システムソフトウェアと
オペレーティング・システム 92−16
（２００３．２．２８）

リモート・デバイス利用に関する汎用的なフレームワークの設計と実装

佐藤 純次† 河合 栄治‡ 中村 豊†
藤川 和利† 砂原 秀樹†

†奈良先端科学技術大学院大学

‡科学技術振興事業団 さきがけ研究 21

概要: 様々なバス・インターフェイスに対応した多くの周辺機器が登場してきているが，それらの周辺機器は計算機に直接接続した利用しかなされていないのが現状である．周辺機器は通常デバイスドライバと呼ばれるカーネルの一機能にて制御され，カーネルレベルにてネットワークを介した周辺機器 (リモート・デバイス) の透過的利用を実現する技術は多数存在する．しかしいずれも，特定の周辺機器の利用や，専用の API を必要とするなど制限が存在する．そこで，カーネルレベルにてリモート・デバイスを利用する汎用的なフレームワークを提案する．本稿では，リモート・デバイス利用を実現するフレームワークの設計と実装について述べる．
キーワード: オペレーティングシステム，UNIX，デバイスドライバ，リモート・デバイス，インターネット

The Design and Implementation of a Generic Framework for Remote Device Control

Junji Sato† Eiji Kawai‡ Yutaka Nakamura†
Kazutoshi Fujikawa† Hideki Sunahara†

†Nara Institute of Science and Technology

‡PREST, Japan Science and Technology Corporation

Abstract: There are many peripheral devices attached locally to computers. These devices are controlled by device drivers in some operating systems and most of these devices do not provide functions for remote usage over the network. Several technologies allow us to transparently use remote devices in the kernel. However there is often a limitation on the type of devices we can use or the absence of the required APIs and libraries that prevents us from accessing remote devices. In this paper, we propose a generic framework for remote device control. This paper describes its design and implementation details.
Keywords: Operating System, UNIX, device driver, remote device, the Internet

1 はじめに

計算機に接続する周辺機器において，様々なバス・インタフェース (USB，IEEE1394，SerialATA など) が仕様化され，それらのバスに対応した多くの周辺機器が登場している．しかし，周辺機器の利用は計算機に直接接続された形態での利用がほとんとである．今日では xDSL に代表される広帯域のネットワークインフラが整備され，インターネットへ接続する計算機が増加の傾向にある．そこで，ネットワークを介して周辺機器を手元の計算機で利用する技術が必要となる．

一方で，インターネットに遍在する計算機に接続された周辺機器 (以下，リモート・デバイスと呼ぶ) を透過的に利用する技術は多数存在する．透過的な利用とは，仮想的に手元の計算機に接続されているかのように利用でき，ネットワーク的に離れた周辺機器を操作できることを意味する (図 1)．リモート・デバイスの透過的な利用を実現する技術は大別して 2 つの手法に分類できる．1 つはユーザレベルでの実現手法であり，もう 1 つはカーネルレベルでの実現手法である．

ユーザレベルの手法は，ユーザプログラムまたはライブラリなどを使用しリモート・デバイスの透過的な利用を実現する手法である．例えば，ESD[1] は計算

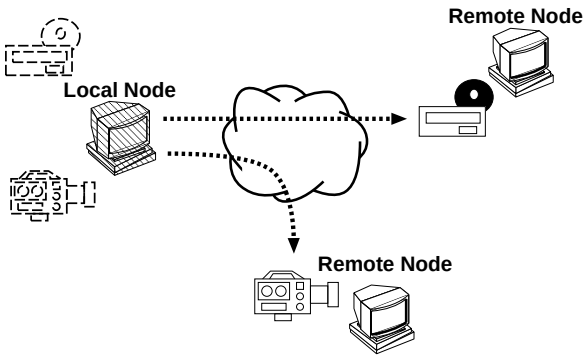

図 1: リモート・デバイスの透過的な利用

機内にデーモンとして常駐するサーバプログラムを介して，他の計算機からサウンドデバイスの利用を可能にする．また，SDL[2] や Microsoft DirectX®[3] などのライブラリは直接周辺機器を操作するための API を提供している．さらに，Jini[4] や UPnP[5] などはミドルウェアとして API のみならず，周辺機器の操作のための仕様も提供している．

