Vol.2009-HPC-121 No.32
2009/8/6

広域ファイルシステムのための分散メタデータサーバの設計

平賀弘平[†1]　建部修見[†1]

広域環境下の複数の組織間で，大容量データを効率良く共有する要求が年々増加している．広域環境では，高遅延通信路間の通信オーバヘッドが大きいため，一貫性制御を行う必要があるメタデータ操作がシステムのボトルネックになるという問題がある．本稿では，広域ファイルシステムのための分散メタデータサーバを設計する．Multiple Master 型のメタデータサーバで，一貫性に結果整合性を採用し，高遅延通信路の影響を受けないメタデータ操作の手法を提案する．提案システムは，ファイルシステムのメタデータを inode 構造体を用いて管理し，複数の inode エントリをアトミックに更新する場合でも，更新の衝突検知と解決操作を行えるプロトコルを備える．

Design of Distributed Metadata Management System for Global File System

Kohei Hiraga[†1] and Osamu Tatebe[†1]

Efficient sharing of large-scale data has been required among multiple organizations in wide area. Global file system is useful to easily share data, but it has a problem of long response time when accessing from a distant location. This paper presents a design of distributed metadata server for global file system. It consists of multiple masters, and ensures eventual consistency. It is not affected by a long latency network. It manages file system metadata using inode structures, having capability to detect and fix collisions of simultaneous metadata updates that affect multiple inodes.

†1 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba

1. はじめに

近年コンピュータの高性能化や価格低下により，様々な組織で PC クラスタを構築し，それらを利用して大規模な計算処理を行う事が容易となっている．それに伴い，計算処理で扱うデータ量は年々増加している．素粒子物理学，天文学，生命科学などの科学技術分野におけるデータインテンシブコンピューティング分野では，広域に散在する計算資源，記憶装置，観測装置を効率的に扱い，組織間の壁を超えて有益なデータの共有を行うグリッド関連技術の発展と標準化が進んでいる．

グリッドコンピューティングの例として，Avian Flu Grid プロジェクト[1] では，鳥インフルエンザ等の感染メカニズム，薬物耐性の解明のために，世界規模に分散した組織間で大量のデータ共有と分析を分担している．

また，天文学の分野の例では，地理的に離れた場所にある観測機器が生成する膨大なデータを，如何に円滑にデータ解析できるかが重要である．これらの大量の天文データを，広域通信路を含む環境で，手軽に，効率的に共有したいという要求がある．

このように，地理的に分散した PC クラスタ間で，データを効率的に共有する必要性が高まっている．これらの例に挙げた広域分散計算機環境では，大容量かつ広域で組織間にまたがるデータ共有を，広域ファイルシステム Gfarm[2] を利用してデータの共有を行っている．

ファイルシステムのメタデータへの操作は，ファイルシステム全体のワークロードの多くを占めている[3]．分散ファイルシステムでは，メタデータ操作がシステムのボトルネックにならないようにするための提案がこれまでなされているが，広域環境では Round Trip Time の長い高遅延通信路が存在するため，そのままでは適用できない．例えば，日米間の RTT は約 130ms 以上，日欧間は約 300ms 以上なので，広域ファイルシステム操作に対するリクエスト応答時間は，高遅延ネットワークを介した通信が何往復するかによって大部分が決定する．そのため，メタデータのシステム全体の厳密な一貫性を保証するための，広域でロックを取得するペナルティが非常に大きいという問題があるからである．

我々はこれまでに，RTT の長い通信路に依存しない，Key-Value の連想配列をベースにメタデータを管理する手法について研究を行ってきた[4]．本稿では，これをベースにデータ構造を一部拡張し，実際のファイルシステムのメタデータを管理するシステム，分散メタデータ管理サーバを設計する．2 章で分散メタデータサーバの設計を述べ，3 章で関連研究を述べる．4 章でまとめと今後の課題について述べる．

