

Gfarm ファイルシステムの分散メタデータサーバ設計

建部 修見^{1,a)}

概要：並列プロセスや多数プロセスからの同時ファイルアクセスに対しスケールアウトするための方法として、これまで複数のメタデータサーバでハッシュ分散させる方法が取られてきた。ハッシュ分散はメタデータサーバの負荷分散には効果的であるが、局所性がないためクライアントから全メタデータサーバへのネットワークトラフィックが均等に発生し、ネットワークに対する性能要求が高くなる。本研究では、Gfarm ファイルシステムにおいて、局所性を考慮することが可能なメタデータの分散方式の提案を行い、分散メタデータサーバの設計を行う。これにより、特にメタデータサーバ数が増加したときに問題となるネットワークトラフィックの軽減を目指す。

1. はじめに

コア数の増大による並列プロセス数の増加、同時実行プロセス数の増加により、ファイルシステムに対するメタデータ処理性能の向上が求められている。特に、近年のサーバコンピュータはコア数が数十万を越え、メタデータ処理性能としては秒間百万オーダーを越える処理性能が求められる。一方、メタデータ処理性能を向上させるために、ファイルデータの管理と、メタデータの管理を別サーバで行うファイルシステムの研究開発が進んでいる。Lustre ファイルシステム [2]、Gfarm ファイルシステム [9]、Ceph ファイルシステム [10] などである。ただし、秒間百万オーダーを越えるメタデータ処理性能を達成するためには、単一メタデータサーバノードでは難しい。そのため、メタデータ処理性能をスケールアウトさせるため分散メタデータサーバの研究開発が行われている。研究レベルでは、IndexFS [6] のように、128 メタデータサーバを用い、秒間百万操作を達成しているものもあるが、運用システムではまだ達成されていない。IndexFS などは、メタデータをハッシュで分散させている。ハッシュ分散は、負荷分散には効果的であるが、局所性がないため、メタデータサーバ数を増やしたときネットワークトラフィックが多くなる。このネットワークトラフィックの増加は性能向上の阻害要因となる。本研究では、局所性を考慮することが可能なメタデータの分散方式の提案を行い、ネットワークトラフィックの軽減を目指す。さらに、Gfarm ファイルシステムにおいて、分散メタデータサーバの設計を行う。

2. 分散メタデータサーバ

ファイルシステムのメタデータとしては、ファイルシステム全体のメタデータ、階層的なディレクトリ構造、各ファイルやディレクトリに関するメタデータ、クォータに関するメタデータがある。また、Gfarm ファイルシステムのように複数のユーザ管理ドメインにおいてファイルアクセスを可能とするためには、ユーザ情報などもメタデータとして管理される。

これらのメタデータにおいて、ファイルやディレクトリに関するメタデータなど、メタデータの多くは配列やハッシュ表で管理できるため分割が容易である一方、階層的なディレクトリ構造は分割が容易ではない。階層的なディレクトリ構造の分散管理については、ディレクトリ構造の管理方法により変わってくる。ディレクトリ構造の管理については大きく以下の三方式がある。

- (1) ディレクトリエントリによる管理
 - (2) フルパス名による管理
 - (3) 親ディレクトリ i ノード番号 + エントリ名による管理
- (1) は UNIX ファイルシステム [4] など広く用いられる方法である。ファイルやディレクトリに関するメタデータはインデックスノード (i ノード) で管理される。 i ノードは配列で管理され、 i ノード番号でアクセス可能である。ディレクトリは、ファイル名 (エントリ名) とその i ノード番号からなるディレクトリエントリで管理される。階層的なディレクトリ構造において、それぞれのエントリはルートディレクトリからのディレクトリ名を / (スラッシュ) で区切るパス名 (フルパス名) で表される。ディレクトリエントリによる管理では、各ディレクトリのディレクトリエ