これらはいずれもユーザ空間から周辺機器を操作する手法をとる．この手法はリモート・デバイスを専用のライブラリやミドルウェアで操作するため，様々なオペレーティングシステム (以下，OS と呼ぶ) へ移植しやすい特徴を持つ．しかし，専用のライブラリやミドルウェアを使うことは，既存のアプリケーションの利用が難しいことを意味する．ソースコードの改変やプログラムの再コンパイル/再リンクを必要とするため，既存環境との親和性が低く，開発コストも高い．

また，ユーザレベルの手法はデバイス操作に際しカーネル内のデバイスドライバ・インターフェイスを経由する．したがって，ユーザ空間で動作するプロセスが直接デバイスを操作するとはいえ，その処理はカーネル空間内で行われ，処理結果がユーザ空間にコピーされる．このように，リモート・デバイスの実現手法は，既存環境との親和性や開発コストの問題，また実際の処理がカーネル空間で行われていることから，ユーザレベルの手法よりもカーネルレベルの手法が有効である．

カーネルレベルでの実現手法はユーザレベルの実現手法で問題となる既存環境との親和性を解決する．カーネルレベルの手法はシステムコールから VFS (Virtual File System) 層 [6][7]，そして各デバイス操作関数へ至るデバイスドライバ・インターフェイスを考慮した設計である．カーネルレベルの手法により，アプリケーション開発者は従来のシステムコールやそのラッパールーチンを含む API を利用でき，既存のアプリケーションもそのまま利用できる．しかし，カーネルレベルの手法はカーネル内における実装レイヤが異なり，各々の実装レイヤに起因する制限がある．

そこで，本稿では既存環境と親和性を保ち，かつデバイスタイプに依存しない汎用的なフレームワークを提案する．伝送路には IP を用いることでこれまでのケーブル長や接続形態などの物理的制約がなくなり，リモート・デバイスの透過的な利用が可能になる．以下，本稿の構成を示す．2 章で既存技術を整理しその問題点について述べる．3 章で本フレームワークの設計や実装について述べる．4 章で様々なデータサイズのブロックを送受信し, フレームワークを介した際のスループットを定量的に評価した．最後に結論を述べる．

2 既存技術

本章では，カーネルレベルでリモート・デバイスの利用を実現している各技術について述べる．

2.1 NFS

NFS（Network File System）[8] は複数の計算機に分散したファイルシステムを共有する技術であり，ファイルシステム・レイヤでリモート・デバイス利用を実現する．NFS では，XDR（eXternal Data Representation）を用いてアーキテクチャに依存しないデータ表現を使用しており，RPC (Remote Procudure Call) の上に実装されている．NFS はリモートノードのファイルシステム (ストレージデバイス) の利用が可能である．

2.2 RFS

RFS (Remote File Sharing)[9] も NFS 同様ファイルシステム・レイヤでリモート・デバイス利用を実現する手法である．RFS は NFS と同じ機能を提供するが，対象をデバイスファイルや名前付きパイプなどすべてのファイルタイプに拡張している．ファイルへの操作は UNIX システムコール・インターフェイスに基づいたメッセージプロトコルを用い，その際 UNIX システムのセマンティクスの保護を考慮し，

表 1: 既存技術の問題点

手法	実装レイヤ	親和性	移植性	汎用性	開発コスト	具体例
ユーザレベル	プログラム	×	○	○	高	ESD
	ライブラリ	×	○	○	高	SDL，DirectX
	ミドルウェア	×	○	○	高	UPnP，Jini
カーネルレベル	ファイルシステム	○	○	×	低	NFS，RFS
	バス	○	○	×	低	iSCSI
	デバイスドライバ	○	○	○	高	スタブドライバ

