分散ストレージのための、因果順序に基づく耐競合バージョン付けプロトコルの設計と評価

慶應義塾大学 環境情報学部
村井研究室
高 星英

指導教員
徳田 英幸
村井 純
楠本 博之
中村 修
南 政樹

概 要

インターネットを利用したコラボレーションシステムには様々な要求がある。万人がそのようなシステムの恩恵を受けられるようにするためには、中でも特に、管理者不在性が重要となる。結果として、管理者による複雑な設定や、日々の保守作業を必要とするサーバを排し、ユーザが普段、利用しているコンピュータに情報や処理を分散させることによる、いわばピアグループによるコンピューティングへの要求が高まっている。

多くのコラボレーションシステムでは、実験的に何かを作って、評価した後にやり直すといったことを繰り返すため、バージョン管理が必要となる。ピアグループが情報を共有するためには、マルチキャストにより分散ストレージを維持するというアプローチが考えられるが、バージョン管理を実現するためにはストレージの一貫性を保証する必要があるため、この場合、マルチキャストのメッセージを全順序 (total order) で順序付けることが考えられてきた。ところが、全順序を保証するマルチキャストプロトコルは高コストであり、帯域を多く消費し、また規模透過性に乏しい傾向があった。

この研究では、この問題に対し、ストレージの一貫性の保証に対する要求を弱め、限定的な期間において全順序が失われることを許すことにより、帯域の消費を抑えた軽量なバージョン付けプロトコルを提案する。失われた全順序は、決定的なアルゴリズムにより回復される。順序付けが一時的であるにせよ失われることの不便は、常に因果順序 (causal order) を保証することにより解消する。

このプロトコルは未実装だが、Java スレッドを用いたシミュレーションを行ない、サーバによるバージョン付けと比較して、帯域の消費はかえって少なく、また、規模が小さいグループでの利用については、個々のコンピュータへの負荷も小さく抑えられることを示した。

Abstract

This thesis is titled "Design and evaluation of a conflict-tolerant versioning protocol based on causal ordering to be applied for distributed storage systems."

There are many requirements for collaborative systems over the Internet. One of the most important requirements is administration-freedom, so that everyone can participate in such a system. As a result, needs are boosting for so-called peer-group computing by distributing information and processing over the computers owned by users, instead of centralizing them to a server which requires complex settings and daily maintenance.

Many collaborative systems require version management, because process of creation, evaluation and redoing is repeated in such systems. Maintaining a distributed storage by multicasting is an approach towards sharing information in a peer group, in which case total ordering of messages has been considered in order to guarantee the consistency of the storage. Such consistency is necessary if version management is to be implemented. However, cost of communication is high for the multicast protocols which guarantee total ordering of messages. They consume much bandwidth, and lack scalability.

This research is an attempt to tackle the problem by weakening the requirement for consistency of the storage, and allowing total ordering to be lost in limited durations. This thesis proposes a light-weight versioning protocol which does not consume much bandwidth. The lost ordering is regained by a deterministic algorithm. Inconvenience of temporarily losing ordering is resolved by always guaranteeing causal ordering.

The protocol has not been implemented, but an experiment in a simulated environment using Java threads has been conducted. The result shows that less bandwidth is used compared with versioning by a server. The load for each computer is low in case of a small group.

目 次

図 目 次

表 目 次

第1章　序論

この章では、研究の目的とそこに至った経緯を説明する。また論文中、特に注意を要する用語を定義する。論文の残りの部分の構成を示す。

1.1　目的

この研究は、全体を統御するプロセスを持たない分散ストレージシステムにおいて、大局的には一貫しているバージョン管理を実現するためのバージョン付けプロトコルを提案し、その評価を行なうことを目的とする。

1.2　経緯

インターネットを利用したコラボレーションシステムには様々な要求がある。

慶應義塾大学 SFC 研究所を中心とするデジタルシネマ研究コンソーシアム [4] (Digital Cinema Consortium; DCC) にて現在、開発が進められている遠隔映像編集システムは、インターネットを利用したコラボレーションシステムの一例である。このシステムでは、管理者不在性の追求に重点の 1 つが置かれている。同コンソーシアムの目的はデジタル映像文化の発展への寄与であり、このシステムを様々な人々が用いることにより、映像制作の裾野が広がり、新しい才能が輩出することを期待している。そのためには、ネットワークに関する特別な知識がなくてもこのシステムを利用できる必要がある。また、特別な計算機資源を要しないことも望ましい。これらのことから、複雑な設定や日々の保守作業を必要とするサーバを排し、ユーザのコンピュータのみでグループを形成してコラボレートできることを目指したシステム設計が現在行なわれている。

コラボレーションでは、各自が持つ素材を実験的に組み合わせて新しい物を作り、評価し、やり直すといった行為が繰り返される。人間のそのような活動を支援するためには、素材や、その組み合わせの変更の履歴を保存し、必要な時に呼び出せることが必要である。すなわち、バージョン管理の実現が望まれており、その基礎となるバージョン付けについて、この研究で考えることにした。

1.3　用語

この論文における用語で、特に注意を要するものを表 1.1 にまとめる。

その他、例えば参考となるシステムに固有な用語等は本文中で説明してから用いる。

表 1.1: 本論文における用語の定義

項番	用語	意　味
1	SCM (Software Configuration Management)	ソフトウェア構成管理。複数人でソフトウェアを開発および保守する目的でバージョン管理を行なうシステム。
2	オーバレイネットワーク (overlay network)	既存のリンクを用いて、その上位層における目的に応じて仮想的なリンク付けを行ない、構成するネットワーク。この論文では IP の上位層で仮想化を行なう例に限定する。
3	受信 (receive)	メッセージの順序付けを行なうメッセージ配信システムにおいて、システムが自他のノードからメッセージを受け取ること。メッセージを受信した時点ではアプリケーションは関知しない。
4	譲渡 (deliver)	メッセージの順序付けを行なうメッセージ配信システムにおいて、システムにより受信されたメッセージをアプリケーションに引き渡すこと。メッセージの順序付けを行なう場合、システムでメッセージをバッファし、順序を入れ替える等の操作を行なうため、受信と譲渡を区別する必要がある。
5	セイフティ (safety)	分散システムの特性のうち、悪いことが起きないという形式で記述されるものの総称。
6	バージョン管理 (version management)	バージョン付け (versioning) の機構に基づき、ユーザによる新しいバージョンの作成や、古いバージョンの参照、回復等、ユーザと変更の履歴との間のインタフェースを提供するシステム。
7	バージョン付け (versioning)	何に対して、どのようにバージョン/リビジョンを割り振るかというポリシーおよびその実現。バージョン管理 (version management) とは異なる。
8	ピアグループ (peer group)	役割において対等なノードから成る (オーバレイ) ネットワーク。
9	プロトコル (protocol)	この論文では分散アルゴリズムの意味で用いる。
10	マルチキャスト (multicast)	自ノードを含むピアグループに対して等しい内容のメッセージを配信すること。この論文を通して IP マルチキャストという意味では用いない。
11	ライブネス (liveness)	分散システムの特性のうち、よいことがいつか起きるという形式で記述されるものの総称。

