先進的基盤ソフトウェア 2巻1号（通号2号）オンライン版 2008年11月11日発行
ISSN 1882-4196

# 先進的基盤ソフトウェア

Joint Symposium for Advanced System Software 2008
(JSASS2008)

2008年9月1日(月)・2日(火)

於 東京大学大学院情報理工学系研究科 秋葉原拠点 (東京都千代田区)

JSASS実行委員会

# 巻頭言

東京農工大学　並木　美太郎

2000 年に第 1 回を開催してから本シンポジウムも 9 回目を数えました。初回は立命館大学びわ湖草津キャンパスで開催して以来、名古屋工業大学、立命館大学、龍谷大学、和歌山大学、(株)エスアールアイのご厚意で白浜での開催といくつかの大学持ち回り、幹事の皆様のご協力でここまで継続することができました。昨年より立命館大学の毛利先生のご尽力により報告集と言う形で発表内容などを冊子体および World Wide Web で発行しております。過去の JSASS については http://www.lavender.org/jsass をご参考いただければ幸いです。

さて、普段研究室内で指導教員、先輩・同僚・後輩からの指導、アドバイス、コメントなどをもらいながら自分の研究を進めていると思いますが、グループ外の研究者の意見を聞くことは、新しいアイデアを得る、自分の考えの不足している点、気づかないことを再発見する良い機会になります。学会の大会、研究会、シンポジウムなどの発表で、一度、そこまでの内容を整理し、予稿集としてまとめ発表することは、成果公表と言う視点からは重要です。学会の持つカルチャに従い、自分の研究成果のオリジナリティ、有用性、信頼性を向上する、これはこれで大事なことですが、率直な意見等は時間的なことからも得られにくいこともあります。

JSASS のような研究会では、むしろ、発展途上、未整理でも良いので自分の着想、進むべき方向を話して意見や感想をもらう、または、実験結果についてその妥当性や詳細を議論する、内部実装について紹介し作ったことのある人たちだけが議論できることを話すとよろしいかと思っています。内容は玉石混淆かもしれませんが、すべての研究開発では、自分のアイデアをダイヤの原石を磨くように、長所を伸ばし、欠点を補い、過去に類のない、よりよいものにして行くことが必要不可欠なので単なる成果報告ではなく、より多様な、深い議論ができることが重要だと思っています。再度、報告集を拝見すると今後さらに発展できる研究開発内容ばかりで今後の進展が期待できると思っております。

JSASS の発表内容はその年のトレンドなどを反映する面もありますが、今年も様々なスペクトルの、様々な立場でのご発表があり、大変興味深いものだったと思います。聴衆からは様々な質問、コメントがあったかと思いますが、素

直な眼で見つめ、自分の研究開発の役に立つように取捨選択していただければと思います。また、夜の懇親会などで先生それから学生さん同士意見交換を行い、親交を深めていただいたように思います。人脈は人生の宝なので、今後も機会がありましたら親交を続けていただければと思います。

いずれにせよ、研究開発では、自分の創意工夫、何よりある種の面白さを大事にしていただければ思っています。自分の考えたこと、自分の作ったものが面白い、役に立つ、今後に期待できる、など、計算機科学・工学、情報工学分野において、モノを生み出す技術者、本質を追究する研究者の愉しみを持って今後も研究開発を進めていただければと思っております。「自分の研究開発はここが凄い！」と胸をはって伝えることが、この分野の、何より発表している方々の今後の糧になるものと考えています。

最後に毎年、開催、とりまとめについて幹事団の皆様のご協力をいただいております。今年は、龍谷大学の芝先生が幹事団長としてとりまとめを、ローカルアレンジとして東京大学の笹田先生が秋葉原での準備と実施を、そして、昨年に続き毛利先生に報告集のとりまとめと印刷・発送を行っていただきました。また、報告集の発刊について立命館大学の大久保先生にご尽力をいただきました。皆様のご尽力に深く感謝いたします。

2008 年 10 月 6 日記

Joint Symposium for Advanced System Software 2008 (JSASS2008)

