情報処理振興事業協会

平成１３年度未踏ソフトウェア創造事業
（新部プロジェクト･マネジャー）

Linux 版ネットワーク RAID ファイルシステムの実用化

成 果 報 告 書

平成 14 年 2 月

松 本　尚
⟨tm@is.s.u-tokyo.ac.jp⟩

情報処理振興事業協会
平成12年度未踏ソフトウェア創造事業
Linux 版ネットワーク RAID ファイルシステムの実用化
要旨

a. プロジェクト名

『Linux 版ネットワーク RAID ファイルシステムの実用化』

b. 開発目的

ローカルエリアネットワーク（LAN）環境を構成する計算機のローカルディスクのうち遊休資源となっているものを有効活用した、低コストで高信頼分散共有ファイルシステムを実現する実用レベルのネットワーク RAID ファイルシステム（NRFS）を Linux 上で開発する。

c. 開発期間

平成13年7月2日～平成１４年2月２８日

d. 実施体制

担当プロジェクト・マネジャー	新部　裕
開発者	松本　尚 (東京大学大学院情報理工学系研究科)
プロジェクト実施管理組織	三菱マテリアル株式会社

e. 開発内容及び成果

(1) 開発内容

(a) マシン障害･ネットワーク障害に対応した NRFS クライアント
(b) NRFS サーバにおける実用版障害復旧機構
(c) RPC の非同期タスクの使用による高速化
(d) ディレクトリチェックサムキャッシュによる高速化
(e) 1000BASE-SX,100BASE-TX 接続による性能試験および負荷試験

(2) 成果

(a) マシン障害･ネットワーク障害に対応した NRFS クライアント
2001 年 11 月より完成しており、2002 年 2 月現在問題は見つかっていない。
(b) NRFS サーバにおける実用版障害復旧機構
2002 年 1 月末より完成しており、2002 年 2 月現在問題は見つかっていない。
(c) RPC の非同期タスクの使用による高速化
クライアントに実装され、複数サーバへの RPC 実行がオーバラップされている。
(d) ディレクトリチェックサムキャッシュによる高速化
サーバに実装され、性能向上が達成された（詳細は本文内)。
(e) 1000BASE-SX,100BASE-TX 接続による性能試験および負荷試験
１ヶ月以上にわたる負荷試験および性能試験を実施した。NRFS の高信頼性高可用性機能は達成されている。ただし、性能試験結果に関しては若干説明がつかない点が残っている。

f. 今後の課題

NRFS の開発は平成１２年度の基本部分の開発に引き続いて行われた。平成１２年度に開発のベースとして Linux 2.2.16 カーネルを採用したため、平成１３年度もこのカーネルバージョンで開発を進めた。NRFS を普及させるためには、最新のバージョンである Linux 2.4.x に対応することが必須である。ただし、両バージョン間には NFS や RPC の実装に大きな差異が見られるため、対応作業は簡単ではない。高信頼性と高可用性を機能として実現できたと自負しているが、運用の容易性が実現できているとは言い難い。一般ユーザでも取扱い可能な運用形態やサポートツールを作成提供する必要がある。

デバッグと試験が開発期間いっぱいにかかってしまったため、期間内に NRFS のリリースができなかった。この点に関しては、緊急の課題として年度内にはアルファリリースを行う予定である。

1.目的及び背景

パーソナルコンピュータおよびワークステーションの価格性能比の改善と低価格化は著しく、オフィスや工場や研究所はもちろん家庭やSOHOにも複数台の計算機が設置されるようになってきている。現在の計算機はアプリケーションや操作環境整備のためプログラムおよびデータの格納場所としてハードディスク装置を使用している。複数台の計算機を使用する場合に、分散共有ファイルシステムによってハードディスク装置を共有できると、一台ごとにディスク内容を整備する必要がないためにマシン管理コストが大幅に低減できる。また、計算機間のデータの移動が共有ファイルによって自然に行われるため、テープやフロッピーディスクを介したデータ移動よりも大幅に手間が少ない。また、明示的に通信プロトコルやファイル転送プログラムによってデータ転送するよりも簡単である。分散共有ファイルシステムでは十分に高速なプロトコルを使用して多くのシステムではデータ転送に必要な時間コストも十分に小さく抑えられている。これらの利点により、複数の計算機を導入している環境では、分散共有ファイルシステムが導入されている。クラスタシステムでは各ノードにおけるデータの共有は、システムの使い勝手を向上させるのに必須であるため、分散共有ファイルシステムが必要不可欠である。逆に、疎粒度で分散並列実行可能なアプリケーションであれば、分散共有ファイルシステムのみを通信同期手段としてクラスタを運用することが可能である。

分散共有ファイルシステムは他のマシン上のハードディスクを仮想的に使用可能にする機構であるため、プログラムやデータは最終的にハードディスク装置内に保存される。ハードディスク装置は物理的に高速稼働する部分が多いために発生する装置自身がクラッシュする確率や、非常に高密度の磁気記憶を行っているため誤り訂正不能なデータ化けが発生する確率が無視できない大きさで存在する。重要なデータや時間をかけて整備した操作環境をハードディスク装置の故障から安全に守ることもシステムとしては非常に重要である。また、安価なハードディスク装置では、装置内に内蔵されている半導体メモリによるキャッシュ機構にエラー訂正機能がなく、このレベルでデータ化けを起こすといったトラブルも存在する。ハードディスク装置上データの安全性向上のために近年普及しつつあるのが、RAID (Redundant Array of Inexpensive Disks)装置である。RAID装置では安価なハードディスク装置を複数使用して、プログラムやデータを冗長な形で格納しておき、1台(もしくは少数台）のディスク装置が故障しても正しい情報が復元可能なディスク装置である。また、複数のディスク装置を同時にストライピングアクセスすることにより、ディスク装置としての性能も単体ディスクよりも向上させることが可能である。

RAIDにおける冗長なデータ形式への変換方式として、ミラーリング方式とパリティ方式が主に使用されている。ミラーリング方式はデータ処理に負荷をかけないことを重視した方式であり、ディスク装置に格納すべきデータの完全なコピーを他のディスク装置にも格納する（図1）。パリティ方式はディスク容量の有効活用を指向した方式であり、例えば2048byteのデータブロックを512byteずつ4つのサブブロックに分け、さらにサブブロック間の排他論理和による512byteのパリティを計算し、5台のハードディスク装置に分散して格納する方式である（図2）。パリティを格納するディスクをデータブロック毎に変えて分散させる方式もある。元データもしくはパリティを格納したディスク装置が故障しても、故障が1台であれば、他のディスク上のデータから故障したディスクのデータが復元できる。

ミラーリング方式のRAIDはディスクに格納するデータに関して変換をかけていないため、メインCPUのソフトウェアによる実現が低コストで可能であり、ソフトウェアで実現されたシステムも多く存在する。これに対して、パリティ方式のRAIDはパリティ生成を行うオーバヘッドが大きいため、ソフトウェアレベルの実装では性能が低下してしまう。ミラーリング方式によるソフトウェアRAIDを採用したとしても、通常のSCSIやIDEで接続されたディスクに対してシステムを停止させずにディスク交換を行うこと（ホットスワップ）は不可能である。これらの理由から、可用性と信頼性が求められる環境では、RAID装置は専用ハードウェアによって構成され、結果として高価なものとなっている。

本開発では高価な専用RAID装置を購入することなしにソフトウェアのみで低コストかつ効率よく高信頼分散共有ファイルシステムをLinux上に提供することを目的にしている。

1.1. 平成１２年度未踏ソフトウェア創造事業との関係

平成１２年度**未踏ソフトウェア創造事業（長谷川正治 PM）**において「ネットワーク RAID ファイルシステムの開発」として NRFS プロトタイプを開発した。NRFS プロトタイプは発生する障害をデータ化けとファイル消失・ディレクトリ消失に限定し、誤り訂正機構をクライアント主導の大幅に簡略化されたものにして、開発期間を短縮させた NRFS の簡易バージョンである。NRFS プロトタイプの開発に当たって、Linux の NFS,VFS, RPC の徹底した解析を行った。Linux 実用版 NRFS はこの NRFS プロトタイプと Linux に関する知識をベースに開発された。平成１３年度の主な開発課題は、マシン障害とネットワーク障害に対応可能にすること、ファイル属性が変化するようなファイル障害に対応すること、障害回復手続きの高信頼化ならびに高速化、Gigabit Ethernet カードを使ったチェックサム計算の高速化である。そして、実用化のための試験運用（公開アルファテスト）を行うことである。

1.2. 平成１３年度開発体制

平成13年度の開発は開発者の松本が基本方式設計を行い、詳細設計に関しては委託開発先のメンバと平均隔週５時間の技術ミーティングにおいて十分な議論をしながら、詳細設計を固めた。その結果を元に、手分けして実装を行った。委託先として開発に参加したのは以下の二社である。

- 三菱マテリアル株式会社
- 株式会社フューチャーテクノロジー

三菱マテリアルは前年度にLinuxのNFS, VFS, RPCの調査解析を担当した関係もあり、主にLinuxのソースに関連が深い部分の開発作業を行い、フューチャーテクノロジーは通信デバイスに近い部分の開発を行った。委託開発参加者ののべ人数は6名を数えるが、その中に三菱マテリアルの**黄強さん**と**鏑木健二さん**の2名が含まれていたことが開発者にとっては非常に幸運であった。この二人がいなければ、Linuxフリークでも Linuxプログラマではない開発者に今回のLinux版NRFSを動かすことは非常に困難だったと思われる。ここに感謝の意を表したい。

2．期待される効果

複数（３台以上）の PC もしくは WS を持っていれば高価な RAID 装置を買うこと無しに高信頼分散共有ファイルシステムであるネットワーク RAID ファイルシステム（NRFS）を使用することができるようになる。訂正可能な障害はファイルシステムが自動的に修正し、ディスククラッシュのようなハードウェア交換が必要な障害に対してもファイルシステムは停止することなく交換作業を行うことが可能である。マシンレベルの冗長性を持っているため、故障ノードの電源を切断して修理作業を行ってもファイルシステムを運用できる（システムレベルのホットスワップ)。そして、故障ノードを通電後、ソフトウェアによって自動的に交換されたディスクのファイルシステムを無故障状態に復帰させる。つまり、データのビット化け等を自動修正し、ディスククラッシュ時もファイルシステムが停止することなく運用でき、ディスククラッシュからの復旧に莫大な作業時間を取られる必要がなくなる。このうちのディスククラッシュに関する NRFS のメリットは既存の RAID 装置でも得ることができるメリットであるが、ホットスワップ可能な RAID 装置はまだまだかなり高価である。データ化けに関しては既存のＲＡＩＤ装置ではディスクドライブに起因するデータ化けは修正可能であるが、コントローラに関するデータ化けは検知できない。

以上に述べたように低投資コストで高信頼分散共有ファイルシステムが構築できるため、今まで以上に多くのプログラマおよびエンジニアがディスクトラブルに対する復旧作業から開放され、本来の業務に専念することができる。

ファイルサーバ用のマシンを新しいマシンに交換する場合も、一台ずつ交換することにより、自動的にファイルシステムの中味を引き継ぐことができる。この様に低コストかつ便利でメンテナンスが容易な NRFS が普及すれば、マシンを買い替える度にローカルディスク上のアプリケーションをすべてインストールしなおす必要のある欠陥オペレーティングシステムによって引き起こされる無駄な作業からも人々を開放することができる。

Gigabit Ethernet レベルの高速 LAN 上において NRFS を使用することにより、SAN（Storage Area Network）として同様な運用も可能となる。NRFS のベースとなる NFS 自体が分散共有ファイルシステムを実現しており、さらに NRFS は高信頼性を付加しているため、NRFS を高速ネットワークと共に使用することで、高価な SAN 用ディスクサブシステムを導入する必要がなくなる。ディスク装置のアクセススピードがシステム全体のボトルネックになるような巨大システムの場合には、多数決誤り検出訂正方式とストライピング方式を併用して性能を向上させることができる。

3．ネットワーク RAID 方式の原理

3.1．計算機アレイによる RAID

分散共有ファイルシステムは複数の計算機によって共有されるため、ディスク故障等によってシステムがダウンすることの影響が非常に大きい。分散共有ファイルシステムには高い信頼性と高い可用性が不可欠である。このため、分散共有ファイルシステムのディスク装置として専用ハードウェアを搭載したRAID装置を導入して信頼性を向上させる方式が一般的になりつつある。しかし、この場合はRAID装置が接続される計算機（ファイルサーバ）にも信頼性と性能が要求され、システムのコストが高くなってしまう。本稿では安価になった普及型のパーソナルコンピュータやワークステーションを活用してファイルシステムの信頼性を向上させる方式を提案する。つまり、RAIDは安価なディスク装置を複数台使用することによって冗長性を増して信頼性を上げていたのに対して、ネットワークRAIDでは安価な計算機を複数台使って冗長性を増して信頼性を向上させる（図3）。

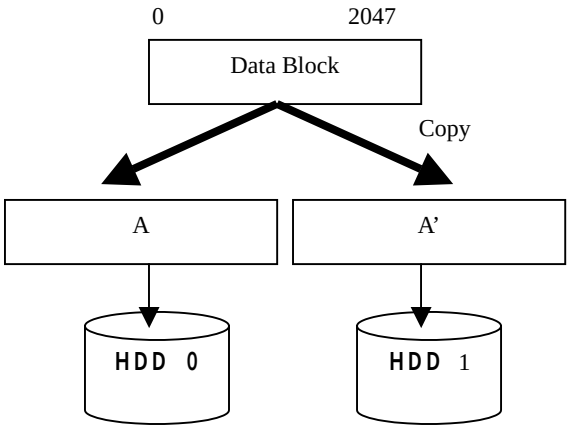


図1 ミラーリング方式

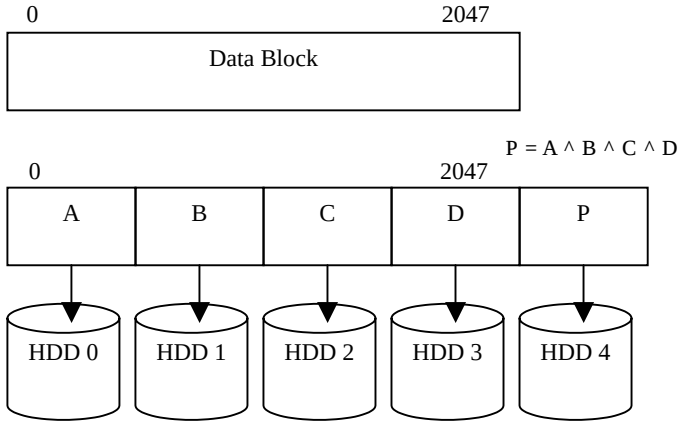


図2 パリティ方式

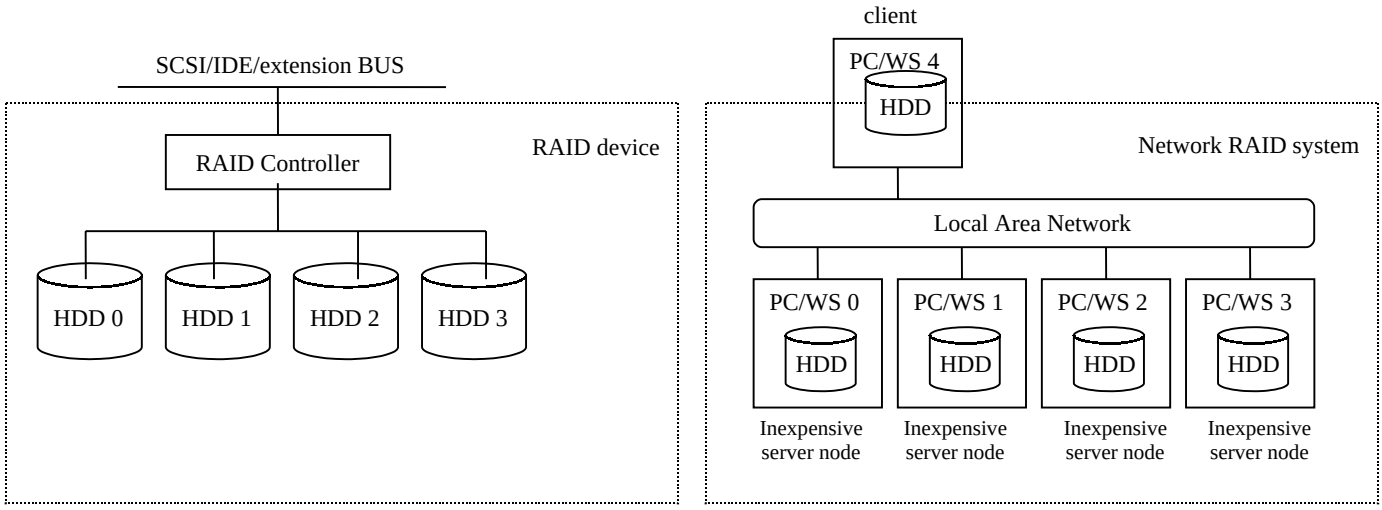


図3 RAID vs Network RAID

ファイルシステムの信頼性向上目的のみを目指して新たに計算機を増設したとすれば実現コストが大きくなるが、分散共有ファイルシステムを必要とする環境には元々複数台の計算機が存在する。これらの資源を活用してファイルシステムの信頼性を向上させれば資源の有効活用であり、全体としてのコストは増大しない。計算機レベルで冗長性を持たせることにより、RAID装置の高コストの原因である、電源の多重化、ファンの多重化が自然に達成され、1台の計算機の電源を落しても他の計算機には影響がでないため活線挿抜のための専用回路も不要である。

3.2．障害検出訂正方式の検討

パリティ方式にはストライピングアクセスが方式自体に組み込まれているため、単純なミラーリング方式よりも性能面では有利である。しかし、ミラーリング方式もストライピングを併用することで性能向上を図ることが可能である。実装上の問題を除けば、ミラーリング方式とパリティ方式の本質的な差はディスク容量の利用率のみということになる。複数の計算機からなる分散計算機環境に分散共有ファイルシステムを導入して、プログラムやデータの共有を推し進めると、各計算機のディスク容量は余り気味になることが多い。これらの余剰ディスクを計算機資源と一緒に有効活用するのがネットワークRAID方式であるため、ディスク容量の利用率は大きな問題とはならない。このため、実装が低コストかつ簡単なミラーリング方式もしくはミラーリングを拡張した方式がネットワークRAIDに適している。