1.4 本論文の構成

この論文の残りは次のように構成される:

第 2章 背景

この研究の契機であり、最終的に実現すべき目標である遠隔映像編集システムについて説明する。また、技術的背景として、当該システムで用いられるであろうオーバレイネットワークの原理とその特徴、およびマルチキャストとメッセージの順序付けについてまとめる。

第 3章 問題の定義

遠隔映像編集システムで実現されるべき分散バージョン管理を、コンピュータソフトウェア開発において現在広く用いられている SCM ツールを参考にしてモデル化する。その上で開発上の要求を整理し、バージョン付けについての問題を明らかにする。

第 4章 プロトコル設計

本研究における中核となるアイデアである、因果順序に基づく耐競合バージョン付けプロトコルを示す。

第 5章 検証と評価

設計したプロトコルが満たすべき特性をインフォーマルな数理的手法により検証する。また、シミュレーションによる実験とその結果および評価を示す。

第 6章 関連研究

分散ストレージにおけるオブジェクトのバージョン付けに関連する既知の研究を紹介し、この研究との比較を行なう。

第 7章 結論

まとめを行ない、問題点を明らかにし、今後の課題を示す。

第2章　背景

この章では、この研究の契機であり、最終的に実現すべき目標である遠隔映像編集システムについて説明する。また、技術的背景として、当該システムで用いられるであろうオーバレイネットワークの原理とその特徴、およびマルチキャストとメッセージの順序付けについてまとめる。

2.1　遠隔映像編集システム

2.1.1　システムの概要

遠隔映像編集システムは、従来、密室で行なわれていた映画の編集作業を、秘密性を保ったままインターネット上に拡張し、物理的に離れた場所にいる関係者が共同して行なえるようにするシステムである。そのシステム構成を図 2.1 に示す。

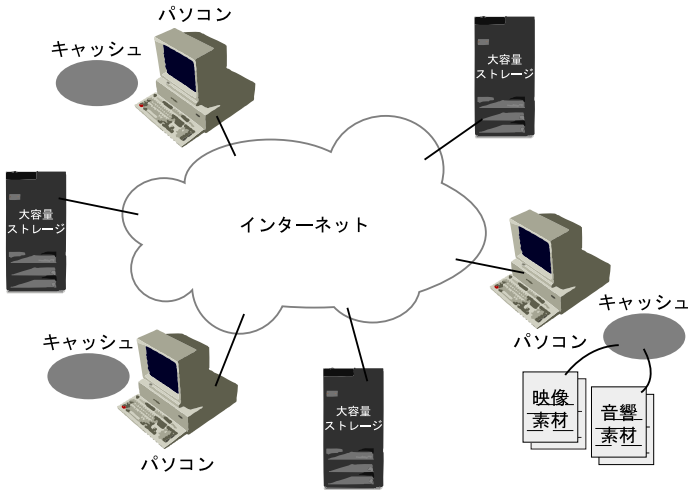


図 2.1: 遠隔映像編集システムの構成

このシステムは、素材を大量に保管する大容量ストレージマシンと、人間とのインタフェースを司るパーソナルコンピュータから成ることを前提にしているが、この役割の分担は明確ではなく、パーソナルコンピュータも、必要となる素材をキャッシュして持つことにより、大容量ストレージマシンと共に分散ストレージを形成する。分散ストレージを形成するという意味において、すべてのノードは平等であり、特別な計算機資源は必要なく、その運用のための管理者も必要としない。

2.1.2 意義

遠隔映像編集システムには次の意義がある:

1. 管理者を必要としないことから、ネットワークに関して特別な知識を持たないグループでもシステムを用いることができる。ユーザは映像や音響の制作者であるので、このことは重要である。

2. サーバ資源とその管理のためのコストが不要であることから、映像をより安価に制作できる。映像の最終的な受け手が支払うコストを下げることを可能にする意味でも、制作者の裾野を広げる意味でも、このことは重要である。

2.2 技術的背景

2.2.1 オーバレイネットワーク

全体を統御する中心を持たず、ピアグループにより分散ストレージ等の分散システムを実現するためには、グループを仮想的にネットワークとして扱い、当該ネットワークに必要な機能を持たせるという考え方が役立つ。このようなネットワークをオーバレイネットワークと呼ぶ。

分散ハッシュテーブル

オーバレイネットワークの分野で現在、盛んに研究が行なわれているものに、分散ハッシュテーブル方式がある。これは、オブジェクト (へのポインタ) をそのハッシュ値が代表するノードに格納させる、すなわち、ネットワーク全体をハッシュテーブルのように用いるというアイデアに基づいている。分散ハッシュテーブル方式では、ノードやオブジェクトの識別子を、一方向ハッシュ関数により均質に名前空間の中に分布させることにより、ノードへの負荷を拡散させつつ、かつ広域なオブジェクトの探索を効率化している。

現在までに、Chord[12], CAN[8],Tapestry[13], Pastry[9] といった具体的なシステムが提案されているが、このうち、Chord と CAN は名前空間を数値的に分割して探索するというアイデアに基づいている。Tapestry と Pastry は経路表による経路制御を行なう。

毎回、広域な探索が行なわれると効率が低下するが、Tapestry では局所化を追求しており、オブジェクトの無制限なキャッシングを許す。探索の際には、ネットワーク的に近傍のキャッシュの存在が確認できる工夫が施されている。

オーバレイネットワークによるマルチキャスト

オブジェクトの格納の負荷については分散ハッシュテーブルによる拡散が可能だとしても、固定的な指示系統を持たないピアグループでは、すべてのノードが情報を共有し、その情報に基づいて自律的な判断を行なわなければならない場面が多発する。

そこで、いわゆるアプリケーションレベルのマルチキャストを行なうことが考えられている。実際に、Tapestry では Bayeux[14] と呼ばれるマルチキャストシステムの提案と実

装が行なわれている。

2.2.2 マルチキャスト

マルチキャストを、単に放送という意味ではなく、自律的な判断のための情報の共有に用いるためには、その信頼性およびメッセージの到達順序が重要となる。

特に、この論文で後に提案する手法を理解する上で必要なことから、ここでは、[1] に則り、マルチキャストにおける信頼性と順序性を定義する。

信頼できるマルチキャスト (reliable multicast)

信頼できるマルチキャストは正当性と合意を満たす:

1. 正当性 (validity)

   もし、正常なプロセスがメッセージ m をマルチキャストするなら、グループ内の正常なプロセスはいずれ m を譲渡される。

2. 合意 (agreement)

   もし、正常なプロセスにメッセージ m が譲渡されるなら、グループ内のすべての正常なプロセスはいずれ m を譲渡される。

FIFO マルチキャスト (FIFO multicast)

FIFO (First-In, First-Out) マルチキャストでは、あるプロセスが m_1 と m_2 をこの順序でマルチキャストしたとすれば、グループ内のすべてのプロセスは m_1 より後に m_2 を譲渡される (図 2.2)。

すなわち、FIFO マルチキャストは、プロセス間の (マルチキャスト) チャネルが FIFO であることを要求している。

因果順序マルチキャスト (causal multicast)

因果順序マルチキャストでは、Lamport が [7] で定義した “happened-before” 関係 (“$\rightarrow$” により表現) を利用する:

