修 士 論 文 の 和 文 要 旨

大学院情報システム学研究科	博士前期課程	情報システム基盤学 専 攻
氏 名	神保 直幸	学籍番号 1253006
論 文 題 目	ファイル同期機能を持つファイルシステムの設計と実装	

要 旨

近年スマートフォンなどのモバイル端末の普及が進み，これらの端末間でのファイル共有が広く利用されている．一般にファイル共有の手法として，ネットワークファイルシステムなどによるファイルの一元管理や，端末間でのファイル同期が挙げられる．

これらのうち，モバイル端末間のファイル共有においては，ファイルがサーバにあるためオフラインアクセスができない一元管理よりも，オフラインでもファイルにアクセス可能なファイル同期が適しており，今後の主流になると考えられる．ファイル同期の例として，ファイル同期ツールやオンラインストレージサービスなどがある．これらの方式ではいずれも端末上のアプリケーションが同期を処理している．同期のためには同期対象のファイルにアクセスする必要があるが，アプリケーションがアクセスできるファイルが制限されているサンドボックス環境では，上記のいずれの方式も使えない．また，これらの方式ではファイルをパスで指定するため，ファイルを移動すると同期設定が維持できないという問題もある．

本研究では端末間でのファイル同期を行うファイルシステムである FSyncFS を提案する．FSyncFS はカーネル空間で動作するため，サンドボックス環境におけるファイルアクセスの制約を受けずに同期処理を行うことができる．また，それぞれのファイルに同期設定情報を持たせることでファイルごとの同期設定を可能にし，ファイルのパスに依存しないファイル同期を実現する．

FSyncFS のプロトタイプとして，PC 上で動作する Linux のファイルシステムを実装した．プロトタイプはスタッカブルファイルシステムとして実装したため，任意のファイルシステムにファイル同期機能を追加することができる．端末間でのファイル同期は専用の同期サーバを介して行われる．また，ローカルファイルアクセスと同期サーバとの通信を並行に動作させ，前者の遅延を少なくした．

本提案の FSyncFS により，アプリケーションがサンドボックス環境で動作するモバイル端末において，基盤的機能としてのファイル同期を，広くアプリケーションに提供できる．

（大学院情報システム学研究科）

平成25年度 修士論文

ファイル同期機能を持つ
ファイルシステムの設計と実装

電気通信大学 大学院情報システム学研究科

情報システム基盤学専攻

1253006　神保 直幸

指導教官　鶴岡 行雄　教授
多田 好克　教授
本多 弘樹　教授

提出日　平成26年2月21日

目 次

図 目 次

表 目 次

第 1 章

はじめに

近年スマートフォンなどのモバイル端末の普及が進み，これらの端末間でのファイル共有が広く利用されている．ファイル共有の利用例として，写真などのメディアファイルを個人が持つ端末間で共有する場合や，複数のユーザからなるグループ内でのドキュメントの共有などがある．一般にファイル共有の手法として，サーバによるファイルの一元管理や，端末間でのファイル同期がある．

ファイルの一元管理を行うための手法として，NFS[1] や Coda[2] などのネットワークファイルシステムを用い，複数の端末から 1 つのサーバ上にあるファイルシステムにネットワーク経由でアクセスを行う方法がある．しかし，この方法では端末がオフラインの際にはサーバ上のファイルにアクセスできない．そのため，特にオフラインでのファイル操作が必要となるスマートフォンなどのモバイル端末では利用できない．また，ファイルアクセスにサーバとの通信によるオーバヘッドが発生するため，無線通信を伴うインターネット環境など，低速なネットワーク環境での利用には適さないという問題がある．

一方，端末間でのファイル同期は，複数の端末上にファイルを複製し，それぞれがローカルに持つファイルを最新のバージョンに保つという目的で利用される．このアプローチではファイルの実体がそれぞれの端末上に存在する．そのため，オフライン環境でのファイルアクセスが可能であり，モバイル環境での利用に適している．また，ファイルアクセスはネットワークを利用しないローカルディスクへのアクセスであるため，通信に起因するオーバヘッドが発生しないという利点がある．

今後はより一層モバイル端末の利用が増加するため，一元管理ではなく同期によるファイル共有が主流となると考えられる．

端末間でデータを同期する方法として，特定のアプリケーションのデータを同期するものと，任意のファイルを同期する汎用的な方法がある．前者の例として，Microsoft の Exchange Server と Outolook を用いた電子メールやスケジュールの同期がある．Microsoft の Exchange Server は，グループウェア機能を持つメールサーバで，サーバ上で管理されるデータを端末上のデータと同期する．しかし，この方法は他のアプリケーションでは利用できない．本研究ではアプリケーションに特化した方法で同期を行うのではなく，多くのアプリケーションから利用できる汎用的なファイル同期の方法を考える．そのような方式として，rsync[5]，Unison[6] などのファイル同期ツールや，Dropbox[3] などのオンラインストレージを利用したファイル同期サービスがある．

端末間で同期を行う場合，端末同士が直接データをやり取りする方法と，サーバを介して通信する方法がある．モバイル環境では端末がオフラインになる場合があるため，本研究では後者の方法を考える．

同期についてはローカルディスクへの変更をサーバに伝えることが必要である．これを自動的に行うためには，ファイルの更新を検出することと更新で生じた差分を抽出してサーバに送ることが必要である．また，他の端末が行ったファイルの更新をサーバから受け取り，ローカルディスクに反映させることも必要である．これらの処理をアプリケーションが行う方法と，OS のカーネルが行う方法がある．

アプリケーションが行う方法では，OS が提供する API によってファイルの更新を検出し，rsync や Unison 等のファイル同期ツールによる差分の抽出，サーバへの送受信とローカルディスクへの反映を行う．Dropbox も同様にアプリケーションで同期を行っている．

NFS はカーネルでファイル共有を行う方法だが，これは一元管理の手法であるため，同期では利用できない．

更新の検出と差分の抽出を OS のカーネルが行い，サーバへの送受信とローカルへの反映をアプリケーションが行うものとして，Intermezzo[4] がある．Intermezzo では一元管理を行うが，ローカルキャッシュによるオフライン操作が可能である．

ところで，Android 端末や iPhone などのモバイル端末では，サンドボックス環境が採用されている．サンドボックスは，悪意のあるアプリケーションからの不正なアクセスを防止するために，アプリケーションがアクセスできるリソースが制限される環境である．したがって，アプリケーションが他のアプリケーションが管理するファイルにアクセスできない．

rsync や Unison などのファイル同期ツールはアプリケーションとして実装されているため，サンドボックス環境では利用できず，各アプリケーションが個別にファイル同期を行う必要がある．たとえば，ブラウザやメーラが個別にファイル同期を行う必要がある．この方法ではアプリケーションに手を加える必要があるため，既存のアプリケーションは同期機能を利用できない．

Intermezzo については，サーバから受信した更新をローカルディスクに反映する処理をアプリケーションが行う．そのため，アプリケーションが他のアプリケーションのファイルにアクセスできないサンドボックス環境では利用できない．

したがって，ファイルやディレクトリの監視と，他端末での更新のローカルディスク上のファイルへの反映を OS のカーネル空間で行わなければサンドボックス環境でのファイル同期は実現できない．

さらに，既存のファイル同期ツールやオンラインストレージサービスでは，同期対象となるファイルをパスによって指定しているため，ファイルを移動すると同期設定が維持できないという問題もある．

本研究では，アプリケーションではなくファイルシステムで同期を行うアプローチを選択し，端末間でのファイル同期を行うファイルシステムである FSyncFS を提案する．

FSyncFS はカーネル空間で動作する．そのため，アプリケーション層からファ

イルやディレクトリの変更を監視する必要がなく，サンドボックス環境でもバックグラウンドでの自動的なファイル同期を実現できる．FSyncFS はユーザプロセスがファイル操作を行った際，操作の種類とその引数をファイルアクセス情報として抽出して同期サーバに送信する．そのため，従来のファイル同期アプリケーションのように同期処理の際にファイルの更新差分を求める必要がない．

また，FSyncFS では同期に関する設定（同期設定情報）をファイルのメタデータとしてそれぞれのファイルに付与することで，ファイルごとの同期管理をパスに依存せずに行うことを可能にする．同期設定情報を編集するための API を FSyncFS が提供することで，ユーザプロセスからファイルごとの同期設定を適切に変更できる．

第 2 章

関連研究

2.1 ファイル同期ツール

rsync は，以前から利用されている代表的なファイル同期ツールである．rsync はファイル・ディレクトリのバックアップやミラーリングを行うためのアプリケーションである．転送元もしくは転送先としてリモートの端末上のファイルやディレクトリを指定することで，端末間でのファイル同期が可能となる．ファイルを転送する際，rsync のアルゴリズムで更新の差分を抽出し，同期の際の通信量を削減している．rsync のアルゴリズムではファイルを固定長のチャンクに分割し，チャンクごとのローリングチェックサムを送信元と送信先で比較することで更新の差分を生成している．rsync のアルゴリズムを用いた別のファイル同期ツールとして，Unison がある．rsync による同期が転送元と転送先を固定した一方向性であるのに対し，Unison は転送元と転送先を区別しない双方向のファイル同期を行う．

これらのアプリケーションはファイル同期のためにユーザがアプリケーションを明示的に実行する必要がある．ユーザプロセスによるファイルの更新時に自動的にこれらのファイル同期ツールを実行するために，OS が提供するファイル・ディレクトリ監視 API を利用することができる．以下でこの API について説明する．

2.1.1 ファイル・ディレクトリ監視 API

ファイル・ディレクトリ監視 API の例として，Linux の inotify や Windows の FindFirstChangeNotification，OS X の FSEvents などがある．これらの API は OS ごとに仕様が異なるが，同様の処理を行うため，ここでは inotify について述べる．

inotify は，ディレクトリやファイルに対する操作をイベントとして検出し，アプリケーションに通知する API である．inotify におけるイベントとは，ディレクトリやファイルに対する作成・編集・削除・移動を指し，以下のフラグで表される．

IN_ACCESS

ファイルがアクセス (read) された．

IN_ATTRIB

メタデータが変更された．

IN_CLOSE_WRITE

書き込みのためにオープンされたファイルがクローズされた．

IN_CLOSE_NOWRITE

書き込み以外のためにオープンされたファイルがクローズされた．

IN_CREATE

監視対象ディレクトリ内でファイルやディレクトリが作成された．

IN_DELETE

監視対象ディレクトリ内でファイルやディレクトリが削除された．

IN_DELETE_SELF

監視対象のディレクトリまたはファイル自身が削除された．

IN_MODIFY

ファイルが修正された．

IN_MOVE_SELF

監視対象のディレクトリまたはファイル自身が移動された．

IN_MOVED_FROM

ファイルがリネームされた際，リネーム前のパスについて発生する．

IN_MOVED_TO

ファイルがリネームされた際，リネーム後のパスについて発生する．

IN_OPEN

ファイルがオープンされた．

イベントを監視すべきファイルやディレクトリを監視対象として指定し，監視対象ディスクリプタによって一意に識別する．監視対象としてディレクトリを指定している場合，IN_ACCESS や IN_ATTRIB などはそのディレクトリ以下のファイルやディレクトリの操作に対しても発生する．

inotify では，監視対象で発生したイベントをイベントキューとして保持するための inotify インスタンスを用いる．アプリケーションへのイベントの通知は inotify インスタンスを介して行われる．1 つの inotify インスタンスは監視対象リストを持ち，複数の監視対象を監視できる．アプリケーションはファイルディスクリプタを用いて inotify インスタンスを参照する．inotify で用いられるシステムコールについて以下で説明する．

int inotify(void);

