平成24年度　　卒業研究論文

キャッシュポリシの動的変更による メタデータサーバの負荷軽減手法

指導教官
松尾 啓志 教授
津邑 公暁 准教授
梶岡 慎輔 助教

名古屋工業大学 工学部情報工学科
平成21年度入学　21115136番

松野雅也

平成25年 2月12日

目 次

1 はじめに

近年，インターネットの普及やネットワークの高速化により大規模なデータ処理の需要が増加の一途を辿っており，それに伴い，大規模なストレージに対する要求が高まっている．しかし，単一のノードのみでストレージを構成するには拡張性に限界があり，また単一障害点となりうるという問題がある．これらの問題を解決するために分散ファイルシステムという技術が提案されている．分散ファイルシステムは複数のノードで構成されたファイルシステムであり，ノードの追加によってストレージ容量の拡張も容易に行うことを可能としている．また，データを複製し，複数のノードに同じデータを配置することでストレージの一部に障害が発生した場合でもデータが失われる可能性を低下させることもできる．一般的な分散ファイルシステムでは，ファイルデータの配置情報や名前空間の管理等を行うために用いる，ファイルの管理情報であるメタデータをストレージから分離し，それらを単一のメタデータサーバで集中的に管理することでストレージに対するアクセス並行性を高め，システム全体の性能向上を実現している．しかし，このような構成の分散ファイルシステムでは単一のメタデータサーバにアクセスが集中してしまうため，メタデータサーバがボトルネックとなり，システム全体の性能が低下するという問題がある．

本研究では，分散ファイルシステムにおける問題点であるメタデータサーバへの集中的なアクセスを緩和する手法の一つである，クライアント側でのメタデータキャッシュに注目した．この手法は，クライアントが一度アクセスしたファイルのメタデータをキャッシュし， 2 回目以降のアクセスでは，クライアントはローカルメモリ上のキャッシュしたメタデータを利用するという手法である．こうすることにより，メタデータサーバへのアクセスを軽減し，負荷の集中を避けることが可能となる．しかし，キャッシュを利用する場合，メタデータサーバとクライアント間でメタデータの一貫性を保証するために制御メッセージが発生する．本研究では，この一貫性制御のために発生する制御メッセージの影響を軽減することを目的とし，一貫性を保証するための既存のキャッシュポリシをファイル単位で設定し、各ファイルの被アクセス状況によりキャッシュポリシを動的に変更する手法を提案し，評価を行った．

本論文の構成は以下の通りである．まず，第 2 章では研究背景として一般的な分散

ファイルシステムの構成，問題点を指摘し，既存の解決策を述べ，関連研究を紹介する．第 3 章では，本研究での提案とその実装について述べ，第 4 章で評価，考察を行い，第 5 章で本研究のまとめを述べる．

2 研究背景

本章では，一般的な分散ファイルシステムの構成と問題点を指摘し，それに対する既存の解決策を述べ，関連研究について紹介する．

2.1 分散ファイルシステム

インターネットの普及に伴って，扱うデータ量が増大し続けている．そのデータ量は数 PB 以上に達しており，ストレージシステムは大規模かつ柔軟な拡張を可能とする構成が要求されている．分散ファイルシステムは，複数の計算機にファイルシステムの機能を分散させることで大規模なストレージを構成するシステムであり，ストレージ容量が不足した場合には新たな計算機を追加することで，柔軟に容量拡張を可能としている．

2.1.1 一般的な構成

一般的な分散ファイルシステムは，ファイルデータの管理情報であるメタデータを，ストレージから分離して扱っている．これにより，ストレージ容量の拡張性と大容量のファイルデータへの高速な読み書きを両立している．通常，分散ファイルシステムへのアクセス手順は，図 1 に示すように，まずメタデータを管理しているメタデータサーバからアクセスしたいファイルのメタデータを取得し，その受け取ったメタデータをもとに，ファイルデータを直接ストレージに要求する．従って，一般的な分散ファイルシステムにおいては，ファイルを要求するクライアントは必ず，メタデータサーバからファイルのメタデータを取得しなければならない．そのため，メタデータサーバの性能がシステム全体に大きな影響を及ぼす可能性がある．

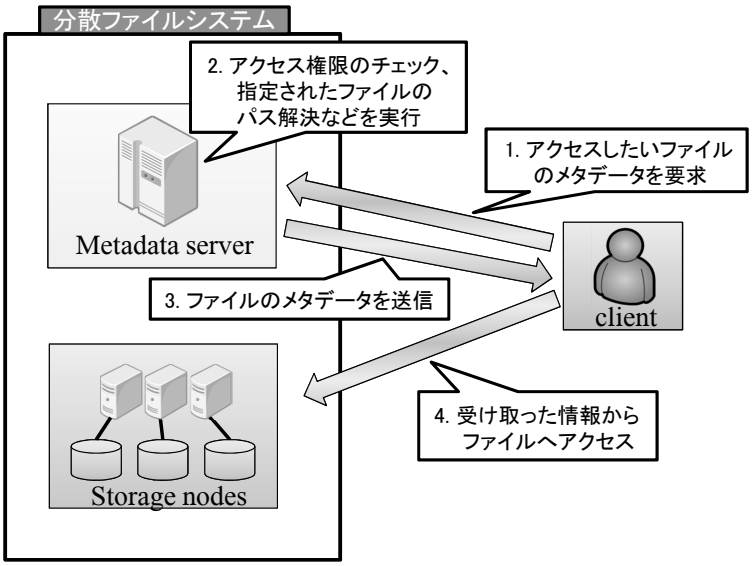


図 1: 分散ファイルシステムの構成とアクセス手順

2.1.2 問題点と解決策

一般的な分散ファイルシステムは，メタデータをストレージから分離して扱うことで，ストレージの高いスケーラビリティとファイルデータの入出力の高速化を実現している．しかし一方で，クライアントは必ずメタデータサーバからメタデータを取得する必要があるため，メタデータサーバに負荷が集中するという問題点がある．負荷の集中により，メタデータサーバの応答が悪くなると，ファイルデータの入出力操作ではなく，メタデータ操作がボトルネックとなり，システム全体の性能が低下することになる．この問題に対処するための解決策として，一度アクセスしたファイルのメタデータをクライアントのローカルメモリ上にキャッシュしておくことが一般的となっている．これにより，クライアントが自身の持つメタデータキャッシュを利用することでメタデータサーバへのアクセスを軽減させることができる．しかしメタデータキャッシュを利用する場合，メタデータの更新が発生すると，クライアントの保持するキャッシュが最新のものではなくなり，メタデータの一貫性が保たれなくなる．そのため，メタデータキャッシュの一貫性を保証する必要がある．

2.2 既存のキャッシュポリシ

分散ファイルシステムにおいてキャッシュの一貫性を保証するために，次の 2 つのキャッシュポリシのうち，いずれかを用いることが一般的である．一つがメタデータサーバがメタデータの更新をクライアントに通知する方法であり，もう一つが，クライアントがアクセスの度に自身のメタデータキャッシュの有効性をメタデータサーバに確認する方法である．本論文では，前者のキャッシュポリシををサーバベース，後者のキャッシュポリシををクライアントベースと定義する．

2.2.1 サーバベース

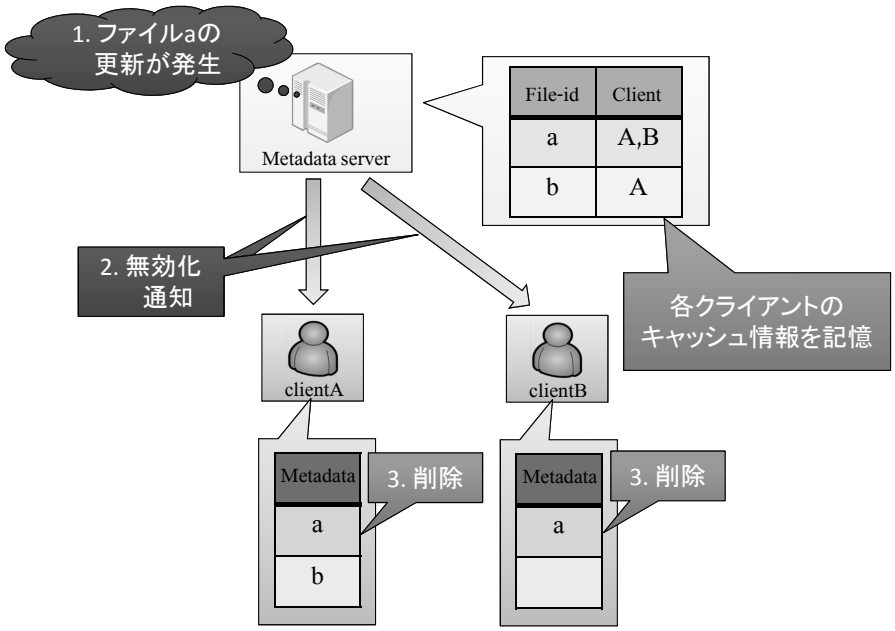


図 2: サーバベースにおける無効化通知の送信

サーバベースのキャッシュポリシでは，メタデータサーバ側が各クライアントのキャッシュ情報を管理することで一貫性を保証している．具体的には，図 2 のように任意のファイルのメタデータが更新される度に，メタデータサーバから，そのメタデータをキャッシュしている全クライアントへ無効化を通知するメッセージが送信される (以