1. 事象 a と b が同一のプロセスにあり、a が b よりも先に起こっていれば $a \rightarrow b$ とする。

2. a がメッセージの送信であり、b がそのメッセージの受信であれば、$a \rightarrow b$ とする。

3. $a \rightarrow b$ かつ $b \rightarrow c$ ならば $a \rightarrow c$ とする。

因果順序マルチキャストでは、メッセージ m_1 と m_2 の間に $m_1 \rightarrow m_2$ の関係があるならば、グループ内のすべてのプロセスは m_1 より後に m_2 を譲渡される (図 2.3)。

因果順序マルチキャストは同時に FIFO マルチキャストを満たす。

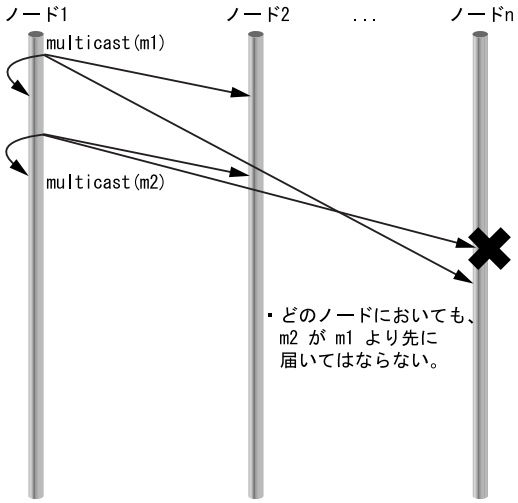


図 2.2: FIFO マルチキャスト

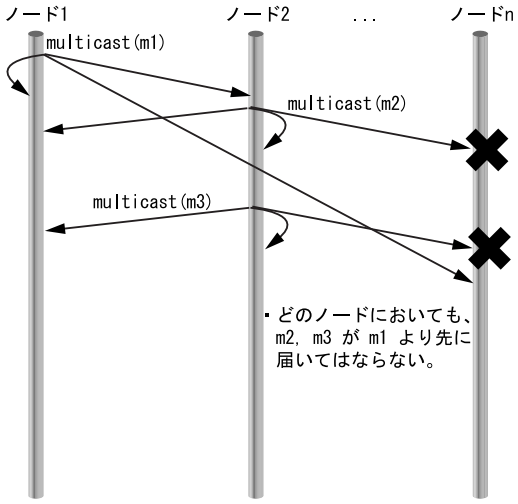


図 2.3: 因果順序マルチキャスト

全順序マルチキャスト (atomic multicast)

全順序マルチキャストでは、グループ内のあるプロセスが m_1 より後に m_2 を譲渡されるなら、グループ内のすべてのプロセスは m_1 より後に m_2 を譲渡される (図 2.4)。

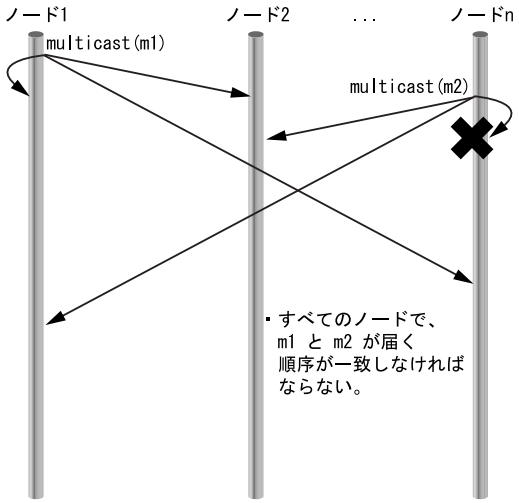


図 2.4: 全順序マルチキャスト

全順序マルチキャストでは、メッセージの到着順に因果関係を要求しないため、因果順序も同時に満たす因果+全順序マルチキャスト (causal atomic multicast) も定義されている。

メッセージの到着順序がノード間で異なると、保持している状態が共有できないことがある[1]ため、ピアグループを用いた分散システムでは全順序が重要となることが多い。

[1]例えば $x = 3$ と $x = 5$ というメッセージの到着順がノード毎に異なることを考えるとよい。

第3章　問題の定義

この章では、遠隔映像編集システムで実現されるべき、ピアグループによる分散バージョン管理を、コンピュータソフトウェア開発において現在広く用いられている SCM ツールを参考にしてモデル化する。その上で開発上の要求を整理し、バージョン付けについての問題を明らかにする。

3.1　分散バージョン管理

3.1.1　ソフトウェアの開発と分散バージョン管理

大規模なコンピュータソフトウェアの開発には数多くのプログラマが関わる。かつ、開発プロジェクトの運営コストの低減のためには、人材を長距離、移動させずに開発に参加させたいという要求がある。また近年、特に活発に行なわれているオープンソースのソフトウェア開発では、世界中の様々な場所からプログラマが自発的にプロジェクトに加わり、開発を行なっている。

これらのことから、コンピュータソフトウェアの開発では、地理的に離れたプログラマ同士の作業環境をネットワークによって結び、分散バージョン管理を実現する SCM (Software Configuration Management; ソフトウェア構成管理) ツールが発達している。

3.1.2　参考としたシステム

SCM ツールとしては、フリーでよく利用されている CVS[3] (Concurrent Versions System) や、商用製品である Perforce[11] (Perforce Software 社), BitKeeper[2] (BitMover 社) 等がある。この研究では、仕様が小さく、かつ複数のファイルのアトミック (分割不能) な更新や更新が起こった際の関連作業者への通知等、十分な機能が揃っていることから、Perforce を例にモデル化を行なった。

3.1.3　主な登場人物

SCM における分散バージョン管理のシステム構成を図 3.1 に示す。

システムで用いられる主な抽象は表 3.1 の通りである。

表 3.1: 分散バージョン管理における主な抽象とその役割

項番	抽象	役　割
1	ユーザ (user)	ワークスペースを介してデポを操作する。ユーザには次の操作が許される: 1. 同期 (デポ上のリビジョンをワークスペースにコピーする) 2. 追加 (ファイルを追加する) 追加したファイルは編集中扱いとなる。 3. 削除 (ファイルに削除マークを付ける) 4. 編集 (ファイルを変更可能にする) 5. 更新 (追加、削除、変更をデポに反映させる) 変更リスト単位で行なわれる。 6. 競合解決 (他のユーザの更新との競合を解決する)
2	管理者 (administrator)	ユーザに対しシステム利用の許可を与えたり、デポの保守を行なう。
3	デポ (depot)	バージョンの履歴を持つデータベース。
4	ワークスペース (work space)	ユーザの作業領域で、ローカルなマシン上にあり、ファイルの作成、編集や開発対象のビルドを行なう。
5	変更リスト (change list)	ユーザが変更を施したファイルの一覧と、付随するメモを含んだリスト。
6	変更番号 (change number)	変更リストをシステムで一意に識別するための番号。
7	リビジョン (revision)	更新の対象となる毎に区別されるファイルの版。
8	リビジョン番号 (revision number)	ファイル毎のリビジョンを表す番号。