2007 年 9 月 1 日(月)・2 日(火)

東京大学大学院情報理工学系研究科 秋葉原拠点（東京都千代田区）

# プログラム

■ 9 月 1 日(月)

○ オープニング: 14:00～14:10

○ セッション 1: 14:10～16:10 オペレーティングシステム



○ セッション 2: 16:20～17:50 セキュリティ



○ 懇親会: 18:30～20:30

■ 9 月 2 日(火)



○ クロージング: 16:50～17:00

# 仮想計算機モニタ Xen における準仮想化環境での OS 間インターフェースの調査

荒木 裕靖 †　　毛利 公一 ††
† 立命館大学大学院理工学研究科　†† 立命館大学情報理工学部

## 1　はじめに

近年，仮想化技術を搭載したマルチコアプロセッサが登場し，普及してきている．従来のシングルプロセッサでは，周波数の高速化で処理能力を向上させていたが，このような方法での処理能力の向上は，発熱や消費電力の問題がある．そのため，複数のプロセッサを利用し，これらに処理を分散させることで処理能力の向上を図る方法が注目されている．マルチコアプロセッサを搭載した計算機は，従来のシングルプロセッサを搭載した計算機と比べ，並列処理を利用したプロセスの実行が優位な環境である．このような優位点は対称型マルチプロセッシングでも見られるが，マルチコアプロセッサではキャッシュを共有することでさらに効率的な処理が可能となっている．

また，近年注目されている仮想化技術は，物理的な計算機資源を OS から隠蔽し，OS に仮想計算機 (Virtual Machine，以下 VM と記す) を提供する．1 台の計算機上に複数の OS を起動させることが可能となっており，このように提供されている VM には，物理的な計算機の資源を自由に割り振ることが可能となっている．近年，高性能化している計算機資源では，今までの計算機 1 台に対して OS1 つといった利用方法と比べ，仮想化技術を用いて 1 台の計算機上で複数の OS を起動させることでより柔軟な資源利用が可能となる．

このようにマルチコアプロセッサと仮想化技術を組み合わせて利用することで，複数の VM に物理的なコアをそれぞれ割り当てることが可能となり，また，割り当てるコア数を指定することで，それぞれの VM の処理能力を指定することが可能となった．これにより，従来，計算機に依存していた OS の処理能力を自由に変更することが可能となり，OS の利用状況やプロセス特性によってプロセッサの割当てを指定することで，計算機資源を有効に利用できるようになった．

しかし，計算機の利用法によっては，プロセッサ利用率が低い場合も多く，その場合はプロセッサの処理能力は常に必要ではない．一方で，数値計算やサーバなどでは利用率が高い場合が多い．このように，ある VM に割当てられているプロセッサが利用されておらず，別の VM に割り振られているプロセッサの処理能力が不足しているような場合，システム全体としては効率的な計算機利用が出来ているとはいえない．

従来の VM 環境では OS と仮想計算機モニタ (Virtual Machine Monitor，以下 VMM と記す) がそれぞれ個別にスケジューリングを行っていた．VMM は，VM 上の仮想プロセッサに対し物理プロセッサを割当て，OS はプロセスに対しプロセッサを割当てて動作している．このような，個別に資源管理を行っている環境では効率的な計算機資源の利用はできない．そこで，VMM と OS のスケジューリングを協調させることで効率的な資源利用が可能な，協調型仮想計算機システムを提案する．

本システムでは，協調したスケジューリングを行うためシステムで利用する VMM と OS に，協調機構が必要となる．

協調型仮想計算機システムを用いることで，複数の OS が 1 台の計算機上で動作するような VM 環境で，プロセッサコアを OS の状況に応じて，適切に OS へ配分することができ，より効率的で適応的な資源利用を実現する．コア資源を動的に配分することによって，必要の無いと思われるコアを停止させ消費電力を抑えることが可能となり，また，動作していないコアを処理能力を必要としている OS に割り当てることで，各 OS の要求を満たすことが可能になる．

