RNCDDS - RANDOM NETWORK CODED DISTRIBUTED DATA SYSTEM
ランダムネットワークコーディングによる分散データシステムとマルチメディアストリーミング

西田 博史 [1], *Thinh Nguyen*[2]

[1] ASUSA Corporation, [2] オレゴン州立大学計算機工学科

概 要

過去に数多くのランダムネットワークコーディングを用いたデータシステムが提唱され、データ保護における堅牢性とデータ保存容量における高効率性を証明してきた。しかしそれらには、(1) エンコードおよびデコードにがかかる時間の長さ、(2) エンコードされたデータへのアクセスの難しさ、という二つの問題が存在し、実用的な運用が困難であった。本稿で紹介する RNCDDS はこれらの問題を新たに開発した (1) 高速なガロア体演算ライブラリと (2) JavaScript プログラムを用いることによって解決している。その結果、RNCDDS は主要なデータシステムである Hadoop (HDFS) や GlusterFS よりも理論的に少ないデータ保存量で同等の堅牢さを実現するだけでなく、それらよりも高速にデータのアップロード、ダウンロードを行うようになっている。また JavaScript プログラムはエンコードされた動画データをウェブブラウザ上で直接 HTML5 の機能を用いて再生することを可能にし、クラウドシステムやコンテンツデリバリーネットワークにおけるデータ保存量を劇的に減少させることを可能にする。

キーワード: 分散システム、データストレージ、ランダムネットワークコーディング、有限体、ガロア体、マルチメディアストリーミング、コンテンツデリバリーネットワーク、クラウドシステム

1. 導入

膨大なデータの保存や半永久的なデータの保管といった目的のため、分散データシステムはクラウドシステムにおいて欠かせない存在となっている。その中でも Hadoop [1] はクラウド業界において実質上の標準システムとなっており、あらゆる企業に採用されているが、Hadoop で使用されているデータのサーバーへの分散方法はその単純さ故、データ保護における堅牢性とデータ保存量の節約を両立させることは困難であった。例えば Hadoop は図 1 のように一つのファイルを幾つかのブロックに区切り（図では三つ）、各ブロックの複製を複数のサーバーに分散する。ここで一クラスターにおいて R 台のサーバーの冗長性、つまり R 台のサーバーがアクセス不能の状況下においてもデータ取り出しが可能であることを保証するためには、Hadoop は本来の $1+R$ 倍のデータ保存量を使用しなくてはならない。例えば $R=2$、つまり二台のサーバーがアクセス不能に陥ってもデータ取り出しを保証する Hadoop クラスターにおいては、1GB のファイルを保存するために 3GB の容量を要することになる。同様のことは更に単純なデータ分散手法を用いる GlusterFS [2] 等にもあてはまる。GlusterFS は図 2 のように元のファイルの複製をそのままサーバーに分配するため、Hadoop 同様 R の冗長性において $1+R$ 倍の保存量を必要とする。これらの手法はその構造の単純さ故、様々なデータシステムに用いられている。

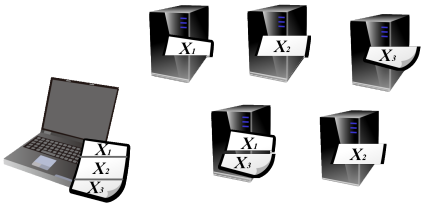


図 1: Hadoop はファイルを幾つかのブロックに区切り、それらの複製をサーバーに分散する。

図 2: GlusterFS はファイルの複製をサーバーに分散する。

一方、ランダムネットワークコーディング（RNC）を用いた分散データシステムは、R の冗長性に対してわずか $1+\frac{R}{3}$ 倍の保存量しか必要としない（第 3 章参照）。これ

は Hadoop や GlusterFS と比較し、$R = 2$ において 44%ものデータ保存量を節約することを意味する。しかしながらこれまでの RNC に基づいたデータシステムには以下のような問題点があった。

1. エンコードおよびデコードにかかる処理時間の長さ
2. クライアントからのデータへのアクセスの難しさ

我々の調査でエンコードおよびデコードにかかる所要時間は**ガロア体**（GF）における演算速度が大いに関係していることが判明したが、(1) 高速 (2) 特許に抵触しない (3) ライセンスの制限が緩やか、これらの条件を全て満たすガロア体演算ライブラリが存在しなかったため、それらにかかる時間を短縮することは非常に困難であった（第 3、4 章参照）。またこれまでの RNC を用いたシステムではユーザーは専用のクライアントプログラムの使用を余儀なくされてきたが、セキュリティの問題や専用のプログラムを立ち上げる不便さから、使用を躊躇するユーザーも少なくなかったと思われる。我々の開発した *RNCDDS* はこれらの問題を以下のプログラムで解決し、RNC を用いた分散データシステムとしては初めて実用的な段階に到達させることに成功したものと考えている。

1. *gf-nishida-16*: 新開発の高速な 16bit ガロア体演算ライブラリ
2. *RNC.js*: RNC でエンコードされたデータをウェブブラウザ上でダウンロード、デコードし、ブラウザの HTML5 動画再生機能を使って動画を再生する JavaScript プログラム

本稿では RNC の基礎の紹介、gf-nishida-16 の簡単な説明と合わせて RNCDDS の構造を解説し、RNCDDS の Hadoop や GlusterFS と比較した場合の長所、マルチメディアストリーミングにおける高効率性について述べる。

