東京工科大学大学院バイオ・情報メディア研究科

修士論文

論文題目

疎結合マルチクラスタ向き 分散ストレージシステム **Elton** の開発

指導教員

田胡 和哉	印

提出日　2016 年 01 月 26 日

提出者

専　攻	コンピュータサイエンス専攻
学籍番号	G2114032
氏　名	水野 拓

2015 年度 コンピュータサイエンス専攻修士論文概要

論 文 題 目	疎結合マルチクラスタ向き 分散ストレージシステム Elton の開発
執 筆 者 氏 名	水野 拓
指 導 教 員	田胡 和哉 教授

【概 要】

クラウドコンピューティングを支える基盤技術の開発競争が激化してきており，この流れは企業・団体にとどまらず教育現場にも及んでいる．例えば本学でも，プライベートクラウド施設「クラウドサービスセンター」とパブリッククラウドを連携させることで，ピーク時に処理を相互に負荷分散可能な Docker ベースのハイブリッドクラウド機構を構築し，提供している．

プライベートクラウドとパブリッククラウドのように，ネットワークや管理者などが異なるような疎に結合した複数のクラスタを連携する際には，ストレージ基盤が課題となる．パブリッククラウドでは，通信量従量課金制をとっているものが多く，ファイル同期がコストの増大につながってしまい，従来のストレージ基盤では対応が難しい．

そこで疎に結合した複数クラスタ環境に適したストレージ基盤として，コミットしたタイミングでデータ共有可能になる分散ストレージシステムを提案する．ファイルの取得を行わないかぎり変更が伝搬しないようにすることで，変更による影響を局所化し，コミット時にバージョンを発行し，データを不変に管理することでキャッシュが容易となる機構を持つ．こうすることでファイル同期のコストを抑える．この分散ストレージシステムを Elton と呼称する．

一般的な分散ファイルシステムでは，中央に大規模なファイルサーバを用意し，ネットワークを介してすべてのノードが接続しファイル共有する．疎結合マルチクラスタ環境では通信経路の確保が難しくなり，分散システム全体の構成が複雑化しやすい．それに対して Elton ではメタデータサーバ同士を接続することでクラスタ間をつなぎ，各ノードはクラスタ内のメタデータサーバに接続することでファイル共有する．こうすることで分散システム全体の構成も簡略化することができる．

性能検証実験の結果ストレージシステムとして実用に値する性能を得ることができた．今後はクラウドサービスセンターのストレージ基盤として活用することを予定している．

A b s t r a c t

T i t l e	Implementation of a distributed storage system Elton for loosely coupled multi-cluster
A u t h o r	Taku MIZUNO
A d v i s o r	Professor Kazuya TAGO

【Summary】

The flow of the competition to develop cloud computing infrastructure technology is also extends to the field of education not only to companies and organizations. For example, it provides to build a Docker-based hybrid cloud mechanism that private cloud facility as “Cloud Service Center” was coordinating public cloud to each other load balancing the processing at the time of the peak in the university.

In loosely coupled multi-cluster, such as between the private cloud Public cloud network and administrators are different there is a problem in the storage system. Synchronization are many files that are taking the amount of communication pay-per-use system in the public cloud leads to increase in cost. Difficult correspondence with that reason conventional storage systems.

So we propose a distributed storage system “Elton” for loosely coupled multi-cluster to become data can be shared by committed timing. It will be the first time get the timing of the file synchronization. And it can be a simple cache that immutable to manage the issue a version at the time of committing data. So you can reduce the cost of file synchronization.

In a typical distributed file system, all nodes are file shares connected via a network to a large file server. Loosely coupled multi-cluster is more difficult secure communication path, and tends to be more complicated configuration of the entire distributed system. However, connecting between clusters by connecting the metadata server between the Elton. Each node file sharing by connecting to a metadata server in the cluster. And it can also be simplified configuration of the entire distributed system by this.

It was possible to obtain a performance worthy practical result as a storage system performance verification experiments. It is scheduled to be used as a storage system of the Cloud Service Center in the future.

目次

図目次

表目次

第 1 章

はじめに

クラウドコンピューティングを用いたサービスが近年急速に普及・発展している．クラウドコンピューティングでは，インターネットを介することでサービスを提供する環境 (サーバ群やネットワーク等) を意識することなく利用できるというメリットがある．特に Amazon Web Services[1] や Google Cloud Platform[2] に代表されるサーバやネットワーク，ストレージなどのインフラ環境を提供する Infrastructure as a Service の普及が進んでいる．気軽にインフラ環境を用意可能かつ運用を気にする必要が無いため，サービスの開発に集中できるというメリットから導入する企業・団体が年々増加している．

クラウドコンピューティングの急速な普及・発展にともないクラウドコンピューティングを支える基盤技術も発展してきている．例えば Docker 社が中心となって開発をしている Docker[3] に代表されるコンテナによる軽量な仮想化技術が出現し，多くのクラウドサービスで基盤技術として導入されている．さらにコンテナのオーケストレーションツール Kubernetes[4] は Google，Docker，Intel，IBM など多くの有名 IT 企業や団体で構成される Cloud Native Computing Foundation[5] に開発が委譲されるなど多くの企業・団体が Docker を支える技術の開発に取り組んでいる．

このクラウドコンピューティング技術開発の流れは企業・団体にとどまらず教育現場にも及んでいる．例えば，本学でも筆者らによってプライベートクラウド施設「クラウドサービスセンター」を開設し，全学生・教職員に対してサービス提供している [6]．さらに，プライベートクラウドであるクラウドサービスセンターとパブリッククラウドを連携させることで，ピーク時に処理をパブリッククラウドにオフロード可能な Docker ベースのハイブリッドクラウド機構を構築し提供している [7]．

プライベートクラウドとパブリッククラウドのようにネットワークや管理者などが異なるような疎に結合した複数のクラスタを連携する際にはストレージ基盤が課題となる．ファイルを共有するためにはネットワークを介する分散ファイルシステムが必要となる．しかし，ネットワークそのものが異なることによる同期遅延や，常に同期する必要がないファイルまで同期してしまうことによるトラフィックの増大などが課題となる．さらに多くのパブリッククラウドでは通信量従量課金制をとっているものが多くファイル同期がコストの増大につながってしまう．

この課題は複数のパブリッククラウドをそれぞれの特徴を活かして組み合わせたい場合や本学クラウドサービスセンターのようにピーク時のみ一時的にパブリッククラウドを利用したい場合，手元にある開発環境サーバとパブリッククラウド上にある本番環境サーバ間でファイル同期したい場合などでも今後考えられる．

そこで疎に結合した複数クラスタ環境に適したストレージ基盤として，コミットしたタイミングでデータ共有可能になる分散ストレージシステムを提案する．ファイル同期に対してコミットの動作を加

え，ファイルの取得を行わないかぎり変更が伝搬しないようにすることで変更による影響を局所化する．こうすることで，不要となる同期を減らすことができるため，同期トラフィックを抑えられる．さらに，コミット時にバージョンを発行しデータを不変に管理することでキャッシュが容易となり，一度取得を行えばローカルファイル同様に扱えるため SSD 等の高速なストレージとの相性が良くなりパフォーマンスの向上が期待できる．この分散ストレージシステムを Elton と呼称する．

本稿では，疎結合マルチクラスタに適したストレージ基盤について検討し，設計・実装を行ったコミットベースの分散ストレージシステム Elton について述べる．

第 2 章

クラウドコンピューティング

2.1　クラウドコンピューティングとは

クラウドコンピューティング (以下，クラウド) とは，共用の構成可能なコンピューティングリソース (ネットワーク，サーバー，ストレージ，アプリケーション，サービス) の集積に，どこからでも，簡便に，必要に応じて，ネットワーク経由でアクセスすることを可能とするモデルであり，最小限の利用手続きまたはサービスプロバイダとのやりとりで速やかに割当てられ提供されるものである [8].

従来ユーザが手元でコンピュータ (ハードウェア)・ソフトウェア・データ等の管理・運用などを行っていた．しかしクラウドでは，インターネットを介することでネットワークの向こう側の雲 (Cloud:クラウド) の中に存在するハードウェア・ソフトウェアを利用する．そのため，ユーザはインターネットに接続可能な環境さえ用意すれば，利用したいものをどこからでも手軽に好きなだけその仕組みを意識することなく利用できるという特徴がある．また，そのハードウェア・ソフトウェアの管理・運用を行う必要がないという点もクラウドの大きな特徴である．

クラウドそのものの中身はサービスを提供する会社が保有する世界中のデータセンターのサーバやネットワーク機器群である．これらは入れ替えの容易な汎用機器やオープンソースソフトウェアを利用しているため安価なサービスの提供を可能にしている．また，大量のサーバ上で，仮想化技術や分散処理技術，大規模コンピュータ運用技術，オープンソースソフトウェア技術を組み合わせることで高機能，高品質のサービスを提供している．

2.2　クラウドコンピューティングの提供形態

クラウドによって提供されるサービス形態のイメージを図 **2.1** に示す．クラウドのサービス提供形態は，図 **2.1** のように「Application」，「Platform」，「Infrastructure」の 3 つに分類されることが多い．それぞれは提供するサービスによって分類され，次のような特徴を持つ．

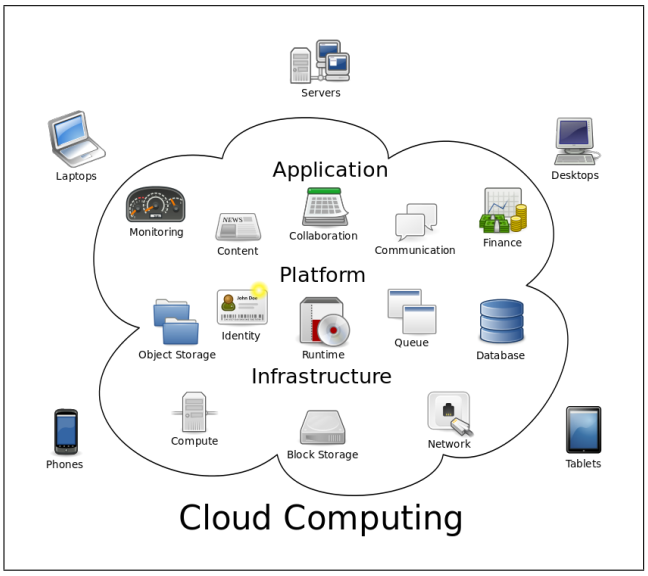

図 2.1　クラウドコンピューティングにおけるサービス提供イメージ ([9] より)

2.2.1　SaaS (Software as a Service)

SaaS とは，主にソフトウェアをウェブアプリケーションの形で提供するサービス形態である．ユーザはサービスを動作させるハードウェア・OS・ソフトウェア等を用意する必要がなく，ウェブブラウザを通して SaaS 提供サーバに接続することでソフトウェアを利用する．

代表的なものに Google Apps[10] が挙げられる．Google Apps では，独自ドメインを登録することでメールサービスを提供する Gmail や Google ドキュメント，Google スプレッドシート，Google スライド等のオフィス系ソフトウェアなど様々な Google のプロダクトを利用できる．

2.2.2　PaaS (Platform as a Service)

PaaS とは，主にアプリケーションの実行用プラットフォームを提供するサービス形態である．ユーザはインターネットを通して実行したいアプリケーションのプログラムを PaaS 提供サーバにアップロードし，そのサーバ上で作成したアプリケーションを実行できる．

