Unstructured P2P 環境に適したデータ管理方式 WOODS を利用したアプリケーションの設計

安田 豊

京都産業大学 理学部

yasuda@cc.kyoto-su.ac.jp

WOODS (Write Once Oriented Data management Structure) はデータのアクセスを追記に基づく Write Once なものに制限することで、更新に伴う一貫性管理とオーバヘッドの問題を解決するデータ管理方式である。本論文では、WOODS を用いたアプリケーションの具体的な設計について示し、WOODS の特性を活かしたまま、P2P 環境に適した実用的なアプリケーションが設計できることを示す。また、処理に必要なロックの範囲が極めて限定的であることを示し、Unstructured P2P 環境に対する WOODS の適性を確認する。

Application design for WOODS : data storage structure of Unstructured P2P

Yutaka Yasuda

Kyoto Sangyo University, Faculty of Science

yasuda@cc.kyoto-su.ac.jp

WOODS, Write Once Oriented Data management Structure solves the problem of complexity and overhead to keep consistency to make a restriction for data accessing only reading and append-type update. This paper shows a design of practical application based on WOODS and comes out how it can keep the benefit of WOODS. And paper also analyze the minimality of lock range to append the data then shows the aptitude of WOODS for Unstructured P2P.

1 はじめに

近年の計算機システムの低価格・高性能化とネットワークの普及に伴い、Peer to Peer (P2P) 技術を用いたネットワークサービスが注目されている。

P2P 技術を応用した分散ストレージシステムの研究もその一つである ([1] [2] [3]) が、それらは以下のような特徴をもつ。

Read Only ではなく、更新可能なファイルシステムの機能を提供するものはそれに伴う一貫性維持のために実装が複雑になり、そのオーバヘッドが問題となる。多くは Structured P2P システムとして実現されているが、これはサイトの参加・離脱に対応するためのオーバヘッドを追加する。より簡易な運用が期待できる Unstructured P2P システムでは逆に一貫性維持がより難しい課題となる。

そこで筆者は、アプリケーションによっては通常のファイルシステムが提供するのと同等のデータ更新機能を必ずしも要求しない点に注目し、データのアクセスを Write Once Read Many (WORM) なものに制限することで、更新に伴う一貫性管理とオーバヘッドの問題を解決する新たなデータ管理方式、WOODS を提案した [4]。

本論文ではこの研究をさらに進めて、現実的なアプリケーションの設計を行う。それによって必要な API の抽出が可能になるとともに、Unstructured P2P 環境で WOODS がもつ更新処理に伴うオーバヘッドの低さが効果的に働くことを検証できるからである。

以下、まず 2 章で WOODS のデータ管理手法について説明する。次に 3 章で設計するアプリケーションの概要と具体的な動作を示し、4 章でアプリケーションの特性と P2P 環境への適性について議論し、5 章で本論文をまとめる。

2 WOODS

2.1 データ構造

WOODS はそのデータを図 1 に示すようなデータ小片のツリーとして管理する。

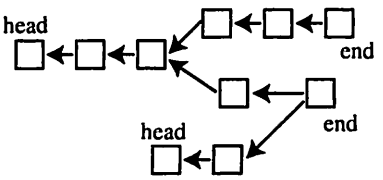

図 1: データの構造

データはすべて図 2 に示すように、id、ボディ、そしてリンクからなる。これをレコードと呼ぶ。

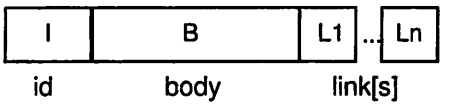

図 2: レコードの構造

レコードは Content Hash 方式で得る id によって特定される。リンクはなくても、複数あってもよい。

$$I = hash(B + L1 + ... + Ln)$$

すなわちレコードはリンクを持たないレコード (head record) から始まり、そこに追記を重ねていき、最後尾のレコード (end record) に至る。本稿ではこうしたレコードの連鎖をチェインと呼ぶ。

チェインは分岐・収束を含むが、そのアクセス順はトポロジカル・ソートによって決定する。ソートしてもなお順位が同位になる範囲のアクセス順はアプリケーションに任せられる。メイル・アプリケーションであれば受付時刻など、順位を決定できる情報をデータに含めておけばよい。

