ソフトウェア基礎論　23－1
（1987. 12. 10）

分散ファイル・システムの比較と問題点

A Comparison and Ploblem of Distributed File Systems

中島達夫　　所真理雄

Tatsuo Nakajima and Mario Tokoro

慶應義塾大学理工学部

The faculty of science and technology

Keio University

あらまし　ワークステーションとローカル・エリア・ネットワークの発達により分散システムの時代へと移りつつある。分散システムを構築する際の最大の問題点はリソースの共有である。多くのＯＳはＵｎｉｘなどのようにファイルを用いてユーザ間で情報の共有を行なっている。そのため、分散ファイル・システムの実現が分散システム構築の際の最大の問題となる。本論文では、最近の分散ファイル・システムについていくつか紹介したあと、我々が現在、開発中の分散ファイル・システムについて述べる。

ABSTRACT The age of distributed systems is coming because of the development of workstations and local area networks. In distributed systems, the most important issues is the shring of resouces. Many operating systems such as Unix make it possible to share resouces through the file system. In this paper, we introduce several distributed file systems, and describe a distributed file system which we are designing and implementing.

1　はじめに

ＴＳＳの時代からワークステーションの時代への流れがますます進んでいる。以前のメインフレームと同じパワーを持つワークステーションが手頃な値段で手にはいるようになり、各自が個人のワークステーションを持ちその力を最大限に利用できるパーソナル・コンピューティングの実現が可能となった。そのため、そのワークステーションをネットワークでつないだ分散システムが今後の計算機の主な形態となると思われる。しかし、分散システムでは、いかにして各ワークステーションのデータを共有するかという問題を解決しなければいけない。現在、多くのＯＳではユーザ間のデータの共有の手段としてファイルを用いている。しかし、分散環境では各ワークステーションが個別にファイル・システムを持っているため、ファイルを共有するためにはそれらのファイル・システムを有機的につなげる分散ファイル・システムが必要不可欠である。

本論文では、現在実際に稼動中の3つの分散ファイル・システムについて述べたあと、現在、我々が開発中であるＴｏｎＴｏｓ分散ファイル・システムについて述べる。

2　分散ファイル・システム

ネットワークにより複数の計算機がつなげられるようになったのは１５年以上前のことである。それから、実用的な分散ファイルが開発されるまで１０年以上かかっている。この節では、ネットワークが開発された初期から分散ファイル・システムが登場するまでの歴史について述べる。

2．1　ファイル転送

ネットワークが開発された初期は計算機間で通信するためのプログラムとして、リモート・ログイン、ファイル転送、リモート・コマンド実行、メイル・システムなどが作られた。しかし、このような環境では一般に、ユーザはリソースの存在する位置がわからないとそれにアク

セスすることが不可能だった。また、ユーザはリモート・サイトにあるファイルをアクセスするとき、ファイル転送によりいったんローカル・サイトにコピーしてからアクセスしなければならなかった。これはファイルのコピーがあちこちのサイトにできてしまうため、ファイルの一貫性と、ディスクの有効利用の面から好ましくない。

２．２　リモート・ディスク

ワークステーションで必要な高速のディスクは大きさや騒音、価格の面から机の上で用いるのには問題があるため、ディスクレス・ワークステーションが今後重要になってくると思われる。Ｓｕｎではこのためにネットワーク・ディスクを導入した。これはリモート・サイトのディスクを自分のサイトのディスクのように使えるようにするためのソフトウエアである。そして、すべてのファイル・アクセスはネットワークを介してこのディスクをアクセスする。この問題点は、ＯＳのディスク・ドライバをネットワーク・ディスクにかえただけなのでネットワークのアクセスが非常に多くなるためサーバの負荷が重くなり効率があまりよくないことである。しかし、リモート・ディスクはＯＳを変更することなくディスクレス・ワークステーションを実現できるので今後も普及すると思われる。

