キャッシュノード機能実装によるＮＦＳの大規模化とその応用

田胡　和哉、小柳　順裕、山下　直人

（東京工科大学）

NFS ファイルシステムを対象として、ファイルキャッシュ ノードを実現するカーネル拡張モジュールを実現した。これにより、従来より多くのクライアントを接続することができるようになるとともに、キャッシュ記憶媒体としてディスク装置等の二次記憶装置を利用すれば、マルチメディアコンテンツ等の大容量ファイルも通常ファイルと区別なく分散環境で扱えるようになる。実装は、おもに、VFS のキャッシュ機能を拡張するためのモジュール、NFS の再エクスポートを可能にするためのモジュール、および、性能最適化を行うためのデーモンから構成されている。実現した大規模分散ファイルシステムを、ディスクレス PC 環境の実現に利用するとともに、これに基づいた大規模 CMS の構築を行っている。実装の詳細、および、利用法について述べる。

１．まえがき

ネットワークの大規模化と利用形態の多様化が進んでいる。企業や教育機関でのイントラネットをおもな対象として、NFS を大規模化することによるネットワークサービス基盤を構築した[1]ので、報告する。

NFS の大規模化の手段として、ＮＦＳサーバノードとクライアントノードの中間に、キャッシュノードを挿入する方法をとる。キャッシュノードは、ＮＦＳサーバノード（以下　サーバ）のファイルイメージを、メモリやハードディスクに一時蓄えるとともに、ＮＦＳクライアントノード（以下　クライアント）に対しては、通常のサーバとして動作する。単一のサーバに複数のキャッシュノードを接続することによって、従来より多くのクライアントを接続することが可能になる。特に、ある組織の全部のクライアント間で単一のファイルシステムを共有できるようになることによる利点は大きい。

キャッシュノードにおける記憶媒体として、ハードディスク等の二次記憶装置も利用することにより、サイズの大きなファイルや、多数のファイルをキャッシュすることができるようになる。これによって、たとえば、大規模なコンテンツ管理システム（ＣＭＳ）を構築したり、ディスクレスＰＣを多数運用したりすることが可能になる。

キャッシュノードの実装は、単純ではない。キャッシュノードは、サーバのファ

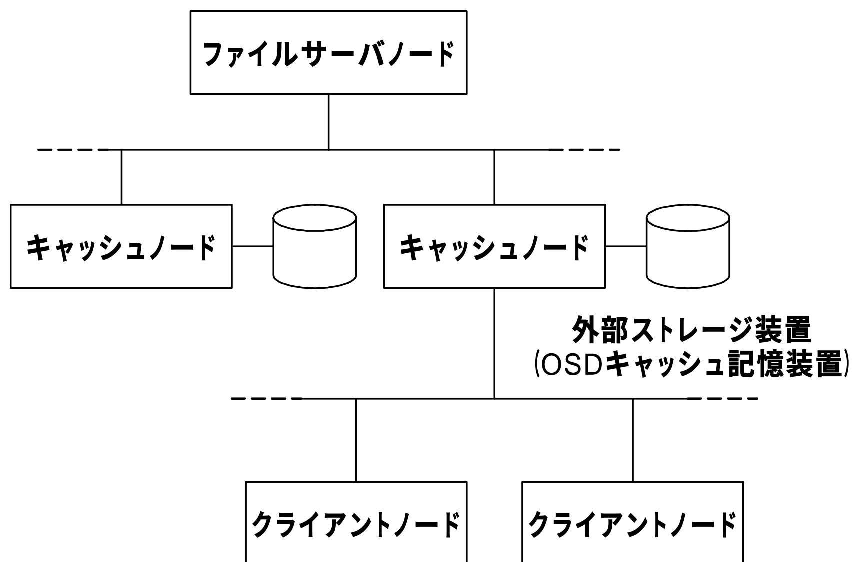

図１　**Community Storage** のシステム構造

イルを、ＮＦＳプロトコルによってインポートした後にクライアントに対して再度エクスポートすることによって実装される。従来、ＮＦＳの実装はこのような再エクスポート機能を実装していなかったので、これを新たに実現する必要がある。また、キャッシュノードにおいては、**Linux** の **VFS** 実装による標準のファイルキャッシュ機能に加えて、**EXT2** 等の既存のファイルシステムをファイルキャッシュのための記憶媒体として利用する機能が新規に実装されている。また、キャッシュ間連携のような、高度な性能最適化機能をインクリメンタルに実装できるようなフレームワークを備えている。