¹ 筑波大学計算科学研究センター

^{a)} tatebe@cs.tsukuba.ac.jp

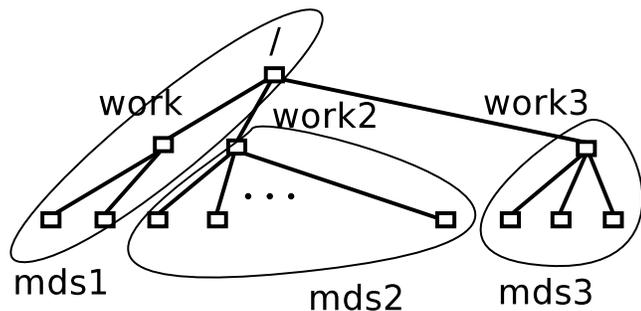


図 1 部分ディレクトリ分割方式。

ントリを検索し、次のディレクトリの i ノード番号を知る。この処理を繰り返して、目的の i ノードをアクセスする。この一連の処理をルックアップと呼ぶ。ルックアップにおいては、ディレクトリ階層の深さだけの検索処理が必要となる。ディレクトリエントリは、エン트리数が多くなったときの検索、追加、削除のコストを減らすため、木構造が用いられる [8]。

(2) はフルパス名をキーとして木構造などで管理する方式である。ルックアップ操作は、一度の検索で行うことができる。ディレクトリ・リスティングは親ディレクトリ名までの前方一致検索で行うため、前方一致検索ができるデータ構造である必要がある。この方式は、ディレクトリの移動についてはディレクトリ以下のエント리를全て移動する必要がありコストが大きい。BetFS [3] では、ディレクトリの移動コストを減らすため、ゾーニングを用いて、部分ディレクトリ単位で管理を行うなどの工夫を行っている。

(3) は特に分散メタデータサーバにおいて用いられる方式であり、親ディレクトリの i ノード番号とエン트리名をキーとして、木構造などで管理する方式である。この方式は (1) と (2) の中間のような方式であり、ルックアップ操作は、(1) と同様に各ディレクトリでのエン트리検索が必要である。そのため、ルックアップのコストはディレクトリ階層の深さに比例する。一方、ディレクトリの移動は (1) と同様にエントリの削除と追加のできるため、コストが小さい。PPMDS [12] や IndexFS [6] で用いられる。

階層的なディレクトリ構造を分散管理する方法として次の三方式がある。

- (1) 部分ディレクトリ分割
- (2) ディレクトリ内分割
- (3) ファイルシステム分割

部分ディレクトリ分割は、階層的なディレクトリ構造の名前空間をディレクトリで分割する方法である。その例を図 1 に示す。図 1 では、/`ディレクトリ` と /`work` ディレクトリ以下のエント리는 `mds1` が管理し、/`work2` ディレクトリ以下のエント리는 `mds2` が管理し、/`work3` ディレクトリ以下のエント리는 `mds3` が管理している。

部分ディレクトリ分割は、(1) ディレクトリエントリに

よる管理において、ディレクトリエントリを分散配置することにより行うことができる。同一ディレクトリは単一メタデータサーバで管理するため、分散させることは比較的容易である。しかしながら、単一ディレクトリに負荷が集中する場合は、負荷を分散させることができない。Lustre ファイルシステムでは、Distributed Namespace (DNE) [7] としてサポートされている。DNE では、ディレクトリを作成するときに、作成したいメタデータサーバを指定する。この方式は、ディレクトリにより管理するメタデータサーバが決まるため、メタデータサーバについて局所性を利用することは難しい。

ディレクトリ内分割は、同一ディレクトリを複数のメタデータサーバで分散管理する方法である。単一ディレクトリに負荷が集中した場合でも、複数のサーバで負荷を分散させることができる。その一方で、同一ディレクトリにおけるファイルの移動、ディレクトリの削除など複数メタデータサーバ間でのトランザクションが必要となる。ディレクトリ・リスティングも、複数のメタデータサーバに分散しているデータを集める必要がある。この分割では、負荷分散を均等に行い、エン트리ごとに管理するメタデータサーバを一意にするため、主にハッシュ分割が用いられる。そのため、分散させるサーバ数は一定であることが多い [10] が、ディレクトリのエン트리数に応じて分散させるメタデータサーバ数を変える GIGA+ [5] のようなアプローチもある。ディレクトリ内分割は、(1) ディレクトリエントリによる管理において、ディレクトリエントリをハッシュ分割することにより行うことができる。ハッシュ分割のため、メタデータサーバについて局所性を利用することが難しい。