ミラーリングをネットワークレベルに拡張した方式として、複数台（2台）の計算機上のディスクに同一のデータを格納しておき、ディスク障害が発見された場合のみ予備のデータを他の計算機から獲得するという方式が考えられる。しかし、安価な計算機を冗長に使用する場合は、ディスク装置の故障のみではなく計算機自体のデータ化けやクラッシュも無視できない。計算機自体のエラーまで考慮に入れて信頼性を考えると、前述のRAIDのミラーリングを単純にネットワークを介して実現した方式では不十分である。計算機レベルでエラーが発生している可能性がある場合は、クライアント側（ファイルをアクセスする側）でデータ読み出し時にデータの正しさを確認する必要がある。つまり、データ読み出し時にミラーされた計算機（サーバ）から複数のコピーを読み出して、本当にデータが一致しているか確認を行う必要がある。データ使用時にデータ不一致が発見されても、ディスク故障以外の場合、どのサーバから送られたデータが間違えているか確定できない可能性があるため、ミラーの数は最低 3セット必要であり、「多数決原理」によって正しいデータを決める必要がある。ディスクレベルでは CRCチェック等によりエラーが発生したディスクのセクタが特定できるが、書き込み前に ECCなしメモリやキャッシュ上でデータ化けが発生した場合等は、冗長度が 2ではエラー発生側を特定できない。多数決原理で正しいデータを確定するため、本方式を今後「多数決方式」と呼ぶことにする。

3.3．ネットワークRAIDの実装方式

前節において、ネットワークRAIDではクライアント側において多数決による格納データの認証を行う必要があることを述べた。クライアント側が複数のミラーサーバからデータを取り寄せて、データを逐一比較していたのではクライアント側のメインCPUに掛ける負荷が大きくなってしまう。また、ネットワーク（LAN）のバンド幅が小さい場合には、複数のミラーサーバからデータを転送することが分散共有ファイルシステムとしてのボトルネックになりかねない。さらに、複数のミラーサーバと通信を行うこと自体が通信オーバヘッドを引き起こし、システム全体の性能を低下させる可能性がある。これらの問題に対する解決策／改善策となる実装方式について述べる。

3.3.1．データ転送量を削減する方式

データのディスクへの書き込みはすべてのミラーサーバに反映される必要があるため、ネットワーク上のデータ転送量を削減することは原理的に不可能である。しかし、読み出しに関してはデータ転送量削減の可能性がある。データの誤りが高い確率で検出できれることがクライアント側の認証の目的である。そこで、サーバが読み出したデータに対応する認証データ（チェックサムやMD5やSHA-1等のハッシュ関数）を一定サイズ（通信パケットサイズ）毎にサーバ側で計算して、1台のサーバはデータと認証データをクライアントに送信し、他のサーバは認証データのみを送信して、クライアントは認証データが一致していることを確認する。そして、認証データが一致していれば、データも一致していると信用してデータを使用する。認証データのbit幅を大きくすればデータ誤りの検出率を高くすることができる。この方式はクライアント側の処理の一部をサーバ側に分散する方式でもあり、サーバにMD5やSHA-1計算用のハードウェアアクセラレータがある場合に特に適している。これらのアクセラレータはセキュリティ対策のために高価なサーバマシンには徐々に普及しつつあるが、ネットワークRAIDがターゲットとするような安価なパーソナルコンピュータには現状では実装されておらず、将来的にそこまで普及するかどうかは定かではない。ただし、ネットワークバンド幅不足がネットワークRAID方式のボトルネックとなっている場合には、たとえソフトウェアでこの実装方式を実現しても効果が非常に大きい。

3.3.2. NICを利用して認証負荷を削減する方式

ネットワークRAIDは分散共有ファイルシステムを対象としているため、本質的に計算機間の通信が発生する。事実上の標準通信プロトコルであるTCP/IPおよびUDP/IPにはデータのチェックサム機能が定義されているため、最近の多くのネットワークインタフェースカード（NIC）にはチェックサム計算支援用ハードウェアが搭載されている。各サーバは生のデータをネットワーク経由でクライアントに転送すると、各サーバのNICはチェックサムを計算して通信パケットに付与する。このチェックサム値（16bit）を認証データとして流用して、クライアント側ではチェックサムが一致していることによりデータが一致していると見做す。ただし、TCP/IPやUDP/IPで通信を行った場合はチェックサムの範囲がヘッダの一部を含むため、若干補正計算を行う必要がある。なお、チェックサム計算支援ハードウェアを持たない場合でも、分散共有ファイルシステムの通信プロトコルとしてTCP/IP（UDP/IP）を用いる場合は通信レイヤで計算されるチェックサムを流用することが可能である。

チェックサムが一致してエラー発生が確認されない通常ケースでは、複数のサーバノードからデータ本体を送信することはネットワークバンド幅の浪費である。既存のNICを流用する場合にはこのオーバヘッドはやむを得ないが、ネットワークRAID対応NICを新たに開発する場合は、チェックサムだけ転送できるような通信オプションを用意することで、この無駄を無くすことができる。ただし、Gigabitクラスの高速ネットワークでは、通信オーバヘッドによってバンド幅を使い切ることが難しいため、この程度のバンド幅の浪費は問題とならない。

また、多数決方式といっても不整合が発見されるまでは、2ノードのデータの一致のみで運用を続けられるので、クライアントにデータ（もしくは認証データ）を転送するのは2台のサーバで十分である。2ノードしかデータ（もしくは認証データ）を転送しない場合はアクセスが一部のミラーサーバに偏っていると、データ誤りを長期間見落として放置してしまう可能性が高くなる。このことは回復不能なエラーの発生率を高めるので、アクセスするミラーサーバの選択制御に十分な注意が必要である。

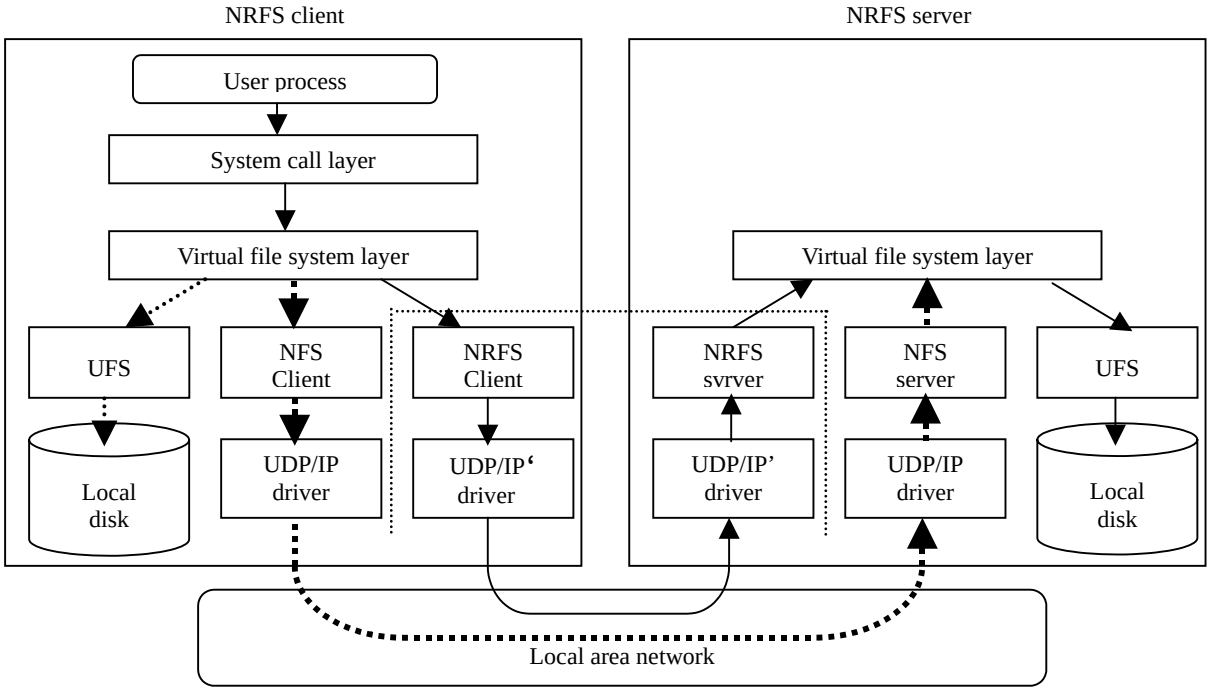


図4 ファイルシステムの構成

3.4. ネットワークRAIDファイルシステム

3.4.1. NRFSの概要

ネットワークRAID方式をベースに高信頼分散共有ファイルシステムであるネットワークRAIDファイルシステム（NRFS）を作成することが最終目的である。ネットワークRAID方式はその上に構築される分散共有ファイルシステムの方式に特別な制限を設けることはない。そこで、既存ファイルシステムとの互換性を重視して、分散共有ファイルシステムの事実上の標準であるSun Microsystems社が開発したNFS (Network File System)をベースにして NRFSを開発する。

NRFSはマウントポイントにおける制限を緩和することによって、複数のミラーサーバを一つのマウントポイントに対して設定可能にする。例えば、nrfs_clientというマシンにおいて

```
mount nrfs_serv1:/usr/local /usr/local
mount nrfs_serv2:/usr/local /usr/local
mount nrfs_serv3:/usr/local /usr/local
```

の三つのマウントコマンドを受け付ける。ただし、マウントポイントにすでにマウントされているファイルシステム階層構造がある場合には、新たにマウントされる階層構造と構造が一致していること（ディレクトリ構成、ファイル名、ファイルサイズ、ファイル属性）を確認する。一致していなければ、どちらかの内容にコピーによって強制的に一致させるか、マウントをあきらめるかオペレータに選択させる。多重にマウントされたファイルシステムはオペレーティングシステムが自動的にNRFSとして取り扱う。NRFSにおけるストライピング動作はマウントコマンドの新オプションとして指定可能にする。

典型的なノーエラー時のファイル読み出しと書き込みは以下のように実行される。

nrfs_client内のプロセスが/usr/local以下のファイルに対して読み出し要求（システムコール）を発行すると、三つのうちの二つのミラーサーバにファイルデータの読み出し要求が送信され、各サーバからデータが認証データ（チェックサム）と共に返送される。nrfs_client側においてチェックサムの一致が確認され、読み出し要求を発行したプロセスにデータが渡される。

nrfs_client内のプロセスが/usr/local以下のファイルに対して書き込み要求（システムコール）を発行すると、データ処理単位に端数があってその部分がローカルにキャッシュされていない場合はまずリモートの読み出しを行う。データ処理単位を満たす書き込みデータが揃ったら、三つのミラーサーバすべてにファイルデータの書き込み要求が書き込みデータとともに送信され、各ミラーサーバではファイルキャッシュもしくはローカルディスクに書き戻される。

3.4.2. NRFSの構成

NRFSはNFSの拡張として実装されるため、VFS（Virtual File System）がサポートするファイルシステムの一種類としてオペレーティングシステムに組み込まれる。図４にNRFSを含むファイルシステムの構成を示す。図では便宜上NRFSクライアントとNRFSサーバが別のノードとして記述されているが、通常各ノードはファイルシステムのある部分に関してはサーバとして振舞い、他の部分ではクライアントとして振舞う。この辺りの事情はNFSと同じである。

NRFSシステムはクライアント側がNRFS clientとUDP/IP driverで構成されており、サーバ側がNRFS server daemonとUDP/IP driverで構成されている。ただし、UDP/IP driverはデータチェックサムのNRFSにおける流用に対応する変更を施す。

4．平成１３年度開発のキーポイント

ネットワーク RAID および NRFS の方式自体は前節に示したとおりであり、平成１３年度の開発目標はその実用版を作ることにあった。この実用版 NRFS を作成するにあたり、動作原理を考案した時には思い至らなかった問題が数多く発生した。その中で、技術的に面白い話題についていくつか紹介する。

4.1．RPC における非同期タスクの活用

Linux の NFS は性能の向上を図るため、ファイル読み出しやファイル書き込みは先読みやバッファリングを活用して、大きな単位で行っている。このため、RPC レベルでは複数の RPC 要求に分割されるが、これらを RPC の非同期タスクという機構を使って同時処理している。非同期に対して同期タスクというのも存在するが、このタイプの RPC が行われると、これに続く RPC はこの同期タスクの RPC の返答が戻るまで遅延される。これに対して、非同期タスクの RPC では、後続の RPC を可能であれば、発行することによって、処理性能を向上させる。NRFS では、通常同期タスクとして処理される lookup や readdir であっても、複数のサーバに対して実行され、サーバ間には依存関係はない。このため、同一メッセージに対する複数サーバ向けの RPC メッセージは非同期タスクとして扱うこととして、性能向上を図った。

4.2．ディレクトリチェックサムキャッシュ機構の新設

NFS では 1 ディレクトリにメモリ 1page 分では納まらない容量のエントリがある場合には、複数回の readdir を発行してディレクトリのエントリ名に関する情報を収集する。こういった大きなディレクトリの 2 回目以降の readdir では前回までに次に読み出すエントリの位置に関するサーバ依存の情報を提供する必要がある。この情報は直前の readdir の実行結果といっしょにクライアントに戻されている。ディレクトリ内のエントリ名の格納位置はサーバに依存しており、同じ名前のエントリであってもマシンが異なれば同じ位置にあることは期待できない。このため、NRFS ではディレクトリの誤り検出をディレクトリ全体の内容のチェックサムを用いて行うこととし、一回目の readdir においてチェックサムデータをすべてのサーバから取り寄せることにした。チェックサムの値が一致していれば内容は正しいものとし、大きなディレクトリに対する 2 回目以降の readdir は正しいと信用されるサーバ 1 台のみを対象として実行する。当然、全部のチェックサムが一致しない場合には、多数決に勝ったサーバが正しいと信用される。このように NRFS の readdir はディレクトリの全エントリに対するチェックサム計算結果を要求する。readdir の度にこのチェックサム値を計算するのは、ディレクトリのサイズが大きい場合には無視できないオーバヘッドが発生すると考え、ディレクトリチェックサムキャッシュ機構をサーバ側に新設することにした。名称通りに一度計算したディレクトリのチェックサムの値をキャッシュしておく機構である。もちろん、ディレクトリのエントリの更新や削除があった場合には該当キャッシュエントリはフラッシュされる。

4.3．ディレクトリチェックサムキャッシュ機構の更新

前小節で述べたように readdir の認証用のチェックサム計算の高速化のためのキャッシュ機構を新設した。しかし、最初に作った機構は、NFS や NRFS からのファイルアクセスでは厳密にキャッシュがフラッシュされて常に正しい値がキャッシュされるように管理されるが、サーバ側でローカルファイルシステムとしてファイルやディレクトリを直接操作した場合にはキャッシュがうまくフラッシュされないという問題が発生した。NRFS の障害復旧機構は NFS の枠組みで障害を取り除くのではなく、通常の通信手段とサーバ上のデーモンによるローカルファイルシステムの操作によって障害復旧を実現している。このため、障害復旧機構が働いて障害を取り除いても、ディレクトリチェックサムキャッシュの値が古いままで、エラーが解除されないという事態が発生した。この問題は NFS のサーバにあるキャッシュをどうやってローカルファイルシステムの操作と同期させるかという一般的な問題である。Linux では、i-node に i_version というメンバを設けて、NFS とローカルファイルシステム（EXT2 や UFS）のどのパスから実ファイルシステムが操作された場合でも、この i_version をインクリメントすることによりバージョン管理していることが判明した。つまり、NFS は NFS でサーバ側に独自のディスクキャッシュを持つことがあるが、このキャッシュに i_version の値もキャッシュしておき、キャッシュ参照された時には通常のヒット判定の他に i_version が現在の値と一致していることを確認する。もしも一致していなければ、ディスクから最新データを読

み込み直す。ディレクトリチェックサムキャッシュ機構にもこの i_version によるバージョン管理を導入することで、実際のディレクトリのエントリとチェックサムキャッシュが矛盾する事態を回避できるようになった。

4.4. 時計のずれに関する問題

NRFS は複数台のサーバに同じファイルシステムを保持することが高信頼性を得るための冗長性の根源である。NRFS がベースにした NFS ではファイルやディレクトリの生成時刻や更新時刻はサーバ側の時計によって管理されている（ただし、NFS Ver.2)。複数あるサーバの時刻が完全に一致していれば問題にならないわけであるが、完全な一致を期待することは不可能であり、NFS と同じ時刻の取扱いをするためには、どのサーバの時刻をユーザに提示するかという問題が発生する。そこで、NRFS ではサーバ内のローカルファイルシステムには該当サーバの時計で刻印するが、クライアントに示す時刻はあくまでもクライアントの時計に基づいて決定することにした。つまり、クライアントと各サーバの時計の時差をあらかじめ測定しておき、この時差で補正した値をファイルやディレクトリの時刻としてユーザに提供するわけである。この方式であれば、ユーザはクライアントの時計と矛盾した時刻の提示は受けず、サーバ間の時刻のずれも大きな問題とはならない。逆に、大きく時刻がずれていても使用中にずれが大きく変化しなければ、問題なく使用可能である。つまり、NRFS のサーバを世界各地に配置して同一マウントポイントに接続しても、時刻のずれによる問題は発生しない。

4.5. パス情報の復元

NFS はサーバがダウンしてもリブートすればコネクションが回復することから、ステートレスの通信によるファイルシステムのように一見思われる。しかし、内部を解析すると、ファイルハンドルと呼ばれるサーバ側のローカルファイルシステムの i-node 情報をベースにしたサーバ依存の状態や各種キャッシュを使ってファイルアクセスの高速化を図っている。一見ステートレスに見えるのは、それらのキャッシュや状態を注意深くフラッシュする機構が付いているからであり、毎回フラッシュするような本当のステートレスな実装では大幅に性能がダウンしてしまう（将来マシンパワーに物を言わせてこちらの実装にする計画もあるようですが)。NRFS は NFS をベースに実装されたため、NRFS 層においてファイルハンドルをサーバごとに管理して、複数のサーバに対するアクセスを行っている。よって、NRFS にとっての根源的な情報はファイルハンドルということになる。しかし、ファイルハンドルは個別のサーバ依存の情報であり、本来根源的な情報はユーザ／アプリケーションが指定した（ファイル）パス情報である。このパス情報が VFS 層から NFS 層・RPC 層に渡る間に分解され、原型が判らなくなってしまってファイルハンドルベースでサーバ・クライアント間の通信がなされている。障害復旧機構にクライアントから障害をレポートする場合に、可能であればパス情報を教える方が見通しがよい。なぜなら、ファイルハンドルを用いて指示を出す場合は、サーバ毎のファイルハンドルをすべて示さないと意味がない。しかし、マウントポイントからのパス情報であればすべてのサーバが理解することが可能である。そこで、VFS が管理するディレクトリキャッシュ（dentry）の内容を逆にたどることにより、パス情報が復元可能なことを突き止めて、障害復旧機構への連絡はパス情報に基づいて行うこととした。