下，このメッセージを無効化通知メッセージと呼ぶ)．これにより，クライアントはメタデータが更新されたことが分かるため，自身のローカルメモリ上からそのメタデータをを削除することで一貫性を保つことができる．サーバベースの利点は，メタデータサーバが一貫性を保証するため，各クライアントは自身のキャッシュの有効性を確認する必要がないという点である．欠点は，メタデータの更新が頻発して発生すると，無効化通知メッセージが増大し，メタデータサーバが一時的に高負荷になってしまう点が挙げられる．また，無効化通知直後はキャッシュが削除されているため，キャッシュミスにより再びパス解決 (ファイルのパス名から i ノードを求める処理) を伴うアクセスが発生することになる．

2.2.2 クライアントベース

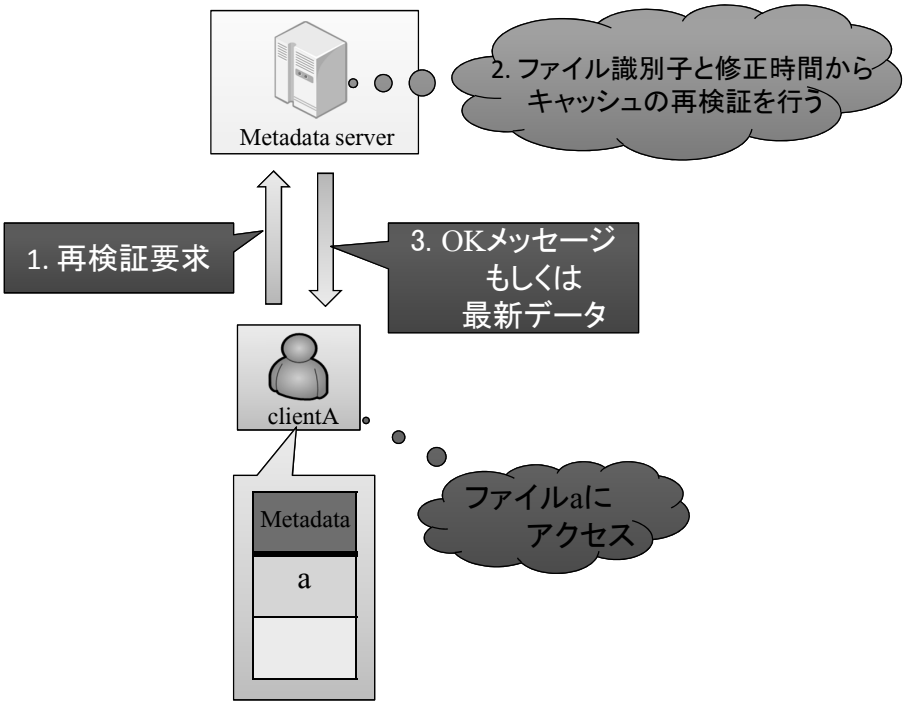


図 3: クライアントベースにおけるキャッシュされたメタデータの再検証

クライアントベースのキャッシュポリシでは，各クライアントがファイルアクセスの度に逐一，自身が持つメタデータキャッシュの有効性をメタデータサーバに問い合

わせることで一貫性を保証する．図 3 で示すように，クライアントはキャッシュしたメタデータを利用する前に，そのメタデータが最新のものであるか否かをメタデータサーバに問い合わせる（この時，パス解決は発生しない）．最新のものであることが確認できた場合は，利用可能となり，最新ではなかった場合にはメタデータサーバが最新のデータを送信することで，一貫性を保証する．クライアントベースの利点は，メタデータサーバはメタデータの変更時に，無効化通知メッセージを送信する必要がない点である．欠点としては，クライアントはたとえキャッシュが最新であっても，メタデータサーバへ有効性を確認するために，メッセージを送信する必要がある点が挙げられる．そのため，メタデータキャッシュの利用時において，サーバベースに比べクライアントベースではオーバヘッドがかかる．

2.3 関連研究

メタデータサーバへのアクセスの集中を緩和する先行研究として，Vilobh Meshram らの研究 [1] や Emorlinskly A. らの研究 [2] はクライアント間でメタデータを取得する手法を提案している．Vilobh Meshram らの研究 [1] ではあるファイルのメタデータを特定のクライアントへ委譲し，委譲されたクライアントがそのファイルのメタデータを管理するというものである．メタデータサーバはどのクライアントへどのファイルのメタデータを委譲しているかを記憶しておくための表を保持し，委譲されたクライアントは他のクライアントへそのメタデータを提供する．委譲を受けていない通常のクライアントはまず，メタデータサーバへアクセスし，どのクライアントが委譲を受けているかを問い合わせ，その情報を用いて委譲されたクライアントへメタデータを要求する．これにより，メタデータサーバへのアクセス集中を緩和することが可能となる．しかし，あるファイルに対するアクセスが集中した場合，メタデータを委譲されたクライアントに負荷がかかることが考えられる．

一方 Emorlinskly A. らの研究 [2] は，クライアント間でメタデータを取得することが可能という点は同じであるが，キャッシュを保持するあらゆるクライアントからメタデータを取得可能という点で異なっている．メタデータサーバは，どのクライアントがどのメタデータをキャッシュしているかを記憶しておくための表を保持し，各ク

ライアントはメタデータサーバからその表の情報を取得して，対応するキャッシュを保持しているクライアントへメタデータを要求する．つまり，あるクライアントは別のクライアントのキャッシュからメタデータを取得し，かつ自身もそのキャッシュを他のクライアントへ提供することができる．この手法は，Vilobh Meshram らの研究 [1] に比べ，特定のクライアントに負荷がかかることがなくなるが，クライアントベースのキャッシュポリシと同様に，キャッシュを利用する際には必ず有効性を確認しなければならない．

またメタデータキャッシュだけでなく，メタデータサーバを複数の計算機で構成する手法もある．これにより，メタデータ操作が分散され，アクセスの集中を避けることができる．しかし，このような場合ではメタデータサーバ間でメタデータの一貫性を保証する必要があり，それを実現するのは一般的に困難である．また，既存の分散ファイルシステムの多くは単一の計算機から構成されるメタデータサーバを利用しているため．本研究ではクライアント側でのメタデータキャッシュに注目することにする．