カーネル中に inotify インスタンスを作成する (初期化)．初期化された inotify インスタンスを示す新しいファイルディスクリプタを返す．

int inotify_add_watch(int fd, const char *pathname, uint32_t mask);

pathname で指定したディレクトリやファイルを監視対象として，fd で指定した inotify インスタンスの監視対象リストに追加する．mask によって監視

するイベントを一つもしくは複数指定する．監視対象と一意に対応する監視対象ディスクリプタを返す．なお，監視対象ディスクリプタはファイルディスクリプタではなく，イベントがどの監視対象で発生したものかを識別するための識別子である．

ssize_t read(int fd, void *buf, size_t count);

ファイルディスクリプタ fd で指定した inotify インスタンスからイベントを取得する．一つ以上のイベントが発生するまで read はブロックされる．成功すると以下の構造体をバッファbuf に書き込む．

```
struct inotify_event {
    int      wd;       /* 監視対象ディスクリプタ */
    uint32_t mask;     /* イベントのマスク */
    uint32_t cookie;   /* イベント群を関連づける一意なクッキー（rename
用）*/
    uint32_t len;      /* 'name' フィールドのサイズ */
    char     name[];   /* NULL で終端された任意の名前. 監視対象がディレクトリである場合にイベントが発生したディレクトリ内のファイルの名前が格納される */
};
```

cookie にはファイルやディレクトリの移動について，移動元と移動先を結びつけるための一意の整数が設定される．監視対象を移動したとき（IN_MOVED_SELF 発生時），移動後もイベントを監視することができるが，移動後の監視対象のパスはわからない．監視対象のパスに関しては inotify_add_watch() で指定した情報以上のことは得られない．また，複数のリンクを持つファイルを監視した際，inotify_add_watch() で指定したパスへの操作は監視できるが，ほかのリンクへの操作は監視できない．wd はイベントが発

生した監視対象を表す監視対象ディスクリプタを示す．mask は発生したイベントの種類を表す．1 イベントの読み込みには read() の count に sizeof(struct inotify_event)+NAME_MAX+1 を指定すれば十分である．発生したイベントは inotify インスタンスがイベントキューとして保持するため．複数のイベントを監視することができる．

int inotify_rm_watch(int fd, int wd);

fd で指定した inotify インスタンス中の監視対象リストから，wd で指定した監視対象を削除する．成功すると 0 を返す．

int close(int fd);

fd で指定した inotify インスタンスをクローズする．

inotify を利用して特定のファイルやディレクトリを監視し，変更があったタイミングで rsync や Unison などのファイル同期ツールを実行することで自動的なファイル同期を実現できる．

2.2 オンラインストレージを用いたファイル同期

Dropbox や iCloud[7] などのオンラインストレージサービスを用いたファイル同期が近年利用されている．これらのサービスは特定のパスにあるファイルやディレクトリを他の端末と同期するという点では 2.1 で述べたファイル同期ツールと同じだが，ストレージスペースをサービスとして貸し出すという点で異なる．オンラインストレージサービスを非サンドボックス環境で利用する場合とサンドボックス環境で利用する場合では動作が異なる．

2.2.1 非サンドボックス環境での利用

ファイルの同期はインターネット上のサーバを介して行われ，クライアント上では専用のアプリケーションが動作する．ファイルやディレクトリの監視のために，クライアントアプリケーションは 2.1.1 で述べた監視 API を用いている．具体的には，Dropbox の場合，Linux では inotify を，Windows では FindFirstChangeNotification を利用している．端末上に専用のディレクトリが作成され，そのディレクトリ以下のファイルやディレクトリを同期対象とする．Google Drive[8] や Sky Drive[9] などのサービスでも Dropbox と同様のインタフェースを提供している．1 章で述べたように，この方法ではサンドボックス環境でのファイル同期を実現できない．そのため，各アプリケーションに同期機能を持たせるための API が提供されている．2.2.2 でその方法と問題点について述べる．

2.2.2 サンドボックス環境での利用

サンドボックス環境で，アプリケーションが用いるファイルの同期を行うには，そのアプリケーション自身がファイル同期を行わなければならない．ファイル同期にオンラインストレージサービスを利用する場合，アプリケーションはオンラインストレージサービスが提供する API を用いて同期機能を実装する必要がある．たとえば Dropbox の場合，Android や iOS などの OS 向けにそのような API を提供している．この場合，アプリケーションを起動するまで同期が行えず，利用するファイルの最新版をサーバから取得できない．このことは端末がオフラインになるタイミングが予測できないモバイル環境で問題となる．

たとえば，2 つのモバイル端末 A，B でアプリケーション App が利用するファイル F を同期することを考える．A によるファイルの更新がサーバ送信されたとき，B がこれを取得するためには以下の条件を満たす必要がある．

- B がオンラインであること

- B 上で App が起動されていること

サーバへの情報の送受信は App が行うため，B は App を起動するまで F の最新版を取得できない．また，B がオフラインのとき B はサーバと通信できないため，B 上で App が起動されていてもオンラインになるまでサーバ上の更新情報を取得できない．

2.3 ネットワークファイルシステム

ネットワークファイルシステムは，端末からネットワークを介してサーバのファイルシステムにアクセスするために利用される．複数の端末から共通のサーバを利用することで，端末間のファイル共有が可能である．ネットワークファイルシステムはアクセス透過性と位置透過性を備えている．ここで，アクセス透過性とは端末がローカルファイルへのアクセスと同様に，ファイルアクセスのインタフェースによってサーバ上のファイルを利用できることを指す．これに対して，たとえば FTP などのファイル転送プロトコルでは，サーバ上のファイルに直接アクセスすることができない．また，位置透過性とは，ファイルが実際におかれている場所を意識する必要がないことを指す．これらの特性によって，端末はサーバ上のファイルをあたかもローカルにあるかのようにアクセスすることができる．ネットワークファイルシステムの代表的な例として，NFS がある．

NFS は主に UNIX 系のシステムで古くから利用されているネットワークファイルシステムである．TCP/IP ベースの RPC を用いた NFS プロトコルが標準仕様として公開されており，多くの UNIX 系システムで利用できる．NFS は LAN での利用を想定しており，高速で安定したネットワーク環境下では高いパフォーマンスで動作する．しかし，インターネットのような低速ネットワークを介して NFS を利用する場合，ファイルアクセスの速度が低下するという問題がある．また，端末をオフラインで利用する場合，サーバ上のファイルにアクセスすることができな

い．さらに，NFS を用いてサイズの大きいファイルにアクセスする場合，ファイルアクセスのための通信が他のトラフィックを妨げるという問題もある．以上の理由から，端末がオフラインの状態でのファイルアクセスが必要なモバイル環境では，NFS は利用できない．

これらの問題を解決するための手法として，端末にローカルキャッシュを備える方法がある．オフライン環境やネットワークが不安定な環境で利用できるネットワークファイルシステムの例として，Coda や Intermezzo がある．

Coda は NFS と同様，サーバのファイルシステムにネットワークを介してアクセスするために用いられるが，ローカルディスク上にファイルのキャッシュを保存することで，オフライン時のファイル操作ができる．クライアントのアプリケーションがファイルアクセスを要求すると，キャッシュマネージャがサーバ上のファイルをローカルキャッシュにコピーし，ファイルアクセスのために行うサーバとの通信を削減する．また，ローカルキャッシュに保存されたファイルへのアクセスはネットワークを介さずに行われるため，オフラインでの利用が可能である．Coda におけるファイルのオフライン操作はネットワークとの切断があらかじめ予測できる環境での利用を想定している．Coda を利用する端末をネットワークから切断する際，ユーザは必要なファイルをローカルキャッシュに取得するためのコマンドである hoard を実行する必要がある．しかし，現在のモバイル環境では端末がネットワークから切断されることが予測できず，Coda のオフライン操作では不十分である．

Intermezzo は Coda に影響を受けたネットワークファイルシステムで，オフライン操作やモバイル環境での利用により適している．Intermezzo はクライアント–サーバモデルで動作し，クライアントはサーバ上のファイルシステムのコピーをローカルディスク上に持つ．そのため，端末がオフラインの状態でもファイルアクセスが可能である．Intermezzo ではオフライン時に行われたファイルやディレクトリの変更を KML(Kernel Modification Log) ファイルとしてローカルディスクに記

録し，InterSync と呼ばれるクライアントとサーバ間での同期を行うツールによってクライアントとサーバ間で KML ファイルの送受信が行われる．カーネル空間で動作する Intermezzo は，ユーザプロセスからのファイルアクセスに応じて KML を生成する．

Intermezzo はオフライン時のファイル可用性が高く，モバイル環境での利用に適したネットワークファイルシステムである．しかし，Intermezzo は他のネットワークファイルシステムと同様，ファイルシステム単位でファイルやディレクトリの一元管理を行うため，ファイル共有の設定情報をファイルごとに管理することができない．共有設定の管理について，モバイル環境では以下のような要求がある．

- モバイル端末ではストレージ容量に制限があるため，容量の大きいファイルやその端末では利用頻度が低いファイルなどを同期対象から除外したい
- モバイル端末と PC 端末でファイルの共有を行うとき，それぞれの端末でファイルを異なるポリシーで整理したい

後者の要求における「整理」について，たとえばスマートフォンにドキュメントを編集・閲覧するアプリケーションが複数ある場合，ドキュメントファイルはアプリケーションごとにまとめて同じディレクトリに置かれる．しかし，PC ではこれらのドキュメントファイルを同じ種類のファイルとして同じディレクトリ内で管理することがある．また，ある端末でファイルやディレクトリの移動を行うと他の端末にもその操作が反映され，端末ごとに独立してディレクトリ構造を管理できないという問題もある．端末間のファイル共有をファイルシステム単位で行うネットワークファイルシステムでは，これらの要求に応えることができない．

2.4 ファイルシステムへの機能の拡張

一般にファイルシステムはディスク上のデータをファイルやディレクトリという形式でユーザ空間に提供するものであり，ファイルへの書き込みやファイルの読

み込み，作成や削除などのファイルアクセスのためのインタフェースを提供する．これによってユーザ空間ではデータがディスク上で物理的にどのように格納されているかを意識せず，抽象的に管理できる．1 章で述べたように，本研究ではファイルシステムによる端末間でのファイル同期を行う．そのためには，一般的なファイルシステムが持つ機能を拡張する必要がある．

ファイルシステムに新たな機能を追加するためには，既存のファイルシステムにコードを追加する方法と，既存のファイルシステムと連携する，独立したモジュールを実装する方法がある．

前者の例として，ext2 の機能を拡張し，ジャーナリング機能を追加した ext3 がある．この方法では物理的なデータのレイアウトなど，ハードウェアと密接に関係する部分まで専用に設計できるという利点があるが，任意のファイルシステムの機能を一般的な方法で拡張するものではない．

後者の方法では拡張したい機能のみを実装した独立したモジュールと，任意のファイルシステムを組み合わせて利用できる．この場合，モジュールをユーザ空間ファイルシステムとして実装する方法とローダブルカーネルモジュール (LKM) であるスタッカブルファイルシステム [16] として実装する方法がある．