## 2　協調型計算機システムと Xen

本研究では，協調型仮想計算機システムの VMM ターゲットとして，Xen[1] を用いることとする．Xen は完全仮想化と準仮想化の二つの仮想化環境を提供している．完全仮想化は，ハードウェアの仮想化支援機構を利用し，OS の特権命令をエミュレートすることで仮想化環境を実現している．そのため，ゲスト OS へ変更を加える必要が無い．一方，準仮想化は，特権命令を直接実行できないゲスト OS のために，ゲスト OS へ変更を加え特権命令をハイパーバイザコールを用いることで仮想化環境を提供している．この二つの仮想化環境のうち，協調型仮想計算機システムでは，以下のような理由から準仮想化環境を利用する．

- **OS 内部に協調機構を実装する必要がある**
  VMM と OS を協調させるため双方に改良を加える必要があり，完全仮想化のようなゲスト OS へ変更を加えない仮想化環境の利用方法では実現不可能である．
- **OS-VMM 間の通信を行う必要がある**
  Xen に実装されている共有ページを利用し，容易に OS-VMM 間の通信，情報共有機構を実装できる．
- **ハイパーバイザコールを監視することで OS の状況を把握できる**
  どのようなハイパーバイザコールが実行されたかを監視することで，ある程度 OS の動向を把握できる．

準仮想化環境でゲスト OS を起動させるためには，ハイパーバイザコールの利用が必要である．そこで，本研究で，ゲスト OS のターゲットとして利用する Lavender[2] を Xen 準仮想化に対応させるために，ハイパーバイザコールの調査を行った．調査は，Xen-3.2.0 のソースコードに付属の mini-os をもとに行った．

## 3 ハイパーバイザコールの調査

### 3.1 mini-os の調査

mini-os では，割り込みの初期化やメモリ管理といったハードウェアの制御を行う処理で，ハイパーバイザコールが利用されている．ハードウェアの制御は，特権命令が使用されるが，ゲスト OS は，仮想化環境上で特権命令が実行できない．そのため，このような処理でハイパーバイザコールが利用される．

ハイパーバイザコールは，ゲスト OS 内で関数として実装され，スタブを利用して呼び出される．図 1 に，mini-os でのハイパーバイザコール実装例を示す．図 1 では，trap.c で割り込みの初期化を行っているが，その処理は，ハイパーバイザコールによって行われている．hypercall-x86_32.h で宣言されている HYPERVISOR_set_trap_table() という関数に，割込みの設定が格納されている trap_table という構造体を引数として渡し，ハイパーバイザコールを呼び出すことで，割込みの初期化を行っている．

このように，ハイパーバイザコールが呼び出されると，ハイパーバイザコール用に確保されたメモリから，該当するハイパーバイザコール関数のメモリ番地を参照する．ここで得られたアドレスを呼び出すことで，ハイパーバイザコールが実行され，実行結果が Xen によって VM 上に反映される．

trap.c

```
void trap_init(void)
{
    HYPERVISOR_set_trap_table(trap_table);
}
```

hypercall-x86_32.h

```
static inline int
HYPERVISOR_set_trap_table(trap_info_t *table)
{
    return _hypercall1(int, set_trap_table, table);
}



#define _hypercall1(type, name, a1)                 \
({                                                  \
    long __res, __ign1;                             \
    asm volatile (                                  \
        "call hypercall_page + name * 32)\u201c  \
        : "=a" (__res), "=b" (__ign1)               \
        : "1" ((long)(a1))                          \
        : "memory" );                               \
    (type)__res;                                    \
})
```

図 1 **mini-os でのハイパーバイザコール呼出し**

3.2 節では，以上の点をふまえ，ゲスト OS への実装手順について述べる．

### 3.2 ゲスト OS への実装手順

まず，ゲスト OS 内でハイパーバイザコールの必要な処理を列挙する．ここでの，ハイパーバイザコールの必要な処理とは，主にハードウェアの操作が必要な処理のことである．例として，図 1 で示したような割込みの初期化，メモリ操作，コンソールへの入出力などが挙げられる．