WOODS が想定する典型的なアプリケーションはこうしたチェインを複数管理する。複数のチェインを識別するためにリンクには名前をつけるものとする。すなわちリンクは名前と id からなる。

2.2 システムモデル

ピアとなるノードには一つの WOODS に基づくデータ管理システムがあり、それを利用する複数のアプリケーションが稼働している (図 3)。

WOODS はレコードを保管し、アプリケーションに対する API と、要求に応じて指定された id のデータをネットワークごしに提供する機能をもつ。このノードを複数結んで WOODS ネットワークを形成する。

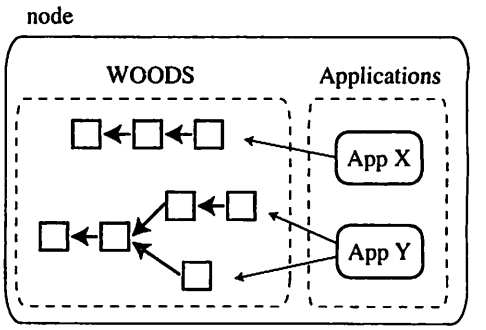

図 3: 典型的なシステム

ネットワークは Unstructured なモデルをとる。本稿では WOODS ネットワークの実現手法については言及しない[1]。

レコードの参照においては、アプリケーションは自ノードのストレージにレコードが存在しておればそれを読み、存在しない場合は WOODS ネットワークからレコードを取得する。

レコードの書き込みすなわち作成は、常に自ノードのストレージに行われる。書き込むアプリケーションが動作しているノードが WOODS ネットワークに参加している時でも、オフラインの時でも、自由にレコードを作成してよい。後にネットワークに参加した時に全てのレコードをマージしても矛盾は生じない。

データの書き換えは Content Hash による id の変更を意味するため、レコードは作成のみが可能であり、書き換えできない。削除は自由に行えるが、その有効範囲はノードローカルであり、すべてのノードで特定のレコードを無効とするような方法は用意されない。

各アプリケーションはチェインごとに最新の end record を保持する。図 3 で App X, Y から伸びるリンクはこの end record 情報を意味している。逆にこれが失われると、アプリケーションはデータに再びアクセスできなくなる。なお end record は複数存在し得るため、一般にアプリケーションは複数の end record を管理できなければならない。

3 設計

3.1 設計するシステム

本研究では P2P 的な協調動作が可能な Blog (Weblog) アプリケーションを設計する。つまり全てのメッセージは共有ストレージに格納し、各 Blog サイトごとに独立したアプリケーションによって管理される。こ

[1]必要な機能は指定された id のデータをネットワークごしに参照させる機能のみであり、Unstructured P2P データ管理システムの多くが適用可能である。

うしたシステムを WOODS によって構築する理由および特徴は以下の通りである。

- Blog システムは多くが集中型アプリケーションとして実装されるが本質的にはトラックバックなどに現れているように、メッセージを介してシステムとシステムが疎に接続されたものと考えられ、Unstructured P2P に適している。
- メッセージング・アプリケーションは基本的に WORM 指向であるため WOODS に適している。
- Blog や SNS (Social Networking Service) に個人がメッセージを蓄積しつつあるが、サービスのダウンや終了と共にそれらにアクセスできなくなるという問題がある。WOODS には無制限のキャッシュやネットワーク分割・再結合への耐性といった特性があり、この問題を解決できる。つまり何かのサービスが停止しても、共有ストレージに蓄積されたメッセージに継続してアクセス可能である。

次節以降に具体的なデータ構造を示し、例とともに処理方法の詳細を述べる。その過程で WOODS に対して行われる更新、すなわちレコードの追加処理の Unstructured P2P 環境への適合性について検討する。

なお説明では各ノードに存在する Blog アプリケーションは一つだけであり、登録ユーザは一人と仮定しているが、同一ノードに複数アプリあるいは複数ユーザが同居したとしても設計上は何も変わらない。

3.2 単純な追加：記事の投稿

提案する Blog システムは図 4 のように複数のアプリケーション (App A〜X) が協調して動作するものである。各アプリケーションは自ユーザの記事をそれぞれ別のチェインとして管理する。