実装の内容は、

１）再エクスポート機能実現のためのＮＦＳ実装の変更と、新規カーネルモジュールの実装

２）**VFS** キャッシュのための記憶媒体として通常のファイルシステムを利用する機能を実現し、かつ、**VFS** の実装には変更を加えないことを可能にする新規カーネルモジュールの実装

からなる。すでに **Ver.1** の実装が完了し、ディスクレスＰＣを運用することが可能になっている。この大規模ＮＦＳを **Community Storage** と名づける。現在、**Community Storage** のプログラムをオープンソースソフトウェアとするための準備を行っている。

Community Storage を用いた、アプリケーションの開発も行っている。単一の大規模分散ファイルシステムを用いてネットブート型のディスクレスＰＣを運用することによって、システム管理が著しく容易化する。すなわち、アプリケーションインストールやパッチ適用等のクライアントシステムのメンテナンスは、システム全体で一回実行すれば完了することになる。ま

た、ファイルバックアップ等のストレージ管理も集中化できる。これは、ＴＣＯ削減に有効であることが期待できる。

さらに、大規模に運営できるＣＭＳを構築し、グループウェアとして利用することを検討している。コンテンツ自体の共有、配信機能はファイルシステムが担うので、ＣＭＳはメタデータの管理のみを実現すればよくなる。このため、大規模システムも比較的容易に実現できる。メタデータの管理基盤としてＪＣＲ(Java Content Repository) API[2]、ビューワとしてMozillaフレームワークを用いたシステムの実現をすすめている。これについても簡単にふれる。

２．NFSファイルシステムの概要

既存の NFS の原理と実装の概要について述べる。NFS分散ファイルシステムは、Sun社によって開発され、Unixシステム用の分散ファイルシステムとして広く利用されてきた。最近では、V4[5]の仕様策定が行われ、大きく発展している。

Linux における実装では、最近のカーネル版で NFS V4 が実用的になりつつあり、さらに改良が続けられている段階である。

NFSは、多くの分散ファイルシステムのなかでももっとも単純な設計をとっており、その簡潔さが発展の要因となっている。一方において、いくつかの原理的な制約を持つ。

現在も広く用いられている NFS V2,V3 は、いわゆるステートレスなアクセス方式をとっている。ファイルのオープン時にファイル名をファイルハンドルに変換した後は、ファイル　ページのアクセスごとにオープン　クローズが行われ、ファイル　アクセス全体としてのオープン　クローズ処理は行われない。したがって、ファイルハンドルとファイル名の間の対応関係の有効期間に関する明確な規定がない。これは原理的な制約であり、NFSでは、実装上の工夫だけではインポートしたファイルを再度エクスポートすることが困難であるとされてきた。NFS V4の規定では、大きく設計方針が変わり、クライアントとサーバの間で状態を共有する、ステートフルなプロトコルになっている。

クライアントキャッシュの運営においても、NFS V2,V3の規定では、コヒーレンシの維持の方式が明確にされていない。一方、V4ではdelegation/recall機構によって、完全ではないものの、大きく改善されている。

今回のキャッシュノードでの実装では、ディスクレス PC の運用の必要性から、NFS V2,V3 クライアントが利用できることを目標とした。一方、キャッシュノード間のコヒーレンシの維持の目的から、サーバノードとキャッシュノード間はNFS V4を用いている。

３．Community Storageシステムの構成

図1に、Community Storageシステムの全体構造を示す。ファイルサーバノードと、キャッシュノードが分散ファイルシステムとしてのサービスを提供し、これにクライアントノードが接続されている。クライアントノードからは、１台のファイルサーバと多数のキャッシュノードからなるシステム全体が１台の NFS ファイルサーバ

であるかのように見える。

ファイルサーバノードとクライアントノードの構造は、通常のシステムと変わらない。ここでは、キャッシュノードの内部構造について述べる。

図 **2** に、キャッシュノードの内部構造を示す。キャッシュノードは、**Linux** で実現されており、これに、キャッシュノード用のカーネルモジュールと、ユーザモードで動作するデーモン **(Scheduler)** 、および、ストレージを追加することによって構成される。両者は、仮想デバイスを利用したメッセージング機能を利用して連携して動作する。**Scheduler** ではキャッシュノード間のネットワークスケジューリングを行う。