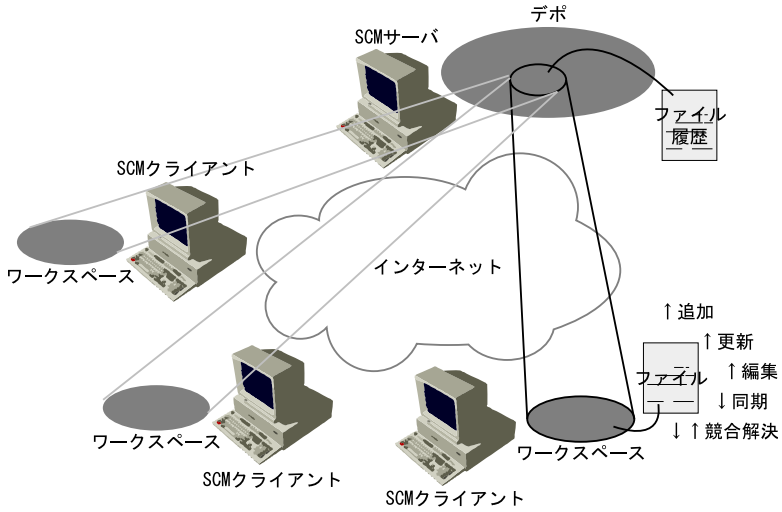


図 3.1: 分散バージョン管理のシステム構成

3.1.4 デポとワークスペース

デポ[1]は、プロジェクトに含まれるファイルすべてについて、その現在および過去のリビジョンと、対応する変更リストを保管するデータベースである。

ワークスペースはユーザが用いるマシン上のディスク領域である。サーバを用いる分散バージョン管理では、1 台のサーバで複数のプロジェクトのバージョンを管理できることから、ワークスペースにはデポの部分集合が対応していると考えることができる。

3.1.5 同期

ワークスペースにデポ上のリビジョンをコピーする操作が同期である。ファイル毎に、どのリビジョンをコピーするか、あるいはどの変更番号に対応するリビジョンをコピーするかを選択できる。

ワークスペースで編集中のファイルについてはコピーされない (コピーするためには編集を解除する必要がある)。また、競合解決 (後述) が必要な場合はその旨がユーザに通知される。

3.1.6 変更リストと更新のアトミック (分割不能) 性

ユーザがワークスペース上のファイルに対して追加、削除、編集の操作を行なうと、そのファイルの識別子と操作内容が変更リストに付け加えられる。

更新の操作を行なうと、変更リストに記載されている操作内容が実行され、デポに反映される。反映はアトミックであり、途中で通信障害等の理由で変更が打ち切られた場合は、変更リスト内のすべての操作が行なわれなかったことになる。

[1]CVS で使われるリポジトリ等の言葉の方が馴染みがある読者も多いと思うが、Perforce における用語であることと、広辞苑第五版に記載されていることから、この論文ではデポという言葉を用いる。

コンピュータソフトウェアの場合、一連の修正の一部だけを受け取ってもビルドできないことがあるし、映像制作でも、例えば動画像と音楽は同時に更新されなければならないといった制限はよくあると考えられるため、更新のアトミック性が満たされることは重要である。

3.1.7 通知

ユーザは、あらかじめデポ上のどの部分 (プロジェクトやサブプロジェクト) に対し、自分の興味があるかを登録しておく。登録した部分に対して更新が行なわれると、ユーザに対して変更内容の通知が行なわれる (Perforce の場合は標準的には電子メールを用いる)。

運用上は、ユーザは通知を受けた時、速やかに同期の操作を行ない、必要ならば競合解決を行なうことが推奨される。

3.1.8 競合の解決

次の競合条件に合致すると、ユーザにその旨が伝えられる (Perforce の場合はエラーメッセージにより伝えられる):

競合条件: ワークスペース上で編集中の (もしくは削除された) ファイルのリビジョンが、デポにある最新のリビジョンよりも古いことをシステムが検出する。

ユーザはこの時、競合解決の操作により、次のいずれかを行なうことができる:

1. デポ上の最新リビジョンを受け入れる。
2. 次の更新時に、ワークスペース上のリビジョンでデポ上の最新リビジョンを置き換える。
3. システムにより 2 つのリビジョンがマージされた内容を (編集後に) 受け入れる。

競合解決のモデルを図 3.2 に示す。

3.2 ピアグループによる分散バージョン管理

3.2.1 概要

以上に示したモデルを、遠隔映像編集システムではピアグループにより実現する必要があり、ここで新たにピアグループによる分散バージョン管理をモデル化する。

モデル化にあたっては、基本的にはサーバによる分散バージョン管理で用いられる抽象をそのまま受け継ぐ。ただし、ピアグループによる分散バージョン管理では、表 3.1に示した抽象のうち、管理者の役割をユーザが分担して担い、デポをそれぞれのマシンが持つ必要がある。

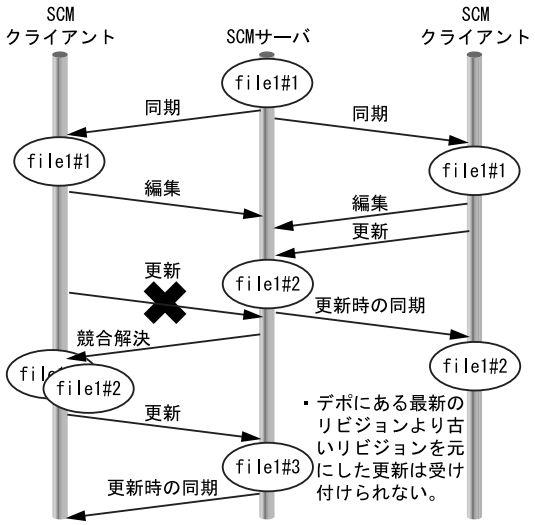


図 3.2: 分散バージョン管理における競合解決

3.2.2 モデル化に際しての議論

この際のモデル化の選択肢としては次がある:

1. 分散ハッシュテーブル方式を用い、各ピアが持つデポに格納される内容を拡散させる。すなわち、バージョン管理されるべきファイルの識別子からハッシュ値を計算し、その値に対応するノード上のデポのみにそのファイルの変更履歴を置く。

2. 各ピアが等しいデポのコピーを持つ。

負荷分散の面では直観的には前者が望ましいと考えられるが、変更リストに掲載されるファイルがすべて同一のデポで管理されないとなると、アトミック性の実現が複雑になると考えられる。そこでこの研究では後者を採用している。

また、後述するシミュレーション実験 (第 5.4節) の結果、後者の方が実際には帯域の消費が少なく、ネットワーク的には負荷が小さいことが予想できることが分かった。

3.2.3 モデル

ピアグループによる分散バージョン管理のシステム構成を図 3.3 に示す。

ピアグループによる実現では、それぞれのピアは自分が参加していないプロジェクトに関して履歴を管理する必要がないため、デポとワークスペースが (扱うファイルの範囲という意味において) 1 対 1 に対応するという特徴を持つ。