ファイルシステムをエクスポートする計算機上でセマンティクスを構成する．RFS ではすべてのデバイスファイルに対するシステムコールが可能となっている．

2.3 バス・コマンドのカプセル化

バス・コマンドのカプセル化は，周辺機器における各バス・インターフェイスのトランザクションを IP データグラム内にカプセル化する手法である．この手法はこれまで専用のケーブルを使い，バス上で制御されていたコマンド及びデータをネットワーク上に延長したモデルである．具体例として iSCSI が存在する．

iSCSI[10] は SCSI トランザクションをインターネット上で送受信するプロトコルである．SCSI デバイスは SCSI 規格によりクラス化されており，様々なクラスが存在する．その中で iSCSI はダイレクトアクセスクラスのデバイス利用が可能である．

2.4 スタブドライバ

スタブドライバ [11] はデバイスドライバ・レイヤでのリモート・デバイス利用を行っている．スタブドライバは IP データグラム内にデバイス操作関数 (デバイスメソッド) 自体をカプセル化し，デバイスデーモンとの間で送受信する．そのため，利用するデバイス毎にスタブドライバとデバイスデーモンの組を必要とする．

スタブドライバはリモート・デバイスとユーザアプリケーションのインターフェイスとして機能し，デバイスドライバとして実装されている．デバイスデーモンはスタブドライバからのリクエストを受け付けるプロセスであり，カーネルスレッドとして実装されている．カーネルスレッドで実装されているのはカーネル空間内のすべての関数や変数に対しアクセスする必要があるためである．

2.5 既存技術の問題点

既存技術を以下の観点からまとめたものが表 1 である．

- 親和性：既存のアプリケーションを改変，再コンパイル/再リンクせずに利用可能である
- 移植性：他の OS への移植が容易である
- 汎用性：デバイスの種類に依存せず使用可能である
- 開発コスト：デバイスドライバやアプリケーションなど新規に作成する必要がある

カーネルレベルの手法は概して利用できるデバイスの種類に制限がある．

NFS はファイルシステムの共有を目的とし，デバイスの利用はハードディスクや CDROM などストレージデバイスに限られる．これらのデバイスはブロック型デバイスに分類され，NFS ではキャラクタ型デバイスを使用できない．

RFS は扱うファイルをデバイスファイルまで拡張しており，デバイスファイルに対するシステムコールをノード間で送受信するプロトコルとなっている．しかし，ファイルシステム・レイヤでの実装のため，パス名検索や inode のリンク，そして参照カウントなどファイルシステム維持の機能を要し，実装は複雑になる．さらに，カーネル空間とユーザ空間でデータの移動を行う汎用のカーネル関数に対し，ネットワーク的な位置を判別するコードを加える．この関数はデバイスドライバのみならず他のサブシステム