2. 関連研究

Hadoop や GlusterFS 以外にも過去に OpenAFS [3]、Ceph [4]、MooseFS [5]、Quantcast File System (QFS) [6]、Pyramid Codes [7] といった数多くの分散データシステムが提唱され、クラウドシステムにおいて重要な役割を担ってきた。それらのうち Hadoop、GlusterFS、OpenAFS、Ceph、MooseFS はデータを符号化（エンコード）せずにサーバーに分散するが、QFS、Pyramid Codes はデータをイレイジャーコーディング（EC）やそれに類似した方法で符号化しており、非符号化データを扱うシステムよりも堅牢性の向上とデータ保存量の節約の両立を実現させている。EC は RNC によく似たエンコード手法を用いるが、RNC ではエンコードされるデータ数にほぼ制限がないのに対し、EC では基本的にエンコード可能なデータ数に制限があり、マルチメディアストリーミングでの使用が限定されてしまう。RNC を基本としたデータシステムは主に P2P ネットワークの分野で提唱されており [8] [9] [10]、その高い堅牢性、柔軟性およびデータ保存量における効率の良さを示してきた。しかし我々の知りうる限り RNC を基本としたデータシステムにおいて、[11] を除きエンコード、デコードにおける処理速度に言及した論文はない。

RNC を用いたマルチメディアストリーミングに関しては、論文の多くが P2P ネットワークでの使用 [12] [13] [14] もしくは無線ネットワークでの使用 [15] [16] に焦点を当てており、クライアント・サーバー型における実用的なシステムでの使用例は P2P を補助的に用いた [17] 以外にほとんど見当たらない。また RNC でエンコードされたデータへのアクセス方法についてもほとんど情報がなく、ウェブブラウザから直接アクセスしてデコードするプログラムの存在は今のところ他に確認されていない。我々の調査した限り、クライアント・サーバー型で RNC でエンコードされたデータをウェブブラウザから直接アクセスし、動画の視聴や保存を可能にするマルチメディアストリーミングシステムは RNCDDS 以外に存在しない。

3. ランダムネットワークコーディングの基礎

RNC は連立一次方程式の特性を生かし、データシステムにおける高堅牢性とデータ容量の節約を両立させる。例えば $x_1 = 3, x_2 = 1, x_3 = 2$ とし、以下のように異なった係数の組み合わせで多数の一次方程式を作るとする。

$$\begin{cases} 2x_1 + 5x_2 + x_3 = 13 \\ 7x_1 + 3x_2 + 8x_3 = 42 \\ 4x_1 + x_2 + 2x_3 = 19 \\ 5x_1 + 4x_2 + 9x_3 = 33 \\ \cdots \end{cases} \tag{1}$$

ここで元の x_1, x_2, x_3 を得るには、(1) にある多数の方程式のうち三つの方程式があれば十分である。これが RNC の基本原理である。RNC は一つのファイルを x_1, x_2, x_3 の三つに分け、以下のように異なった組み合わせの係数 a で三つ以上の一次方程式を作る（エンコード）。

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 = b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + a_{2,3}x_3 = b_2 \\ a_{3,1}x_1 + a_{3,2}x_2 + a_{3,3}x_3 = b_3 \\ \cdots \end{cases} \tag{2}$$

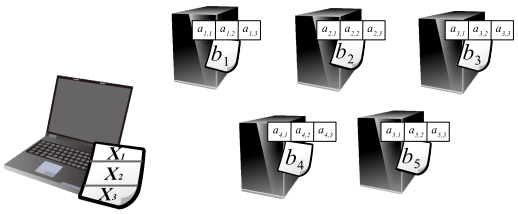


図 3: RNC はエンコードしたデータをサーバーに分散する。エンコードされた一つのファイルのサイズは元のファイルの約 $\frac{1}{3}$ である。

そして $b_1, b_2, b_3, \cdots$ の一つずつとそれらを求めるのに使用された係数 a を図 3 のように各サーバーに配布する。元のファイルを復元するには複数あるサーバーのうち三台のみからエンコードされたファイルを集め、連立一次方程式を解き x_1, x_2, x_3 を求め（デコード）、それらを連結させればよい。言い換えれば、多数あるエンコードされたファイルを持っているサーバーのうちどれでも三台のサーバーがアクセス可能であれば、元のファイルを復元することができる。そのため図 3 ではどの二台のサーバーがアクセス不能に陥ってもデータの取り出しが可能である。つまり冗長性は $R = 2$ である。

ここで注意すべきは $b_n, \forall n$ のサイズが x_1, x_2, x_3 と同じ、つまり元のファイルの $\frac{1}{3}$ であるという点である。これは (2) における演算を全てガロア体で行うからである。ちなみに係数 a のサイズは、例えば RNCDDS では一つ 2byte（三つで 6byte）と通常 b_n のサイズと比較して非常に小さく、そのため各サーバーに保存されるエンコードされたファイルのサイズは元のファイルの $\frac{1}{3}$ であると言っても過言ではない。結果として図 3 におけるシステム全体のデータ保存量は $\frac{1}{3} \times 5 = \frac{5}{3}$ となり、同等の冗長性 ($R = 2$) での Hadoop や GlusterFS で必要な保存量 3 の $\frac{5}{9}$ 倍（約 44%の節約）となる。