キャッシュノード用のカーネルモジュールは、**Data Path Switch (DPS)** とよばれる。これは、

1) **VFS** の動作を変更することによって、個々のファイルごとに、キャッシュ用のファイルを割り付け、これを利用してキャッシュ動作を行う
2) サーバノードのファイルを **NFS** プロトコルによってインポートした後に、再度 **NFS** プロトコルによってエクスポートする
3) クライアントノードからのファイルアクセスパターンのログを取得する

機能を持つ。ファイルごとのキャッシュ媒

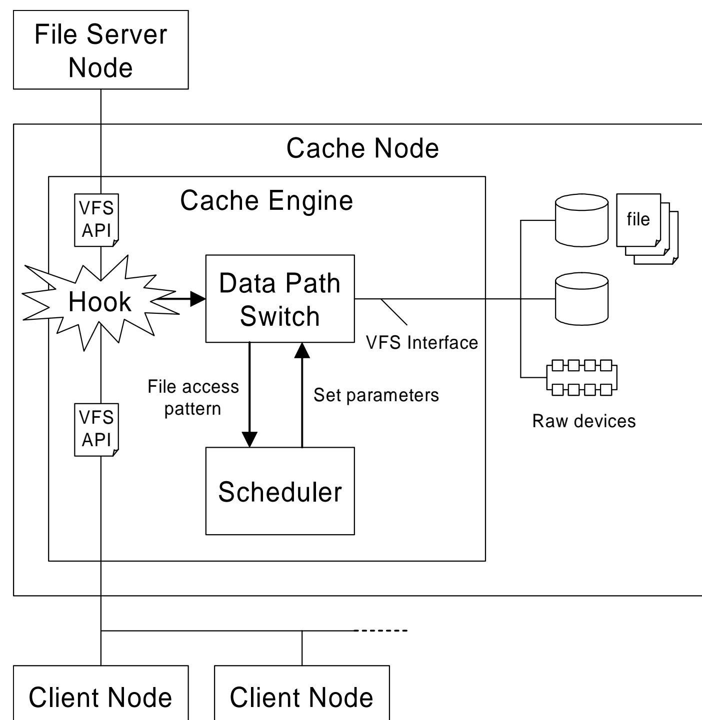

図２　キャッシュノードの構造

体も、それぞれ別個のファイルを用いるので、キャッシュ用の記憶媒体として、主記憶上のファイルシステム（**RAM** ファイルシステム）、通常のファイルシステム、**OSD**[4]等を自由に選択することができる。**DPS** は、キャッシュ動作中にキャッシュ記憶媒体を変更する機能、サーバノードへのライトバックの速度を調節する機能、サーバノードからのプリフェッチの速度を調節する機能、ライトバックの際の、キャッシュイメージのバージョンを管理する等を持つ。また、**NFSv4** プロトコル[5]で規定されている **Delegation/Recall** 機能に対応したキャッシュのライトバック、無効化機能を持つ。

キャッシュノードには、ストレージが付加される。ファイルキャッシュに用いる記憶媒体として、ハードディスクによる通常のファイルシステムのみならず、**OSD** も用いることができるようにする。**OSD** は、記憶装置にファイルシステム機能の一部をオフロードし、セクタ単位ではなく、ファイル単位で記憶装置にアクセスするための外部記憶アクセス規約である。

４．DPS の構造

DPS の実装の詳細について述べる。**Linux** のファイルシステムは、**VFS** システムを共通インタフェースとして構成されており、ファイル　ページの主記憶上でのキャッシング、および、ディレクトリシステムのキャッシュングを共通化して実装している。

Linux のファイルシステムを実装する重要なデータ構造として、**inode** と **dentry** がある。ファイル実体のおのおのに **inode** が割り当てられ、状態を管理している。また、ファイル名の各々に **dentry** が割り当てられている。これらのメタデータは、**VFS** によって主記憶上でキャッシュされている。また、ファイル　ページは、ページ管理システムによって主記憶上にキャッシュされている。

Linux カーネルの実装は、きわめて頻繁に変更が加えられる。キャッチアップを容易にすることを目的として、**Community Storage** では、**VFS** 実装に変更を加えることなく、ファイルキャッシュ機能を一般的な方法で拡張することを試みている。このための、**inode** の、**i_op** フィールドにキャッシュのための構造体を付加することにより、**inode** 構造体自体の定義を変更することなく機能拡張を行った。これを利用することにより、**inode** に新たな付加情報を定義することが可能になる。

inode の状態に関して、外部キャッシュ中であることを示す付加情報を新たに追加した。外部キャッシュ状態とは、**VFS** に本来実装されているメモリページを用いたキャッシュとは別個に、**VFS** インタフェースを経由して別個のファイルを用いたキャッシュを利用中であることを示す。外部キャッシュは、独自に先読み、後書き機能を持ち、ファイル内容全体を別個のファイルシステムを用いて保持する機能を持つ。外部キャッシュがヒットした場合には、本来のファイルではなく、外部キャッシュのファイルに対して入出力が行われる。このような、動作変更を実装するために、外部キャッシュ状態中は、**NFS** の **inode_operations**、および、**file_operations** は別のものに置き換えられる。

