary8,
12        max_symbols,
13        max_symbol_size+size_crc);
14
15    kodocpp::decoder decoder = decoder_factory.build();
```

Bloco de Código 4.8: Inicializar codificador e descodificador DAPRAC

A criação dos buffers de leitura e escrita dos pacotes está definido no bloco de código 4.9. Tanto o buffer *data_in* como o *data_out* vão ter tamanhos diferentes dos buffers criados no PRAC. Isto porque no PRAC cada pacote tinha tamanho *max_symbol_size* mas no DAPRAC cada pacote vai ter tamanho *max_symbol_size* + *size_crc* pois vai ser adicionado um CRC interno ao pacote. Assim cada buffer terá o tamanho de o número de pacotes multiplicado pelo tamanho de cada pacote que será *(max_symbol_size+size_crc)*max_symbols*. Os buffers *data_in* e *data_out* serão atribuídos ao codificador e descodificador pelas chamadas das funções KODO *encoder.set_const_symbols()* e *decoder.set_mutable_symbols()* respetivamente e da mesma forma que foi feito anteriormente. O vetor *coeff* guarda os coeficientes aleatórios que são gerados no processo de codificação e o vetor *symbol_data* vai ser onde o codificador vai

codificar cada pacote. Este último vetor tem então o tamanho de um pacote que no DAPRAC é *(max_symbol_size+size_crc)*, pelos motivos já falados anteriormente.

```
1    std::vector<uint8_t> data_in((max_symbol_size+size_crc)*max_symbols);
2    encoder.set_const_symbols(data_in.data(), encoder.block_size());
3
4    std::vector<uint8_t> data_out((max_symbol_size+size_crc)*max_symbols);
5    decoder.set_mutable_symbols(data_out.data(), decoder.block_size());
6
7    std::vector<uint8_t> coeff(max_symbols);
8    std::vector<uint8_t> symbol_data(max_symbol_size+size_crc);
```

Bloco de Código 4.9: Criação dos buffers DAPRAC

O processo de adicionar o CRC interno aos pacotes está apresentado no bloco de código 4.10. Neste processo cada pacote é preenchido com valores aleatórios usando a função *generate()* do C++. De seguida é calculado o CRC sobre esses dados do pacote com a função *crc.Update()* do Cryptopp. Depois de termos o pacote com dados e o seu CRC, estes dois conjuntos de dados são adicionados ao buffer *data_in* por esta ordem específica, ou seja, primeiro é adicionado o pacote e de seguida o seu CRC. Assim o codificador ao ler do buffer *data_in* vai ler o pacote mais o CRC como sendo um pacote só e fazer a sua codificação. Este processo é realizado aos restantes pacotes até o buffer estar totalmente preenchido.

```
1    for (uint32_t i=0;i<max_symbols;i++){
2
3      std::generate(tmp_packet.begin(), tmp_packet.end(), rand);
4
5      crc.Update(&tmp_packet[0],max_symbol_size);
6      crc.TruncatedFinal(&tmp_crc[0],max_symbol_size);
7
8      std::copy(tmp_packet.begin(),tmp_packet.end(),std::back_inserter(data_in))
           ;
9      std::copy(tmp_crc.begin(),tmp_crc.end(),std::back_inserter(data_in));
10
11   }
```

Bloco de Código 4.10: Adicionar CRC interno aos pacotes

O processo de codificação é semelhante ao do PRAC. Este processo começa com a geração dos coeficientes aleatórios e a multiplicação desses coeficientes por cada pacote usando o protocolo RLNC para produzir as combinações lineares. De seguida é calculado um CRC externo a cada pacote codificado, os pacotes codificados podem eventualmente sofrer adição de erros da mesma forma do PRAC (4.2.1) e posterior a isso é calculado novamente um CRC externo do pacote para saber se este sofreu ou não alteração de bytes. Desta forma é possível marcar o pacote como válido ou inválido caso este sofra adição de erros ou não. Os pacotes válido são adicionados ao descodificador normal e ao descodificador DAPRAC enquanto que os inválidos são adicionados ao descodificador DAPRAC através da função *receive_entry*. Esta função é semelhante à existente

no PRAC e permite receber $g + n$ pacotes para o descodificador DAPRAC começar, onde g corresponde ao tamanho da geração e n o número de pacotes adicionais que são enviados.

4.3.1 Método receive_entry

O método *receive_entry* (bloco de código 4.11) vai receber os pacotes codificados e guardar esses pacotes em uma matriz de pacotes *totalpackets*, juntamente com os seus coeficientes e o seu CRC. Neste método não interessa se os pacotes que chegaram estão ou não com erros pois o algoritmo de correção do DAPRAC é diferente do PRAC. Este método termina quando o número de pacotes recebidos for superior ao tamanho da geração g, ou seja, quando $max\_symbols + p >= g$, onde p é o número de pacotes enviados a mais. O descodificador DAPRAC só funciona quando tiver, no mínimo, $g + 1$ pacotes armazenados.