ファイルシステム分割は、ファイルシステムの階層的なディレクトリ構造全体を階層構造とは関係なく分割する方式である。特定ディレクトリのエン트리数などの影響を受けることなく処理を分散させることができる。その一方で、ファイルの移動、ディレクトリの削除など複数メタデータサーバ間でのトランザクションが必要となる。ディレクトリ・リスティングも、複数のメタデータサーバに分散しているデータを集める必要がある。この分割は、(2) フルパス名による管理、(3) 親ディレクトリ i ノード番号 + エン트리名による管理のように、階層的なディレクトリ構造をキーとバリューのペアで管理するものについて行うことができる。この分割も、負荷分散を均等に行い、エン트리ごとに管理するメタデータサーバを一意にするため、主にハッシュ分割でなされる。ハッシュ分割のため、メタデータサーバについて局所性を利用することが難しい。

3. Gfarm ファイルシステムにおける分散メタデータサーバ設計

この節では、Gfarm ファイルシステムにおいて、並列

プロセスや多数プロセスからの同時ファイルアクセスに対しスケールアウトするための、分散メタデータサーバ設計を行う。特に、より多くのクライアントからの並列アクセスに対応するため、局所性を考慮した設計を行う。ここでいう局所性とは、各クライアントに対し、近いメタデータサーバをアクセスするということである。近さは、ネットワーク的に近いということだけではなく、アクセスを分散させるためにクライアントごとに担当のメタデータサーバを割り当てる場合もある。後者は、特に大規模 PC クラスタやスパコンにおいて、例えば 1,000 ノードごとに担当メタデータサーバを割り当てるなどを想定している。

3.1 部分ディレクトリ分割

Gfarm ファイルシステムには、エクサバイト級の大規模ストレージのための設計 [11] として、Gfarm URL 形式の他 Gfarm ファイルシステムへのシンボリックリンク機能がある。この機能は、エクサバイト級の大規模ストレージにおいて問題となるメタデータのデータサイズの増大を解決するために導入されたものである。メタデータのデータサイズは、ファイル数に比例する。一方、メタデータは高速性のためメモリに保持したい。そのため、複数のメタデータサーバでメタデータを分割する必要がある。この分割にあたり、[11] では、Gfarm URL 形式のシンボリックリンクが検討された。Gfarm URL は、メタデータサーバ、ポート番号、パス名で構成され、任意の Gfarm ファイルシステムのファイルあるいはディレクトリを指し示すことができる。図 1 において、/work2 以下の管理をメタデータサーバ mds2 に移譲する場合は/ディレクトリに以下のようにシンボリックリンクを作成する。

```
% gfln -s gfarm://mds2/ /work2
```

gfln はシンボリックリンクを作成するコマンドであり、このコマンドにより、`gfarm://mds2/`を指し示すシンボリックリンクが/work2 に作成される。この機構は、部分ディレクトリ分割を行うことに相当しており、Gfarm URL 形式のシンボリックリンクを作成することにより、部分ディレクトリ分割を自由に行うことができる。

3.2 シンボリックリンクによるディレクトリ内分割

Gfarm ファイルシステムにおいて、ディレクトリ内分割を行うことは想定されていないが、Gfarm URL 形式のシンボリックリンクを用いることにより、同様の分割を行うことが可能である。図 2 にシンボリックリンクによるディレクトリ内分割を示す。この図では、/work ディレクトリを mds1 と mds2 の 2 サーバにより管理している。/work/{a1,a2,a3} は mds1、/work/{a4,a5} は mds2 が管理し、それぞれを参照するためのシンボリックリンクが作成されている。ここで、ファイルを管理しているメタデータサーバをそのファイルのホームメタデータサーバ

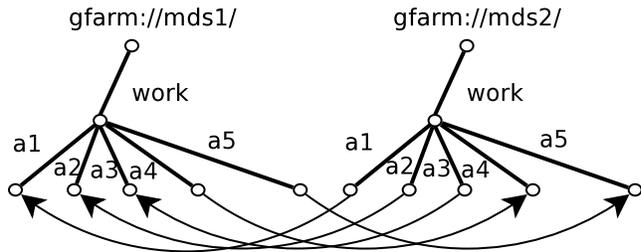


図 2 シンボリックリンクによるディレクトリ内分散方式。

と呼ぶ。また、今後この方式をシンボリックリンク方式と呼ぶ。

シンボリックリンク方式においてディレクトリ内分割を行う場合、ファイル等のエントリを作成するたびに、分散しているメタデータサーバ全てにおいて、ホームメタデータサーバに対しシンボリックリンクを作成する必要がある。つまり、ファイル作成のコストが、メタデータ数を n としたとき $O(n)$ となる。メタデータサーバ数を増やすごとにファイル作成コストが増大するため、メタデータサーバ数を増やすことにより、ファイル作成性能は落ちてしまう。