DPS がこのようなフック機能を実現す

る。DPSは、Schedulerと仮想デバイスを通じたメッセージング機構によって接続されている。外部キャッシュを開始する際、対象となるファイルシステムのルートinodeをSchedulerから渡されると、DPSはそのsuperblockのsuper_operationsを置き換えるとともに、ルートinodeのi_opフィールドを書き換えて付加情報を追加する。これによって、以後のファイルシステムの動作において、すべてのファイルについてそのキャッシュ動作を変更することが可能になる。

inode構造体のライフサイクルは、Schedulerがスケジューリングを行う。外部キャッシュ中のファイルのinodeは、アクセスされることがなくてもカーネル内から削除されることがないよう、状態ビットが立てられている。アクセスされない期間が長くなると、外部キャッシュのライトバックを完了した後、この状態ビットを降ろす。これによって、通常のinodeキャッシュのライフサイクルにしたがって、メモリ不足時にはinodeが削除され、外部キャッシュも同時に消滅する。

サーバファイルのオープン時にdelegationの状態を確認し、delegationが得られれば外部キャッシュを開始する。外部キャッシュ中にrecallイベントが起きると、外部キャッシュをライトバックして外部キャッシュを終了する。

５．NFSの再エクスポート

Linux のNFS実装において、インポートされたファイルを再度エクスポートできるようにする方法と、解決する必要があった問題点について述べる。

一般に、ファイルシステムをNFSを用いてエクスポートできるようにするためには、superblockにexport_operationsを定義する必要がある。したがって、今回の実装においても、NFSのsuperblockにexport_operationsを定義すればよいことになる。これによって、比較的容易にファイルがエクスポートできるようになる。しかしながら、inodeおよびdentryのキャッシングに関する原理的な問題があり、これだけでは永続的に運用することができない。

Linuxのカーネルでは、長期に参照されないinodeおよびdentry構造体は、カーネルのアドレス空間から消去される。その一方において、NFS V3では、いわゆるステートレスなファイル アクセス方式をとっており、ファイル名のルックアップをせずにファイル　アクセスを開始する場合がある。そのため、上記のメタデータのキャッシュ機構から一度廃棄されたinodeやdentryを、通常の手順で再ロードすることなく再利用が開始される場合がある。このため、inodeやdentryをファイルハンドルを識別子として再ロードする機構が必要になる。特にdentryの再ロードは本来のNFSのプロトコルにはない機能であり、カーネル外の機構を援用した拡張が必要になった。

Community Storageの実装では、dentryのキャッシュを削除する際、そのディレクトリ名とファイルハンドルをデーモンに伝達する。デーモンは、この情報をキャッシュとして保持する。デーモンは、通常のユーザモードで実行されており、仮想空間上で動作する。このため、デーモンを用いてカーネルの実記憶上にあるdentryキャッシュを、仮想記憶上に拡張したこと

になる。これらのキャッシュにも存在しないファイルが参照された場合には、デーモン間の通信機構を利用し、ファイルハンドルを引数としてそのファイル名をサーバに問い合わせる。

dentry の再ロードは、**export_operations** の **get_dentry** 関数の実装で必要となる。キャッシュノードの **NFS** サーバ　モジュールがクライアントからのファイル入出力要求に答えるためにファイルハンドルから **dentry** を得ようとするとき、カーネル内の **dentry** のキャッシュを探索する。**dentry** がキャッシュ中にないときは、デーモンの再ロードを要求する。**dentry** の再ロードは、単に、デーモンがそのファイルをオープンするだけで実行できる。デーモンは、ファイルハンドルとファイル名の対応表を保持しているので、これを探索してファイル名を得、これを通常の方法でオープンする。キャッシュノードの **NFS** クライアント　モジュールは、通常の手順でファイル名のルックアップ処理を行う。この過程で関連する **dentry** はすべてカーネル内のキャッシュにロードされることになる。これが完了した後、再度ファイルハンドルから **dentry** を得る処理を実行すれば、**NFS** サーバ　モジュールは求める **dentry** を得ることができる。