２．３　分散ファイル・システム

分散ファイル・システムは分散システム実現のキーポイントとなっている。もはや、分散ファイル・システムなしではワークステーションは存在しないとまでいえるようになってきている。初期の分散ファイル・システムではアクセスの透過性の実現が問題となった。ここでは、リモートにあるファイルもローカルにあるファイルと同じ形態でアクセスできるようにすることが最大の問題だった。その次の世代の分散ファイル・システムではファイルの位置の透過性が目標になった。つまり、ファイルがどの位置に存在していてもユーザはその位置を知らずにアクセスすることが可能となる。そして、現在の世代の分散ファイル・システムでは、ネーミングとファイル・アクセスの効率化が開発の中心となっている。次節では、この例としてＮＦＳ[1]，Ａｎｄｒｅｗ[2]，Ｓｐｒｉｔｅ[3]の３つのファイルシステムについて述べる。これらは、いままでの分散ファイル・システムと異なりネーミングを工夫し、ファイル・アクセスを効率よくするためキャッシュを導入している。

３　分散ファイル・システムの現状

この節では、ＮＦＳ，Ａｎｄｒｅｗ，Ｓｐｒｉｔｅの３つの分散ファイル・システムについて、ネーミング、キャッシュの管理、ディレクトリの管理、ファイルの書き込み、トランスポート・メカニズムについて紹介し、それらの問題点について検討する。

３．１　ＮＦＳ

ＮＦＳはＳｕｎが作った分散ファイル・システムである。このシステムではＵｎｉｘ間だけではなくＭＳ－ＤＯＳやＶＭＳなどヘテロジェニアスな環境でのファイル・アクセスを実現している。そのため、ファイル・アクセスのためのプロトコルを公開しそのプロトコルに従えば、どのファイル・システムのファイルでもアクセスすることが可能である。

３．１．１　ネーミング

ＮＦＳではリモート・サイトのファイルをアクセスするためリモート・サイトのサブ・ツリーをローカル・サイトにマウントする。これにより、他のサイトのファイルを自分のサイトのファイル・システムの一部としてアクセスすることが可能となる。この方式は非常に柔軟が高く様々な利用形態に対応できる。

３．１．２　キャッシュの管理

ＮＦＳではファイルの読みこみのときメモリ上にブロック・キャッシュをとることにより高速化を実現している。しかし、ＮＦＳでの書き込みポリシでは他のサイトのキャッシュの無効化まではおこなわないため、ファイルの一貫性を保つため、リードするごとにリモート・サイトからファイルを読みこまなければいけない。だが、それでは効率が悪くなるため、ＮＦＳでは一貫性を多少犠牲にして、ファイルをオープンしてからクローズするまでだけそのファイルのキャッシュを有効にしている。しかし、Ｕｎｉｘでは一度読んだファイルのブロックを二度アクセスすることは少ないのでこの方法では効率は思ったほど向上しない。しかし、ヘテロジェニアスな環境ではキャッシュの管理を複雑にすると、プロトコルが複雑になり各ＯＳのファイル・システムの変更が困難となるので、効率を犠牲にしてプロトコルを簡単にしている。

３．１．３　ディレクトリの管理

ＮＦＳでは、ディレクトリの解析はサーバで行なわれる。つまり、クライエントがファイルのパス名のコンポネントをサーバに渡すと、サーバでディレクトリを解析しそれが指すファイルへのポインタが返される。この方式はサーバの負荷を増やすがディレクトリのフォーマットが異なっていても統一的にディレクトリを扱えるため、

異なるファイル・システムをつなぐことが可能となる。

３．１．４　ファイルの書出し

ＮＦＳではファイルの書出しはライト・スルーでおこなっている。ファイルへのライトするごとにリモート・サイトがアクセスされるためサーバの負荷が非常に重くなる。また、ライト・システムコールの内部で、ＲＰＣをそのまま用いるとそれが終了するまでいつも待たされるためパフォーマンスが悪くなるので、サーバへのライトを実際におこなうのはｂｉｏｄというデーモンにまかせ、システム・コールはすぐにリターンする。

３．１．５　トランスポート・メカニズム

ＮＦＳはトランスポート・メカニズムとしてＲＰＣを用いている。このＲＰＣでは、下位のプロトコルとしてＵＤＰを用いている。また、エラー処理としては簡単なリトライのみしかサポートしていない。これは、ＮＦＳのサーバはステートレスなので、おなじＲＰＣを何度繰り返してもいつも同じ結果を返すためである。また、メッセージ転送時のデータのコピーを減らすためメモリーバッファという概念を導入している。これでは、メッセージは１１２バイトごとのリンクド・リストとして管理され、例えばトランスポートヘッダをつけるときは、そのリンクド・リストの前にエンキューしコピーはおこなわれない。そのため、データのコピーは最小ですむ。

