システムソフトウェアと
オペレーティング・システム 63－1
（1994. 3. 22）

分散協調作業のための一貫性制御プロトコルに基づく分散ファイルシステム

上原 敬太郎　宮澤 元　猪原 茂和　益田 隆司

東京大学 大学院 理学系研究科 情報科学専攻
〒113 東京都 文京区 本郷 7–3–1

要旨

グループ CAD やグループソフトウェア開発などの協調作業を行うアプリケーションを効率良く実装するための枠組として、我々はメモリマップに基づく分散オペレーティングシステム Lucas の研究を行っている。本稿ではファイル特有のインタフェースを用いて分散ファイルの一貫性プロトコルを効率化する手法と、複数の一貫性プロトコルを簡潔に記述するためのプロトコル記述言語の概要について述べる。分散ファイルの一貫性プロトコルには様々なものがあるため、Lucas ではアプリケーションの性質によってユーザーが簡潔な記述でカスタマイズできるような枠組みを提供することでこれに対応する。

Coherence Protocols for Distributed Cooperative Applications in Lucas Distributed File System

Keitaro Uehara, Hajime Miyazawa, Shigekazu Inohara and Takashi Masuda

Department of Information Science, Graduate School of Science,
University of Tokyo
7–3–1 Hongo, Bunkyo-ku, Tokyo 113

Abstract

In cooperative applications such as group CAD and group software development environment, multiple processes share complicated structured data. The Lucas Distributed File System, a platform for developing such cooperative applications, provides an effective way for sharing complicated data structures through memory-mapped files. This paper describes optimized shared memory protocols that take advantage of file interfaces, and an overview of interface for coherence protocols description included as a part of the file system. The protocol description system enables users to customize multiple protocols in order to adapt them for particular applications.

1 はじめに

近年注目を集めているグループCADやグループソフトウェア開発などの協調作業を行なうアプリケーションでは、ポインタを含む複雑なデータ構造を複数のプロセスが共有して使用することが多い。そのようなアプリケーションを効率良く実装するための枠組として、我々は64ビット仮想空間を用いたオペレーティングシステムLucasの研究開発を行なっている。Lucasファイルシステム(以下LucasFS)では64ビットの広大な空間を利用してmemory-mapped fileを永続的にアドレス空間上の特定の位置に置くことによって、複数のファイルにまたがるデータ構造間の参照関係を直接ポインタとして表現できるようになる[5]。また、memory-mapによって同一ホスト内での複数プロセス間での内容の変更を即時に伝播させることを可能としている。

LucasFSではmemory-mapped fileの実現に分散共有メモリの技術を用いているため、mapped fileの一貫性保持の実現方法が重要となる。キャッシュのアクセスパターンはアプリケーションの性質によって異なるため、単一の一貫性プロトコルでは常に最適な結果が得られるわけではないからである。そこで、ユーザーがアプリケーションの種類に応じて自由にキャッシュ一貫性プロトコルを記述することができる機構をシステムに組み込むによってこの問題を解決する。本論文ではLucasFSにおける分散共有メモリプロトコル記述インタフェースを提案し、それの概観を述べる。この記述言語は、簡潔な記述によってユーザーが協調作業用のキャッシュ一貫性プロトコルを組み込むためのものである。いくつかの代表的な協調作業用キャッシュ一貫性プロトコルを挙げ、それらを記述する上での我々のインタフェースの有用性を論じる。