キャッシュノードとサーバ間では **NFS V4** プロトコルを用いる。一方、キャッシュノードとクライアント間では **NFS V3** もしくは **V2** プロトコルを用いる。**V2,V3** プロトコルはステートレス型であるので、ファイル　ページをリクエストするたびにオープン処理を行う。一方、**V4** プロトコルはステートフル型であるので、オープン処理によってステートを作成する処理が必要になる。実際には、**V4** プロトコルの **delegation** 機能によって、オープン処理を多数回繰り返すことによるオーバヘッドは大きくないが、**NFS** の実装上、多少の変更を必要とした。

６．Scheduler

Scheduler は、１秒ごとに、**DPS** の詳細な動作状態を仮想デバイスを経由したメッセージの形式で受け取る。この情報に基づいて、ログの作成、および、スケジューリングを行う。現状では、**inode** のライフサイクルのスケジューリングを行っている。また、将来、キャッシュ間連携に関するスケジューリングもスケジューラを用いて実現する予定である。

多岐にわたる機能を系統的に実現できるようにするために、スケジューラは、**C++** を用いたフレームワーク構造をとっている。フレームワークは、**DPS** において外部キャッシュ状態にある個々のファイルに対応して、オブジェクトを生成する。このオブジェクトのプロパティによって、対応するファイルの統計情報を得られるようになっている。統計情報として、アクセス回数、キャッシュ状態、ファイルサイズ、ダーティ部分のサイズ等がある。また、このオブジェクトのメソッドをよびだすことによって、**DPS** にコマンドを発行し、対応するファイルのキャッシュ状態を変更することができる。外部キャッシュ状態が終了すると、自動的に削除される。すなわち、このオブジェクトは、**DPS** によってカーネル内にキャッシュされているファイルの状態を、ユーザプログラムレベルで１対１に反映してい

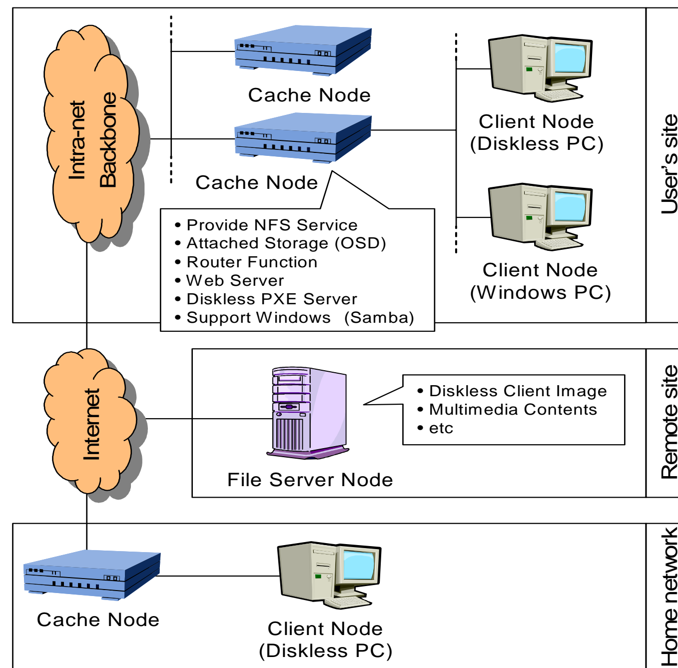

図３　**Community Storage** によるシステムの例

る。このオブジェクトの機能を、継承によって拡張することによって、キャッシュ機能に関する種々のスケジューリングが、ユーザモードで動作するデーモンによって行えるようになっている。

７．アプリケーション

現在、**Linux** カーネル **2.6.14** を対象として、**Community Storage** の **Ver.1** が実装されている。これを利用して、

１）　ディスクレス **PC** の運営
２）　**CMS** の構築

を行っている。また、研究室環境に実際に適用することによって、実証実験を行っている過程である。現在の実証環境は、**IBM** 社製の **RAID** によるサーバと３台のキャッシュノードから構成されており、２箇所のオフィスで運営されている。また、家庭にあるキャッシュノードから利用できるように準備を行っている。図３にシステムの構成例を示す。

７．１　データ管理機構としての利用

ネットブート型のディスクレス **PC** を大規模分散ファイルシステムを用いて運用することによって、大規模な組織体におけるクライアント **PC** の管理コストを大幅に削減することが可能になると期待される。また、利用者の観点からもメリットが大きい。サーバ上には最大規模の **Linux** ディストリビューションがインストールされ、常に最新の状態に管理維持される。クライアントでは、そのなかから必要なアプリケーションのみが自動的にダウンロードされることになる。これによって、まったく労力を払うことなく必要なアプリケーションを自由に利用することができるようになる。