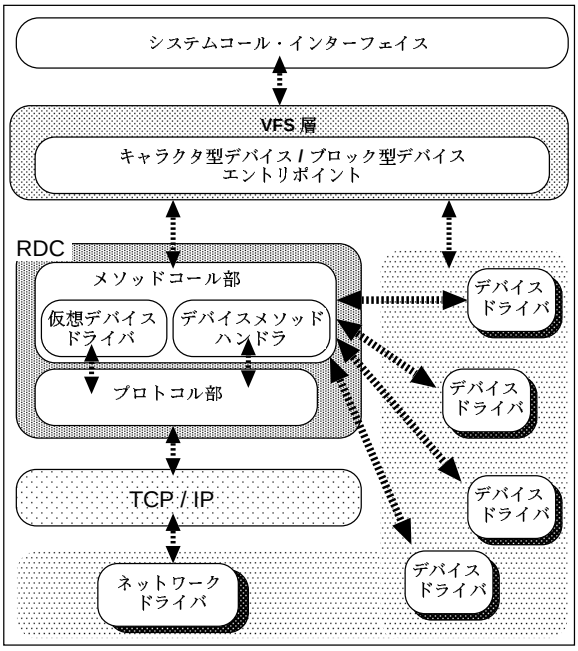

図 2: 提案手法におけるデバイスドライバ・インターフェイス

からも呼び出されるため，既存のカーネル構成に対し大幅な変更を要する．

iSCSI は SCSI トランザクションを IP データグラムにカプセル化するため，NFS の欠点であるデバイスタイプの問題を意識する必要がない．しかし，SCSI デバイスしか利用できない欠点を持つ．OS によっては SCSI エミュレーション機能を有し，非 SCSI デバイスに対し SCSI プロトコルを用いて操作可能であるが，すべてのデバイスが SCSI エミュレーションで操作できる訳ではない．

スタブドライバは，VFS アーキテクチャとデバイスドライバの間に抽象化レイヤを設け，通常のファイルアクセスと同様の仕組みを提供する．また，デバイスメソッド自体を IP データグラム内にカプセル化しノード間で送受信するため，デバイスの種類に依存せずリモート・デバイスの利用が可能である．しかし，スタブドライバ–デバイスデーモンの組を用意する必要があり，リモート・デバイスの利用形態は 1 対 1 に限られ，拡張性に欠ける．

3 提案手法

本章では提案手法の概要について述べる．

3.1 リモート・デバイス・コントロール (RDC)

提案フレームワークは既存環境と親和性を保ち，カーネルレベルの手法で問題となるデバイスタイプの依存性問題を解決する．また，スタブドライバの欠点を解決し，多対多の利用形態が可能な拡張性を持つ．

提案フレームワークは既存の UNIX システムにおける VFS 層とデバイスメソッドの間に位置する (図 2)．VFS 層はファイルオブジェクトの種類を隠蔽するとともに，ファイルオブジェクトの操作関数を明確に定義している．通常，ユーザアプリケーションによるデバイスファイルへの要求は，システムコールから VFS 層へ渡される．VFS 層はデバイスファイルの vnode (または inode) に格納されているメジャー番号をもとに適切なデバイスメソッドを呼び出す．

そこで，VFS 層からデバイスメソッドへ至る従来のインターフェイスにネットワーク的な位置を抽象化するレイヤを設ける．抽象化レイヤはデバイスタイプ，メジャー番号，操作関数を示すフラグなどのデバイス操作情報を IP データグラム内にカプセル化し，リモートノードへ送信する．これによりデバイスタイプに依存しない操作が可能になる．

また，リモート・デバイスを利用するためにノード間で確立された論理的なチャネルをコネクションとして管理し，コネクション ID でチャネルを識別する．コネクションには利用するリモート・デバイスが関連付けられているので，複数のリモート・デバイスを利用することができ多対多の利用形態を可能にする．本稿では，提案フレームワークを，**リモート・デバイス・コントロール** (**Remote Device Control**, 以下 **RDC**) と呼ぶ．

3.2 機能概要

RDC はカーネルに登録するソフトウェアモジュールである．モジュールをプロトコル部とメソッドコール部に分離する．

- プロトコル部：RDC にて定義したプロトコルを処理し，ノード間の通信を処理する部分
- メソッドコール部：OS 内部における実際のデバイスメソッドを処理する部分

プロトコル部はデバイス操作情報のカプセル化と，ソケットコネクションを担当する．メソッドコール部は，ローカルノードとリモートノードにて異なる機能を提供する．すなわち，ローカルノードでは仮想デバイスドライバを提供し，リモートノードではデバイスメソッド・ハンドラを提供する．

プロトコル部は主に RDC プロトコルを処理するため，OS に依存しない構成である．一方，メソッドコール部はデバイスメソッドの要求を代替するため，OS に依存した構成となる．