デポ間の一貫性をどのようなレベルで、どのように保証するかが、このモデルを実現する上での課題となる。

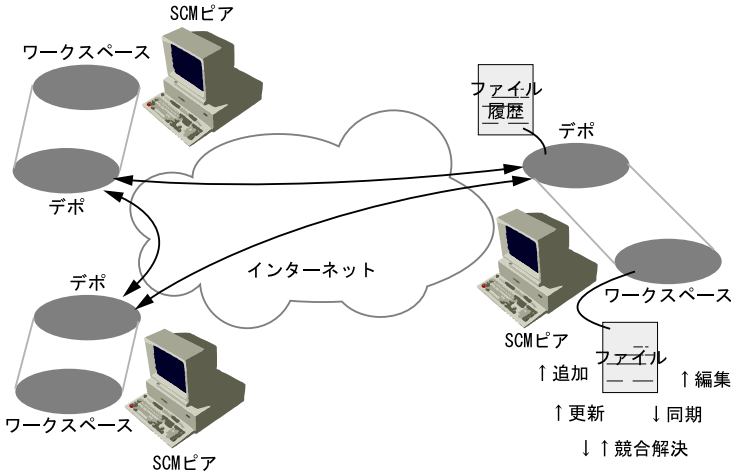


図 3.3: ピアグループによる分散バージョン管理のシステム構成

3.3 満たすべき特性

一貫性はできるだけ厳密に保証されるのがよいが、分散システムにおける一貫性の保証については通信や計算力のコストの問題がある。そこで、実用上は困らないが、できるだけ一貫性に対する要求の弱いバージョン管理を検討した結果、ピアグループによる分散バージョン管理が満たすべき特性は次ではないかとの仮説を持つに至った。

3.3.1 セイフティ特性

特性: 更新の因果順序は破れない。

理由付け

あるユーザが連続して行なった 2 つの更新のうち、後者が前者を打ち消すようなものがあるとすると、この更新順序がノードに依って入れ替わることは避けなければならない。このことから更新の FIFO が破れないという条件が必須となる。

次に、ある更新の内容を受けて、ユーザが別のファイルを修正しなければならないことがある。例えば、コンピュータソフトウェアにおいては、自分が使っているライブラリの API が変更されたような場合である[2]。この場合も、ノードに依って更新順序が入れ替わると混乱が生じるし、そうでない場合との区別を更新の意味解釈を抜きにシステムが行なうことはできないので、因果順序が守られることが望ましい。

逆に、因果順序と独立した全順序については、メッセージを整流させるシーケンサとし

[2] 実際にはこの場合も、アンドゥーを容易にしたり、ユーザがデポ内において API が矛盾している状態に対処しなければならないことを避けるためには、1 つの変更リストによりアトミックな更新を行なうことが望ましく、それに対応できるようにプログラマの組織化やルール作りを行なうのがよい。具体的には、API を変更したプログラマ自身が、旧 API を用いているすべての事例を新 API を用いるように修正するのが最もよい。

てのサーバを用いる場合でも、通信路の遅延があるため、オフラインでコミュニケーションを行なっていたとしてもユーザの意図通りにできるとは限らない。例えば電話で話しながら 2 人のユーザが順番に更新を行なっても、実時間で後に行なった操作がサーバに先に到達することがあり得るため、確実に順序付けたい場合には、サーバからの完了通知を待ち、因果順序に基づいて更新を行なう必要がある。

これらのことから因果順序が常に満たされることをシステムの要件とした。

3.3.2 ライブネス特性

特性: 更新の全順序はいずれ回復する。

ユーザの意図通りにならないとしても、全順序が満たされない場合、同じファイル識別子に対する同じリビジョン番号が示す内容が、ノードに依って異なるという状況が起き得るため望ましくない。かといって、常に全順序を保証するとなるとコスト高である。そこで、全順序が破れている可能性が検出された場合、すべてのノードで同一の内容を示すようにリビジョン番号が付け替えられることをシステムの要件とした。

3.4 導かれる問題

3.4.1 前提条件の分析

ピアグループによる分散バージョン管理において、意味のある全順序の破れが起こり得るのは、同一のファイルに対して独立して複数のノードが更新を行なった場合である。

このことは、第 3.1.8節に示した競合条件と一致しており、各ノードにとっては、競合条件が発生した際の解決のプロトコルの一環として全順序の回復が行なわれるべきだということが分かる。

3.4.2 解くべき問題

以上から、ピアグループによる分散バージョン管理におけるバージョン付けプロトコルを新たに設計する必要があるが、このプロトコルは、

1. セイフティ特性を満たさなければならないことから、因果順序に基づく必要がある。
2. ライブネス特性を満たさなければならないことから、競合条件が起こった際にそれを検出し、回復を行なう必要がある。故障が起きても、そこから回復することによりシステムを維持できることを耐故障と呼ぶように、このことは耐競合と言い替えることができる。

すなわち、この研究において解くべき問題とは、因果順序に基づく耐競合バージョン付けプロトコルを設計することである。

第4章　プロトコル設計

この章では、本研究における中核となるアイデアである、因果順序に基づく耐競合バージョン付けプロトコルを示す。

4.1　基本的なアイデア

4.1.1　局所的情報に基づく自律的なバージョン付け

プロトコルの設計に当たっては、余分なメッセージを用いず、各ノードが自律的な判断に基づいてバージョン付けを行なうことにより帯域を節約することが望ましい。

このプロトコルが満たすべき順序付けとして、因果順序 (常に) と全順序 (破れてから回復してよい) がある。このうち、因果順序については、メッセージに全ノードの論理時計 [7] を載せることにより、アプリケーションに譲渡する契機を受信側で自律的に判断できるプロトコルが知られている [10]。

全順序については、シーケンサとなるノードを用いない限りこのようなことはできない。シーケンサとなるノードを導入すると、ピアグループによる分散バージョン管理のモデルが崩れるため、それは避けるとすると、何らかの形で条件を緩めることにより、自律的な判断を可能にしなければならない。

この研究では、全順序は常に満たしている必要はなく、順序の破れが検出されてから回復すればよいので、受け取った 2 つの更新を決定的アルゴリズムにより比較することで一意に順序付けを行なうという手法を採用できる。

4.2　因果順序に基づく耐競合バージョン付けプロトコル

4.2.1　プロトコルの概要

提案するプロトコルの概要は次の通りである:

1. 各ノードは、自ノードにおいて更新が行なわれた際、変更リストを信頼できるマルチキャストにより送信する。その際、メッセージにはベクタ時計 [10] を含める。
2. メッセージを受け取ったノードは、ベクタ時計による因果順序の判定アルゴリズムに則って、自ノードのデポへの譲渡が可能なメッセージを判定する。
3. 因果順序を満たすメッセージがある場合、それが競合条件を満たしていなければ、すぐにデポに譲渡する。

4. 競合条件を満たしている場合、決定的なアルゴリズムにより、問題となっている 2 つのリビジョンのうち採用するリビジョンを選択する。採用されなかったリビジョンは、当該リビジョンの枝リビジョンに位置付けられる。採用されないリビジョンが複数ある場合は、枝リビジョンの間で更に決定的なアルゴリズムによる順序づけを行なう。