前者のフレームワークとして，FUSE[10] や LUFS[11] などがある．これらを利用する場合は，開発やメンテナンスのコストが低いことや，柔軟性が高く，実装によって多様なリソースをファイルシステムとして提供できるという利点がある．FUSE による実装例として，ファイルアクセスのインタフェースによって Wikipedia の記事の閲覧や編集を可能にするもの [12] や，GMail のストレージに E メール形式でデータを保存するファイルシステム [13] などがある．しかし，これらはユーザ空間で動作するため，カーネル空間で動作するものに比べてコンテキストスイッチの回数が多く，性能が低いという欠点がある．

LKM として実装する方法では，モジュールはカーネル空間で動作する．この方法による実装は，ユーザ空間ファイルシステムの実装よりも高速に動作する．ス

タッカブルファイルシステムは論理的にはVFS層とローカル・ファイルシステムの中間層で動作し，下層で動作するファイルシステムを任意に選択できるという特徴がある．スタッカブルファイルシステムの実装例として，複数のファイルシステムのUnion mountを実現するAufs[14]や，モバイル端末からクラウドストレージ上のデータにアクセスするためのRFS[15]などがある．

Aufsは，ローカルのファイルシステムをブランチとして扱い，複数のブランチを透過的に重ね合わせて単一のディレクトリツリーとして提供するものである．read-onlyのブランチに対する書き込みを提供するためのcopy-on-writeや，特定のファイルをユーザに対して隠蔽するWhiteoutなどの機能がある．

RFSは，バッテリーの残量やネットワーク状況など，モバイル端末の状態を考慮してクライアントセントリックなキャッシュマネジメントを行うネットワークファイルシステムである．これはモバイル環境における端末間でのファイル共有を目的としている点では本研究と同じである．しかし，サンドボックス環境での利用やファイルごとの共有設定について考慮されていないと言う点で本研究と異なる．

第 3 章

提案システムの設計

FSyncFS の設計について述べる．FSyncFS では共通の同期サーバを介して複数端末間でのファイル同期を行う (図 3.1)．

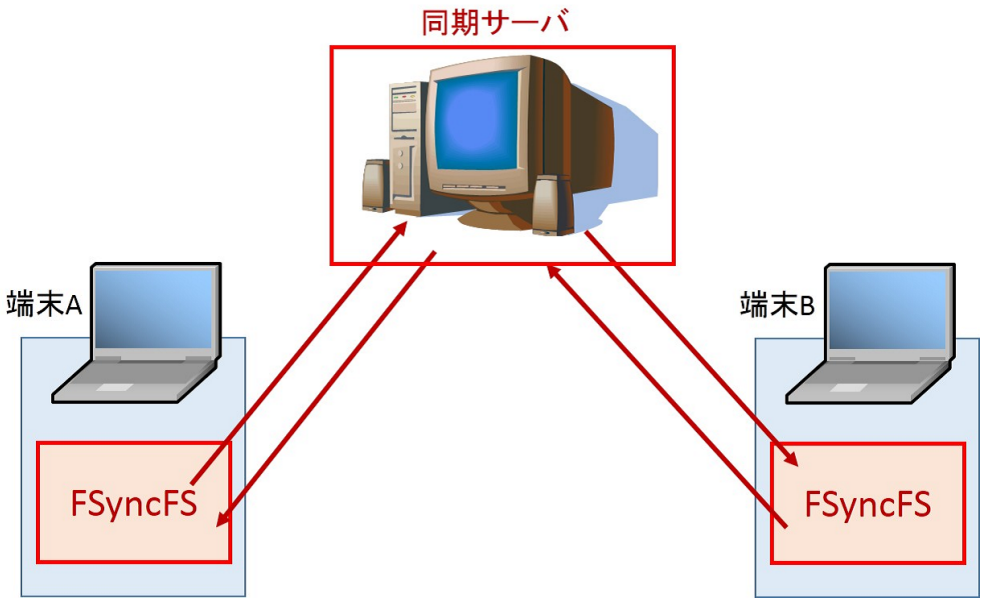

図 3.1: FSyncFS の概要

FSyncFS はローカルディスク上のファイルへの操作に加え，後述する手順に基づいて同期サーバとの通信を行い，ファイルの変更情報を送受信する．端末上で実行されたファイル操作を他の端末に伝播させるために同期サーバを用いる．ここでは，端末上で動作するファイルシステムと，端末からのメッセージを処理するた

めの同期サーバを設計する．

3.1 前提条件と要求条件

提案システムであるFSyncFSの設計にあたって，ユーザと端末が満たすべき前提条件は以下の通りである．

1. ユーザは識別のためにグローバルユニークなユーザIDを持つ
2. 端末は単一のユーザによって利用される
3. 1人のユーザが複数の端末を利用する

FSyncFSが満たすべき要求条件は以下の通りである．

1. ファイル同期のためのユーザによる明示的な操作を必要とせず，同期が自動的に行われること
2. OS環境ごとの差異を考慮し，ファイルのパスに依存しないファイル同期を行うこと
   たとえば，同期対象ファイルを移動しても同期設定を維持できる．また，同期対象ファイルを端末ごとに異なるパスに置くことができる
3. サンドボックス環境をもつモバイル端末上でも動作すること

3.2 構成と機能

FSyncFSの構成を図3.2に示す．FSyncFSはカーネル空間で動作し，他のクライアント端末との同期は同期サーバを介して行われる．端末上で実行されたファイル操作は後述するファイルアクセス情報に変換され，端末と同期サーバの間で送受信される．FSyncFSでは各端末が持つファイルを，ファイルIDによって関連付け

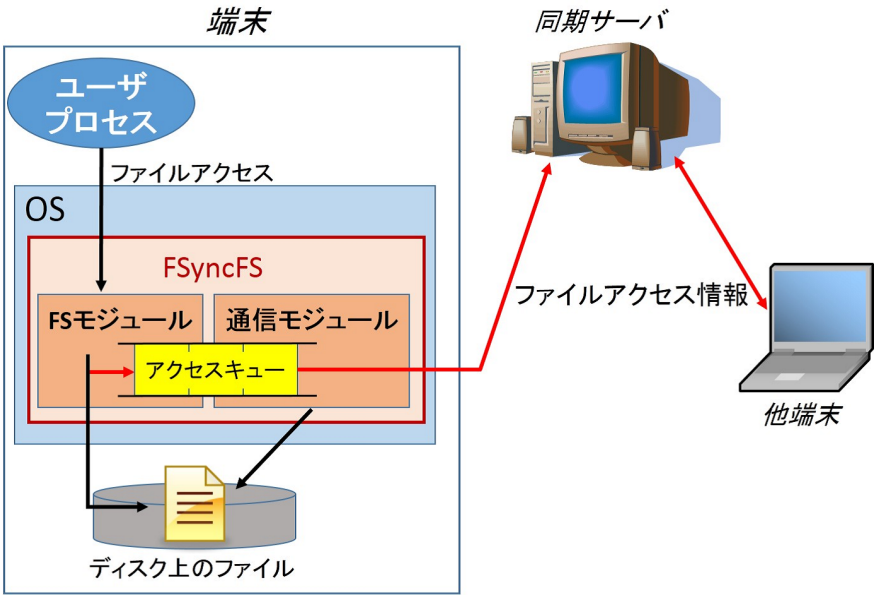

図 3.2: 端末の構成

る．同じファイル ID を持つファイルはファイル名が異なっていても同じファイルとして扱い，同期サーバはファイル ID やそのファイルを持つ端末の情報を持つ．

端末は FSyncFS を利用するための設定情報を，設定ファイルとして持つ．設定ファイルには，同期設定情報の初期設定，端末番号，利用する同期サーバの IP アドレスまたはドメイン名を記録する．同期設定情報の初期設定は，端末上で新規作成されたファイルに付与するデフォルトの同期設定情報である．端末番号は，個人が利用する複数の端末を識別するためにユーザが与える番号である．ユーザ ID と端末番号をあわせたものを，以降では端末 ID と呼ぶ．利用する同期サーバの IP アドレスまたはドメイン名は，他端末が行ったファイル操作を同期サーバから取得するために利用する．

3.2.1 クライアントの構成と機能

端末上の FSyncFS は FS モジュール，通信モジュール，アクセスキューから構成される．また，FSyncFS 上のファイルは，同期に必要な設定情報 (同期設定情報) をメタデータとしてファイルごとに持つ．FS モジュールはローカルファイルの操作を行い，実行された操作を表すファイルアクセス情報をアクセスキューに追加する機能を持つ．ファイルアクセス情報には，実行されたファイル操作の種類 (create や write など) とその引数，同期設定情報が含まれる．通信モジュールはアクセスキューから FS モジュールによって追加されたファイルアクセス情報を取得し，同期サーバにその情報を送信する．また，定期的に同期サーバに問い合わせ，他端末が実行したファイル操作のファイルアクセス情報を取得し，ローカルディスク上のファイルに反映する．

3.2.2 同期サーバの構成と機能

FSyncFS の同期サーバの構成を図 3.3 に示す．同期サーバは通信モジュール，ID-Path 変換機構，競合検出機構，各端末に対応するクライアントキューから構成される．

通信モジュールは端末からのファイルアクセス情報を待ちうけ，他端末に対応するクライアントキューに追加する機能を持つ．この際，ファイルアクセスの競合があった場合には，競合検出機構がこれを検出してファイルを複数のバージョンにフォークさせる．また，各端末からの問い合わせを待ち，ファイルアクセス情報を送信する機能を持つ．また，ID-Path 変換機構は，ある端末から送信されたファイルアクセス情報中のファイル ID を他の端末上でのパスに変換し，それぞれの端末に対応するクライアントキューに追加する．各端末は各々のクライアントキューから，他端末からのファイルアクセス情報を取得する．

管理情報テーブルにファイルのパスと ID の対応情報や各端末上のファイルが最

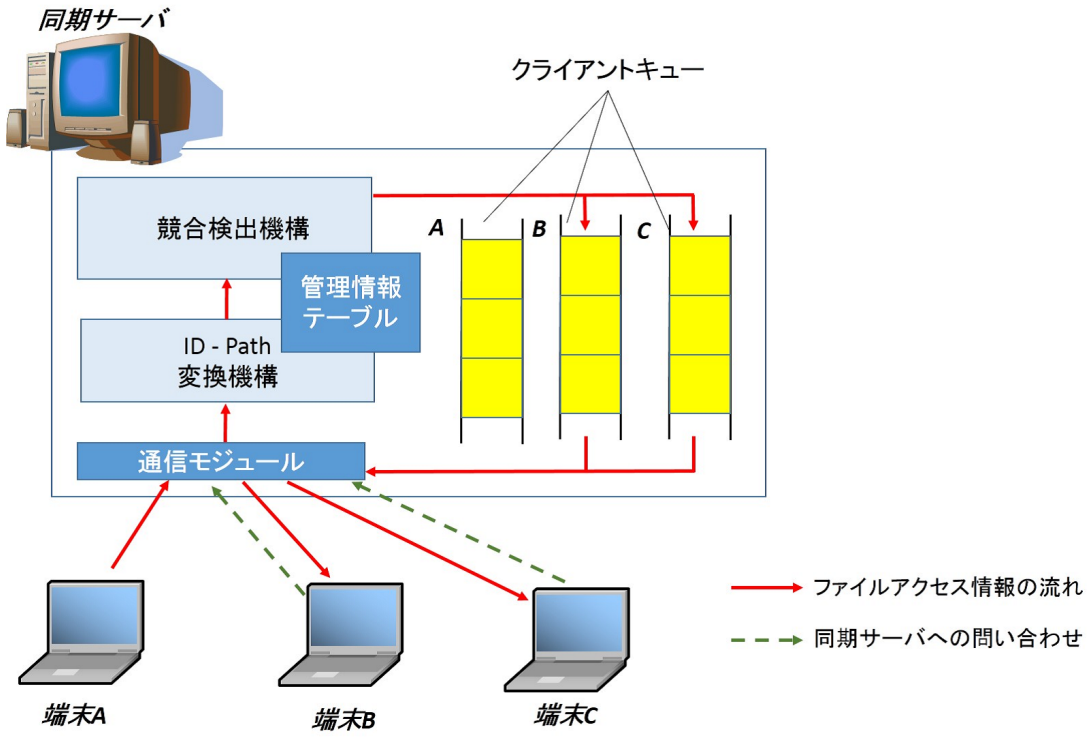

図 3.3: 同期サーバの構成

新かどうかを表す情報を記録し，ID–Path 変換機構と競合検出機構がこれらの情報を利用する．

3.3 同期設定情報

FsyncFS では，ファイル同期に必要な情報である同期設定情報をそれぞれのファイルがメタデータとして持つ．同期設定情報はファイルの作成時に，設定ファイルに書き込まれた初期設定に従って設定される．設定ファイルには同期設定情報の初期設定の他に，問い合わせを行う同期サーバの IP アドレス（またはドメイン名）や他の端末で新規に作成されたファイルを置くディレクトリのパスなど，ファイル同期のために FSyncFS が利用する情報が書き込まれる．

