Vol.2016-OS-138 No.6
2016/8/8

Instant Cloud FS：広域分散環境で即興に構築できる分散ファイルシステム

利光 宏平[1,a)]　田浦 健次朗[1,b)]

概要：近年、ビッグデータ処理の需要が高まっている。しかし、処理したいデータが初めから目的のリソース上にあるとは限らない。そこで、分散ファイルシステムを導入して、リモートのデータにネットワーク越しですぐにアクセスできるようにすることを考える。しかし、分散ファイルシステムは、一般的に固定されたノードの集合 （LAN 内、Grid 環境など）で構成されることを前提としている。例えば、NAT の影響でリモートからアクセスできない手元の PC を含めることや、ファイアーウォールの影響がある 2 つのリージョンにまたがるクラウド群で構成することは非常に困難である。また、分散ファイルシステムでは、導入に手間がかかるという欠点もある。そこで、これらの問題を解決するために、Instant Cloud FS (ICFS) という分散ファイルシステムを開発した。ICFS では、手元の PC を構築ノード群に加える、複数のリージョンにまたがってノード群を構成するという、従来の分散ファイルシステムでは難しかった環境で分散ファイルシステムを構築できる。また、構成ノード群は、非特権ユーザでも ICFS の設定ファイルを書き換えるだけですぐに変更ができ、かつ各ノードで必要とされる前作業も少なくなっている。実際に、ICFS は、AWS のクラウド 20 台 (東京 10 台、ソウル 10 台)、東京大学のクラウド 10 台、手元の MacBook Pro という、広域かつ NAT やファイアーウォールの影響がある環境で構築できるということが確認できている。

キーワード：分散ファイルシステム

Instant Cloud FS : A Distributed File System for Instant Deployment across Multiple Environments

Kohei Toshimitsu[1,a)]　Kenjiro Taura[1,b)]

Abstract:
For fast data access to remote data, it is desirable to have a distributed file system that facilitates data sharing across multiple, separately administered environments such as a user's desktop PC and remote machines, public clouds and private clusters, clouds spanning across multiple regions, and so on. To this end, we developed Instant Cloud FS (ICFS). ICFS can be deployed with getting over NAT or firewall. ICFS achieved high user-ability, and has a result in being deployed among a notebook PC and 30 nodes in Tokyo and Seoul.

Keywords: distributed file system

1. 背景

近年, 大規模なデータをコンピュータで処理する機会が多くなってきている. このような大規模なデータは, 性能がそれほど高くないユーザのデスクトップ PC1 台だけで処理することは不可能で, スーパーコンピューターを含むクラスタや, Amazon Web Service(AWS)?などのクラウドを大量に動かして, 大規模に並列処理することが不可欠である.

[1] 東京大学大学院情報理工学系研究科
University of Tokyo
[a)] toshimitsu@eidos.ic.i.u-tokyo.ac.jp
[b)] tau@eidos.ic.i.u-tokyo.ac.jp

ところで, そういうビッグデータが初めから処理するためのクラスタやクラウド上にあるかというと, 必ずしもそうではない. 大規模処理ができるクラスタやクラウドの中で作られることもあるが, 地理的に離れた別のクラウドやクラスタの中にある場合もあれば, ユーザのデスクトップ PC や外付けのハードディスクの中にあるかもしれない. そのような状態において, 目的のクラスタやクラウド等でデータ処理を行うためには, 当然データのコピーや移動が必要である. しかし, データをネットワークを通じてコピー・移動させる間に, データのサイズによっては数日かかる可能性があり, 実際にデータ処理を始めるまでに大幅なタイムラグが生じてしまう.

そこで, データをコピーするのに並行して, クラウド・クラスタから, 別のクラウド・クラスタやデスクトップ PC のファイルを扱うことができるように, 分散ファイルシステム**??**を導入するという考えがある. 分散ファイルシステムとは, ネットワークを通じて別のマシンのファイルにアクセスすることを可能にするファイルシステムのことである. クライアント側から見れば, ネットワークを経由したリモートのファイルがローカルディスク上のファイルと全く同じように見えるというものである.