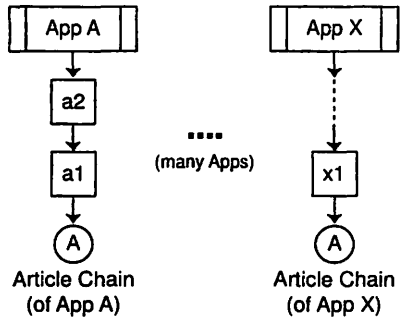

図 4: 記事とアプリケーション

記事の投稿は投稿順に記事をチェインへ追加することで実現される。例えば App A はまずはじめに記事チェイン (Article Chain) 先頭の目印となる内容が空のレコード (図中の丸印に A) を作成し、以後はそれにつなげる形で記事 a1, a2 を登録している[2]。

この状態から記事 a3 を追加する (図 5)。

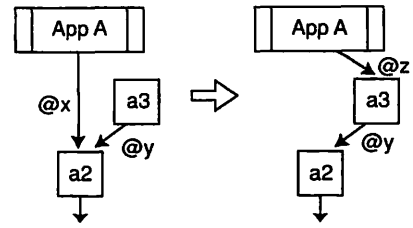

図 5: (左) 投稿記事の生成 (右)end point の付け替え

レコードの追加は下のような手順で行われる。

step 1 App A は追加すべきチェインの end point (a2) を調べ、

step 2 そこへのリンク (@y) を含む投稿記事レコード (a3) を生成し、(図 5(左))

step 3 これをチェインへ追加する (図 5(右))

App A にとって a3 をチェインに追加するとは、自身の end point を a2 から a3 に更新することを意味する。これは App A がリンクを @x から @z に付け替えたと考えてもよい。

この状況ではデータの更新者が一つしか存在しない (single writer) ため、問題がおきないのは当然である。更新者が複数存在する場合 (multiple writers) について、次節以降でコメント処理を例に説明する。

3.3 コメントの記入

Blog では記事にコメントが付けられることがある。提案システムではコメントは異なるノード上のアプリケーションから記入される。図 6 は App A の記事 a1 に対して App B がコメント b#1 を付けた場合である。コメントは記事とは別のコメント用チェイン (Comment Chain) に連結して管理し、記事チェインとはリンク名によって区別する。

コメントの追加は以下のような手順で行われる。

step 1 App B は App A に対してコメント・チェインの end point を問い、

step 2 これと対象記事 (a1) へのリンクを共に含むレコード (b#1) を生成し、

step 3 App A に対してコメント・チェインへの追加要求を行う

[2]チェイン先頭の目印は単に図示する際の判りやすさのために用いた。実装では単に head record をリンク無しの投稿記事とすればよい。

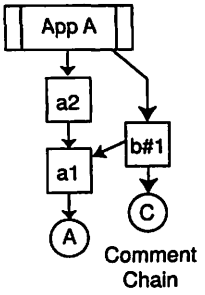

図 6: コメント

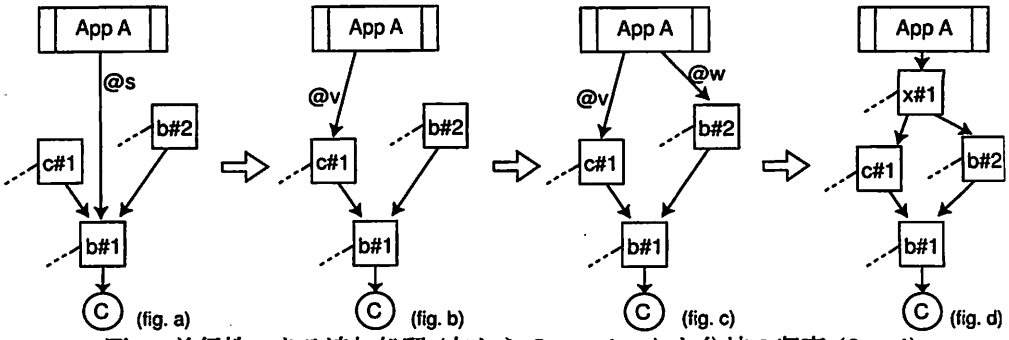

図 7: 並行性のある追加処理 (左から fig. a, b, c) と分岐の収束 (fig. d)