2 Lucasファイルシステムの概要と現状

この章では後の章を説明するのに最低限必要なLucasFSのアーキテクチャについて述べる。

LucasFSは現在DECstation 5000上のMachマイクロカーネルの上にExternal Pagerのインタフェースを用いて実装されている。LucasFSでは、各サイト上のディスクを管理しているストレージサーバ(SS)と各サイト上でキャッシュを管理するキャッシュサーバ(CS)を置き、これらの間で分散共有メモリのプロトコルを用いてキャッシュの一貫性を保証している。各サイト上のクライアントプロセスはファイルを仮想空間にマップすることでそのサイト上のCSと通信をする。CSはキャッシュがサイト上にあればそれをクライアントのアドレス空間にマップし、なければそのファイルが存在するサイト上のSSと通信し、キャッシュを要求する(図1)。CSとSSを分離することでユーザープロセスとキャッシュとのローカルな関係とそのネットワーク上でのキャッシュのグローバルな状態(分散共有メモリでいうディレクトリ)を分けて管理することができる。

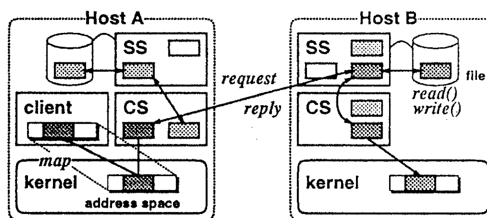

図 1: LucasFS の構成

ローカルサイト上のCSとSSの間の通信はメッセージを用いずに関数呼び出しとして実現されている。これによりローカルサイト内の通信のメッセージ数を減らしている。

2.1 Lucasファイルシステムの一貫性プロトコル

LucasFSでは分散環境上のファイルのキャッシュ一貫性プロトコルには原則的に厳密な一貫性を保証する分散共有メモリのプロトコル(write-invalidation)を使用している。これは分散協調作業のためにクライアント同士で即時に変更を伝播したいという要求からである。しかし厳密な一貫性を保証するセマンティクスではwriteが頻繁になるような協調作業の場合、単純なwrite-invalidationだけでは効率が落ちてしまうことが考えられる。

LucasFSではこの問題を解決するためにファイルとしてのインタフェースを用いた最適化を行なっている。mapped fileにはmapとunmapによる明確なセッションの区切りが存在し、またmapの際に指定するアクセスモードによってread/writeモードでマップしているクライアントの参照とread-onlyでマップしているクライアントの参照を区別することが可能になる。この情報を使って将来的に起こるであろうアクセスの先読みが可

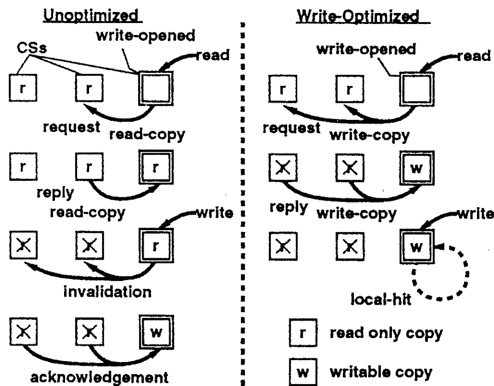

図 2: write-optimized プロトコル

能になる。

read/write モードでオープンしているクライアントのアクセスパターンを考えると、データを読んで、計算して、書き戻す、という一連の動作によって作業するケースが多いと考えられる。write-invalidation のプロトコルに従うと、このようなアクセスパターンでは最初の read で read コピーを要求するフォールトが起こり、次の write で write コピーを要求する、という処理が行われる。write モードでオープンされているクライアントからの read によるページ要求を write とみなすことでこの冗長な処理を省くことができる (図 2)。このプロトコルを write-optimized プロトコルと呼ぶことにする。このプロトコルは write モードでオープンした際に必ず排他的なキャッシュを要求する、すなわち migration を行っている、とみなせる。従って write-invalidation と migration の両プロトコルの中間の性質を示すことになる。

もう 1 つの最適化は read-only モードでマップしているクライアントに関するものである。協調作業の種類によっては、複数の読み手が常に存在しているのだが、それらは必ずしも最新のコピーを参照する必要はなく、それよりも書き込みの際の効率を落とさないようにした方が好ましいケースも考えられる。マップする際に古い read コピーでも構わないという指定をしたクライアントは、最新でない代わりにページアウトされる頻度の低い (高速に参照可能な) コピーを得ることができる (read-optimized プロトコル)。