Vol.2009-HPC-121 No.32
2009/8/6

2. 分散メタデータサーバの設計

2.1 概　　要

広域ファイルシステムのリクエスト応答時間の短縮のために，ファイルシステムのメタデータの管理手法を検討し，メタデータへの Read/Write 操作の応答時間を短縮する手法を提案する．我々はこれまでに，RTT の長い通信路に依存しない，Key-Value の連想配列をベースにメタデータを管理する手法について研究を行ってきた．メタデータサーバをグリッド環境の各拠点に分散配置し，クライアントとメタデータサーバ間は低遅延な通信路を使いレイテンシを抑える．サーバ間の通信は，バックグラウンドで非同期に行い，サーバ間のデータの一貫性は，結果整合性 (Eventual Consistency)[5] を保証した．本稿では，これをベースにデータ構造を一部拡張し，実際のファイルシステムのメタデータを管理する分散メタデータ管理サーバを設計する．

ファイルシステムのファイルやディレクトリに関するメタデータは，inode と呼ばれる構造で管理する．複数の inode エントリをアトミックに更新する操作のサポートを行った．サーバ間の inode の一貫性は結果整合性を保証するプロトコルを採用する．inode への更新が起こると，他のサーバにバックグラウンドで非同期に更新が伝播し，自動的に同期する．クライアントは，拠点内のメタデータサーバに対して inode 操作を行う．メタデータサーバは，自身の持つ inode への操作が完了したら，他拠点のメタデータサーバと通信を行うことなく，直ちにクライアントに応答を返す．

2.2 システム構成

提案システムの構成を**図 1** に示す．SiteA，SiteB，SiteC はそれぞれ PC クラスタを保有している地理的に分散した異なる拠点を表す．各拠点にメタデータサーバを配置し，各メタデータサーバはシステムが管理する全メタデータの複製を保持する．

クライアントはメタデータ操作の為の API 呼出しを通じて，自拠点内にあるメタデータサーバに対して Read/Write リクエストを発行する．クライアントのリクエストは低遅延な LAN を介して処理される．各メタデータサーバはすべてのメタデータの複製を保持しているので，メタデータサーバはクライアントに即座に応答を返すことができる．メタデータサーバは Multiple Master 型の構成を採用し，複数クライアントからの Read/Write 操作の並列性を確保する．

2.3 サーバ間のメタデーター貫性

前節で示したような Multiple Master 型のシステム構成の場合，サーバ間の一貫性を維持するためには，分散ロックや 2-phase commit 等を利用した複雑でオーバヘッドの大きいプロトコルの設計を行う必要がある．広域環境においては，他拠点との通信遅延が長いため，一貫性を維持するためのプロトコルは致命的な応答時間の増大に繋がってしまう．

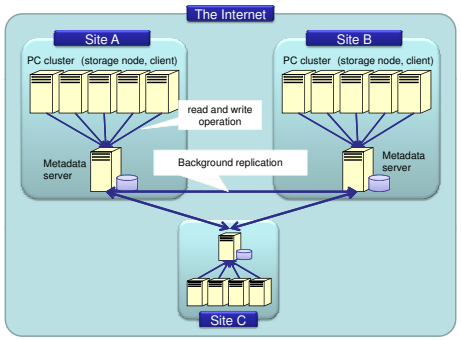

図 1　提案システムの構成．SiteA,SiteB,SiteC は PC クラスタを保有している地理的に分散した異なる拠点を表す．各拠点にはメタデータサーバが分散して配置され，クライアントからのリクエストを受け付ける．

提案システムでは，一貫性モデルとして結果整合性を採用する．結果整合性は制約の緩い一貫性で，常に最新のデータにアクセスできるとは限らないが，最終的には最新のデータにアクセスできるようになる事を保証する．結果整合性を採用した理由は 2 つある．一つ目は，前述の通り，高遅延通信路を含む環境下で厳密な一貫性を実現すると，書き込みが集中したり，2-phase commit を実装することが避けられないからである．前者は書き込みのスケールアウトが期待できなくなる．後者は，広域通信路で行ったときのオーバヘッドが大きすぎるという問題がある．二つ目は，CAP 定理[6] より，Consistency，Availability，Partition のうち 2 つまでしか達成することができないという問題がある．広域ファイルシステムにおいては，ファイルシステムの consistency は必ずしも要求されないことが多い．従って本システムでは，厳密な一貫性を保つことで，ファイルシステム全体の性能が低下するより，むしろパフォーマンスを出すことを優先する．もし厳密な一貫性がどうしても必要な場合は，ロックサービスの利用等で，高通信コスト，低可用性なロック操作を行うことを許容する．これらの理由から，結果整合性を選択する．