5. 採用されなかったリビジョンを元に編集を行なっているワークスペースを持つノードでは、ユーザへの通知を行ない、競合解決を促す。

6. 一連の解決が終了すると、リビジョンの示す内容は全ノードで一致している。

4.2.2 因果順序の保証

ベクタ時計による因果順序保証プロトコル

ベクタ時計により因果順序を保証するプロトコルは Schiper[10] らにより発案され、Birman らの ISIS Toolkit[1] でも採用されている。

基本的なアイデアは次の通りである:

1. ノード毎に、メッセージの送受信等のイベントの発生時に加算されるカウンタ (論理時計) を持ち、その初期値について同意しているものとする (例えば 0)。

2. ノードは、自分以外のノードについても、自分が知る限りの論理時計の値を保持している (初期値は与えられているので、少なくとも初期値を示している)。

3. メッセージをマルチキャストする際は、自ノードの論理時計と、自分が知る限りのその他のノードの論理時計の値をメッセージに添付して送付する。

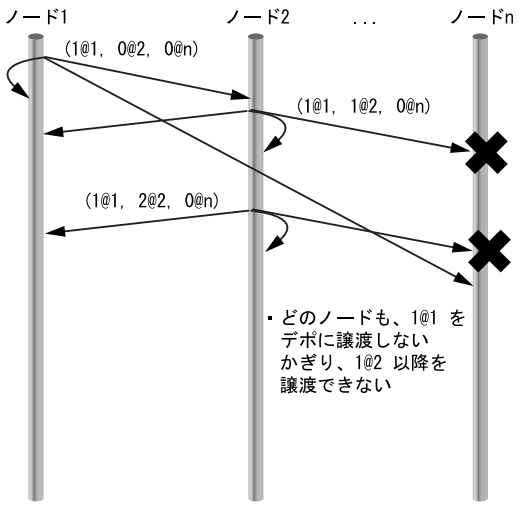
4. メッセージを受け取ったノードは、自分が知る、送り元の論理時計の値よりもメッセージに添付された値の方が 1 つだけ大きい場合、自分が知る送り元の論理時計の値を更新する (そうでない場合、キューにメッセージを保管する)[1]。

   その他の論理時計については、もしメッセージに添付された値が、自分が知る値よりも大きい場合には、因果的に先行する、自分の知らないメッセージが存在することを示しているので、そのメッセージが到着するまで、受け取ったメッセージをアプリケーションに譲渡するのを待つ。

分散バージョン管理への適用

ピアグループによる分散バージョン管理では、マルチキャストは更新時のみ行なうようにできる。そして、サーバによる分散バージョン管理のモデルではシステムで一意だった変更番号を、ノード毎にローカルな値として持ち、変更番号を論理時計として用いることでこのプロトコルを利用できる (図 4.1)。

[1] 対応するリンクの FIFO が保証されている場合はこの判定は省略できる。



c@n という表記は、<c = 変更番号, n = ノード > の組による論理時刻を表す。

図 4.1: ベクタ時計による因果順序の保証

4.2.3 耐競合性の実現による全順序の緩やかな回復

競合の検出と対処

因果順序が満たされ、デポに譲渡されたメッセージは、以下のマルチデポにおける競合条件を満たすかどうかを検査される:

マルチデポにおける競合条件:

デポにある最新のリビジョン $\geq$ メッセージのファイルのリビジョン

競合条件を満たす場合、メッセージのファイルのリビジョンと、その番号に対応する、デポにおけるリビジョンとの間で、決定的アルゴリズムによりどちらを採用するかを判定する。(図 4.2)。

無効となったリビジョンは、当該リビジョンの枝リビジョンに割り当てられる。無効となったリビジョンが複数ある場合は、枝リビジョンの間で更に決定的なアルゴリズムによる順序づけが行なわれる。

無効となったリビジョンを元にして編集中のファイルがワークスペースにある場合、ユーザに通知して競合解決を促す。

更新の順序を決定するアルゴリズム

1. 対象となるファイルの内容が一致する場合、両者は区別しない。
2. 対象となるファイルの指紋を MD5 等の一方向ハッシュ関数 (システム毎に一意) により算出し、値の大きい方を採用する。

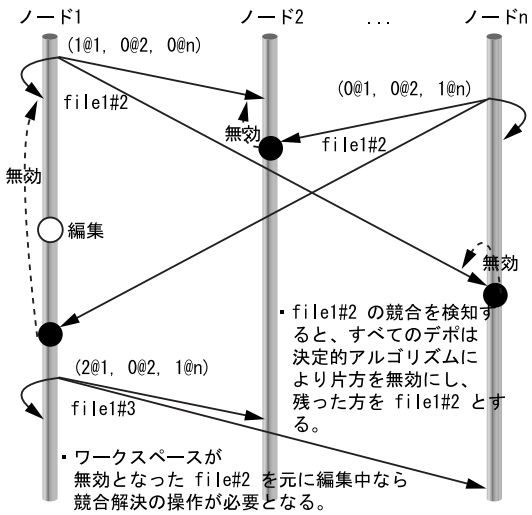


図 4.2: 全順序の回復

3. 稀に、ハッシュ値が一致する場合は、その旨を記録するマークを付け、参照される際にはユーザに警告する。

4.2.4 考察

このプロトコルは、因果順序の保証においては変更番号を論理時計として用い、全順序の回復においては競合解決を用いるというように、分散バージョン管理の特徴を活かし、それと融合する形で順序づけを行なっている。特別な機構を後から用意していないことから、プロトコルのオーバヘッドが小さく、効率的であることが予想できる。

第5章　検証と評価

この章では、設計したプロトコルが満たすべき特性をインフォーマルな数理的手法により検証する。また、シミュレーションによる実験とその結果および評価を示す。

5.1　特性の検証

5.1.1　セイフティの検証

セイフティ特性、

特性: 更新の因果順序は破れない。

は、ベクタ時計のプロトコルにより満たされることが証明されている。
インフォーマルには次のように証明できる:

1. プロトコルにより、各ノードにおける論理時刻はマルチキャストの送信毎に加算される。

2. 信頼できるマルチキャストであるので、送信されたメッセージは必ず届き、いずれすべてのノードに届く。同一のノード x から m_1, m_2 の順序でマルチキャストした場合、x の論理時刻が最初に t だったとすると、m_1 の送信時に時刻は $t+1$, m_2 の送信時には $t+2$ となる。受け手 y はプロトコルにより必ずこの順序でデポに譲渡することが保証される。従って、FIFO マルチキャストが成立する。

3. プロトコルにより、マルチキャストによるメッセージを受信した際にしか、他ノードを示す論理時刻は更新しない。m_2 を受信した時、受信側 y での x の論理時刻は $t+2$ となる。

4. これ以降、y がマルチキャストを行なう場合、ノード z がそのメッセージを受信した後、デポに譲渡するためには、プロトコルは z における x の論理時刻が $t+2$ 以上であることを要求する。FIFO が成立しているので、このことを満たすためには、z が m_1, m_2 を受信している必要がある。

5. この関係は因果順序の定義そのものである。

5.1.2　ライブネスの検証

ライブネス特性、

特性: 更新の全順序はいずれ回復する。

は、次のようにインフォーマルに証明される:

1. システムを構成するピアグループの任意の部分集合が、等しいリビジョン番号を持つファイルを編集しており、同時に更新したとする。

2. プロトコルにより、これらのファイルを含む内容が、ピアグループの部分集合に含まれる各々のノードからマルチキャストされる。変更は因果的に互いに無関係であるため、順不同で各々のデポに譲渡されるが、信頼できるマルチキャストであるので、いずれ、すべての変更がすべてのデポに譲渡される。

3. 各々のデポでは、2つずつファイルの指紋の比較を行なうが、計算される指紋の値は一方向ハッシュ関数の性質上、不変であるので、次の場合を除いて、ファイル群の最終的な順位付けは一意に決定される:

   (a) ファイルの内容が一致する場合
   この場合は区別できないので、順序付けに影響しない。

   (b) 稀にハッシュ値が一致した場合
   システムの障害の可能性より低い確率で発生すると考えられるため、事故として扱うことができる。

4. 従って、すべての変更がすべてのデポに譲渡された時点で、全順序が成立する。

5.2 シミュレーションシステム

5.2.1 概要

第 4章で設計したプロトコルは未実装であるが、次の目的で、シミュレーションシステムを Java のマルチスレッドプログラムとして実装した:

1. 提案したプロトコルに誤りがないかをソフトウェアの実行により検証する。

2. 提案したプロトコルがソフトウェアにより実装可能であることを示す。

3. 提案したプロトコルの挙動をシミュレーション環境にて測定する。

5.2.2 実装

次の 2つのプログラムを作成した:

1. この論文で提案するプロトコルに基づき、ピアグループによるバージョン付けをシミュレートするプログラム

2. サーバによるバージョン付けをシミュレートするプログラム[1]

[1] 分散ハッシュテーブルを用いたバージョン付けとも等価である。

各々のプログラムはノード数を引数として受け取り、ある 1 つのファイルのリビジョンの推移について、予め決められた条件でシミュレートし、ネットワークを流れたメッセージ数を出力する。ピアグループによるバージョン付けの場合は、ベクタ時計の比較回数も出力する。

5.3 実験

次の条件でシミュレーション実験を行なった:

1. ノード数が 1,5,10,50,100,125,250,500 の場合について計測する。
2. 50% のノードがランダムな間隔で編集と更新 (必要な場合は競合解決も) を行ない、残りの 50% が通知を受けて同期する。
3. 各ノードは、10 回の更新操作をローカルに検出すると停止する。

実際には、例えば 500 人が参加する開発/制作プロジェクトで、250 人がある特定のファイルに対して更新する権限を持つということはまずない。上の条件は過剰な負荷をシステムに与えるためのものである。

5.4 結果と評価

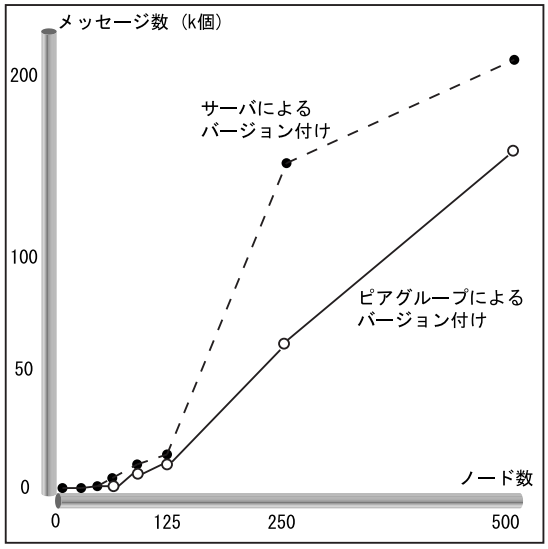
5.4.1 メッセージ数

測定結果

シミュレートされたメッセージ数を図 5.1に示す。
サーバによる場合と比較して、ピアグループによるバージョン付けの方が、ネットワークを流れるメッセージ数は少ないという結果となった。

考察

この結果は、この研究で提案するプロトコルが局所性に優れていることを示していると考えられる。
ピアグループによる分散バージョン管理では、更新時以外にはメッセージはネットワークを流れない。
一方、サーバによる分散バージョン管理、あるいは分散ハッシュテーブルを用いた分散バージョン管理では、更新だけでなく、同期や編集などの操作もすべてネットワークを介する。逐次、サーバへの問い合わせを行なうので、ネットワークの帯域の消費が大きくなると考えられる。
勿論、メッセージの数だけではなく、そのサイズも考慮しなければならないが、次に示すように、メッセージのサイズについては両者で大きな差がないことが分かる:



ただし、10 回の更新操作をローカルに検出して全ノードが停止するまでを計測。

図 5.1: ネットワークを流れるメッセージ数 (シミュレーション)

1. ピアグループによる場合、更新時にファイルの内容がマルチキャストされる。サーバによる場合、同じ内容が同期時に転送される。同期は全ノードが行なうので、結果は等しくなる。また、同期は各ノードが更新の通知を受けた際に自律的に行なうのでマルチキャストにできない。マルチキャストの実装に依っては帯域の節約が期待できるが、サーバによる場合にはそれが適用されない。

2. ピアグループによる場合、ノード数の増加に伴い、メッセージに添付されるベクタ時計のサイズが増大する。

   ベクタ時計のサイズは次のように求まる:

   (a) システムに参加しているノードの数が決まっており、その間に一意な順序付けを行なえるなら、論理時計が 4 バイトの場合 500 ノードで 約 2KB となる。

   (b) ノードが動的に求まる場合は、128 ビット (16 バイト) の識別子を用いる場合、論理時計が 4 バイトならば 500 ノードで約 10KB となる。

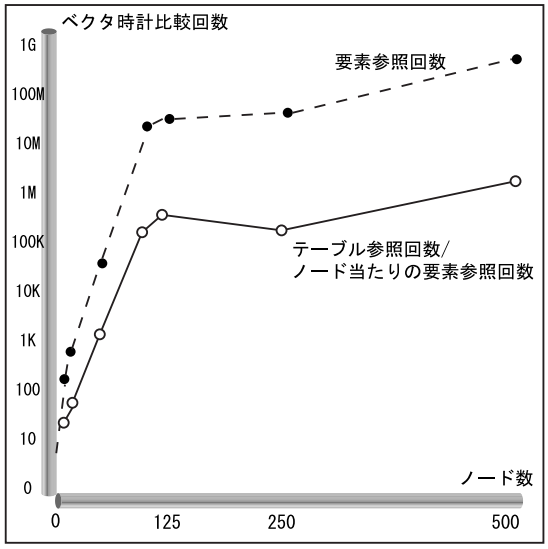
   例えばこの論文の LaTeX 2_ε ファイルのサイズが約 45KB だということを考えると、更新時に変更リスト上のファイルの内容と共に送信されるだけなので、大きなオーバヘッドにはならないと考えられる。

規模透過性としては、ノード数が数十個あたりまでなら実現できていると言える。

5.4.2 ベクタ時計の比較回数

測定結果

シミュレートされたベクタ時計の比較回数を図 5.2に示す。



ただし、10 回の更新操作をローカルに検出して全ノードが停止するまでを計測。

図 5.2: ベクタ時計の比較回数 (シミュレーション)