3.3 セマンティクス

VFS 層と連携して動作するには以下の情報を必要とする．

- ファイルオブジェクト (file 構造体，inode/vnode 構造体)
- ドライバ・エントリポイント

デバイスを操作する場合，通常 VFS 層にてデバイスファイルに関連付けられたファイルオブジェクトのセットアップが行われる．ファイルオブジェクトはデバイス操作に関する様々な情報を持っており，デバイスメソッドを呼び出す時に必要となる．また，ファイルオブジェクトにはドライバ・エントリポイントが含まれている．ドライバ・エントリポイントは関数のポインタをメンバーに持つ構造体であり，デバイスメソッドのアドレスを含んでいる．RDC ではファイルオブジェクトとドライバ・エントリポイントをセマンティクスと定義し，各ノード上で保持される情報とする．

3.4 デバイス操作情報

提案フレームワークは VFS 層にてデバイス操作が明確に定義されている構成を利用し，ノード間で送受信するデバイス操作情報を少なくした．デバイス操作情報とは，以下の情報を意味する．

- デバイスタイプ
- メジャー番号，マイナー番号
- デバイス操作のタイプ

VFS 層はこれらのデバイス操作情報とシステムコールから渡される引数をもとにデバイスメソッドを呼び出す．提案フレームワークでは，デバイス操作情報をプロトコル部にてカプセル化し，ノード間で送受信する．

3.5 仮想デバイスドライバ

仮想デバイスドライバはローカルノードにおけるリモート・デバイス操作のためのドライバ・エントリポイントである．仮想デバイスドライバはデバイスドライバとしてカーネルに登録される．システムコールを通じたデバイスへの要求はすべて仮想デバイスドライバが一旦受け付ける．

Linux における動作例を図 3 に示す．アプリケーションからビデオデバイスに対して read() システムコール要求があると，カーネル内部ではカメラデバイスの read メソッドを，変数として登録されている chr_dev から検索する．変数 chr_dev は配列として用意されており，メジャー番号を添字として一意に指定可能である．仮想デバイスドライバは通常のデバイスドライバ同様 chr_dev に登録されるので，仮想デバイスドライバの read メソッドが呼び出される．仮想デバイスドライバは引数とデバイス操作情報をプロトコル部へ渡し，処理結果を待つ．プロトコル部ではこれらの情報をカプセル化した RDC パケットを生成しリモートノードへ送信する．

デバイスメソッド・ハンドラは受信した RDC パケットからメジャー番号とメソッドタイプを調べ，実際のデバイスメソッド video_read() を呼び出す．デバイスメソッドからの処理結果が返ってきたら，結果をプロトコル部に渡し，プロトコル部は処理結果を RDC パケットにカプセル化しローカルノードへ送信する．

3.6 デバイスメソッド・ハンドラ

デバイスメソッド・ハンドラはリモートノードにおいて，ローカルノードからのデバイスメソッドを実デバイスのデバイスメソッドに変換する．

Linux のマルチメディアデバイスのフレームワークである V4L (Video for Linux) では，ビデオデバイスに対する read() システムコールはカーネル内部で video_read() を呼ぶ．デバイスメソッド・ハンドラはプロトコル部から受け取った引数をもとに video_read() を呼び出す．