```
1  bool DapracDecoder::receive_entry(std::vector<uint8_t> & symbol_data, std::
       vector<uint8_t> & coeff, std::vector<uint8_t> & hash )
2  {
3    totalpackets.push_back(symbol_data);
4    totalhash.push_back(hash);
5    totalcoeff.push_back(coeff);
6
7    total_packets++;
8
9    if(total_packets >= symbols+p){
10     return true;
11   }
12   else
13     return false;
14 }
```

Bloco de Código 4.11: Função receive_entry

4.3.2 Método decode

O descodificador DAPRAC vai descodificar os pacotes codificados várias vezes usando todas as combinações de g pacotes dos $g+p$ pacotes disponíveis, ou seja, terá de existir c $= {}^{g+p}C_g$ descodificadores diferentes. Para isso criamos um vetor de descodificadores juntamente com c buffers *data_out* e atribuímos cada *data_out* a um descodificador diferente (bloco de código 4.12). Estes descodificadores vão descodificar os pacotes originais e dar origem aos pacotes descodificados que são chamados pacotes pré-descodificados. No final deste processo vamos ter c**max_symbols* pacotes descodificados. Estes pacotes tem a denominação p_i^{xy} como sendo o pacote pré-descodificado i usando a combinação dos pacotes codificados m_x e m_y com $x \neq y$. Neste processo existe um método (blco de código 4.13) que calcula todas as combinações c possíveis, e estas combinações serão usadas para atribuir cada pacote codificado a um descodificador.

Todos estes pacotes pré-descodificados são guardados numa matriz *matriz_packs* com tamanho c*$max\_symbols$.

```
std::vector<kodocpp::decoder> decoders;

int c=fact(max\_symbols+p) / (fact(p) * fact(max\_symbols));

for(uint32_t i=0;i<c;i++){

      decoders[i]=in_decoder_factory.build();
      data_outs[i]={};
      data_outs[i].resize(decoders[i].block_size());
      decoders[i].set_mutable_symbols(data_outs[i].data(),decoders[i].block_size
          ());
   }
```

Bloco de Código 4.12: Atribuir cada data_out a um descodificador

```
std::vector<std::vector<int>> makeCombi(int n, int k)
{
  std::vector<std::vector<int>> ans;
  std::vector<int> tmp;
    makeCombiUtil(ans, tmp, n, 0, k);
    return ans;
}

void makeCombiUtil(std::vector<std::vector<int>>& ans,
          std::vector<int>& tmp, int n, int left, int k)
{
    if (k == 0) {
        ans.push_back(tmp);
        return;
    }

    for (int i = left; i < n; ++i)
    {
        tmp.push_back(i);
        makeCombiUtil(ans, tmp, n, i + 1, k - 1);
        tmp.pop_back();
    }
}
```

Bloco de Código 4.13: Calcular todas combinações possíveis

4.3.2.1 Localizar erros

Para descobrir a localização dos erros (bloco de código 4.14) percorremos todas as posições de todos os pacotes pré-descodificados p_i^{xy} com i fixo da matriz *matriz_packs*. Para a mesma

posição vamos comparar os valores dessa posição em todos os pacotes pré-descodificados p_i^{xy}. Se os valores de todos os pacotes forem iguais, então essa posição não contém erros e avançamos para a próxima posição. Caso contrário, encontramos uma posição com erros e esse index é adicionado a um vetor que guarda todas as posições de erros *wrong_locations*.

```
1  std::vector<int> wrong_locations(num_erros);
2  std::vector<uint8_t> valores(1);
3
4  for(uint32_t i=0;i<matriz_packs[0].size();i++){
5         valores.clear();
6         for(uint32_t j=0;j<matriz_packs.size();j++){
7         if(j*(symbols)<matriz_packs.size()){
8
9             uint8_t v1=matriz_packs[0][i];
10            std::copy(matriz_packs[j*(symbols)].begin()+i,matriz_packs[j*(symbols
                  )].end()-symbol_size+i+1,std::back_inserter(valores));
11
12            if(v1!=valores[0]){
13            t1=1;
14            wrong_locations.push_back(i);
15             t1=0;
16
17            break;
18        }
19        valores.clear();
20     }
21     }
22 }
```

Bloco de Código 4.14: Descobrir localização de erros

4.3.2.2 Corrigir erros

Para corrigir as posições com erros (bloco de código 4.15), o DAPRAC vai permutar as posições com erros de todos os pacotes pré-descodificados, como ilustrado na Figura 3.3. O número de permutações possíveis será igual a c^{num_erros}, onde c são todas as combinações possíveis dos $g + p$ pacotes codificados e *num_erros* todos os erros que foram adicionados nos pacotes.

Para calcular essas permutações existe uma função *convert()* (bloco de código 4.16) que converte um número decimal numa certa base N. O número decimal será um valor i, onde $0 <= i < c^{num\_erros}$ e a base N será o valor de todas as combinações possíveis c. O resultado desta função *convert()* é um vetor que contém os index dos pacotes pré-descodificados na matriz *matriz_packs*.

Existe um conjunto de pacotes *choosen_packs* p_i^{xy} com x e y fixos, escolhidos aleatoriamente do conjunto dos pacotes pré-descodificados onde vão ser efetuadas as permutações. Em cada iteração i é calculada uma permutação e essa permutação é efetuada nas posições com erros dos

pacotes escolhidos. Após ser feita a permutação é calculado novamente o CRC interno sobre as primeiras *max_symbol_size* posições do pacote, isto porque agora o pacote tem tamanho *max_symbol_size* + *size_crc*. Caso o CRC calculado seja igual ao CRC presente no pacote então a permutação testada é a correta e o descodificador tem os pacotes descodificados corretamente.