以上 2 つの最適化を導入して、メッセージ数がどのように減少するか、シミュレーションを行ってみた。その結果を図 3に示す。

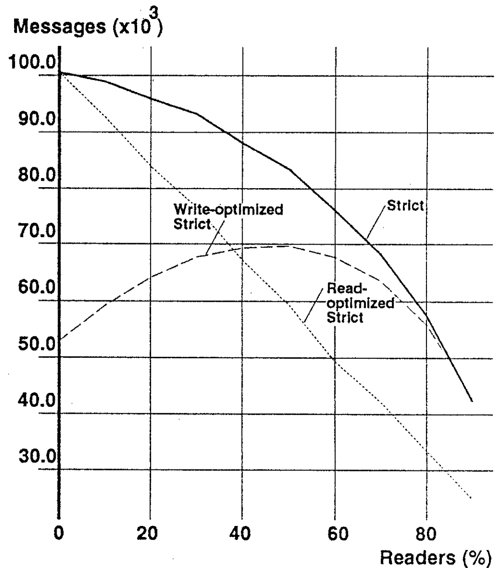

図 3: オプティマイズされたプロトコルのメッセージ数

10 台のホスト上に 1 つずつのプロセスを配置し、100 ページの共有ページに対して合計 10 万回の read および write アクセスを行う。図の x 軸は 10 個のプロセスのうちの reader が占める割合を、y 軸はその時に発生するメッセージ数を表している。writer のアクセスは 40% が write、60% が read としている。一般に write-invalidate プロトコルは writer が多くなるとメッセージ数は増加するが、write-optimized プロトコルでは writer が多い場合に特に効果の大きい最適化となっていることが、このグラフから見て取れる。

2.2 dirty キャッシュのフラッシュ時の最適化

ファイルシステムでは不慮の事故などによって dirty なキャッシュの内容が失われてしまうことを防ぐため、一定時間毎に dirty なページの内容をディスクに書き戻すという操作が行われる。LucasFS でも 30 秒に 1 回 dirty なページをメモリからフラッシュして SS に書き戻すという操作を行っている。

あるホスト上で dirty なキャッシュをフラッシュする時にキャッシュの状態をどのように変化させるかが問題となる。そのホスト上での使用が終わった後も dirty なままキャッシュを残しておけば、将来そのホストで書き込みを行うクライアントが現れた際にキャッシュは有効

となる。しかし、その前に別のホストでそのページを読むクライアントが現れれば、write権をSSに返還する要求メッセージが来ることになる。一般のアクセスにおけるreadとwriteの割合を考えれば、後者の起こる確率の方が高いと言える。

対照的にキャッシュのフラッシュ時にdirtyなキャッシュのwrite権をSSに戻し、readコピーとして保持するアプローチを考える。この場合は別のホストでreadが起こった際にinvalidationが必要ないため効率良く処理することができる。しかし逆に同じホストでwriteが続く場合には、一旦SSに返してしまったwrite権をすぐに要求する結果となり、無駄な処理となる。

以上をまとめると、(1) writeが続く限りdirtyなキャッシュ(とそのwrite権)は保持したい、(2)長い間writeされないキャッシュのwrite権は回収したい、という2つの要求がある。そこでLucasFSではキャッシュフラッシュの際にこの2つの要求に対応できるような最適化を施した。まず、CSがdirtyページのフラッシュを行う際に、ページの保護属性を一旦read-onlyにする。この時write権はCSが保持したままにしておく。従ってSSからはこのCSはただのwriterとして扱われる。次のフラッシュまでの30秒までに書き込みが起これば、ページの保護属性をread/writeに変更する。この操作はSSとメッセージ通信を行う必要はない(write権は保持している)ため、非常に高速に行える。もし次のフラッシュまでに書き込みがなければ(属性がread-onlyのままならば) write権をSSに戻す。これによって、書き込みが頻繁に起こっているホストのwrite権がSSに戻されてしまうことはなく、長い時間アクセスされないdirtyキャッシュのwrite権はSSに戻されることになる(図4)。