2.4 基本データ構造

メタデータサーバは，ファイルやディレクトリ等のファイルシステム上のオブジェクトに関する情報を管理する．本システムでは，これらのメタデータを inode と呼ばれるデータ

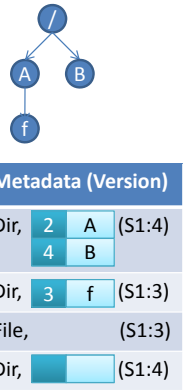

図 2 基本データ構造．各 inode エントリには，エントリのバージョンが格納される．ディレクトリに関する inode エントリには，そのディレクトリに含まれるオブジェクトの inode 番号と名前のペアが含まれる．

構造で管理する．inode は Unix 系ファイルシステムで使われているデータ構造で，各オブジェクトに関する情報が格納される．inode エントリは，inode 番号と呼ばれるユニークな識別子で参照される．ディレクトリに関する inode エントリには，そのディレクトリに含まれるオブジェクトの inode 番号と名前のペアが含まれる．クライアントは，inode エントリをルートディレクトリから再帰的に辿ることで，目的のオブジェクトに到達する．

分散配置された各メタデータサーバは，この inode 番号とオブジェクトメタデータのペアからなる Key-Value の連想配列をそれぞれローカルに保持し，管理する．inode に対する Write/Read 操作リクエストは，すべてのメタデータサーバで並行に発行される．各々のサーバの inode エントリに対する更新は，他のサーバに非同期に伝播され，自動的に同期する．他のサーバに更新が伝播するよりも早く，他拠点のクライアントによって inode エントリに対する更新が起こった場合，inode エントリへの更新が衝突する場合がある．したがって，お互いが持っている Key-Value の連想配列を Eventually に同期し，一貫性を維持するプロトコルが必要となる．更新が衝突した inode エントリが含まれる場合は，何らかのマージルールで衝突を解決し，最終的にはすべてのメタデータサーバが同じ inode テーブルを持つ状態に収束しなければならない．また，たとえ衝突する更新が発生しても，ディレクトリツリーの木構造が崩れて，ルート inode エントリから辿り着けない inode エントリが現れてはならないという問題もある．

2.5 ユニークな inode 番号の生成

ファイルシステムが新しい inode 番号の生成を必要とした時は，ユニークな番号を生成する必要がある．分散システムにおいて，あるコンポーネントがシステム全体でユニークとなるような番号を生成するには，他のコンポーネントが自分と同じ番号を生成しない事を保証すれば良い．そこで，分散システムの各コンポーネントには，起動時にあらかじめシステム全体でユニークとなる識別子（例えば FQDN 等）を割り当てる．新しい inode 番号を生成するには，自身の識別子と，コンポーネントが各々管理する非負の整数の対によって，システム全体でユニークとなるよう番号を生成する．

2.6 Vector Clock を用いた inode のバージョン管理

結果整合性を採用している提案システムでは，同時に 2 つ以上のクライアントから inode の更新操作が行われた場合，更新が衝突する可能性がある．Multiple Master 型で厳密な一貫性のための排他制御を実現するには，Write/Read 操作時に，広域に分散するメタデータサーバ間で分散ロックを取得する必要があるが，これを行わない．提案システムでは，分散ロックの代わりに Vector Clock[7] を用いる．Vector Clock は，それぞれのサーバ識別子とサーバローカルなカウンタのペアのベクトルで，半順序が定義される．メタデータサーバの inode の各エントリは，Vector Clock によってバージョニングされる．バージョニングによって，inode の更新操作の順序関係を保存し，順序が定義されなければ，更新操作により衝突を検知する．もし更新操作が衝突したら，一時的にメタデータサーバ間で inode に違いが起こるが，予め定義するマージアルゴリズムに従って，システムが自動的に衝突を解決し，いずれすべてのメタデータサーバ通知される．これにより，結果整合性を守る．