しかし, 現状の分散ファイルシステムは, 固定されたノードの集合で構築されるのが基本である. そのノードの集合を変更するなどといったことは現状難しくなっている. その基本的な問題は LAN をまたがった通信である. 例えば, クラウド・クラスタとデスクトップ PC との間での通信において, デスクトップ PC からのクラウド・クラスタへの通信は SSH（Secure Shell）などで可能なことが知られているが, クラウド・クラスタからデスクトップ PC への通信は NAT の関係もあり, 当たり前にできるかというとそうではない. また, 同様に, 地理的に離れた異なるクラウド間での通信, 異なるクラスタ間での通信, クラウドからクラスタへの通信, クラスタからクラウドへの通信も同じくファイアーウォールの関係で自由にできるとは限らない. したがって, クラウド・クラスタからデスクトップ PC, クラウドからクラスタ, クラスタからクラウド, クラウドから別のクラウド, クラスタから別のクラスタとディレクトリをマウントすることがこれまでの分散ファイルシステムと同じようにできるか不明瞭であるということになる. また, 一般的な分散ファイルシステムは, 各ノードに対して面倒な設定作業を強いるものが多い. これらのことから, 分散ファイルシステムは複数の環境をまたがって構築されることはほとんどなく, 固定されたノードの集合で構築されるのが基本となっている.

これらの問題点を踏まえて, デスクトップ PC, クラウド, クラスタといった様々な環境間でも利用できる分散ファイルシステム Instant Cloud FS (ICFS) の実装を行った. ICFS に含まれる特徴としては, 地理的に離れたコンピュータ間でも分散ファイルシステムを構築できるようになる, 各ノードでのインストールなどによる負担を最小限に抑える, ノード構成をローカルでユーザが簡単に設定できるなどがある.

2. 関連研究

Google File System**?** は, Google が自社用に開発した分散ファイルシステムである. 安価なクラスタコンピュータを大量に用いた場合でも, フォールトトレランス性を持つことができ, 多数のクライアントがいても高いパフォーマンスを発揮できるように設計された点で注目された. Google File System では, チャンクのサイズが 64MB と大きいためメタデータの管理がしやすいようになっている. 一貫性については, 書き込みには対応しておらず,「追記」で保証されるように作られている. アーキテクチャは, クライアント・単一のメタデータサーバ（マスター）・複数のチャンクサーバという形で, クライアントがデータにアクセスするときは, まずマスターにアクセスし, 格納場所を教えてもらってからチャンクサーバにある実際のデータにアクセスするという形になる. マスターは, メタデータの保存に加え, 一貫性を保つためのロックや複製の管理, チャンクサーバの監視などの役割も持つ. また, Hadoop File System**?** は, Hadoop**?**の MapReduce などによる処理に対応させた分散ファイルシステムであり, Google File System の影響を強く受けたものである.

Ceph**??**は, 分散ファイルシステムである前に, まずベースとなるのは RADOS と呼ばれる分散オブジェクトストレージである. オブジェクト ID を hash と CRUSH (Controlled Replication Under Scalable Hasing) と呼ばれるアルゴリズムで扱い, 各オブジェクトがどこのストレージサーバに格納, 複製されるかを決めるという形で分散オブジェクトストレージを構成している. この分散オブジェクトストレージに, オブジェクトとファイルの変換を行ったものが分散ファイルシステム CephFS である. CephFS では, 数台のメタデータサーバをメタデータ管理用に追加する必要がある. メタデータサーバを複数利用することで, 担当するファイルシステムの領域を分担したり, アクセスが集中している領域を複数のメタデータサーバが一緒に担当したりと, 単一のメタデータサーバを利用するときと比較して負荷分散の点で利点がある. メタデータサーバが複数台で構成されている分散ファイルシステムとして有名である.

これらの分散ファイルシステムは, 基本的に広域分散環境で構築されることを想定されていない.

Gfarm**?**は, 筑波大学で開発された分散ファイルシステムである. Google File System と同様で, 単一のメタデータサーバをもつアーキテクチャの分散ファイルシステムであるが, Google File System はデータにアクセスする場合必ずメタデータサーバにアクセスする必要がある. このため,

実際のデータにアクセスするまで多少の時間がかかる．ここで, Gfarm では, クライアントノードとファイルシステムノード（Google File System でいうチャンクサーバ）が同じ機器にあることを許可することで, ローカルにある (同じ機器にある) データにアクセスする際にはレイテンシが少なくなるという利点を生み出した．また, 複数クライアントからの同一ファイルへのアクセスに対しては, 1 クライアントが read/write モードのとき, 他のクライアントが read only モードでアクセスしようとした場合にはアクセスを認め, read/write モードでアクセスしようとした場合にはその時点でアクセス中のクライアントによるプロセスが終了するまで待たせるという形をとって対処している．さらに, read/write モードのプロセスが終了したら, 必ずすべての複製ファイルに変更点を反映させて一貫性を保つ仕組みになっている．