RNC における唯一の欠点はエンコード、デコードに要する処理時間であろう。ガロア体において複数の一次方程式を作り（エンコード）、そして三つの一次方程式から解を求める（デコード）作業には通常かなりの時間を要するが、我々は RNCDDS の開発時にそれらに要する時間はガロア体における演算速度が大きく関係していることを発見した。開発初期に我々はあるオープンソースのガロア体演算ライブラリを使用していたが、後にその演算速度が非常に遅いことが判明し、我々は RNC を利用した実用的なデータシステムの構築には高速なガロア体演算ライブラリが必要不可欠であるという結論に達した。その後 *GF-Complete* [18] という高速なオープンソースライブラリを用いることによって演算速度における問題を解決したが、GF-Complete は特許に抵触している恐れがあるとして後に作者により削除された [19]。GF-Complete は CPU 内の SSE の命令を利用して非常に高速なガロア体での演算を実現していたが、それが特許抵触の原因となったようである。しかしそれが逆に我々に新たなガロア体演算ライブラリ gf-nishida-16 を作り出すきっかけを与えることとなった。

4. GF-NISHIDA-16

第 2 章で述べた符号化されたデータを使用する分散データシステムではガロア体での演算が不可欠であるが、それにはかなりの時間と CPU パワーが費やされるため、システム全体のパフォーマンスを向上させることが困難であった。またガロア体での演算速度を改善するにあたっても、高速なガロア体演算アルゴリズムは特許を持っているものが多く、実現は容易でなかった。gf-nishida-16 [20] はそれらを解決するもので、特許に抵触する恐れのない技術を用い 16bit のガロア体（$GF(2^{16})$）での演算速度を飛躍的に向上させる。また gf-nishida-16 のソースコードは 2-Clause BSD ライセンスの下で `https://github.com/scopedog/gf-nishida-16/` において公開されている。本稿では gf-nishida-16 の詳細には触れずベンチマークのみを記述する。詳しい構造などは技術報告書 [21]（日本語）か [22]（英語）を参照して頂きたい。簡潔に言えば gf-nishida-16 は二段階メモリー探索を最適化することによって高速化を図っている。

ベンチマークは以下のガロア体演算ライブラリを対象に、乗算と除算における所要時間を計測して行った。

gf-nishida-8　gf-nishida-16 の 8bit 版。

gf-nishida-16　本章で紹介した主要ライブラリ。

gf-nishida-region-16　gf-nishida-16 の高速版で $a \times x$ または $a \div x$ の繰り返し計算において a が固定で x が変数の場合に有効。RNCDDS に使用されている。詳細は [22] [21] の第 4 章を参照。

gf-complete-*　SSE を利用した高速な $GF(2^n)$ 演算ライブラリ [18] [19]。

gf-basic-8　シンプルな 8bit ライブラリ [23]。

gf-plank-*　[24] に基づくライブラリ。

gf-plank-logtable-16　gf-nishida-16 に似た 16bit ライブラリ [24]。

gf-clmul-128　Solaris のソースコードから取り出した 128bit のプログラムで、$GF(2^n)$ の乗算に特化した

表 1: $GF(2^n)$ における繰り返し乗除算に所要された時間 (ms)。短いほど高速である。

ライブラリ	乗算	除算
gf-nishida-region-16	**41583**	**41557**
gf-complete-64	55106	4424996
gf-nishida-8	61171	86159
gf-basic-8	61195	114292
gf-nishida-16	118850	118973
gf-complete-32	168429	10053109
gf-plank-logtable-16	244935	251010
gf-plank-32	314016	27664514
gf-plank-16	391792	369352
gf-plank-8	406299	360553
gf-clmul-128	1281013	-
gf-ff-64	5231520	5393946
gf-ff-32	10464125	105356429
gf-aes-gcm-128	14013111	-

Intel の CLMUL 命令セットを使用。

gf-ff-* [25] よりダウンロードしたライブラリ。

gf-aes-gcm-128 FreeBSD のソースコードから取り出した 128bit のプログラムで、何ら高速化のための手法を用いていないプログラム。

ライブラリによっては除算の関数を持たないものもあるため、それらは結果から除外した。

ベンチマークの結果は表 1 の通りであり、演算に要した時間が短いほど高速である。結果的に gf-nishida-16、特に RNCDDS で使用されている gf-nishida-region-16 は乗除算の両方においてその高速性が示されており、特許侵害の危険性のなさを含めて gf-nishida-16 は RNC における最適なガロア体演算ライブラリだと考えられる。

5. RNCDDS

5.1. エンコード

四台のサーバーにエンコードされたデータを送信するとすると、RNCDDS におけるクライアントはまず各データに対する RNC の係数 $a_{k,1}$, $a_{k,2}$, $a_{k,3}, \forall k \in \{1,2,3,4\}$ を決める（(2 参照)。次にクライアントはそれらをファイルサイズ、ファイルの更新された時間といった情報と共に各サーバーに配布し、その後ファイルのエンコードを開始する。エンコードはファイルを 6byte 毎に区切って行われ、その 6byte のデータを更に図 4 のように 2byte 毎