```
1     uint32_t result=pow(c,num_erros);
2
3     for(uint32_t i=0;i<result;i++){
4
5        id==convert(i,id,num_erros,c);
6
7        for(uint32_t j=0;j<num_erros;j++){
8
9            vals=matriz_packs[id[j]*(symbols)][wrong_locations[j]];
10           vals_[0]=vals;
11           choosen_packs[0][wrong_locations[j]]=vals;
12    }
13
14       //comparar crc internos de cada permutacao
15       std::copy(choosen_packs[0].begin()+symbol_size-size_crc,choosen_packs[0].
             begin()+symbol_size,std::back_inserter(old_crc_in));
16
17       crc.Update(&choosen_packs[0][0],symbol_size-size_crc);
18       crc.TruncatedFinal(&tmp_crc_in[0],size_crc);
19
20      bool teste=std::equal(old_crc_in.begin(),old_crc_in.end(),tmp_crc_in.begin
            ());
21
22       if(teste==true){
23         std::cout<<"Pacote "<<i<<" Igual"<<std::endl;
24         break;
25       }
26       else
27         std::cout<<"Pacote "<<i<<" Diferente"<<std::endl;
28
29       old_crc_in.clear();
30    }
```

Bloco de Código 4.15: Permutações sobre os pacotes

```
1  std::vector<int> convert(int Decimal,std::vector<int> & id,int num,int base)
2  {
3     // Se decimal >0 converte
4     if (Decimal != 0) {
5       convertTo(Decimal,id,num,base);
6     }
7     else{
8        for(int i=0;i<id.size();i++)
9            id[i]=0;
10    }
```

```
11     return id;
12 }
13
14 void convertTo(int N,std::vector<int> & id,int num,int base)
15 {
16   int i=num-1;
17
18   while(N>0){
19
20     int cop=N;
21
22     // encontrar resto
23     // quando divide N por base
24     int x = N % base;
25     id[i]=x;
26     i--;
27     N /= base;
28   }
29 }
```

Bloco de Código 4.16: Método de converter decimal para base N

Tendo o DAPRAC descodificado os pacotes que à partida estão corretos, temos de garantir que esses resultados estão realmente corretos. Para isso vamos comparar o output obtido do DAPRAC com o buffer *data_in* que continha os dados iniciais. Caso esta comparação seja igual, então os dados corrigidos e descodificados pelo DAPRAC estão realmente corretos.

4.4 Instruções

4.4.1 Incluir kodo-cpp e kodo-c na aplicação

Primeiramente é necessário fazer o download do código fonte do repositório github da Steinwurf. Para isso basta fazer **git clone git@github.com:steinwurf/kodo-c.git** para descarregar o código kodo-cpp e **git clone git@github.com:steinwurf/kodo-rlnc-c.git** para descarregar os código kodo-c .

De seguida é necessário criar os exemplos KODO e os testes unitários com os seguintes comandos:

```
1 $ cd ~/kodo-cpp/
2 $ python waf configure
3 $ python waf build
4 $ python waf --run_tests
```

Bloco de Código 4.17: Crias exemplos KODO e testes unitários

O comando de configuração *waf* garante que todas as ferramentas necessários para o KODO

estão disponíveis e preparadas para criar o KODO. Este passo vai também descarregar algumas livrarias adicionais para uma pasta chamada *resolved_dependencies* dentro da pasta kodo. Este processo terá de ser repetido tanto para o kodo-cpp como também para o kodo-c.

Depois de ter os passos anteriores configurados com sucesso é necessário criar uma biblioteca partilhada com o seguinte comando, onde o *PATH_TO_DAPRAC* seria a localização da aplicação DAPRAC.

```
$ python waf install --install_shared_libs --install_path="PATH_TO_DAPRAC"
```

Bloco de Código 4.18: Criar shared library

Se o comando anterior correr OK, dentro da pasta da aplicação DAPRAC terá de estar um ficheiro com o nome *libkodo_c.so*. A biblioteca Cryptopp pode ser descarregada no site https://www.cryptopp.com/ e descompactada dentro da pasta da aplicação DAPRAC na pasta *include*. De seguida é necessário criar um ficheiro Makefile que será usado para fazer a compilação da aplicação. este ficheiro terá a seguinte intrução para compilação em C++:

```
main.o:
g++ daprac.cpp -g3 -ggdb -O0 -Wall -Wextra -Wno-unused -o daprac -lcryptopp -
    lpthread -std=c++11 -I./include -L. -Wl,-lkodoc \-Wl,-rpath .
```

Bloco de Código 4.19: Makefile

Por fim basta correr o ficheiro Makefile e correr o binário *daprac* criado na compilação. Todos os argumentos da aplicação são passados na linha de comandos, sendo os argumentos os seguintes:

1. Executar PRAC(1) ou DAPRAC(2)
2. Número de pacotes (tamanho da geração)
3. Tamanho de cada pacote (bytes)
4. Número de pacotes adicionais a enviar
5. Probabilidade de erro em cada byte de um pacote

Em relação ao número de pacotes, ter em atenção que um aumento deste número pode ter os seguintes efeitos:

- A complexidade computacional vai aumentar e pode diminuir o tempo na fase de codificação/descodificação
- No RLNC, o *overhead* por pacote vai aumentar devido ao aumento dos coeficientes de codificação

- O atraso de descodificação por pacote vai aumentar, isto porque quando aumentamos o número de pacotes codificados o descodificador precisa de receber mais pacotes para descodificar o bloco completo dos dados

Quando são criados o codificador e o descodificador é necessário passar também o tamanho do corpo finito conhecido também como GF. Um corpo finito é um corpo que contém um número finito de elementos onde regras especiais são definidas para as operações aritméticas. Estas regras garantem que qualquer que seja a operação aritmética, o resultado dessa operação é sempre um elemento desse corpo.

- Aumentando o corpo finito vai aumentar a probabilidade de sucesso na fase de descodificação
- Um aumento garante que exista uma menor probabilidade de serem escolhidos coeficientes aleatórios diferentes para codificar cada pacote
- Mas por outro lado um aumento do corpo finito vai aumentar também a complexidade computacional o que pode resultar em aplicações mais lentas

Em baixo temos um exemplo de como executar a aplicação daprac(2) com 3 pacotes originais com tamanho de 15 bytes, uma probabilidade de 5% de erro em cada byte de um pacote e a ser enviado mais um pacote adicional.