サーバはカーネル内のマルチ・スレッドサーバとして実現されている。そして、リクエストが来るごとにインアクティブなサーバがそれを処理する。

３．２　Ａｎｄｒｅｗ

ＡｎｄｒｅｗはＣＭＵで作られた分散ファイル・システムである。このシステムはＵｎｉｘの４．２ＢＳＤ上に作られ、他のＯＳ上での使用は考えていない。このファイル・システムの特徴は数千台のワークステーション環境で用いるための分散ファイル・システムを作ることである。このシステムでは各自のワークステーションであるＶｉｒｔｕｅとファイル・サーバであるＶｉｃｅの２つの分類される。Ｖｉｃｅ上のファイルは単一のツリーとして管理され、そのファイルはＶｉｒｔｕｅからは同じ形態でアクセスすることが可能である。

３．２．１　ネーミング

Ａｎｄｒｅｗでは共有ファイルはすべてＶｉｃｅ上の単一のツリーとして管理される。そして、そのツリーは各自のワークステーションのファイル・システムにマウントすることができる。これにより、Ｖｉｃｅ上のファイルがローカル・ファイルのようにアクセスできるようになる。

３．２．２　キャッシュの管理

Ａｎｄｒｅｗではファイルをアクセスするときファイル全体をローカル・ディスクへキャッシュする。つまり、ファイルをオープンしようとすると、いったんファイル全体をローカル・サイトのディスクにコピーしてからアクセスされる。このようにしたのは、Ｕｎｉｘでのファイル・アクセスはファイル全体を１度なめるようにアクセスすることが多いためである。しかし、ローカル・ディスクの存在が前提なためディスクレス・ワークステーションに対応することは不可能である。しかし、ローカル・ディスクが十分な量があれば多くのファイルがキャッシュされるのでサーバへのアクセスはほんとんどおきない。しかし、この方式だと１度目のアクセスはファイル全体のコピーが必要なのでオープンの処理はファイルのサイズに比例した時間がかかる。また、ディスク容量より大きなサイズのファイルをアクセスすることはできないなどの問題が残されている。

Ａｎｄｒｅｗではキャッシュの効率を向上させるためＮＦＳより複雑なキャッシュ管理アルゴリズムを持っている。Ａｎｄｒｅｗの初期のバージョンではファイルをオープンするごとにサーバにアクセスしてキャッシュが古くなっていないかをチェックし、キャッシュの一貫性を保っていた。しかし、例えば、ｌｓなどのコマンドを実行すると、何度もｓｔａｔシステム・コールが実行され、そのたびごとサーバへのアクセスがおこなわれるのでパフォーマンスが悪くなる。そのため、新しいバージョンではファイルのキャッシュを持っているサイトをサーバが知っていて、どこかのサイトがファイルへ書き込みをおこなうと、そのファイルのキャッシュを持つすべてのクライエントのキャッシュを無効にする。これによりサーバの負荷が減り効率が向上した。

３．２．３　ディレクトリの管理

Ａｎｄｒｅｗではサーバの負荷を軽くするため、ディレクトリの解析をクライエントでおこなっている。つまり、ディレクトリを読むときはクライエントにいったんディレクトリ・ファイル全体をキャッシュしてからローカル・サイトでディレクトリの解析をおこなう。しかし、すべてのディレクトリが同じフォーマットであることを前提としているため、Ａｎｄｒｅｗでは異なったファイル・システムをつなぐことは不可能である。

３．２．４　ファイルの書き込み

Ａｎｄｒｅｗではファイルの書き込みはクローズしたときに行なわれる。つまり、書き込みごとにサーバへアクセスしないので効率が向上する。しかし、コンパイル時のテンポラリ・ファイルのように、作られてすぐ消されてしまうファイルまではサーバへ書きだされるためサーバへのアクセスは思ったより減らないので効率はあまり向上しない。

３．２．５　トランスポート・メカニズム

Ａｎｄｒｅｗではトランスポート・メカニズムとしてＲＰＣを用いている。このＲＰＣではＮＦＳ同様、下位のプロトコルとしてＵＤＰを用いている。しかし、ＮＦＳと異なりサーバはステートを持っているため、複雑なエラー処理をおこなっている。