Gfarm は, 広域分散環境も想定しているが, 広域分散環境とは Grid 環境のことで, ネットワークなどの複雑な設定が必要または NAT やファイアーウォールによっては構築に参加できないノードが存在することがある．

GMount?は, SSH と FUSE をつかって, クライアントが従来の分散ファイルシステムと比較して容易に分散ファイルシステムを構築できることに焦点をおいた分散ファイルシステムである．GMount では, メタデータサーバを利用せず, 実データとメタデータを同じノードに格納する方式を取っている．また, 各ストレージノード間のネットワークは, 通信が重くなりがちな全対全の完全グラフたけではなく, ノード間のネットワーク距離を意識した木構造でつなげられるようになっている．これらのことにより, クライアントと目的のデータが同じノードにある (ローカルアクセスできる) 場合は, すぐにアクセスでき, 前述の木構造のために近いノードからデータ捜索ができるため, 実データにたどり着くための時間が非常に短くなる．また, SSHFS では, 通信が 1 対 1 でしかできないため, 多くのノードを利用する際手間がかかるが, 1 対多で接続できる SSHFS-MUX というモジュールをつかって, ユーザが簡単にノード間をネットワークをつなぐげることができるようになっている．本研究とは, SSHFS を利用するという類似点がある．なお, SSHFS と rsync を用いて, 同時にファイル同期システムを導入できる Cynk?というシステムもある．

3. Instant Cloud FS

この章では, Instant Cloud FS(ICFS) を用いると何ができるとかを説明する．

ICFS で様々な環境間で分散ファイルシステムを構築する方法について述べる．まず, 分散ファイルシステムを構築するすべてのノードに必要とされる条件は以下のみである．

- SSHFS をインストールする
- 手元の PC から SSH で (何段かかってもいいので) アクセスできる．

その後, ホームディレクトリ上に ”icfs” というディレクトリをつくり, 共有したいファイルを ”icfs/shr” というディレクトリにおけば, ICFS によって全ノードがアクセスできるようになる．

その後, ローカルで”Config”というファイルに, 各ノードの SSH 接続の関係を記述する．図 3.1 は, その一例である．

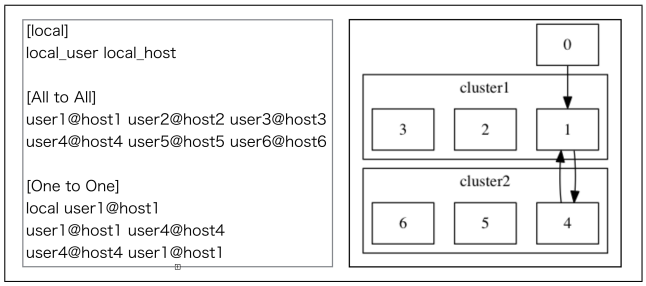

図 1 Config 一例

この Config の書き方について説明する．まず, 最初の [local] には, 手元の PC のユーザ名, ホスト名を記述する．

次の [All to All] については, 同じ行に書かれたノードが, それぞれ SSH が自由にできることを意味する．この例において, user1@host1・user2@host2・user3@host3 の 3 つのノードは自由に SSH ができる．user4@host4・user5@host5・user6@host6 の 3 つも同様である．

最後の [One to One] は, [All to All] の外側での通信を意味する．行の一番左のノードがクライアントノード, そこから右にはサーバノードを記述する．サーバノードはいくつあってもよい．この例だと, ローカルから user1@host1・user1@host1 から user4@host4・user4@host4 から user1@host1 という通信が [All to All] の通信以外に可能だということになる．

Config に求められる条件は, 全てのノードへローカルから SSH で辿りつけること, 全てのノードがどれか 1 つの [All to All] グループに属し, かつ重複してグループに属さないこと, ローカルから直接 SSH でアクセスできるマシンは 1 つであることである．グループに属するノード数は 1 ノードでも構わない．

Config を書いた後は, ICFS のプログラムを実行する．Config を読み取り, マウントの仕方を決定し, その情報を記したテキストファイルを生成する．その後, 各ノードにそのテキストファイルを分散し, 全ノード一斉にそのファイルに従ってマウントのコマンドを実行させる．