```
1 $ make
2 $ ./daprac 2 3 15 1 0.05
```

Bloco de Código 4.20: Executar DAPRAC

4.5 Resumo

Este capítulo apresenta as bibliotecas de programação usadas na implementação das duas técnicas na Sec. 4.1. De seguida são apresentados os detalhes da implementação da técnica PRAC (Sec. 4.2) e da técnica DAPRAC (Sec. 4.3). Aqui são apresentados os principais métodos usados no processo de codificação dos dados, na introdução de erros, na localização dos erros e por fim na correção desses erros. Por fim na Sec. 4.4 é explicado quais os requisitos a instalar para o funcionamento do programa.

Capítulo 5

Avaliação de desempenho

Neste capítulo são efetuados os testes de performance que permite comparar os resultados obtidos entre o programa Packetized Rateless Algebraic Consistency (PRAC) e o Data Aware Packetized Rateless Algebraic Consistency (DAPRAC). Na secção 5.1 é feita uma análise matemática para comparar o número de tentativas na fase de correção, o número de pacotes pré-descodificados e o número de descodificações entre os dois programas. Na secção 5.2 é feito uma comparação gráfica usando diferentes parâmetros. A secção 5.2.1 apresenta a comparação do tempo de descodificação em função da probabilidade de erro. Na secção 5.2.2 temos o tempo de descodificação e o tamanho da geração. Por fim, na secção 5.2.3, a percentagem da descodificação ser bem sucedida em função do número de pacotes que são enviados adicionalmente. Destas comparações obtemos os gráficos onde é possível observar as diferentes curvas para o PRAC, o DAPRAC e o descodificador Random Linear Network Coding (RLNC) normal.

A eficiência do algoritmo DAPRAC em relação ao algoritmo PRAC encontra-se principalmente na fase de correção dos erros. O DAPRAC permuta apenas um reduzido número de permutações nas posições com erros enquanto que o PRAC testa todas as permutações dos valores possíveis até encontrar uma solução. Contudo existem ainda alguns pontos fracos no algoritmo DAPRAC, nomeadamente não ser recomendado colocar erros nas mesmas posições em g pacotes dos $g+1$ recebidos, isto porque não iria ser encontrada uma permutação correta para a correção. Outro problema é que o *CRC* interno que vai adicionado aos pacotes também não dever conter erros pois assim seria impossível saber se o pacote recebido estaria ou não com erros e não seria possível corrigir.

5.1 Análise

1) ***Número de tentativas na correção:*** Na fase de correção, o número de possíveis permutações que o descodificador PRAC tenta depende do número de erros em posições diferentes e do tamanho do corpo finito. Sendo e o número de erros em diferentes posições ao longo dos pacotes o descodificador tem N possíveis tentativas até uma estar correta. O

valor de N para o PRAC é:

$$N_{PRAC} = q^e \tag{5.1}$$

onde q é o tamanho do corpo finito.

Por outro lado, o número de permutações que o descodificador DAPRAC necessita do número de combinações possíveis c e também do número de erros e, e o valor N será:

$$N_{DAPRAC} = c^e \tag{5.2}$$

2) ***Número de pacotes pré-descodificados:*** depois de o descodificador DAPRAC receber o número mínimo de pacotes para começar o processo de correção de erros que é g+1, o número de pacotes pré-descodificados criados é:

$$P_{pre} = g * (g+1) \tag{5.3}$$

3) ***Número de descodificações:*** : O algoritmo DAPRAC tem de efetuar um número fixo de descodificações que depende de g. Ele precisa de descodificar g pacotes de um conjunto de $g+1$ pacotes recebidos:

$$D_{DAPRAC} = \binom{g+1}{g} = g+1 \tag{5.4}$$

5.2 Comparação de performance

Os testes foram efetuados numa máquina virtual Ubuntu 20.04 LTS com 4 núcleos de Central Process Unity (CPU) e 5 GB de memória Random Access Memory (RAM). O código de codificação dos pacotes foi o RLNC. Os pacotes tinham tamanho de 15 bytes e um corpo finito Galois Field (GF)(2^8), ou seja, 256 elementos no algoritmo PRAC. No algoritmo DAPRAC os pacotes tinham 19 bytes (15 bytes de dados mais 4 bytes correspondentes ao Cyclic Redundancy Check (CRC) interno).