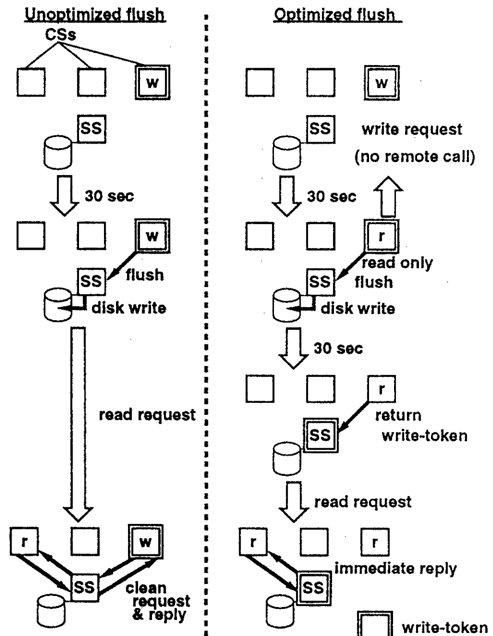

図 4: 最適化されたキャッシュフラッシュプロトコル

3 複数の一貫性プロトコルの必要性

従来の多くの分散ファイルシステムは単一のプロトコルで実装されていて、ファイルシステムとしての通常の使用では十分な性能を引き出している。一方協調作業のための分散共有メモリプロトコルにはさまざまな種類のものがあり、これらはどの1つが常に別の1つよりも優れている、と言った類のものではなく、アプリケーションの性質やデータのアクセスパターンによって選ばれるべきものである。複数の一貫性プロトコルを用意し、ユーザーが適切なものを選ぶことで性能が得られるようにした分散共有メモリシステムにMuninがある[3]。協調作業ではアプリケーション毎の性質を活かして数多くの最適化ができると思われるので、LucasFSではさまざまなプロトコルをアプリケーションの性質に従って柔軟に選択するために、簡潔な記述でプロトコルをカスタマイズする機構を提供することを考えた。

この章では協調作業に有効と思われるプロトコルを紹介する。次の章でプロトコルカスタマイズ機構の概要を述べる。

3.1 アクセスパターンと一貫性プロトコル

プロトコルの性質を決定する要素には、(1)一貫性の厳密さ、(2)メッセージ数、(3)遅延時間(latency)、の大きく3つのパラメータがある。一般にこの3つの要素全てを満たすことは難しい。特に1と2, 3は相反することが多い。厳密な一貫性を求めれば求めるほどキャッシュの無効化(invalidation)のメッセージなどが占める割合が大きくなり、効率は落ちる。逆に効率を上げるためにメッセージ数を減らそうとすれば厳密な一貫性が失われる。

また2と3も必ずしも両立するものではなく、しばし

ば衝突する概念である。latency を小さくしようとするとメッセージ数が多くなってしまう、あるいはメッセージ数を少なくしようと lazy に処理すると実際に処理を行う際の latency は大きくなる、と言った具合である。特に協調作業においては、メッセージ数の増加による全体的な処理時間の遅延を犠牲にしてまでも、latency を下げた方が好ましいようなケースも存在する。

ここでは協調作業に適していると思われるいくつかのプロトコルを挙げ、それらの特徴と有効性を述べる。

- **厳密な一貫性を保証するプロトコル**

  write-invalidation プロトコルや write-optimized プロトコルがこれに当たる。複数の writer や writer と reader が同時に参照を行うような場合には効率が落ちる。