2.6.1 複数 Key のアトミックな更新操作

ファイルシステムへの操作によって，同時に複数の inode エントリが変更される例を以下に示す．

ファイルの作成

（1） ユニークな inode 番号を生成し，ファイルを表す新しい inode エントリを作成
（2） 親ディレクトリの inode エントリに，新しく作成した inode エントリへのリンクを追加

ファイルの移動

（1） 移動先ディレクトリ inode エントリに，移動ファイルへのリンクを追加
（2） 移動元ディレクトリ inode エントリから，移動ファイルへのリンクを削除

ファイルの削除

（1） 親ディレクトリの inode エントリから，削除対象ファイル inode エントリへのリンクを削除
（2） 削除対象ファイルの inode エントリに，削除フラグをセットする

上記の複数の inode の変更において，どちらかの変更が衝突してしまうと，ディレクトリの木構造が崩れてしまう．ディレクトリ木構造を崩さないよう inode を更新するには，inode の複数のエントリをアトミックに更新する必要がある．そこで，既存の分散 Key-Value ストアの実装[4),8)] の，Vector Clock による単一 Key に対するオブジェクトバージョニングをベースとして，複数 Key にまたがるアトミックな更新時に，正しく更新の衝突の検知を行えるように拡張する．

基本的な方針は，アトミックな複数の変更のうち，どれか一つでも衝突していれば，全体が衝突したと検知する．inode エントリを更新する場合，どの inode 番号の，どのバージョンを更新するのかを指定する．複数の inode エントリをアトミックに更新する場合は，更新する inode エントリの inode 番号 (key) とエントリ内容 (value) のペアのリストと，その更新範囲の inode エントリの現在のバージョンをすべて反映した Vector Clock を指定する．メタデータサーバは更新リクエストを受け取ると，更新される全ての inode エントリの Vector Clock と，更新リクエストの Vector Clock を比較し，更新の衝突が起こっていないか確かめる．比較不能な Vector Clock が一組でも存在するならば，更新が衝突しているとみなし，更新範囲の inode エントリを全てブランチにする．衝突の解決は，衝突の起こった二つ以上のブランチのアトミックな更新範囲を全て読み込み，その範囲のバージョンを全て反映した Vector Clock を指定して，範囲全体を上書きするアトミック更新を行うことで解決する．

以降，具体例を挙げて，更新を行い衝突が発生した場合としなかった場合のシステムの詳細を説明する．

2.6.2 通常の更新処理

inode の更新操作の具体例を**図 3** に示す．例では，広域分散計算機環境の二つの拠点に，それぞれサーバ S1 とサーバ S2 を分散配置している．更新操作を行う直前まで，二つのサーバは同じ inode を保持している．また，ファイルやディレクトリのメタデータには，ユーザ ID，グループ ID，サイズ，タイムスタンプ，パーミッション等の情報が含まれるが，簡略化のため，エントリの区分 (ファイル，ディレクトリ，削除済み) と，ディレクトリの場合は下位 inode エントリへのリンク情報について考える．