5．Linux NRFS クライアントの実装

5.1 要求概要

NFSプロトコルを利用し、図5の示すように、１つのマウントポイントに、複数（最大8）サーバのディレクトリをマウントして、一つの仮想ディレクトリ構造として使用することより、低コスト高信頼性のネットワーク・レイド・ファイルシステムを実現する。

1) ルート権限でマウントして使用できる。
   注：将来的に、一般ユーザが意識せずに使用できるようにしたい
2) NRFS ファイルシステムの障害検出、復旧できる
   ① ファイルデータ化け、欠落
   ② ファイル/ディレクトリの誤り、欠落
3) 個別のサーバが故障しても、処理が続けられる
   ① ネットワーク障害
   ② サーバが異常停止

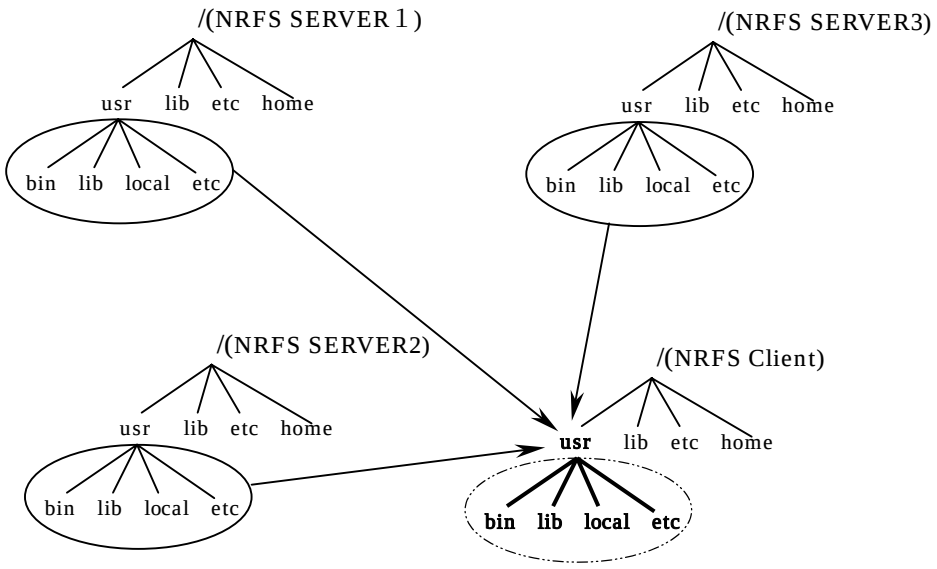


図５．NRFS ファイルシステムのマウント

5.2. 構造体及び定義の実装詳細

5.2.1. NRFS の構造体定義

NRFS ファイルシステム情報を保持する重要な構造体について説明する。

1）NRFS のファイルシステムタイプ
2）NRFS のスーパーブロック情報
3）NRFS の inode 情報
4）NRFS のファイルシステム固有情報
5）NRFS 層ディレクトリエントリキャッシュ構造体
6）NRFS のデータ読み出し要求用構造体
7）NRFS のデータ書き込み要求用構造体

5.2.1.1. ファイルシステムの定義

NRFS と VFS の接点であり、NRFS を VFS に追加するため、欠かせない手続きである。NRFS のカーネルモジュールが

ロードされる時　　**register_filesystem(struct file_system_type)**
アンロードされる時　**unregister_filesystem(struct file_system_type)**

を呼び、VFS がサポートするファイルシステムリストから追加・削除する。

表 1. ファイルシステム定義

構造体定義	NRFS ファイルシステム

Include/linux/fs.h
 	
`struct file_system_type {`
`  const char *name;`
`  int fs_flags;`
`  struct super_block  *(*read_super) (struct super_block *, void *, int);`
`  struct file_system_type * next;`
`};`
	fs/nrfs/inode.c
	
	`struct file_system_type nrfs_fs_type = {`
	`  "nrfs",`
	`  0,`
	`  nrfs_read_super,`
	`  NULL`
	`  }`

5.2.1.2. スーパーブロック情報の定義

NRFS ファイルシステムのスーパーブロック情報構造体の定義は表３のようになる。
１つマウンポイントに、最大８つの NRFS サーバを同時にマウントできる。マウントされたサーバについて、それぞれサーバの固有情報

struct nrfs_server　（表３）
struct nrfs_fh

及びクライアントとサーバとの間のシステム時計のズレ値を配列にすることにより保持する。これらの情報は、マウント時にフィックスされる。

表 2. スーパーブロック情報構造体定義

構造体	備考

include/linux/nrfs.h
`#defined NRFS_MAXSRVS  8`
 	
include/linux/nrfs_fs_sb.h
 	
`struct nrfs_sb_info {`
`  struct nrfs_server  s_server[NRFS_SUPER_MAX];`
`  struct nrfs_fh  s_root[NRFS_SUPER_MAX];`
`  int  s_lagtime[NRFS_SUPER_MAX];`
`  int  s_cnt;`
`  int  flags;`
`  int  rsize;`
`  int  wsize;`
`  unsigned int  acregmin;`
`  unsigned int  acregmax;`
`  unsigned int  acdirmin;`
`  unsigned int  acdirmax;`
`};`
	表 3
	NFS と同じの構造体を使用して多重化
	クライアントとサーバ間の時計ズレ
	マウントした NRFS ｻｰﾊﾞ数
	マウンドオプションの保存用
	ブロック読込みサイズ
	ブロック書込みサイズ
	ファイルキャッシュのタイムアウト最小値
	ファイルキャッシュのタイムアウト最大値
	ディレクトリキャッシュのタイムアウト最小値
	ディレクトリキャッシュのタイムアウト最大値

表 3. サーバー構造体定義

構造体	備考

include/linux/nrfs_fs_sb.h
 	
`#defined NRFS_MAXSERVERS  3`
 	
`struct nrfs_server {`
`  struct rpc_clnt * client;`
`  char *  hostname; /* remote hostname */`
`  char *  dirname;  /* server exported path */`
`  unsigned int  bsize;  /* server block size */`
`};`
	RPC クライアントハンドル
	サーバのホスト名
	マウント元のパス名（***障害通知発行時使用***）
	サーバブロックサイズ

5.2.1.3. inode 情報の定義

NRFS サーバから取得した inode 番号を保持できるように、表 4 のような NFS の inode 情報構造体（struct nrfs_inode_info）を作成する。

表 4. NRFS の i ノード情報構造体定義

構造体	備考

include/linux/nrfs_fs_i.h
 	
struct nrfs_inode_info {
  struct pipe_inode_info pipeinfo;
  unsigned short flags;
 	
  unsigned long read_cache_jiffies;
  unsigned long read_cache_mtime;
  unsigned long attrtimeo;
 	
  /*
  * H.Q: The follow field is for nrfs repair protocol
  */
  ino_t ino[NRFS_MAXSERVERS];
  __u32 atimes[NRFS_MAXSERVERS];
  __u32 mtimes[NRFS_MAXSERVERS];
  __u32 ctimes[NRFS_MAXSERVERS];
  unsigned short dir_read_from;
  unsigned short fattr_from;
  unsigned int repair_count;
 	
  struct nrfs_wreq * writeback;
};
	VFS からは inode->u.pipe_I になる
	
	それぞれのサーバにおいて固有の inode 情報を多重化して保持する
	inode 番号、または、fileid
	最後アクセスした時間
	最後変更した時間
	作成した時間
	ディレクトリ読出し用信用できるサーバ番号
	ファイル属性が使われているサーバの番号
	ファイルブロック復旧要求発行カウンタ
	
	遅延書きタスクの待ちポイント

5.2.1.4. ファイルシステム固有情報の定義

それぞれのサーバにおいて、ぞれぞれのファイル/ディレクトリの固有情報を保持する、重要な構造体である。ファイル/ディレクトリアクセスに備え、ファイル・ディレクトリの走査で成功する度に、VFS のディレクトリエントリ構造体の d_fsdata フィールドにセットし、保持される。

表 5. ファイルシステム固有情報構造体定義

構造体	備考

include/linux/nrfs.h
 	
struct nrfs_fsdata {
  struct nrfs_fh fhs[NRFS_MAXSERVERS];
};
	ディレクトリの走査成功する度に、VFS のディレクトリエントリ構造体の d_fsdata フィールドにセットされます。

5.2.1.5. NRFS 層ディレクトリエントリキャッシュ構造体の定義

NFS のディレクトリエントリキャッシュを流用する。NRFS としてのディレクトリエントリを確認できるように、inode 番号を多重化する。

表 6. ファイルシステム固有情報構造体定義

構造体	備考

include/linux/nrfs.h
 	
struct nrfs_dirent {
dev_t dev; /* device number */
ino_t ino[NRFS_SUPER_MAX]; /* inode number */
__u32 * entry; /* three __u32's per entry */
u32 cookie; /* cookie of first entry */
unsigned short valid : 1, /* data is valid */
locked : 1; /* entry locked */
unsigned int size; /* # of entries */
unsigned long age; /* last used */
unsigned long mtime; /* last attr stamp */
struct wait_queue * wait;
};
	inode 番号（ino、fileid に対応する）を多重化する。

5.2.1.6. RPC タスク管理構造体

NRFS において、１つのファイルシステム操作要求に対し、複数サーバに非同期 RPC コールを使用するため、RPC コールの管理が必要となる。（読み書きの操作要求について後記参照）

表 7. RPC タスク管理構造体

構造体	備考

include/linux/nrfs.h
 	
struct nrfs_proc_mgn {
int req_cnt; /* call count in one NRFS request */
int req_done; /* all call requested flag */
struct wait_queue *wait; /* NRFS request wait point */
int status[NRFS_MAXSRVS]; /* rpc call status for all */
};
 	
struct nrfs_rcall_inf {
int cb_idx; /* async-call identifier */
struct nrfs_proc_mgn *proc_mgn;
};
	RPC コール管理構造体
	
	RPC コール管理用 RPC タスク情報構造体

5.2.1.7. データ読み出し要求用構造体

非同期 RPC タスクを使用して読出しを行うため、すべてのサーバに対して、同じデータの読出しを要求する RPC タスクを管理する必要がある。
その他、データの最読み要求、障害復旧要求を発行するための情報を保持する。

表 8. データ読み出し(READ)要求用構造体の定義

構造体	備考

fs/nrfs/read.c
 	
読出しコール管理用構造体
`struct nrfs_rreqmgn {`
`  struct dentry *   mr_dentry; /* inode from which to read */`
`  struct page *     mr_page;   /* page to be read */`
`  struct nrfs_fattr mr_fattr[NRFS_MAXSERVERS];`
`  void *            mr_bufffer;`
`  int               mr_offset; /* read offset for re-read  */`
`  int               mr_length; /* read size for re-read  */`
`  int               mr_stat[NRFS_MAXSERVERS];`
`  int               mr_sumdat[NRFS_MAXSERVERS];`
`  int               mr_rqid;   /* read request ID  */`
`  int               mr_scnt;   /* number of server alive  */`
`  int               mr_txcnt;  /* sent read request counter */`
`  int               mr_rvchk;  /* first answer got flag */`
`  int               mr_lock;   /* lock flag for update page data */`
`  int               mr_idxdat; /* server index that called for data */`
`  int               mr_idxok;  /* server index that reply good data */`
`  int               mr_idxng;  /* server index that reply bad data  */`
`#ifdef NRFS_CHECKSUM_USE`
`  u32                 mr_sumdat[NRFS_MAXSERVERS];`
`#endif`
`};`	

読み出しコール用情報構造体
`struct nrfs_rreq {`
`  struct nrfs_csumreadargs ra_args;  /* XDR argument struct */`
`  struct nrfs_csumreadres  ra_res;   /* ... and result struct */`
`  struct nrfs_fattr        ra_fattr;  /* fattr storage */`
`  struct nrfs_rreqmgn *    ra_rqmgn;`
`  u32                      ra_proc;  /* rpc program number */`
`  int                      ra_idx;   /* point to which buf it used */`
`};`	

5.2.1.8. データ書き込み要求用構造体

非同期 RPC タスクを使用して書き出しまたは遅延書き出しを行うため、すべてのサーバに対して、同じデータの書き出しを要求する RPC タスクを管理する必要がある。
その他、障害復帰要求を発行するための情報を保持する。

表 9. データ書き込み(WRITE)要求用構造体の定義

構造体	備考

include/linux/nrfs_fs.h
 	
書き込みコール管理用構造体
struct nrfs_wreqmgn {
struct file * mw_file; /* dentry referenced */
struct page * mw_page; /* page to be written */
unsigned int mw_offset; /* offset within page */
unsigned int mw_bytes; /* dirty range */
pid_t mw_pid; /* owner process */
int mw_rqid;
int mw_scnt;
int mw_rqcnt;
int mw_stat;
unsigned short mw_flags; /* status flags */
unsigned int mw_count; /* user count */
struct waite_queue * mw_wait; /* wait for complete */
struct nrfs_fattr mw_fattr[NRFS_MAXSERVERS];
struct nrfs_wreq * mw_wreq[NRFS_MAXSERVERS];
};	

書き込みコール情報構造体
struct nrfs_wreq {
struct rpc_listitem wb_list; /* linked list of req's */
struct rpc_task wb_task; /* RPC task */
struct nrfs_writeargs wb_args; /* NRFS RPC stuff */
struct nrfs_fattr wb_fattr; /* file attributes */
struct nrfs_wreqmgn * wb_reqmgn;
unsigned short wb_flags; /* status flags */
int wb_idx;
};	

5.2.2. VFS 層の変更

NRFS サポートできるようにするため、VFS 層に対して、下記のの変更・追加定義を行った。

1） VFS のスーパーブロックの変更
2） VFS の inode 構造体の変更
3） VFS の dentry フラグ定義の追加

5.2.2.1. VFS のスーパーブロックの変更

VFS のスーパーブロック構造体に、NRFS のスーパーブロック情報エントリを追加する。

表 10. VFS のスーパーブロックの変更

構造体	備考

include/linux/nrfs_fs.h
 	
struct super_block {
struct list_head s_list;
kdev_t s_dev;
unsigned long s_blocksize;
unsigned char s_blocksize_bits;
unsigned char s_lock;
unsigned char s_rd_only;
unsigned char s_dirt;
struct file_system_type *s_type;
struct super_operations *s_op;
struct dquot_operations *dq_op;
unsigned long s_flags;
unsigned long s_magic;
unsigned long s_time;
struct dentry *s_root;
struct wait_queue *s_wait;
 	
struct inode *s_ibasket;
short int s_ibasket_count;
short int s_ibasket_max;
struct list_head s_dirty;
 	
union {
struct minix_sb_info minix_sb;
struct ext2_sb_info ext2_sb;
・・・
struct nrfs_sb_info nrfs_sb;
・・・
void *generic_sbp;
} u;
/* for VFS *only*. */
struct semaphore s_vfs_rename_sem;
};
	構造体の定義について、表 1 を参照

5.2.2.2. VFS の inode 構造体の変更

VFS の inode 構造体に、NRFS の inode 情報エントリを追加する。

表 11. VFS の inode 構造体の変更

構造体	備考

include/linux/nrfs_fs.h
 	
struct inode {
struct list_head i_hash;
struct list_head i_list;
struct list_head i_dentry;
unsigned long i_ino;
unsigned int i_count;
kdev_t i_dev;
umode_t i_mode;
nlink_t i_nlink;
uid_t i_uid;
gid_t i_gid;
kdev_t i_rdev;
off_t i_size;
time_t i_atime;
time_t i_mtime;
time_t i_ctime;
unsigned long i_blksize;
unsigned long i_blocks;
unsigned long i_version;
unsigned long i_nrpages;
struct semaphore i_sem;
struct semaphore i_atomic_write;
struct inode_operations *i_op;
struct super_block *i_sb;
struct wait_queue *i_wait;
struct file_lock *i_flock;
struct vm_area_struct *i_mmap;
struct page *i_pages;
struct dquot *i_dquot[MAXQUOTAS];
unsigned long i_state;
unsigned int i_flags;
unsigned char i_pipe;
unsigned char i_sock;
int i_writecount;
unsigned int i_attr_flags;
__u32 i_generation;
union {
・・・
struct nrfs_inode_info nrfs_i;
・・・
} u;
};
	構造体の定義について、表 4 を参照

5.2.2.3. VFS の dentry フラグ定義の追加

障害復旧されたエントリのキャッシュ情報の補完処理及び、障害通知間引き処理ために、 下記の定数定義を追加定義し、VFS の dentry 構造体（表）の d_flags を利用する。

```
#define DCACHE_NRFSFS_RENAMED 0x0010    /* work same as NFSFS_RENAMED   */
#define DCACHE_NRFSFS_ILL_FHL 0x0020    /* NULL file handle exist       */
#define DCACHE_NRFSFS_ILL_SVR 0x0040    /* STALED server exist          */
#define DCACHE_NRFSFS_REPAIR  0x0080    /* repair request has been done */
```

表 12. VFS の inode 構造体の変更

構造体	備考

include/linux/nrfs_fs.h
 	
struct dentry {
int d_count;
unsigned int d_flags;
struct inode * d_inode;
struct dentry * d_parent;
struct dentry * d_mounts;
struct dentry * d_covers;
struct list_head d_hash;
struct list_head d_lru;
struct list_head d_child;
struct list_head d_subdirs;
struct list_head d_alias;
struct qstr d_name;
unsigned long d_time;
struct dentry_operations *d_op;
struct super_block * d_sb;
unsigned long d_reftime;
void * d_fsdata;
unsigned char d_iname[DNAME_INLINE_LEN];
};
	多重化したファイルハンドルをセットする。
	構造体の定義について、表 5 を参照