- **複数の読み手 (reader) と 1 つの書き手 (writer) を同時に認めるプロトコル**

  writer とともに古い read コピーのみを参照する reader が複数存在できるプロトコルで、read-optimized プロトコルはこれに当たる。厳密な協調処理を行うプロセス群には使用できないが、キャッシュ無効化のメッセージを発行しないため、writer 1 人に reader 多数というケースでは特に高速な実行が可能となる。

  このプロトコルと厳密な一貫性を保つプロトコルの中間的な存在としてアクセス間の causarity を保証する causal-consistency に基づいたプロトコルもある [2]。同期命令などの拡張無しで read と write の間の causarity が保証されるので、アプリケーションの種類によっては非常に有用なプロトコルであると思われる。

- **同期命令に基づくプロトコル**

  分散共有メモリの weak consistency あるいは release consistency に分類されるプロトコルでは、普通のアクセスに加えて acquire, release という特殊な制御命令を提供している [4]。これらのプロトコルでは制御点においてのみ一貫性を保証し、それ以外の所では一貫性を緩めることで複数のアクセスをひとまとめにして扱い、メッセージ通信の回数を減らして効率を上げている。acquire と release の間だけにアクセスを制限すれば Andrew ファイルシステムのセッションセマンティクスと等価なセマンティクスを実現できる。

- **update ベースのプロトコル**

  複数の writer が同時に存在し、しかもそれらが各々データの更新を行うような場合を考えると、このようなケースでは invalidate に基づくプロトコルでは、最悪の場合にはアクセスの度に invalidation が起こるといういわゆる ping-pong 現象が起きてしまい、効率的にキャッシュの一貫性を保つことはできない。このような場合には変更されたキャッシュの内容を共有しているサイトにマルチキャストする update に基づくプロトコルが最も適していると言える。

- **memory-mapped stream 通信**

  1 つのプロセスがデータを書き込み、もう 1 つのプロセスがそれを読み込むというストリームのような一方向の通信を考える。メモリマップでストリーム通信をする際に、これまでにあげたような demand driven に基づくプロトコルでは、読み取る側の latency が問題となる。書き終わったページのうち、読み取る側の欲しているページをいかに素早く転送し latency を小さくするかが、ストリーム用のプロトコルの大事なポイントとなる。そのために、マップされたデータの型情報を元にしてプリフェッチを行う、などと言った最適化が考えられる [8]。

4 ユーザーレベルでのプロトコルカスタマイズ機構

LucasFS では前の章で述べた複数のプロトコルをユーザーが簡潔な記述でカスタマイズできるような機構を提供する。

4.1 プロトコル記述の問題点とその解決法

実際に C 言語で複数のプロトコルを記述してみた結果、いくつかの点でプロトコルを記述するのに注意すべき点、難しい点が存在することがわかった。

第 1 は CS とカーネルとの間の通信によって状態数が多くなってしまう点。これは SS から見た CS の状態は一緒でも、CS の内部でデータがカーネルにあるかサー

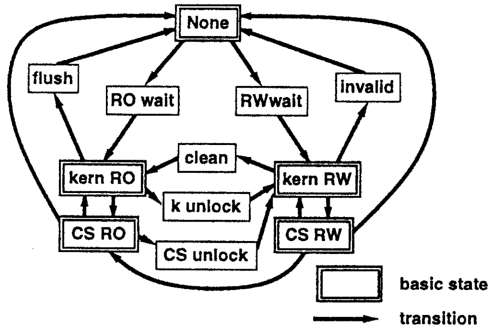

図 5: C で記述した場合の CS の状態遷移図

バにあるかで処理が変わってくる、というものである図5を見るとわかるように、5つの基本状態の他にカーネルとの通信によって起こる遷移の中間状態というものが多数存在し、非常に複雑な状態遷移となっている(なおこの図にはキャッシュフラッシュの最適化のための状態は含まれていないため、実際にはもっと多くの状態が存在する)。