マウント後は, 以下の図 3.2 ようなディレクトリがすべてのノードのディレクトリ”ICFS”上に現れ, 全ノードのディレクトリ”shr”内を扱うことができるようになる．

```
test/sim/
├── 0_shr -> /home/ubuntu/test/mnt/1_mnt/0_shr
├── 1_shr -> /home/ubuntu/test/mnt/1_shr/
├── 2_shr -> /home/ubuntu/test/mnt/1_itr/2_shr/
├── 3_shr -> /home/ubuntu/test/mnt/1_itr/2_itr/3_shr/
├── 4_shr -> /home/ubuntu/test/mnt/4_shr/
├── 5_shr -> /home/ubuntu/test/itr/5_shr/
└── 6_shr -> /home/ubuntu/test/itr/6_shr/
```

図 2 構築後ディレクトリ構造例

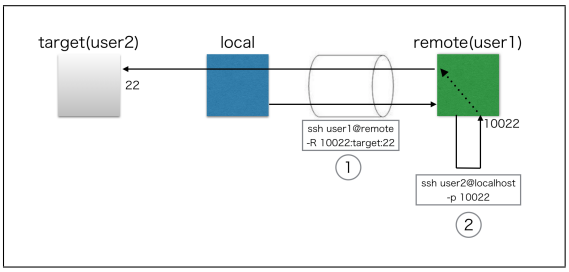

図 3 リモートポートフォワード

以上が ICFS の機能である. まとめると,
(1) 各ノードでいくつか作業を行う
(2) ローカルで Config を正確に書く
(3) ローカルでプログラムを動かす

たったこれだけの作業で, 固定されたノードの集合内だけでなく, 他のクラウド・クラスタ, さらにはデスクトップ PC といった様々な環境で分散ファイルシステムをすぐに構築することができる. 構築したい環境が変化しても, 各ノードでの作業が少なく, あとは Config を書き直すだけでいいので, 柔軟かつ迅速な対応が可能である.

4. 実装

4.1 要素技術

まず, 本研究を実装する際に用いた要素技術について述べる.

4.1.1 SSH

今回の研究で, 直接 SSH?で接続できないサーバに接続するために用いる技術がポートフォワード（トンネリング）である. ポートフォワードは, あるコンピュータの特定のポート番号に送られてきた通信データを, 別のコンピュータの特定のポート番号に転送する技術のことである. ローカル側のポートをリモートに転送する「ローカルポートフォワード」と, リモート側のポートをリモートに転送する「リモートポートフォワード」がある. 今回は利用するのはリモートポートフォワードである.

リモートフォワードでは, リモートホストの特定のポートに送られてきた通信データを, ローカルのコンピュータを通じて, 別のコンピュータに転送することができる. ローカルがリモートに SSH ログインする際に, リモートフォワードを示すオプション「-R」と, リモートの特定ポート・転送先のコンピュータ（ターゲット）・ターゲットのポートを指定しておく. この状態で, リモートの特定ポートに送られたデータは, ローカルを通じてターゲットの特定ポートに送られる. 即ち, ローカルがリモートフォワード指定した状態でリモートに SSH ログインしているときに, リモートの特定ポートに SSH ログインすると, ログイン先がリモートではなく, ターゲットにログインができるということになる. 図 4.1 では, その様子を表している.

本研究では, 単純に SSH で接続できないサーバに接続できるようにするためにリモートポートフォワードを用いている. なお, ポートフォワードを用いる際は, ”-N”や”-f”というオプションと一緒に実行することが多い. ”-N”はリモートでコマンドを実行しないということを指定するオプション, ”-f”はバックグラウンドで実行することを示すオプションで, この 2 つを併用することで, 特にログインやリモートでのコマンド実行を行うことなくポートフォワードのみを実現するプロセスの生成が可能となる.

4.1.2 SSHFS

SSHFS(SSH Filesystem)[1] は, SSH でアクセスができるリモートマシンのシステムを, ローカルマシンのディレクトリにマウントするプログラムである. SSHFS は, SSH のファイル転送プロトコルである sftp と FUSE を組み合わせて実装されている. FUSE[2] とは, カーネルコードを修正せずとも, ユーザ空間でファイルシステムにアクセスする処理を行わせることで独自のファイルシステムを実装することができるモジュールである. SSH を用いているため, 鍵認証の仕組みなどは SSH と同じである.