3.4 並行性のあるコメント処理

次にコメント b#1 のみの状態 (図 6) から、同時に App B, C が App A の任意の記事に対してコメントを記入した場合の手順について説明する。

step 1 App B, C が同時に App A にコメント・チェインの end point を問い、同じ値 (b#1) を得る

step 2 取得した end ponit (b#1) と対象記事へのリンクを含むレコード (c#1, b#2) を App B, C がそれぞれ独立に生成し (図 7a)、

step 3 App A に対して App B, C が同時にコメント・チェインへの追加要求を行う

結果、偶発的に App C の要求が先に受理され、end point は @s から @v に変更されたとする (図 7b)。その後 App B の要求に従って App A は b#2 をチェインに加えようとするが、@v を捨てて b#2 に end point を付け替えることはできない。c#1 がチェインから外れてしまうためである。このような場合 App A は @v を捨てず、単純に @w を加えることによって b#2 をチェインに取り込めばよい (図 7c)。

つまり、こうした並行性を含む場合でも、App A は他の複数プロセスから送られる step 1, 3 の処理要求を単に到着順に処理するだけでよく、step 1〜3 を通したロック等は不要である。特に step 1 は更新処理を含まないため並行処理が可能である。step 3 の性質については 4.2 で詳しく論じる。

なおアプリケーションのレコード探索順はトポロジカル・ソートによって決定されるため、図 7c で示したリンクの分岐 (@v, @w) はアプリケーションの実行に矛盾を生じさせない。

また図 7d の x#1 の追加が示すように、これらの分岐はその後のコメントの追加によって自然に収束するため、アプリケーションにとって無限に増大する負担とはならない。

3.5 全体像

前節までで記事とコメントについて説明したが、提案システムにはそれ以外にコメント控えとトラックバックの二つのチェインがある。本節ではそれらについて説明し、システムの全体像 (図 9) を示す。

コメントは図 8 のように対象となった元記事を保持するアプリケーション App A のコメント・チェイン (点線) の一部であるとともに、コメントをつけた側である App B のコメント控えチェイン (二重線) の一部にもなる。

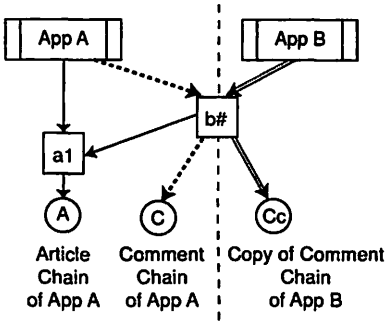

図 8: 二つのチェインに属するコメント

このようにすることでコメントを付けられた側だけでなく、付けた側もその記録をたどることができる[3]。

他の記事の引用に際して引用元が参照可能になるよう、記事には参照リンクを含めることができる。

トラックバックは図 9 のように引用記事、被引用記事双方へのリンクをもつポインタとして実現し、記事等と同様に専用のチェイン (一点鎖線) で管理する。

処理手順としては、App B が App A の記事を引用する場合、まず引用元を示すリンク (波線) を含めた記事を作成し、自己の記事チェインに追加する。次に記事と引用元の双方をリンクに含むトラックバック・レコー

[3]提案システムは P2P 環境でのアプリケーション間連携を目的としているため、こうしたアプリケーション相互の関連づけを実現することは重要である。例えばユーザ間の関連づけが Blog より強い傾向のある SNS には同様の機能が見られる。

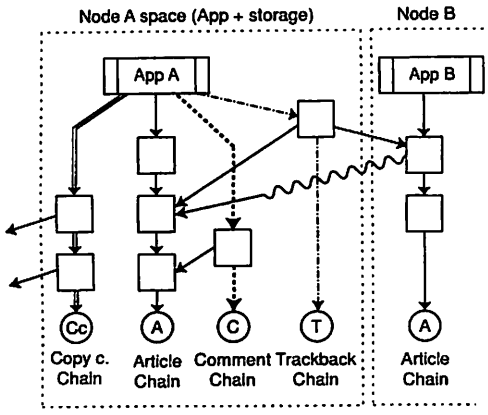

図 9: 全体構造

ドを作成し、これを App A のトラックバック・チェインに追加するよう依頼する。

図 9 にある破線の囲みは、各ノードがレコードを主体的に管理する範囲を示している。これはデータの格納位置と必ずしも一致しない。例えばコメント・レコードは二つのノードの異なるチェインに属しているため、両ノードのストレージに格納されているだろう。