（1） S1 が属する拠点内のクライアントが S1 に接続し，mkdir /A を発行する．

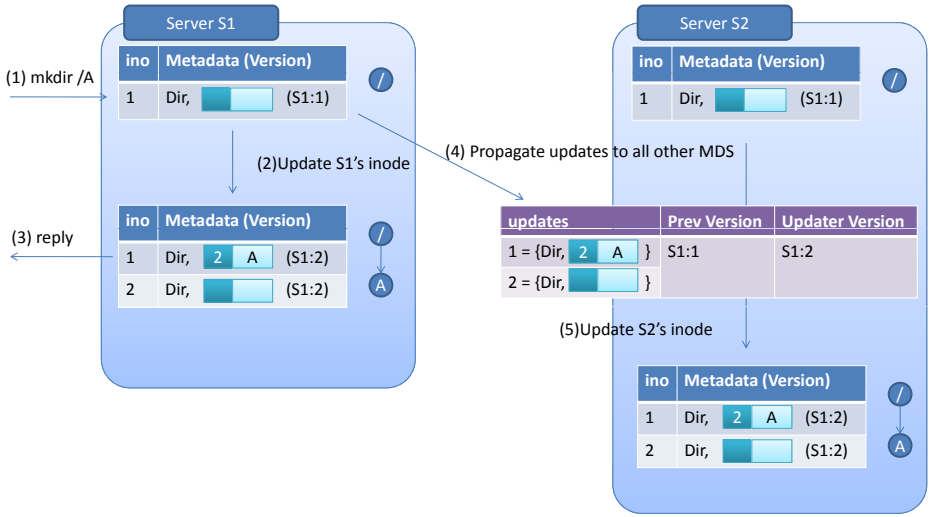

図 3 通常更新処理．クライアントが S1 に mkdir を行った場合の，S1,S2 間の更新伝播の例．

（2） S1 は，自身の管理する inode からこれから更新する inode エントリを読み込み，現在のバージョン (S1:1) を取得する．更新後のバージョンは，現在のバージョンに，S1 の最新クロックから作成したバージョン (S1:2) を加えたバージョン ((S1:1) + (S1:2) = (S1:2)) とし，inode 番号 1 と 2 をアトミックに更新する．
（3） S1 は，ローカルの inode を更新した後，即座にクライアントに応答を返す．
（4） S1 は，更新が起きた全ての inode エントリと，以前のバージョン (S1:1) と，更新時に加えた S1 のバージョン (S1:2) を，他のメタデータサーバに非同期で伝播する．
（5） S2 は，S1 から更新を受け取ると，自身のローカルの inode に反映する．更新前のバージョンと更新者のバージョンを組み合わせて更新後のバージョン (S1:2) を作成し，自身のローカルにある inode の該当エントリのバージョン番号 (S1:1) と比較する．inode 番号 1 では，ローカルバージョン (S1:1) < 更新後バージョン (S1:2)，inode 番号 2 では，ローカルバージョン () < 更新後バージョン (S1:2) で，どちらも更新後のバージョンが後のバージョンと判断できるので，S2 は，この更新をローカル inode にコミットする．