例えば, リモートマシンのディレクトリ ”/target” をローカルのディレクトリ ”/mnt” にマウントする場合は, 以下のコマンドを入力することになる.

```
$ sshfs remote_user@remote:/target /mnt/
```

続いて, 本研究で実行する, 複雑な SSHFS のマウントを 2 つ説明する.

1 つは, リモートポートフォワードを用いたものである. これは, ノード B からノード A に SSH で接続ができず, 逆にノード A からノード B に SSH で接続ができる場合で, ノード B がノード A を SSHFS でマウントするときに行うものである. まず, ノード A がノード B 側にリモートポートフォワードでトンネルをつくる. 例として, ノード B 側のポート 10022 に送られたパケットをノード A 側のポート 22(SSH 用のポート) に送るようなトンネルをつくる. その後, ノード B はノード A 側が指定したポートで SSHFS マウントを行う. この場合, ユーザ名はノード A のもの, ホストは localhost を指定して, ポート 10022 で SSHFS をノード B が行うことになる. これで, 直接 SSH 接続ができない方向でも SSHFS でマウントができるということになる.

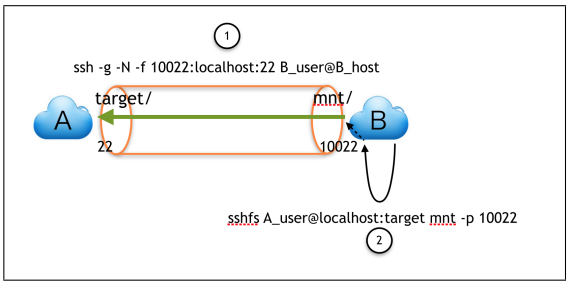

図 4 リモートポートフォワードを用いた SSHFS

図 4.2 はこれを示す.

もう 1 つは, 多段 SSH を用いた SSHFS である. たとえば, SSH 接続がノード A からノード B, ノード B からノード C と可能なときに, ノード A がノード C のディレクトリをマウントすることを考える. まず, ノード B は, ノード C のディレクトリ”target/”を自分自身の”middle/”というディレクトリにマウントする. その後, ノード A がノード B のディレクトリ middle/を自分自身のディレクトリ”mnt/”にマウントすれば, ノード B を通じてノード C のディレクトリ”target/”を結果的にマウントできる. 図 4.3 はこれを示す.

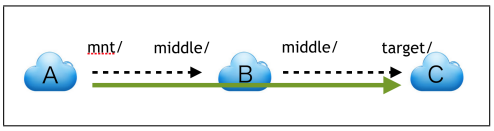

図 5 2 段マウント

4.1.3 GXP

GXP[3][4] は, 分散環境におけるシステム管理から並列処理といったところまでを, 極力小コストかつ多様な環境で行うことができる分散シェルである.

GXP は python で記述されている. GXP のソースはすべてのノードに置く必要はなく, 基準となる 1 つのノードに置いておけば, ノード接続の際に自動分散される.

主なコマンドは, ノード間の SSH 接続状況を入力する”use” コマンド, use コマンドで入力した情報に基づき各ノードと接続する”explore”コマンド, そして全ノードで同じコマンドを実行する ”e” コマンドである.

ICFS では, 生成したコマンドを全ノードで一斉に動かすためにこの GXP を用いている.

4.2 コマンドの生成

ここから先は, これらの技術を用いて行った実装について具体的に述べる. なお, 実装中の例には, 第 3 章でも使用した, 以下の環境を利用している

ICFS のプログラムが行うことは、主に

- Config を読み、マウント数をなるべく少なくできるようなデータ構造を作る
- 作ったデータ構造から、マウントのコマンドを決定する
- 各ノードで決定したコマンドを実行する

の 3 つである.

4.2.1 データ構造

ICFS では、1 つのノードが他の全ノードのディレクトリを SSHFS でマウントする数はノード数に比例する。よって、これを全ノードが行うと、マウント数はノード数の 2 乗に比例する。ノード数が増えるとマウント数がかなり多くなるため、マウント数を減らす工夫が ICFS には含まれている。その中の 1 つとして、ノードを木構造にし、各ノードが親ノードと子ノードのディレクトリのみをマウントするという方式で、マウント数をノード数の 2 乗からノード数そのものに比例するようにしたものがある。

まず、Config を読み込み、ノードで図 のような木構造を作る. 木構造を作るプログラムは、子ノードの数がなるべく 2 を超えないようなアルゴリズムになっている.