また、シンボリックリンクを作成中に、同一名のエントリを別のクライアントがまだシンボリックリンクを作成していないメタデータサーバに作成した場合など、一貫性が崩れてしまうことがある。一貫性を崩さないためには、以下の方法が考えられる。

- (1) エントリごとにファイルを作成するホームメタデータサーバを決める。
 - (2) エントリ作成について分散トランザクションを行う。
- (1) は、エントリごとにエントリを作成するホームメタデータサーバが決められるため、クライアントごとのメタデータサーバの局所性とは関係なくエントリを作成することとなる。一方で、エントリを作成するメタデータサーバが決まっているため、エントリ作成についての一貫性の問題は生じない。そのため、シンボリックリンク作成は必ず成功する。また、(2) は、後述するが、エントリ作成ごとに分散トランザクションのコストが必要となってしまふ。

シンボリックリンク方式は、分散メタデータサーバのサーバ数を増やすことにより、性能をスケールアウトすることはできないが、各クライアントが必ず異なるファイルを作成することが保証され、かつそのファイル作成がただちに他のメタデータサーバに反映されなくても構わないような限られた場合においては有用である。このようなケースは多くの広域ファイル共有において当てはまる。例えば、あるサイトにおいてファイルを生成し、そのファイルを皆で共有する場合である。ファイル作成は一箇所であり、特定のメタデータサーバに作成される。作成されたファイルはただちに皆が参照できなくても、そのうち参照できるようになればよいという場合である。

このとき、ファイル作成において、前述のような一貫性

の問題は起こらないため、クライアントごとに近い(担当)メタデータサーバにファイルを作成してもよい。シンボリックリンクの作成により、他メタデータサーバから参照が可能となるが、ファイル作成後直ちに参照できなくてもよいため、ファイル作成時には行わない。これにより、ファイル作成時のコストは、近い(担当)メタデータサーバへのファイル作成だけのコストとなり、ファイル作成性能をメタデータサーバの数に応じてスケールアウトさせることができると考えられる。

シンボリックリンクは、(後でまとめて)非同期に作成する。このようにすることにより、ファイル作成時の応答時間を増やすことなく、ディレクトリ内分散を行うことができる。ただし、シンボリックリンク作成については、非同期になったことにより、応答時間に含まれなくなっただけで、コストは変わらず $O(n)$ である。つまり、メタデータサーバが暇なときに実行できればそのコストを隠蔽することが可能であるが、ずっと高負荷であればそのコストは見てくる。

これまでは、ファイルの生成について考察してきたが、そのほかの操作を考察する。まず、ファイルの更新については、元ファイルは一つしかないため、その元ファイルを更新することにより問題なく行うことができる。ファイルの削除については、完全に削除するためには、元ファイルと全てのシンボリックリンクを消去することとなる。しかしながら、シンボリックリンクの削除と、同名のファイル作成が同じタイミングで行われると一貫性の問題が生じる。ファイル削除のためのシンボリックリンクの削除と、ファイル作成のためのシンボリックリンク作成が同時に行われるためである。そのため、ファイル削除におけるシンボリックリンクの削除はロックする、あるいはトランザクションとする必要がありそうである。この問題は厄介なようにも思われるが、元ファイルだけを削除するようにすると解決することができる。このとき、ファイル削除後は、参照先のないシンボリックリンクが残ることとなるが、その場合は、ファイルが消去されたとして取り扱う。その後、同一ファイルが作成されたときには、同じホームメタデータサーバに作成する。このようにすると、シンボリックリンクは削除されないが、一貫性の問題を解決することができる。