3 提案手法

本章では，既存のキャッシュポリシであるサーバベースとクライアントベースをファイル単位で，動的に変更する提案手法について説明する．

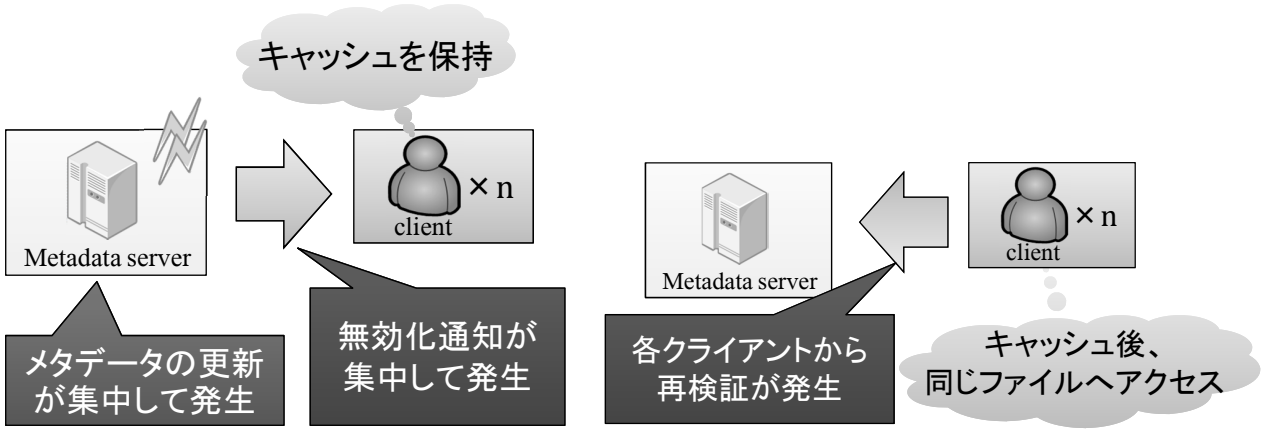
3.1 ファイル単位のキャッシュポリシ

既存の手法では，すべてのファイルに対して同じキャッシュポリシを適用している．しかし，メタデータの一貫性を保証するために，各キャッシュポリシごとに発生する制御メッセージの通信コストは，ファイルへのアクセス状況によって異なることが考えられる．本節では，既存のキャッシュポリシであるサーバベースとクライアントベースで発生する制御メッセージについて説明し，ファイル別にキャッシュポリシを設定する方法について述べる．

3.1.1 一貫性保持のための制御メッセージ

サーバベースにおいて発生する一貫性保持のための制御メッセージは，メタデータサーバがクライアントに対して送信する無効化通知メッセージである．無効化通知メッセージは，メタデータ更新時に発生し，そのメタデータをローカルメモリ上にキャッシュしている全クライアントに対して送信する必要がある．このため，複数のメタデータの更新が短時間に集中して発生した場合には無効化通知メッセージが増大してしまう．その結果メタデータサーバに負荷がかかり，システム性能が低下してしまう恐れがある．従って，更新が集中して発生する状況ではサーバベースで扱うべきではないと考えられる．一方，クライアントベースにおいて発生する一貫性保持のための制御メッセージは，クライアントがキャッシュしたメタデータの有効性を確認するために，メタデータサーバに対して送信する再検証要求である．これは，クライアントがキャッシュしているメタデータを利用する度に発生するため，頻繁に参照される状況ではクライアントベースで扱うべきではないと考えられる。

以上から，各メタデータはその被アクセス状況によって発生する制御メッセージが異なるため，適用すべき，適切なキャッシュポリシが異なるといえる．従って，キャッシュポリシをファイル単位で設定する必要がある．



(a) 無効化通知メッセージの増大　　(b) 再検証によるオーバヘッド

図 4: 一貫性制御のための通信コスト

3.1.2 キャッシュポリシの設定

ファイル単位でキャッシュポリシを設定するために，通常のメタデータにキャッシュポリシ情報を追加する．クライアントは，最初のメタデータフェッチ時にメタデータと一緒にキャッシュポリシ情報を受け取り，次回以降は，そのキャッシュポリシ (クライアントベースもしくはサーバベース) に基づいて，キャッシュしたメタデータの利用時における動作を判断する．つまり，サーバベースであった場合にはメタデータキャッシュを有効性の確認なく利用でき，クライアントベースであった場合には有効性を確認するために，再検証を行うことになる．

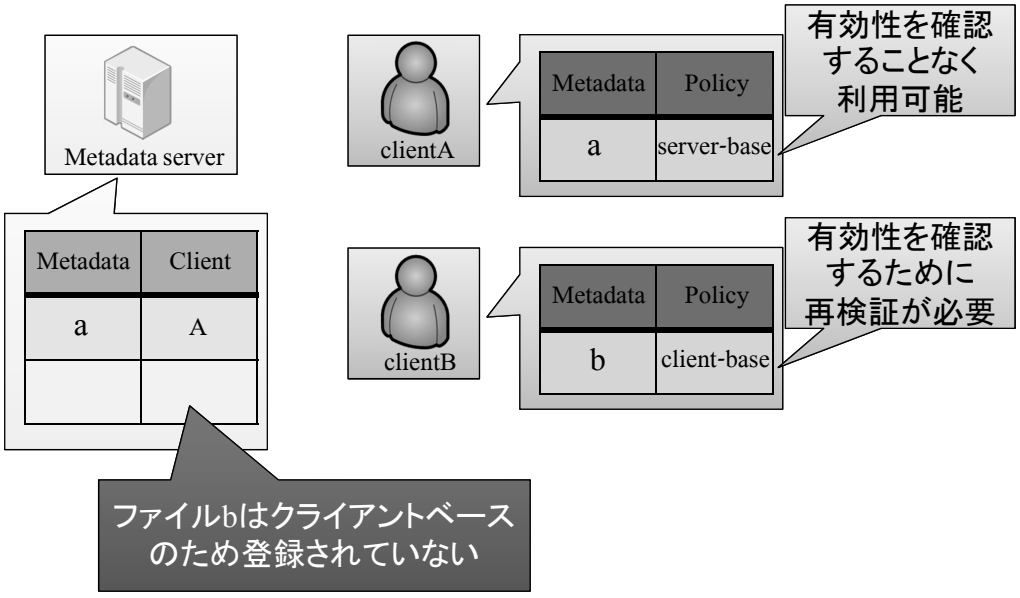


図 5: ファイル別のキャッシュポリシ

3.2 キャッシュポリシの動的変更

ファイル単位でキャッシュポリシを設定することで，各メタデータに適した一貫性制御を行うことが可能となる．しかし，ファイルへのアクセス状況の変化に伴い，適切なキャッシュポリシも変化することが考えられる．そこで，ファイルへのアクセス状

況を観測し，その変化に合わせて適切なキャッシュポリシに変更する．キャッシュポリシの変更による目的は，一貫性制御のための通信コストの軽減である．サーバベースにおける通信コストは，無効化通知メッセージであり，これはメタデータの更新が集中した場合に増大する．また，クライアントベースにおける通信コストは，メタデータキャッシュの有効性を確認するための再検証要求であり，オーバヘッドとなりうる．

その一方，サーバベースではメタデータキャッシュ利用時に再検証は発生せず，クライアントベースではメタデータ更新時において無効化通知メッセージは発生しない．従って，無効化通知メッセージの増大を引き起こすようなファイルをクライアントベースへ変更し，そうでないファイルはサーバベースへと変更することで一貫性制御のための通信コストを軽減することができる．本節では，キャッシュポリシの変更時における一貫性の問題，キャッシュポリシの変更方法，動的に変更を行う条件について述べる．

3.2.1 キャッシュポリシ情報の一貫性問題

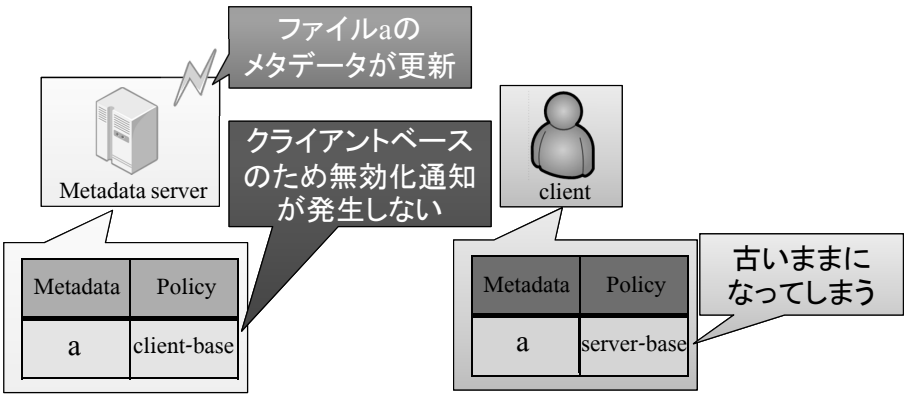
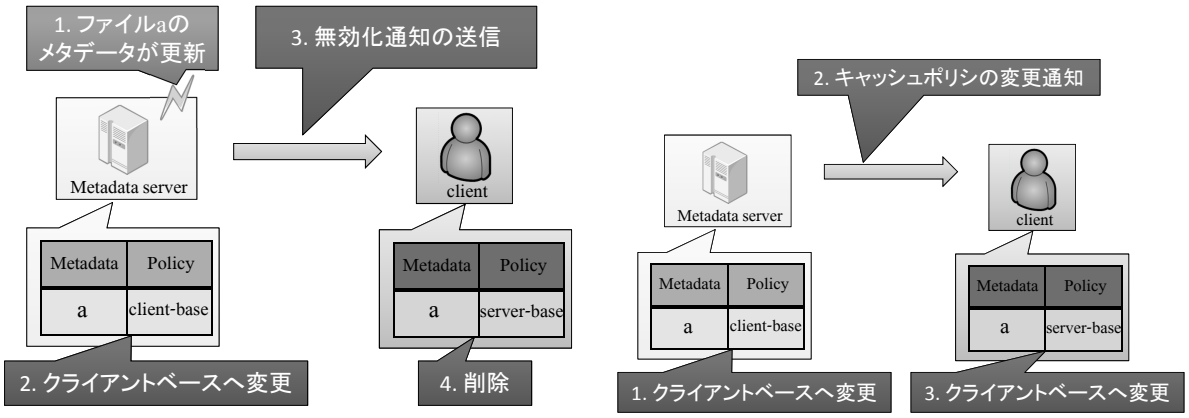