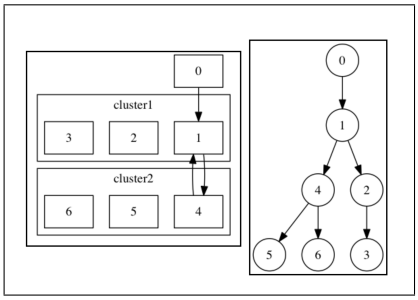

図 6 Config と木構造

4.2.2 コマンドの決定

木構造をつくった後は, マウント先とマウントするディレクトリを決める. まず, 各ノードが用意する 2 つのディレクトリとその役割について述べる.(ディレクトリ $HOME/ICFS/ 以下に置かれる)

- ”mnt” - 親ノードと自分自身のディレクトリをマウントする (ノード 0 にはない)
- ”itr” - 子ノードのディレクトリをマウントする.

ディレクトリ”mnt”には, 親ノードのディレクトリ”mnt”,”itr”,”shr” すべてをマウントする. これによって, 親ノードのさらに親ノード側のファイル, 親ノードの, 自分以外の子ノード側のファイル, 親ノードのファイルすべてをみることができる.

ディレクトリ”itr”には, 子ノードのディレクトリ”shr”と”itr”をマウントする. これによって, 子ノードのファイル, 子ノードのさらに子ノード側のファイルを見ることができる.

なお, ノード 0 には”itr”は作られず, 子ノードを持たないノードには”itr”は作られない.

こうして, すべてのノードで同じ作業を行うことで, 結果

的に多段 SSHFS の形でそれぞれのノードが他のノードすべてのディレクトリを見ることができるようになる.

ディレクトリが決まったら, グループ内完全グラフのときと同じように, マウントのコマンドを生成する. 今回は, 直接 SSH できるかリモートポートフォワードが必要かのどちらかを選ぶだけである.

例として、ノード 4 でどのようになるか説明する.

ノード 4 の親ノードはノード 1, 子ノードはノード 5 と 6 である. ノード 5 と 6 は子ノードを持たないので, ノード 4 はノード 5, 6 の”shr”のみをマウントする. ノード 1 について, ノード 4 はノード 1 の”mnt”, ”itr”, ”shr” をマウントする. この時点で, ノード 4 はノード 1, 4, 5 の”shr”を直接マウントできていることになる.

ここで, ノード 1 の”mnt”は, 親ノードであるノード 0 の”shr”をポートフォワードを利用した上でマウントしていることになっているので, ノード 4 はノード 0 の”shr”を 2 段 SSHFS でマウントできていることになる. 同様に, ノード 1 の”itr”は, 子ノードであるノード 2 の”itr”と”shr”をマウントしていることになっているので, ノード 4 はノード 2 の”shr”を 2 段 SSHFS でマウントできていることになる. 更に, ノード 2 の”itr”は, 子ノードであるノード 3 の”shr”をマウントしていることになっているので, ノード 4 はノード 3 の”shr”を 3 段 SSHFS でマウントできていることになる.

以上から, ノード 4 は結果的に全ノードのディレクトリ”shr”をマウントできているということになる.

最後に、このままでは各ノードで見えかたが異なるので、マウント法則に従ってシンボリックリンクを行うことで、第 3 章で説明したようなディレクトリ構造に見えるようにする.

今回は木構造のみを説明したが, SSHFS の段数が多くならないように, マウント数は多くなってしまうが, [All to All] の中では完全グラフ形式でマウントするという方法も実装している.

4.2.3 コマンドの実行

各ノードで決めたコマンドを実行するために, 実行するノードの番号を頭に加えた上でテキストファイルに全てコマンドを書き込む。その後、GXP で全ノードにそのテキストファイルを分散し、そのノード番号のコマンドを各ノードそれぞれ実行させるという形をとっている.

5. 実験・評価

5.1 実験環境

計測における環境は, 以下のとおりである.

- MacBookPro13 - ローカルマシンとして使用
- AWS EC2 リージョン 東京 - 10 台
- AWS EC2 リージョン ソウル - 10 台
- 東京大学情報理工学研究科 IST クラウド - 10 台

SSH の接続環境を図 5.1、ping による Roud trip time(RTT)・iperf によるバンド幅性能を図 5.2 に示す.