3.3 キャッシュファイルによるディレクトリ内分割

シンボリックリンクによるディレクトリ内分割では、元ファイルは割り当てられたメタデータサーバでしか管理されないため、ファイルの参照のためには、そのメタデータサーバへアクセスすることとなる。多くのクライアントからアクセスが集中すると、そのメタデータサーバにアクセスが集中してしまう。また、遠隔のアクセスの場合、必ず遠隔アクセスが起きてしまい、応答時間が悪くなってまう。

この問題を解決するために、シンボリックリンクではなく、キャッシュファイルを導入する。キャッシュファイルは、シンボリックリンクのように参照先の情報を保持するが、参照先ファイルの内容をキャッシュとして保持する。これにより、何度も参照されるファイルについて、元ファイルを管理するメタデータサーバへの問い合わせを省略することができる。以下、キャッシュファイル方式と呼ぶ。

キャッシュのポリシーについてはさまざま考えられる。キャッシュ時期については、キャッシュファイル作成時、初めてのアクセス時、アクセス回数が閾値を超えたとき、アクセスサイズが閾値を超えたときなどである。キャッシュの方法についても、同期的にキャッシュが終わるまで応答を返さない、非同期にキャッシュを行い、キャッシュが完了する前は元ファイルを直接参照するなどである。

また、キャッシュしたファイルの更新についてもさまざまなポリシーが考えられる。書込モードでオープンする場合は、キャッシュを破棄して参照先ファイルを参照する、Write-through でキャッシュと参照先ファイルの両方を更新する、Write-around で参照先ファイルを更新する、Write-back でキャッシュを更新するなどである。

キャッシュの有効性については、Gfarm ファイルシステムの場合、各ファイルは、更新のたびに世代番号が更新されるため、世代番号により確認することが可能である。

また、キャッシュファイルの容量についても考慮する必要がある。キャッシュファイルはストレージ容量が足りなくなった場合は削除対象となり、LRUなどで管理する必要がある。なお、ファイルをキャッシュすると、それだけストレージサーバの空き容量は少なくなるが、キャッシュファイルは削除可能であるため、ストレージサーバの空き容量としてはキャッシュ容量は含まない。

キャッシュファイル方式は、ファイルの内容をキャッシュ可能ということを除き、実質上はシンボリックリンク方式と変わりはない。シンボリックリンクはキャッシュしないキャッシュファイルと考えることもできる。また、キャッシュファイルの作成については、全メタデータサーバで作成する必要がある、一貫性の問題や作成のコストは同様である。ただし、キャッシュが完了して、有効期限内であれば元ファイルのメタデータサーバへの参照をしなくてアクセス可能となる。

3.4 分散トランザクション

シンボリックリンク方式、キャッシュファイル方式のいずれの場合も、有用なケースではそれぞれのクライアントが重複のないファイルを作成し、その反映を直ちに行わなくても良いケースである。これらのケースでは、メタデータサーバ数に応じて並列プロセスや多数プロセスからの同時ファイルアクセスをスケールアウトさせることができると考えられる。設計上は、それぞれのクライアントは近い

(担当)メタデータサーバのみのアクセスとなり、ネットワーク負荷も低く抑えることができ、メタデータサーバ数を増やしてもネットワークがボトルネックとはならないと考えられる。

一方で、各クライアントが作成するファイルに対し、重複が生じる可能性のある場合を考える。これまでの議論では、(1) ファイルごとに作成するメタデータサーバを特定する、あるいは(2) ファイルごとに分散トランザクションを行う方法を紹介した。本節では、(2) についてアルゴリズムとコストの検討を行う。

分散トランザクションは、障害が起こりうるケースと障害を想定しないケースで複雑さが大きく異なる。Gfarm ファイルシステムは、単一障害点がなく、メタデータサーバの障害、ファイルサーバの障害などが発生したとしてもフェイルオーバーする。そのため、障害が発生しないとして考える。

障害の発生を仮定しない場合、two-phase commit (2PC) [1] が一般的である。2PC は、コーディネータと参加者において分散トランザクションを行う場合、コーディネータは全参加者に VOTE_REQUEST メッセージを送付し、WAIT 状態に遷移する。参加者は、ローカルなコミットの準備ができていれば VOTE_COMMIT を返信して READY 状態に遷移する。そうでなければ、VOTE_ABORT を返信して ABORT 状態に遷移する。コーディネータが、全ての参加者から VOTE_COMMIT を受信すると、COMMIT 状態に遷移し、全参加者に GLOBAL_COMMIT を送信する。そうでない場合は、ABORT 状態に遷移し、GLOBAL_ABORT を全参加者に送信する。参加者は GLOBAL_COMMIT を受信すると COMMIT 状態に遷移し、トランザクションをコミットする。GLOBAL_ABORT を受信すると ABORT 状態に遷移し、アボートする。