代表的なものに Google App Engine[11] が挙げられる．Google App Engine では，ウェブアプリケーションを Java や Python・Go 言語を用いて開発し，Google が提供するサーバ上で実行・公開することができる．

2.2.3 IaaS (Infrastructure as a Service)

IaaS とは，主にハードウェア (CPU・メモリ・ストレージ) やネットワーク等のインフラ環境を提供するサービス形態である．ユーザには多くの場合インターネットを通してサーバ環境が提供される．自身でミドルウェア・ソフトウェアのインストールやアプリケーションの実行ができる．

代表的なものに Amazon EC2[12] が挙げられる．Amazon EC2 では，ユーザはサーバ環境を「インスタンス」という単位で提供され，インスタンスのスペック (CPU 数・メモリ量・ストレージ量等) や実行時間およびデータ転送量に応じて従量課金を行う．用途に合わせてスペックやインストールするミドルウェア・ソフトウェアを自由に変えることができる．

2.3 クラウドコンピューティングの種類

クラウドの利用形態には主にサーバが稼働する場所によって，図 **2.2** のように「Private/Internal(プライベートクラウド)」，「Public/External(パブリッククラウド)」，「Hybrid(ハイブリッドクラウド)」の 3 形態に分類され，次のような特徴を持つ．

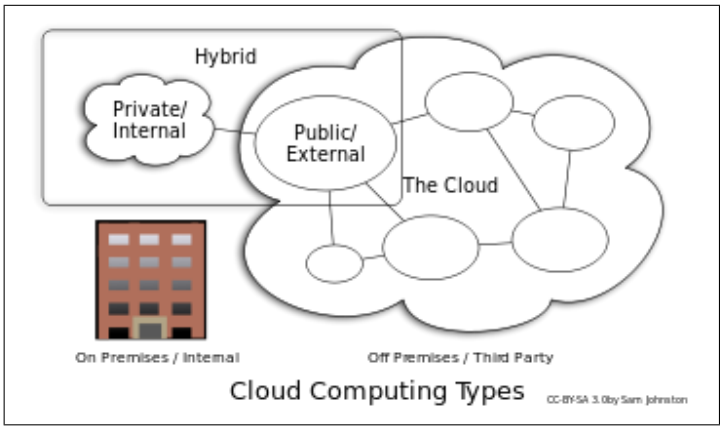

図 2.2 クラウドコンピューティングの種類 ([9] より)

2.3.1 パブリッククラウド

パブリッククラウドとは，インターネット経由で一般向けに提供されるクラウドサービスを指す．多くの場合クラウドサービスという言葉が用いられた場合はこのパブリッククラウドを指す．

パブリッククラウドでは，提供される全てのリソースがクラウドサービスプロバイダのデータ・センター内に置かれ，サーバ仮想化技術やネットワーク仮想化技術によって隔離されるものの，ハードウェア機器のリソースは他のユーザと共有している場合が多い．

2.3.2　プライベートクラウド

プライベートクラウドとは，企業・教育機関等である特定のユーザに対して提供されるクラウドサービスを指す．前述のパブリッククラウドはハードウェア機器等のリソースを自身で持たないのに対し，プライベートクラウドでは自身でリソースを持ち管理・運用を行う．

プライベートクラウドという言葉が用いられた場合，自身のデータセンター内で全てのリソースを管理・運用するオンプレミス型を指すが，クラウドサービスプロバイダがリソースを持つホスティング型のプライベートクラウドも登場している．

2.3.3　ハイブリッドクラウド

ハイブリッドクラウドとは，前述のパブリッククラウド・プライベートクラウドの欠点を補うため，双方の利点を取り入れたクラウドサービスを指す．

パブリッククラウドは費用対効果・利便性の高さから様々なところで活用されているが，基幹システムのデータ (顧客情報や機密情報) 保存等の厳重管理が必要なケースや，現行保持しているリソースからの移行を考慮した際にシステム全てをパブリッククラウドに移行することは困難な場合がある．一方プライベートクラウドは，キャンペーンサイト等の一時的な大量の処理のためにインフラ投資をすることは難しいが，パブリッククラウドあれば必要なリソースのみ用意することが容易なため効率的といえる．

このような場合に「セキュリティ要件等によってクラウド化が容易でないシステム」はプライベートクラウドに，「クラウド化をしたほうが効率化が図れるシステム」はパブリッククラウドに置き，双方を連携することで柔軟なシステムを構築するという考え方がハイブリッドクラウドである．

2.4　クラウド基盤技術の発展

近年，クラウドコンピューティングの普及とともに，その構成技術の発展も著しいものとなっている．特に Docker に代表されるコンテナによる軽量な仮想化技術が出現し，多くのクラウドサービスが導入するなど注目が集まっている．さらに，Docker を中心とした様々なプロダクトも出現しており，様々な企業・団体による開発競争が繰り広げられている．

2.4.1　コンテナ型仮想化機構 – Docker

Docker とは

Docker とは，Docker,Inc が開発したコンテナ型の仮想化機構である．Docker を使用することにより，オーバーヘッドの少ないコンテナ内で隔離された環境を簡単に用意することができる．

元々 Docker は同社が提供していた PaaS のサービス基盤として開発されたという背景を持つ．

一般的に PaaS 型のクラウドサービスでは，アプリケーションの実行フレームワークや各種ライブラリ・ミドルウェア等の環境をクラウド側で用意するが，サービスの利用者によっては自身のアプリケーションで必要なライブラリ等が存在しないなど，用意されている環境では十分でない場合がある．そこで，利用者が独自の PaaS 環境を自由に構築し，その上で動かすアプリケーションもパッケージング可能なように作成されたコンポーネントの一つが Docker である．そのため，Docker はあくまで「アプリケーションとその実行環境」をまとめてパッケージングしたアプリケーションコンテナの役割を果たす

ために生まれたものであり，その目的はハイパーバイザ型仮想化のようなゲスト OS を提供することではないと言える．

コンテナ型仮想化

コンテナ型仮想化とは，Linux カーネルが持つ複数の機能を組み合わせ，ユーザ空間を「コンテナ」単位で分割する技術である．代表的なものとして，「Docker」，「Linux Containers[15](LXC)」，「rkt[16]」などが挙げられる．さらに，コンテナ型仮想化技術の標準化に向けて「Open Container Projecto」が発足し，標準仕様 App Container の開発が進められている [17].

ハイパーバイザ型仮想化では，仮想ハードウェア上にゲスト OS を動作させることによって完全な仮想化を実現しているのに対し，コンテナ型仮想化では，ホスト OS 上に隔離された空間を作り出すことによって，外部からは独立した物理マシン，もしくは仮想マシンであるかのように見せかけている．

ハイパーバイザ型仮想化とコンテナ型仮想化仮想化の違いを図 **2.3** に示す．図 **2.3**(左) はハイパーバイザ型仮想化の場合の環境である．この環境では，ホスト OS 上でハイパーバイザを動作させ，そのハイパーバイザがハードウェアのエミュレーションを行う．これによりその上でゲスト OS を動かすことで完全に独立マシンとして動作させることができる．

それに対し図 **2.3**(右) のコンテナ型仮想化では，ホスト OS とコンテナがカーネルを共有することでハイパーバイザ型のゲスト OS に当たる部分をそれぞれプロセスとして扱う．その際，それぞれのコンテナ内で表 **2.1** に示すような様々なリソース空間を隔離することで外部からは独立したマシンとして扱うことができる．ハイパーバイザ型に比べハードウェアエミュレーションを行うオーバーヘッドがなくなるため，起動が高速になるという特徴もある．

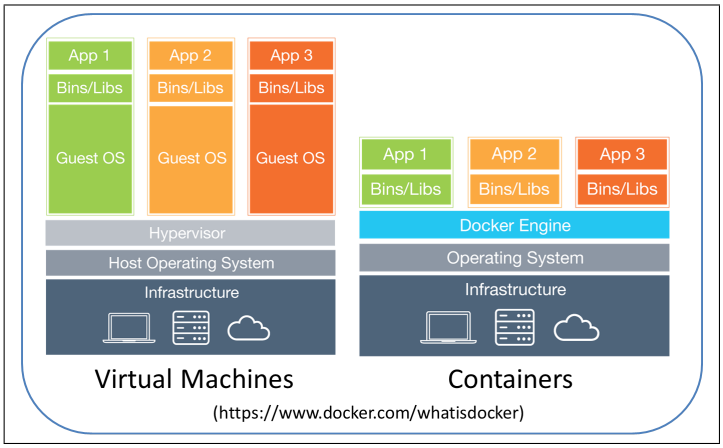

図 2.3　ハイパーバイザ型とコンテナ型の比較

表 2.1　コンテナによって隔離されるリソース

リソース	説明
プロセス	コンテナごとに固有のプロセステーブル
ディレクトリ	コンテナごとに固有のルートディレクトリツリー
ネットワーク	コンテナごとに仮想 NIC
CPU	コンテナごとに使用可能な CPU コア数，CPU 割当時間
メモリ	コンテナごとに使用可能な物理メモリの容量

コンテナによるプロセス実行環境の隔離

プロセス実行環境の隔離は，Namespaces[18] によって行われる．ユーザプロセスが動作する空間を隔離する Namespaces はコンテナ型仮想化を実現する上で核と言える機能であり，Docker では以下のような Namespaces を用いてプロセス実行環境を隔離している．

- PID Namespaces
  - プロセスが動作する空間を生成・隔離する
- User Namespaces
  - UID/GID 等のユーザ/グループ情報を隔離する
- Mount Namespaces
  - プロセスから見えるファイルシステムのマウント空間を隔離する
- UTS Namespaces
  - uname と Posix メッセージ・キューの空間を隔離する
- IPC Namespaces
  - System V IPC と Posix メッセージ・キューの空間を隔離する
- Network Namespaces
  - ネットワークデバイスや IP アドレス，ルーティングテーブル等を隔離する

また，プロセスを割り当てるリソースの隔離は cgroups(Control Groups)[19] によって行われる．Docker では，複数ある cgroups の機能から割り当てる CPU 時間を制限する”cpu”および割り当てる CPU を制限する”cpuset”，メモリの使用量を制限する”memory”を使用してリソースの隔離をしている．

Docker イメージによるディレクトリの隔離

Docker がコンテナごとにディレクトリを隔離する仕組みは，基本的には従来から使用されている”chroot”と同様である．Docker を利用する際にコンテナイメージとして用意する Docker イメージには，コンテナに割り当てるディレクトリの内容がイメージとしてそのまま格納されている．Docker では，そのイメージをコンテナのルートディレクトリに割り当てることによって，コンテナから見えるルートディレクトリの隔離を実現している．

仮想ブリッジによるネットワークの隔離

Docker のネットワークの隔離は仮想 NIC と仮想ブリッジによって実現される．Docker のネットワーク構成の全体像を図 **2.4** に示す．それぞれのコンテナには，サーバの物理 NIC とは別に専用の仮想 NIC が割り当てられるため，コンテナの内部からはこの仮想 NIC のみが見える状態となる．そして，この仮想 NIC は仮想ブリッジに接続され，コンテナ同士はこの仮想ブリッジを経由することで，お互いにリンクすることも可能になる．

コンテナが外部ネットワークと通信する場合，仮想ブリッジと物理 NIC の間で IP マスカレードが行われ，物理 NIC の IP アドレスを代表アドレスとして共有する形で外部ネットワークに接続する．また，外部ネットワークからコンテナに通信する場合，コンテナを起動するホスト側でポートフォワーディングが行われ，物理 NIC の特定ポートに到達したパケットを特定コンテナの仮想 NIC 上の特定ポートに転送するというルーティングがされるため，通信が可能となる．これにより，外部ネットワークのホストから見ると自身がアクセスしているサーバの実体がコンテナかどうかは意識する必要はなく，通常のアプリケーションが特定ポートで接続を待ち受けているように振る舞う．