図 6: キャッシュポリシ情報の一貫性問題

キャッシュポリシを変更する際には，メタデータキャッシュの一貫性と同様に，メタデータサーバとクライアント間でキャッシュポリシ情報の一貫性を保証しなければならない．図6に示すように，あるメタデータのキャッシュポリシがメタデータサーバはクライアントベース，クライアントではサーバベースとなっている状況を考える．こ

の場合，メタデータが更新されるとクライアントがキャッシュしているメタデータは最新のものではなくなるが，メタデータサーバではキャッシュポリシがクライアントベースとなっているため無効化通知メッセージが発生しない．従って，クライアントのキャッシュは古いままとなり，メタデータの一貫性が保証されなくなる．そのため，キャッシュポリシの変更タイミングについて考える必要がある．

3.2.2 サーバベースからクライアントベースへの変更



(a) 更新時に変更　　(b) 変更通知による変更

図 7: サーバベースからクライアントベースへの変更方法

サーバベースをクライアントベースに変更する場合は，不適切なタイミングでキャッシュポリシを変更すると，前述したように一貫性の問題が発生する．そこで，キャッシュポリシ情報の一貫性を保証するために 2 つの方法が考えられる．一つは，キャッシュポリシの変更を，メタデータ更新時に限定することである．メタデータ更新時にキャッシュポリシを変更することで，各クライアントは，最新のメタデータをフェッチすると同時に，キャッシュポリシ情報も最新のものとすることができるからである．もう一つの方法は，キャッシュポリシが変更される度にメタデータサーバが，キャッシュポリシの変更を通知するメッセージを全クライアントに送信するものである．この方法の場合，キャッシュポリシの変更をメタデータ更新時に限定されることなく，任意のタイミングで行うことが可能となる．しかし，キャッシュポリシの変更通知は，無効化通知メッセージと同様にキャッシュを保持する全クライアントに送信する必要がある．

サーバベースからクライアントベースへ変更する必要があるのは，無効化通知メッセージの増大を防ぐためである．そのため，キャッシュポリシ変更時にメッセージを送信するのではなく，メタデータ更新時に変更を行う方法を採用する．

3.2.3 クライアントベースからサーバベースへの変更

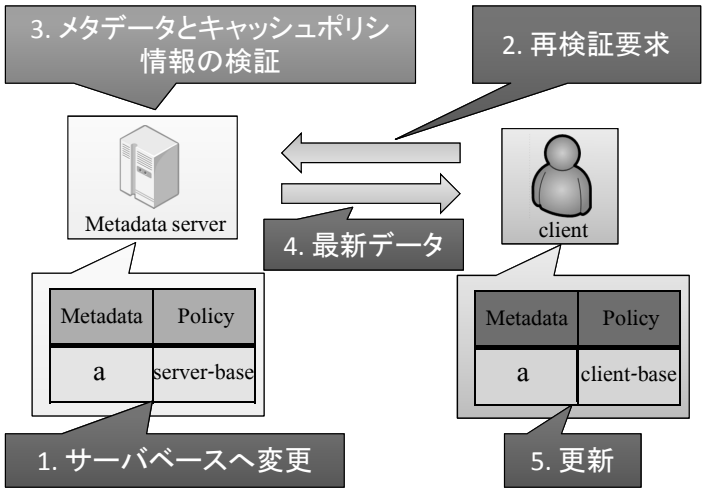


図 8: 再検証時にキャッシュポリシ情報を更新

クライアントベースからサーバベースへ変更を行うのは，メタデータキャッシュの再検証を抑制するためである．つまり，クライアントにおけるキャッシュ利用時のオーバヘッドをなくすことである．クライアントベースからサーバベースへの変更を行う場合には，図 8 のように，各クライアントのメタデータキャッシュの再検証時に，キャッシュポリシ情報の整合性も判定し，一致しない場合にはクライアント側のキャッシュポリシ情報をサーバベースに変更することでメタデータと同様に一貫性を保証することが可能である．そのため，前述したような一貫性の問題は発生しない．つまり，クライアントベースをサーバベースに変更するタイミングは，メタデータ更新時に限定されず，かつキャッシュポリシ情報の変更を通知する必要もない．従って，クライアントベースからサーバベースへの変更は，メタデータサーバにおいて任意のタイミングでキャッシュポリシを変更することで実現できる．

3.2.4 変更条件

キャッシュポリシの動的変更により達成すべきことは，サーバベースにおける無効化通知メッセージの増大を避け，かつクライアントベースによる再検証のオーバヘッドを削減することである．これは，無効化通知の増大を引き起こすファイルのみをクライアントベースへ変更し，それ以外のファイルをサーバベースとして扱うことで達成できる．無効化通知メッセージの増大を避けるためには，メタデータの更新が集中することを予測し，そのようなファイルをクライアントベースに変更することが必要となる．本実装では，メタデータの更新の集中を予測するための変更条件として，各ファイルにおける更新頻度を検出する方法と更新タイミングが近いファイル群を検出する方法の二つをそれぞれ実装した．前者の条件は，メタデータの更新頻度が高いファイルほど他のファイルと更新タイミングが重なりやすく，無効化通知メッセージの増大を引き起こす可能性が高いと考えられるためである．後者の条件は，更新タイミングが近いファイル群を観測することで無効化通知メッセージの増大を引き起こしたファイルを検出するものである．そして，これらの条件を満たさない場合にサーバベースへと変更することで，無効化通知メッセージの増大を避けつつ，再検証のオーバヘッドを削減する．

3.2.4.1 更新頻度による変更

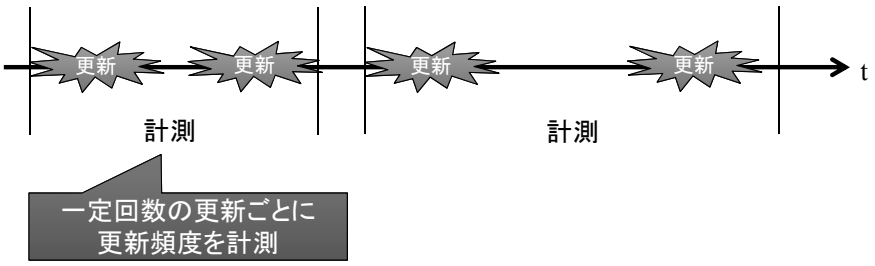


図 9: 更新頻度の計測

この方法では，メタデータに一定回数の更新が発生する度にそれまでに経過した時

間から単位時間当たりの更新頻度を求め，その値がある閾値を越えた場合にクライアントベースへと変更し，閾値に満たなかった場合はサーバベースへと変更する．すなわち，更新頻度が高いファイルをクライアントベースとし，そうでないものをサーバベースへと変更する．