デバイスメソッド・ハンドラは仮想デバイスドライバとしてカーネルに登録されない．これはリモート

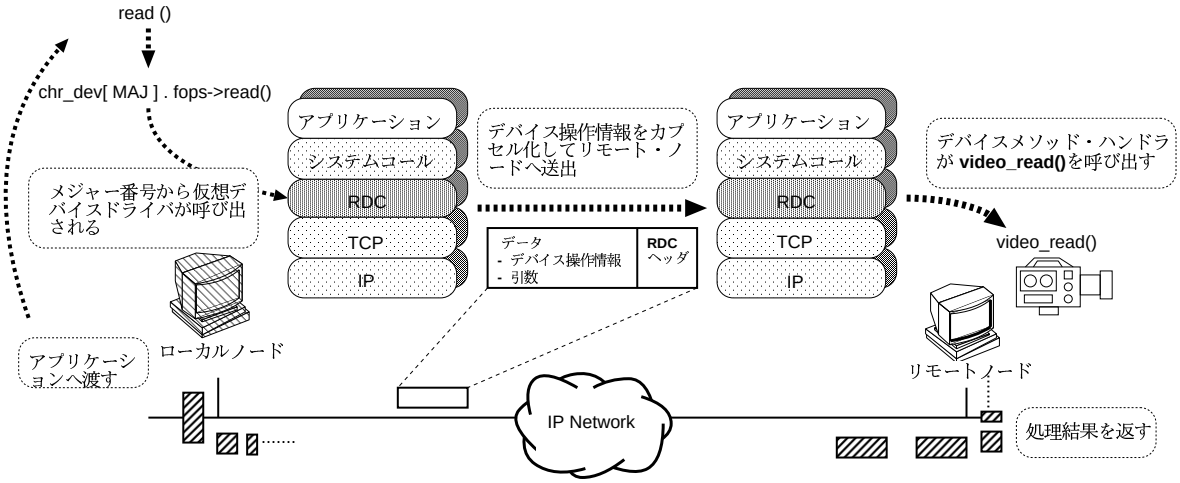

図 3: 動作例

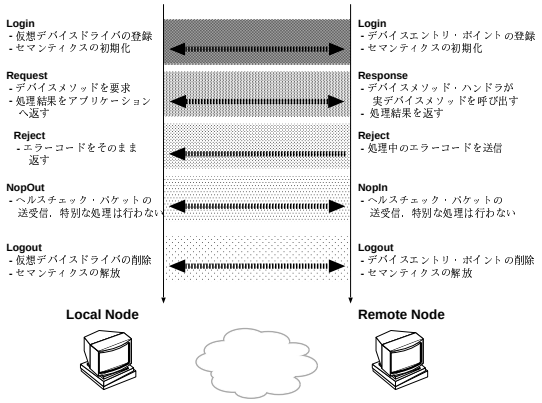

図 4: 各コマンドの動作

ノードではアプリケーションに対してインターフェイスを提供する必要がないからである．リモートノードではビデオデバイスに対する read() システムコールは video_read() を呼ぶ．

逆に，デバイスメソッド・ハンドラが video_read() を誤ってハンドリングすることはない．なぜなら，リモートノードはセマンティクスを別に管理しているからである．

3.7 RDC コマンド

図 4 に RDC における処理手順を示す．RDC は 5 つのフェーズを持ち，各フェーズにおける操作をコマンドとする．RDC プロトコルにて定義するコマンドは以下の通りである．

- Login コマンド
  RDC ノード間のコネクションを開始するコマンド．リモート・デバイスにおけるセマンティクスのセットアップと，ローカルノードに対し仮想デバイスドライバを提供する．

- Logout コマンド
  RDC ノード間で確立されていたコネクションを終了するコマンド．オープンされていたセマンティクスや，ソケットコネクションを終了するとともに，カーネルから仮想デバイスドライバを削除する．

- Request/Response コマンド
  RDC コネクションにて主に使用されるコマンド．デバイスメソッドの送受信を行う．

- NopOut/NopIn コマンド
  ノード間のヘルスチェックを行うコマンド．

- Reject コマンド
  リモートノードにおいてエラーが発生した場合に送信されるコマンド．

Login コマンドはローカルノードとリモートノードの間で論理的なコネクションを確立する．その際，両ノードにてセマンティクスのセットアップが行われ，VFS 層と連携するための準備を行う．コネクション確立後は，Request/Response コマンドを送受信することでリモート・デバイスの利用が可能である．コネクションを終了する場合は，Logout コマンドにより行われる．