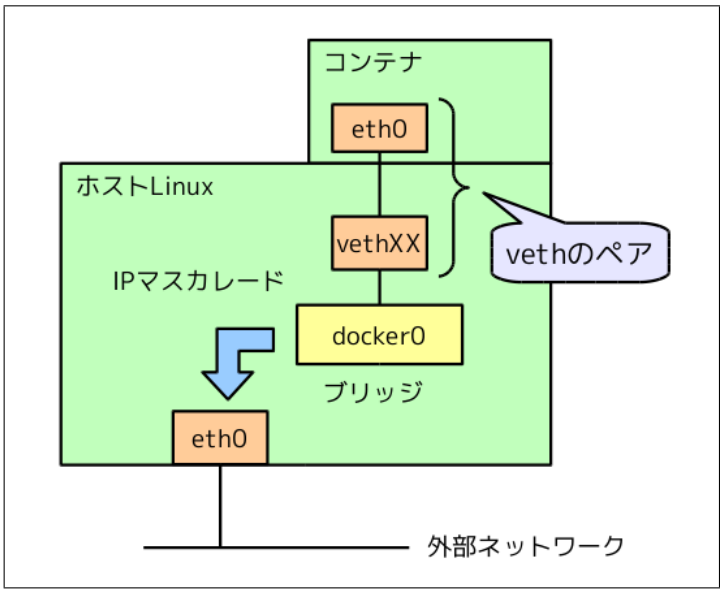

図 2.4　Docker のネットワーク構成 ([20] より)

Dockerfile による高ポータビリティの実現

Docker が近年注目を浴びている一つの要因としてその高いポータビリティが挙げられる．Docker では，Dockerfile というファイルを作成し，そのファイルの情報を元にコンテナのイメージをビルドすることができる．この Dockerfile はプレーンテキストファイルであり，バージョン管理システム等と非常に相性が良く，昨今の”Immutable Infrastructure”[*1]思想に基づいた”Infrastructure as Code”[*2]を実現する上で非常に都合のいいツールであったと考えられる．

[*1] 一度サーバを構築したらその後はサーバのソフトウェアに変更を加えないこと
[*2] インフラ構成をテキストでコード化し，コードリポジトリ上で管理する手法

また，Docker ではレイヤ型のファイルシステムを採用しており，Dockerfile におけるビルドは行ごとにレイヤとして管理される (図 **2.5**)．この仕組みによって，一度ビルドした工程 (行数) はレイヤとして保存され，二度目以降のビルドではそのレイヤがそのまま利用されるため，共通のビルド項目が省略され，コンテナイメージを高速に生成可能である．また，この仕組み上同じ工程のビルドは二度されないため，プロビジョニングにある程度の冪等性を持たせることが可能となり，これも Infrastructure as Code を実現する上で重要な要素であると言える．

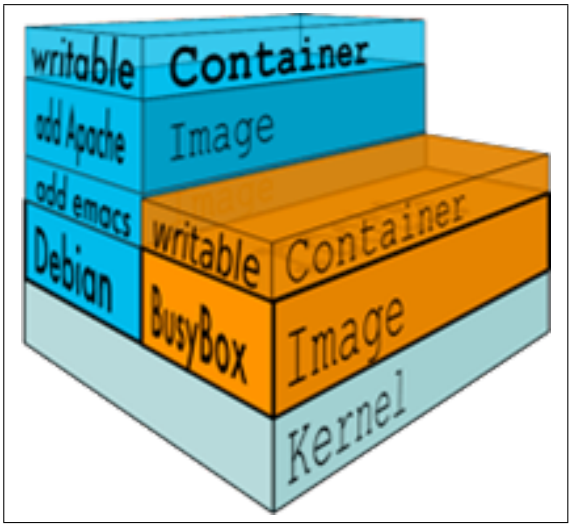

図 2.5　レイヤ型ファイルシステムのイメージ ([3] より)

2.4.2　Docker を中心としたプロダクト

Docker の普及にともなって様々なプロダクトも登場している．Docker,Inc が提供している多種多様なプロダクトから様々な企業・団体が提供しているプロダクトまであり，その開発競争は一段と激しくなっている．

Docker Engine

Docker Engine[21] とは，Docker を利用するためのコアとなるランタイムである．Docker コンテナの生成・実行・管理，Docker イメージのビルドなど Docker を利用する命令はこの Docker Engine に対して行われ，すべてを管理・実行している．もともと Docker と呼ばれていたものが Docker Engine となり，現在 Docker という名称は，様々プロダクトを合わせた全体のプラットフォームを指している．

この Docker Engine は Client/Daemon 構成をとっており，内部では REST API を用いた HTTP で通信を行っている．また，この REST API は外部ライブラリからも使用可能であり，現在では各種プログラミング言語でこの API を利用できるライブラリが提供されている．

Docker Registry

Docker Registry[22] とは，Docker Engine で利用される Docker イメージを管理・共有するためのリポジトリである．複数の Docker Engine 環境で Docker イメージを共有したい際には，この Docker Registry に対して Push し，必要になった際には Docker Registry から Pull をしてくる形で共有する．この Docker Registry は単純な REST API を提供する Web サーバとして動作し，高い可用性を持っている．

Docker Registry 自体はオープンソースとして提供されているが，Docker,Inc が提供する公式のリポジトリ”Docker Hub”やオンプレミスやプライベートクラウドなどのエンタープライズ向けにセキュリティを強化した”Docker Trusted Registry”なども登場している．

Docker Compose

Docker Compose[23] とは，複数のコンテナで動作するアプリケーションの依存関係情報を 1 つのファイルとしてまとめることで，コンテナ同士の連携を容易にするものである．Docker で Web アプリケーションを動作させる際に，すべてをひとつのコンテナにまとめるのではなく，アプリケーションが動作するアプリケーションコンテナ，データベースが動作するデータベースコンテナのように複数のコンテナに分割し，それぞれを連携させることで 1 つの動作をする形を取ることが多い．しかしこのような場合，コンテナの数が多くなると連携が複雑になり，複数の環境で再現することが困難になりやすい．Docker Compose では，起動に必要なコンテナ情報や連携に必要な情報，環境変数等を 1 つのファイルに記述し，それをまとめて起動することができる．そのため，複数の環境で再現することが容易であるという Docker のメリットを損なわずに規模の大きなアプリケーション開発が容易となる．

また，Docker Compose ではアプリケーションコンテナのみ数を増やし，スケールアウトさせるといった操作も容易であるためコンテナの管理が容易になるというメリットもある．

Docker Swarm

Docker Swarm[24] とは，Docker Engine を複数のホストで動作させ，それらを連携させることによって Docker Engine のクラスタを構成・管理するためのものである．Docker Engine を複数組み合わせ，1 つのリソースプールとして束ねることで大規模な Docker 環境を構築し，コンテナをどのホスト上で動作させるのかをスケジューリングすることができる．

クラスタに属するホストはクラスタを管理する Swarm Manager とクラスタを形成する Swarm Node のいずれかになり，クラスタへの命令は Swarm Manager を経由して行われる．

Docker Machine

Docker Machine[25] とは，Docker 環境を構築するためのものである．Docker を動作させるのに必要な Docker Engine のインストールや各種設定，Docker Swarm の設定などをコマンドラインから自動で行うことができる．

CoreOS

CoreOS[26] とは，以下のような特徴をもった Linux ディストリビューションである．

- 小さくかつ堅牢なコア
- 安全なアップデート
- アプリケーションコンテナ (Docker)
- クラスタリング
- 分散システムツール
- カスタマイズ可能な SDK

余計なものをそぎ落とし，コンテナを動かすことに特化したディストリビューションであり，追加でアプリケーションを動かす際でもコンテナで動かすことを前提としている．標準でクラスタリング機能を持っているため，コンテナを動かすためのクラスタを組むことが容易であるという点も大きな特徴である．

Kubernetes

Kubernetes とは，Docker コンテナによるクラスタ構築のためのスケジューリングを行うオープンソースプロジェクトである．もともと Google が提供しているサービス群で使用していたコンテナ運用技術をオープンソースとして公開し，現在は Google，Docker，Intel，IBM など多くの有名 IT 企業や団体で構成される Cloud Native Computing Foundataion に開発が移譲されている．

Docker Compose のように複数のコンテナを組み合わせて利用する際に，コンテナクラスタを Pod と呼ばれる単位でまとめ，複数の Docker ホスト上のどこかにまとめて配置する．こうすることでネットワークやストレージを共有するようなコンテナクラスタの場合パフォーマンスに影響がでないよう配置することができる．

また，コンテナの管理機構も備わっているため，コンテナの状態を確認することやアクセスの分配なども可能である．

Google Container Engine

Google Container Engine[27] とは，Google Cloud Platform が提供する Docker コンテナの実行を支える強力なクラスタマネージャおよびオーケストレーションシステムである．ユーザが定義する要件 (CPU やメモリなど) に基づいてコンテナをクラスタにスケジュールし，自動的に管理する．Kubernetes を基盤として構成されている．

Amazon EC2 Container Service

Amazon EC2 Container Service[28] とは，Amazon Web Services が提供するスケーラビリティが高く高性能なコンテナ管理サービスである．クラスタ全体のコンテナ配置をスケジューリング可能で簡単な API で Docker 対応のアプリケーションの起動・終了することができる．

第 3 章

Docker を用いた異種クラウド間連携

3.1　Docker を用いたハイブリッドクラウド機構

本学ではコンテナ型仮想化技術 Docker を用いて，プライベートクラウドであるクラウドサービスセンターとパブリッククラウドである Amazon Web Services(以下 AWS) を連携したハイブリッドクラウド機構を構築している [7][29].