5.2.3. SUNRPC 層の変更

NRFS の処理用 RPC タスクの非同期化、ネットワーク障害及び個別サーバの異常停止に備え、NRFS タスクのタイムアウト処理を行うため、クライアント構造体の変更（表 12）を行い、下記の定数定義の追加する。

```
#define CLNT_NRFS_ACTIVE     100 /* the last nfs error code is 99(WFLUSH) */
#define CLNT_NRFS_STALE      101
#define CLNT_NRFS_TRIAL      102

#define RPC_TASK_NRFS        0x1000   /* NRFS task recognization */
#define RPC_TASK_NRFSWRITE   0x2000   /* an NFS writeback */
#define RPC_IS_NRFS(t)       ((t)->tk_flags & RPC_TASK_NRFS)
```

表 13. SUNRPC のクライアント構造体の変更

構造体	備考

include/linux/nrfs_fs.h
 	
struct rpc_clnt {
unsigned int cl_users; /* number of references */
struct rpc_xprt * cl_xprt; /* transport */
struct rpc_procinfo * cl_procinfo; /* procedure info */
u32 cl_maxproc; /* max procedure number */
char * cl_server; /* server machine name */
char * cl_protname; /* protocol name */
struct rpc_auth * cl_auth; /* authenticator */
struct rpc_stat * cl_stats; /* statistics */
 	
unsigned int cl_softrtry : 1,/* soft timeouts */
cl_intr : 1,/* interruptible */
cl_chatty : 1,/* be verbose */
cl_autobind : 1,/* use getport() */
cl_binding : 1,/* doing a getport() */
cl_oneshot : 1,/* dispose after use */
cl_dead : 1;/* abandoned */
unsigned int cl_flags; /* misc client flags */
unsigned long cl_hardmax; /* max hard timeout */
 	
struct rpc_portmap cl_pmap; /* port mapping */
struct rpc_wait_queue cl_bindwait; /* waiting on getport() */
 	
int cl_nodelen; /* nodename length */
char cl_nodename[UNX_MAXNODENAME];
unsigned short cl_nrfscallcnt; /* NRFS call count */
unsigned short cl_srvstat; /* NRFS server status *
* 100: normal, *
* 101: stale, *
* 102: trial */
struct semaphore cl_sema; /* semaphore for cl_srvstat */
};
	以降のフィールドは
	ネットワークまた個別サーバ障害処理のために追加する

5.3. VFS における変更

1） NRFS ファイルシステムの登録

VFS のスーパーブロック構造体と inode 構造体に、NRFS のスーパーブロック情報エントリと inode 情報エントリを追加し、NRFS のカーネルモジュールの初期化する際、NRFS のファイルシステムタイプを VFS のサポートするファイルシステムタイプリストに登録すること（図 6）より、NRFS が使用可能になる。

```
/* File system information */
static struct file_system_type nrfs_fs_type = {
        "nrfs",
        0               /* FS_NO_DCACHE - this doesn't work right now*/,
        nrfs_read_super,
        NULL
};

/* Initialize NRFS */
int init_nrfs_fs(void)
{
#ifdef CONFIG_PROC_FS
        rpc_register_sysctl();
        rpc_proc_init();
        rpc_proc_register(&nrfs_rpcstat);
#endif
    return register_filesystem(&nrfs_fs_type);
}


/* kernel module load/unload procedure */
#ifdef MODULE
EXPORT_NO_SYMBOLS;
/* I just port nfs to nrfs and maintain it */
MODULE_AUTHOR("Huang Qiang <huang@mmc.co.jp>");

int init_module(void)
{
        return init_nrfs_fs();
}

void cleanup_module(void)
{
#ifdef CONFIG_PROC_FS
        rpc_proc_unregister("nrfs");
#endif
        unregister_filesystem(&nrfs_fs_type);
        nrfs_free_dircache();
}
#endif
```

図 6. NRFS モジュール内の登録処理

2） ファイルシステムマウントシーケンスの変更

表 14 のようにマウントフローを変更することより、1 つのマウントポイントに複数の NRFS サーバのディレクトリをマウントできるようになる。

クライアントとサーバとの間のシステム時計ズレは、マウント時にサーバ毎に計算し、NRFS のスーパーブロック情報に保持する。NRFS サーバの現在時刻の取得するため、新にサーバ時刻を問い合わせて RPC スタブを追加する。

表 14. カーネル内のマウントフロー及び NRFS のための変更点

マウントフロー	備考

sys_mount(dev_name, dir_name, type, new_flag, mount_data) {
　if リマウント {
　　マウントオプションをカーネル空間にコピー(to page);
　　do_remount(dir_name, flags, page);
　　後始末処理、do_remount から貰った結果をマウントコマンドに返す；
　}
　マウントオプションをカーネル空間にコピー(to page);
　サポートしているファイルシステムであるかをチェック;
　ローカルであれば、物理デバイス関連チェック;
　未使用のデバイス番号を取得（to dev）；
　do_mount(dev, dev_name, dir_name, type, flag, page) {
　　マウント先がディレクトリかをチェックし、マウント先の dentry を取得(ERROR: -ENOTDIR);
　　ERROR =-EBUSY;
　　if マウントポイントがマウント済み {
　　if (マウント NRFS) {
　　取得したデバイス番号を返す；
　　ERROR = do_mount_nrfs_n(dentry, fstype->name, flag, page) {
　　dentry よりスーパーブロック(sb)を取得する。
　　dentry のファイルシステムタイプは NRFS かをチェックする
　　shrink_dcache_sb（sb）;
　　fsync_dev(sb->dev);
　　fstype->read_super();
　　}
　　}
　　後始末処理、ERROR を sys_mount に返す
　　}
　　read_super(dev, type, flags, data, 0) {（スーパーブロック取得、ERROR: -EINVAL);
　　デバイス番号のチェック；
　　if 指定したデバイス番号はマウント済み　{
　　そのデバイス番号にマウントしているスーパーブロックを do_mount()に返す；
　　}
　　ファイルシステムタイプ名（type）から指定するファイルシステムを探し出す；
　　新しいスーパーブロック用メモリ確保；
　　指定するファイルシステム ***（NRFS）*** の read_super()手続きを呼び出す；
　　指定するファイルシステムのスーパーブロックを do_mount()に返す
　　}
　　スーパーブロックが指定したデバイス番号にマウントできるかをチェック（ERROR: -EBUSY）
　　VFS のマウント済みリストに追加する（ERROR: －ENOMEM）；
　　マウントポイント（dentry）に取得したスーパーブロックを関連付ける;
　　後始末処理、マウンド結果を sys_mount()に返す
　}
　後始末処理、do_mount から貰った結果をマウントコマンドに返す；
}
	1) １つ目の NRFS のマウントは従来通りのマウントフローで行ないます。
	2) 2 つ目の NRFS のマウントは、マウント済みの分岐に入り、
	① マウント済のマウントポイントのスーパーブロックに新しい NRFS サーバ情報を追加する
	② マウント済のマウントポイントの i ノードに新しい NRFS サーバ関する情報（i ノード番号等）を追加する
	
	変更点
	↑
	|
	|
	|
	|
	|
	↓
	変更点
	
	NRFS の read_super を呼び出し、サーバ毎に、クライアントと　サーバ間のシステム時計ズレを含むスーパーブロック情報を取得する。

5.4. NRFS 層の内部処理

5.4.1. ファイル・ディレクトリ情報の取扱い

NRFS のファイル・ディレクトリ情報の取扱いは下記通りに行う。

１） ファイル ID（fileid）

NRFS の inode 情報の ino フィールドに多重化して保持する。VFS は inode->u.nrfs_i.ino より参照可能である。

NRFS 層において、キャッシュ更新時、キャッシュされているファイル/ディレクトリエントリと問い合わせ結果ファイル･ディレクトリエントリが一致しているかどうかを確認するために必要とする最も重要な情報である。

VFS が inode キャッシュを作成する時に必要とする inode 番号は次のように作成する。但し、idx はファイル/ディレクトリの属性情報が NRFS のファイル/ディレクトリの属性情報として使用されるサーバのインデックス番号である。

```
unsigned long  nrfs_fid_vno(unsigned long idx, unsigned long fileid)
{
        int shiftb = sizeof(unsigned long) * 8 - 3;
        return (unsigned long) ((fileid << 3 >> 3) | (idx << shiftb));
}
```

図 7. ファイル ID から VFS の inode 番号を生成する

２） ファイルハンドル

VFS の dentry の d_fsdata フィールドに多重化して保持する

それぞれのサーバにあるファイル･ディレクトリをアクセスする時に必要となる最も重要な情報である。

３） ファイル/ディレクトリの時間属性

全てのサーバからのアクセス時刻(atime)、編集更新時刻(mtime)、新規作成時刻(ctime)は、それぞれ、NRFS の inode 情報の atimes、mtimes、ctimes フィールドに多重化して保持し、キャッシュ情報の有効性の確認、更新の必要性を判断する重要な情報である。

VFS の inode キャッシュに使用する際、マウント時に取得したサーバとクライアントのシステム時計ズレ値で補正する。

４） NRFS のファイル/ディレクトリの属性情報

多数決によるサーバの返答情報の照合で、勝ち組の中から、任意の１つのサーバから返答された結果を使用する。 VFS の inode キャッシュに保持し、必要に応じてユーザープロセスに渡される NRFS のファイル/ディレクトリ情報として、下記通りである。

- **タイプ（type）　　障害検出の照合対象）**
- **モード(mode)　　障害検出の照合対象)**
- **リンク数(nlink)　　障害検出の照合対象)**
- **ユーザ ID(uid)　　障害検出の照合対象)**
- **グループ ID(gid)　　障害検出の照合対象)**
- **サイズ(size)　　障害検出の照合対象)**
- **ブロックサイズ(blocksize)**
- **デバイス番号(rdev)**
- **ブロック数(blocks)**
- **アクセス時間(atime)**
- **編集更新時間(mtime)**
- **新規作成時間(ctime)**

5.4.2. NRFS 操作要求手続きの実装

NRFS では、ファイルデータの READ/WRITE を除くすべてのファイルシステムの操作要求に対して、3.1.6 節の RPC タスク管理構造体を用いて、サーバ毎に非同期 RPC 発行して問い合わせる。その後、発行した非同期タスクが完了するまで、図 8 の関数を呼び、自らカーネルのスケジュラの待ち行列に入る。

```
static int nrfs_proc_wait_on_request(struct nrfs_proc_mgn *rpmgn)
{
        struct wait_queue wait = {current, NULL};
        int retval;

        /* raise all done flag */
        rpmgn->req_done = 1;

        /* wait all reply arrive */
        add_wait_queue(&rpmgn->wait, &wait);
        for (;;) {
                current->state = TASK_INTERRUPTIBLE;
                retval = 0;
                if (!rpmgn->req_cnt)
                        break;
                retval = -ERESTARTSYS;
                if (signalled())
                        break;
                schedule();
        }
        remove_wait_queue(&rpmgn->wait, &wait);
        current->state = TASK_RUNNING;

        return retval;
}
```

図 8. リクエスト依存スリープ手続き

同じ操作要求のために発行した全ての非同期タスクが完了した時点で、カーネルスケジュラは親サービスを待ち行列から出し、親サービスの処理が再開する。

1） 問い合わせ結果の照合する。(詳細 5.3 節参照)
2） 呼び出し側に結果を返す
3） エラーが検出した場合、正常サーバに復旧要求を発行する。

1）ディレクトリエントリの読み出し

1 回目の読み出し(ディレクトリのオフセットを示す cookie 値が 0 である)時に、全ての NRFS サーバに対してチェックサム付きディレクトリ読み出しを発行し、エラー検出を行う。同時に、結果が使われるサーバのインデックス i-node 情報（dir_read_from、3.1.3 節参照）として記憶する。

2 回目以降の読み出しは、i-node に保存された情報を元に、1 つサーバのみに対して、チェックサムなしのディレクトリ読み出しを行う。

2）ファイルデータの読み込み

NRFS クライアントにおいて、NRFS 仮想ディレクトリ以下のデータに対し、1 つの読み込み要求は複数サーバに対して行う必要がある。さらに下記処理は読み込み中に行う必要がある。

① マウントした全ての NRFS サーバの内、1 つのサーバーのみに、読み込みデータと読み込みデータのチェックサムデータの転送を要求する。その他のサーバには読み込みデータのチェックサムデータの転送だけを要求する。
② 多数決原理に基づき、チェックサムデータの比較により、受信したデータの正当性を確認する。

③　受信したデータにエラー等を検知した場合、エラー修正に必要な情報をパラメータに、正常サーバの一つにエラー訂正要求を送信する。

データの読み込みおよび読み込み時の処理要求に対して、NRFS クライアントのデータ読み込み処理は次のステップに分けて行う。

①　マウントした NRFS サーバ分のデータ格納領域を確保する（図 5 参照、read_request、複数の read_call の管理も兼ねる）それぞれサーバにデータ転送を要求する際、データの格納先を１）で確保した領域を指定する。（図 9 参照、read_call_1，read_call_2，…，read_call_n）

②　転送要求した全てのサーバからの返答が揃った時点で、チェックサムデータを比較する。

③　すべてのチェックサムデータが一致している場合、VFS が読み込み要求する時に指定したキャッシュに読み込んだデータをコピーし、キャッシュページのロックを解除する。

④　チェックサムデータの不一致が生じた場合、一致多数のサーバのデータを正当とし、不一致チェックサムデータの転送元がチェックサムデータのみを転送している場合、データ訂正情報をパラメータに、その転送元サーバにデータ訂正の要求を発行する。

- データ転送の要求先を兼ねている場合、一致多数のサーバの１つから、データ転送を要求し、データが受信完了した時点で、上記①の処理を行う。

⑤　同一ファイル同一サーバの誤りに対して一定回数 N までデータ訂正要求を発行するが、一定の回数 N を超えた場合、データの訂正要求は発行しない。一定回数 N は次の式を使用して、データファイルサイズ(SIZE)より求める。

(SIZE >> 13)　>>　((N − a) x 3)　　0　　a = 0: if SIZE < 4096
a = 1: if SIZE > 4096)

⑥　不一致チェックサムデータの転送元サーバは以降のデータ転送要求対象サーバとしない。

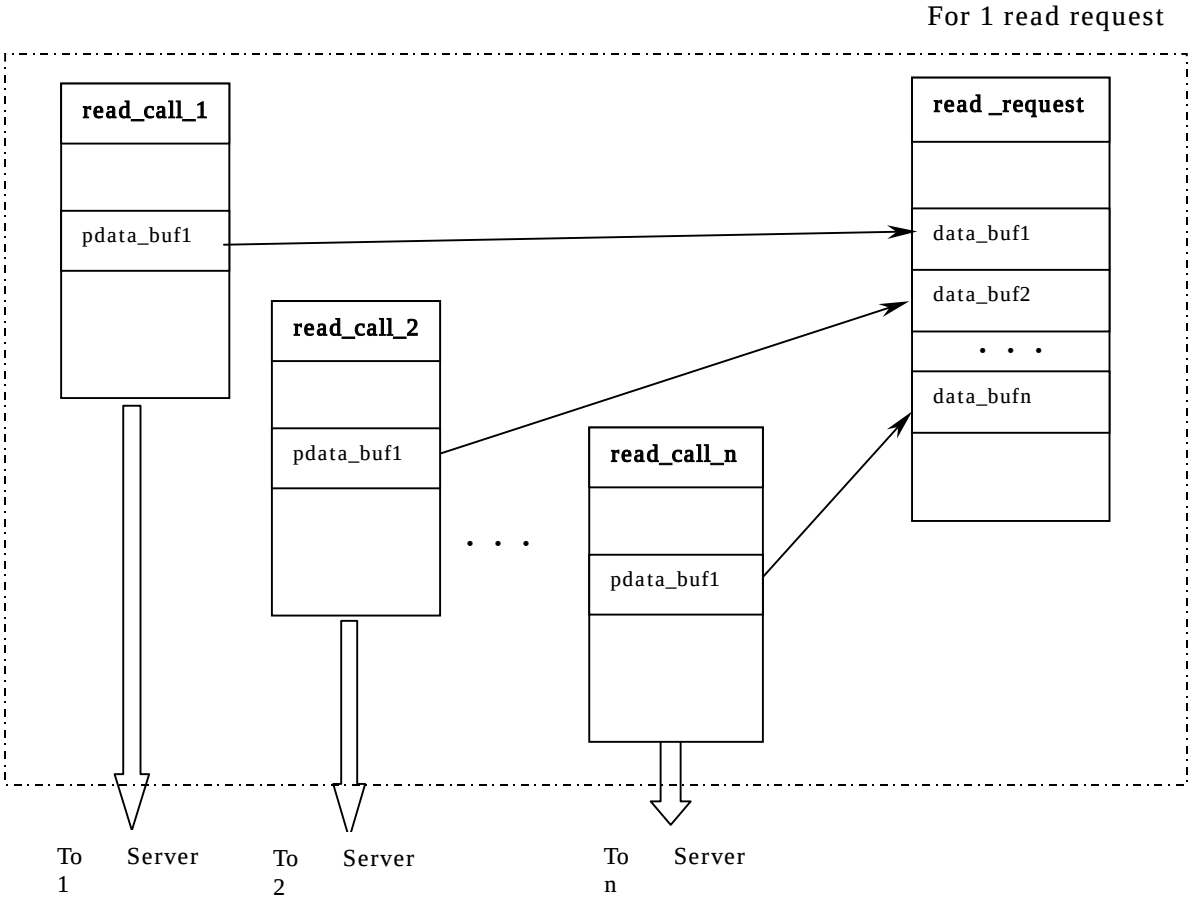


図 9.　読み込み処理使用データ間の関係

2）ファイルデータのの書き込み

NRFS クライアントにおいて、NRFS 仮想ディレクトリ以下のデータに対し、１つの書き込み要求は同じく、複数サーバに対して行う必要である。サーバへの書き込み処理は次のように行っている。(図 10 参照)