第2には分散環境においてはメッセージの到着順が必ずしも一定していない点である。そのために現在処理できないメッセージをキューにつないだり、適切な時に取り出して処理したりといった記述が必要になる。しかしユーザーが常にそういう処理を記述するのは煩わしいし、バグを生ずる原因にもなる。

第3にページというものが単体で扱われることもあるが、他のページと組合わさって協調動作を行うこともあり得る、という点である。例えばストリームやシーケンシャルファイルアクセスなどを効率良く行うためには、ファイル中の連続したページをいくつかまとめて送る、といった動作が必要になるし、また dirty なページや古いページなどのファイルとは関係ない属性によるページの集合に対して動作を行う場合もある。

第1の点は CS とカーネルとを同一視して見せることで複雑さを吸収できるのではないかと考える。ユーザーは単にキャッシュの保護属性を read only にする、などと記述すれば、明示的な記述はなくとも内部状態に従って必要な処理が適切に行われる。カーネルからのページ要求などはユーザーに適切な形でイベントとして渡してやればよい。

第2の点はメッセージやカーネルからの要求を全てイベントとして扱い、イベントは処理可能な時にのみ発火し、それ以外のイベントは処理可能になるまでキューに蓄えることにする。これにより2つのイベントの到着順が入れ替わったとしてもユーザーはそれを意識せずに記述することが可能となる(`recv A; recv B` という記述があった時に、メッセージが B→A の順に届いたとしても全く同じ様に実行されることになる)。また複数種類のイベントを待ち合わせて実行を分岐する `select` 命令も必要になる。1つのイベントを待つ `recv` 命令は `select` の特殊な形態とみなすこともできる。

第3の点はページの集合や連続したページといった概念を抽象的に記述できるようにすれば解決する。あるページの次のページを参照したり、複数のページをまとめてメッセージとして送受信できるようなインタフェースを設ける必要がある。

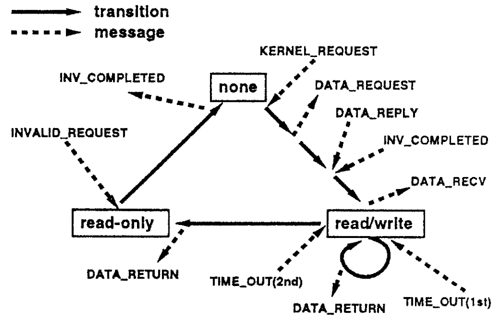

図 6: CS の状態遷移図

4.2 プロトコル記述の例

一例として write-invalidation プロトコルをこのプロトコル記述言語に従って記述してみる。プロトコル記述言語としては、C 言語への組み込みが簡単である、という理由から Tcl [9] をベースにし、キャッシュ制御用のプリミティブを追加している。

このプロトコルでは、CS のキャッシュの状態はキャッシュがない(`NONE`)、read-only なキャッシュを保持している(`RO`)、read/write 可能なキャッシュを保持している(`RW`)の3つの状態があり、これらの間をカーネルからの要求、SS からの要求といったイベントに従って遷移することになる(図6)。

CS とカーネルとが同一視されているため、CS の状態数は3つとかなり単純化されている。図7(a)の3〜12行がカーネルからの要求に対する記述である。クライアントからの要求イベント(`CLIENT_REQUEST`)が起こった場合には、ページの要求を SS に発行し(イベ

ント名 DATA_REQUEST)、返事を受け取る。そして set_protection の一文で保護属性を変更している (実際の処理としてはカーネルの要求に従い、データを供給したり、あるいはロックを外したりといった適切な処理が行われることになる)。SS からの要求に対する記述は 13 ～ 16 行で、保護属性を変え、acknowledgement を発行している。

少し複雑なのはキャッシュフラッシュ時に汚れたページを SS に返す処理を行う部分である (図 7の 18 ～ 28 行)。第 2.2 節で述べた最適化によって 2 段階に分けてページの状態を変化させて処理するために、最初はページの属性を read-only に変えるがページセット dirty_page からは外さない。次にこれが呼ばれた時に属性が read-only のままならば、その時 dirty_page から外す、ということをしている。ここでは省略したが、DATA_RETURN を受け取った SS はページをディスクへ書き戻す、という処理を行う。