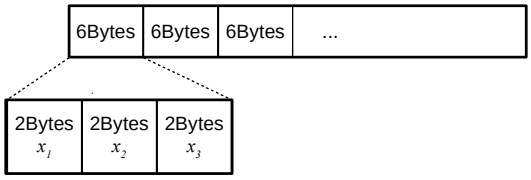


図 4: RNCDDS はファイルを 6byte 毎に区切り更にその中の 2byte 毎を x_1, x_2, x_3 に割り当てる。

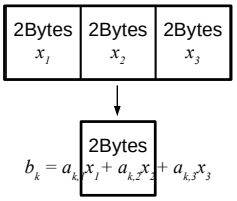
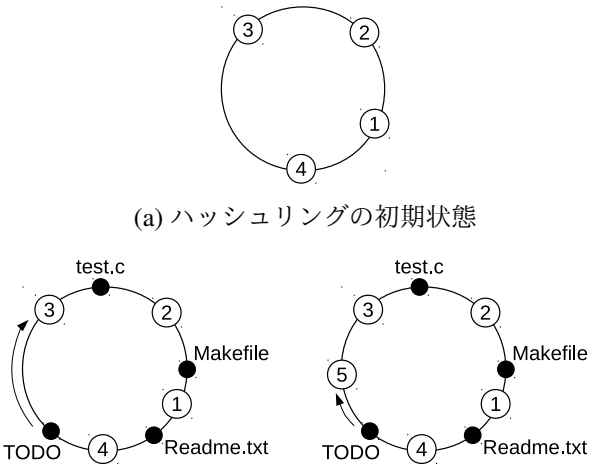


図 5: RNCDDS は 6byte のデータ x_1, x_2, x_3 をエンコードし 2byte のデータ b_k を得る。

に区切って x_1, x_2, x_3 に割り当て、図 5 のように $b_k = a_{k,1}x_1 + a_{k,2}x_2 + a_{k,3}x_3, \forall k \in \{1,2,3,4\}$ を gf-nishida-16 を用いて算出することによって行われる。新たに求められた 2byte のデータ b_k はサーバー k に送られ、一連の作業はファイルの末尾に到達するまで行われる。ここでエンコードされたデータのサイズは元のファイルの $\frac{1}{3}$ であるが、先に述べたようにサーバーに保存されるファイルには RNC の係数や元のファイルの情報がヘッダーとして付加されるので、最終的にサーバーに保存されるファイルのサイズは元のファイルの $\frac{1}{3}$ よりも若干大きくなる。RNCDDS では元のファイルの属性、所有者 ID、SHA256 によるチェックサム値等も含むため、168byte をヘッダーとして使用している。

5.2. デコード

デコードはまず目的のファイルを所有しているサーバーを探すことから始まるが、第 5.3 章で述べるように RNCDDS はこれを Hadoop などで採用されているメタデータサーバーを使用せず、**コンシスタントハッシング** [26] というより安全な手法を用いて行う。ファイルの探索において、もしファイルを所有しているサーバーの台数が三台未満であればデコードは失敗となる。もし三台以上であればクライアントはそのうち三台のサーバーを選択し、まず RNC の係数 $a_{k,1}, a_{k,2}, a_{k,3}$ を含むヘッダー情報をそれらから取得する。そしてエンコードされたデータをダウロードしながら、2byte の b_k 毎に以下の連立一次方程式からガウスの消去法を用いて x_1, x_2, x_3 を求め、そ



(a) ハッシュリングの初期状態

(b) サーバー追加前　　(c) サーバー追加後

図 6: コンシスタントハッシングによるファイル管理例。

れらを連結して 6byte 毎に元のデータを復元する。

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 = b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + a_{2,3}x_3 = b_2 \\ a_{3,1}x_1 + a_{3,2}x_2 + a_{3,3}x_3 = b_3. \end{cases} \tag{3}$$

この作業はファイルの末尾まで行われ、作業が終了した時点で復元されたファイルは SHA256 で検査される。

5.3. コンシスタントハッシングによるファイル管理

Hadoop ではメタデータサーバーを使ってどのサーバーがどのファイルを所有しているかなどの情報を管理している。しかしメタデータサーバーは分散システムにおいていわゆるアキレス腱ともなりうり、メタデータサーバーがアクセス不能になると、システム全体でのデータの取り出しも不可能になるという重大な欠点を持っている。RNCDDS はその問題を解決するため、メタデータサーバーのような情報を管理する特別なサーバーを使用せず、コンシスタントハッシング（CH）[26] という手法を用いてどのサーバーが目的のファイルを持っているかなどの情報を、クライアントが直接割り出すことができるように工夫されている。

例として 4 台のサーバーと test.c, Makefile, Readme.txt, TODO の 4 つのファイルが存在するとする。CH ではまず図 6 (a) に見られるようにハッシュ値を用いてリング上にサーバーを割り当てる。次いで同様に図 6 (b) のようにリング上にファイルを割り当てるが、各ファイルの所有者はそのファイルの位置から時計回りに最も近いサーバーとなる。例えば図 6 (b) ではサーバー 3 がファイル TODO を所有することになる。CH の利点はサーバーの追加、削除に柔軟に対応し、更にファイルを所有しているサーバーを高速に割り出すようになっていることで、サーバーの追加、削除の際にはファイルを所有するサーバーの変更が最低限で収まるように設計されている。例えば図 6 (c) のようにサーバー 5 が追加されたとすると、ここでファイルの所有者が変更されるのは TODO のみで、TODO の新たな所有者はサーバー 5 となるが他のファイルは影響を受けない。