① 書き込みコール管理および書き込み情報を保持する一時的メモリ領域（write_request）を確保する。

② 全てのサーバに対して、同じ書き込み内容は１）で保持している書き込み情報を利用するように書き込みコールタスク(write_call_x)を作成する。

③ 作成した書き込みコールタスクのスケジュリング：

- RPC データ転送用スロットが確保できる場合、実行遅延（3 秒）を設定して、書き込み待ち行列（write_queue）に入れる。書き込み待ち行列の最大待ち数（WRITEBACK_MAX）は１２８である。
- RPC データ転送用スロットが確保できない場合、タスクが同期実行を行う。
- 待ち中のタスク数が最大待ち数の３/4 になった、待ち行列先頭にあるタスクが KRPC 実行の待ち行列に移され、書き込み内容を転送する実行状態に入る。

④ 下記の場合、その書き込み実行タスクは即 KRPC 実行の待ち行列に移され、書き込み内容を転送する実行状態に入る。

・ １つの書き込み要求内容サイズがキャッシュページサイズに達した。
・ 他のプロセスが書き込み内容に対して読み込み要求した。

⑤ 同じ書き込みに対しての書き込みタスクの実行がすべて終了した時点で、書き込み要求があるキャッシュページの更新済みフラグをセットして VFS に知らせる。

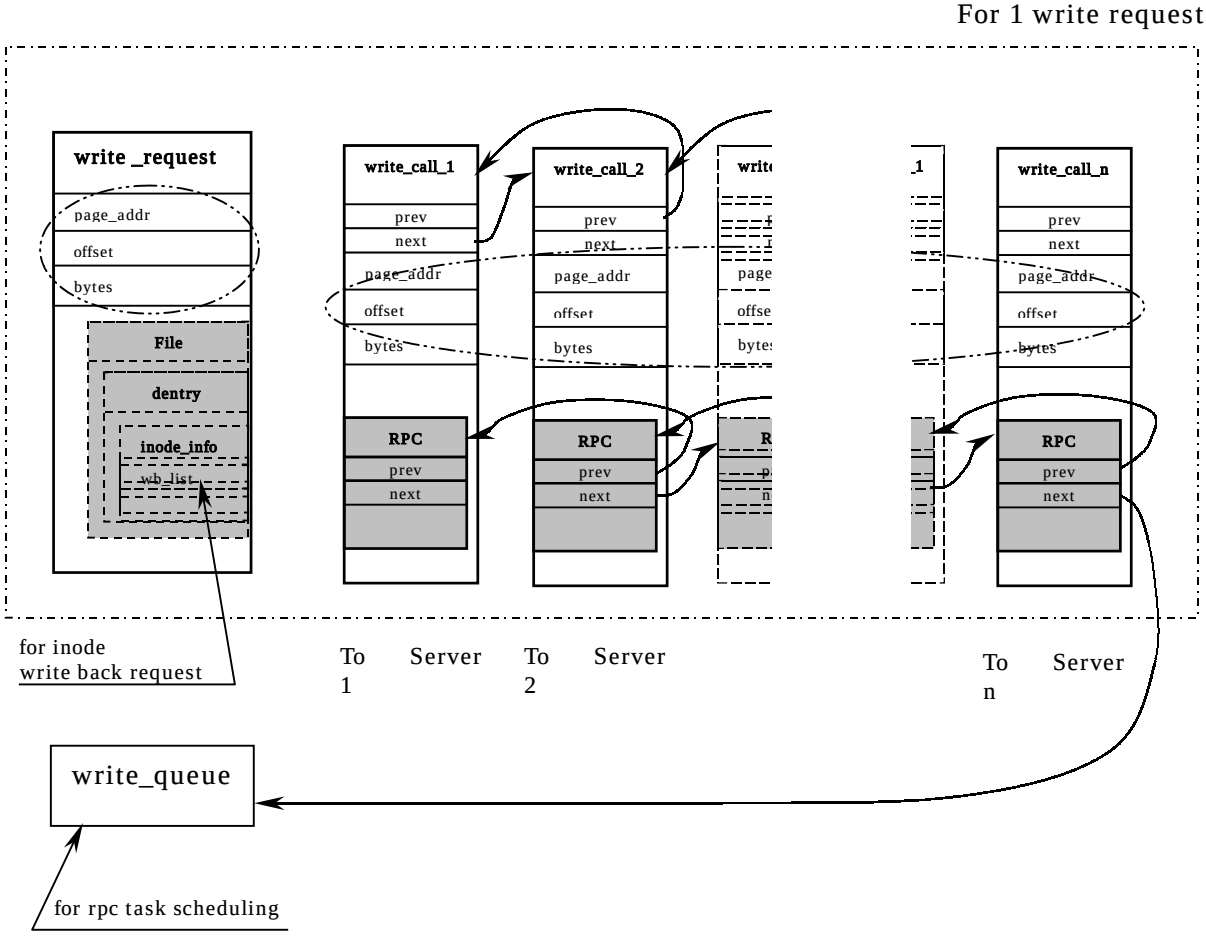


図 10. 書き込み処理使用データ間の関係

5.4.3. 障害検出

全ての NRFS サービスコールにおいて、まず、RPC コールステータスチェックを行い、問合せのステータス別に下記の処理を行う。

① 全て失敗である場合、NRFS サービスコールが失敗として、エラー検出は行わない。

② 成功少数である場合、NRFS サービスコールが失敗であるが、成功少数サーバの方に、エラーが存在する。エラータイプは表 2 が示すように決める（readdir、statfs を除く）。

③ 成功多数である場合、NRFS サービスコールが成功であるが、失敗少数サーバー方に、エラーが存在する。エラータイプは表 3 が示すように決める（readdir、statfs を除く）。

④ 全て成功（NRFS_OK）の場合、必要に応じて、チェックサム値や、ファイル属性等の照合を行う。

- 前記ステータスチェックのみで終了する NRFS サービスコール
  REMOVE、RENAME、LINK、SYMLINK、RMDIR、STATFS
- ファイル属性チェックが必要とする NRFS サービスコール（表 15 参照）
  GETATTR、SETATTR、LOOKUP、CREATE、MKDIR、WRITE
- その他のチェックが必要とする NRFS サービスコール
  READLINK　パス名及び文字列長さ
  READDIR　チェックサム値（ディレクトリ以下、すべてのファイル名）
  READ　チェックサム値（ファイル中身）

表 15. ファイル/ディレクトリ属性一覧

	照合	備　考
struct nrfs_fattr {	—	
enum nrfs_ftype type;	する	障害検出チェック対象
_u32 mode;	する	障害検出チェック対象
_u32 nlink;	する	障害検出チェック対象
_u32 uid;	する	障害検出チェック対象
_u32 gid;	する	障害検出チェック対象
_u32 size;	する	障害検出チェック対象、但し、ファイルのみ
_u32 blocksize;	しない	サーバ依存属性であるため
_u32 rdev;	しない	サーバ依存属性であるため
_u32 blocks;	しない	サーバ依存属性であるため
_u32 fsid;	しない	サーバ依存属性であるため
_u32 fileid;	使用済み	inode番号として使われている
struct nrfs_time atime;	しない	
struct nrfs_time mtime;	しない	ファイル･ディレクトリの更新有無の確認は行う
struct nrfs_time ctime;	しない	
};		

5.4.4. 障害発生時の処理

障害検出した時点で、下記の復旧要求のための情報を揃えて、正常サーバ（多数決の勝ち組の任意１つ）に障害発生を通知する。

1） マウントした全てのサーバ IP アドレスと公開パス名
2） 障害が発生したエントリのマウントポイントまでの絶対パス名、障害通知を発行する。但し、RENAME と LINK の処理時、変更元と変更先毎に、復旧パス名は親ディレクトいに対して指定する。
復旧パスは VFS の dentry キャッシュ逆引きすることより求める（図 11 参照)。
3） 障害が発生したエントリがファイルの場合、必要に応じて、ブロック復旧出来るように、オフセットとレングス情報

障害通知発行後、重複または不要な障害通知を間引くため、障害通知発行済み情報（DCACHE_NRFSFS_REPAIR、3.2.3 節参照）を該当エントリの dentry キャッシュに、キャッシュが有効期間中または次回の走査するまで、付けて保持する。障害通知を間引くケースは下記の通りである。但し、障害通知発行済み情報はつける。

- サーバがダウンしている間。
- 親ディレクトリのファイルハンドルが NULL である。
- 親ディレクトリに対して障害通知済みである。
- ファイル復旧通知後検出したファイルブロックエラー

障害が復旧された後に、クライアントキャッシュを補完復元するために、ファイルハンドル取れないエントリの dentry キャッシュの d_flags に NRFSFS_ILL_FHL を付加する。

```
static char *nrfs_get_path(struct dentry *dentry, const char *name, char *buf, int buflen)
{
        struct dentry *root = dentry->d_sb->s_root;
        char *end = buf + buflen;
        char *retval = NULL;
        int namelen = 0;

        *--end = '\0';
        buflen--;
        if (dentry->d_parent != dentry && list_empty(&dentry->d_hash)) {
                buflen -= 10;
                end -= 10;
                memcpy(end, " (deleted)", 10);
        }

        /* add my name first */
        if (name) {
                namelen = strlen(name);
                buflen -= namelen + 1;
                end -= namelen;
                memcpy(end, name, namelen);
        }

        for (;;) {
                struct dentry * parent;

                /* add '/' to left */
                *--end = '/';
                retval = end;

                if (dentry == root)             break;

                dentry = dentry->d_covers;
                parent = dentry->d_parent;
                if (dentry == parent)           break;

                namelen = dentry->d_name.len;
                buflen -= namelen + 1;
                if (buflen < 0)                         break;

                end -= namelen;
                memcpy(end, dentry->d_name.name, namelen);
                dentry = parent;
        }
}
```

図 11. 復旧パスを求める手続き

5.4.5. 障害復旧後の処理

ネットワークの障害またはサーバがダウンした間、そのサーバにあるファイル/ディレクトリの情報及びファイルハンドルを更新復元する必要である。

NRFS のファイル/ディレクトリを走査する処理の中に、VFS の denry キャッシュの d_flags をチェックする処理（図 12）を埋め込み、ファイル/ディレクトリの情報及びファイルハンドルを更新復元する必要のあるエントリを見つけると、処理しているエントリのパスを遡って、復帰したサーバに問合せを発行し、サーバがダウンしていた間に取得できなかった情報（ファイルハンドルと時間属性）取得し、dentry と i-node キャッシュを補完する（図 13）。

```
int nrfs_dentry_is_recoverable(struct dentry *dentry)
{
        struct nrfs_server *server = NRFS_DSERVER(dentry);
        struct nrfs_fh *fh = NRFS_FH(dentry);
        int i, scnt = NRFS_DSVRCNT(dentry);
        int srvidx = -1, recovered = 0;

        /* check ill file handle and server status */
        for (i = 0; i < scnt; i++) {
            if (NRFS_FH_ISNULL(fh[i])) {
                  if (rpc_test_srvstat(server[i].client, CLNT_NRFS_ACTIVE)) {
                      dfprintk(VFS, "NRFS: recoverable dentry found (server#%d come back) !\n", i);
                      srvidx = i;
                      break;
                  }
            }
            else
                  recovered++;
        }

        /* entry have been recovered, clear ill flag */
        if (recovered == scnt) {
            dentry->d_flags &= ~DCACHE_NRFSFS_ILL_FHL;
            dfprintk(VFS, "NRFS: nrfs_fhget[%s/%s], NRFSFS_ILL_FHL resetted\n",
                      dentry->d_parent->d_name.name, dentry->d_name.name);
        }

        return srvidx;
}

/* check and try to recover null file handle setted when server down */
if (NRFS_NULLFH_EXIST(dentry)) {
          idx = nrfs_dentry_is_recoverable(dentry);
          if (idx >= 0) {
                    dfprintk(VFS, "NRFS:  scanning all parent of [%s] ... \n",
                              dentry->d_name.name);
                    nrfs_recover_dentry(dentry, idx, 0);
          }
}
```

図 12. ファイルハンドルの補完個所のチェック

```
int nrfs_recover_dentry(struct dentry *dentry, int idx, int level)
{
        struct nrfs_server *server = NRFS_DSERVER(dentry);
        struct dentry *parent = dentry->d_parent;
        struct inode *inode = dentry->d_inode;
        struct nrfs_fh *pfh = NRFS_FH(parent);
        struct nrfs_fh *fh = NRFS_FH(dentry);
        struct nrfs_fattr fattr;
        int lcur = level + 1;
        int status = 0;

        if (NRFS_FH_ISNULL(pfh[idx])) {
                /* when ill server exist, we must look back from mount point */
                if ((dentry->d_flags & DCACHE_NRFSFS_ILL_SVR) && !NRFS_IS_ROOT(parent))
                        parent->d_flags |= DCACHE_NRFSFS_ILL_SVR;
                /* recover my parent first */
                if (nrfs_recover_dentry(parent, idx, lcur)) {
                        return -NRFSERR_NOENT;
                }
        }

        /* lookup entry for recover */
        dfprintk(VFS, "NRFS:  try to recover [%s] by %d up.\n", dentry->d_name.name, lcur);
        status = nrfs_proc_recover_lookup(&(server[idx]), &(pfh[idx]),
                                                dentry->d_name.name, &(fh[idx]), &fattr);
        if (!status) {
                if (inode) {
                        inode->u.nrfs_i.ino[idx] = fattr.fileid;
                        inode->u.nrfs_i.atimes[idx] = fattr.atime.seconds;
                        inode->u.nrfs_i.mtimes[idx] = fattr.mtime.seconds;
                        inode->u.nrfs_i.ctimes[idx] = fattr.ctime.seconds;
                }
                dentry->d_flags &= ~DCACHE_NRFSFS_ILL_SVR;
        }
        else if (dentry->d_flags & DCACHE_NRFSFS_ILL_SVR) {
                /* notify server repair this entry now */
                int i, scnt = NRFS_DSVRCNT(dentry);
                for (i = 0; i < scnt; i++) {
                        if ((i != idx) && (!NRFS_FH_ISNULL(fh[i])))
                                break;
                }
                nrfs_do_repair_call(dentry->d_parent,
                                        dentry->d_name.name, i, 0, 0, 0);
        }

        return status;
}
```

図 13. ファイルハンドルの補完処理

5.5. SUNRPC における変更

通常の RPC タスクは、SUNRPC スケジュラによって図 14 のように実行される。 タイムアウト発生した際、call_timeout ステップで、タイムアウト時間を調整した上、call_transmit ステップから、サーバからレスポンスあるまで、リトライを繰り返して行う。

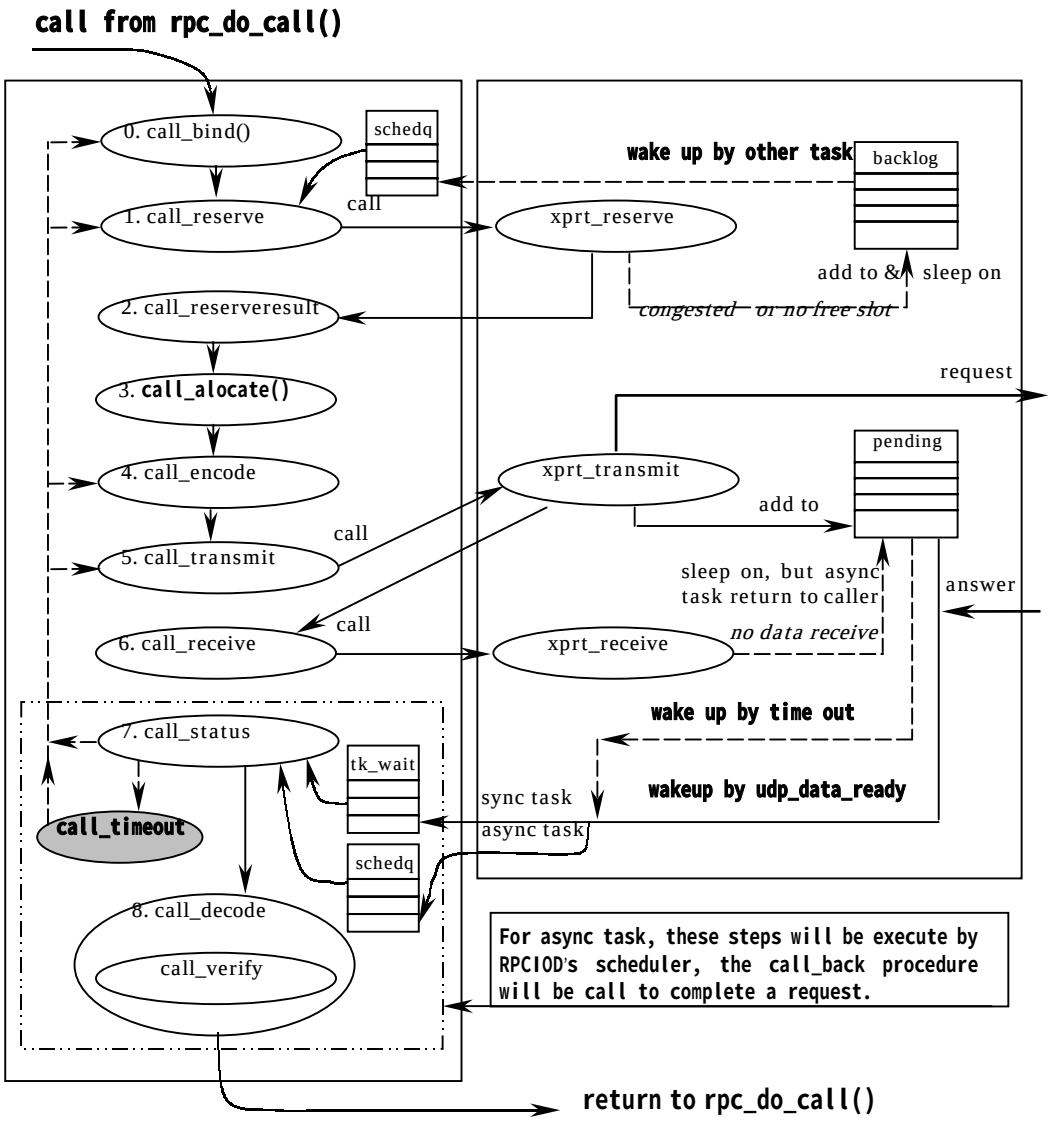


図 14. SUNRPC タスク実行サイクル

個別のサーバの通信異常（ネットワーク障害またはサーバダウン）が発生しても、NRFS ファイルシステムとしての処理を続けさせるために、NRFS の RPC タスクの処理をメジャータイムアウトした時点で中断させるように実装変更する。具体的には、