Os erros são adicionados nos primeiros g pacotes bem como nos pacotes que são enviados adicionalmente. A forma de adicionar os erros é a mesma para os dois algoritmos, existindo uma probabilidade de erro em cada byte definida pelo utilizador. Os valores apresentados nos gráficos resultam apenas de tempos em que a descodificação foi bem sucedida.

Em todos os descodificadores foi escolhido um tamanho de geração g={2,3,4,5,8}. No algoritmo PRAC foi escolhida uma probabilidade de erro, p={2%,5%}, e no DAPRAC, p={2%,5%,6%,8%,10%}. No PRAC não ultrapassamos a probabilidade de erro de 5% pois os erros introduzidos já eram

superiores a 3 na maioria das vezes e o tempo de descodificação do PRAC para 4 ou mais erros era muito elevado. Para 4 erros, nesta máquina virtual, demorava acima de 10 minutos. Os resultados foram medidos 100 vezes e foi calculado a média desses resultados.

5.2.1 Tempo de descodificação versus Probabilidade de erro

Como vemos no gráfico da figura 5.1, o tempo de descodificação do PRAC aumenta à medida que a probabilidade de erros aumenta, como seria de esperar. De notar que quando a probabilidade de erros aumenta de 2% para 5% o tempo aumenta de 50 ms para 2300 ms com g=3, ou seja, um aumento de cerca de 45 vezes mais. Acima de uma probabilidade de erro de 5% o descodificador PRAC vai encontrar demasiadas vezes 4 ou mais erros e o tempo de descodificação vai ser superior a 10 minutos na máquina testada. Assim esses valores foram retirados deste gráfico.

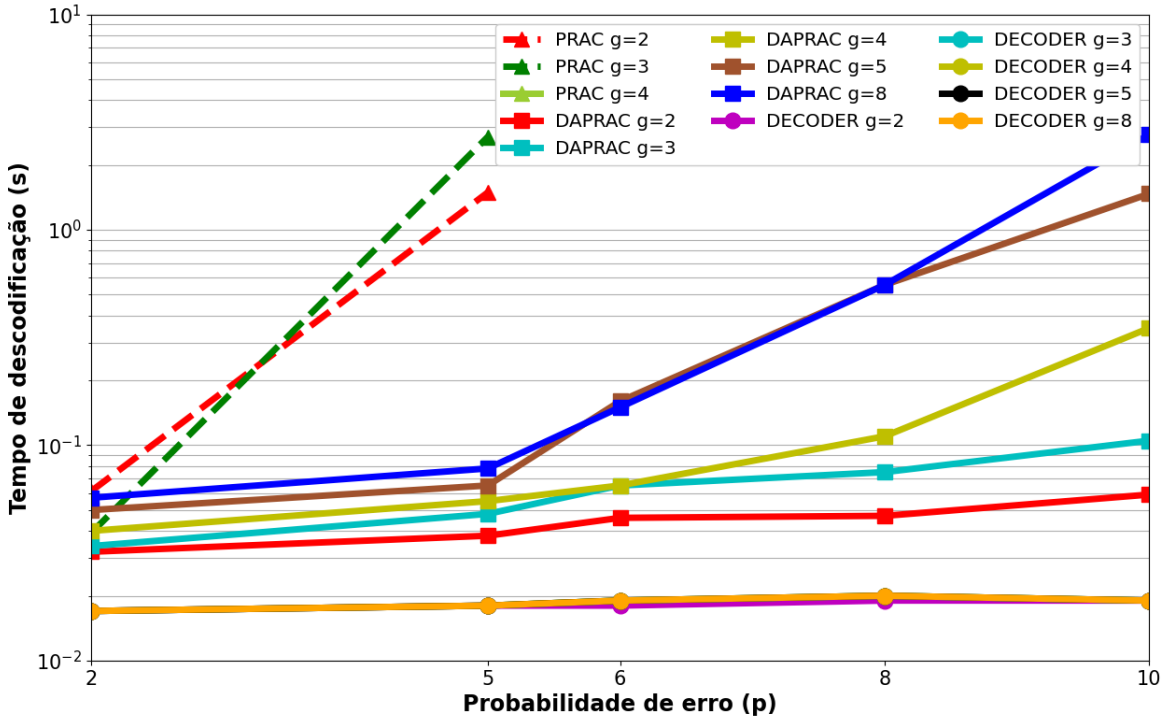


Figura 5.1: Tempo de descodificação versus Probabilidade de erro

As curvas do descodificador DAPRAC aumentam com o aumento do tamanho da geração como seria de esperar. Com g=5 e g=8 existe um aumento de cerca de 45 vezes mais do tempo de descodificação. No entanto para tamanhos de geração pequenos o DAPRAC continua a apresentar uma melhor performance em relação ao PRAC.

O descodificador RLNC normal mantém tempos constantes pois independente da probabilidade de erro este descodificador não vai fazer nenhuma permutação de valores mas sim receber novas retransmissões de pacotes até conseguir descodificar os dados corretos. No entanto a retransmissão de pacotes não é uma mais valia, pois iria aumentar a sobrecarga do sistema dependendo do número de erros.

5.2.2 Tempo de descodificação versus Tamanho da geração

No gráfico da figura 5.2 vemos os resultados do tempo de descodificação de cada descodificador em relação ao tamanho da geração. O tempo de descodificação é proporcional ao número de permutações que o descodificador faz para corrigir os erros. As curvas do PRAC para probabilidades de erros fixas praticamente não variam qualquer que seja o tamanho da geração.

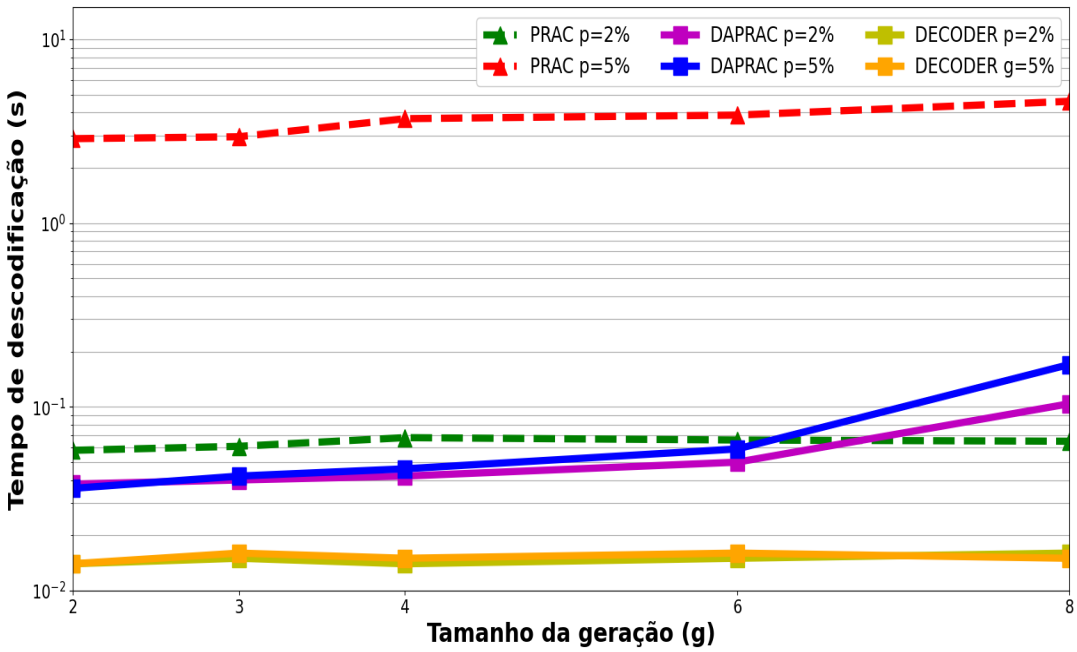


Figura 5.2: Tempo de descodificação Vs Tamanho da geração

No algoritmo DAPRAC o tempo tende a aumentar à medida que o número de pacotes aumenta. No entanto o tempo de descodificação do DAPRAC é inferior ao PRAC para probabilidades de erros de 5% nestes testes efetuados. Seria recomendado usar o descodificador DAPRAC em cenários em que temos bastantes erros e um baixo tamanho de geração.

O descodificador normal apresenta novamente valores baixos e constantes pois com um aumento dos pacotes recebidos mais fácil será a forma de obter os dados originais para uma probabilidade de erro fixa.

5.2.3 Percentagem de sucesso versus Pacotes adicionais

Um dos requisitos dos descodificadores PRAC e DAPRAC é receber um número mínimo de pacotes adicionais para poderem começar a deteção e correção dos erros. Este gráfico (figura 5.3) relaciona a percentagem de sucesso da descodificação dos vários descodificadores em relação a um aumento do número de pacotes enviados adicionalmente.