このハイブリッドクラウド機構は軽量なコンテナ型仮想化技術を用い，図 **3.1** のようにプライベートクラウドとパブリッククラウド双方に負荷を分散している．オンプレミス型のプライベートクラウドでは最小限のリソース確保のみを行い，負荷が増加する期間には図 **3.2** のように従量課金モデルであるパブリッククラウドに対してさらなる負荷分散をすることで，閑散期のコストカットを実現している．また，プライベートクラウドでのサーバ運用を最小限に留めることにより，故障管理・運用による負担も軽減できる．このハイブリッドクラウド機構を”Juneau”と呼称している．

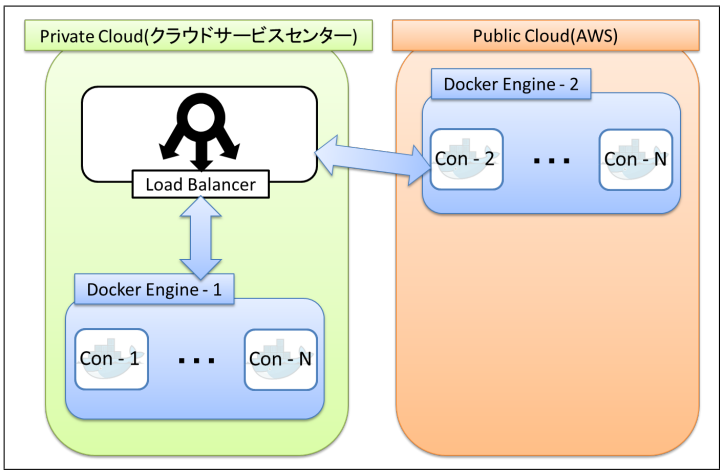

図 3.1　Juneau の動作イメージ図 (閑散期)

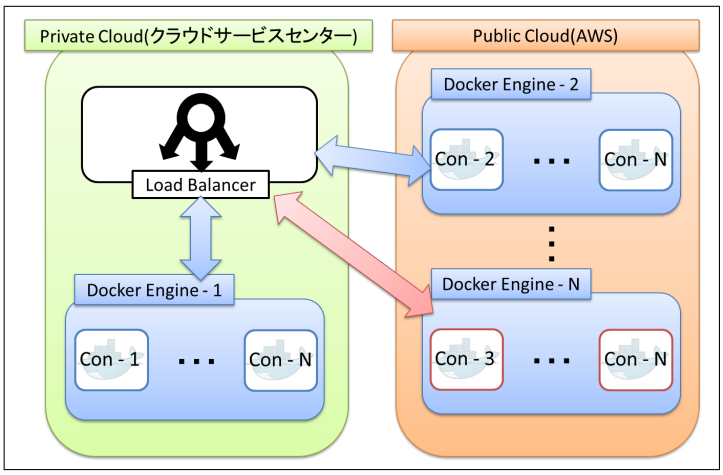

図 3.2　Juneau の動作イメージ図 (ピーク時)

3.2　Docker を用いた異種クラウド間連携の課題

プライベートクラウドとパブリッククラウドを連携させたハイブリッドクラウドのような異種クラウド連携では多くの場合図 **3.3** のような構成となる．本学のハイブリッドクラウド機構も同様の構成をとっている．

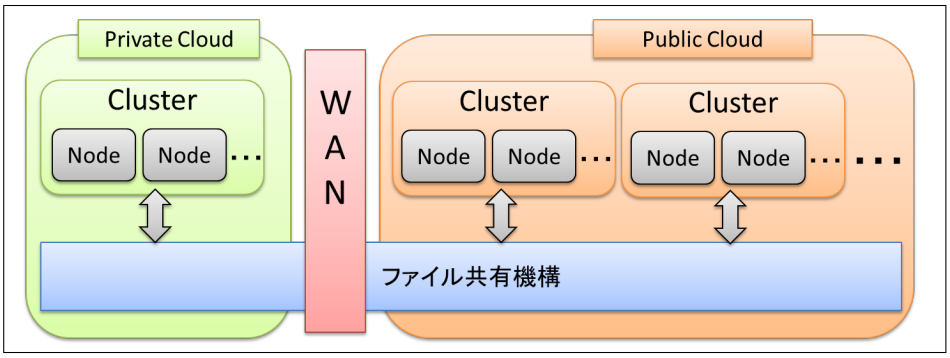

図 3.3　ハイブリッドクラウドの構成イメージ図

クラスタ間においても Docker のコンテナイメージやコンテナ内で利用するファイルなどを共有する必要がある．しかし，異種クラウド間連携ではネットワークが異なり，Infiniband[30] 等の高速なネットワークではなく通常の WAN で接続される．そのため，ファイル共有の際の同期遅延，同期トラフィックの増大，セキュアな経路確保などの課題を解決しなければならない．

現行のハイブリッドクラウド機構の構成では図 **3.3** のファイル共有機構に Docker Registry を用いた Push/Pull 方式を採用している．Push/Pull 方式では，共有すべきコンテナイメージに変更があった際に Docker Registry に対して Push をすることで共有を可能にし，必要となったら Docker Registry から Pull をすることで利用可能な状態にしている．そのため，コンテナ内で利用するファイルをすべてコンテナイメージとして保存する必要があり，少しの変更のたびにコンテナイメージを作成しなければ

ならず柔軟性にかけるという課題がある.

Docker の利用に限らず異種クラウド間で連携する際にはファイル共有機構に大きな課題を抱えると考えられる.

第 4 章

疎結合マルチクラスタにおけるストレージ基盤の検討

4.1　必要とするストレージ基盤

図 **3.3** のようにネットワークが異なり，Infiniband[30] 等の高速なネットワークではなく WAN で接続されていたり，そもそも管理者が異なるなど疎に結合している複数クラスタを疎結合マルチクラスタと定義し，それに適したストレージ基盤について検討する．

疎結合マルチクラスタに適したストレージ基盤として求められる要件は以下の様なものが挙げられる．

- 柔軟なファイル操作
- ネットワークトラフィックの抑制
- 各種パブリッククラウドとのシームレスな連携

これらの要件を満たすようなストレージ基盤が求められる．

4.1.1　柔軟なファイル操作

前述の Docker による利用に限らず，様々なアプリケーションで共有されたファイルを利用することを想定すると，ファイルシステムの形でも利用できることが求められる．ネットワークを介してファイルが共有できる分散ファイルシステムの形である必要がある．

4.1.2　ネットワークトラフィックの抑制

多くのパブリッククラウドでは通信量従量課金制をとっている．パブリッククラウドを利用することでインフラ投資における費用対効果が上がることが見込まれるが，クラスタ間のファイル共有による同期トラフィックが重なり，逆にコストが増大することが考えられる．コストを抑えるためにもネットワークトラフィックを抑える必要がある．

さらに，ネットワークトラフィックが増えることにより同期遅延による不整合も考えられる．不整合を起こさないためにもネットワークトラフィックを抑える必要がある．

4.1.3　各種パブリッククラウドとのシームレスな連携

現在パブリッククラウドベンダーは大小様々なものが登場しており，主要なものでも Amazon WebServices，Google Cloud Platform，Microsoft Azure[31] などが挙げられる．それぞれ提供サービス，料金形態など特徴が異なるため，用途に合わせて様々なパブリッククラウドの機能を組み合わせることができることが望ましい．

4.2　既存のストレージ基盤

現在，データセンター等のクラスタ環境で利用されているストレージ基盤には主に以下のようなものがある．

4.2.1　NAS – NFS

NAS(Network Attached Storage) はローカルストレージにアクセスするのと同様にネットワーク内のファイルサーバ上にあるファイルにクライアントコンピュータからアクセスするための分散ファイルシステムおよびプロトコルの総称であり，代表的なものとして NFS(Network File System)[32] がある．

NFS ではファイルサーバ上で NFS Server を動作させ，クライアントから NFS Client を経由してファイルアクセスをする．

LAN 内でファイル共有する際には，簡単に使えるため利用されるケースが多い．

4.2.2　SAN – ファイバチャネル

SAN(Storage Area Network) は複数のコンピュータとストレージ間を結ぶ高速なネットワークである．ストレージ装置とサーバなどのコンピュータをつなぐためのネットワークで，専用のプロトコルや機器で構築される．LAN などの汎用的なネットワークからは切り離されて独立して運用されるため LAN 内のネットワークに負荷をかけずに大規模なストレージを構築できる．

機器間の通信でファイバチャネルプロトコルを用いて光ファイバーケーブルやファイバチャネルスイッチなどの機器を組み合わせて構築される FC-SAN が代表的である．

高速で大規模なストレージを作ることが可能だが，機器が高額でコストがかかるケースが多い．

4.2.3　GFS

GFS(Google File System)[33] は Google が開発しているデータセンタ向きの大規模分散ファイルシステムである．安価な PC を複数台組み合わせることで分散ファイルシステムを構築しており，サーバが故障することを前提で故障してもデータが喪失しないよう分割して複数サーバに保存するよう設計されているという特徴がある．

チャンクと呼ばれる 64 メガバイトのサイズに固定されたデータに分割して保存され，上書きすることはほとんどなく，通常は追記および読み込みを中心に利用される．

GFS はマスターノード，チャンクサーバ，クライアントの 3 種類で構成され，マスターノードがディレクトリツリーやチャンク情報の管理を行い，チャンクサーバはマスターノードにぶら下がる形になっ

ている．

実際のファイルアクセスはクライアントから行い，POSIX 互換ではないが，一般的なファイル API が提供されている．

同社の主要なサービスである検索エンジンや各種サービスのデータストレージとして利用されている．

4.3　ストレージ基盤の検討

上記のようなストレージ基盤を疎結合マルチクラスタにおけるストレージ基盤として検討を行ったが，ネットワークトラフィックとパブリッククラウドとの連携が大きな課題となる．

もともとこれらのストレージ基盤は LAN 内で共有データを共有することを想定していることが多く，ネットワークトラフィックによるコストを想定していない．また，ファイバチャネルなどはそもそも機器自体にコストが掛かってしまうため，コストを抑えたい疎結合マルチクラスタのようなクラスタ環境には向かないと考えられる．

さらにパブリッククラウドとの連携を検討する際にはセキュアな通信経路の確保が必要となる．これらのストレージ基盤を使う際には拠点間 VPN などの特別な機器で LAN と同様にネットワークを扱う必要がある．パブリッククラウドの中には拠点間 VPN に対応しているものもあるが，さらなるコストがかかることや，パブリッククラウドベンダーの選択肢を狭めてしまうという欠点もある．

ストレージ基盤はエンタープライズ向けも含め多くのコストがかかってしまうケースが多いため，疎結合マルチクラスタ環境のように通信に多くのコストがさらにかかってしまうケースには現状対応出来ていないと考えられる．

これらのことから疎結合マルチクラスタ環境ではストレージそのもののコスト，ファイル同期にかかる通信コスト，パブリッククラウドとの連携を考慮した新たなストレージ基盤が必要であると考えられる．

第 5 章

疎結合マルチクラスタ向きストレージシステムの提案

5.1　分散ストレージシステム – Elton

疎結合マルチクラスタにおいて，既存のストレージ基盤を利用する際には前述 (4.3 節 – p.18) のように通信を含めたコストとパブリッククラウドとの連携に課題を抱えている．

そこで，コミットしたタイミングでデータ共有可能になる分散ストレージシステムを提案する．ファイル同期に対してコミットの動作を加え，ファイルの取得を行わないかぎり変更が伝搬しないようにすることで変更による影響を局所化する．こうすることで，アプリケーションやデータの特性により不要となる同期を減らすことができるため，同期トラフィックを抑えることができる．さらに，コミット時にバージョンを発行し，データを普遍に管理することでキャッシュが容易となり，一度取得を行えばローカルファイル同様に扱えるため SSD 等の高速なストレージとの相性が良くなりパフォーマンスの向上が期待できる．このような仕組みをもつコミットベースの分散ストレージシステム (Commit-based Distributed Storage System) を Elton と呼称する．

5.1.1　不変 (Immutable) なデータ管理

通常ファイルに対して更新が行われると図 **5.1** のようにそのファイルが上書きされ，変更前の状態に戻ることはできない．それに対し，不変なデータ管理では，更新ごとに図 **5.2** のようにバージョンをつけ，別のファイルとして作成され，以降は作成したファイルは変更することなく保存される．

こうすることで，分散環境において分散キャッシュを行ったとしてもキャッシュの有効期限のタイミング等を気にする必要がなく，バージョンを指定して取得すればファイルの変更がないことが保証されるため必ず正しいデータを共有可能になる．さらに，更新の歴史を残すためバックアップ/リストアが容易になることや永続キャッシュが可能になるのでキャッシュを通常のローカルファイル同様に SSD 等の高速なディスクで扱うことができるというメリットもある．