1） 次のように NRFS のサーバ状態を定義する。
- **通常（CLNT_NRFS_ACTIVE）**
- **異常（CLNT_NRFS_STALE）**
- **確認中（CLNT_NRFS_TRIAL）**

2） NRFS の RPC クライアントに、実行中の RPC タスクカウンターとサーバ状態を保持するメンバーフィールド（3.3 節を参照）を用意する。

3） NRFS の RPC タスク作成時、サーバの状態ををチェックする。
- 「通常」である場合、RPC_TASK_NRFS フラグを付けて、タスクを発行する。
- 「異常」である場合、「確認中」にセットし、異常サーバ向け、返答があるまで終了しない非同期 RPC タスク作成し発行する。この非同期タスクについて、タイムアウトの値は NFS の現行値を使用した NULL スタブであり、コールバック処理はサーバから返答がある時、サーバ状態フラグを「正常」に戻すのみとする。
- 「確認中」である場合、NRFS タスクの作成をキャンセルする。

4） RPC スケジュラ-のタイムアウト処理を変更し、メジャータイムアウトに達した NRFS のタスクを中断する。実行中の同じサーバへの RPC タスクがないことを確認し、そのサーバの状態を異常に切り替える。

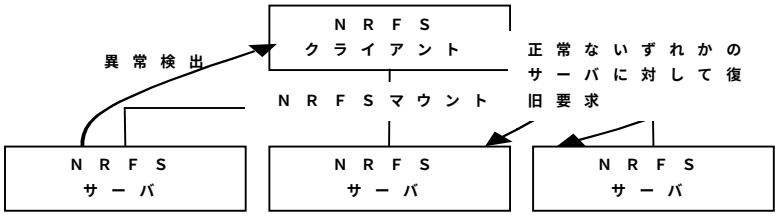
6．復旧処理

6.1．復旧処理の概要

今回、復旧処理について、復旧要求を受け取ったサーバが、復旧対象サーバ、または、正常サーバといった前提で復旧を行うという前回までの方針を、復旧要求を受け取ったサーバは、単に調停サーバとして動作し、リカバリ対象について他のサーバと、多数決の確認を行った上で少数派のサーバに対して復旧を促すといった方針で再検討を行うこととした。

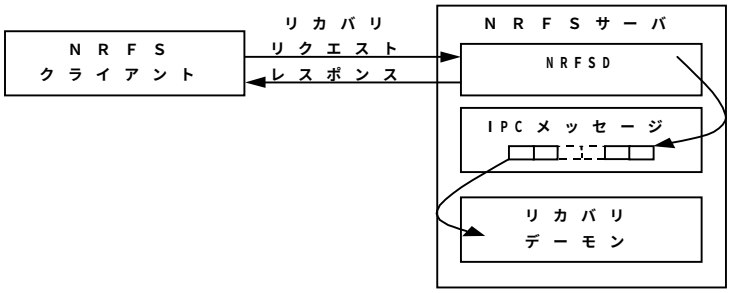
復旧処理の流れ:
復旧処理は、NRFS クライアントが、NRFS マウント中の少数サーバ上でのファイルシステム上の不整合の検出時に、いずれかの正常サーバにリカバリ・リクエストを発行することにより開始される。（6.2. リカバリ・リクエスト、レスポンスメッセージ参照）

リカバリ・リクエスト



リカバリ・リクエストを受け付けた NRFS サーバは、リクエスト時に受け取った復旧要求情報に、クライアントの接続情報を付加して、自サーバ上で動作するリカバリデーモンに対して IPC メッセージを使用して通知を行う。(6.3. 復旧情報参照)

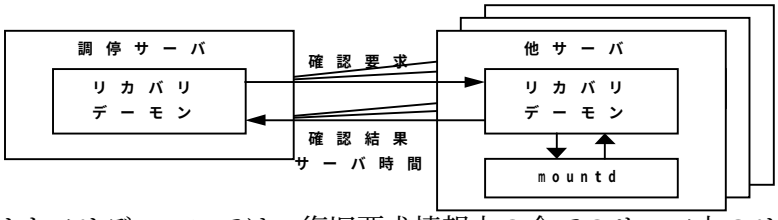
復旧要求通知



クライアントからの復旧要求を受け取ったサーバ（以降、調停サーバ）上のリカバリデーモンは、IPC メッセージを通じて受け取った復旧要求情報を元に、復旧要求情報中の各サーバ情報で示される他の全てのサーバ上で動作するリカバリデーモンに接続を行い、クライアントの IP と、それぞれのサーバの公開パス名を通知し、クライアントの確認を要求する。

クライアント確認要求を受け取ったリカバリデーモンは、クライアントの IP アドレスから逆引きしたホスト名、および、自サーバの IP アドレスと、復旧要求情報中のサーバ情報より、自サーバの公開パス名を取得し、自サーバで動作中の mountd のマウント中のクライアント情報からの復旧要求であることを確認し、確認結果を、調停サーバ上のリカバリデーモンに返す。

クライアントのマウント確認



調停サーバ上のリカバリデーモンでは、復旧要求情報中の全てのサーバ上のリカバリデーモンからのクライアントのマウント確認ができている場合に以降の復旧処理を行う。

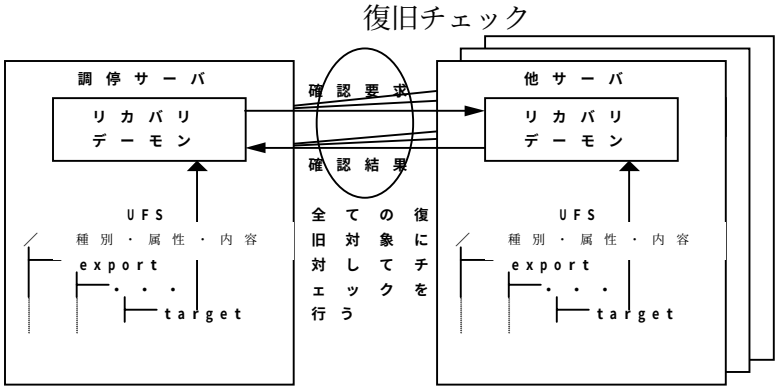
クライアント確認時点ですべてのサーバで、クライアントのマウント確認が出来ない場合、復旧処理を中止する。

追加補足:

調停サーバは、NRFS サーバから復旧要求受付時に、調停サーバ内のメッセージキャッシュバッファに要求メッセージを、バッファの空き容量が許す限り先読みキャッシュを行い、一定時間内に同一の復旧要求メッセージを受け取った場合に、受け取った同一の（重複する）復旧メッセージを破棄するように実装されている。

上記の実装は、直前に受け取って実行中の復旧処理および、すでにメッセージバッファにキャッシュされているメッセージに対する、重複する復旧要求メッセージの受付を防止するためで、NRFS クライアントのアクセス速度と、リカバリデーモンの復旧速度の速度差（NRFS クライアントのアクセス速度＞リカバリデーモンの復旧速度）から生じる過剰な復旧要求による IPC メッセージキューのオーバーフロー防止と、重複メッセージの廃棄を目的としている。

復旧処理:
　復旧処理では、種別、属性、内容の順に、各サーバ上の復旧対象毎に多数決を行い、サーバ間で対象の種別、属性、内容に食い違いがある場合、多数決における少数派のサーバに対して復旧要求を行うものとする。



種別チェック：対象の存在チェック兼用
　種別チェックでは、各サーバに対して、復旧対象の種別（ファイル、ディレクトリ、シンボリックリンク）の確認を行い、多数決に負けた少数派のサーバの情報に、対象削除および対象復旧要求を行う。
　多数決に負けた少数派のサーバ上に対象が存在しない場合には、対象復旧要求のみ、逆に、対象が存在する場合には、対象削除要求のみを行う。

属性チェック:
　属性チェックでは、各サーバに対して、復旧対象の属性（アクセスパーミッション、所有者 ID、グループ ID、ファイルサイズ、最終アクセス時間、最終更新時間）の確認を行い、多数決に負けた少数派のサーバの情報に、属性復旧要求を行う。
　属性復旧要求は、すでに対象削除、復旧要求を行ったサーバには発行しない。

属性チェックでは、対象の属性中の以下に示す属性についてチェックを行う。

属性復元対象の属性

属性名	属性の意味	チェック 復旧	備考
type	ファイルタイプ＊	可	種別チェックで使用される。 ファイルタイプが、ファイル、ディレクトリ、シンボリックリンクの場合のみ相互に復旧可能
mode	アクセスパーミッション	可	chmod() を使用した復旧
nlink	ハードリンク数	不可	変更不可能な情報
uid	所有者 ID	可	chown() を使用した復旧
gid	グループ ID	可	
size	ファイルサイズ	可	truncate() を使用した復旧
blksize	ブロックサイズ	不可	サイズに連動して変化
rdev	デバイスファイル時のデバイスタイプ	不可	変更不可能な情報
blocks	ブロック数	不可	
fsid	デバイス番号	不可	
fileid	i-node 番号	不可	
atime	最終アクセス時間	不可	変更しない
amilli		不可	変更方法無し
mtime	最終更新時間	不可	変更しない
mmilli		不可	変更方法無し
ctime	i-node 更新時間	不可	変更不可能と思われる
cmilli			

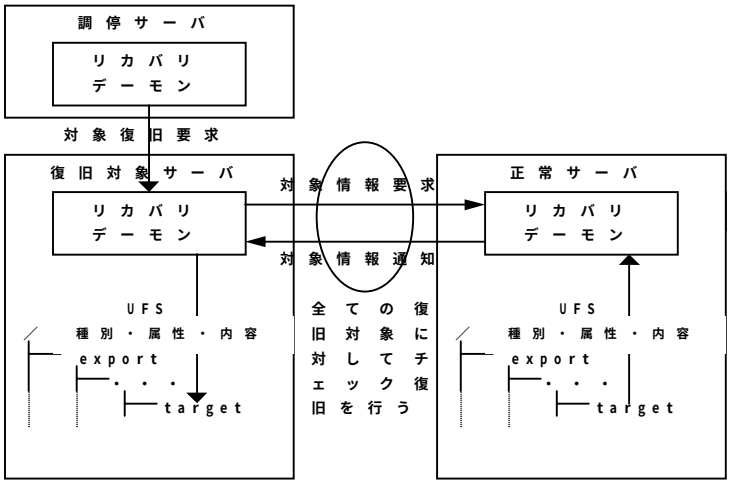
内容チェック：（内容チェックが要求されている場合）

内容チェックでは、各サーバに対して、復旧対象の内容（ファイルの場合：ファイルの内容、ディレクトリの場合：ディレクトリエントリ名、シンボリックリンクの場合：シンボリックリンク名）の確認を行い、多数決に負けた少数派のサーバのに対して、内容復旧要求を行う。

内容復旧要求は、すでに対象削除、復旧要求を行ったサーバには発行しない。

復旧要求:
　復旧対象毎に、種別、属性、内容の順に、各サーバ上の復旧対象毎に多数決を行い、サーバ間で対象の種別、属性、内容に食い違いがある場合、多数決における少数派のサーバに対して復旧要求を行う。
　復旧要求では、復旧対象と、復旧マーク、および、復旧に使用可能なサーバ（多数決時の多数派のいずれかのサーバ）の IP アドレスを通知し復旧の完了をまって、次の対象について多数決を続ける。
　復旧要求を受け取ったサーバは、通知を受けた対象について、復旧に使用可能なサーバに、復旧対象についての問い合わせを行い、自サーバ上の復旧対象の復旧を行う。
　ディレクトリに対する復旧については、復旧のためのプロセスを、fork() 後、調停サーバに復旧終了を通知する。
　fork() されたプロセスは、以後、親プロセスと独立して、対象ディレクトリ以下の全てのエントリについて、種別、属性、内容について、復旧に使用可能なサーバに問い合わせを行いながら、確認、復旧を行う。
　ディレクトリ以下の全てのエントリについて、確認、復旧が終了した時点で処理を終了する。

対象復旧



１．対象削除
　復旧対象に対して削除要求が行われた場合、復旧対象を削除する。

２．対象復旧
　復旧対象に対して復旧要求が行われた場合、復旧に使用可能なサーバに対して、復旧対象を要求し、自サーバ上に保存する。
　保存後、復旧対象の属性の修正を行う。
　復旧対象がディレクトリの場合は再帰的にディレクトリ下のエントリについて復旧が行われる。

３．属性復旧
　復旧対象に対して属性復旧要求が行われた場合、復旧に使用可能なサーバに問い合わせ、自サーバ上の対象の属性を変更する。

4． 内容復旧
　復旧対象に対して内容復旧要求が行われた場合、復旧に使用可能なサーバに問い合わせ、自サーバ上の対象の内容を変更する。
　内容更新後、復旧対象の属性の修正を行う。
　復旧対象がディレクトリの場合は再帰的にディレクトリ下のエントリについて内容復旧が行われる。

6.2. リカバリ･リクエスト、レスポンスメッセージ

リカバリ･リクエスト ： 20

リカバリ･リクエストメッセージ

形式	データ名	データの意味
U32	srvn	復旧確認を行うサーバの情報数
SRVINF [N]	srvinf	復旧確認を行うサーバの情報 N = srvn
U32	tarn	復旧対象情報数
TARINF [N]	tarinf	復旧対象情報 N = tarn

リカバリレスポンスメッセージ

形式	データ名	データの意味
U32	status	実行ステータス

6.2.1. サーバ情報の内容：

SRVINF（サーバ情報）

形式	データ名	データの意味
U32	ipaddr	復旧確認を行うサーバの IP アドレス
U32	pathlen	復旧確認を行うサーバの公開パス名長
U32[N]	pathname	復旧確認を行うサーバの公開パス名 N = (pathlen+3)>>2

6.2.2. 対象情報の内容：

TARINF（対象情報）

形式	データ名	データの意味
U32	check	０：内容チェックなし／１：内容チェックあり
U32	tarlen	対象パス名長
U32[N]	tarname	対象パス名 N = (tarlen+3)>>2
U32	offset	内容チェック時のオフセット　無効の時は 0xFFFFFFFF
U32	length	内容チェック時のレングス　　無効の時は 0xFFFFFFFF

6.2.3. リカバリ･リクエスト構造体

リカバリ･リクエストメッセージは、受信ソケットバッファから直接ＩＰＣ送信バッファに展開されるため、デコードに使用される **NRFS** サーバ内の構造体は存在しない。

6.2.3.1. リカバリ･リクエストデコーダ定義

```
int nfssvc_decode_reparirargs(struct svc_rqst *rqstp, u32 *p,
                                        struct nfsd_reparirargs *args)
```

6.2.3.2. リカバリレスポンス構造体

リカバリレスポンスでは、ステータス情報以外の情報を返却しないためエンコードに使用される固有の **NRFS** サーバ内の構造体は存在しない。

6.3. 復旧情報

IPC メッセージ：2048

復旧要求情報（RECINF）

形式	データ名	データの意味
CLIINF	cliinf	復旧要求を行ったクライアント情報 NRFSD にて付加する。
U32	srvn	復旧確認を行うサーバの情報数
SRVINF [N]	srvinf	復旧確認を行うサーバの情報 N = srvn
U32	tarn	復旧対象情報数
TARINF [N]	tarinf	復旧対象情報 N = tarn

6.3.1. クライアント情報の内容

CLIINF（クライアント情報）

形式	データ名	データの意味
U32	xid	復旧要求番号
U32	ipaddr	復旧要求を行ったクライアントの IP アドレス
U32	uid	復旧要求を行ったクライアントのユーザ ID
U32	gid	復旧要求を行ったクライアントのグループ ID

サーバ、対象情報については、6.2.1、6.2.2 参照

6.3.2. メッセージキューバッファ

調停サーバが受け取ったメッセージは、一旦以下のバッファに先読みされる。

構造体	説明
#ifndef MSGMAX #define MSGMAX 4080 #endif	ＩＰＣメッセージサイズ
#define POOL_MAX 16 #define MSG_SIZE ((MSGMAX + 0xfff) & 0x1000)	
static int msg_qid; static int msg_size[POOL_MAX]; static char msg_pool[POOL_MAX][MSG_SIZE];	ＩＰＣメッセージキューＩＤ メッセージ長配列 メッセージ格納バッファ

6.3.3. 調停サーバ内のリカバリメッセージ

リカバリメッセージは、メッセージキューバッファを経由して、調停サーバによって以下の構造体に展開される。

構造体	備考
typedef struct	
{	
unsigned long xid;	トランザクションＩＤ
unsigned long cip;	クライアントＩＰアドレス
uid_t uid;	クライアントユーザＩＤ
gid_t gid;	クライアントグループＩＤ
int serv_ok;	認証確認サーバ数
int serv_ln;	サーバ情報数
int serv_cn;	コネクションサーバ数
struct	
{	
unsigned long serv_ip;	サーバＩＰアドレス
char * serv_ex;	サーバ公開ディレクトリ
struct sockaddr_in	
serv_sk;	サーバアドレス情報
int serv_fd;	サーバのコネクションＦＤ
int serv_at;	クライアント認証結果
unsigned long type;	公開ディレクトリファイルタイプ
struct stat stat;	公開ディレクトリファイル属性
unsigned long csum;	公開ディレクトリチェックサム
unsigned long mark;	復旧マーク
} serv_rp[SERVER_MAX];	
long targ_ch;	内容チェックフラグ
char * targ_pt;	復旧対象パス
unsigned long targ_of;	復旧対象オフセット
unsigned long targ_ln;	復旧対象レングス
char	
sbuff[BUFFER_SIZE_BYTE];	送信バッファ
buffer_t sendbuff;	
char	
rbuff[BUFFER_SIZE_BYTE];	受信バッファ
buffer_t recvbuff;	
} recovery_t;	

6.4. リカバリデーモン間で取り交わされるメッセージ

調停サーバ、および、各リカバリデーモン間で取り交わされるメッセージを以下に示す。

6.4.1. クライアント認証

リカバリ処理に先立って、調停サーバと、全てのリカバリデーモンの間で取り交わされるメッセージ。

各リカバリデーモンは、調停サーバから通知されるサーバＩＰアドレス、クライアントのＩＰアドレス、クライアントがマウントしているディレクトリと、自サーバ上の mountd の dump 情報を元に、クライアントの認証を行い、認証の結果をＯＫ（０）、ＮＧ（～０）で返す。