3.2.4.2 更新タイミングによる変更

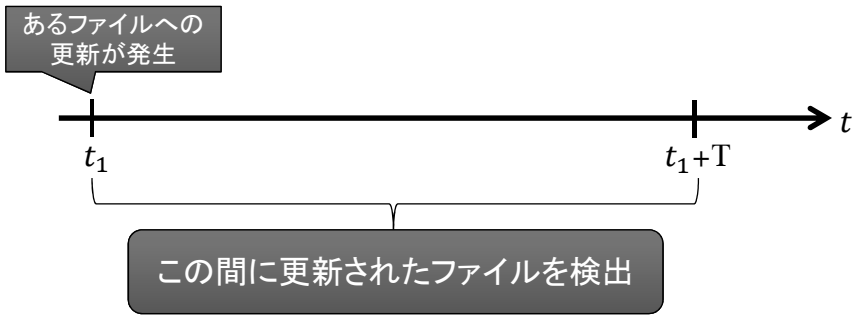


図 10: 一定時間内に更新されたファイルの検出

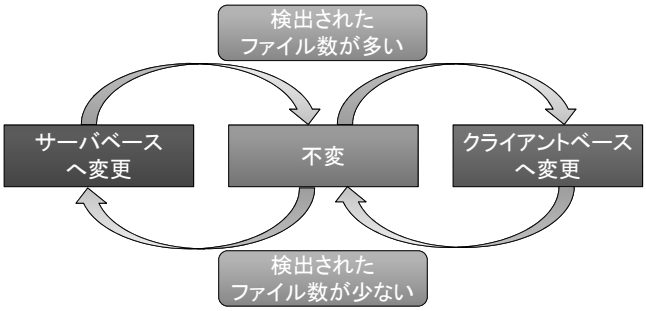


図 11: キャッシュポリシ状態の遷移

この方法では，あるファイルのメタデータの更新発生後，一定時間内にメタデータが更新されたファイルを検出し，そのファイル数が多い（無効化通知メッセージの増大を引き起こした）場合には，検出されたファイル全てをクライアントベースへ変更

するものである．つまり，メタデータの更新タイミングが近いファイルすべてをクライアントベースへと変更する．しかし，検出される度に変更を行うとキャッシュポリシの変更が頻繁に起きる可能性があるため，図 11 のように 3 つの状態を遷移するようにする．つまり，更新タイミングが他の多数のファイルと近かった場合にクライアントベースの方へ，そうでない場合はサーバベースの方へと状態遷移するようにし，各ファイルのメタデータの更新時に状態をチェックすることでキャッシュポリシの変更を行うようにする．

4 評価と考察

本章では，既存のキャッシュポリシであるサーバベース，クライアントベースそれぞれにおける動作を確認し，各キャッシュポリシで生じる問題について評価を行う．そして，本提案手法を適用した場合との比較を行い，その有効性を示し，キャッシュポリシ変更条件についての考察を行う．

4.1 評価環境

既存のキャッシュポリシであるサーバベース，クライアントベースを分散ファイルシステムの一つである Gfarm 上にそれぞれ実装し，そこへ本提案手法であるファイル単位でのキャッシュポリシの動的変更を実装した．評価環境は以下の通りであり，以降，本章の評価における分散ファイルシステム Gfarm の構成は，メタデータサーバを 1 台，クライアントノードを 5 台としている．

表 1: 評価環境

Gfarm version	gfarm-2.5.5
CPU	Core i5 750 / 2.67GHz
Memory	8GB DDR3-1333MHz Dual Channel
OS	Ubuntu Server 12.04
Network	Ethernet(1000BASE-T)

評価を行うにあたり，ファイルの読み書きによるメタデータへのアクセスを想定して，touch() と stat() を用いた．touch() はメタデータの更新を伴う処理であり，stat() はメタデータの取得を伴う処理である．また，MPI を用いて 1 クライアントノード上に複数のプロセスを生成し，各プロセスに独自のマウントポイントを持たせることで複数クライアントによるアクセスを仮想的に実現している．

4.2 予備評価

本提案手法の評価を行うにあたり，まず実装したサーバベース並びにクライアントベースの動作と問題点を確認するための予備評価を行った．

4.2.1 動作確認

まず始めに，動作を確認するためにクライアントノード上に 50 クライアントを生成し，以下のような実験を行った．

1. ある 1 クライアントが 50 ファイルに対して touch 操作を行う
2. touch 操作終了後，残りの 49 クライアントが各ファイルに対して順番に stat 操作を 2 回行う

この一連の操作を繰り返し，その間におけるメタデータサーバ上での CPU 使用率の推移を測定した．

実験結果を図 12 に示す． CPU 使用率が高くなっている部分（25s，44s，etc.）が touch 操作時を表し，その間の CPU 使用率が低くなっている部分が stat 操作時を表している．まず touch 操作時に注目すると，クライアントベースに比べサーバベースの方が CPU 使用率が高くなっている．これは，クライアントベースでは通常通り更新のみ行われるのに対し，サーバベースではキャッシュしている全クライアントに対する無効化通知メッセージが発生しているためだと考えられる．続いて stat 操作時に注目する．サーバベースの場合，1 回目の stat 操作時ではキャッシュが削除されているため通常通りパス解決を伴うアクセスが生じ，2 回目の stat 操作時にはキャッシュが存在す

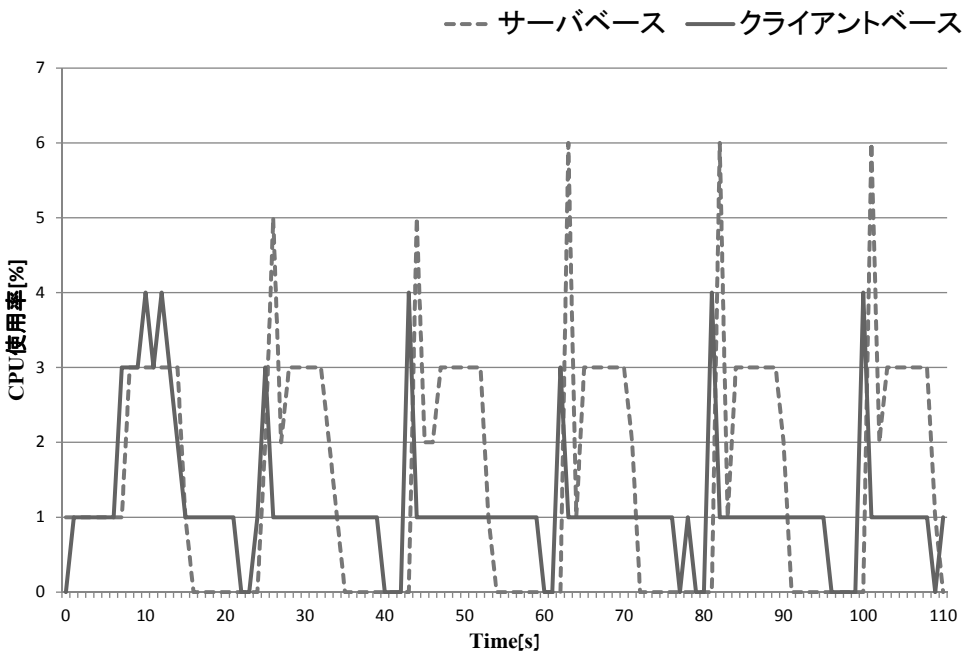


図 12: メタデータサーバ上の CPU 使用率の推移

るため即座に利用でき，アクセスが生じないことから，一定の CPU 使用率が続きその後で，CPU 使用率がほぼ 0%となっている．一方クライアントベースでは，1 回目，2 回目どちらの stat 操作時にも必ず再検証要求が発生するが，パス解決は発生しない．従って，CPU 使用率が低く，ほぼ一定となっている．以上から，各キャッシュポリシが正しく動作しているものと考えられる．

4.2.2 制御メッセージの影響

まず，サーバベースで発生する無効化通知メッセージの影響について実験を行った．実験内容は以下の通りで，無効化通知メッセージの増大を引き起こすために複数ファイルに対して並列に更新を行っている．