2PC では、コーディネータと参加者はコミット、アボートの前に、それぞれ WAIT 状態、READY 状態という状態がある。2PC を Gfarm ファイルシステムの分散メタデータサーバで行うためには、Gfarm ファイルシステムにおいて状態を持たせる必要がある。そこで、READY 状態を示すものとして、READY ファイルを提案する。READY ファイルは READY 状態を示す中間的な状態であるため、ここでの要件は、以下のようになる。

- (1) 排他的に作成する。既に READY ファイルが存在する場合は EEXIST を返す。
- (2) READY ファイルはホームメタデータサーバ情報を持つ。
- (3) stat や open は ENOENT を返し、失敗する。
- (4) unlink は ENOENT を返し、失敗する。

READY ファイルは、そのメタデータサーバにおいてローカルにコミットの準備ができていないことを示すためのものであるため、(1) の要件にあるように排他的に作成し、一意

性を保証する。コーディネータの情報を保持するために、(2) のように READY ファイルにホームメタデータサーバの情報を持たせる。なお、この READY ファイルの作成とホームメタデータサーバの情報の保持はアトミックに行う必要がある。この READY ファイルは、2PC の READY 状態に相当する、ファイル作成のための中間的な状態のものであり、まだファイルは存在していない。そのため、(3) のように、stat や open に対しては、ファイルが存在しないという意味の ENOENT エラーを返し、失敗させる。unlink についても、勝手に READY ファイルが削除されてしまうと READY 状態の意味がなくなってしまうため、(4) のように、ENOENT エラーを返し、失敗させる。ただし、この分散トランザクションがアボートしたときには、コーディネータは READY ファイルを削除することとなるため、そのためのインターフェースは必要である。

この READY ファイルを用い、以下のように分散トランザクションを行う。

- (1) ファイルを作成するクライアントを担当するメタデータサーバはホームメタデータサーバとなり、コーディネータとして、他の全メタデータサーバに対し、READY ファイルを作成する。
- (2) 全メタデータサーバに対し、READY ファイルの作成が成功したら、ホームメタデータサーバにファイルを作成し、READY ファイルをキャッシュファイルに変更する。
- (3) どれかのメタデータサーバに対し、READY ファイルの作成に失敗したら、作成した READY ファイルを削除する。

(1) は、REQUEST_VOTE の送信に相当する操作である。READY ファイルの作成は、ローカルなコミットの準備に相当している。(2) のケースで、全メタデータサーバに対し、READY ファイルの作成に成功したら、全参加者から VOTE_COMMIT が返信されたことに相当する。この時、ホームメタデータサーバにファイルを作成し、コーディネータは COMMIT 状態に遷移する。他の全メタデータサーバに作成した READY ファイルをキャッシュファイルに変更する操作は、GLOBAL_COMMIT の送信に相当する。(3) のケースで、どれかのメタデータサーバに対し、READY ファイルの作成に失敗したら、競合が発生しており、コーディネータは ABORT 状態に遷移する。このとき、作成した READY ファイルを削除するが、この操作は GLOBAL_ABORT の送信に相当する。

このアルゴリズムでは、READY ファイルを他の全メタデータサーバに対し作成している。合意を得るためだけであれば、過半数のメタデータサーバで十分であるが、その場合、ファイルを参照するとき、そのメタデータサーバにキャッシュファイルが作成されていない可能性があり、ファイルが存在するかどうかの確認のために、他のメタ

データサーバを参照する必要が出てきてしまう。

3.5 分散ディレクトリの作成

ディレクトリ内分割を行うためには、図 2 に示されるように、分散させる全メタデータサーバにおいてディレクトリを作成し、それらのディレクトリに対し、分散しているメタデータサーバのリストの指定を行う必要がある。