Vol.2009-HPC-121 No.32
2009/8/6

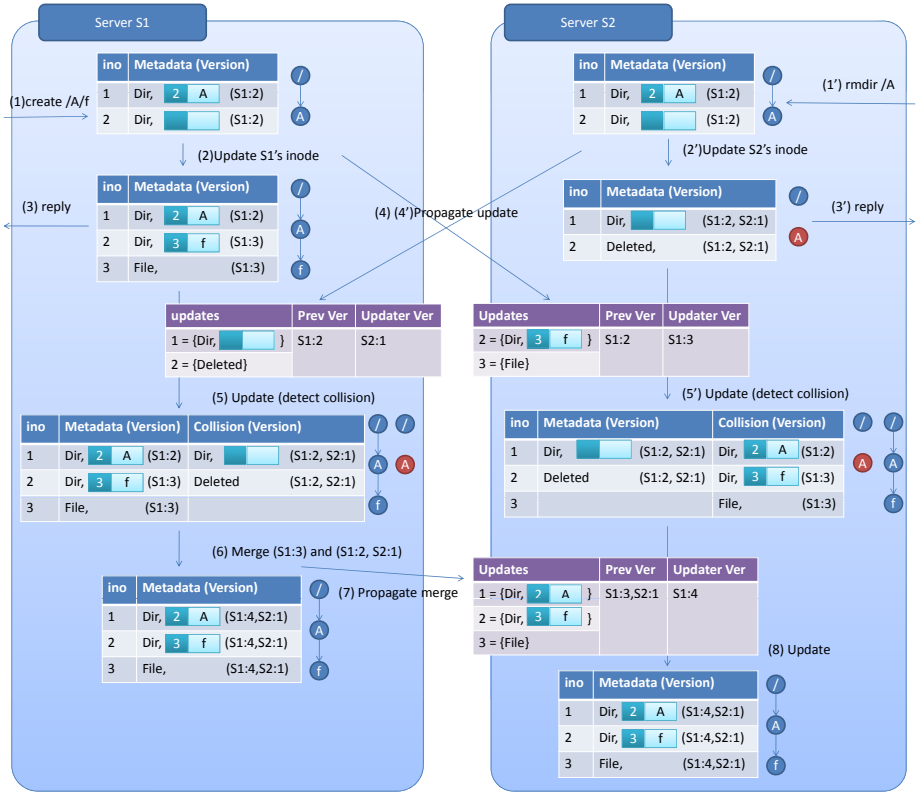

図 4 更新の衝突と衝突の解決．クライアントが S1,S2 に同時に更新処理を行った場合の，衝突の検知とマージ処理の例．

2.6.3 衝突する更新処理と衝突の解決

同時に複数のサーバがクライアントから更新操作を受け付けた場合の例は**図 4** のようになる．サーバ S1 とサーバ S2 の初期状態は，前節で挙げた例の最終状態からの続きとする．

(1) S1 は，クライアントからファイル/A/f を作成する操作を発行される．同時に，S2 は別のクライアントから，ディレクトリ/A を削除する操作を発行される．

(2) S1 と S2 は，それぞれ自身のローカルの inode を，発行された変更操作に従って更新する．S1 は，新たに inode 番号 3 を作成し，ディレクトリ/A のエントリに，inode 番号 3 へのリンクを作成する．inode 番号 2 と 3 のバージョンは (S1:2)+(S1:3)=(S1:3) となり，アトミックに更新する．S2 では，ディレクトリ/A の inode エントリに削除フラグを立て，ディレクトリ/のエントリから/A へのリンクを削除する．inode 番号 2 と 3 のバージョンは (S1:2)+(S2:1)=(S1:2,S2:1) となり，これらをアトミックに更新する．

(3) S1 と S2 は，ローカルの inode を更新し終えると，クライアントに応答を返す．

(4) S1 と S2 は，更新された inode エントリ番号とその内容，更新前のエントリのバージョン全てを反映したバージョン (S1:2) と，各サーバの最新クロック,S1 では (S1:3), S2 では (S2:1) をそれぞれ自分以外のメタデータサーバに非同期で伝播する．

(5) S1 は，S2 から送られてきた更新内容を，自分のローカル inode へ反映を試みる．更新前のバージョンと更新者のバージョンを組み合わせて更新後のバージョン (S1:2,S2:1) を作成し，これから変更を行うローカル inode エントリ 1 と 2 のバージョンと比較する．inode 番号 1 では，ローカルバージョン (S1:2) ＜ 更新後バージョン (S1:2,S2:) になるが，inode 番号 2 ではローカルバージョン (S1:3) と更新後バージョン (S1:2,S2:1) は，比較不能状態となる．この時，inode 番号 1 のエントリは，バージョンの比較では更新後の (S1:2,S2:1) の方が後と判断されるが，(S1:2,S2:1) の更新全体をアトミックにコミットしなければならないため，アトミックな更新単位で全体が衝突が起きた事とみなし，両方のバージョンの全てのエントリを保管する．S2 側も，同様の手順で更新の衝突を検知し，衝突した全てのエントリを保管する．