Ａｎｄｒｅｗの初期のバージョンではサーバはクライエントから要求をうけとるごとに新しいプロセスを作りそれを処理していた、また、サーバ間で共有する情報はファイルに置いていた。しかし、Ｕｎｉｘではプロセス生成のオーバヘッド、及びファイル・アクセスのオーバヘッドは非常に大きいのでこの方法はサーバの負荷を重くしていた。そのため、新しいバージョンでは１つのＵｎｉｘプロセス上にコルーチンで動くライト・ウエイト・プロセスを実現し、要求がくるごとに新しいライト・ウエイト・プロセスを作り処理をおこなう。また、共有情報がサーバ内のメモリ上におけるのでオーバヘッドが減り効率が向上した。しかし、Ｕｎｉｘではコルーチンの実現は効率がよくないため、サーバはカーネル内でのマルチ・スレッド・サーバとして実現するほうが好ましい。

また、Ａｎｄｒｅｗではファイル全体の転送が非常に多くなるため、ラージ・ファイルの転送の効率がよいトランスポート・メカニズムが必要となる。

３．３　Ｓｐｒｉｔｅ

ＳｐｒｉｔｅはＵＣＢで作られた分散ファイル・システムである。Ｓｐｒｉｔｅではディスクレス・ワークステーションでの使用を前提としているためキャッシュ・メカニズムが充実している。また、このシステムでは各ワークステーションは１００Ｍバイト以上のメモリを持っていることを前提としてキャッシュ・メカニズムを設計している。この考えは、現在のメモリの価格低下を考えると好ましいと思う。この分散ファイル・システムはＳｐｒｉｔｅ分散ＯＳ上で動いている。

３．３．１　ネーミング

Ｓｐｒｉｔｅでのネーミングは Ｐｒｅｆｉｘ　Ｔａｂｌｅというメカニズムを用いている。これには、パス名とそのパス名が指すファイルが存在するサーバとそのサーバ内でのオフセットが記述されている。

／	サーバＡ	ｏｆｆｓｅｔ　１００
／ｕｓｒ	サーバＢ	ｏｆｆｓｅｔ　２００
／ｕｓｒ／ｔｍｐ	サーバＣ	ｏｆｆｓｅｔ　１５０

パス名を解析する際、一番長くマッチするものが選び、残りのパス名をそのサーバに送る。つまり、／ｕｓｅｒ／ａをアクセスしようとすると、これは／が一番マッチするのでサーバＡにｕｓｅｒ／ａを送る。また、／ｕｓｒ／ｔｍｐ／ｂをアクセスしようとすると、／ｕｓｒ／ｔｍｐ／がマッチし、サーバＣにｂが送られる。また、サーバが反応しないときはブロードキャストにより他のサーバにファイルのリプリケーションが存在しないかを訪ねる。そして、もし、リプリケーションが存在すると、それを持つサーバから結果が返され、それを新しいエントリとしてテーブルに登録し、それから後はその新しく登録されたサーバをアクセスする。

また、Ｐｒｅｆｉｘ　Ｔａｂｌｅではテーブルを作るのにもブロードキャストを用いている。つまり、自分が解釈できないパス名はブロードキャストを用いて他のサーバに聞く。そして、返ってきた結果をテーブルに登録する。このように、スタティックな方式にブロードキャストを組み合わせることにより柔軟な方式を実現している。

３．３．２　キャッシュの管理

Ｓｐｒｉｔｅではファイルのキャッシュはブロック単位に行なわれる。そして、キャッシュはＮＦＳと同様メモリ上に置かれる。これは、ディスクレス・ワークステーションの使用を考えているからである。つまり、各ワークステーションが１００Ｍバイト以上のメモリを持っていると、ユーザが使うファイルの大部分はローカル・サイトのキャッシュ上にもってくることが可能である。そのため、１たんキャッシュがとられるとディスク・アクセスがなくなるためファイル・アクセスの効率は非常によくなる。しかし、Ｓｐｒｉｔｅではファイルの先読みをおこなっていないためリードのパフォーマンスはＮＦＳより悪い。

このシステムはキャッシュの一貫性の維持のため非常に複雑なメカニズムを持っている。今、あるクライエントがオープンしているファイルを別のクライエントがオープンしようとすると、前のオープンがリード・オープンならそのファイルに関するすべてのキャッシュを無効にし、ライト・オープンならキャッシュをフラッシュしてからキャッシュ・メカニズムをストップする。この方式