次に SS の方の記述を少し見てみる。図 7(b) は CS からのデータ要求 (DATA_REQUEST) の一部である。問題は、メッセージを送ってそれの返答 (acknowledgement) を待つところである。ここで、CS が普通にメッセージを受け取って返事を送る場合と、CS と SS が同時にメッセージを送信するという競合状態が起こる場合が考えられる。このため select による分岐が行われる。図 7(b) では 6 行目で送ったメッセージに対し、7 行目からの select 文で 2 通りの処理を記述している。ここでは writer が data と invalidation を全て受け取って正常に遷移が行われる場合 (DATA_RECV) と、reader がデータを返してきた (DATA_RETURN) ためにそれに代わって acknowledgement(INV_COMPLETED) を転送しなくてはならない場合、が書かれている。select に現れないイベントが到着した場合には自動的にキューに入れられ、必要なときに取り出されることになる。

4.3 プロトコル記述言語の有効性の検討

このプロトコル記述システムで、4.1 にあげた (1) CS とカーネルの間のメッセージ交換の抽象化、(2) メッセージの到着を含めたイベントの抽象化、(3) 複数のページなどの集合の抽象化、といった課題はクリアできたと考える。

このようなプロトコル記述言語を提供する際、次に問題となるのは出来上がったプロトコルが正しく動くかどうかを実際に動かしてみないとわからないという点であろう。プロトコルの正当性や完全性を形式的な処理で証明することができる形式的プロトコル記述系や言語も存在するが、正当性を検証するためにはプロトコル全体をスクラッチから形式言語で書かなくてはならない。我々のシステムは簡潔な記述によってプロトコルをカスタマイズするために導入したのでこのような形式的仕様記述の形は取っていないが、エラーや無限ループ等のプロトコルのバグの発見を手助けする機構は必要であると考えている。

```
 1: select {
 2:  ...
 3:  CLIENT_REQUEST : page rmsg {
 4:   ...
 5:   send SS { DATA_REQUEST RW }
 6:   recv DATA_REPLY bmsg
 7:   mrecv INV_COMPLETED $bmsg(inv_count)
 8:   set_data $page $bmsg(data)
 9:   set_protection $page RW
10:   send SS { DATA_RECV }
11:   add_page $dirty_page $page
12:  }
13:  INVALID_REQUEST : page rmsg {
14:   set_protection $page NONE
15:   send $rmsg(send_to) { INV_COMPLETED }
16:  }
17:  ...
18:  TIME_OUT : rmsg {
19:   forall page $dirty_pages {
20:     if {[get_protection $page] == RW} {
21:       set_protection $page RO
22:       send SS { DATA_RETURN [get_data $page] }
23:     } else {
24:       send SS { DATA_RETURN WO }
25:       remove_page $dirty_pages $page
26:     }
27:   }
28:  }
29: }
```

(a)CS のプロトコル記述

```
 1: select {
 2:  ...
 3:  DATA_REQUEST : page rmsg {
 4:   ...
 5:   msend $readers($page) { INVALID_REQUEST }
 6:   send $writer($page) { DATA_REPLY RW $data }
 7:   select {
 8:     DATA_RECV : rmsg {
 9:       set $writer($page) $rmsg(reply_to)
10:     }
11:     DATA_RETURN : rmsg {
12:       send $rmsg(reply_to) { INV_COMPLETED }
13:     }
14:   }
15:  }
16: }
```