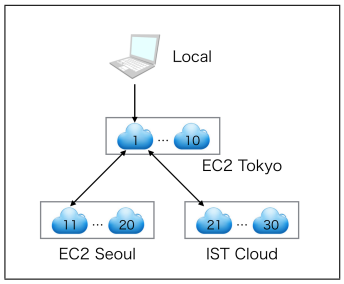

図 7 接続環境

Client - Server	RTT[msec]
Local - EC2 Tokyo(1)	10.9
EC2 Tokyo(1) - EC2 Seoul(11)	32.8
EC2 Tokyo(1) - IST Cloud(21)	9.53
EC2 Tokyo(1) - EC2 Tokyo(2)	0.79

Client - Server	Throughput [MB/sec]
EC2 Tokyo(1) - Local	43.5
EC2 Tokyo(1) - EC2 Seoul(11)	15.2
EC2 Tokyo(1) - IST Cloud(21)	62.8
EC2 Tokyo(1) - EC2 Tokyo(2)	118.1

図 8 RTT および iperf 性能

以上のマシンで, グラフ構造あるいは木構造を用いて分散ファイルシステムを構築した. 各リージョン (東京, ソウル, IST) の中のノード数は, 全リージョン等しく 3 台から 10 台 (全体で 9,12,15,...30 台) の中で変化させた. ベンチマークは, write/read のスループットの測定に IOR を, メタデータ性能の測定に mdtest を使用した.

5.2 マウント時間

マウントにかかった時間を, ノード数を変化させながら計測した.(図 5.3) ノードが増えてもマウント時間は 10 秒以内に抑えることができた. なお, GXP の explore の時間は含めていない. これには, 平均して 15 秒ほどかかるが, これを含めても実用的な速度であると考えられる.

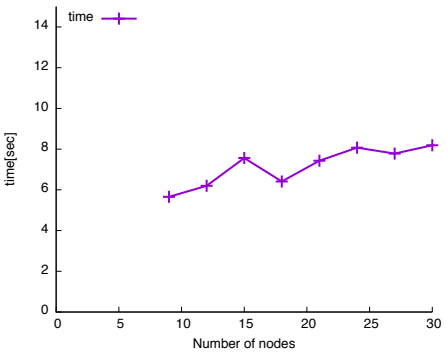

図 9 マウント時間

5.3 並列 read/write, メタデータ性能

最後に, 全クライアント一斉に write/read や, メタデータ操作を行った際の性能を計測した. 今回は, (自分のマシン番号) + i) / (全クライアント数) のノードに対してすべてのノードが write/read や directory create を行い, すべてのクライアントを合計性能を出した. 変数 i の値には, ほとんどのノードが自分のリージョン内で操作を行うことになる 1 と, すべてのノードが別のリージョン内のノードに操作を行うことになる (全ノード数) / 2 の 2 つを採用した. この実験で出てほしい結果とは以下の 2 つである.

(1) i = 1 のとき, クライアント数の増加によってスケールすること
(2) i = (全ノード数 / 2) のとき, ボトルネックになる通信箇所の性能と比較して同じくらいの性能がでる.

以下, その結果である.(図 5.14, 図 5.15)

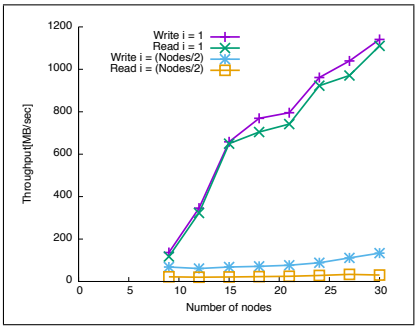

図 10 並列 IO 性能

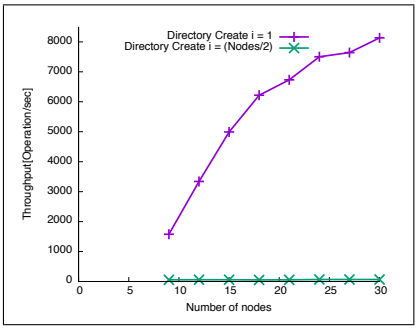

図 11 並列メタデータ操作性能

$i = 1$ のときは, read/write, directory creation のどちらでもスケールすることが確認できた.