RNCDDS ではまずサーバー側がクラスター内の全てのサーバーのホスト名を保持しており、クライアントがサーバーに接続した際にそのサーバーリストを一台のサーバーから受け取る。なお全てのサーバーが共通のサーバーリストを保持しているものと仮定する。クライアントはそのリストを元に図 6 (a) のようなハッシュリングを作り、図 6 (b) (c) のようにどのサーバーが目的のファイルを持っているかを割り出す。RNC では最低三台のサーバーが目的のファイルを所有していなければならないが、CH により割り出されたサーバーを server04 とすると、他のサーバーに関してはサーバーリスト上にある server04 に続くサーバー（例 : server05、server06）が所有しているものと見なす。これによるファイル検索のための計算量は $O(1)$ で、非常に効率の良い検索を実現する。また RNCDDS は何らかの理由で CH に不具合が起きた状態でも、クライアントから全サーバーにメッセージを送ることによって、目的のファイルの所有者を発見することができる仕組みになっている。こういった点でも RNCDDS はデータ保護に万全を期している。

5.4. プログラム

RNCDDS は三つの C プログラムと一つの JavaScript プログラムで構成されており、C プログラムは FreeBSD 11 と CentOS 7 で動作確認されている。また C プログラムは OpenSSL, FUSE [27], libevent2 [28] （CentOS 用のみ、FreeBSD 版には kqueue を使用）といったごく少数の外部ライブラリにのみ依存するように設計されており、他のプラットフォームへの移植が容易に行えるようになっている。

rncddsd は C で書かれたサーバープログラムで rncdds や rncfsd といったクライアントプログラムと通信し、データの送受信、読み込み、保存を行う。通常はデーモンプロセスとして使用される。

rncdds は C で書かれたコマンドラインのクライアントプログラムで、Hadoop での `hadoop fs` コマンドに相当し、ファイルのエンコード、デコード、サーバーへ／から

のアップロード、ダウンロード等を実行する。例えば

```
% rncdds put fileA /dirA/
```

はファイル fileA をサーバーの/dirA ディレクトリにアップロードする。rncdds は並列パイプライン処理により複数のファイルの読み込み、エンコード、アップロード、ダウンロード、デコード、保存を同時に行うように設計されており、高速なディレクトリのアップロード、ダウンロードが可能となっている。第 7 章でのベンチマークでは特にディレクトリのアップロードで優れた結果を出している。

rncfsd はサーバーに保存されたデータをファイルシステムとしてローカルなホストからアクセスできるようにする C プログラムで、FUSE [27] というユーザーランドでファイルシステムを実現するライブラリを利用して作られている。これは非常に便利で、サーバーに保存されたファイルをあたかも自分のマシン上に存在するかのように扱うことができるが、様々なオーバーヘッドのためアップロード、ダウンロードに おける速度は rncdds よりも劣る（表 2 参照）。しかし一旦ダウンロード、デコードしたファイルに関しては SSD や HDD にキャッシュとしてローカルホストに保存する仕組みになっており、再デコーディングによる手間を最低限に抑えるように工夫されている。

rncdds, rncfsd は共にサーバーに保存されたファイルの整合性をそのタイムスタンプやサイズなどで検査し、デコードされたファイルに関しては SHA256 で元のファイルとの同一性を確認し、安全を図っている。また三つの C プログラムはメモリー使用量にも工夫がなされており、わずか 4GB の RAM でそれらを同時に同一ホストで走らせることが可能となっている。

RNC.js は JavaScript プログラムで、RNC でエンコードされたデータをサーバーからダウンロードし、デコードを行いながらデータが動画であればウェブブラウザ上で再生を行う。ダウンロードは XMLHttpRequest 関数を用いて HTTP で行い、動画再生には HTML5 の機能を使用する。現時点ではセグメント化された MP4 と WebM フォーマットの再生に対応している。デコードによるオーバーヘッドのため、RNC.js による動画再生は通常の MP4 や WebM データの再生に比べて若干 CPU パワーを多く消費するが、その差は大きくない。図 7 は通常の MP4 再生時（図左側）と RNC.js による再生時（図右側）での CPU 消費量を比較したものであるが、両者の開始時でのページ読み込みにかかる CPU 消費量（図内の突起）を除くと、それぞれの消費量に大きな差がないことが解かる。例えば通常の MP4 の再生で消費される CPU パワーが約 6〜7%だとすると、RNC.js での再生における CPU 消費量は大雑把に 7〜9%である。これは gf-nishida-16 の高速性が大きく貢献し