表 2: 計算機環境

CPU	PentiumIII 1000MHz
メモリ	512MB
ネットワークインターフェイス	100Mbps イーサネット
グラフィックデバイス	nVIDIA RIVA TNT2 32M
OS	Vine Linux 2.5 (2.4.18)
フレームバッファドライバ	VESA 2.0 FB
コンパイラ	gcc (2.95.3)

NopOut/NopIn コマンドは具体的な処理を行わず，単に ack を返すだけである．これにより，ネットワークの障害や計算機の故障によるコネクションの切断を未然に防ぐ．

Reject コマンドは常にリモートノードから送信される．通知されるエラーはそのまま仮想デバイスドライバへ返される．

4 評価

本章は提案フレームワークのスループットを定量的に評価した．

4.1 評価方法

評価に用いた計算機環境を表 2 に示す．提案フレームワークはカーネルレベルでリモート・デバイスを利用する構成である．そのため，既存のデバイスドライバ・インターフェイスにネットワーク的な位置透過を実現する抽象化レイヤを追加する．そこで，従来の VFS 層からデバイスメソッドへ至るデバイス操作の処理時間と，提案フレームワークを経由した際の処理時間を比較し，提案フレームワークのスループットを計測した．

評価には Linux カーネルのフレームバッファ機能を使い，フレームバッファデバイス /dev/fb に対しユーザ空間のアプリケーションから read()/write() システムコールを行う．評価は以下の 4 つの方法を行い，データの転送に要した時間を計測した．

方法 1: RDC を経由しない
　フレームバッファデバイスに直接 read()/write() を行う

方法 2: RDC を経由する
　ループバックインターフェイスを利用し，ネットワークを介さずに read()/write() を行う

方法 3: RDC を経由する
　リモートノードに接続し，ネットワークを介してフレームバッファデバイスに read()/write() を行う

方法 4: ftp による転送
　ネットワークの帯域を計測する

4.2 実験結果

図 5 は各方法における read 要求の処理時間の結果である．方法 1 と方法 2 を比較した場合，RDC を経由すると 15.09%のオーバーヘッドを要することが分かった．また，方法 1 と方法 3 を比較した場合，データ転送の時間 (方法 4) を除いたオーバーヘッドは 15.78%である．このことから，ループバックインターフェイスではなく，ネットワークインターフェイスを利用したとしても処理時間の差は少ない．

図 6 は write 要求の処理時間の結果である．方法 1 と方法 2 を比較した場合，RDC を経由すると 196.74%のオーバーヘッドを要する．これは RDC を経由せず直接フレームバッファデバイスへ write() した時の約 3 倍以上の時間を要する．同様に，データ転送の時間 (方法 4) を除いて方法 1 と方法 3 を比較した場合も，オーバーヘッドは 199.84%となり約 3 倍以上かかる．

一方，RDC を経由した場合の処理時間の平均を表したのが図 7 である．処理時間は方法 1 と方法 2 の差，および方法 1 と方法 3 の差 (ただし，ネットワークの転送時間 -方法 4- を除く) の平均をとっている．この図から RDC を経由した場合の実効スループットは約 318M ビット /秒となることが分かった．RDC の処理時間は read()/write() ともに，情報量に対して線形に増加する．ネットワークの転送時間を考慮しても，オーバーヘッドはそれ程変化しない．むしろ，転送する情報量に比例して処理時間が増加する．

5 まとめ

本研究では，ネットワーク上に遍在する計算機に直接接続された周辺機器の透過的な利用の実現を目指した．その際，既存技術で問題となるデバイスタイプの依存性や，利用形態のスケーラビリティを解決することを目的とした．提案フレームワークは，