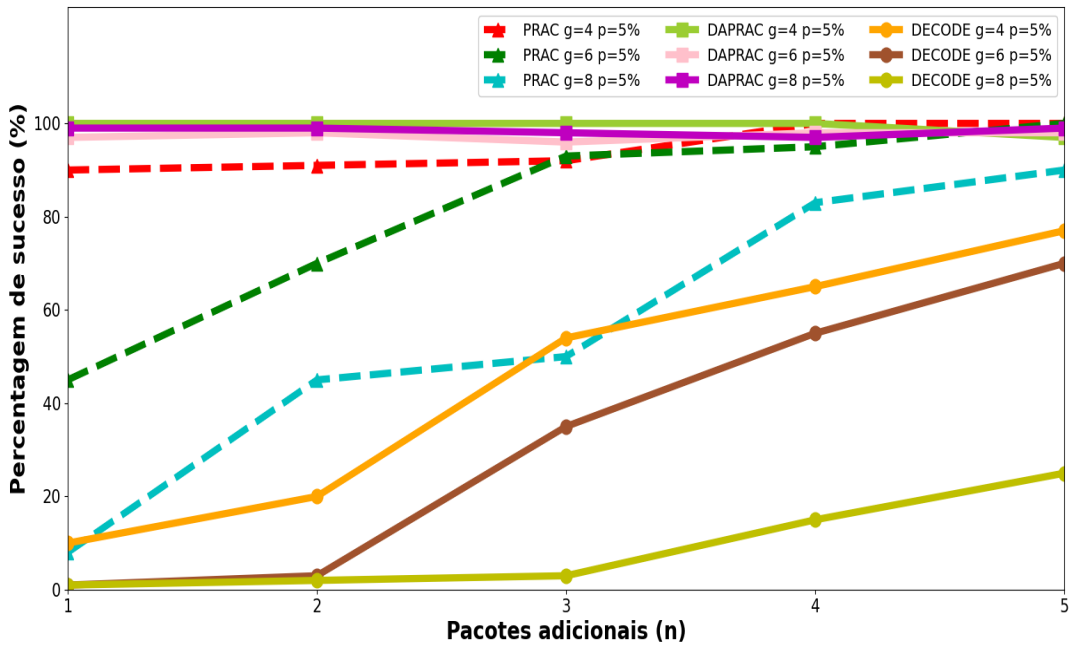


Figura 5.3: Percentagem de sucesso Vs Pacotes adicionais

O algoritmo DAPRAC apresenta uma percentagem de sucesso sempre perto de 100% pois o número de pacotes adicionais enviados não tem interferência no processo de correção. Este aumento de pacotes adicionais enviados no DAPRAC seria prejudicial, pois como já vimos um aumento do número de pacotes recebidos iria aumentar o tempo de descodificação.

O PRAC aumenta a sua percentagem de sucesso com um aumento do número de pacotes adicionais, pois a matriz de pacotes válidos será maior em relação aos pacotes corrompidos.

O descodificador normal estará sempre dependente de pacotes adicionais para g e p fixos. Para tamanhos de geração maiores a probabilidade de sucesso será baixa para um n baixo, isto porque será necessário mais pacotes adicionais para corrigir os erros que já existem nos g pacotes recebidos.

5.3 Resumo

Neste capítulo são apresentados os testes analíticos e de performance efetuados. Na Sec. 5.1 é feita uma analise analítica entre a técnica PRAC e DAPRAC. De seguida é feita uma comparação de performance baseada em várias métricas. Na Sec. 5.2.1 é comparado o tempo de descodificação em função da probabilidade de erro. Na Sec. 5.2.2 é comparado o tempo de descodificação em função do tamanho da geração. Por fim na Sec. 5.2.3 é calculada a percentagem de sucesso em função dos pacotes adicionais enviados.

Capítulo 6

Conclusões

Neste projeto implementámos um algoritmo de correção de erros em armazenamento distribuído recorrendo a códigos de *network coding*, nomeadamente o Random Linear Network Coding (RLNC). Esta técnica permite combinar os dados a enviar, num conjunto de pacotes codificados, recorrendo a combinações lineares usando coeficientes seleccionados aleatoriamente. Assim, o recetor pode fazer a descodificação e obter os dados originais sem ser necessário uma retransmissão de pacotes. Para isso usamos uma biblioteca em C++, KODO, que disponibiliza uma grande quantidade de algoritmos de *network coding* e tem a vantagem de apresentar elevada performance e flexibilidade. Este algoritmo Data Aware Packetized Rateless Algebraic Consistency (DAPRAC) desenvolvido permite que, mesmo que alguns pacotes codificados contenham erros, do lado do recetor seria possível na mesma obter os dados originais. Para isso era necessário que do lado do recetor se colocasse uma quantidade de pacotes acima do necessário para assim evitar retransmissões de dados ao longo dos sistemas distribuídos. Isto seria vantajoso pois quando falamos em sistemas distribuídos estamos a falar de grandes quantidades de dados, e que se fosse necessário uma retransmissão poderia levar demasiado tempo bem como a sobrecarga que iria se colocada no canal de comunicação.

Este algoritmo veio melhorar a performance de um outro algoritmo que já tinha sido implementado anteriormente com o mesmo objetivo. Esse algoritmo é o Packetized Rateless Algebraic Consistency (PRAC), mas como vimos o PRAC apresenta um elevado tempo de descodificação caso o número de erros nos pacotes de dados seja superior a 4, pois é um algoritmo que tenta descobrir as posições corretas dos erros através de força bruta.

Com o desenvolvimento do algoritmo DAPRAC concluiu-se que apresenta melhores níveis de performance em relação ao PRAC. Com o DAPRAC conseguimos descodificar bem mais do que 4 erros em tempo bem inferior ao PRAC e com gastos computacionais menores. No entanto o DAPRAC apresenta ainda alguns pontos a melhorar. O tempo de descodificação aumenta quando o número de pacotes recebidos também aumenta e este algoritmo não deve receber erros na mesma posição em mais do que g pacotes de um conjunto de $g + 1$ e nem no Cyclic Redundancy Check (CRC) interno, pois então a descodificação não iria acontecer.