図 5.1　通常 (Mutable) のデータ管理のファイル更新イメージ図

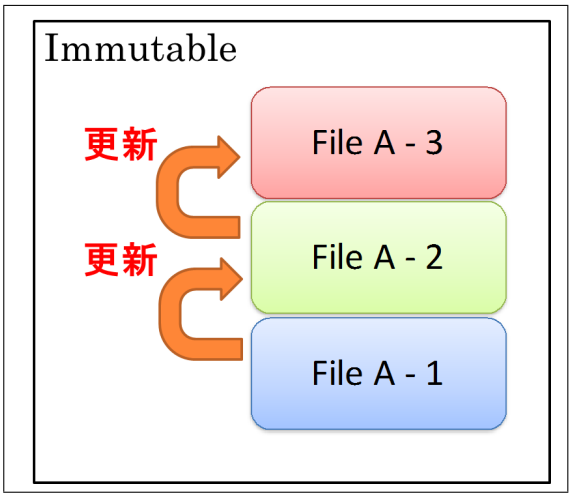

図 5.2　不変 (Immutable) なデータ管理のファイル更新イメージ図

5.1.2　ファイル共有方法

NFS によるファイル共有では通常図 **5.3** のような構成をとり，NFS Client からのファイル操作がすべてネットワークを通して NFS Server に対して実行される．

NFS Client 側で一度読み込んだファイルをキャッシュすることが可能だが，他で変更があるたびにキャッシュの削除命令が送られるなど，疎結合マルチクラスタ環境では同期トラフィックが増大し，コストが高くなったり，同期遅延による不整合が起こる原因となる．

そこで，Elton の構成を単一クラスタ内で図 **5.4** のような構成にする．中央にメタデータを管理するメタデータサーバを用意し，そこに各ノードがぶら下がる形になる．メタデータサーバには各ノードで保存されているファイル Key と最新バージョン番号，ファイルの各バージョンがどこのノードで管理されているのかが保存される．

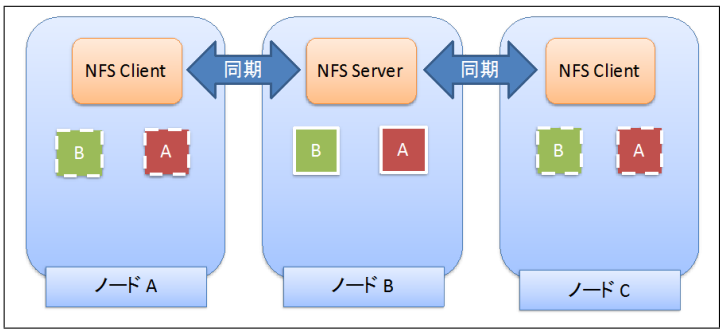

図 5.3　NFS によるファイル共有方法

具体的なファイル共有の動作としては，図 **5.4** のノード A においてファイル A のバージョン 2 が必要になった際には，まずローカルキャッシュを参照し，この場合キャッシュがないためメタデータサーバを経由してファイルの取得を行う．ファイル A のバージョン 2 が管理されているのはノード C であるため，ノード C からファイルを取得してローカルキャッシュすることでデータの共有を行う．

同様にノード A でファイル A の更新を行う際には，まずメタデータサーバに対してファイル A の最新バージョンを問い合わせ，最新バージョン +1 である 3 が返り，そのバージョンをもとにファイル A のバージョン 3 が作成される．作成後はメタデータサーバに対してファイル A のバージョン 3 をノード A で管理していることを伝えることですべての環境で共有可能な状態とする．

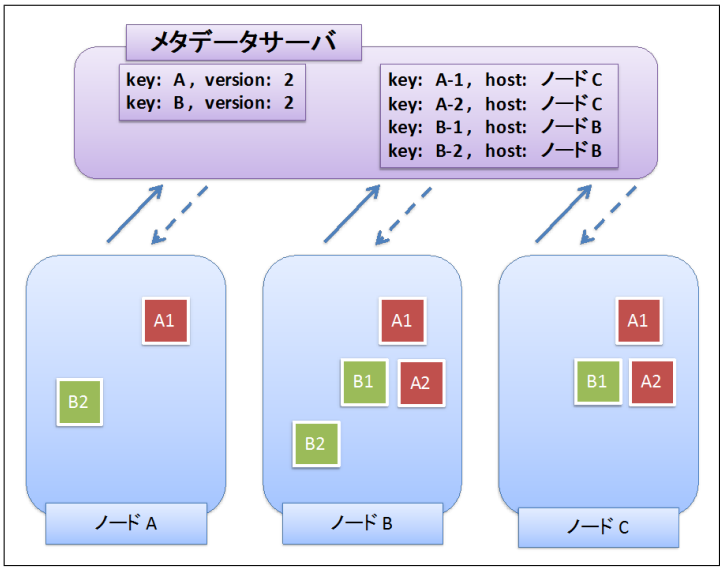

図 5.4　Elton によるファイル共有方法

疎結合マルチクラスタ環境では図 **5.5** のように前述の単一クラスタ構成をそれぞれのクラスタに用意し，メタデータサーバ同士を接続する．複数のクラスタにまたがったファイル共有でもメタデータサーバがそれを仲介することで単一クラスタ時と同様に共有することができる．

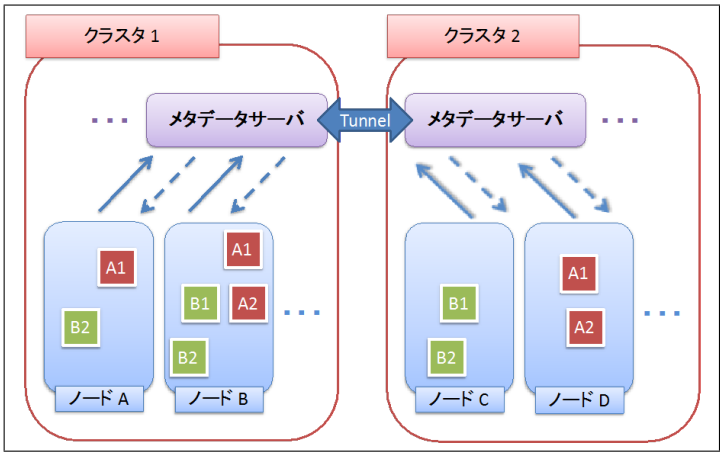

図 5.5　疎結合マルチクラスタ環境における Elton のファイル共有方法

5.2　Elton を用いた疎結合マルチクラスタ構想

従来 NFS 等を使ったクラスタ構成では，図 **5.6** のように中央に大規模なファイルサーバを用意し，各ノードがすべて接続できる必要があった．

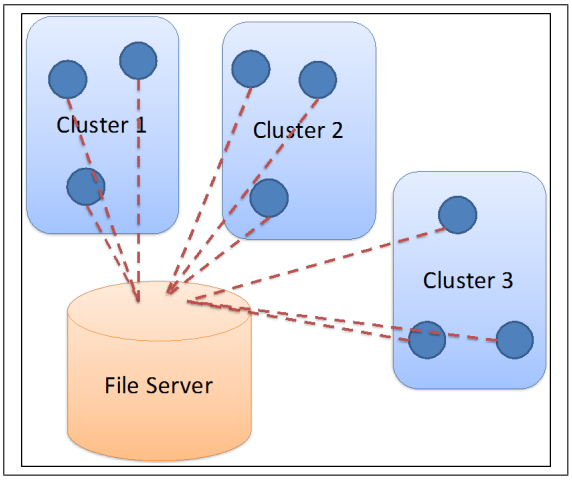

図 5.6　NFS を用いた従来のクラスタ構成

しかし，Elton を用いたマルチクラスタでは，各クラスタ内のメタデータサーバさえネットワークで繋がっていれば，各ノードで管理されるファイルを全体として共有する．こうすることで疎結合マルチクラスタでは従来よりも分散システム全体の構成を簡略化することができる．

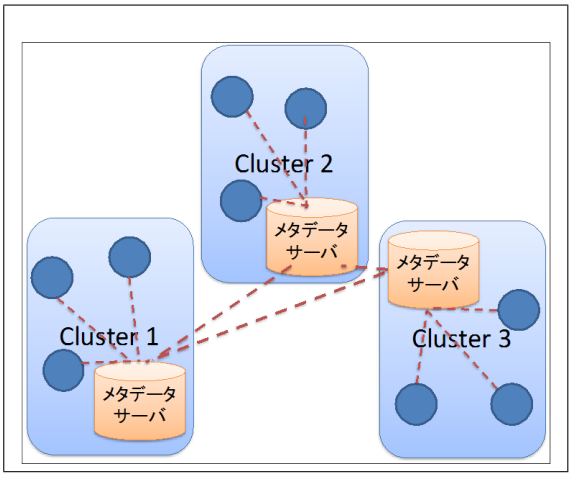

図 5.7　Elton を用いたクラスタ構成

5.2.1　ファイルシステムとしての利用

Elton ではファイルシステムとしてのインタフェースを提供する．ディレクトリツリーごと Elton で管理をし，コミットをした段階のファイルシステム情報が複数の環境で共有可能となる．さらに共有する際に最低限共有されるのはディレクトリツリー情報が入ったテキストファイルのみとすることで，不必要なファイル同期を行わない仕組みをとる．こうすることで，複数ノード，複数クラスタで共有しても必要なファイル同期のみに抑えることができる．

さらにこの共有方法はコンテナ型仮想化で有効利用できる．Docker でコンテナ同士でファイル共有する際には，コンテナイメージに元々ファイルを入れて共有する方法と Docker Volume マウントの機能を用いる方法がある．

コンテナイメージに元々ファイルを入れる方法では，イメージのビルドの段階でファイルをコンテナ内に保存し，そのイメージを元に複数のコンテナを動作させて共有する方法である．この方法の場合，共有することは簡単だが，変更があるたびにイメージをビルドする必要があり，管理が複雑化する．そこで一般的には Docker Volume マウントを利用する方法で，Docker Engine が動作するホスト上とディレクトリを共有することでコンテナ間でファイル共有する方法がとられる．しかしこの方法でも Docker Engine のホストが複数台になった際にはそのディレクトリを NFS 等の方法で共有する必要がある．

Elton を用いてディレクトリ共有を行えば，コンテナイメージにビルドする必要もなく，新たにファイル共有機構を用意する必要がなくなるというメリットがある．

図 **5.8** はファイルシステムとしての利用イメージである．コンテナ本体の情報は変えず Elton のファイルシステムを外付けし，メタデータサーバを経由してファイルシステム全体を共有する．こうすることでノード，クラスタ間をファイルシステムそのものが移動してくる感覚で利用することが可能になる．

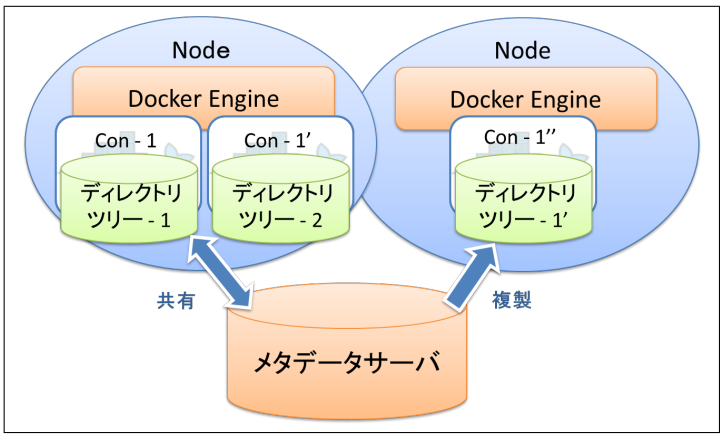

図 5.8　ファイルシステムとしての利用イメージ

5.2.2　分散ストレージレイヤとしての利用

Elton では HTTP のインタフェースも提供し，分散ストレージレイヤとして扱うこともできる．このストレージレイヤでは，複数ノード，複数クラスタ間やインターネットを介した CDN(Contents Delivery Network) を構築することができる．