4 考察

4.1 end point への追加アルゴリズム

本稿では前節 3.4 までは直感的な理解を助けるためにリンクを「付け替える」ものとして説明した。ここで追加されるレコードがもつリンク先やアプリケーションの保持する end point が複数であることを意識したうえで、チェインへのレコード追加を実現するアルゴリズムを以下のように整理する。

seq 0 あるレコード (R) を App X のチェイン X に追加する要求が来たら、

seq 1 App X 自身が持つチェイン X の end point を得る (集合 A とする)

seq 2 R がもつリンク X のリンク先を得る (集合 B とする)

seq 3 積集合 $P = A \cap B$ を求め、

seq 4 end point に R を加え、P を取り除く

このシークエンスは前節までの各 step 3 によって起動される。step 3 を分解したものと考えてもよい。

4.2 ロックの局所化

3.4 では、そこで示した処理手順について、プロセス間での競合を意識せず step 単位で処理してよいことを示した。ここでは 4.1 で示した各シークエンスの競合について検討する。

まず seq 1,2 は read 処理であり、ロック等は発生しないか共有ロックにとどまる。他のプロセスによる更新作業の影響で A と B が異なる時点の値で得られたとしても問題ない。seq 3 の P は該当プロセスにローカルな資源であり競合はない。つまり seq 4 だけが end point の管理表という共有資源の更新処理を内包する、排他的ロックを要する対象となる。

管理表をアプリケーション・メモリ内の配列とし、直接プログラムが操作するような実装では、seq 4 は自然に一つの atomic な処理となり得るが、管理表を DBMS で実現するような場合、seq 4 は複数の query となるだろう。このような場合、seq 4 を以下のように分解することでロック範囲をさらに局所化できる。

seq 4.1 end point に R を加え、

seq 4.2 end point から P を取り除く

この結果、seq 4 全体を一つのトランザクションとせずに済む。例えば RDBMS では seq 4.1 は一行の INSERT、4.2 も一行の DELETE で実現されるだろうから、DBMS が持つ最もプリミティブな処理単位に作業をおさめることができる。

もし大量削除が相対的に重い処理となるような DBMS では、seq 4.2 をさらに分解して P の要素を一つずつ DELETE してもよい。削除順は問わない。全ての要素を削除する前に他のプロセスが seq 1 の end point 取得を要求した場合、アプリケーションはその時点の情報をそのまま渡せばよい。単に図 7c に示した冗長なリンクが生じるだけである。

結論として、WOODS においてはノード間で協調して行うロックは不要であり、ノード内のロックについても管理表における一行単位の追加・削除処理を行ロックして atomic に処理できれば良く、他は全てロック不要か、seq 単位に共有ロックしながら複数プロセスで並行処理してよい。

P2P 環境ではノード間での対話を要する、あるいは粒度の大きなロックは、その時間的長さと処理の不確実性から行うべきでないため、WOODS のこの特性は P2P システムに適したものと言える。

4.3 関連研究

分散ファイルシステムにおける資源管理手法では一般に性能、可用性の向上と整合性維持コストとのトレード・オフが問題となる。その手法は、単一の更新者のために資源をロックし、他からのアクセスをブロックする悲観的 (pessimistic) 手法と、同期なしの更新を認めるかわりに複数更新者による更新内容の衝突を事後に検出・修復する楽観的 (optimistic) 手法に大別される。

P2P 環境ではネットワーク及び各ノードの可用性を低く見積もるために悲観的手法は不向きである。また、通信遅延が LAN に較べて大きく、ロックなどの状態情報の交換が重くスループットが下がりやすい。さらにメッセージ喪失率の高さがノード故障やデッドロックの検出コストを押し上げることからも、P2P 環境では楽観的手法が適している場合が多い。