次に，これらの処理で使用するハイパーバイザコールを選択し，利用のための実装を行う．Xen に用意されている各ハイパーバイザコール関数のアドレスを，格納するためのメモリ領域を確保し，ゲスト OS 内で利用するスタブ，関数を実装する．

最後に，前述したゲスト OS 内の処理部分を，ハイパーバイザコール呼出し関数へと変更することで Xen 準仮想化環境での動作が可能となる．

## 4 おわりに

本稿では，VMM と OS が協調し動作する協調型仮想計算機システムを提案した．VMM と OS が協調しスケジューリングを行うことで，柔軟な資源管理が可能となり，実計算機以上のパフォーマンスを実現することができる．また，協調型仮想計算機システム構築のため，VMM-OS 間インターフェースであるハイパーバイザコールの調査を行い，ゲスト OS への変更点について述べた．

今後の予定として，本稿で挙げた，ハイパーバイザコールの実装を行い，ターゲットである Lavender を Xen の準仮想化環境で起動させるといったことが挙げられる．また，協調型仮想計算機システムに必要な各機構について調査と考察を行い，順次実装を行ってゆく．

## 参考文献


[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield:“Xen and the Art of Virtualization,” In Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, pp. 164–177 (2003).

[2] 毛利公一，大久保英嗣:”マイクロカーネル Lavender の設計と開発”，電子情報通信学会論文集 (D-I)，Vol.J82-D-I，No.6，pp.730-739，(1999).

# Xenにおける準仮想化環境での VMM-OS間インターフェースの調査

立命館大学理工学研究科
毛利研究室
荒木裕靖

## 目次

- はじめに
  - 協調型仮想計算機システムの提案
- 仮想計算機モニタXen
  - 完全仮想化と準仮想化
  - 協調型仮想計算機システムでのXenの利用環境
- mini-osの調査
  - ハイパーバイザコールの利用
  - ハイパーバイザコールの実装方法
- おわりに
  - まとめ

# はじめに(1/4)

- マルチコアプロセッサと仮想化技術の普及
  - 家庭用計算機だけでなくサーバ，組込みなどの広い分野での研究，利用
- 協調型仮想計算機システムの開発
  - 従来のシステムより柔軟に計算機資源の利用が可能
  - システム全体で，見かけ上，実計算機以上のパフォーマンスを実現可能

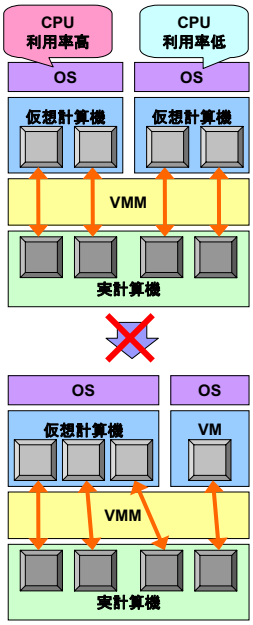


# はじめに(2/4)

- 従来の仮想計算機環境
  - 独立した資源管理
  - 各々の動作状況を把握していない
- 構築時に割り当てられた資源を独占する
  - 動的な資源の割当てに対応していない

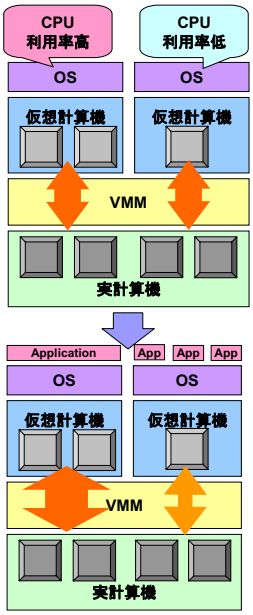
⇒ **協調型仮想計算機システムの利用**

- 本来の実計算機以上のパフォーマンス
  - 切り替えることで見かけ上，システム上に多くの資源が搭載されているように見える
- 協調した資源管理を実現
  - OSの状況を細かく知ることができる
  - 変動させるタイミングを決定できる