さらに、それらの操作は一貫性のためにアトミックである必要がある。アトミックでない場合は、同時に異なるクライアントが異なる分散メタデータサーバのリストで分散ディレクトリの作成をしたときに競合してしまい、両方共失敗してしまうだけでなく、その間にそのディレクトリの下にファイルなどが作成される可能性がある。

そのために 3.4 節で述べた分散トランザクションを用いる。ただし、READY ファイルを作成するのではなく、READY ディレクトリの作成となり、またキャッシュファイルに変更するのではなく、分割メタデータサーバリストをもつディレクトリへの変更となる。

ディレクトリ作成の場合は、分散させる全メタデータサーバでディレクトリ作成の合意がとれるだけではなく、全メタデータサーバでディレクトリ作成の完了を待つ必要がある。そうしないと、ファイル(キャッシュファイル、あるいは分散トランザクションにおいては READY ファイル)作成などに失敗するメタデータサーバがでてきてしまう。そこで、分割する全てのメタデータサーバにおいてディレクトリ作成の完了を示すため、ディレクトリに完了情報をもたせる。コーディネータは、分散トランザクションで分散ディレクトリを作成したあと、ディレクトリ作成の完了を示すため完了状態にする。分散ディレクトリが作成されていても、それは全メタデータサーバにおいて合意が取れたということであり、全メタデータサーバにおいてディレクトリ作成が完了しているとは限らないため、そのディレクトリ以下のエントリに対する操作を行うときには、そのディレクトリが完了状態となるのを待つ必要がある。

4. Gfarm ファイルシステムの分散メタデータサーバの実装

4.1 部分ディレクトリ分割

部分ディレクトリ分割については、3.1 節に記述した通り、Gfarm URL 形式のシンボリックリンクにより実装することができる。

4.2 ディレクトリ内分割

ディレクトリ内分割のためには、まず分散ディレクトリの作成が必要となる。作成の手順については、3.5 節で述べたが、分割メタデータサーバリストを保持については実装上の問題となる。ディレクトリに情報を持たせる方法としては、ディレクトリの拡張属性が考えられる。ただし、

拡張属性の設定とディレクトリの作成をアトミックに行うことが必要であり、そのためのインターフェースを用意する必要がある。ディレクトリ作成の完了情報についても、ディレクトリの拡張属性を利用することができる。

キャッシュファイルは、参照ファイルのメタデータサーバ等の情報と、キャッシュしたファイルの取得時刻、i ノード番号、世代番号、またキャッシュ中を示す状態をもつ。ファイルアクセス時、キャッシュが完了しており、有効であればキャッシュしたファイルをアクセスする。ファイルを書込モードでアクセスする場合は、3.3 節で記述したキャッシュのポリシーに従い実装する。キャッシュが行われていない、あるいは無効である場合は、こちらも 3.3 節のポリシーに従い実装する。キャッシュ中のとき、キャッシュを待ってからアクセスするか、参照ファイルをアクセスするかが考えられる。

キャッシュ容量の管理を行うために、キャッシュしたファイル全体の容量の管理を行う。また LRU (Least Recently Used) によるキャッシュファイルの破棄を行うため、二重連結リストを用いキャッシュファイルのアクセス状況を管理する。

分散トランザクションで必要となる、READY ファイル、READY ディレクトリの実装としては、Gfarm ファイルシステムに新しい READY ファイルタイプ、新しい READY ディレクトリタイプを準備する。この新しいタイプでは、3.4 節で述べた通り、排他的作成を行い、stat、open、unlink に対しては ENOENT エラーを返す。また必要なホームメタデータサーバの情報、あるいは分散メタデータサーバリストを拡張属性として持ち、それらは作成時にアトミックに付加される。

5. まとめ

並列プロセスや多数プロセスからの同時ファイルアクセスに対しスケールアウトするための方法として、これまで複数のメタデータサーバでハッシュ分散させる方法が取られてきた。ハッシュ分散はメタデータサーバの負荷分散には効果的であるが、局所性がないため、メタデータサーバ数が多くなったときにネットワークトラフィックが増えてしまう。

本研究では、クライアントとメタデータサーバの局所性を考慮した分散メタデータサーバの設計を行った。この設計では、クリティカルセクションにおけるネットワークトラフィックを抑え、多数のメタデータサーバによるスケールアウトを目指している。同一ディレクトリを複数のメタデータサーバで分散管理する分散ディレクトリの設計においては、各クライアントは近い(担当)メタデータサーバにエントリを作成し、他メタデータサーバからのアクセスはキャッシュファイルを用いる。キャッシュファイルの作成にはコストがかかるが、各クライアントが必ず異なる