$i =$ (全ノード数 / 2) のときを考える. まず, IO 性能について考える. 今回, ボトルネックになるのは EC2 東京 - EC2 ソウル間である. この間で, 全クライアントの 3 分の 2 がデータの転送を行う. 1 クライアントの転送データ量を *data*, クライアント数を *clients*, EC2 東京・EC2 ソウル間の iperf 性能を *bandwidth* とおいてとき, その時間は, read が write の 2 倍時間がかかることを考慮して, read/write 含めて,

$$\frac{(data) \times (clients \times \frac{2}{3})}{(bandwidth)} \times (1+2)$$

となる. つまり, 全データ量も EC2 東京 - EC2 ソウル間の転送時間もクライアント数に比例して増えるため, $i =$ (全ノード数 / 2) のときの IO 性能はあまりスケールしないということになり, 性能の向上自体は見られないが, 理にかなった結果になる.

つぎに, メタデータ性能について考える. 同様に, ボトルネックは EC2 東京 - EC2 ソウル間である. 1 クライアントがつくるディレクトリ数を *directories*, クライアント数を *clients*, EC2 東京・EC2 ソウル間の RTT を *rtt* とおいたとき, この間でかかる時間は,

$$directories \times (clients \times \frac{2}{3}) \times (rtt \times 4)$$

となる. つまり, IO 性能同様, 全オペレーション数も EC2 東京 - EC2 ソウル間の時間もクライアント数に比例して増えるため, $i =$ (全ノード数 / 2) のときのメタデータ性能も理にかなった結果になる.

6. おわりに

本研究では, 普通は固定されたノードで構成される分散ファイルシステムを, デスクトップ PC, クラスタ, クラウドなど複数の環境で構築できる Instant Cloud FS の実装とその評価を行った. 実際にクラウド 30 台で問題なく分散ファイルシステムが構築できた. この ICFS は, 各クライアントに求める条件も少なく, 木構造の場合ではマウント時間も十分少なく, ローカルで Config を書いてプログラムとシェルスクリプトを動かすだけで構築ができ、かつその構成の変更も柔軟に行うことができるという点でユーザに負担の少ないものができたと言える.

しかし, 並列 IO 性能や並列メタデータ性能を改善するために, ファイルのレプリケーションとその配置方法, 同期のスケジューリングなどを実装する必要がある. また, 障害発見および復帰などのフォールトトレランス機能も必要である. 今後は, ユーザにとって利用しやすいということと, 十分な性能を兼ね備えた分散ファイルシステムを目指す.

参考文献

[1] Amazon Web Service. `https://aws.amazon.com/jp/`.
[2] Benjamin Depardon, Gaël Le Mahec, and Cyril Séguin. Analysis of Six Distributed File Systems. Research report, February 2013.
[3] TIS 株式会社. 分散ファイルシステムの wan 越え同期レプリケーションの検証. 2014.
[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *SIGOPS Oper. Syst. Rev.*, Vol. 37, No. 5, pp. 29–43, October 2003.
[5] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*,

MSST '10, pp. 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[6] Apache Hadopt. `https://hadoop.apache.org/`.

[7] 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*, OSDI '06, pp. 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[8] Sage A. Weil. *CEPH: RELIABLE, SCALABLE, AND HIGH-PERFORMANCE DISTRIBUTED STORAGE*. PhD thesis, UNIVERSITY OF CALIFORNIA SANTA CRUZ, 2007.

[9] Osamu Tatebe, Kohei Hirage, and Noriyuki Soda. Gfarm Grid File System. *New Generation Computing*, Vol. 28, No. 3, pp. 257–275, 2010.

[10] Nan Dun, Kenjiro Taura, and Akinori Yonezawa. GMount: An Ad Hoc and Locality-Aware Distributed File System by Using SSH and FUSE. *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, pp. 188–195, May 2009.

[11] Dun Nan, Angkasa Sugianto, Taura Kenjiro, and Chen Ting. Cynk: A Hybrid Rsync and SSH Filesystem for Cloud Computing. *HPC*, Vol. 2011, No. 39, pp. 1–7, jul 2011.

[12] OpenSSH. `http://www.openssh.com/`.

[13] SSH Filesystem. `http://fuse.sourceforge.net/sshfs.html`.

[14] Filesystem in Userspace. `http://fuse.sourceforge.net/`.

[15] GXP. `http://www.logos.ic.i.u-tokyo.ac.jp/gxp/`.

[16] Kenjiro Taura. GXP: An Interactive Shell for the Grid Environment. In *Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems*, IWIA '04, pp. 59–67, Washington, DC, USA, 2004. IEEE Computer Society.