1. ある 5 クライアントが並列に，合計 100 ファイルに対して touch 操作を行う（1 クライアント当たり 20 ファイル）

2. touch 操作終了後に，stat 操作を行うクライアント数を 50，100，150，200 とする

この一連の操作を繰り返し，touch 操作時における CPU 使用率の最大値をクライアント数別に測定した．

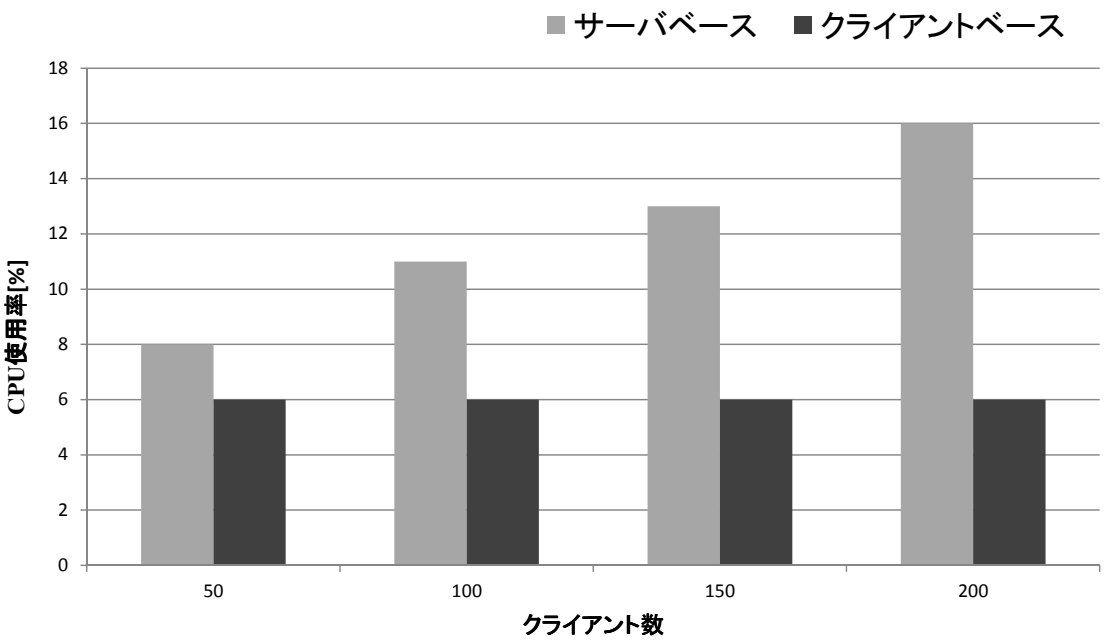


図 13: クライアント数別の CPU 使用率

実験結果を図 13 に示す．クライアントベースでは，stat 操作を行うクライアント数の増加による影響はなく，CPU 使用率はほとんど変化しないことが分かる．これはクライアントベースの場合，更新時には制御メッセージは発生しないためである．一方，サーバベースの方では stat 操作を行うクライアント数に比例して，CPU 使用率も増加していることが分かる．これは，サーバベースの場合， touch 時にキャッシュしている全クライアントに無効化通知メッセージを送信する必要があるためである．この実験結果から，サーバベースにおける無効化通知メッセージの増大の影響は大きく，これを回避することは有益であり，クライアントベースへと変更することで回避できると考えられる．

続いて，クライアントベースにおける再検証による影響について評価を行った．評価を行うにあたり，メタデータ操作のパフォーマンス測定のために mdtest ベンチマークを利用した．mdtest は MPI を利用した並列メタデータベンチマークテストで，指定

したディレクトリ直下に並列にファイルとディレクトリの create/stat/delete を行う．本評価では，再検証による影響を測定することが目的のため，300 個のファイルに対し単一プロセスによる stat 操作を行った．また，マウントの際には fuse のオプションを指定することでカーネルキャッシュを行わないようにしている．評価はキャッシュをしないもの，クライアントベース，サーバベースの 3 つのパターンについてスループットを評価した．

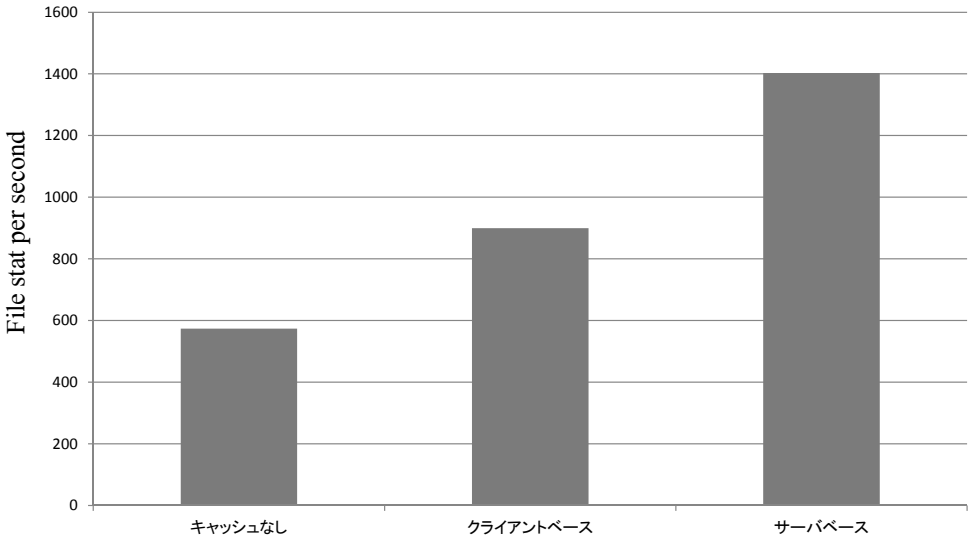


図 14: 再検証の影響

評価結果を図 14 に示す．まず，キャッシュをしないものは各 stat 操作時にパス解決を伴うアクセスが発生するため，3 つの中でスループットが最も悪い結果となった．続いて，クライアントベースはキャッシュを保持できる分，キャッシュしないものに比べ，スループットは良いことが分かる．しかし，サーバベースと比べスループットが低下している．これは，サーバベースではキャッシュの有効性を確認する必要がなく再検証が発生しないためであると考えられ，キャッシュの再検証によるオーバヘッドの影響であるといえる．

以上の予備評価から，サーバベースにおける無効化通知メッセージの増大を避けつつ，クライアントベースにおける再検証を削減する必要があることを確認した．

4.3 既存手法との比較

本節では，提案手法とサーバベース，クライアントベースそれぞれを比較し，提案手法の優位性を示す．

サーバベースにおける無効化通知メッセージの増大を回避し，かつクライアントベースにおける再検証のオーバヘッドを軽減できることを示すために，クライアントノード上に 200 プロセスを生成し，単一ディレクトリ内の 300 ファイルに対して実験を行った．実験内容は以下の通りであり，一部のファイルに対して更新を集中させている．なお，各ファイルは実験の開始時に生成し，このファイル数の割合は，更新タイミングが重なるファイルに比べ，そうでないファイルの方が一般的に多いことを想定している．

1. ある 5 クライアントが 300 ファイル中，特定の 100 ファイルに対して並列に touch 操作を行う
2. touch 操作終了後，200 クライアントが 300 ファイルに対して順番に stat 操作を 2 回行う

この一連の操作を繰り返し，その間におけるメタデータサーバ上の CPU 使用率の推移を測定した．実験はサーバベース，クライアントベース，提案手法それぞれに対して行っている．また，提案手法においてキャッシュポリシの変更条件は更新タイミングによる変更を用い，初期状態はサーバベースとしている．

実験結果は図 15，16，17 に示す．サーバベースでは，touch 操作時（570s，945s，etc.）に一時的に負荷が高くなり，その後一定時間負荷が高くなっていることが分かる．これは，無効化通知メッセージの増大による影響とその後のキャッシュミスによるものだと考えられる．クライアントベースでは，サーバベースに比べ touch 操作時（576s，948s，etc.）の最大負荷が 7%に抑えられているが，stat 操作開始時から終了時まで（579s∼915s，963s∼1299s，etc.）は，ほぼ一定の負荷がかかっている．これは，クライアントベースの場合には touch 操作時にいて制御メッセージは発生しないが，キャッシュ利用時において再検証が発生するためだと考えられる．一方，提案手法では 1011s あたりまでサーバベースと同様の動作をしているが，それ以降ではサーバベースのように touch 操作時（1320s，1695s，etc.）において CPU 使用率が 7%以上となるよ