(b)SS のプロトコル記述

図 7: プロトコル記述の例

もう1つの問題は、ユーザーが記述できる自由度がまだ高すぎるためユーザーの記述量が多い点である。複数プロトコル間の共通項を括り出し、なるべくブラックボックス化してしまった方が、ユーザーの記述の点から言っても、実行の効率の点から言っても好ましいはずである。いくつかプロトコルを記述してみて、一階層だけでなく多数の階層でデザインができるようなシステムを用意できれば、記述のし易さと自由度の両方を満足するようなシステムになると考えている。

5 関連研究

従来の分散ファイルシステムでは、完全な分散共有メモリのプロトコルに基づいているものは少ない。例えばSpriteでは書き手が複数になるとキャッシュの使用をやめて遠隔サービスに切り替える。Andrewではそもそも厳密な一貫性を保証しておらず、もっと緩い一貫性(セッションセマンティクス)に基づいている[6]。その理由はこれまでの単一ユーザーのアプリケーションにおいては一般にファイルの共有(write sharing)は稀であるということと、分散ファイルシステムはより広域のネットワークで使用されるケースが多いため、厳密な一貫性を保証するよりも効率の面を優先する場合が多いということが挙げられる。invalidationプロトコルで厳密な一貫性を保証している分散ファイルシステムに関する研究としてはEchoファイルシステム[1]がある。

複数の一貫性プロトコルを用意した協調作業用の分散共有メモリシステムとしてMuninが挙げられる[3]。Muninではデータごとにユーザーが指定することで予め用意されたいくつかの一貫性プロトコルのうちのどれを使うかを決めることができる。

形式的プロトコル記述言語にはPSF [7]やLOTOSなど多数存在する。これらのプロトコル記述言語の主な目的はプロトコルの仕様記述と検証である。これらは分散システムの構成、状態、メッセージ、等の仕様を記述することで、プロトコルの正当性を判別したり、テストを自動生成して検証したりすることができる。

6 まとめ

本論文ではLucasファイルシステムの概要と現状を述べ、その上でファイルであることを利用した分散共有メモリプロトコルの最適化手法について述べた。また協調作業用のプラットフォームとして確立していくためには複数のキャッシュ一貫性プロトコルをサポートできる必要があることを述べ、分散共有メモリプロトコルをユーザーがカスタマイズできる方法としてインタプリタ型言語によるプロトコルの組み込みの概要を述べた。

理想的なプロトコルの記述は、CSとSSの状態遷移図のようなものをユーザーが入力として与えると、適切なプロトコルに変換して実行されるようなモデルであろう。今回のプロトコル記述言語によって複数の一貫性プロトコルの共通項を見つけ出し、このようなモデルに近づけることが将来的な目標である。

参考文献

[1] A. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo Distributed File System. Technical Report 111, Degital SRC, 1993.

[2] F. Boyer. A Causal Distributed Shared Memory Based on External Pagers. In *Proceedings of the USENIX Mach Symposium*, pp. 41–57, Nov 1991.

[3] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pp. 152–164. Association for Computing Machinery SIGOPS, Oct 1991.

[4] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 15–26. IEEE, May 1990.

[5] 猪原茂和, 上原敬太郎, 宮澤元, 益田隆司. オペレーティングシステムLucasにおける64ビットアドレス空間の管理. In *6th SWoPP*, Aug 1993.

[6] E. Levy and A. Silverschatz. Distributed File Systems: Concepts and Examples. *ACM Computing Surveys*, Vol. 22, No. 4, pp. 321–374, Dec 1990.

[7] S. Mauw and G. J. Vetlink, editors. *Algebraic Specification of Communication Protocols*. Cambridge university press, 1993.

[8] 宮澤元, 猪原茂和, 上原敬太郎, 益田隆司. Lucasオペレーティングシステムにおけるメモリマップ技術を用いたストリーム型プロセス間通信. 第47回(平成5年後期)情報処理学会全国大会, volume 4, pp. 23–24, Oct 1993.

[9] J. K. Ousterhout. Tcl: An Embeddable Command Language. In *1990 Winter USENIX Conference Proceedings*, 1990.