CDN を構築することで，HTTP を介したマルチメディア配信が可能になり，サービス提供するアプリケーションの幅が大きく広がる．CDN でマルチメディア配信をする際には多くの場合キャッシュを利用して配信の高速化・負荷軽減を行っている．この時大きな問題となるのがキャッシュの破棄のタイミングである．配信するコンテンツに変更があった際にキャッシュの破棄命令を出すことで破棄させる方法や TTL[*1]を設定することで正しいコンテンツの配信を行う場合が多い．

しかし Elton を用いた CDN では，Elton のデータ管理は不変であるため，キャッシュ破棄という考え方が必要なく，取得したいファイルが意図せず古いファイルであるというケースは存在しない．そのため管理が容易な CDN を構築することが可能になる．

図 **5.9** は分散ストレージレイヤとしての利用イメージである．分散ストレージレイヤの利用では，各ノード・クラスタ内で動作させる Elton の HTTP サーバがインタフェースを提供する．この HTTP サーバはインタフェースの提供だけでなくローカルでキャッシュを行う．一度利用したファイルはローカルにキャッシュされ，以降のリクエストはこのキャッシュを用いて利用される．こうすることで各ノード・クラスタ内で頻繁に利用されるものがキャッシュされることになり，効率の良いコンテンツの配信が可能になる．

[*1] Time To Live

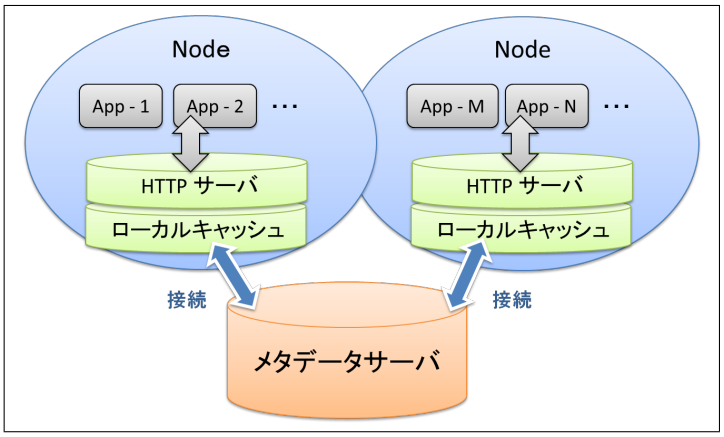

図 5.9　分散ストレージレイヤとしての利用

第 6 章

実現手法

6.1 設計

6.1.1 Elton Master

Elton Master は Elton 全体で管理されるファイルのメタデータ情報の管理，ファイル取得のプロキシを行うなど本機構の中心となるサーバである．このサーバはクラスタ内に最低 1 台動作することを前提とし，以下のような機能を持つ．

- バージョン管理・発行
  - 作成・変更時にバージョンを発行
- ファイルキャッシュの探索・プロキシ
  - ファイル取得リクエスト時にキャッシュを探索
  - 担当ノードに取得リクエストをプロキシ
- バックアップ・キャッシュ等のスケジューリング

6.1.2 Elton Slave

Elton Slave はファイルの作成・更新・取得を行うための HTTP インタフェースを提供するサーバである．Elton Master に対してファイルの作成・更新等のメタデータの送信，ファイル取得の際には Master へのリクエストの送信やローカルでのファイルキャッシュを行う機能を備えている．

6.1.3 Eltonfs

Eltonfs は Elton Master に保存されているファイルをファイルシステムインタフェースで提供するものである．大きな特徴として，ファイルシステムのディレクトリツリーも 1 つのファイルとして Elton で管理する点である．こうすることでさまざまなディレクトリツリーそのものを共有可能になる．

6.1.4 Docker Volume Plugin for Eltonfs

Docker Volume Plugin for Eltonfs は Eltonfs を使って Docker コンテナ内で利用するファイル群を共有するためのものである．Docker で提供されている Volume Plugin 機能 [41] を Eltonfs 用に用意して利用する．

6.2 実装

Elton 全体の実装で利用している環境を表 **6.1** に示す．Elton の開発・動作確認はすべて CentOS 7.2.1511，開発言語は Go 言語 1.4.2 で行っている．内部通信には Google が開発しているオープンソースの RPC ライブラリとフレームワークである gRPC[35] を利用している．gRPC は HTTP/2[36] を利用しているため通信をセキュアにすることが容易であるという点で採用した．また，Google が開発しているオープンソースのシリアライズフォーマットである Protocol Buffers[37] は gRPC のデータ層でデフォルトであるため採用している．

表 6.1　Elton の実装に用いた環境

項目	環境
OS	CentOS 7.2.1511 (Kernel 3.10.0-327.3.1.el7.x86_64)
開発言語	Go 言語 1.4.2
内部通信	gRPC
シリアライズフォーマット	Protocol Buffers v3.0.0-beta-1

6.2.1 Elton Master

Elton Master は，gRPC のサーバとして動作する．内部では，メタデータ管理用の Key/Value ストアを保持している．Key/Value ストアには BoltDB[38] を使用している．

Elton Master を複数動作させてクラスタ連携をする際には，Key/Value ストアの情報を共有する必要があるが，共有してしまうと同期トラフィックが発生してしまう．そこで，Elton Master それぞれで独自に Key/Value ストアを保持し，同期を行わない実装とした．

具体的なファイル管理方法はファイルを作成する際にまず，そのファイルにおいてバージョンを発行する代表となる Elton Master を決定する．以降の更新時は必ずバージョン発行の問い合わせを代表となる Master に対して行い，バージョン情報の整合性を保っている．こうすることで，少なくとも代表である Master に対して問い合わせを行えば必ずファイル情報を取得できるような仕組みになっている．

ファイル取得時はリクエストされるファイル Key・バージョン番号を元にまずリクエストを受けた Master のローカル Key/Value ストアを探索する．ローカルにファイル情報が無ければそのファイルの代表である Master に対してリクエストをプロキシし取得を行う．取得後はその情報をローカルの Key/Value ストアにファイル情報をキャッシュすることで以降は自身がリクエストを処理できるようにしている．

また，Elton Master はバックアップの機能も組み込んでいる．ファイル作成リクエスト時に自動でバックアップが行われるため，別にバックアップの仕組みを持たなくていいような構成にしている．

6.2.2 Elton Slave

Elton Slave は，Elton の HTTP インタフェースを提供するサーバとして動作する．Elton Master との接続は全て gRPC で行い，アプリケーション等とのやり取りを HTTP で中継する．

Elton においてファイルの保持を行うのはこの Elton Slave である．作成したファイルも取得したファイルも同様の永続キャッシュとして扱うことで，Elton Master を介した全体としてファイルを管理している．この仕組みによってキャッシュを有効活用し，ネットワークトラフィックを抑える仕組みをとっている．

6.2.3 Eltonfs

Eltonfs は，FUSE(Filesystem in Userapace)[39] と呼ばれるファイルシステムにアクセスするための処理をユーザ空間で行わせることができるライブラリを用いてファイルシステム実装を行った．FUSE でファイルシステムを実装するための Go 言語ライブラリとして go-fuse[40] を使用している．

ファイルシステム全体は Elton 全体で共有されている Read-Only の下層ディレクトリと更新分を扱う上層ディレクトリの 2 層からなる．下層のディレクトリツリーは，テキストファイルに保存されたパス情報・Elton 内部で扱われるファイル Key・バージョン番号・管理ホスト名等を含んだ JSON データをロードすることで構成される．このテキストファイルを Elton で共有することでファイルシステム情報を共有することができるため分散ファイルシステムとして動作させることができる．

変更分は上層のディレクトリに保存され，ある段階でコミットの操作をすることでディレクトリツリーを保存したテキストファイルが更新され，Elton で共有される．

ディレクトリツリー内で扱われているファイル自体も Elton で共有されるため，最初にファイルが利用される際に Elton Master 経由でファイルの取得を行い，以降はローカルキャッシュされるため通常のローカルファイルとして扱うことができる．Eltonfs も Elton Slave 同様ファイル永続キャッシュとして作成・取得したファイルを管理し，不要なファイル同期を抑え，ネットワークトラフィックを抑制している．

6.2.4 Docker Volume Plugin for Eltonfs

Docker Volume Plugin for Eltonfs は，Eltonfs を Docker のボリュームとしてコンテナ内にマウントするためのプラグインである．プラグインの実装には Docker のプラグインヘルパーの Go 言語実装である go-plugins-helplers[41] を利用している．

ボリュームプラグインとして実装することで同一ホストの Docker Engine 上では Eltonfs のファイルシステムを複数コンテナで共有することが可能になる．別ホストの Docker Engine 上では図 **6.1** のように Read-Only のファイルシステム部分は Elton Master を経由してファイルが共有される．変更分は上層のファイルシステムに保存され，コミットされると以降複数のノード，コンテナで共有することが可能になる．

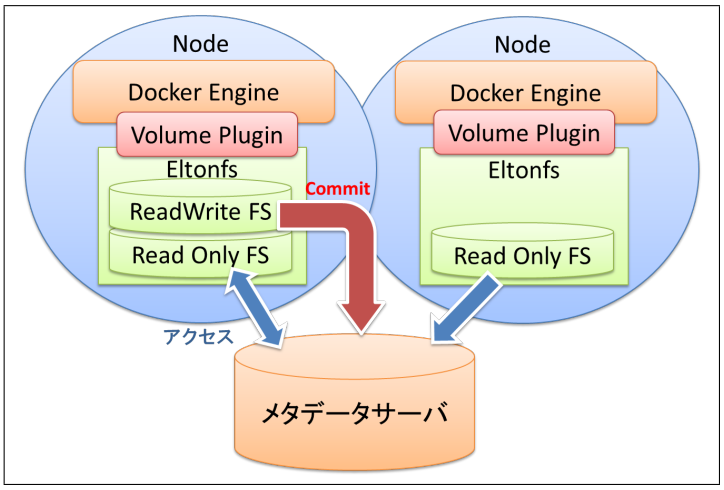

図 6.1　Eltonfs を用いた Docker におけるファイル共有

第 7 章

評価

7.1 性能検証実験

Elton の性能を検証するため，プライベートクラウドとハイブリッドクラウドを連携させるケースを想定して環境を構築，検証を行った．検証実験に使用した環境は表 **7.1**，表 **7.2** の通りである．

表 7.1 検証実験に利用したプライベートクラウド環境

項目	環境
Elton Master・NFS Server 用途サーバ	4 コア メモリ 4G VM * 1 台
Eltonfs・NFS Client 用途サーバ	4 コア メモリ 4G VM * 1 台
ディストリビューション	CentOS 7.2.1511 (Kernel 3.10.0-327.3.1.el7.x86_64)

表 7.2 検証実験に利用したパブリッククラウド環境

項目	環境
パブリッククラウドベンダー	Amazon Web Services EC2
リージョン	東京
Elton Master，Eltonfs・NFS Client 用途サーバ	t2.micro インスタンス * 1 台
ディストリビューション	CentOS Linux 7 x86_64 HVM EBS 20150928_01