システム全体としては要素参照回数で示されるだけの整数の比較が行なわれる。ノード当たりはテーブル参照回数で示されるだけの整数の比較が行なわれる。

考察

ノード数の増加に伴う比較回数の増加が大きいことが分かるが、計算量としては要素当たり数命令であり、無視できる程度であるのと、ノードに閉じた資源の消費であるので、系全体に及ぼす影響は微々たるものだと考えられる。

また、帯域的に十分に規模透過性が実現できる範囲がそもそも数十ノード程度であり、その範囲ならば全く問題にならないと考えられる。

第6章　関連研究

この章では、ピアグループオーバレイネットワーク上の分散ストレージにおけるオブジェクトのバージョン付けに関連する既知の研究を紹介し、この研究との比較を行なう。

6.1　OceanStore

6.1.1　概要

OceanStore[6] は Tapestry オーバレイネットワークを用いた広域分散ストレージユーティリティである。OceanStore は、持続する情報への継続したアクセスを提供することを目指してカリフォルニア大学バークレィ校にて開発が進められており、耐故障性、可塑性を重要視している。

バージョン管理に関連しては、Deep Archival Storage と呼ばれるものが検討されている。これは、過去のすべてのバージョンを erasure codes[1] により符号化し、数百、数千のノードに拡散して格納し、そのうち小数の断片のみから復元を可能にするというものである。これにより、全地球が壊滅的な打撃を受けない限り、アーカイブ化されたデータを保証できるという。

6.1.2　本研究との比較

[6] はバージョン管理に言及しているが、具体的な更新アルゴリズムの記述はない。

Tapestry オーバレイネットワークは分散ハッシュテーブル方式に基づくが、第 5.2節で明らかにしたように、分散ハッシュテーブルによってオブジェクト毎にバージョンを管理するルートノードを置くよりも、この研究で提案するモデルを用いた方が帯域の使用効率の面では望ましい。また、更新のアトミック性も分散ハッシュテーブル方式では実現しにくい。

6.2　PAST

6.2.1　概要

PAST[5] は Pastry オーバレイネットワークに基づく不揮発性分散ストレージであり、バージョンの概念を持たない。

[1] 例として Read-Solomon codes や Tornado codes がある。データを n 個の断片として扱い、更に多くの断片に変換し、そのうちどの n 個 (程度) の断片によっても元のデータを再現できるという符号化アルゴリズム。

6.2.2 本研究との比較

バージョンの概念を持たないことにより、分散システムにおけるバージョン付けの困難さをそもそも回避しているが、ダイナミックなコラボレーションに使えないという問題点がある。

6.3 みかん

6.3.1 概要

みかん [15] (み んなで かん たんファイル共有) は慶應義塾大学 大学院で開発されたファイル共有ツールであり、SMTP サーバによりメッセージを整流するというアプローチをとっている。

6.3.2 本研究との比較

みかんでは SMTP サーバをシーケンサとして用いることにより全順序を実現しているが、SMTP サーバはファイル共有に参加するノードとは限らず、通常は別に用意されているので厳密な意味でピアグループとは呼べない。

みかんはデポとワークスペースに対応するディスク空間の抽象を行なっており、競合の検出自体は行なうが、明確な解決モデルを持たない。映像作品の制作のような厳格なバージョン制御を要求するアプリケーションに応用するためには未だ工夫が必要である。

ただし、実効的な手軽さは評価できる。

第7章　結論

この章では、まとめを行ない、問題点を明らかにし、今後の課題を示す。

7.1　まとめ

7.1.1　本研究の成果

1. ピアグループによる分散バージョン管理のモデル化を行ない、
2. 当該モデルに沿ったバージョン付けの問題を、因果順序の保証と緩やかな全順序回復により解決し、
3. かつ、サーバによる分散バージョン管理と比較して、帯域の使用効率と計算量の面で性能の劣化を防ぐことができた。

7.2　今後の課題

7.2.1　規模透過性の実現

現状では、ノード数が数百のオーダーになると、帯域の使用効率および計算量の面で、サーバによるバージョン管理ほどではないが、著しい性能低下が起こることが予想される。

これについては、管理単位の分割等を用いることにより、バージョン管理との統合的なアプローチにより解決を図れる見込みがあり、更に検討を行なう。

7.2.2　実装

アルゴリズムの検証に終わっているので、実際に利用できるソフトウェアを実装したい。その上で、第 3.3節では仮説に留まっている要件について検証しなければならない。

この研究が提案するプロトコルでは、競合解決が完了するまで、更新履歴の全順序が失われる場合があるため、あるファイルのあるリビジョン番号が示す内容が、ある期間、ノード間で一致しないことがあり得る。このことが実用上、問題となるのか、問題となるとすればどの程度、重大かについて調査したい。

7.2.3 分散バージョン管理の設計

バージョン付けの問題を解決したのみであるので、ユーザインタフェースや API を持つ統合的な分散バージョン管理、ひいては分散ストレージシステムの設計を行ない、実装し評価したい。

謝辞

本研究を進めるにあたり，ご指導頂きました慶應義塾大学環境情報学部教授の村井純博士，徳田英幸博士，同学部助教授 楠本博之博士，中村修博士, 同学部専任講師 南政樹氏に感謝します。また、どんなときも見捨てずにご指導くださいました斉藤賢爾さんに熱く強く感謝します。本当にありがとうございました。

関連図書

[1] Ken Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, Vol. Vol.9, No. No.3, August 1991.

[2] BitMover, Inc. BitKeeper – the scalable distributed software configuration management system, 1997-2002. Hypertext document. Available electronically at http://www.bitkeeper.com/.

[3] CollabNet, Inc. Concurrent Versions System – the open standard for version control, 1999-2002. Hypertext document. Available electronically at http://www.cvshome.org/.

[4] Digital Cinema Consortium. Digital Cinema Consortium. Hypertext document. Available electronically at http://dcc.imgl.sfc.keio.ac.jp/.

[5] Peter Druschel and Antony Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings HotOS VIII, Schloss Elmau, Germany*, May 2001.

[6] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gunnadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

[7] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, Vol. Vol.21, No. No.7, July 1978.

[8] Sylvia Ratnasamy, Paul Francis, Richard Karp Mark Handley, and Scott Shenker. A scalable content-addressable network. In *Proceedings ACM SIGCOMM, San Diego, CA*, August 2001.

[9] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.

[10] A. Schiper, J. Eggli, and A. Sandoz. A new algorithm to implement causal ordering. In *Proceedings of the 3rd International Workshop on Distributed Algorithms, Lecture Notes on Computer Science 392*, 1989.

[11] Perforce Software. Perforce Software – the fast software configuration management system, 1996, 2003. Hypertext document. Available electronically at http://www.perforce.com/.

[12] Ion Stoica, Robert Morris, M. Frans Kaashoek David Karger, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*, August 2001.

[13] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB//CSD-01-1141, U.C.Berkeley, April 2000.

[14] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the Eleventh International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2001)*, June 2001.

[15] 西村祐貴. SMTP を利用したファイル共有システムに関する研究. Master's thesis, 慶應義塾大学 大学院 政策・メディア研究科, March 2003.