同期設定情報はファイルの実体と関連付けて管理され，ユーザプロセスは後述する API によってその内容を変更することができる．同期設定情報として以下のものを定義する．

同期モード

そのファイルが同期対象であるかどうかを表す情報．ON または OFF で表す．同期が ON であれば以下の同期設定情報に基づいて同期処理を行う．OFF であればローカルのみで利用されるファイルとして扱われる．

同期サーバ情報

そのファイルが同期のために利用するサーバの IP アドレス（またはドメイン名）．

ファイル ID

ファイルに付与されるグローバルユニーク ID．ユーザ ID と端末番号および FSyncFS が管理する端末内でユニークな番号を用いて生成される．

アクセス制御情報

同期に参加する端末のパーミッション情報．その端末を使用するユーザのユーザ ID を用いて指定する．指定する情報は、R/W/X/S(同期)/M(メタデータ編集) である．これらのパーミッション情報とユーザ ID（および明示的に指定されていないユーザを表す “others”）の組み合わせによってアクセス制御情報が指定される（例　“user1@example.com;rw-s-,user2@example.com;rw-sm,others;r–s-”）．同期パーミッションを持つ端末にはそのファイルの変更が反映される．同期パーミッションを持たない端末はそのファイルへの更新を受け取ることができない．メタデータ編集パーミッションを持つ端末は同期メタデータのうちファイル ID を除くメタデータを，メタデータ編集 API を用いて編集できる．メタデータ編集パーミッションを持たないクライアント

はメタデータ編集 API を用いて同期モードのみ編集できる．端末が FSyncFS を最初にマウントするときに，ユーザ ID を用いて認証を行う．

3.4 メタデータ編集 API

FSyncFS は 3.3 で述べた同期設定情報をユーザプロセスから取得・変更するための API として以下のものを提供する．

int sync(int fd, int mode);

ファイルディスクリプタ fd が示すファイルの同期モードを mode に変更し，変更後の同期モードを返す．mode には ON/OFF/NOTIFY フラグを指定する．NOTIFY が指定された場合にはメタデータの変更は行わず，現在の同期モードを返す．

int server(int fd, char* addr, int operation);

ファイルディスクリプタ fd が示すファイルの，同期サーバ情報を operation に従って変更または通知する．operation には MODIFY/NOTIFY フラグを指定する．MODIFY が指定された場合にはファイルの同期サーバ情報を，addr で指定した同期サーバの IP アドレス（またはドメイン名）に変更する．IP アドレスには，“.” で区切られた整数を文字列として指定する．NOTIFY が指定された場合はメタデータの編集は行わず，現在の同期サーバ情報を addr が指すバッファに書き込む．

int sync_chmod(char *path, char *uid, int mode);

path で与えられるファイルの uid で与えられるユーザ ID に関するパーミッションを mode にしたがって変更する．uid には文字列で表されるユーザ ID もしくは “others” が指定される．mode は READ，WRITE，EXEC，SYNC，META の論理和で指定する．

3.5 ファイルアクセス情報

3.2 で述べたように，FSyncFS では端末とサーバの間でファイルアクセス情報の送受信を行い，各端末がローカルディスクへの反映を行うことでファイル同期を実現する．他端末での操作をローカルファイルに反映するためにはサーバから受信するファイルアクセス情報に端末上でのパス情報が必要であり，端末からサーバに送信するものとはその内容が異なる．また，ファイルアクセス情報を端末間で交換するのはファイルに変更が加わる操作のみであり，open()，read()，close() 等の実行時にはサーバとの通信は行わない．以下に具体的な例を示す．

create() 実行時

端末は，端末 ID$< dev_id >$，同期設定情報 $< metadata >$，作成したファイルのパス $< path >$ を，以下の形式で同期サーバに送信する．

“$C, < dev\_id >, < metadata >, < path >$”

端末からの問い合わせに対して，同期サーバは以下の形式の情報を送信する．

“$C, < metadata >, < path >$”

create が実行されたことを示すため，先頭に “C” を書き込む．他端末で作成されたファイルをどこに置くかはファイルアクセス情報を受け取った端末が設定ファイルを用いて決定する．ファイルを作成した端末と同じパスに置くことを可能にするために，サーバが送信するファイルアクセス情報にはファイルを作成した端末上でのパスが含まれる．新規ファイルを置くディレクトリのパスは端末ごとにユーザが決定し，設定ファイルに書き込む．端末が送信したファイルアクセス情報は同期サーバによって編集され，各端末のクライアントキューに追加される．

write() 実行時

端末は，端末 ID$< dev_id >$，ファイル ID$< file_id >$，write() の引数 $<$

$args >$ を，以下の形式で同期サーバに送信する．

“$W, < dev\_id >, < file\_id >, < args >$”

端末からの問い合わせに対して，同期サーバは以下の形式で端末上でのパス $< path >$ と write() の引数 $< args >$ を送信する．

“$W, < path >, < args >$”

write が実行されたことを示すため，先頭に “W” を書き込む．$< file\_id >$ は同期サーバによって $< path >$ に変換され，各端末のクライアントキューに追加される．また，$< args >$ には書き込み開始オフセット，バイト数，書き込まれたデータが含まれる．

rename() 実行時

端末は，端末 ID$< dev_id >$，ファイル ID$< file_id >$，リネーム前のパス $< from >$，リネーム後のパス $< to >$ を，以下の形式で同期サーバに送信する．

“$R, < dev\_id >, < file\_id >, < from >, < to >$”

rename が実行されたことを示すため，先頭に “R” を書き込む．ファイルのリネーム時にはファイルの名前だけでなくパスも変更される可能性があるため，リネーム前のパスとリネーム後のパスをサーバに送信する必要がある．ただし，各端末上のファイルのパスとファイル ID の変換は同期サーバが行うため，この操作に関するファイルアクセス情報は他端末に送信せず，管理情報テーブルの情報を書き換える．

link() 実行時

端末は，ファイル ID$< file_id >$ と新しいリンクのパス $< new\_path >$ を，以下の形式で同期サーバに送信する．

“$L, < file\_id >, < new\_path >$”

link() が実行されたことを示すため，先頭に “L” を書き込む．同期サーバは，同期対象ファイルの各端末上でのパスのリストを管理情報テーブルにもつ．create() 実行時に最初のパスが管理情報テーブルに追加され，link() 実行時には新しいパスが追加される．同期サーバから端末に送信されるファイルアクセス情報には受信端末上でのパスが含まれるが，これはその端末上での最も新しいパスである．

unlink() 実行時

端末は，端末 ID$< dev_id >$，ファイル ID$< file_id >$，削除されたパス $< pathname >$ を，以下の形式で同期サーバに送信する．

“$U, < dev\_id >, < file\_id >, < pathname >$”

unlink() が実行されたことを示すため，先頭に “U” が書き込まれる．unlink() が実行されると，同期サーバの管理情報テーブルから対象のパスの情報が削除される．すべてのパスが削除されると，それ以降そのファイルへの他端末からの操作は送信されなくなる．

3.6 ファイル同期の手順

複数端末間でファイルの同期を行う際，アクセス競合が発生する場合がある．サーバを用いたファイル同期の場合，複数の端末がほぼ同時に同じファイルに書き込みを行ったとき，それぞれの変更が正しく行われない場合がありえる．たとえば，サーバ S を介して同期しているファイル F に，まず端末 A が書き込み，その後端末 B が書き込みを行うことを考える．ただし，端末 A が書き込むときには端末 A 上の F と端末 B 上の F のデータが一致しているとする．A が F を編集したとき，B がその編集をローカルに反映後 B が F に書き込みを行う場合，競合は起こらない (図 3.4)．

しかし，A による書き込みと B による反映の間に B による書き込みが行われた

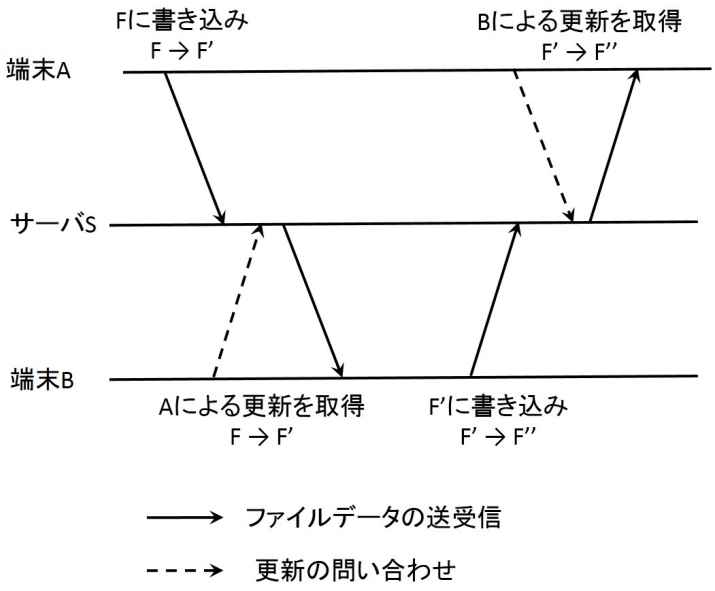

図 3.4: 通常の同期処理

場合，アクセス競合が発生し，A，B 双方の持つ F のデータに矛盾が生じてしまう (図 3.5).

アクセス競合を解決するための方法として，FSyncFS ではファイルのバージョンをフォークさせる方法を採用する．この方法では，同じファイルに対する同時アクセスを検出し，競合のあるファイルとして複数のバージョンを作成する．この方法では各端末のファイルアクセスが制限されず，ファイルにアクセスしたすべてのユーザのファイル操作が失われない．同期サーバの競合検出機構が同時アクセスを検出する機能を持つ.

このために，同期サーバは各ファイルのコピーを持つ．また，管理情報テーブルでは，各ファイルの最新版をそれぞれの端末がもっているかどうかを dirty_bit として記録する．以下ではこの情報を F_d_X とする．F_d_X が 1 の場合，端末 X はファイル F への最後の更新を受け取っていることを表し，F_d_X が 0 の場合は端末 X はファイル F への最後の更新を受け取っていないことを表す.