では、オープンが競合するときはオーバヘッドが大きくなるが、通常これはまれにしかおきないのでこの方式は好ましいと思われる。また、ファイルをオープンしたとき、キャッシュがローカル・サイトに存在すると、常にサーバをアクセスしてキャッシュが古くなっていないことを確認する。この方式は、Ａｎｄｒｅｗの初期バージョンと同様、サーバのアクセスが多くなり効率が悪くなる。また、クライエントのみでおこなうような最適化（パス・ネーム・キャッシュなど）を用いることは不可能である。

３．３．３　ディレクトリの管理

Ｓｐｒｉｔｅでは自サイトでパス名が解析できなくなるとそこから先のパス名をまとめてサーバに渡す。そして、サーバがそれを解釈しその結果としてファイルへのポインタを返す。このように、Ｓｐｒｉｔｅではサーバのロードを低くするということをあまり考えていないのでクライエントの数が増えるとサーバの負荷が大きくなる可能性はある。

３．３．４　ファイルの書き込み

Ｓｐｒｉｔｅではファイルの書き込みはライト・バックで行なわれる。つまり、ファイルへのライトがおこなわれてもすぐにサーバはアクセスされない。そして、書き込みが行なわれたブロックはサーバへ３０秒に１回定期的に書きだされる。これはＵｎｉｘでのファイル・ブロック・キャッシュの書きこみと同じポリシーである。しかし、テンポラリ・ファイルなどはサーバに書きだす必要がないので、効率をあげるためにはこの時間を長くする必要がある。しかし、こうすると、クラッシュ時にデータが大量になくなる可能性がある。

３．３．５　トランスポート・メカニズム

Ｓｐｒｉｔｅではトランスポート・メカニズムとして前の２つのシステム同様ＲＰＣを用いている。しかし、このＲＰＣでは、下位のレイヤとして他のシステムと異なりＩＰを用いている。また、このＲＰＣはエラー処理を完全にサポートしている。つまり、リプライが返らないとクライエントはリクエストを再送する。そして、リクエストがいったん実行されたときはリプライは取っておかれリクエストが再送されてきたときは、そのとっておいたリプライを返す。

また、リクエスト、リプライのアックはできるだけピギーバックするようにして効率を向上させている。また、パラメータの性質を利用してデータのコピーを減らすようにしている。

また、サーバはＮＦＳ同様カーネル内にマルチ・スレッド・サーバとして複数個作られる。これはＮＦＳと同じメカニズムである。

３．４　分散ファイル・システムのまとめ

以上この節では３つの分散ファイル・システムについて述べた。ＮＦＳは様々なファイル・システムをつなげることができるが、効率の面で問題がある。Ａｎｄｒｅｗではキャッシュをとるための時間がかかりすぎる。Ｓｐｒｉｔｅは分散ファイル・システムとしての完成度は非常に高いが、オープン時に必ずサーバをアクセスするのは問題がある。また、ヘテロジェニアスな環境に体してなにも考慮されていない。

４　ＴｏｎＴｏｓ分散ファイル・システム

ＴｏｎＴｏｓ分散ファイル・システムは３節で述べた分散ファイル・システムの欠点を補った分散ファイル・システムである。このファイル・システムはＮＦＳと同時に用いることによりヘテロジェニアスな環境でも用いることが可能である。つまり、このファイル・システムを持つサイトからＮＦＳのみ持つサイトのファイルをアクセスできるし、ＮＦＳのみ持つサイトからＴｏｎＴｏｓのファイルをアクセスすることもできる。また、ＮＦＳのサイトからのマウントはサーバに直接せず、他のクライエント・サイトにマウントし、このクライエントがサーバからキャッシュをとることによりサーバの負荷を減らすことが可能となる。

４．１　ネーミング

ＴｏｎＴｏｓでのネーミングはＮＦＳと同様、リモート・マウントにより他のサイトのファイル・システムをマウントする。この方法は、小規模なシステムから大規模なシステムまでサポートできるため非常に柔軟性が高い。しかし、ＮＦＳと異なり、サーバはサブ・ツリーを他のクライエントから容易にアクセスできるようネーム・サーバにシンボリック名として登録する。そして、クライエントは、このシンボリック名を用いてマウントする。これは、ＡＴ＆ＴのＲＦＳと非常に似ている。このとき、同じ名前のサブ・ツリーが複数存在すると、それらはリプリケーションとして扱われる。これらは、プライマリとセカンダリとに分けられ、クライエントは通常プライマリのサーバのサブ・ツリーをマウントする。しかし、このサーバがダウンしたときはセカンダリのサーバのサブ・ツリーがプライマリとなり、クライエントはこれをマウントする。これにより、アベイラビリティは向上する。クラッシュしたプライマリは立ち上がるとセカンダリとなり、新しいプライマリからファイルをコピーしてくる。