また、データ管理の観点からも、分散ファイルシステムを利用することは有効である。利用者のすべてのファイルは最終的にサーバに集約される。したがって、サーバにおいてデータのバックアップをとったり、バージョン管理を行ったりすることによって組織全体のデータ管理が行えることになる。また、キャッシュノードはキャッシュとして動作するので、ハードディスク容量が不足しても性能が低下するだけで、動作を継続することができる。これは、従来多大な労力をともなっていた、分散環境における容量管理作業の負担を軽減できることを意味する。

この環境は、ブロードバンド通信網を利用することによって、家庭や遠隔サイトにも拡張することができる。現在、家庭に安価なキャッシュノードを設置する準備を行っている。これによって、家庭で職場の計算機環境をそのまま利用できることになる。また、データセキュリティの観点からも有効である。また、海外にオフィスを拡張する際にも同様の方法をとることができる。

ネットブート型の環境は、通常の PC のみならず、組み込み機器やゲーム機の利用においても有効である。組み込み機器では、コストの点から大きな二次記憶装置を備えることができない。分散ファイルシステムを用いて二次記憶装置の集約化を行うことによって、組み込み機器の二次記憶装置容量の削減、ひいては、コスト削減に有効である。また、ソフトウェアのメンテナンスの点からもきわめて有利である。

同様のことは、ゲーム機にも言うことができる。ゲーム機は、同一仕様で大量に生産されるので、その機能のわりに安価である。Linux が動作するクライアント機として魅力的である。ディスクレスでゲーム機を運用する環境として Community Storage を利用する。

ディスクレス環境の今ひとつの興味深い利用法として、Xen 等の VM との併用があげられる。Xen を Community Storage 上で動作させることにより、大規模分散環境で VM を移動することを検討している。

７．２　CMS 基盤としての利用

組織体における情報がデジタル化されるにしたがって、その管理が重要な課題となりつつある。Community Storage の今ひとつの利用目的として、Content Management System(CMS)があげられる。具体的には、コンテント　ファイルを蓄積し、どこからでもアクセスできるようにする基盤として Community Storage を利用する。これに、メタデータの分散管理機構を付加すれば、大規模に運用可能な CMS を構築することができる。メタデータ管理機構として、JCR（Java Content Repository）API の大規模分散実装と、その上で動作するコンテント　データベース管理システムを構築している。クライアントで動作する GUI 部分として Mozilla をもちいる。Mozilla は、そのフレームワークとして、XUL とよばれる XML ドキュメントを用いてカスタマイズすることにより、高機能のコンテント表示、作成システムを構築できる。さらに、そのメタデータは、XML を用いて交換することができるようになる。現在、これらの方針にもとづいて、Raptor とよばれる CMS システムを構築中である。

８．オープンソース化

６月末を目処として、**Community Storage** のオープンソース化の準備を行っている。**Mozilla** をベースとした **CMS** ビューワとあわせて、**OSCJ.net** を通じて公開する予定である。プロジェクトの概要は、**http://www.teu.ac.jp/t-lab/Projects.html** に掲載されている。

９．あとがき

大規模分散ファイルシステム **Community Storage** の実現方式の詳細、および、利用方法について述べた。現在実証実験の過程にあり、種々の拡張作業を行っている。実環境での利用に基づく評価を行う予定である。結果について再度報告を行いたい。

参考文献

[1] Koyanagi,M., et. Al.:プラグイン方式によるファイルキャッシュ記憶向けリソーススケジューラの実現, 情報処理学会システムソフトウェアとオペレーティング・システム研究会報告、OS-101 (2006).

[2] Java Community Process: JCR170: Content Repository for Java Technology API, http://www.jcp.org/en/jsr/detail?id=170

[3] Mozilla : XUL User Interface Language, http://www.mozilla.org/products/xul/

[4] Ralph O. Weber, Object-Based Storage Device Commands (OSD), http://www.t10.org/ftp/t10/drafts/osd/osd-r10.pdf

[5] B. Callaghan and D. Robinson and R. Thurlow and Sun Microsystems, Inc. and C. Beame and Hummingbird Ltd. and M. Eisler and D. Noveck and Network Appliance, Inc., RFC3530, http://www.rfc.net/