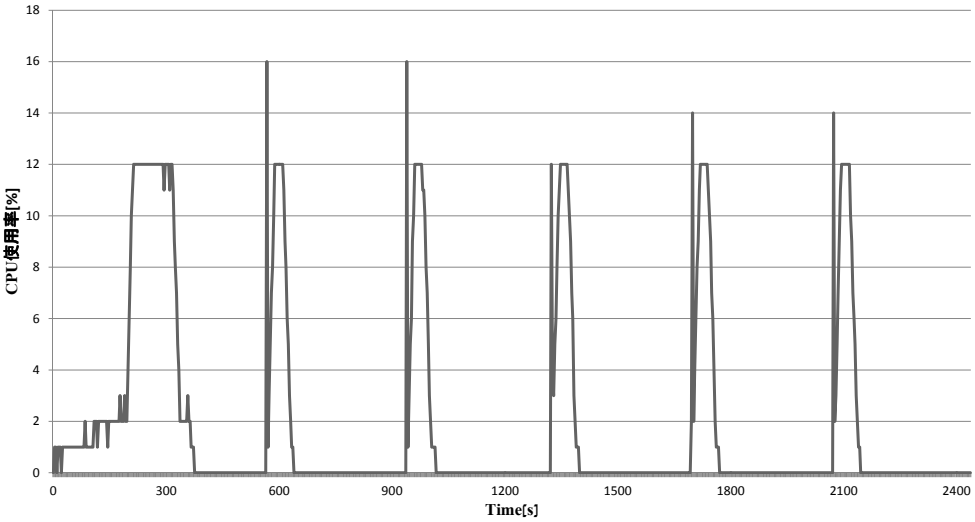


図 15: サーバベースでの CPU 使用率の推移

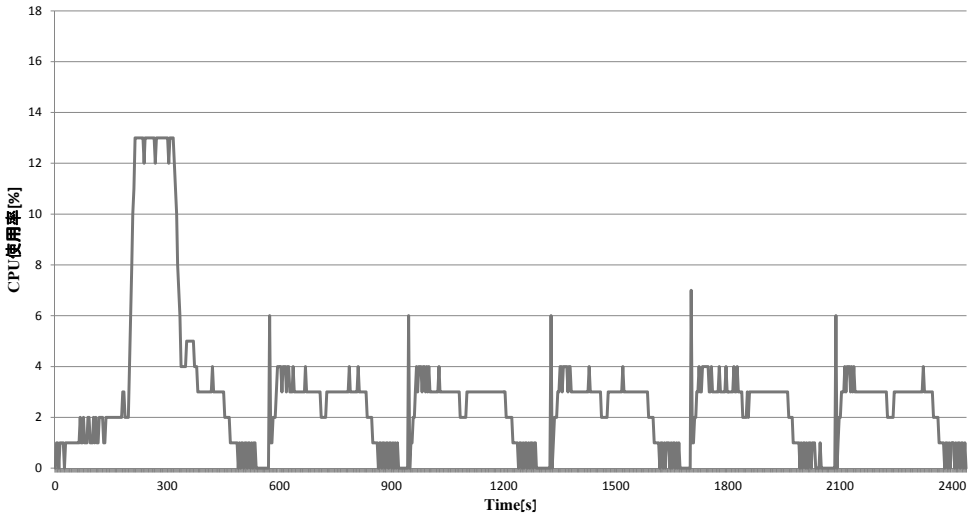


図 16: クライアントベースでの CPU 使用率の推移

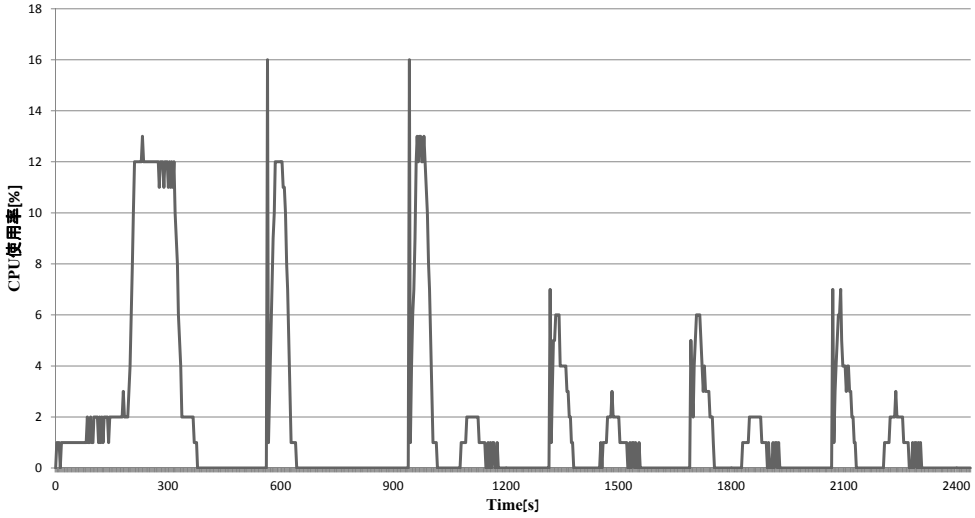


図 17: 提案手法での CPU 使用率の推移

うな状態がなくなり，かつ stat 操作時（1347s∼1683s，1731s∼2067s，etc.）においてもクライアントベースより負荷がかかっていないことが分かる．これは，更新タイミングが集中しているファイルを検出し，それらだけをサーバベースからクライアントベースへと変更することで無効化通知メッセージの増大による影響が軽減され，それ以外のファイルはそのままサーバベースとして扱ったためであると考えられる．また，更新タイミングが重なるファイルはクライアントベースとして扱っているため，サーバベースと比べ stat 操作時（1059s∼1203s，1443s∼1587s，etc.）において，一時的に CPU 使用率が上がっていることが分かる．

また，提案手法適用時のネットワークの通信量の変化についても測定した．測定には nethogs コマンドを用い，メタデータサーバにおいて発生した通信量の推移を測定している．

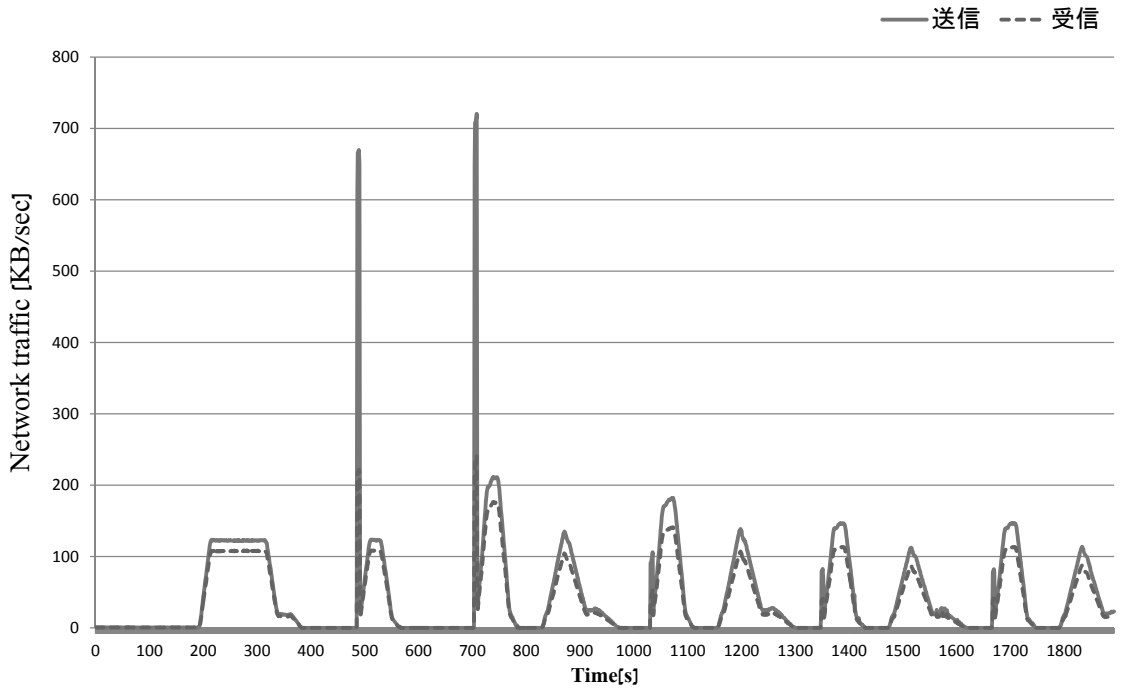


図 18: メタデータサーバで発生した通信量

測定結果は図 18 に示す．結果から，493s，698s 辺りにおいて通信量が一時的に多くなっていることが分かる．これは，すべてのファイルのメタデータをサーバベース