この分散ファイル・システムでは、プロセス・マイグレーションをおこなったときファイル・アクセスの効率を向上させるため、各サイトのファイル・システムのツリーの作り方を工夫している。各サイトのローカル・ファイル・ツリーは次の3つにわけられる。

1) Ｒｅｓｉｄｅｎｔ Ｔｒｅｅ
2) Ｎｏｍａｌ Ｔｒｅｅ
3) Ｔｅｍｐｏｒａｒｙ Ｔｒｅｅ

Resident Treeには各サイトが共通に持っているファイルが存在している。これに属するファイルはディスクレス・ワークステーションでは立上り時に全部ローカル・サイトのメモリ上にキャッシュとして持ってこられる。また、プロセスが他のサイトにマイグレーションしたとき、Resident Treeのファイルは移動した先のサイトのファイルがアクセスされる。Normal Treeは各サイトごとのローカル情報、そのワークステーションのユーザのファイル、そして、通常は使われないコマンドなどが置かれている。ディスクレス・ワークステーションでは、これらのファイルは必要に応じてサーバから持ってこられる。また、プロセスがマイグレーションしたときは、Normal Treeは移動前のサイトのものがアクセスされる。つまり、そのようなプロセスは移動前のサイトのNormal Treeへのポインタを持つことになる。Temporary Treeにはコンパイルの中間形などが置かれる。つまり、このツリーにあるファイルはキャッシュがあふれないかぎりサーバに書きだされない。ここにあるファイルは、メモリ上に作られ、ディスク・アクセスされずに消される。

４．２ キャッシュの管理

ＴｏｎＴｏｓではＳｐｒｉｔｅと同様にブロック単位のキャッシュをメモリ上におこなう。このキャッシュはそのファイルのＩＤ（ネットワークＩＤ＋ローカル・ファイルＩＤ）とそのファイル内のブロック番号からなる。このブロックはＬＲＵアルゴリズムに基づきおいだされる。また、間接参照を持つブロックのキャッシュも重要である。このファイル・システムも各ワークステーションが１００Ｍバイト以上のメモリをもっていることを前提としている。また、ディスクレス・ワークステーションを効率よくサポートするためResident Treeに関してはシステムのブート時にバルク・データ転送プロトコルを用いてローカル・サイトのキャッシュに転送される。また、ディレクトリの解析はＡｎｄｒｅｗ同様クライエントでおこなわれる。

ＴｏｎＴｏｓでのキャッシュの一貫性の維持は、ファイルへのライトはリードより少ないのを利用して、ライトよりリードのパフォーマンスを重視する。ＴｏｎＴｏｓではリード時はサーバにアクセスしない。しかし、ライト・オープンの時は必ずサーバへそれを知らせる。サーバはそれにより、キャッシュを持っているサイトを知っているのでこのときはすべてのサイトのキャッシュをストップする。また、このとき、キャッシュの内容が変更されていたらそれをフラッシュしてからキャッシュをストップする。この方法はリード時にはサーバへのアクセスが必要ないので効率が向上する。

４．３ ファイルの書きこみ

ＴｏｎＴｏｓではファイルの書きこみ戦略としてライト・バックを用いている。そして、Ｓｐｒｉｔｅと同様に定期的にダーティ・ブロックの書きだしをおこなう。これはNormal Treeに対してのみおこなう。Resident Treeの変更に関しては一貫性の管理など問題が残されているがこれについては今後の課題である。しかし、現在作成中のプログラムなどはエディティングとコンパイル、実行のくりかえしなので、このようなファイルは定期的に書きこむ時間を長くすれば効率がよくなる。しかし、クラッシュしたときは情報が大量になくなる可能性があるためこのへんのトレードオフが難しい。

４．５ トランスポート・メカニズム

ＴｏｎＴｏｓではトランスポート・メカニズムとしてＲＰＣを用いている。このＲＰＣはこのファイル・システム専用のもので、このために最適化している。つまり、ファイルのアクセスはオープンしてから、クローズするまで複数回のリード・ライトがおこなわれる。そのため、バーチャル・サーキットをベースにしたＲＰＣが好ましい。つまり、