クライアント認証で取り交わされるメッセージの書式を以下に示す。

クライアント認証要求（調停サーバ → リカバリデーモン）

形式	データ名	データの意味
U32	command	０：クライアント認証であることを示す。
U32	cip	復旧要求を行ったクライアントのＩＰアドレス
U32	sip	リカバリデーモンが実行されているサーバのＩＰアドレス
U32	pathlen	復旧確認を行うサーバの公開パス名長
U32[N]	pathname	復旧確認を行うサーバの公開パス名 N = (pathlen+3)>>2

クライアントの認証結果（
デーモン→調停サーバ）

形式	データ名	データの意味
U32	status	０　：認証ＯＫ ～０：認証ＮＧ

リカバリデーモンでは、調停サーバとの間でクライアントの認証に成功すると、次の構造体に、通信情報を記録、保持しておき、以降のチェック、復旧処理に使用する。

構造体	備考
typedef struct	
{	
unsigned long cip;	クライアントＩＰアドレス
unsigned long serv_ip;	サーバＩＰアドレス
char * serv_ex;	サーバ公開パス
unsigned long targ_ct;	パス深度カウンタ
unsigned long targ_ch;	内容チェックフラグ
unsigned long targ_of;	内容チェックオフセット
unsigned long targ_ln;	内容チェックレングス
char * targ_pt;	復旧対象のパス
unsigned long targ_type;	復旧対象のファイルタイプ
struct stat targ_st;	復旧対象の属性情報
char pbuff[BUFFER_SIZE_BYTE];	パス展開バッファ
buffer_t pathbuff;	
char wbuff[BUFFER_SIZE_BYTE];	チェックサム計算用の
buffer_t workbuff;	ワークバッファ
char sbuff[BUFFER_SIZE_BYTE];	送信バッファ
buffer_t sendbuff;	
char rbuff[BUFFER_SIZE_BYTE];	受信バッファ
buffer_t recvbuff;	
} check_t;	

クライアント認証に成功した場合、cip，serv_ip，serv_ex が、クライアント認証要求メッセージで初期化される。

setv_ex が、指し示す先の実体は、pathbuff をへて、pbuff の先頭を指すことになる。

6.4.2. 復旧対象情報要求

各リカバリデーモンは、調停サーバから復旧対象パス、内容チェックの有無、内容チェック時のオフセット、レングス値を受け取り、復旧対象の、種別、モード値、ＵＩＤ、ＧＩＤ、サイズ、チェックサム値を返す。

復旧対象情報要求で取り交わされるメッセージの書式を以下に示す。

復旧対象情報要求（調停サーバ → リカバリデーモン）

形式	データ名	データの意味
U32	command	１：復旧対象情報要求であることを示す。
U32	check	０：内容チェックなし／１：内容チェックあり
U32	offset	内容チェック時のオフセット　無効の時は 0xFFFFFFFF
U32	length	内容チェック時のレングス　　無効の時は 0xFFFFFFFF
U32	tarlen	対象パス名長
U32[N]	tarname	対象パス名 N = (tarlen+3)>>2

復旧対象情報（リカバリデーモン→調停サーバ）

形式	データ名	データの意味
U32	type	対象の種別 ０：対象は存在しない １：ファイル ２：ディレクトリ ３：リンク（シンボリック） ４：その他
U32	mode	対象のモード情報
U32	uid	対象のユーザＩＤ
U32	gid	対象のグループＩＤ
U32	size	対象のサイズ
U32	csum	対象のチェックサム

復旧対象情報要求を受け付けると、復旧要求メッセージ中の、内容チェックフラグ、内容チェックオフセット、内容チェックレングス、復旧対象のパスが、対応する(targ_で始まる)構造体に保持される。

要求を受け付けた結果として返却する、復旧対象のファイルタイプ、復旧対象の属性情報についても、対応する(targ_で始まる)構造体に保持される。

targ_ct のパス深度カウンタは、ディレクトリの再起的下降、上昇時に、パス深度の検証に使用され、復旧対象情報要求時に０が設定される。

6.4.3. 直前に復旧対象情報要求を行った対象の復旧要求

各リカバリデーモンは、調停サーバから復旧タイプ、復旧に使用可能なサーバのＩＰアドレス、公開パスを受け取り、直前に復旧対象情報を返した対象に対して復旧タイプで示される復旧を行い復旧結果をかえす。

復旧要求で取り交わされるメッセージの書式を以下に示す。

復旧要求（調停サーバ → リカバリデーモン）

形式	データ名	データの意味
U32	command	１０：復旧要求であることを示す。
U32	recType	復旧タイプ １：対象新規作成 ２：対象削除 ３：対象削除後新規作成 ４：属性復旧 ８：内容復旧
U32	sip	復旧に利用可能なサーバのＩＰアドレス
U32	pathlen	復旧に利用可能なサーバの公開パス名長
U32[N]	pathname	復旧に利用可能なサーバの公開パス名 N = (pathlen+3)>>2

復旧結果（リカバリデーモン→調停サーバ）

形式	データ名	データの意味
U32	status	０　：復旧ＯＫ ～０：復旧ＮＧ

リカバリデーモンは、復旧要求を受け付けると、直前に復旧対象情報要求を受け付けた対象について、復旧タイプで指定された復旧を試みる。

復旧時には、復旧要求時に受け取った、復旧に利用可能なサーバのＩＰアドレス、公開パス名を使用して、対象についての復旧情報を取得（ファイルタイプ、属性、データ）後、復旧を行う。

6.5. リカバリデーモン間メッセージ

.対象復旧時に、リカバリデーモン間で取り交わされるメッセージを以下に示す。

6.5.1. 復旧対象情報要求

対象復旧要求を受けたリカバリデーモンは、復旧に利用可能なリカバリデーモンに接続を行い、復旧対象パス、内容チェックの有無、内容チェック時のオフセット、レングス値を、復旧に利用可能なリカバリデーモンに送り、復旧対象の、種別、モード値、ＵＩＤ、ＧＩＤ、サイズ、チェックサム値を受け取る。

復旧対象情報要求で取り交わされるメッセージの書式を以下に示す。

復旧対象情報要求

形式	データ名	データの意味
U32	command	１：復旧対象情報要求であることを示す。
U32	tarlen	対象パス名長
U32[N]	tarname	対象パス名 N = (tarlen+3)>>2

復旧対象情報

形式	データ名	データの意味
U32	type	対象の種別 ０：対象は存在しない １：ファイル ２：ディレクトリ ３：リンク（シンボリック） ４：その他
U32	mode	対象のモード情報
U32	uid	対象のユーザＩＤ
U32	gid	対象のグループＩＤ
U32	size	対象のサイズ
U32	csum	対象のチェックサム

リカバリデーモンでは、リカバリデーモンとの間で復旧対象情報要求を受け付けると、次の構造体に、通信情報を記録、保持しておき、以降のチェック、復旧処理に使用する。

構造体	備考
typedef struct {	
unsigned long targ_ct;	パス深度カウンタ
char * targ_pt;	復旧対象のパス
unsigned long targ_type;	復旧対象のファイルタイプ
struct stat targ_st;	復旧対象の属性情報
char pbuff[BUFFER_SIZE_BYTE];	パス展開バッファ
buffer_t pathbuff;	
char sbuff[BUFFER_SIZE_BYTE];	送信バッファ
buffer_t sendbuff;	
char rbuff[BUFFER_SIZE_BYTE];	受信バッファ
buffer_t recvbuff;	
} repair_t;	

復旧対象情報要求を受け付けると、復旧要求メッセージ中の、復旧対象のパスが、対応する(targ_で始まる)構造体に保持される。

要求を受け付けた結果として返却する、復旧対象のファイルタイプ、復旧対象の属性情報についても、対応する(targ_で始まる)構造体に保持される。

targ_ct のパス深度カウンタは、ディレクトリの再起的下降、上昇時に、パス深度の検証に使用され、復旧対象情報要求時に０が設定される。

6.5.2. 直前に復旧対象情報要求を行った対象のデータ転送要求

対象復旧要求を受けたリカバリデーモンは、復旧に利用可能なリカバリデーモンに対してデータのオフセット値、レングス値を、復旧に利用可能なリカバリデーモンに送り、復旧対象の、データを受け取る。

データ転送要求で取り交わされるメッセージの書式を以下に示す。

復旧対象データ転送要求

形式	データ名	データの意味
U32	command	４：復旧対象データ転送要求であることを示す。
U32	offset	データ転送を行うオフセット　無効の時は 0xFFFFFFFF
U32	length	データ転送を行うレングス　　無効の時は 0xFFFFFFFF

復旧対象転送データ

形式	データ名	データの意味
U32	status	０　：データ転送ＯＫ ～０：データ転送ＮＧ
U32	length	転送データブロックサイズ
unsigned char [length]	block	転送データブロック
以降、全データ転送完了まで、length、block 繰り返し。 全データ転送完了時には、length として０が返される。		

復旧対象データ転送要求を受け付けると、直前に復旧対象情報要求を受け付けた対象のデータの転送を開始する。

対象がファイルの場合は、オフセット、レングスで使用されたファイルの内容、シンボリックリンクの場合は、リンク情報の転送を行う。

データ転送を行う対象のパス名、タイプ、属性は、直前の復旧対象データ転送要求で構造体内に保持されている内容が使用される。

6.5.3. 直前に復旧対象情報要求を行った対象ディレクトリのエントリ情報要求

対象復旧要求を受けたリカバリデーモンは、復旧に利用可能なリカバリデーモンに対して対象のディレクトリエントリオフセット値を、復旧に利用可能なリカバリデーモンに送り、復旧対象エントリの、種別、モード値、ＵＩＤ、ＧＩＤ、サイズ、チェックサム、対象のディレクトリエントリオフセット値、対象エントリ名を受け取る。

エントリ情報要求で取り交わされるメッセージの書式を以下に示す。

復旧対象エントリ情報要求

形式	データ名	データの意味
U32	command	２：復旧対象エントリ情報要求であることを示す。
U32	offset	対象のディレクトリエントリオフセット値 先頭のエントリ要求時は０を指定

復旧対象エントリ情報

形式	データ名	データの意味
U32	type	対象の種別 ０：対象は存在しない １：ファイル ２：ディレクトリ ３：リンク（シンボリック） ４：その他
U32	mode	対象のモード情報
U32	uid	対象のユーザＩＤ
U32	gid	対象のグループＩＤ
U32	size	対象のサイズ
U32	offset	対象のディレクトリエントリの次のエントリの オフセット値
U32	entlen	対象エントリ名長
U32[N]	entname	対象エントリ名 N = (tarlen+3)>>2

復旧対象エントリ情報要求を受け付けると、直前に復旧対象情報要求を受け付けた対象ディレクトリ内のエントリ情報を返却する。

要求を受け付けた結果として返却する、ディレクトリエントリのファイルタイプ、属性、対象名については、対応する(targ_で始まる)構造体に保持され、対象エントリについて復旧対象情報要求を受け付けたものと同じに扱われる。(つまり、この直後に、データ転送可能な状態となる。)
対象ディレクトリエントリの次のエントリのオフセット値は、システムコール、telldir(), seekdir() で使用される値と同一のものである。

6.5.4. 直前に復旧対象情報要求を行った対象ディレクトリのエントリ存在チェック要求

対象復旧要求を受けたリカバリデーモンは、復旧に利用可能なリカバリデーモンに対して対象のディレクトリエントリ名を、復旧に利用可能なリカバリデーモンに送り、復旧対象エントリの、種別、モード値、ＵＩＤ、ＧＩＤ、サイズを受け取る。

エントリ存在チェック要求で取り交わされるメッセージの書式を以下に示す。

復旧対象エントリ存在チェック要求

形式	データ名	データの意味
U32	command	２：復旧対象エントリ存在チェック要求であることを示す。
U32	entlen	対象エントリ名長
U32[N]	entname	対象エントリ名 N = (tarlen+3)>>2

復旧対象エントリ情報

形式	データ名	データの意味
U32	type	対象の種別 ０：対象は存在しない １：ファイル ２：ディレクトリ ３：リンク（シンボリック） ４：その他
U32	mode	対象のモード情報
U32	uid	対象のユーザＩＤ
U32	gid	対象のグループＩＤ
U32	size	対象のサイズ

復旧対象エントリ存在チェック要求を受け付けると、直前に復旧対象情報要求を受け付けた対象ディレクトリ中に対象エントリ名で指定されたエントリが存在するかチェックを行い、エントリが存在する場合、対象のタイプ、属性を通知する。

対象が存在しない場合、全ての値が０で通知される。

復旧対象エントリ情報要求と違い、復旧対象エントリ存在チェック要求の結果、ディレクトリエントリのファイルタイプ、属性、対象名については、対応する(targ_で始まる)構造体に保持されることはない。

7. checksum readdir

7.1 checksum readdirの概要

ＵＦＳ（およびＮＦＳ）においてｒｅａｄｄｉｒシステムコールでは、ディレクトリ内のエントリの読み出し順についての規定は無いため、NRFS クライアントが NRFS サーバに対して readdir リクエストを発行した場合、サーバ毎に異なった順番でディレクトリエントリ名を通知することになる。
一度のリクエストで、ディレクトリ内の全てのエントリを通知することができれば NRFS クライアントでサーバ間のディレクトリエントリを比較することが可能であるが、NFS（NRFS）での最大約１６ＫｂｙｔｅのＵＤＰパケットサイズを考えた場合、大量のエントリを持つディレクトリでは、複数回のリクエストの発行によりディレクトリエントリを取得する必要が生じ、NRFS クライアントで、全てのエントリを読み終えた後、サーバ間のエントリ比較を行うことは、現実的ではないと考えられる。
このため、NRFS では、NRFS サーバでディレクトリ内の全エントリ名のチェックサムを取り、チェックサムとディレクトリ内の総エントリ数を通知する checksumreaddir プロシージャを実装することとした。

checksumreaddir プロシージャの処理を行う場合に、大量のエントリを持つディレクトリに対して、毎回チェックサム計算を行うのは、処理時間を考慮した場合無駄であるため、checksumreaddir で受け付けたディレクトリのチェックサムを NRFS サーバ内にキャッシュしておき 2 度目以降に受け付けたリクエストに対しては、チェックサムキャッシュ内から、チェックサムと総エントリ数を返すものとした。
チェックサムキャッシュにおいて、チェックサムキャッシュの新鮮さを保つためチェックサムのエントリを更新する可能性のある create, remove, rename, link, symlink, mkdir, rmdir の各プロシージャに対して、リクエスト対象をエントリとして持つディレクトリのチェックサムキャッシュが存在する場合には、チェックサムキャッシュをフラッシュする処理を追加することとした。
また、NRFS を経由しないローカルでのディレクトリエントリの変更時に、i-node 情報中の i_version を使用した、チェックサムキャッシュフラッシュ機能を実装することとした。

処理概要:
checksumreaddir は、NRFS クライアントからのリクエストを受け付けると対象ディレクトリのファイルハンドルから、i-node 情報を取得し、取得した i-node 情報の i_ino（i-node 番号）, i_dev（デバイス番号）をキーとして、ディレクトリエントリチェックサムキャッシュの act（キャッシュ有／無効フラグ）フラグが有効（１）となっているものから、対象ディレクトリのディレクトリエントリチェックサムキャッシュを探す。
検索時に、対象ディレクトリに対する、ディレクトリチェックサムキャッシュが存在する場合、対象ディレクトリの i-node 情報中の i_version と、検索されたディレクトリチェックサムキャッシュの ver 値を比較し、一致していない場合は、ディレクトリチェックサムキャッシュの act 値を無効（０）にすることによりキャッシュをフラッシュした後以降の処理では、対象ディレクトリに対する、ディレクトリチェックサムキャッシュが存在しなかったものとして処理を進める。

対象ディレクトリに対する、ディレクトリチェックサムキャッシュが存在する場合は、レスポンスメッセージにディレクトリチェックサムキャッシュのチェックサム、総エントリ数をセットし、通常の readdir と同じ処理を行った後に NRFS クライアントに対してレスポンスを行う。
＊リクエスト、レスポンスメッセージについては、7.2. checksumreaddir リクエスト、レスポンスメッセージ参照
＊チェックサムキャッシュデータについては、7.3. チェックサムキャッシュテーブル参照

対象ディレクトリに対するディレクトリチェックサムキャッシュが存在しない場合には、対象ディレクトリ内のエントリを、レスポンスバッファカウント数までは、レスポンスメッセージ中にチェックサムエントリを埋め込みながら、エントリ名に対するチェックサム計算を行い、バッファカウントを超え～最後のエントリの間はエントリ名に対するチェックサム計算のみを行う。（つまり、

チェックサム計算のために、ディレクトリ中のエントリを、全て読み込む。）
　全てのエントリ名についてチェックサム計算が終了した時点で、チェックサムとチェックサム計算時にカウントしたエントリ数を、ディレクトリチェックサムキャッシュに登録し、メッセージ中に埋め込んだ後に **NRFS** クライアントに対してレスポンスを行う。

　ディレクトリチェックサム登録時には、登録対象のディレクトリの **i-node** 情報中の **i_ino**（**i-node** 番号）**, i_dev**（デバイス番号）をキーとして、ディレクトリエントリチェックサムキャッシュの **act**（キャッシュ有／無効フラグ）フラグが有効（１）となっているものを検索し、該当する、ディレクトリチェックサムが存在した場合には、該当ディレクトリのチェックサム、ディレクトリ中の総エントリ数、検索キーとして使用する **i_ino, i_dev**、キャッシュフラッシュに使用する **i_version** を設定した後 **act** を有効にする。

　登録対象のディレクトリに対する、ディレクトリエントリチェックサムキャッシュが存在しなかった場合は、ディレクトリエントリチェックサムキャッシュテーブルから、使用されていない（**act == 0**）ディレクトリエントリチェックサムキャッシュを調べ、使用されていないディレクトリエントリチェックサムキャッシュが存在した場合には、該当ディレクトリのディレクトリエントリチェックサム情報を登録する。