検証の比較対象として NFS を採用し，同様の検証を行うことで評価を行う．

7.1.1 ファイルシステム性能評価

ファイルシステムとしての性能評価を行う．測定は bonnie++[42] を利用する．リスト **7.1**，リスト **7.2** のようなスクリプトを用いて Eltonfs・NFS Client 用途サーバに対して I/O 性能測定を行った．

測定は各環境のメモリサイズ (プライベートクラウド: 4GB，プライベートクラウド: 1GB) に合わせて s オプション (メモリサイズの 2 倍)，r オプション (メモリサイズ) をリスト **7.1**，リスト **7.2** のように調整している．

リスト 7.1　プライベートクラウド用測定スクリプト (privateperform.sh)

```
1: #!/bin/sh
2:
3: bonnie++ -d TARGET_DIR -n 0 -s 8192 -r 4096 -u root
```

リスト 7.2　パブリッククラウド用測定スクリプト (publicperform.sh)

```
1: #!/bin/sh
2:
3: bonnie++ -d TARGET_DIR -n 0 -s 2048 -r 1024 -u root
```

測定結果を表 **7.3**，表 **7.4** に示す．

表 7.3　Sequential Read の測定結果

環境	XFS	Eltonfs	NFS
プライベートクラウド	107170K/sec	86950K/sec	74165K/sec
パブリッククラウド	72060K/sec	63823K/sec	14833K/sec

表 7.4　Sequential Write の測定結果

環境	XFS	Eltonfs	NFS
プライベートクラウド	103250K/sec	103050K/sec	18565K/sec
パブリッククラウド	64285K/sec	67253K/sec	28709K/sec

表 **7.3**，表 **7.4** から，読み込み，書き込みともに Eltonfs の方が NFS よりも性能が高いことがわかる．ベースのファイルシステムである XFS と同等の性能が出ているため高速に動作している．

特にリアルタイム同期のためのファイルロックがある NFS は書き込み性能が低く，単純なローカルファイルとして扱うことが出来る Eltonfs は性能が高いことがわかる．

これらのことからリアルタイム同期が不要なケースにおいては Eltonfs は有用であると考えられる．

7.1.2　ネットワークトラフィック性能評価

ネットワークトラフィックの性能評価を行う．測定のシナリオは，以下の通りである．

1. dd コマンドを用い，プライベートクラウド上の Eltonfs・NFS Client 用途サーバの対象ディレクトリにファイルを書き込む
2. dd コマンドを用い，パブリッククラウド上の Eltonfs・NFS Client 用途サーバの対象ディレクトリに共有されたファイルを読み込む
3. 2 を 5 分間隔で 3 回繰り返す

プライベートクラウドでの書き込みはリスト **7.3**，パブリッククラウドでの読み込みはリスト **7.4** のようなスクリプトを用いて実行し，ネットワークトラフィックの測定を行った．

リスト 7.3　ファイル書き込み用スクリプト（ writeperform.sh ）

```
1: #!/bin/sh
2:
3: dd if=/dev/zero of=TARGETDIR/bigfile bs=8k count=8192
```

リスト 7.4　ファイル読み込み用スクリプト（ readperform.sh ）

```
1: #!/bin/sh
2:
3: dd if=TARGETDIR/bigfile of=/dev/null bs=8k
```

図 **7.1**，図 **7.2** は Elton Master と NFS Server のネットワークトラフィックを測定したグラフである．図 **7.1** においてトラフィックが上がっている 14:25 ごろは Eltonfs からファイルの読み込みを最初に行った時刻である．それに対し図 **7.2** はファイル書き込みを行った 14:46 ごろにトラフィックが一度上がり，以降読み込みを行った 14:50 ごろ，14:55 ごろ，15:00 ごろに再度トラフィックが上がってる．同様に図 **7.3**，図 **7.4** はパブリッククラウド上の Eltonfs と NFS Client のネットワークトラフィックを測定したグラフである．図 **7.3** においてトラフィックが上がっている 14:25 ごろはファイルの読み込みを最初に行った時刻である．それに対し図 **7.4** は読み込みを行った 14:50 ごろ，14:55 ごろ，15:00 ごろにそれぞれトラフィックが上がってる．

NFS では書き込みを行った際に NFS Server に対して通信があるのに対して Eltonfs では読み込みが行われるまで通信が行われていないことがわかる．それぞれ初回のトラフィックがほぼ同等であることグラフからがわかり，読み込みで毎回通信のある NFS の方が全体で見るとトラフィックが多くなっていることがわかる．これらのことから，Elton を用いたファイル共有ではネットワークトラフィックを抑えられていることがわかる．また，書き込み時ではなく読み込み時にファイル共有が行われていることから，不要なファイル同期によるネットワークトラフィックも抑えられていることがわかる．