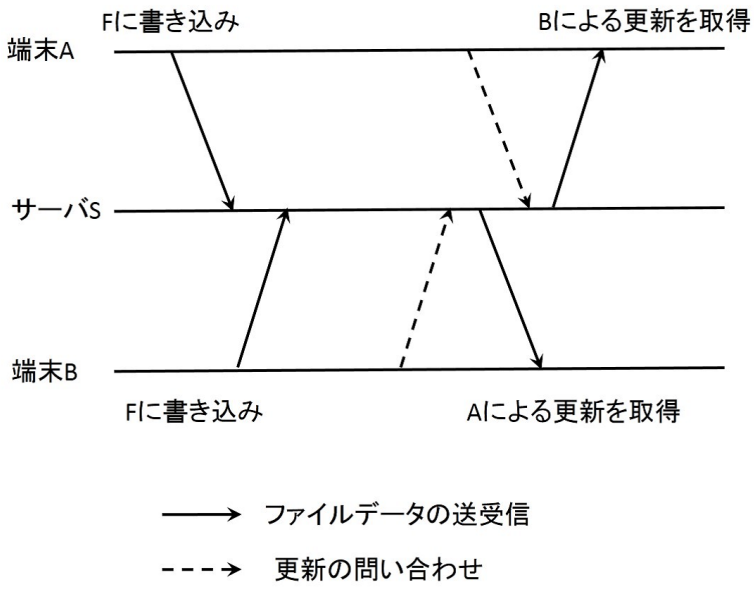

図 3.5: ファイルデータに矛盾が発生する場合

以下では，ファイルへの同時アクセスが発生しない場合と発生する場合のそれぞれについて，端末 A，B と同期サーバ，ファイル F を用いて FSyncFS の動作手順を述べる．ただし，まず A が書き込み後，B が書き込み，A による書き込みの実行時点では，A と B がそれぞれ持つ F のデータが一致していることとする．そのため，この時点では F_d_A と F_d_B は共に 1 となっている．

3.6.1 同時アクセスが起こらない場合

同時アクセスが起こらない場合の処理の流れを図 3.6 に示す．A 上で F が編集されたとき，編集済みのファイルを F' とする．A から F へのファイルアクセス情報 $\delta(F \to F')$ を受け取ると，同期サーバはF_d_A を確認する．同期サーバは F_d_A が 1 であるため同時アクセスが起こっていないと判断し，A に受信が終了したことを通知し，同期サーバ上の F を F' に更新する．このとき，同期サーバは F_d_B を 0 に書き換える．その後 B は同期サーバに問い合わせ，同期サーバから B に A によ

るファイルアクセス情報 $\delta(F \rightarrow F')$ が送信される.$\delta(F \rightarrow F')$ を受け取った B は, B 上の F に A による操作を反映し，端末間でのファイル同期が完了する．また，$\delta(F \rightarrow F')$ を B に送信した同期サーバは,F_d_B に 1 をセットする．

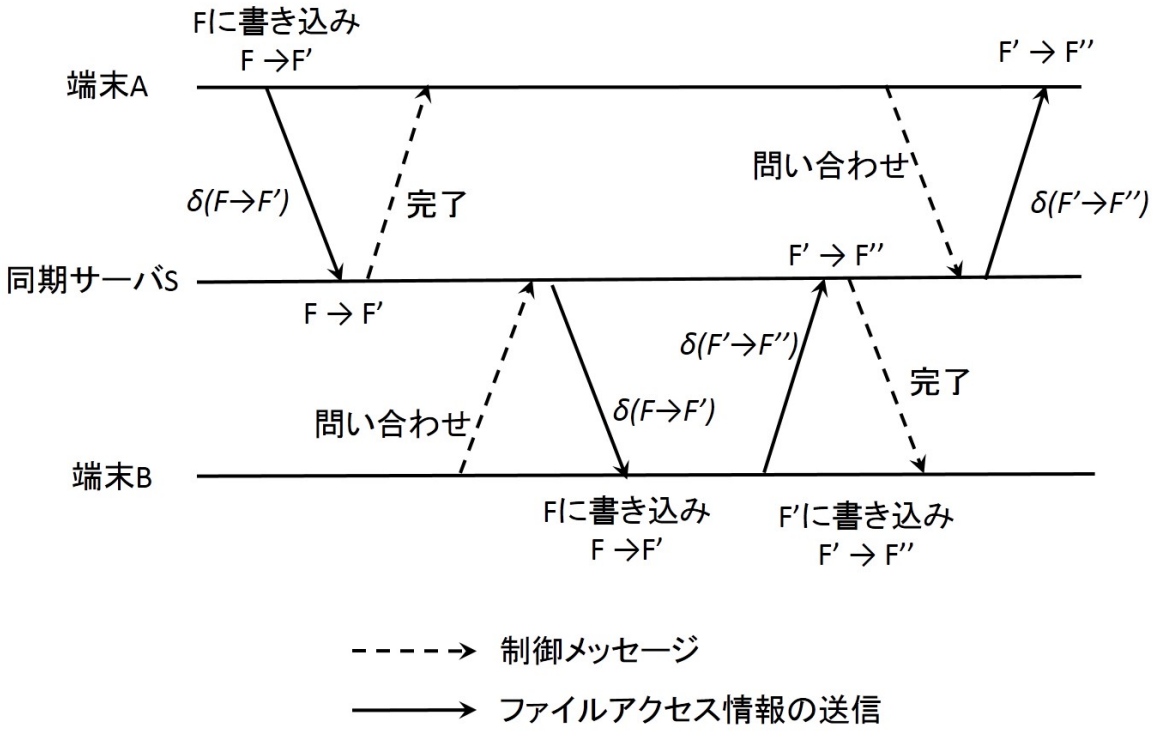

図 3.6: FSyncFS における通常の同期処理

3.6.2 同時アクセスが起こる場合

同時アクセスが発生する場合の処理の流れを図 3.7 に示す．A による F の編集は正常に受理され，3.6.1 の場合と同様に，同期サーバは受信完了の通知を A に送る．また，同期サーバ上の F は F' に更新される．B が A によるファイルアクセス情報を受け取る前に B 上の F を編集し，端末上で F'' に更新する状況を考える．B から $\delta(F \rightarrow F'')$ を受け取った同期サーバは,F_d_B を確認する．この場合，F_d_B は 0 であるため，同期サーバは同時アクセスを検出し，B には受信完了の通知ではなく，

競合メッセージとメタデータを含めた F' のすべてのデータを送信する．

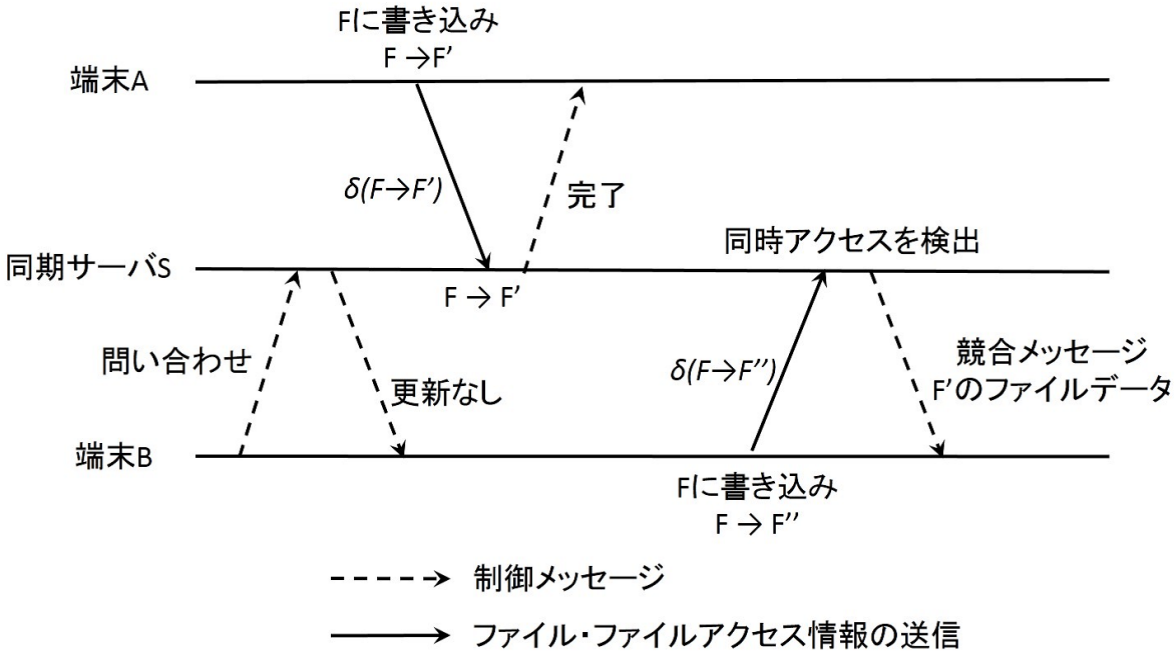

図 3.7: ファイルへの同時アクセスが発生した際の同期処理

競合メッセージには，F に対する同時アクセスが起こったことの通知と F'' のための新しいファイル ID が含まれる．F'' の ID は，F のファイル ID と同期サーバ上で管理されるユニークな番号をつなげたものとする．競合メッセージを受け取った B は，F" の同期モードを OFF にし，ファイル ID を書き換える．B 上で F'' がクローズされたときに，ファイル名を末尾に "_conflict" を付けたものに変更する．また，同期サーバから受け取った F' のデータに基づき，B 上に F' を作成する．

以上の操作によって，端末 A，B によるファイルの編集情報が失われないアクセス競合への対策を行う．これらの処理の結果，A は F' を，B は F' と F'' をそれぞれ持つ．

第 4 章

提案システムの実装

FSyncFS の実装方式について述べる．FSyncFS は Linux 上のローダブルカーネルモジュールとして実装する．これはスタッカブルファイルシステムとして動作するため，任意のローカルファイルシステムに同期機能を追加できる．今回の実装では，ローカルファイルシステムとして ext4 を用いた．

ファイルの実体と，同期設定情報を持つメタデータファイルはローカルファイルシステム上の別のファイルとして保存される．端末ではユーザプロセスが実行する各種ファイル操作に対応した VFS 層の処理をフックし，操作の種類と引数を抽出する．この情報と，メタデータファイルから取得した同期設定情報をファイルアクセス情報としてアクセスキューに追加する．

通信モジュールはカーネルスレッドとして非同期に動作する．これによってローカルファイルアクセスと同期サーバとの通信が並行に動作し，前者の遅延が少ない．通信モジュールは同期サーバと通信し，ファイルアクセス情報を送受信する．端末とクライアントの間では，TCP/IP を用いたソケット通信を行う．同期サーバ上には各端末に対応するキューがあり，端末からのファイル操作は同期先となる他端末に対応するキューに追加される．それぞれの端末は自端末に対応するキューからファイルアクセス情報を取得し，通信モジュールがローカルファイルに操作を反映する．

以下，4.1 で FSyncFS の端末におけるファイルの表現方法について述べ，4.2 で端末が行う処理の流れについて述べる．4.3 では，同期サーバの実装について述べる．