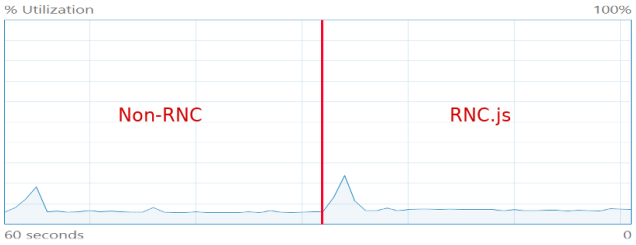


図 7: 通常の MP4 の再生と RNC.js を使用した際とにおける CPU 消費量の比較。大きな違いはない。

(a) 五台の独立した HTTP サーバー

(b) 五台の RNCDDS サーバーによる HTTP システム

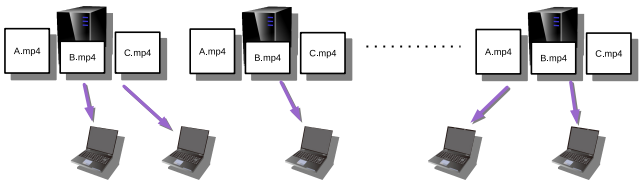
図 8: RNCDDS による負荷分散システム例。

ているためだと思われる。なお将来 RNC のデコード機能がウェブブラウザに直接実装されれば、この CPU 消費量は更に低下するものと思われる。

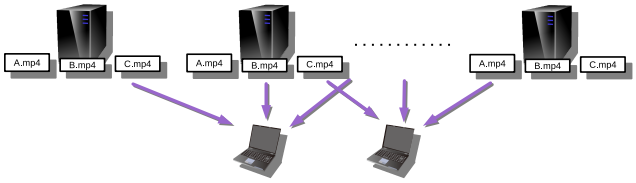
5.5. RNCDDS による負荷分散

RNCDDS はデータを分散してデータ保護を行うだけでなく、負荷分散のために用いることも可能である。ここでは RNCDDS の特殊な使用例として HTTP サーバーの負荷分散を行うシステムを紹介する。図 8 (a) のように五台の独立した HTTP サーバーが存在し、それぞれ異なったウェブサイトを提供しているものとする。次にこの五台のサーバーが所有しているウェブサイトを全て図 8 (b) のように五台の RNCDDS サーバーによるシステムに蓄える。ここで各 RNCDDS サーバーに第 5.4 章で紹介した rncfsd プログラムを走らせると、五台の全ての RNCDDS サーバーから蓄えたウェブサイトのデータにアクセス可能となる。更にこの五台の RNCDDS サーバーに Apache, Nginx などの HTTP サービスプログラムを走らせると、クライアントはどの RNCDDS サーバーからも蓄えられた全てのウェブサイトにアクセスできることとなり、これにより負荷分散が可能となる。これには以下のような利点がある。

1. 図 8 (a) のような独立した HTTP サーバーでは一台



(a) CDN ではファイルの複製が多数のサーバーに保存される



(b) RNCDDS を用いることによりデータ保存量が $\frac{1}{3}$ になる

図 9: RNCDDS を用いた CDN 例。

のサーバーにアクセスが集中した場合、高負荷によりそのサーバーがアクセス不能に陥ることがある。図 8 (b) のようなシステムを利用すれば五台のサーバーに負荷が分散し、アクセス不能に陥る可能性が減少する。

2. RNCDDS システムへのデータ移行の際、データに冗長性を持たせると（例 $R = 2$）、五台の RNCDDS のサーバーの内二台が故障などでアクセス不能に陥った場合でも残りの三台でサービスを継続させることが可能で、耐久性、信頼性を増加させることができる。

なおデコードによるオーバーヘッドは、rncfsd がデコード済みのデータをキャッシュとして一定時間蓄えるため緩和される。

6. RNCDDS によるマルチメディアストリーミング

現在多くのマルチメディアストリーミングサービスが、クラウドシステムと密接な関係にあるコンテンツデリバリーネットワーク（CDN）を介して提供されている。CDN では図 9 (a) のようにファイルの複製が多くのサーバーに置かれ、クライアントからのアクセスを分散する仕組みになっている。しかしこれには膨大なデータ保存量が必要で、結果的に以下のような深刻なパフォーマンス劣化を引き起こす。

1. SSD に対する HDD の使用率が増加し、スループットが低下する。
2. キャッシュヒット率も低下し、同様にスループットが低下する。

ここで CDN 内に保存されているデータを図 9 (b) のように RNC でエンコードされたデータに置き換えると、保存容量を $\frac{1}{3}$ にすることができ、費用の削減だけでなくスループットの改善も可能となる。この場合クライアントは RNC.js を介して三つのサーバーにアクセスすることとなり、動画データの場合はそのままウェブブラウザ上での再生が可能である。このデータ保存量の削減による恩恵は非常に大きく、我々は RNCDDS が多くの CDN で利用されることを望んでいる。

また注目すべきは、この RNCDDS への移行においてサーバー側の設定変更がほとんど必要ないということである。実際にサーバー管理者が行わなければならない作業は

1. 元のファイルを RNC でエンコードされたファイルに置き換える。
2. Nginx や Apache の設定にファイルにおいて、Cross-Origin Request Shaing (CORS) を有効にするため”add_header ’Access-Control-Allow-Credentials’ ’true’;”のような設定を付け足す。

の二つのみで、2. はクライアントが三つの異なった HTTP サーバーに接続できるようにするためである。またクライアントはサイズが $\frac{1}{3}$ のデータを三つダウンロードするため、総トラフィック量に変化がないことに注意頂きたい。