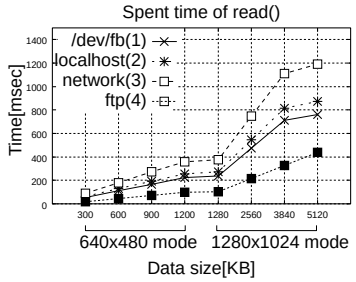

図 5: 実験結果 - read()

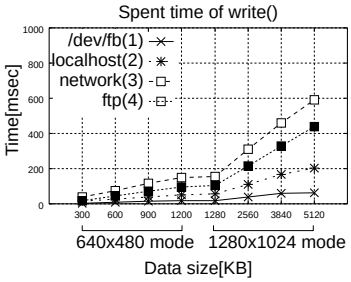

図 6: 実験結果 - write()

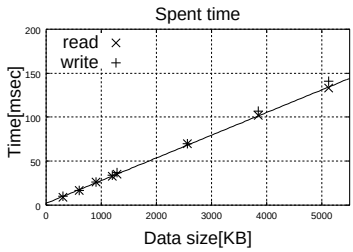

図 7: オーバーヘッドの平均値

カーネルレベルで実現し VFS 層とデバイスドライバ・インターフェイスの間に位置する．VFS 層におけるドライバ・エントリポイントの定義を利用することで，デバイス操作を汎用化し必要最低限の操作情報でリモート・デバイスを操作する．よって，既存のデバイスドライバ・インターフェイスと親和性を保つことができ，カーネルへの変更を必要としない．

また，モジュールとしてカーネルに動的に追加することで複数のノードでリモート・デバイスの利用が可能である．リモート・デバイスの利用に際し論理的なコネクションを確立する．このコネクションは ID によって管理されるため，複数のリモート・デバイスの利用が可能である．コネクション確立後は，仮想デバイスデバイスドライバが VFS からの要求をハンドリングし，リモートノードのデバイスメソッド・ハンドラへ送り，デバイスメソッド・ハンドラが実デバイスメソッドの呼び出しを代替する．

提案フレームワークのスループットを計測するため，フレームワークを介した時に既存のデバイスドライバ・インターフェイスに対しどの程度オーバーヘッドを与えるか定量的に評価した．その結果，実効スループットは約 318M ビット /秒となり，ネットワークの転送時間を考慮しても，オーバーヘッドはそれ程変化しないことが分かった．

今後の課題として，提案フレームワークの汎用性の向上，異なるオペレーティングシステム間での運用，セキュリティ機能の追加など実現する必要がある．

参考文献

[1] “ESD - Enlightened Sound Daemon”, http://www.tux.org/~ricdude/EsounD.html

[2] “SDL - Simple Directmedia Layer”, http://www.libsdl.org

[3] “Microsoft DirectX®”, http://www.microsoft.com/directx/

[4] “Jini Network Technology”, http://www.jini.org/

[5] “The Universal Plug and Play forum”, http://www.upnp.org

[6] S.R.Kleiman, “Vnodes: An Architecture for Multiple File System Types in Sun UNIX”, Proceedings of the Summer 1986 USENIX Conference, Jun 1986.

[7] David S.H. Rosenthal, “Evolving the Vnode Interface”, Proceedings of the Summer 1990 USENIX Conference.

[8] Brent Callaghan, “NFS Illustrated”, Addison Wesley , Dec 1999.

[9] Andrew P.Rifkin, Michael P.Forbes, Richard L.Hamliton, Michael Sabrio, Suryakanta Shah, Kang Yueh, “RFS Architectual Overview”, USENIX, 1989.

[10] Julian Satran, Kalman Meth, Costa Sapuntzakis, Mallikarjun Chadalapaka, Efri Zeidner, “iSCSI”, Internet draft, November 2002, work in progress.

[11] 佐藤友隆，中山健，小林良岳，前川守,「カーネルレベルで実現したネットワーク透過な周辺機器制御の枠組み」，電子情報通信学会，2001 年 6 月．