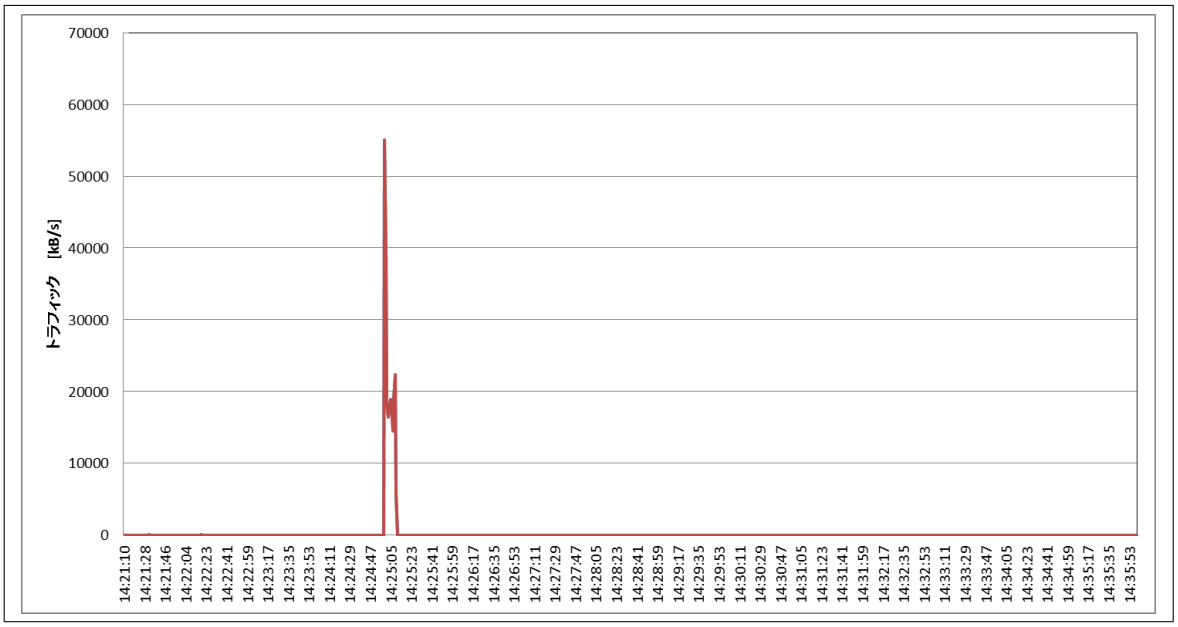

図 7.1　Elton Master ノードのトラフィック

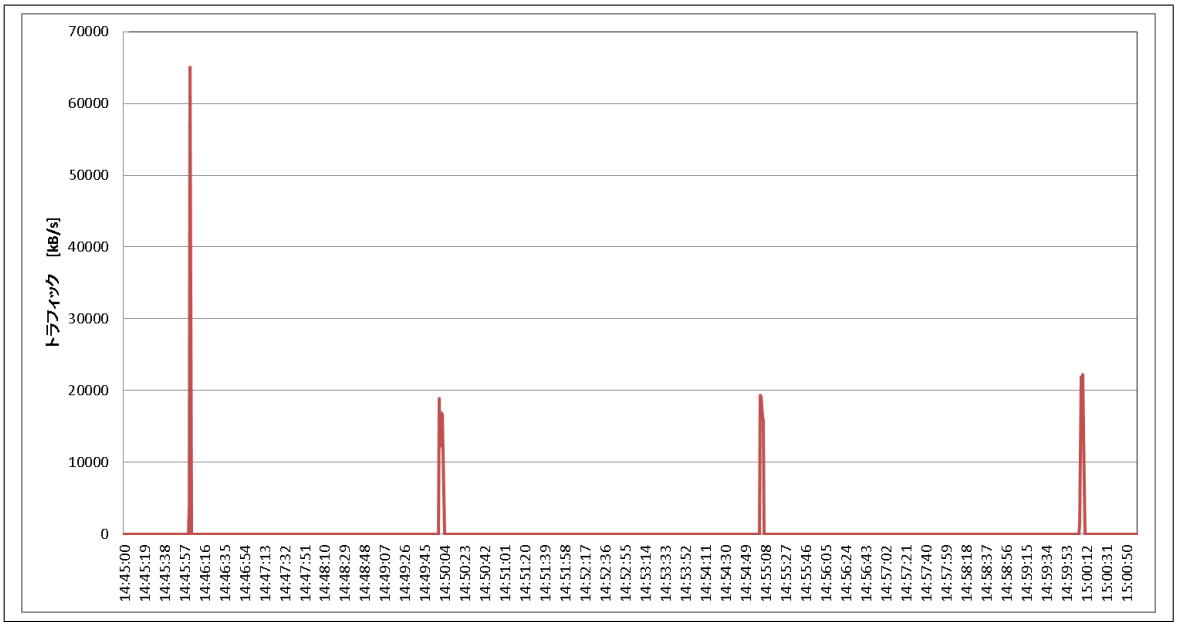

図 7.2　NFS Server ノードのトラフィック

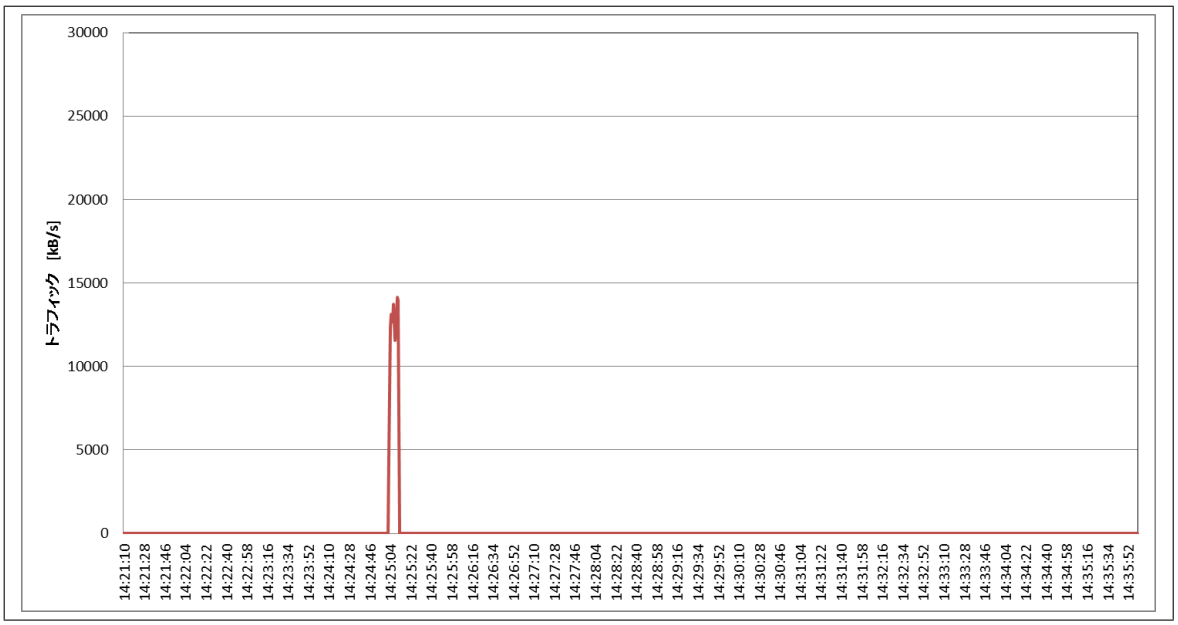

図 7.3　パブリッククラウド上の Eltonfs ノードのトラフィック

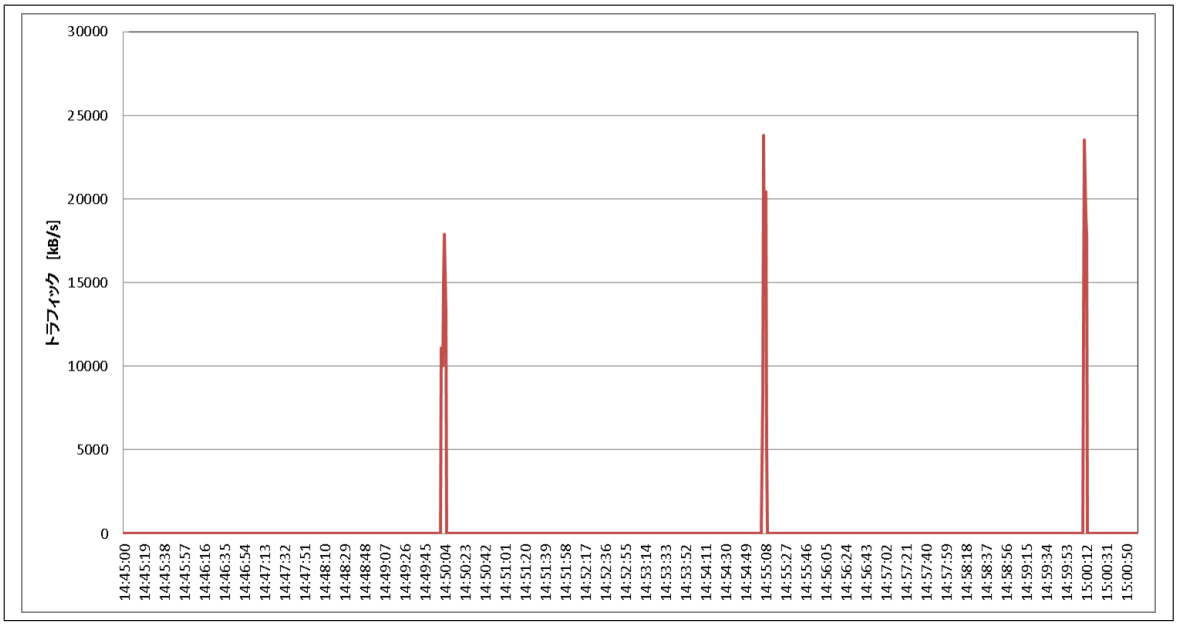

図 7.4　パブリッククラウド上の NFS Client ノードのトラフィック

第 8 章

結論

8.1 まとめ

本稿では，疎結合マルチクラスタ向き分散ストレージシステムとして Elton の開発を行った．7 章より，Elton はストレージ基盤として実用に値する性能であることがわかった．これにより，Elton を疎結合マルチクラスタ環境のストレージ基盤とすることで，最小限のコストでサービスの運用が可能な構成を取ることができる．さらに Elton の持つデータ管理機構・キャッシュ機構により，高性能かつ容易なマルチメディア配信機構の提供も期待できる．今後は本学クラウドサービスセンターのストレージ基盤としてさらなる活用が期待される．

8.2 クラウドサービスセンターでの利用

クラウドサービスセンターにおける直近の導入予定として，マルチメディア配信基盤としての利用が検討されている．オープンソースの動画配信プラットフォームである Kaltura Community Edition[43] の基盤を Elton とすることでキャッシュを有効利用した配信が可能になる．本学では Moodle がすでに導入されており，Moodle と Kaltura を組み合わせた配信 [44] の利用が検討されている．今後はさらなる利用が進められることが期待される．

8.3 今後の課題

8.3.1 Eltonfs の改良

現状開発した Eltonfs は FUSE ベースでの実装となっている．今後さらに I/O の性能等が求められた際には FUSE ベースをやめ，ファイルシステムとしてきちんと実装することが求められる．

また，現状の実装では動作させるアプリケーションによって相性の問題が発生することがわかっている．現在わかっているケースは Eltonfs をマウントしたディレクトリ上で WebDAV を動作させようとするとファイルの書き込みがうまくいかないことがある．相性問題を発生させないよう実装を改良していく必要がある．

8.3.2　ストレージ管理機構の拡充

現在の Elton ではストレージ管理機構はバックアップ，バックアップからの自動リストアのみである．大規模な運用を行う際には，どこでどのようなファイルが管理されているか，リソースはどの程度か，各コンポーネントの死活監視等さまざまな管理機構が必要となる．こういった管理機構の拡充は不可欠であると考えられる．

謝辞

本研究を行うにあたり，日々多くの助言，御指導を頂きました田胡和哉教授に深く御礼申し上げます．

多くの時間を共に過ごし，様々の知識とアドバイスをいただいた田胡・柴田研究室の先輩方・Google先生，そして多くの笑いを提供してくれた仲間たちに感謝いたします．

参考文献

[1] Amazon Web Services,Inc : “Amazon Wb Services” http://aws.amazon.com (2016/01/07).

[2] Google : “Google Cloud Platform” https://cloud.google.com/ (2016/01/07).

[3] Docker,Inc : “Docker - Build, Ship, and Run Any App, Anywhere” https://www.docker.com/ (2016/01/05).

[4] Kubernetes : “Kubernetes by Google” http://kubernetes.io/ (2016/01/07).

[5] Cloud Native Computing Foundation : “Home — Cloud Native Computing Foundation” https://cncf.io/ (2016/01/07).

[6] 東京工科大学 : “プレスリリース：東京工科大学がキャンパスシステムをフルクラウド化” http://www.teu.ac.jp/press/2014.html?id=97 (2016/01/07).

[7] 東京工科大学 : “プレスリリース：最新のクラウド技術を用いた学内システムを学生が構築” http://www.teu.ac.jp/press/2015.html?id=19 (2016/01/07).

[8] Arch For Startup : “連載企画: 第 1 回「クラウドコンピューティングとは」” http://www.archforstartup.com/blog/2015/04/21/the-future-of-cloud1/ (2016/01/05).

[9] Wikipedia : “クラウドコンピューティング” https://ja.wikipedia.org/wiki/クラウドコンピューティング (2016/01/09).

[10] Google : “Google Apps for Work” https://apps.google.com/intx/ja/ (2016/01/12).

[11] Google : “Google App Engine” https://cloud.google.com/appengine/docs (2016/01/12).

[12] Amazon Web Services,Inc : “Amazon EC2” https://aws.amazon.com/jp/ec2/ (2016/01/12).

[13] 東京工科大学 田胡・柴田研究室 大矢 涼介 : “座席コードを用いた出席管理システムの開発” (東京工科大学卒業論文 2014 年度).

[14] Moodle : “Moodle” https://moodle.org/ (2016/01/12).

[15] LinuxContainers.org : “Linux Containers” https://linuxcontainers.org/ja/lxc/introduction/ (2016/01/14).

[16] CoreOS,Inc : “rkt” https://coreos.com/rkt/docs/latest/ (2016/01/14).

[17] CoreOS,Inc : “App Container and the Open Container Project” https://coreos.com/blog/app-container-and-the-open-container-project/ (2016/01/14).

[18] LWN.net : “Namespaces in operation, part 1: namespaces overview” http://lwn.net/Articles/531114/ (2016/01/15).

[19] Paul Menage : “Karnel.org Documentation CGROUPS” https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt (2016/01/15).

[20] enakai00 : “Docker のネットワーク管理と netns の関係” http://enakai00.hatenablog.com/

entry/20140424/1398321672 (2016/01/15).
[21] Docker,Inc : “Docker Engine” https://www.docker.com/docker-engine (2016/01/16).
[22] Docker,Inc : “Docker Registry” https://docs.docker.com/registry/ (2016/01/13).
[23] Docker,Inc : “Docker Compose” https://www.docker.com/docker-compose (2016/01/16).
[24] Docker,Inc : “Docker Swarm” https://www.docker.com/docker-swarm (2016/01/16).
[25] Docker,Inc : “Docker Machine” https://www.docker.com/docker-machine (2016/01/16).
[26] CoreOS,Inc : “CoreOS” https://coreos.com/using-coreos/ (2016/01/16).
[27] Google : “Google Container Service” https://cloud.google.com/container-engine/ (2016/01/16).
[28] Amazon Web Services,Inc : “Amazon EC2 Container Service” https://aws.amazon.com/jp/ecs/ (2016/01/16).
[29] 田中 遼，田胡 和哉 : “コンテナ型仮想化機構を用いた大学向けハイブリッドクラウドの構築” (情報処理学会 第 77 回全国大会講演論文集，2015 年).
[30] Wikipedia : “Infiniband” https://ja.wikipedia.org/wiki/InfiniBand (2016/01/12).
[31] Microsoft : “Microsoft Azure” https://azure.microsoft.com/ (2016/01/16).
[32] NFS : “ArchWiki” https://wiki.archlinux.org/index.php/NFS (2016/01/18).
[33] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung : “The Google File System” (SOSP 2003).
[34] Docker,Inc : “Volume plugins” https://docs.docker.com/engine/extend/plugins_volume/ (2016/01/18).
[35] grpc.io : “GRPC” https://inside.t-lab.cs.teu.ac.jp/inside/4624/201601151104.html (2016/01/18).
[36] IETF : “Hypertext Transfer Protocol Version 2 (HTTP/2)” https://tools.ietf.org/html/rfc7540 (2016/01/18).
[37] Google Developers : “Protocol Buffers” https://developers.google.com/protocol-buffers/ (2016/01/18).
[38] Github : “bolt” https://github.com/boltdb/bolt (2016/01/18).
[39] IBM : “Develop your own filesystem with FUSE” http://www.ibm.com/developerworks/library/l-fuse/ (2016/01/18).
[40] Github : “go-fuse” https://github.com/hanwen/go-fuse/tree/master/fuse (2016/01/18).
[41] Github : “go-plugins-helpers” https://github.com/docker/go-plugins-helpers (2016/01/18).
[42] Wikipedia : “Bonnie++” https://en.wikipedia.org/wiki/Bonnie++ (2016/01/21).
[43] Kaltura : “Kaltura CE” http://corp.kaltura.com/Deployment-Options/Kaltura-Community-Edition (2016/01/21).
[44] 篭谷 隆弘 : “マルチメディアコラボレーション基盤プロトタイプの構築” (仁愛大学研究紀要 人間生活学部篇 第 2 号 2010).

業績

[1] 水野 拓, 古谷 文弥, 田胡 和哉 : “分散データリポジトリ Raptor の実装と評価” (情報処理学会 第 77 回全国大会講演論文集, 2015).

[2] 古谷 文弥, 水野 拓, 田胡 和哉 : “知的財産作成を支援するクラウドアプリケーション IWB の開発と評価” (情報処理学会 第 77 回全国大会講演論文集, 2015).