4.1 ファイルの表現

ここで，Linux におけるファイルの扱いと提案システムである FSyncFS でのファイルの表現について述べる．

4.1.1 Linux のファイルシステム

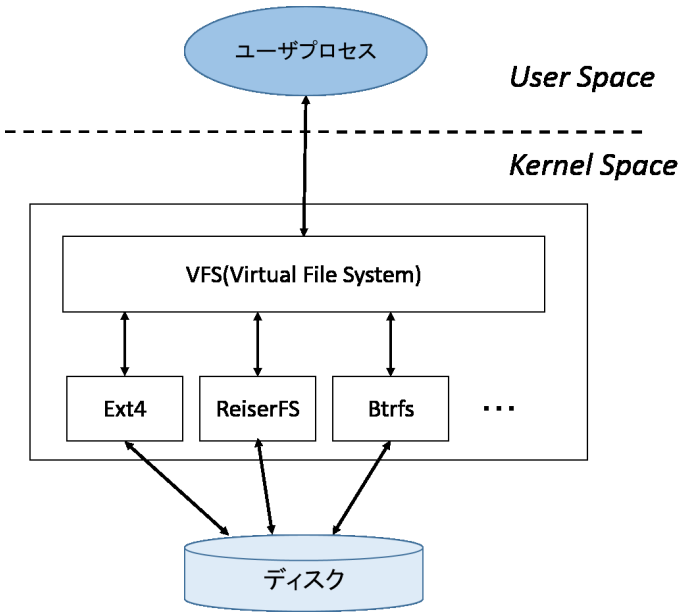

図 4.1: Linux の VFS

ファイルシステムとは，ディスクなどに記録されたデータをファイルという形式でユーザ空間に提供し，ディレクトリによる階層的な管理を可能にするシステムである．Linux では多様なファイルシステムを利用することができ，これらはそれぞれ異なる特徴を持っている．ここではデータの記憶領域としてディスクを用いる一般的なファイルシステムを利用する場合について説明する．

Linux では，VFS(Virtual File System) によって実装の異なるファイルシステムの差異を吸収し，ユーザ空間に統一的なインタフェースを提供する．VFS の概要を図 4.1 に示す．これによってユーザプロセスは動作しているファイルシステムの

種類を意識せず，抽象的にファイルにアクセスできる．VFS 上でのファイルは file 構造体や inode 構造体などの複数のオブジェクトによって構成される．これらのオブジェクトにはローカルファイルシステム固有のファイル操作関数が登録されている．ユーザプロセスからのファイルアクセス要求があると，VFS はオブジェクトに登録された対応するファイル操作関数を実行することで各ファイルシステムに処理を渡している．VFS を構成するオブジェクトの関係を図 4.2 に示す．

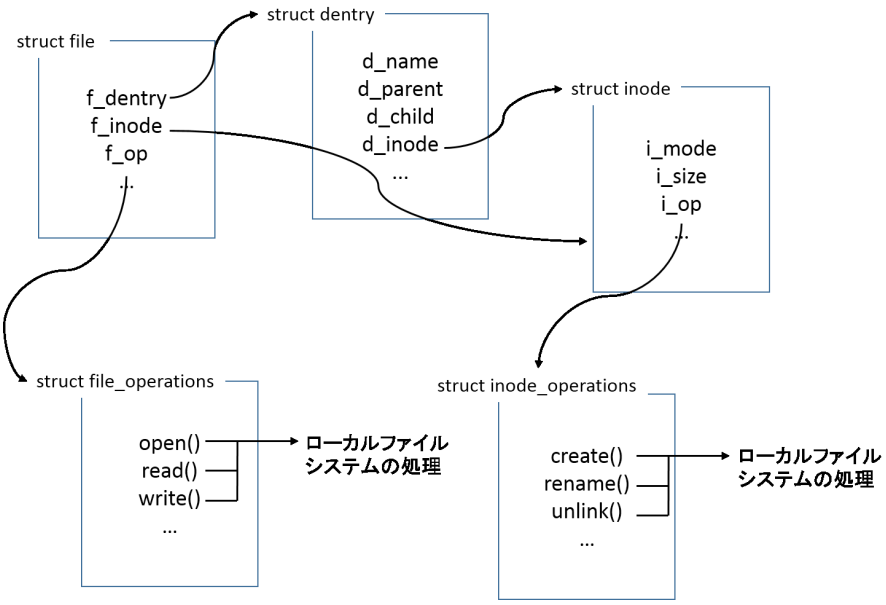

図 4.2: VFS を構成するオブジェクト

file 構造体は open されたファイルを管理する構造体である．ユーザプロセスが open() や read()，write() などのシステムコールを実行すると，ファイルディスクリプタの操作を介して file 構造体が持つ f_op に登録された関数が呼ばれる．ここには主にオープン済みのファイルに対して実行されるファイル操作関数が登録されている．

inode 構造体は，ファイルの長さやパーミッション情報などのメタデータを持つ．create() や link() など，ファイルの実体に対して実行するファイル操作関数が i_op

に登録される．

dentry 構造体はファイルやディレクトリの名前を管理し，パス名によるルックアップのためのキャッシュとして利用される．ファイル・ディレクトリ名や親ディレクトリへのポインタ，ディレクトリの場合にはそこに含まれる子オブジェクトへのポインタを管理する．

4.1.2 FSyncFS でのファイルの表現

FSyncFS はスタッカブルファイルシステムとして，図 4.3 のように動作する．

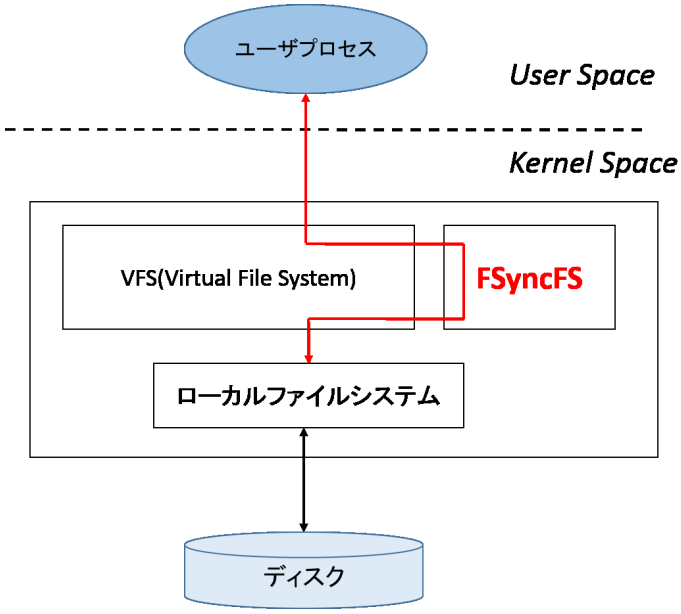

図 4.3: FSyncFS の動作図

VFS は FSyncFS をローカルファイルシステムとして扱い，FSyncFS は下層で動作するローカルファイルシステムに対して VFS のようにふるまう．ユーザプロセスがシステムコール呼び出しによってファイルアクセスを要求すると，VFS は FSyncFS のファイル操作関数を呼び出す．FSyncFS はその後さらに VFS 層の関数を呼び出すことでローカルファイルシステムに処理を渡す．その前後に FSyncFS

は同期サーバとの通信やメタデータ操作などの端末間でのファイル同期に必要な独自の処理を行う．

FSyncFS 上のファイルは file や inode などの VFS と同様のオブジェクトによって構成される．これらのオブジェクトはそれぞれ，ファイルシステム固有のデータを保持する領域にローカルファイルシステムを構成するオブジェクトへのポインタを持つ．図 4.4 に FSyncFS におけるファイルの構成から，file 構造体に関連する部分を抜粋して示す．

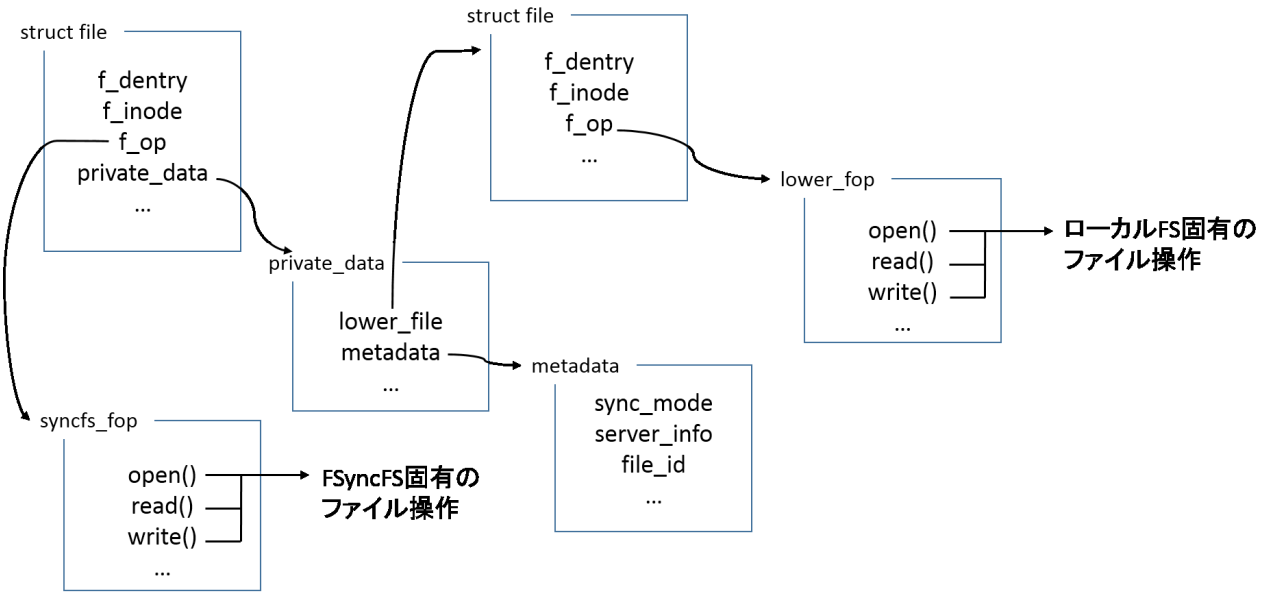

図 4.4: FSyncFS を構成するオブジェクト

file 構造体の場合，private_data にローカルファイルシステム上の file 構造体へのポインタを保持する (ソースコード 4.1，ソースコード 4.2)．

```
1 struct file {
2   struct path   f_path;
3 #define f_dentry  f_path.dentry
4   struct inode    *f_inode; /* cached value */
5   const struct file_operations  *f_op;
```

```
6
7     spinlock_t    f_lock;
8     atomic_long_t   f_count;
9     unsigned int    f_flags;
10    fmode_t     f_mode;
11    loff_t      f_pos;
12    struct fown_struct  f_owner;
13    const struct cred *f_cred;
14    struct file_ra_state  f_ra;
15
16    void      *private_data;
17            ⋮
18  };
```

ソースコード 4.1: struct file(抜粋)

```
1  struct fsyncfs_file_info {
2    struct sync_metadata *metadata;
3    struct file *lower_file;
4    const struct vm_operations_struct *lower_vm_ops;
5  };
```

ソースコード 4.2: private_data が指す構造体

また，FSyncFS が用いる同期設定情報もメタデータとして private_data に格納される (ソースコード 4.3).

```
1  struct sync_metadata{
2    int sync_mode;
3    char server_ip[16];
4    char file_id[256];
5  };
```