現在の RNCDDS の試作システムでは、図 10 のようにまずクライアントがポータルサイトサーバーにアクセスして RNC.js と三つの RNC でエンコードされたファイルのサーバーに関する情報をダウンロードし、次にエンコードされたデータをそれらのサーバーからダウンロードして動画を再生するような仕組みになっている。データのやり取りは全て HTTP で行われる。RNC.js を使った動画のデモは `http://rnc01.asusa.net` で視聴でき、元となった動画（非 RNC）は `http://rnc01.asusa.net/videos` で視聴できる。なお上記サイトにある RNC.js は試作版で、最新のものとは大きく異なることに留意して頂きたい。

7. ベンチマーク結果

RNCDDS のパフォーマンスを測るため、我々は巨大なディレクトリ（5,533 個のサブディレクトリと 80,919 個のファイルを含む）とその 63GB のアーカイブ（tarball）ファイルのアップロード、ダウンロードに要した時間を、Hadoop, GlusterFS と共に計測した。各計測は 10 回繰り返され、その平均値を算出した。計測に使用されたサーバー、クライアントのハードウェアの仕様は以下の通りである。

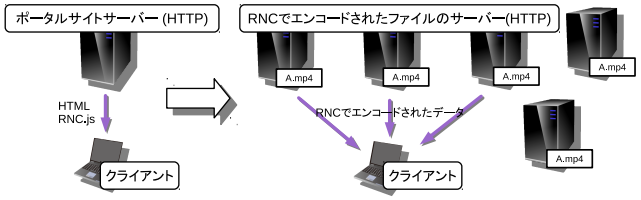


図 10: クライアントはポータルサイトサーバーにアクセスし、その後三つの RNC サーバーからエンコードされたデータをダウンロードする。

CPU　Intel Xeon E3-1225v5

RAM　DDR4-2133 ECC Non-buffered 64GB

NIC　Intel X550 10GBase-T（スイッチの都合上、計測では 1Gbps を使用）

HDD　メインストレージ: 2TB 64MB Cache 7200RPM SATA3×2 の RAID1, `rncfsd` の書き込み用のキャッシュ: 1TB 64MB Cache 7200RPM SATA3

SSD　`rncfsd` 用の読み込みキャッシュ: 128GB SATA3

データ保存用には四台のサーバーを用い、Hadoop ではその四台をスレーブノードとして使用した。クライアントは一台で、データの冗長性にはいずれも $R = 1$、つまり Hadoop と GlusterFS では二つの複製ファイルが、RNCDDS（`rncdds` と `rncfsd`）では四つのエンコードされたファイルが四台のサーバーに分散されるように設定した。OS は CentOS 7 である。ファイル、ディレクトリのアップロード、ダウンロードにおいて、Hadoop には

```
% hadoop fs -copyFromLocal/-copyToLocal
```

コマンドを用い、`rncdds` には

```
% rncdds put/get
```

コマンドを用いた。GlusterFS および `rncfsd` はファイルシステムとしてデータを扱うため、

```
% cp -a
```

をデータをマウントしたディレクトリに対して行った。

結果は表 2 の通りである。全ての計測において `rncdds` は Hadoop, GlusterFS を上回っており、特にディレクトリのアップロードにおいてはそれらの約 $\frac{1}{4}$ の時間で処理を終えている。小さなファイルの取り扱い時における Hadoop の処理速度の劣化は既知の問題で、[29] [30] などに報告されている。FUSE を使ったファイルシステムによるオーバーヘッドのため `rncfsd` は `rncdds` と比較し全体的に処理速度が劣っており、特にディレクトリのダウンロードにおいてそれが顕著になっている。これは `rncdds` が複数のファイルのダウンロードとデコードを同時に行うのに対し、`rncfsd` では一つのファイルの処理が終わるまでファイルシステムから次の処理を与えられないことに起因している。注意して頂きたいのは、`rncdds`, `rncfsd` は Hadoop, GlusterFS よりもかなりの CPU パワーを消費することである。そのため RNCDDS を使用の際には高速な CPU の使用を推奨する。

表 2: 63GB のファイル／ディレクトリのアップロード／ダウンロードに要した時間（短いほど高速）。

	Hadoop	GlusterFS	rncdds	rncfsd
ファイルアップロード	10m30.2s	17m26.4s	**8m49.6s**	8m56.6s
ディレクトリアップロード	48m34.0s	54m18.5s	**12m31.1s**	17m04.5s
ファイルダウンロード	10m31.9s	10m51.3s	**8m40.0s**	9m12.8s
ディレクトリダウンロード	20m51.0s	25m10.0s	**16m51.1s**	34m54.5s

8. 結論と課題

本論文では RNCDDS の詳細について解説し、そのデータ保護における堅牢性とデータ使用量における高効率性、マルチメディアストリーミングとの親和性、データシステムとしての処理速度の高さを紹介してきた。RNCDDS はその実用性の高さから、将来クラウドシステムの発展に貢献し、マルチメディアストリーミングにおいても重要な役割を果たすものと我々は考えている。

現時点で RNCDDS は分散データシステムとしての実装を完了しており、CDN やマルチメディアストリーミングシステムにも対応できるようになっているが、便宜性を考慮して CDN やマルチメディアストリーミングシステムに特化した実装を検討している。また RNC.js は最適化の余地が残されており、今後更なる高速化と動画再生時の操作性の向上を目指す予定である。研究課題としては CDN、マルチメディアストリーミングシステムにおけるネットワーク帯域及びデータ保存量の観点からの最適なエンコードファイルの分散方法の模索が残されている。これは最適化問題となり、成功すれば更にデータ保存量を減少させることが可能になると思われる。

9. 参考文献

[1] “Hadoop,” `http://hadoop.apache.org/`.