(6) S1 は，inode エントリの衝突を検知すると，衝突の解決をする．衝突の解決は，衝突している二つ以上のバージョンに関係する全ての inode エントリを，新しいデータで上書きして行われる．ここでは，(S1:3) と (S1:2,S2:1) の衝突の解決を行うため，inode 番号 1,2,3 の全てのバージョンのデータとバージョンを読み込み，更新前バージョン (S1:3)+(S1:2,S2:1)=(S1:3,S2:1) に，S1 の最新クロックのバージョン (S1:4) を加えた (S1:4,S2:1) で，新しいデータを inode 番号 1,2,3 にアトミックに上書きする．衝突解決の具体的な方法は，次節で述べる．ここでは，S1 の操作を残すよう衝突を解決する．

(7) 衝突の解決によって行われた複数の inode エントリへのアトミックな更新を，自分以外のメタデータサーバに非同期で伝播する．

(8) S2 は，S1 から送られてきた更新を自身のローカル inode に反映させる．この時，更新後のバージョン (S1:4,S2:1) は，S2 の inode 番号 1,2,3 の全てのバージョンとの比

較により子孫と見なせるので，S2 のローカル inode にある複数のバージョンを統合する．

2.6.4 衝突解決サーバの選択と，inode の衝突解決アルゴリズム

更新の衝突が起こった場合，システムによる自動的な inode の衝突解決を行うが，更新の衝突を検知した際，各々のサーバが衝突解決を行うと，衝突解決のための更新がまた衝突してしまう問題がある．そのため，衝突解決を自動で行う担当サーバ (衝突解決サーバ) を一台選択しなければならない．衝突解決者の選出には，システム管理者がマニュアルオペレーションで設定する方法や，paxos アルゴリズム[9] の利用が考えられる．

inode の衝突解決アルゴリズムは従来研究で提案されている手法を想定している．ファイルの更新が衝突した場合は，ファイルを別の名前にリネームし，同じディレクトリに保存することで解決する．前節の例のように，更新の衝突により親ディレクトリが失われてしまった場合には，親ディレクトリを復活させて解決する方法と，親ディレクトリを失ったエントリを (/orphan 等の) 特別なディレクトリに移動する方法が考えられる．衝突した複数の更新操作は，衝突解決サーバに到着した順番で直列化され，順番にマージ処理を行う．解決方法に複数選択肢がある場合は，ユーザが予め衝突解決ポリシーを設定する．

3. 関 連 研 究

AFS[10] では，クライアントが AFS のファイルサーバから切断されると使用不可能になるが，ネットワークが切断して，ファイルサーバから切断された場合でも，クライアントのローカルキャッシュを利用して，限定的にファイルシステムを利用可能にする拡張についての研究[11] がある．AFS の後継の Coda ファイルシステムでも同様に，非接続動作の拡張がされている[12),13)]．どちらの研究も，クライアントのローカルストレージにあるメタデータを参照し，切断時は更新をローカルキャッシュに対して行い，後でファイルシステムとの回線が繋がった時に，それまでローカルで行った変更を，サーバに同期させるというアプローチを取っている．本研究では，更新は常に非同期で伝播されるが，関連研究では，更新が非同期に行われるのはクライアントがファイルシステムから切断されている時に限る．そのため，通常はキャッシュ一貫性を維持するために，ファイルサーバと通信を行うので，遅延が発生する．また，更新の衝突が発生した場合，ユーザが手動で解決しなければならないが，本研究では，メタデータサーバが自動で解決する方針を採用している．

Dynamo[8] は high available な Key-Value のハッシュストレージシステムである．Vector Clock によるオブジェクトバージョニングを行っている．Key のハッシュ値を元にストレージノードを決定し，データを分散管理する．広域で運用する場合は拠点外にあるストレージノードが選択される可能性もあり，その場合リクエストの応答時間が長くなると考えられる．また，複数の Key をアトミックに更新する仕組みは提案されていない．提案システムでは，複数 Key のアトミックな更新操作をサポートしている．

Chord[14] 等の Distributed Hash Table を用いると，Key-Value のデータを多数のノードに分散させることができる．データの所在はハッシュ関数によって決定するため，広域環境では，基本的にデータアクセスは高遅延ネットワークを経由して行う事になり，リクエスト応答時間の増加につながる．