として扱うことで無効化通知メッセージが増大したためである．しかしそれ以降では，通信量が一時的に多くなることがなくなっていることが分かる．これは，一部のメタデータのキャッシュポリシをサーバベースからクライアントベースへ変更することで，無効化通知メッセージの増大を抑制したためである．従って，キャッシュポリシの変更により，通信量を抑えることができていることが分かる．

続いて，提案手法適用時のスループットについて測定した．測定には mdtest を用い，単一クライアントによる，300 ファイルに対する stat 操作を行った．ただし，この時もマウント時にはカーネルキャッシュを行わないようにし，提案手法において，その内の 100 ファイルはクライアントベースとして扱い，残りの 200 ファイルはサーバベースとして扱う．また，クライアントベースとサーバベースの測定結果は，図 14 のものと同一である．

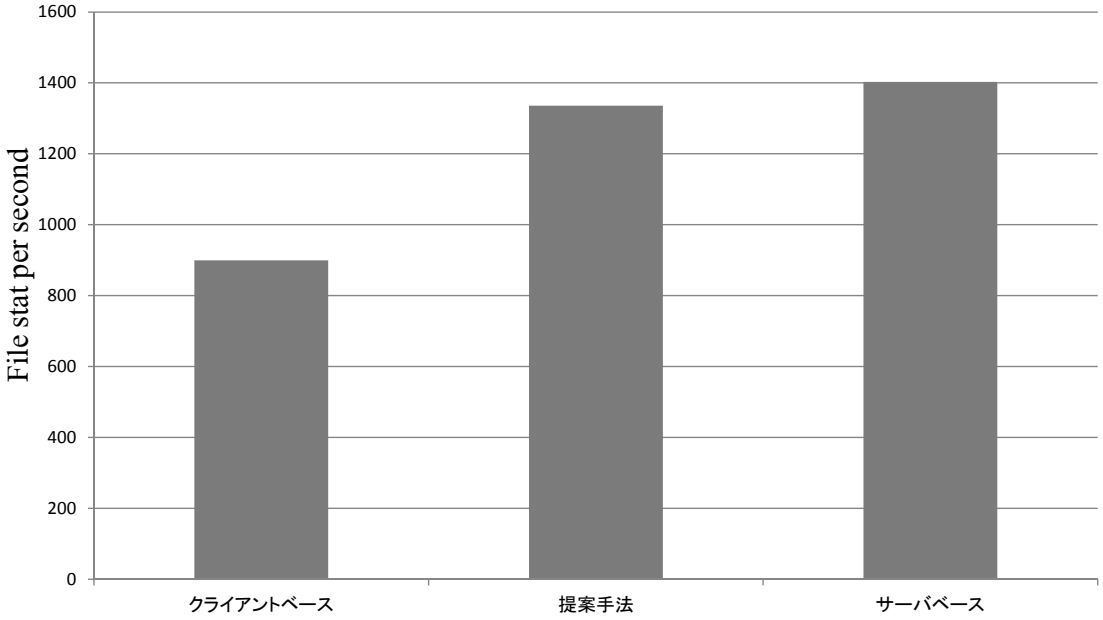


図 19: 再検証のオーバヘッドの軽減

図 19 に示す評価結果から，提案手法ではクライアントベースとサーバベースが混合した状態となっているため，クライアントベースよりスループットが高く，サーバベースよりは低くなっていることが分かる．

以上の評価から，提案手法では無効化通知メッセージの増大を回避しつつ，再検証によるオーバヘッドを軽減することが可能であることを示した．

4.4 変更条件の考察

現在の実装では，キャッシュポリシの変更条件として更新頻度による変更，更新タイミングによる変更の 2 つを実装している．更新頻度による変更の場合，更新頻度が高いファイルをクライアントベースへと変更するため，無効化通知メッセージの増大を回避できる可能性が高いと考えられる．しかし，更新頻度が高いからといって，無効化通知メッセージの増大を引き起こすわけではないため，必要のない場合にもクライアントベースへの変更が発生し，再検証によるオーバヘッドにつながる可能性がある．また逆に，更新頻度が低くてもメタデータサーバにおいてメタデータの更新が集中して発生する場合が考えられ，無効化通知メッセージの増大を防ぐことができない可能性もある．一方，更新タイミングによる変更では更新タイミングが近いファイル群をすべてクライアントベースへと変更するため，実際に無効化通知メッセージの増大を引き起こしたファイルのキャッシュポリシを変更することになり，より適した変更条件だと考えられる．しかし，現在の実装している変更方法では更新タイミングが他の多数のファイルと近い場合が連続して発生した時のみしかキャッシュポリシが変更されず，直近の更新状況から変更を行っている．

以上から，更新タイミングが他の多数のファイルと近かった頻度，つまり無効化通知メッセージの増大を引き起こした頻度を用いて変更する方法が考えられる．この変更条件を用いることで，より適切にメタデータの更新タイミングが他のファイルと近いファイルをクライアントベースへ変更することができると考えられる．

5 まとめと今後の課題

本研究では，分散ファイルシステムにおけるメタデータキャッシュに注目し，その一貫性制御を行う既存の方法であるサーバベース，クライアントベースについて述べ，それらをファイル単位で，動的に変更することでメタデータサーバの処理負荷を軽減

する手法を提案した．各キャッシュポリシで発生する問題である無効化通知メッセージの増大，キャッシュの再検証によるオーバヘッドを評価し，提案手法による有効性を確認した．その結果，提案手法では無効化通知メッセージの増大を回避し，かつキャッシュの再検証によるオーバヘッドを軽減できることを示した．

本研究の今後の課題としては，4.4 節で述べたようなキャッシュポリシの変更条件を用いて，より適切なキャッシュポリシの変更を行うようにすることが考えられる．また，現在の実装では，メタデータの更新時にしか変更が起こらないため，ファイルの生存期間からメタデータの再検証時に変更を行うなどより柔軟なキャッシュポリシの変更を行うことも考えられる．さらには，サーバベースにおける無効化通知後のキャッシュミスをなくすため，無効化通知メッセージを受け取ったクライアントはキャッシュを削除するのではなく再検証を行うように改善することで，よりメタデータサーバの負荷を軽減することが可能であると考えられる．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜った名古屋工業大学の松尾啓志教授，津邑公暁准教授，梶岡慎輔助教に深く感謝致します．また，本研究の際に多くの助言，協力をして頂い松井俊浩准教授，齋藤彰一准教授，松尾・津邑研究室並びに齋藤研究室，松井研究室の皆様に深く感謝致します．

参考文献

[1] Vilobh Meshram, Xiangyong Ouyang, and Dhabaleswar K. Panda. Minimizing Lookup RPCs in Lustre File System using Metadata Delegation at Client Side. *The Ohio State University on Department of Computer Science and Engineering*, March 2011.

[2] Ermolinskly A. and Tewarl R. C2Cfs: A Collective Caching Architecture for Distributed File Access. *in Proceedings of IEEE International Conference on High Performance Computing and Communications*, pp. 642–647, November 2009.

[3] Qian Zhang, Dan Feng, and Fan Wang. Metadata Performance Optimization in Distributed File System. *in Proceedings of IEEE/ACIS International Conference on Computer and Information Science*, pp. 476–481, November 2012.

[4] Xiuqiao Li, Bin Dong, Limin Xiao, Li Ruan, and Dongmei Liu. CEFLS: A cost-effective file lookup service in a distributed metadata file system. *in Proceedings of IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 25–32, December 2012.

[5] Venkata Duvvuri, Prashant Shenoy, and Renu Tewari. Adaptive leases: A strong consistency mechanism for the world wide web. *in Proceedings of IEEE Transaction of Knowledge and Data Engineering*, pp. 1266–1276, July/August 2003.

[6] 建部修身, 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：ハイパフォーマンスコンピューティング, 2007-HPC-113, pp. 7–12, December 2007.

[7] 建部修身, 曽田哲之. 広域分散ファイルシステム gfarm v2 の設計と実装. 情報処理学会研究報告：ハイパフォーマンスコンピューティング, 2004-HPC-099, pp. 145–150, July 2004.

[8] mdtest. `http://sourceforge.net/project/mdtest/`.