Assim em trabalho futuro seria implementar um novo algoritmo mais eficiente (Segmented Packetized Rateless Algebraic Consistency (S-PRAC)[15]) que permite detetar e corrigir erros em ambientes com elevado ruído. A ideia do S-PRAC seria dividir o pacote em segmentos de tamanho igual e em cada segmento adicionar um CRC-8 e no final de cada pacote adicionava um CRC-32. Depois de descobrir a localização dos erros usando uma forma semelhante ao PRAC, o número de permutações vai ser feito apenas por segmentos e não no pacote completo reduzindo assim o tempo de descodificação.

Bibliografia

[1] Erasure coding. https://searchstorage.techtarget.com/definition/erasure-coding. Accessed: 2020-11-29.

[2] Crypto++ library 8.2. https://www.cryptopp.com/. Accessed: 2020-11-29.

[3] Understanding erasure coding and its difference with raid. https://stonefly.com/blog/understanding-erasure-coding. Accessed: 2020-12-13.

[4] Implementing the power of erasure correcting codes. https://www.steinwurf.com/products/kodo.html. Accessed: 2020-11-29.

[5] Raid (redundant array of independent disks). https://searchstorage.techtarget.com/definition/RAID, . Accessed: 2020-11-29.

[6] Raid levels explained. https://uk.pcmag.com/storage/7917/raid-levels-explained, . Accessed: 2020-11-29.

[7] Error detection techniques. https://www.faceprep.in/computer-networks/computer-networks-error-detection/. Accessed: 2020-11-29.

[8] Wei dai. http://www.weidai.com/. Accessed: 2020-11-29.

[9] Georgios Angelopoulos, Anantha P Chandrakasan, and Muriel Médard. Prac: Exploiting partial packets without cross-layer or feedback information. In *2014 IEEE International Conference on Communications (ICC)*, pages 5802–5807. IEEE, 2014.

[10] Juan Cabrera, Giang Nguyen, Daniel E Lucani, Morten Pedersen, and Frank HP Fitzek. Taking the trash back in: practical joint channel and network coding for improving ieee 802.11 networks. In *European Wireless 2017; 23th European Wireless Conference*, pages 1–5. VDE, 2017.

[11] CKP Clarke. R&d white paper. *Reed-Solomon error correction,"WHP*, 31, 2002.

[12] Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE transactions on information theory*, 56(9):4539–4551, 2010.

[13] Kerim Fouli, Muriel Médard, Kavim Shroff, and Code On Network Coding LLC. Coding the network: Next generation coding for flexible network operation. August 2018.

[14] Christina Fragouli, Jean-Yves Le Boudec, and Jörg Widmer. Network coding: an instant primer. *ACM SIGCOMM Computer Communication Review*, 36(1):63–68, 2006.

[15] Kurniawan D Irianto, Juan A Cabrera, Giang T Nguyen, Hani Salah, and Frank HP Fitzek. S-prac: fast partial packet recovery with network coding in very noisy wireless channels. In *2019 Wireless Days (WD)*, pages 1–7. IEEE, 2019.

[16] Katina Kralevska. Applied erasure coding in networks and distributed storage. *arXiv preprint arXiv:1803.01358*, 2018.

[17] Code On Network Coding LLC. Random linear network coding a tutorial. September 2013.

[18] Code On Network Coding LLC. Random linear network coding the next generation of coding. December 2013.

[19] Code On Network Coding LLC. Random linear network coding a technical feature overview. August 2018.

[20] Code On Network Coding LLC. Random linear network coding for storage and content delivery. September 2018.

[21] Sarah Edge Mann. The original view of reed-solomon coding and the welch-berlekamp decoding algorithm. 2013.

[22] SL Maskara and S Chakrabarti. Understanding error control coding. *IETE Journal of Education*, 35(1-2):3–21, 1994.

[23] Pouya Ostovari and Jie Wu. *Towards network coding for cyber-physical systems: Security challenges and applications*. Wiley, 2017.

[24] David A Patterson, Garth Gibson, and Randy H Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, 1988.

[25] James S Plank. Erasure codes for storage systems: A brief primer. *; login:: the magazine of USENIX & SAGE*, 38(6):44–50, 2013.