Google File System[15] では，メタデータアクセスがファイルシステムのボトルネックになることを防ぐため，ファイルのブロックサイズを大きくし，管理しなければならないメタデータの総量を小さくすることで，メタデータを全てメモリーに乗せ，高速なメタデータ操作を行う工夫がなされている．また，メタデータの情報を一部クライアントにキャッシュして，メタデータサーバに掛かる負荷の軽減を行っている．しかし，メタデータサーバはシングルサーバ構成であるため，広域環境ではクライアントは高遅延通信路を経由してメタデータサーバにアクセスすることになるため，遅延の影響は避けられない．

4. お わ り に

本稿では，広域ファイルシステムのための分散メタデータサーバの設計について述べた．Multiple Master 型の分散システムで，一貫性は結果整合性を選択し，高遅延通信路の影響を受けないファイルシステムの inode 更新処理方法を提案した．複数の inode エントリをアトミックに更新する場合でも，更新の衝突検知と解決操作を行うプロトコルの設計を行った．

今後の課題としては，システムを実装し，広域分散環境においてリクエスト処理性能を評価する事である．

謝辞 本研究の一部は，文部科学省次世代 IT 基盤構築のための研究開発「e-サイエンス実現のためのシステム統合・連携ソフトウェアの研究開発」における研究課題「研究コミュニティ形成のための資源連携技術に関する研究」（データ共有技術に関する研究）および文部科学省科学研究費補助金特定領域研究「情報爆発時代に向けた新しい IT 基盤技術の研究」（公募研究 A02-17，課題番号 21013005）による。

参 考 文 献

1) : Avian Flu Grid. http://avianflugrid.pragma-grid.net/.
2) 建部修見，曽田哲之：広域分散ファイルシステム Gfarm v2 の実装と評価，情報処理学会研究報告 2007-HPC-113，pp.7–12 (2007).
3) Roselli, D., Lorch, J.R. and Anderson, T.E.: A Comparison of File System Workloads, *In Proceedings of the 2000 USENIX Annual Technical Conference*, pp.41–54 (2000).
4) 平賀弘平，建部修見：広域ファイルシステムにおける分散メタデータサーバの検討，情報処理学会研究報告 2009-HPC-119，pp.139–144 (2009).
5) Vogels, W.: Werner Vogels "Eventually Consistent" (2007). http://www.allthingsdistributed.com/2007/12/eventually_consistent.html.
6) Gilbert, S. and Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, *SIGACT News*, Vol.33, No.2, pp.51–59 (2002).
7) Lamport, L.: Time, clocks, and the ordering of events in a distributed system, *Commun. ACM*, Vol.21, No.7, pp.558–565 (1978).
8) DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W.: Dynamo: amazon's highly available key-value store, *SIGOPS Oper. Syst. Rev.*, Vol. 41, No. 6, pp. 205–220 (2007).
9) Lamport, L.: Paxos Made Simple, *ACM SIGACT News*, Vol.32, No.4, pp.18–25 (2001).
10) Howard, J.H.: An Overview of the Andrew File System, *USENIX Winter Technical Conference* (1988).
11) Huston, L.B. and Honeyman, P.: Disconnected operation for AFS, *MLCS: Mobile & Location-Independent Computing Symposium on Mobile & Location-Independent Computing Symposium*, Berkeley, CA, USA, USENIX Association, pp.1–1 (1993).
12) Satyanarayanan, M.: Coda: A Highly available File System for a Distributed Workstation Environment, *IEEE Transactions on Computers*, Vol.39, pp.447–459 (1990).
13) Kistler, J.J. and Satyanarayanan, M.: Disconnected operation in the Coda File System, *ACM Trans. Comput. Syst.*, Vol.10, No.1, pp.3–25 (1992).
14) Stoica, I., Morris, R., Karger, D., Kaashoek, M.F. and Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications, *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA, ACM, pp.149–160 (2001).
15) Ghemawat, S., Gobioff, H. and Leung, S.-T.: The Google File System, *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp.20–43 (2003).