このＲＰＣでは最初のリクエストによりコネクションが設定される。そのため、リクエスト、リプライのアックは次のリクエスト、リプライのメッセージにピギーバックすることが可能となる。このコネクションはサーバが一定時間リクエストがこないのを検出してコネクションを解放する。そして、次のリクエスト時に再び新たなコネクションを設定する。このようにすることにより、コネクションの設定時にパラメータをネゴシエートすることによりフローコントロールなどをおこなうことができネットワーク、及びサーバの負荷を軽くすることができる。

また、ワークステーションの立上り時にResident Treeをすべて転送するためバルク・データ転送プロトコルが必要である。これは、大量データ転送専用の効率のよいプロトコルである。

ＴｏｎＴｏｓでのサーバはＮＦＳ、Ｓｐｒｉｔｅ同様カーネル内のマルチ・スレッド・サーバとして作られる。

５　分散ファイル・システムから分散ＯＳへ

ＴｏｎＴｏｓはＵｎｉｘ上の分散ファイル・システムではなくＴｏｎＴｏｓ分散ＯＳ上の１コンポーネントとして最終的には作る予定である。この節ではＴｏｎＴｏｓ分散ＯＳの他の特徴について述べる。

５．１　仮想記憶

Ｕｎｉｘでは仮想記憶のスワップ・エリアはローカル・ディスク上にファイル・システムを介さず直接アクセスしている。しかし、ＴｏｎＴｏｓでは仮想記憶はファイル・システムを介してアクセスする。つまり、各プロセスのスワップ領域はファイルとして作られる。そのため、スワップエリアを容易にリモート・サイト上に作ることができるため、プロセス・マイグレーションの実現が容易となる。

５．２　プロセス管理

ＴｏｎＴｏｓでのプロセスはＵｎｉｘのプロセスと違いライトウエイト・プロセスをサポートしている。これはスタック・エリア以外の領域、及びオープン・ファイル情報などの各プロセスの環境もすべて共有しているプロセスである。また、プロセスがフォークするときの効率を向上させるためCopy-on-writeを用いている。

ＴｏｎＴｏｓのプロセス管理の最大の特徴はプロセス・マイグレーションのサポートにある。プロセス・マイグレーションで重要なのはプロセスが移動してもファイル・アクセスの環境が変わらないことである。プロセスがマイグレートされても、スワップエリアはファイルとして管理されるため分散ファイルを介して簡単にアクセスできる。このとき、マイグレーションを行なう前にすべてのダーティ・ページをスワップ・ファイルに書きだし、次にプロセスの環境をすべて移動先のサイトの転送する。そのとき、オープン中のファイルはリモート・サイトへそのファイルのＩＤや現在のオフセットを転送する。オープン中のテンポラリ・ファイルはマイグレーションがおきるとサーバに書き出され、新しいサイトはそれをサーバから読み出す。また、新たにオープンするファイルはNormal Treeに関しては移動前のサイトのものをアクセスする。Resident Tree、Temporary Treeに関しては移動先のものが用いられる。

５．３　ＩＰＣ

分散ＯＳではネットワーク透過のＩＰＣが非常に重要である。ＴｏｎＴｏｓではネットワーク透過なポートを用いてこれを実現している。これはＭａｃｈ［４］で用いられているものとほとんど同じである。これを用いることにより各プロセスは相手プロセスが存在するサイトを知らずに通信することが可能となる。

６　結論

本論文では現在、稼動中の３つの分散ファイル・システムについて比較、検討をおこなった。そして、これらの欠点を改良したＴｏｎＴｏｓ分散ファイル・システムについて提案した。これは我々が現在設計中の分散ＯＳ：ＴｏｎＴｏｓの１コンポーネントである。そして、最後にＴｏｎＴｏｓ分散ＯＳについて簡単に述べた。

[1] R.Sandberg et al. "Design and Implementation of Sun Network File System" Proc. Summer USENIX 1985

[2] J.H.Morris et al. "Andrew: A Distributed Personal Computing Environment" CACM Vol 29 No 3

[3] M.Nelson et al. "Caching in the Sprite Network Filesystem" Proc. SOSP 1987

[4] M.Accetta et al. "A New Kernel Fundaton for Unix Development" Proc. Summer USENIX 1986