ソースコード 4.3: 同期設定情報のメタデータ

FSyncFS 固有のファイル操作 (ソースコード 4.4) では，同期設定情報を用いたファイル同期処理と VFS 層のファイル操作関数の呼び出しを行う．このとき，FSyncFS は引数としてローカルファイルシステムの file 構造体へのポインタを渡し，VFS によってローカルファイルシステム固有のファイル操作関数が呼び出される．

```
1 const struct file_operations fsyncfs_main_fops = {
2   .read    = fsyncfsfs_read,
3   .write     = fsyncfs_write,
4   .open    = fsyncfs_open,
5   .flush     = fsyncfs_flush,
6   .release   = fsyncfs_file_release,
7             ⋮
8 };
```

ソースコード 4.4: FSyncFS 固有のファイル操作関数群 (一部抜粋)

inode 構造体や dentry 構造体についても同様にしてデータの保持や処理関数の呼び出しが行われる．

4.2 ファイル操作の処理手順

本研究では FSyncFS のプロトタイプとして，create()，open()，read()，write() の処理を実装した．また，現在はテキストファイルに対応している．ここでは，これらの処理の処理手順についてそれぞれ述べる．

create()

ユーザプロセスが create() システムコールを呼び出すと，カーネル内では O_CREAT フラグつきの sys_open() 関数が実行される．その後 VFS 層で vfs_create() 関数が呼び出され，この内部で inode 構造体の create 関数が呼ばれることで FSyncFS に処理が移る．FSyncFS はこのとき，まずメタデータファイルをローカルファイルシステム上に作成する．ファイルの作成には

vfs_create() 関数を用いる．引数としてローカルファイルシステム上の inode 構造体や dentry 構造体へのポインタを渡す．作成されたメタデータファイルには設定ファイルに従って同期設定情報が書き込まれる．ファイル ID$< f_id >$は，ユーザ ID$< user_id >$，端末番号$< dev_num >$，ファイル番号$< file_num >$を用いて，以下の形式で生成する．

“$< user\_id >$-$< dev\_num >$-$< file\_num >$”

このときファイル番号として，端末内でユニークな番号を与える．その後，同期設定情報のファイル ID を用いてデータファイルの名前を生成する．現在の実装では，データファイルの名前は”.$< file\_id >$”の形式である．生成した名前のファイルを，データファイルとしてメタデータファイルと同様の方法で作成する．メタデータファイルとデータファイルの作成が終了すると，同期設定情報とファイルのパスを含んだファイルアクセス情報をアクセスキューに追加して create 処理を終了する．

open()

ユーザプロセスから open() が実行されると，カーネル内では sys_open() 関数が実行される．その後 VFS 層で vfs_open() 関数が呼び出され，この内部で file 構造体の open() 関数が実行されることで処理が FSyncFS に移る．FSyncFS はまず，file 構造体の private_data にメタデータを格納するための領域を確保する．次に dentry_open() 関数によってメタデータファイルをオープンし，書き込まれている同期設定情報をメタデータ領域に格納する．その後，ファイル ID を元にデータファイルの名前を特定してオープンする．オープンされたデータファイルは private_data 内に格納される．

read()/write()

オープンされたファイルに対して read や write が実行されると，FSyncFS は private_data に格納されたデータファイルに対してこれらの処理を行う．ま

た，writeの場合はオフセット，カウント，書き込まれたデータ，ファイルID，ユーザIDをアクセスキューに追加する．

4.3 同期サーバの実装

同期サーバのプロトタイプをLinuxで動作するアプリケーションとして実装した．プロトタイプでは通信モジュール，ID–Path変換機構，管理情報テーブル，クライアントキューが動作する．ただし，管理情報テーブルはID–Path変換機構が利用するデータのみ保持する．

通信モジュールは端末からメッセージを受け取ると，その内容からメッセージの種類を判別する．現在同期サーバが受け付けられるメッセージの種類は，ファイルの作成とファイルへの書き込みのファイルアクセス情報および各端末からの新規ファイルアクセス情報の問い合わせである．

メッセージの種類がファイルアクセス情報だった場合，さらにファイル操作の種類を判別し，ID–Path変換機構が送られてきたファイルアクセス情報の編集とクライアントキューへの追加を行う．以下では，ファイルアクセスを行った端末を実行端末，それ以外の端末を受信端末と呼ぶ．

メッセージの種類がファイルの作成だった場合，メッセージに含まれるファイルIDとファイルのパスを抽出し，管理情報テーブルに書き込む．なお，管理情報テーブルはsqliteを用いたリレーショナルデータベースとして実装した．そのため，sqliteのSQL文を呼び出すことで管理情報テーブルへの情報の追加が実行される．管理情報テーブルは各端末におけるファイルのパスを持つ必要がある．プロトタイプでは，この時点では受信端末上でのパスは“$/new\_file/ <$ ファイル名 $>$”に固定している．管理情報テーブルへの情報の追加後，ID–Path変換機構がファイルアクセス情報の内容を3.3での設計にしたがって編集し，受信端末に対応するクライアントキューに追加する．

メッセージの種類がファイルへの書き込みだった場合，メッセージからファイル ID と実行端末の端末 ID を抽出する．その後，ID–Path 変換機構が管理情報テーブルを参照してファイルアクセス情報の内容を編集し，受信端末に対応するクライアントキューに追加する．

メッセージの種類が新規ファイルアクセス情報の問い合わせだった場合，通信モジュールが受信端末に対応するキューからファイルアクセス情報を取り出し，順次送信する．

第 5 章

評価と考察

5.1 性能評価

FSyncFS の FS モジュールについて性能評価を行った．ext4 ファイルシステムと wrapfs[17] を比較対象とし，ファイルファイルアクセスの性能を評価した．ただし，wrapfs はユーザ空間からの処理をローカルファイルシステムに伝える処理のみを行う単純なスタッカブルファイルシステムである．FSyncFS および wrapfs の下層ではローカルファイルシステムとして ext4 が動作している．write() システムコールを用いてファイルへの書き込みを繰り返し，その速度を計測した．計測開始時に空のファイルを作成し，write() システムコールを用いて “hello” という文字列をファイルに書き込む．また，2 回目以降の書き込みはファイルの末尾に同じ文字列を追記する．

図 5.1 にその結果を示す．

図 5.1 で縦軸は処理時間 (単位：秒)，横軸は write() の実行回数である．測定の結果，FSyncFS の FS モジュールは ext4 と比較して 10 倍程度のオーバヘッドで動作することがわかった．wrapfs も ext4 と比較するとオーバヘッドが発生しているが，FSyncFS より高速である．そのため，FSyncFS のオーバヘッドは実装方式に依存するものではなく，今後の性能向上が可能であると考えられる．

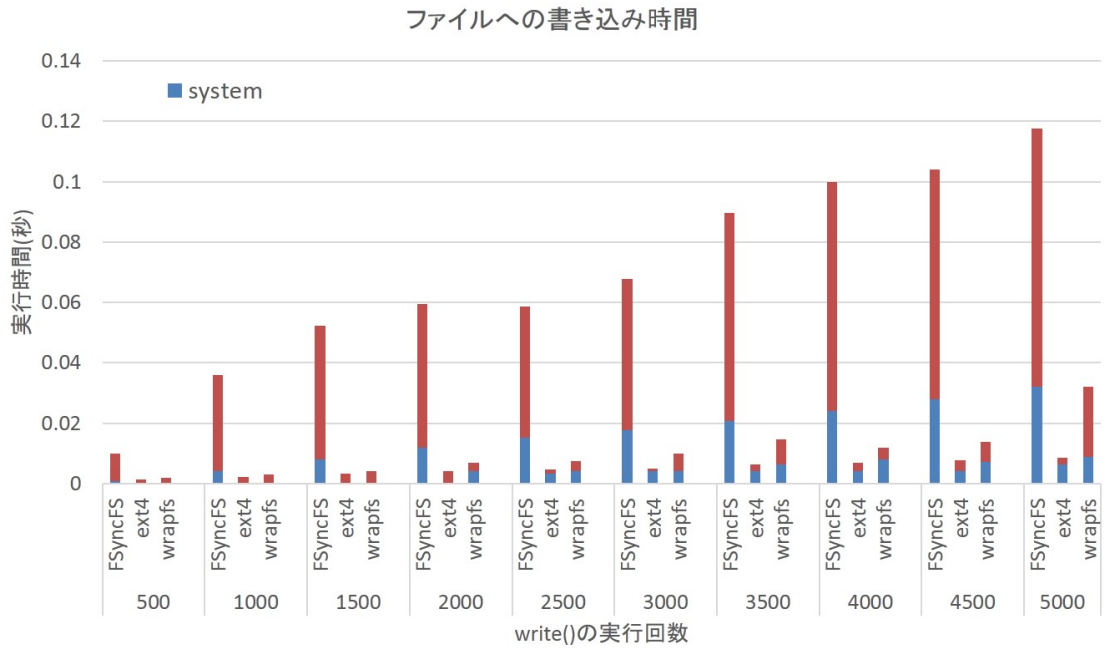

図 5.1: FS の性能の計測

5.2 既存システムとの比較

提案システムである FSyncFS とネットワークファイルシステムおよび同期アプリケーションの特徴を比較する．ネットワークファイルシステムにおけるファイルアクセスにはサーバとの通信を伴うため，特に低速なネットワーク環境下ではローカルファイルシステムと比較した場合にファイルアクセス速度が低下する．対して既存の同期アプリケーションではファイルの実体が端末上に存在することが保証できるため，アクセス速度の低下が発生しない．FSyncFS ではファイルアクセス処理に同期処理を追加しているため，多少の性能低下が発生するが，サーバとの通信は非同期に行われるため，ローカルファイルシステムに近い性能で動作する．また，FSyncFS は各ファイルが同期設定情報を持つため，同期の ON/OFF や利用する同期サーバの情報をファイルごとに設定できる．一方，ネットワークファイルシステムではディレクトリ単位で共有を行うため，ファイル単位での共有設定がで

きない．同期アプリケーションにはファイル単位で同期設定が可能なもの [4][5] もあるが，同期設定を維持したファイルの移動ができない．これらのツールでは設定ファイルに同期対象ファイルのパスや同期先の情報を書き込むことで同期に必要な情報を設定している．ファイル同期ツールは設定ファイルに従って同期を行うが，同期対象ファイルを移動すると設定ファイルに書かれたファイルのパスと実際のパスが異なるものになるため同期処理が実行できない．

ファイル同期と同様にファイルやデータの複製を複数の端末やサーバに持たせるものとして，バックアップやレプリケーションがある．データのバックアップは，複製されたデータの完全性を確保するために行われる．利用中のデータが失われた場合の復旧のために，ある時点でのデータの複製をバックアップとして保存する．たとえば，データに対して不正な操作が行われたとき，保存されたバックアップを用いることで障害発生前の状態に復元することができる．データのバックアップは，ユーザが端末上で利用するデータやサーバ上に保存されたデータに対して広く行われている．レプリケーションでは，データの可用性を向上させるために，計算機上のデータへの操作を複製されたデータに対してリアルタイムに反映させる．たとえば，災害の発生などによってサーバが利用できなくなった場合，レプリケーションによって作成されたサーバの複製を利用することで，システムを継続して動作させることができる．バックアップやレプリケーションは耐障害性の向上を目的としているため，データの複製をユーザが普段利用しない端末やサーバに置く必要がある．これらの技術は端末間でのファイル共有を目的としたものではないため，本研究では対象としていない．