　ディレクトリエントリチェックサムキャッシュテーブル上に、使用されていないディレクトリエントリチェックサムキャッシュが存在しない場合は、ディレクトリエントリチェックサムキャッシュテーブル上に登録されているディレクトリエントリチェックサムキャッシュの総エントリ数を調べ、ディレクトリエントリチェックサムテーブル上に登録されているエントリ中で最小の総エントリ数と登録対象のディレクトリの総エントリ数を比較し、登録対象ディレクトリの総エントリ数の方が大きい場合には、該当チェックサムを上書きする。

　上記のどの条件にも該当しない場合は、登録対象のディレクトリに対する、ディレクトリエントリチェックサムキャッシュは登録されない。

　NRFS クライアントからの **checksumreaddir** 以外の **create, remove, rename, link, symlink, mkdir, rmdir** の各リクエスト受付時には、ディレクトリ操作時に、操作対象ディレクトリのファイルハンドルから、**i-node** 情報を取得し、取得した **i-node** 情報の **i_ino, i_dev** をキーとして、ディレクトリエントリチェックサムキャッシュの **act** が有効となっているものから、対象ディレクトリのディレクトリエントリチェックサムキャッシュを探し、対象ディレクトリに対するディレクトリエントリチェックサムキャッシュが存在する場合には、該当ディレクトリエントリチェックサムキャッシュの **act** を無効にすることによってキャッシュをフラッシュする。

　上記の各処理で使用する情報および操作の依存性から、**checksumereaddir** のディレクトリエントリチェックサムキャッシュの実装は、**NRFS** の **VFS** アクセス層に実装を行っている。

　ディレクトリエントリチェックサムキャッシュの更新操作時には、複数スレッド間の競合防止のため、カーネルセマフォを使用した排他機構を使用している。

7.2. checksumreaddir リクエスト、レスポンスメッセージ

checksumreaddir : 21

リクエスト

形式	データ名	データの意味
U32[8]	fh	ファイルハンドル
U32	cookie	クッキー
U32	count	カウント
U32	sumtype	チェックサムタイプ： ０固定
U32	sumsize	チェックサムサイズ： ４固定

レスポンス

形式	データ名	データの意味
U32	status	実行ステータス
U32	sumtype	チェックサムタイプ： ０固定
U32	sumsize	チェックサムサイズ： ４固定
U32 [(sumsize+3]>>2]	checksum	ディレクトリ内の全エントリ名のチェックサム
U32	tcpunt	ディレクトリ内の総エントリ数
U32	next	次のエントリ
ENTRY	entry	ディレクトリエントリ next != 0 の時ディレクトリエントリ next == 0 の時このフィールドは存在しない
U32	eof	終端マーク

7.2.1. ディレクトリエントリ情報の内容：

ENTRY（ディレクトリエントリ）

形式	データ名	データの意味
U32	fileid	ファイルＩＤ
U32	filelen	ファイル名長
U32 [(filelen+3)>>2]	filename	ファイル名
U32	cookie	クッキー
U32	next	次のエントリ
ENTRY	entry	ディレクトリエントリ next != 0 の時ディレクトリエントリ next == 0 の時このフィールドは存在しない

7.2.1.2. checksumreaddir のリクエスト構造体(@/include/linux/nfsd/xdr.h)

構造体	備考

struct nfsd_csumreaddirargs {
　　struct svc_fh　　fh;
　　__u32　　cookie;
　　__u32　　count;
　　__u32　　sumtype;
　　__u32　　sumsize;
};
	
	ファイルハンドル
	クッキー値
	受信可能サイズ
	チェックサムタイプ
	チェックサムサイズ

7.2.1.2.1. checksumreaddir のリクエストデコーダ定義

```
int nfssvc_decode_checksumreadargs(struct svc_rqst *, u32 *,
                                   struct nfsd_checksumreadargs *);
```

7.2.1.3. checksumreaddir のレスポンス構造体(@/include/linux/nfsd/xdr.h)

構造体		備考
struct nfsd_csumreaddirres {		
unsigned long	sumtype;	チェックサムタイプ
unsigned long	sumsize;	チェックサムサイズ
char *	sumdata;	チェックサムデータ
unsigned long	tcount;	総エントリ数
unsigned long	count;	送信ザイズ
};		＊＊

＊＊ディレクトリエントリの実体は、送信バッファにダイレクトに設定されるため構造体メンバは存在しない。

7.2.2.1. checksumreaddir のレスポンスエンコーダ定義

```
int nfssvc_encode_checksumreadres(struct svc_rqst *, u32 *,
                                  struct nfsd_checksumreadres *);
```

7.2.3. checksumreaddir のプロシージャ定義(@/fs/nfsd/nfsproc.c)

```
static int
nfsd_proc_csumreaddir(struct svc_rqst *rqstp,
                                  struct nfsd_csumreaddirargs *argp,
                                  struct nfsd_csumreaddirres  *resp);
```

7.3. チェックサムキャッシュテーブル

チェックサムキャッシュテーブル

データ名	説明
CHECKSUM_CACHE[n]	チェックサムキャッシュ ｎは、３２

7.3.1. チェックサムキャッシュの内容：

CACHE_ENTRY（チェックサムキャッシュ）

データ名	説明
act	キャッシュ有効／無効フラグ １：有効／０：無効
ino	i-node 番号
dev	デバイス番号
ver	i-node の i_version 値
tcount	ディレクトリ内の総エントリ数
sumtype	チェックサムタイプ
sumsize	チェックサムサイズ
sumdata	チェックサムデータ

7.3.2. ディレクトリチェックサムキャッシュテーブル構造(@/fs/nfs/vfs.c)

構造体等	備考
`#define NFSD_SUMSIZE_MAX 4` `#define NFSD_SUMTABL_MAX 32`	キャッシュ数
`#define CHECK_I_VERSION`	
`struct checksum_cache {` `int act;`	有／無効フラグ
`unsigned long ino;`	i-node 番号
`kdev_t dev;`	デバイス番号
`#ifdef CHECK_I_VERSION` `unsigned long ver;`	i_version 値
`#endif` `unsigned long tcount;`	以下キャッシュ情報

`unsigned long sumtype;`
`unsigned long sumsize;`
`unsigned char sumdata[NFSD_SUMSIZE_MAX];`
`};`	

`static struct semaphore csum_cache_sem;`	カーネルセマフォ
`static struct checksum_cache csum_cache_table[NFSD_SUMTABL_MAX];`	キャッシュテーブル

8. time の概要

NRFS クライアントでは、同時に複数の NRFS サーバ上のファイルシステムをマウントするため、各サーバでの管理方針の違いにより、各 NRFS から通知されるサーバ上のファイルの時間(atime, mtime, ctime 値）が、ばらばらになってしまう場合がある。

NRFS クライアントでは、ユーザに対して時間を通知する（ls－l 等で表示される時間等）場合に何らかの方法で、各 NRFS サーバと NRFS クライアントとの間で時間を調整し、NRFS クライアントので管理する時間に沿った時間通知を行う必要がある。

このため、NRFS では、NRFS サーバの現在時間を通知し、NRFS クライアントに対して時間調整を行う為の情報を提供する time プロシージャを実装することとした。

time プロシージャは、NRFS クライアントからのリクエストを受け付けると、サーバ上での現在時刻を 1970/1/1 00:00:00 UTC からの経過秒で、NRFS クライアントにレスポンスする。

＊リクエスト、レスポンスメッセージについては、8.1. time リクエスト、レスポンスメッセージ参照

8.1. time リクエスト、レスポンスメッセージ

time : 22

リクエスト

形式	データ名	データの意味
---	---	パラメータ無し

レスポンス

形式	データ名	データの意味
U32	status	実行ステータス
U32	time	サーバの現在時刻（1970/1/1 00:00:00 UTC からの経過秒）
U32	mill	ミリ秒 － 予約　0

9. NRFS の基本性能評価

デスクトップマシンとサーバマシンを使って基本性能評価とディレクトリチェックサムキャッシュの効果の評価を行った。本章ではその結果を示す。

9.1. デスクトップマシンによる基本性能評価

以下のようなスペックのデスクトップマシンを使用して、2Gbyte のファイルを local ディスクから NFS もしくは NRFS に転送する時間を測定する実験と、逆に NFS もしくは NRFS から local ディスクに転送する実験を行った（**Large File 転送**）。

- P-III 800MHz，128MB，440BX
- Celeron 850MHz/128KB，128MB（Server４）
- NIC 3COM 3C905B-J-TX
- Red Hat Linux 6.2J Second Edition / Kernel 2.2.16-3

結果は以下の通りである。ただし、NRFS3 はサーバ 3 台（3 冗長度）の NRFS、NRFS4 はサーバ４台（4 冗長度）の NRFS である。

Large File 転送（2.0GB のファイルコピー）

操作	所要時間（秒）	性能（MB/秒）
local から NFS へ	1001	2.00
local から NRFS3 へ	1647	1.22
local から NRFS4 へ	2152	0.93
NFS から local へ	967	2.07
NRFS3 から local へ	1236	1.62
NRFS4 から local へ	1231	1.62

この結果から、NRFS へのファイル書き込みの性能がかなり NFS より劣っていることが判る。ただし、このデスクトップマシンでは NFS への書き込みが 2Mbyte/sec しか性能が出ていないため、書き込み時にはすべてのサーバにデータのコピーを送っていることを考慮に入れると、この性能劣化は通信自体のオーバヘッドに拠るものだと考えられる。このため、Gigabit Ethernet 等の高速通信を使用すれば大幅に改善されるはずである。この推論の正否は、後述のサーバマシンを使った測定によって示される。読み出しがそれほど劣っていないのは、読み出し時には 1 台のサーバのみがデータ転送を行い、他のサーバはチェックサムのみを転送するため、ネットワークへの負荷が軽いからだと考えられる。

次に同一スペックのデスクトップマシンを使用して、たくさんのファイルが含まれる大きなディレクトリのコピーを行った（**Many Files 転送**）。サンプルに使ったディレクトリは Linux のカーネルソースディレクトリを複数並べて作った 2.271Gbyte のディレクトリである。

Many Files 転送（2.271ＧＢのディレクトリコピー）

操作	所要時間（秒）	性能（ＭＢ/秒）
local から NFS へ	1403	1.62
local から NRFS3 へ	2337	0.97
local から NRFS4 へ	2864	0.79
NFS から local へ	1360	1.67
NRFS3 から local へ	1652	1.37
NRFS4 から local へ	1820	1.25

やはり、NRFS への書き込みが NFS に比べて大きく劣っている。この理由は、Large File 転送の場合と同様に、100BASE-TX のネットワーク自体が飽和してしまっていると考えられる。

9.2. サーバマシンによる基本性能評価

以下のスペックのサーバマシンを使用して、デスクトップマシンと同様に Large File 転送と Many Files 転送の性能評価を行った。ただし、総データ転送サイズを若干小さくして実験を行った。このサーバ機には 100BASE と Giga の Ethernet が搭載されている。両者による性能の違いを知るために、それぞれのネットワークで実験を行った。

- P-III 1GHz, 512MB, ServerSet III HE-SL
- 100BASE-TX:intel82559, 1000BASE-SX(GbE):SysKonnect SK-9843
- Red Hat Linux 6.2J Second Edition / Kernel 2.2.16-3smp

Large File 転送（1.26GB のファイルコピー）

操作	NIC	所要時間（秒）	性能（MB/秒）
local から NFS へ	100	116.1	10.4
local から NFS へ	GbE	51.4	23.4
local から NRFS へ	100	346.2	3.47
local から NRFS へ	GbE	85.4	14.1
NFS から local へ	100	136.1	8.83
NFS から local へ	GbE	64.13	18.74
NRFS から local へ	100	158.2	7.60
NRFS から local へ	GbE	95.7	12.56

なお、この測定において NRFS はサーバ 3 台の構成である。まず、デスクトップ機よりも圧倒的に 100BASE の接続でも NFS の性能が高いことが注目される。これはマシン自体のスペックが高いことよりも、NIC のドライバの性能が大きく響いている可能性が高い。100BASE の書き込みでは NFS よりも NRFS は 3 分の 1 の性能であるが、GbE であれば 4 割程度しか性能が低下していない。
やはり、100BASE ではネットワーク負荷がかかり過ぎることが判る。また、GbE を使用した NRFS は 100BASE の NFS よりも性能が高いため、100BASE の NFS の性能に満足しているユーザであれば、GbE 環境でより高性能と高信頼性を享受することが可能である。もちろん、ディスクトラブルは少々性能がダウンしても回避したいユーザには現状の NRFS でも十分使ってもらえると考えている。

次に Many Files 転送の結果を示す。

Many Files 転送（1.429GB のディレクトリコピー）

操作	NIC	所要時間（秒）	性能（MB/秒）
local から NFS へ	100	298.6	4.56
local から NFS へ	GbE	189.3	7.20
local から NRFS へ	100	564.2	2.42
local から NRFS へ	GbE	284.1	4.80
NFS から local へ	100	308.0	4.43
NFS から local へ	GbE	234.5	5.81
NRFS から local へ	100	369.5	3.69
NRFS から local へ	GbE	301.0	4.53

単一ファイルの転送よりも NFS, NRFS 共に大幅に性能が悪い。しかし、GbE を使った NRFS は 100BASE の NFS よりも性能が高い。

9.3. ディレクトリチェックサムキャッシュの効果

デスクトップマシンとサーバマシンを使用して、ディレクトリチェックサムキャッシュが存在しない場合と存在する場合に関して、checksumreaddir の所要時間を測定した。測定したのは、クライアントがサーバマシンに checksumreaddir を発行して結果が返るまでの時間である。

デスクトップマシンとして以下のスペックのマシンを使用した。

- P-III 800MHz，128MB，440BX
- NIC 3COM 3C905B-J-TX
- Red Hat Linux 6.2J Second Edition / Kernel 2.2.16-3

デスクトップマシンの checksumreaddir の所要時間（ミリ秒）

エントリ数	1,000	2,000	10,000
総文字数	20,000	40,000	200,000
キャッシュなし	2.376	2.703	5.456
キャッシュあり	2.089	2.096	2.096

エントリ数は checksumreaddir の対象となったディレクトリが何個のエントリ（ファイルまたはディレクトリ）を持っているかを示し、総文字数はそのディレクトリに含まれるエントリの名前の総文字数である。1,000 エントリ、20,000 文字は少し想定ディレクトリが大きすぎるきらいはあるが、100BASE のネットワークではレーテンシが大きいにもかかわらず、有意な時間差が観測された。

次に、以下のスペックのサーバマシンを使用して同じく checksumreaddir の所要時間を測定した。ネットワークには GbE を使用した。

- P-III 1.26GHz，512MB，ServerSet III HE-SL
- GbE:SysKonnect SK-9843
- Red Hat Linux 6.2J Second Edition / Kernel 2.2.16-3smp

サーバマシンの checksumreaddir の所要時間（ミリ秒）

エントリ数	1,000	2,000	10,000
総文字数	20,000	40,000	200,000
キャッシュなし	0.997	1.114	2.652
キャッシュあり	0.766	0.731	0.725

デスクトップマシンに比べて、プロセッサやマシンの性能も向上しているがネットワークが格段に速くなったため、キャッシュの効果が相対的に増している。実験で想定したディレクトリはかなり大きいものであるが、最近の OS では実行バイナリのほとんどを同じディレクトリにフラットで格納するもの等が存在するため、非現実的な値であるとは言い切れない。そのため、NRFS にはディレクトリチェックサムキャッシュを実装することにした。

10. 開発の現状と今後の課題

Linux 2.2.16 カーネル用 NRFS は、2 月 15 日に開催されたシンポジウムにおいてデモ映像をお見せしたように、かなりの完成度で動いている。Linux カーネルの make 中に NRFS サーバのうちの 1 台のネットワークケーブルを故意に切り離しても、make 作業は継続される。そして、ケーブルを戻すと何事もなかったかのように、障害復旧機構によって切り離されている間に発生したサーバ間のディスク内容の不一致は解消される。このような芸当まがいのことを行うことが可能なまでになっている。また、負荷試験として故意にエラーを発生させながら、何十時間に渡って運用しても問題が発生しない。さらに、この負荷試験の最中にケーブルを抜き差しするような危険なテストも行っている。しかし、NRFS は高信頼ファイルシステムを自称する基本ソフトウェアであるため、慎重な上にも慎重なテストが必要である。何よりも、故意に発生させたエラーパタンでは、エラーを尽くせないところが非常にテストを困難にしている。

４章の開発のキーポイントで述べたように、Linux 実用版 NRFS の開発において、予期せぬ事態がいくつか発生し、デバッグのために開発予定が 2 ヶ月程度遅れてしまった。このため、当初予定のうち、GbE を使った通信のチェックサム機構を流用した NRFS の低オーバヘッド化に挑戦する時間がなかった。通信と高信頼性のための処理を融合することでオーバヘッドを削減するというアイデアは NRFS の大きな特徴の一つであるため、来年度にでもこの課題には是非挑戦したいと考えている。また、開発の遅れのために、NRFS のリリースが開発期間内にできなかったことは非常に残念である。Linux 2.2.16 カーネルベースの機構であるため、ちょっとリリース時期を逸した感は否めないが、物好きな人たちのために早くリリースを行いたいと考えている。もちろん、NRFS の使用に伴って善意の人々が何か被害を被っても何の保証もできないことは予め断っておきます。

NRFS の本格的な普及を目指すためには、以下の項目をクリアする必要があると思われる。

1. Linux 2.4.x カーネルへの対応
2. NRFS の設定・運用を簡素化するツール作り
3. ロック機構への対応
4. 性能の改善（NFS からの性能ダウンを最小限に抑える）

来年度以降これらの課題に挑戦する機会があれば、ぜひ挑戦してみたいと考えている。

NRFS の研究開発を通じて NFS の問題点が数多く実感された。あくまでも分散共有ファイルシステムつまりシステムを複数のマシンやユーザで共有しているだけで、同一のファイルが効率良く共有できることは何も保証していない。誰があるファイルを更新しても、その更新が他のマシンで観測可能になるのには少なからず時間を要する。また、ロック機構は外部の lockd として独立に提供されているだけで、ファイルシステムとしては排他性を何も保証していない。高信頼分散共有ファイルシステム上に、高性能かつ高信頼のデータベースシステムを構築してみたいと考えていたが、上記のような NFS の性質はデータベースのストレージとしては適しているとは言い難い。そこで、開発者が考案提唱しているメモリベース通信を活用したデータベース用高性能ストレージの検討も NRFS の改良と共に今後は進めていきたいと考えている。