しかし楽観的手法は更新衝突とその修復が大きな問題となる。Ivy[2] は更新可能なファイルイメージを提供するが、ネットワーク分割後のマージで発生する衝突の解決にはユーザの手作業を要する。Coda も更新可能なファイルシステムであるが "Conflicts are ultimately an application-specific concept" であるとして [5]、完全な自動修復は考慮されていない。

楽観的更新によって発生する矛盾解決は、たとえ自動化できたとしても全てのシステムに対する一般化は困難であると指摘されており [7]、むしろ CVS (Concurrent Version System) や [8] のように更新の衝突は検出可能でも防止はできず、修復は行わない例が少なくない。

もし Blog システムを他の分散ストレージシステムによって実現した場合、コメントやトラックバックの追記にはノードをまたいだロックか同時追記による衝突の事後検出が必要となる。その影響を低くするためにはコメントを記事データへの追記ではなく独立したファイルへの記入とそのポインタ情報の追記とするなどの工夫が可能であるが、これをつきつめれば結局 WOODS によるデータ構造に類したものとなってしまう。

WOODS は本質的にオフライン更新後のマージでさえ矛盾が発生せず [4]、4.2 で示したようにオンラインでの追記もノード間でのロックや状態の保持を必要としない点で優れている。唯一ロックが必要になるのは end point の更新時、すなわちレコードのチェインへの追加処理だけであるが、それもノード内ローカルかつ極小の範囲で済む。

5 まとめ

本論文では、WOODS を用いたアプリケーションの具体的な設計について示し、そこで必要となるレコード追加処理の詳細を検討した。

例に用いたのは P2P 的な環境で動作する Blog システムであったが、検討した追加処理はすべての WOODS を用いるアプリケーションに求められる一般的な処理である。

また、追加によって発生するデータ更新処理に必要なロックの範囲が極めて限定的であることを示し、他の分散ストレージ・システムにおける更新処理と比較して WOODS の P2P 環境への適性を確認した。

一般に、P2P 分散ストレージで従来的なファイルイメージを提供し、ブロック単位またはバイト単位で更新を可能にするには一貫性維持のための大きなオーバーヘッドと高い可用性をシステムに要求する。

WOODS は、WORM 的なアプリケーションが、よりその特性に適したデータ構造を用いることで、そうした実装・運用面での困難さを低減できる共通基盤を提供するものである。

本研究は WOODS に則したデータ構造で、WOODS の特性を活かしたまま、P2P 環境に適した実用的なアプリケーションが設計できることを示した。

設計した Blog システムは現在実装中であり、主要な API がほぼ出揃った。今後はユーザインタフェイスをもち、実際に操作が可能な状態まで実装を進める予定である。ノード間プロトコルの仕様を決定し、アプリケーションレベルでの記事の削除機能も追加する。

参考文献

[1] F. Dabek et al. : Wide-area cooperative storage with CFS, *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.

[2] A. Muthitacharoen et al. : Ivy: A Read/Write Peer-to-peer File System, *Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[3] J. Kubiatowicz et al. : OceanStore: An Architecture for Global-Scale Persistent Storage, *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Nov. 2000.

[4] 安田 豊 : Unstructured P2P 環境に適したデータ管理方式, マルチメディア通信と分散処理ワークショップ論文集, 情報処理学会シンポジウムシリーズ Vol. 2005, No. 19, 2005 年 11 月

[5] M. Satyanarayanan : The evolution of Coda, *ACM Transactions on Computer Systems (TOCS), Vol. 20, No. 2*, 2002.

[6] J. J. Kistler, M. Satyanarayanan : Disconnected Operation in the Coda File System, *ACM Transactions on Computer Systems (TOCS), Vol. 10, No. 1*, 1992.

[7] Y. Saito, M. Shapiro : Optimistic replication, *ACM Computing Surveys (CSUR), Vol. 37, No. 1*, March 2005

[8] 塚田 大, 鈴木 勝博, 阿部 洋丈, 加藤 和彦 : インターネットを介した協調作業のためのファイル同期システム, 情報処理学会研究報告 Vol. 2005 No. 79, 2005 年 8 月