ファイルを作成することが保証され、かつそのファイル作成がただちに他のメタデータサーバに反映されなくても構わないような限られた場合においてはこの方式は有用である。このようなケースは多くの広域ファイル共有において当てはまる。このとき、キャッシュファイルの作成は、非同期に作成する。このようにすることにより、ファイル作成時の応答時間を増やすことなく、ディレクトリ内分散を行うことができる。

また、一般的なケースで、各クライアントが重複するファイルを作成する可能性があるケースについては、ファイル、ディレクトリ作成の一意性を保証するための分散トランザクションを提案した。本分散トランザクションは、2PCをベースとし、ファイルシステムで実現できるものである。そのため、2PCのREADY状態に相当するREADYファイルを新たに導入した。また、分散ディレクトリの作成については、分散トランザクションと作成完了を示す完了情報を持たせた。

今後、本設計と実装方針に基づき実装を進め、Gfarmファイルシステムの分散メタデータサーバの評価を進める予定である。

謝辞 本研究の一部はJST-CREST「ポストペタスケールデータインテンシブサイエンスのためのシステムソフトウェア」、「EBD：次世代の年ヨッタバイト処理に向けたエクストリームビッグデータの基盤技術」、「広域撮像探査観測のビッグデータ分析による統計計算宇宙物理学」による。

参考文献

- [1] Bernstein, P. A., Hadzilacos, V. and Goodman, N.: *Concurrency Control and Recovery in Database Systems*, Addison-Wesley (1987).
- [2] Braam, P. J.: *Lustre*. <http://www.lustre.org/>.
- [3] Jannen, W., Yuan, J., Zhan, Y., Akshintala, A., Esmet, J., Jiao, Y., Mittal, A., Pandey, P., Reddy, P., Walsh, L., Bender, M., Farach-Colton, M., Johnson, R., Kuszmaul, B. C. and Porter, D. E.: BetrFS: A Right-Optimized Write-Optimized File System, *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pp. 301–315 (2015).
- [4] McKusick, M. K., Joy, W. N., Leffler, S. J. and Fabry, R. S.: A Fast File System for UNIX, *ACM Trans. Comput. Syst.*, Vol. 2, No. 3, pp. 181–197 (1984).
- [5] Patil, S. V. and Gibson, G. A.: Scale and Concurrency of GIGA+: File System Directories with Millions of Files, *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, pp. 1–14 (2011).
- [6] Ren, K., Zheng, Q., Patil, S. and Gibson, G.: IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion, *Proceedings of the IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)*, pp. 237–248 (2014).
- [7] Simmons, J. S., Leverman, D., Hanley, J. and Oral, S.: Lustre Distributed Name Space (DNE) Evaluation at the Oak Ridge Leadership Computing Facility (OLCF),

- Technical Report ORNL/TM-2016/608, Oak Ridge National Laboratory (2016).
- [8] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M. and Peck, G.: Scalability in the XFS File System, *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, pp. 1–15 (1996).
 - [9] Tatebe, O., Hiraga, K. and Soda, N.: Gfarm Grid File System, *New Generation Computing*, Vol. 28, No. 3, pp. 257–275 (2010).
 - [10] Weil, S. A., Brandt, S. A., Miller, E. L. and Long, D. D. E.: Ceph: A Scalable, High-Performance Distributed File System, *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, pp. 307–320 (2006).
 - [11] 永岡孝, 石津晴崇, 大西健司, 高杉英利, 建部修見: クラウドにおける大規模ストレージシステムの必要性和その検討 ~ Gfarm v2.4 を拡張した EB 級システム ~, 研究報告ハイパフォーマンスコンピューティング (HPC), Vol. 2011-HPC-130, No. 34, pp. 1–6 (2011).
 - [12] 平賀弘平, 建部修見: ノンブロッキングトランザクションに基づく分散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と実装, 研究報告ハイパフォーマンスコンピューティング (HPC), Vol. 2012-HPC-135, No. 28, pp. 1–9 (2012).