5.3 FSyncFS における同期のセマンティクス

FSyncFS では，アクセス競合への対策として，複数端末からの同時アクセスを検出したときにファイルのバージョンをフォークさせる．ここではバージョンの

フォークが発生する状況について考察する．

FSyncFS では，ユーザプロセスによるシステムコールの呼び出しを単位として，ファイルアクセス情報が同期サーバに送信される．1 つの端末上で実行されたファイル操作の順序と同期サーバが受け取るファイルアクセス情報の順序は一致する．ファイル同期を一方向で行う場合，同期元の端末と同期先の端末が共にオンラインであればそれぞれが持つファイルのデータは一致する．また，この場合いずれかの端末が一時的にオフラインであったとしても，全体としてオンラインの時間が十分にあればデータの矛盾が生じることなく同期されていく．

双方向で同期を行う場合，3.6 で述べたようにアクセス競合が発生する場合がある．端末 A と端末 B が同期サーバ S を介してファイル F の同期を行う場合を考える．ここで，A と B がそれぞれ 1 回ずつ F に書き込みを行うこととし，端末上での書き込みや S への問い合わせをイベントとして表 5.3 に示す．また，各イベントの発生時刻と処理時間を表 5.3 のように定義する．

表 5.1: 発生するイベント

a	A による F への書き込み
b	B による F への書き込み
req_A_i	A による S への問い合わせ
req_B_i	B による S への問い合わせ

各端末からの S への問い合わせは定期的に行われ，たとえば req_A_i の直前の A による問い合わせは req_A_i-1 と表す．ここでは $t(req\_A\_i-1) < t(a) < t(req\_A\_i)$ と $t(req\_B\_i-1) < t(b) < t(req\_B\_i)$ が成り立つこととする．

以降，A による書き込み後，B による書き込みが行われることを考える．このときバージョンのフォークが起こらないためには，B は A による書き込みの情報を受

表 5.2: 発生するイベントとその時刻

$t(a)$	A がファイル F に書き込む時刻
$t(b)$	B がファイル F に書き込む時刻
$d(A-S)$	A からのメッセージが S に到達するまでの時間
$d(B-S)$	B からのメッセージが S に到達するまでの時間
$t(req\_A\_i)$	A が S に問い合わせる時刻
$t(req\_B\_i)$	B が S に問い合わせる時刻
s_time	メッセージを受け取った S が処理を終了するまでの時間

け取ってから書き込みを行う必要がある．$t(a)$ と $t(b)$ が式 5.1 を満たすとき，バージョンのフォークは起こらない．

$$t(a) + d(A-S) < t(req\_B\_i - 1) + (2 \times d(B-S) + s\_time) < t(b) \qquad (5.1)$$

すべての端末が常にオンラインであっても，バージョンのフォークを完全に防ぐためには，各端末がファイルに排他的にアクセスする必要がある．しかし，FSyncFS では各端末がローカルディスク上のファイルに制約なくアクセスできることを優先したため，ファイルのロックによる排他制御は採用しなかった．また，アクセス競合が発生した際に，いずれかの端末の操作のみを受け付ける方法でも，バージョンのフォークを防ぐことができる．しかし，この方法では同時アクセスを行った端末のうち，1 つの端末以外が行った操作が失われる．FSyncFS ではユーザによるファイル操作情報の保護を優先し，この方法は採用しなかった．

5.4 オフライン操作

ファイルをオフライン環境で利用する場合，端末上で実行されたファイルやディレクトリへの変更をローカルディスク上に記録しておく必要がある．FSyncFS ではこれらの情報をファイルアクセス情報として同期サーバに送信している．FSyncFS がネットワークからの切断を検出し，ファイルアクセス情報をアクセスログファイルとしてローカルディスクに記録することでオフライン操作が実現できると考えられる．同様の方法が Intermezzo で用いられている．2 章で述べたように，Intermezzo は KML にファイルアクセスのログが記録され，InterSync によって KML のサーバとの同期が行われる．ネットワークとの接続の有無に関わらずアクセスログをファイルに記録する Intermezzo と異なり，FSyncFS では FS モジュールと通信モジュールがメモリ上のアクセスキューを介してファイルアクセス情報のやり取りを行う．そのため，オフライン操作を実現するためにはネットワークとの接続が失敗したことを検出してファイルアクセス情報を記録しておく仕組みが必要になる．FSyncFS にこの機能を実装する場合，通信モジュールがネットワークの状況を判断してサーバへの送信とログファイルへの記録を適切に選択するということが考えられる．

5.5 ファイルアクセス情報の最適化

FSyncFS では，ファイルアクセス単位でファイルの同期を行う．システムコール呼び出しによるファイル操作が行われるたびにファイルアクセス情報のサーバへの送信が発生し，他端末も同数のファルアクセス情報を受信する必要がある．そこで，ファイルアクセス情報の最適化による通信の効率化を行うことが考えられる．具体的には，ファイルへの連続した書き込みを一つにまとめることで，ファイルアクセス情報の数を減らし，サーバとの通信量を低減させることができる．

ファイルへの書き込みの際にサーバに送信する情報には端末 ID$< dev_id >$，ファ

イル ID$< file_id >$，書き込み位置$< offset >$，書き込まれる文字列$< data >$とその長さ$< length >$が含まれる．FSyncFS で端末 A がファイル F の先頭に長さ$< len1 >$の文字列 “$< data1 >$” を書き込んだとき，ファイルアクセス情報は以下のようになる．

“$W, < dev\_id >, < file\_id >, 0, < len1 >, < data1 >$”

その後端末 A が続けて長さ$< len2 >$の文字列 “$< data2 >$” を書き込んだとき，ファイルアクセス情報は以下のようになる．

“$W, < dev\_id >, < file\_id >, < len1 >, < len2 >, < data2 >$”

これらのファイルアクセス情報はそれぞれ別のファイルアクセスとしてサーバに送信され，他端末に伝えられる．2 つのファイルアクセス情報は，以下のようにまとめられる．

“$W, < dev\_id >, < file\_id >, 0, < len1 + len2 >, < data1 >< data2 >$”

FSyncFS ではファイル操作単位でサーバとの通信を行うが，この方法によって各端末がファイルアクセス情報の最適化を行うためには，サーバへのファイルアクセス情報の送信の前に後続のファイル操作を待つ必要がある．しかし，モバイル環境で FSyncFS を利用する場合，端末がオフラインになることを考慮する必要がある．上記の方法では “$< data1 >$” の書き込み後，“$< data2 >$” の書き込み前に端末 A がオフラインになった場合，“$< data2 >$” だけではなく “$< data1 >$” の書き込みも他端末に反映することができない．そこで，上記のファイルアクセス情報の最適化を同期サーバが行うことを考える．この場合，ファイル操作を実行した端末が同期サーバに送信する情報量は削減することができないが，他の端末が受け取るデータ量と各端末がローカルのファイルを更新するための操作数を低減させることができる．

第 6 章

おわりに

端末間でのファイル同期機能を持つファイルシステム FSyncFS を提案した．FSyncFS はカーネル空間で動作するため，アプリケーションが利用できるファイルが制限されたサンドボックス環境においても利用できる．各ファイルに同期設定情報をメタデータとして持たせることにより，ファイルのパスに依存しないファイル同期を行う．これによって，同期設定を維持したままファイルを移動すること，同期対象ファイルを端末ごとに異なるパスへ配置すること，ファイルごとに同期に関する設定を行うことが可能である．FSyncFS は同期サーバを用いてファイル同期を行う．各端末は，ユーザプロセスがファイル操作のためのシステムコールを実行したことをトリガにして同期サーバへのファイルアクセス情報の送信を行う．また，自動的に他端末からのファイルアクセス情報を取得し，自端末上のファイルの更新を行う．

本研究では FSyncFS をローダブルカーネルモジュールとして実装した．これは Linux カーネル内でスタッカブルファイルシステムとして動作する．これによってユーザは下層で動作するローカルファイルシステムを自由に選択することができ，FSyncFS を用いて任意のファイルシステムにファイル同期機能を拡張することが可能である．また，FSyncFS はファイル操作を表すファイルアクセス情報を同期サーバに送るため，既存のファイル同期アプリケーションのようにファイル更新の差分を計算する必要がない．ローカルファイルアクセスの遅延を少なくするために，同期サーバとの通信は並行に動作させた．

本提案の FSyncFS により，サンドボックス環境として動作することが一般的となったスマートフォンにおいて，基盤的機能としてのファイル同期を，広くアプリケーションに提供できる．

謝辞

本研究を遂行するにあたっては，いろいろな方々にお世話になりました．

まず指導教員の鶴岡行雄先生には日頃から熱心なご指導，そしてご鞭撻を賜わりました。またご多忙中にもかかわらず論文の草稿を丁寧に読んで下さり，大変貴重なご助言をいただきました，ここに厚く御礼申し上げます。

また，多田好克先生，小宮常康先生には研究を進める上で貴重な意見や助言をいただきました，ここに感謝の意を表します，

そして本研究が行なえたことは，研究方針や方法論について議論をし，共に研究生活をおくってきた，鶴岡研究室をはじめとした基盤ソフトウェア学講座の学生諸氏のおかげでもあります。最後に，これらの皆さんに感謝いたします．

参考文献

[1] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C.Beame, M. Eisler and D. Noveck : “Network File System(NFS) Version 4 Protocol, Request for Comments 3530.”April 2003.

[2] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki,E. H. Siegel and D. C. Steere : “Coda: A highly available file system for a distributed workstation environment,” IEEE Trans. Comput. vol. 39, pp. 447-459, April 1990.

[3] Dropbox, https://www.dropbox.com/, 2013 年閲覧.

[4] P. J. Braam, M. Callahan, and P. Schwan : “The InterMezzo filesystem. ” In Proceedings of the O’Reilly Perl Conference 3, August 1999.

[5] A. Tridgell and P. Mackerras : “The rsync algorithm.” Technical Report TR-CS-96-05, Dept. of Computer Science, The Australian National University, Canberra, Australia, 1996.

[6] B. C. Pierce and J. Vouillon : “What ’s in Unison? A formal specification and reference implementation of a file synchronizer.” Technical Report MS-CIS-03-36, Dept.of Computer and Information Science, University of Pennsylvania, 2004.

[7] iCloud, http://www.apple.com/icloud/, 2013 年閲覧.

[8] Google Drive, http://www.google.com/drive/about.html, 2014 年閲覧.

[9] SkyDrive, http://windows.microsoft.com/ja-jp/skydrive/download/, 2014 年閲覧.

[10] M. Szeredi : “File system in user space.” http://fuse.sourceforge.net/, 2014 年閲覧.

[11] LUFS, http://sourceforge.net/projects/lufs/, 2014 年閲覧.

[12] WikipediaFS, http://wikipediafs.sourceforge.net/, 2013 年閲覧.

[13] GMail Filesystem over FUSE, http://sr71.net/projects/gmailfs/, 2014 年閲覧.

[14] Aufs, http://aufs.sourceforge.net/, 2013 年閲覧.

[15] Y. Dong, H. Zhu, J. Peng, F. Wang, M. P. Mesnier, D. Wang, and S. C. Chan : “RFS – A Network File System for Mobile Devices and the Cloud.” ACM SIGOPS Operating Systems Review Volume 45 Issue 1, January 2011 Pages 101-111.

[16] E. Zadoc, I. Badulescu and A. Shender : “Extending file systems using stackable templates.” Proceedings of USENIX Annual Technical Conference, 1999.

[17] Wrapfs: A Stackable Passthru File System, http://wrapfs.filesystems.org/, 2014 閲覧.