[2] “Glusterfs,” `https://www.gluster.org/`.

[3] “Openafs,” `https://www.openafs.org/`.

[4] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell

D. E. Long, and Carlos Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2006, OSDI '06, pp. 307–320, USENIX Association.

[5] "Moosefs," https://moosefs.org/.

[6] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly, "The quantcast file system," *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1092–1101, Aug. 2013.

[7] Cheng Huang, Minghua Chen, and Jin Li, "Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems," *Trans. Storage*, vol. 9, no. 1, pp. 3:1–3:28, Mar. 2013.

[8] Kien Nguyen, Thinh Nguyen, Yevgeniy Kovchegov, and Viet Le, "Distributed data replenishment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 2, pp. 275–287, Feb. 2013.

[9] B. Li and D. Niu, "Random network coding in peer-to-peer networks: From theory to practice," *Proceedings of the IEEE*, vol. 99, no. 3, pp. 513–523, March 2011.

[10] Alexandros G. Dimakis, P. Brighten Godfrey, Yunnan Wu, Martin J. Wainwright, and Kannan Ramchandran, "Network coding for distributed storage systems," *IEEE Trans. Inf. Theor.*, vol. 56, no. 9, pp. 4539–4551, Sept. 2010.

[11] Ádám Visegrádi and Péter Kacsuk, *Efficient Random Network Coding for Distributed Storage Systems*, pp. 385–394, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[12] Kien Nguyen, Thinh Nguyen, and Sen-Ching Cheung, "Video streaming with network coding," *J. Signal Process. Syst.*, vol. 59, no. 3, pp. 319–333, June 2010.

[13] Anh Tuan Nguyen, Baochun Li, and Frank Eliassen, "Chameleon: Adaptive peer-to-peer streaming with network coding," in *Proceedings of the 29th Conference on Information Communications*, Piscataway, NJ, USA, 2010, INFOCOM'10, pp. 2088–2096, IEEE Press.

[14] B. Barekatain, D. Khezrimotlagh, M. Aizaini Maarof, H. R. Ghaeini, S. Salleh, A. A. Quintana, B. Akbari, and A. T. Cabrera, "MATIN: A Random Network Coding Based Framework for High Quality Peer-to-Peer Live Video Streaming," *PLoS ONE*, vol. 8, pp. e69844, Aug. 2013.

[15] Luísa Lima, Steluta Gheorghiu, João Barros, Muriel Médard, and Alberto Lopez Toledo, "Secure network coding for multi-resolution wireless video streaming," *IEEE J.Sel. A. Commun.*, vol. 28, no. 3, pp. 377–388, Apr. 2010.

[16] D. Vukobratovi ☒, C. Khirallah, V. Stankovi ☒, and J. S. Thompson, "Random network coding for multimedia delivery services in lte/lte-advanced," *IEEE Transactions on Multimedia*, vol. 16, no. 1, pp. 277–282, Jan 2014.

[17] Z. Liu, C. Wu, B. Li, and S. Zhao, "Uusee: Large-scale operational on-demand streaming with random network coding," in *2010 Proceedings IEEE INFOCOM*, March 2010, pp. 1–9.

[18] James S. Plank, Kevin M. Greenan, and Ethan L. Miller, "Screaming fast galois field arithmetic using intel simd instructions," in *11th USENIX Conference on File and Storage Technologies (FAST 13)*, San Jose, CA, Feb. 2013, pp. 298–306, USENIX Association.

[19] James S. Plank and et al., "Gf-complete: A comprehensive open source library for galois field arithmetic, version 1.0," http://web.eecs.utk.edu/~plank/plank/papers/CS-13-716.html.

[20] Hiroshi Nishida, "gf-nishida-16 web site," https://github.com/scopedog/gf-nishida-16/.

[21] 西田 博史, "gf-nishida-16: シンプルかつ高速な $gf(2^{16})$ 演算ライブラリ," https://github.com/scopedog/gf-nishida-16/blob/master/gf-nishida-16-ja.pdf.

[22] Hiroshi Nishida, "gf-nishida-16: Simple and efficient $gf(2^{16})$ library," https://github.com/scopedog/gf-nishida-16/blob/master/gf-nishida-16.pdf.

[23] "Basic library for calculation on finite field," http://www.codeforge.com/article/242688/.

[24] James S. Plank., "Fast galois field arithmetic library in c/c++," http://web.eecs.utk.edu/~plank/plank/papers/CS-07-593/.

[25] Antonio Bellezza, "Binary finite field library," http://www.beautylabs.net/software/finitefields.html.

[26] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on*

Theory of computing, New York, NY, USA, 1997, STOC '97, pp. 654–663, ACM.

[27] "Fuse (filesystem in userspace)," https://github.com/libfuse/.

[28] "libevent - an event notification library," http://libevent.org/.

[29] "Dealing with hadoop's small files problem," http://snowplowanalytics.com/blog/2013/05/30/dealing-with-hadoops-small-files-problem/.

[30] "The small files problem," http://blog.cloudera.com/blog/2009/02/the-small-files-problem/.