# はじめに(3/4)

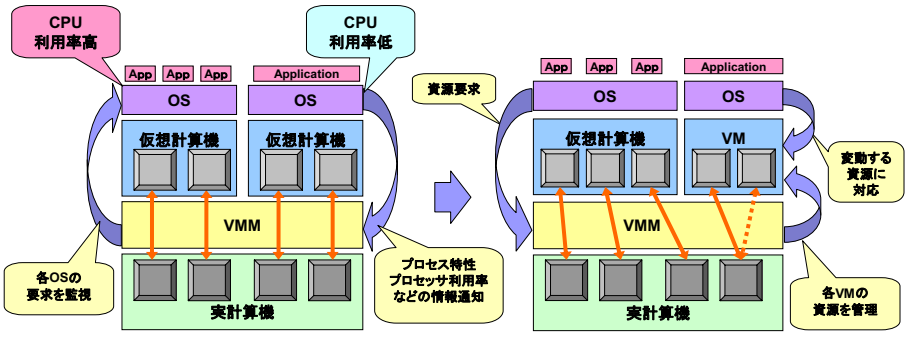
- 構築時に資源を処理能力として割当てる
  - 動的に資源の配分を変更することが可能
- 問題点
  - OSの都合を考えない資源配分
  - OSの知らぬところで資源の配分を変更される
  - OSが必要としている時に必要な資源を割当てることが不可能
- 協調動作することで上記の問題を解決
- VMMとOSの協調
  - 協調型仮想計算機システムの構築






# はじめに(4/4)

- 協調型仮想計算機システム
  - 仮想化技術とマルチコアプロセッサの利用




**システム全体として，見た目上，実計算機以上のパフォーマンスを実現**

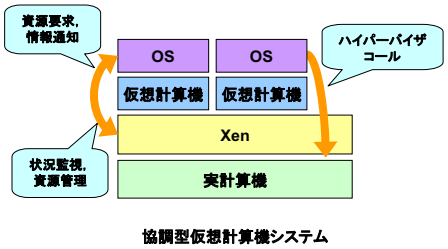
# 仮想計算機モニタXen

- 完全仮想化
  - 仮想化支援機構を利用
  - 特権命令をエミュレートする
  - ゲストOSに対しての変更を加える必要が無い
- 準仮想化
  - 特権命令を実行するためにハイパーバイザコールを利用
  - ゲストOSに対して変更を加える必要がある



# 協調型仮想計算機システムとXen

- OSとVMMが協調する必要がある
  - 協調機構の実装が必要
  - VMM-OS間の通信が容易
  - ハイパーバイザコールを追跡することでOSの要求を監視できる




協調型仮想計算機システム

- 協調型仮想計算機システムでは準仮想化を利用

# 準仮想化環境

- ゲストOSは特権命令を実行できない
  - ハイパーバイザコールを利用する
- ハイパーバイザコール
  - ゲストOSが特権命令を利用するために使用
  - 特権OSがゲストOSを管理するために使用
  - Xenが仮想計算機に処理内容を反映する
- 準仮想化環境でゲストOSを動作させるためには
  ハイパーバイザコールの利用が不可欠



# ハイパーバイザーコールの調査

- mini-osでの実装を参考に調査
  - 割り込みの設定，メモリ管理などで利用
  - 呼び出しのために関数を設定
  - コールのためにスタブを利用
- Xenの機能を利用する箇所をすべて置き換え
  - 準仮想化での起動が可能

# ハイパーバイザコール一覧

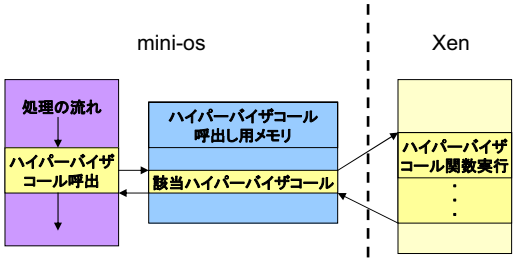
- ハイパーバイザコール(番号順)
  - x86-32bit環境
  - 色付はmini-osで使用されているもの(26個)

| | | | |
|---|---|---|---|
| set_trap_table | update_descriptor | vm_assist | xenoprof_op |
| mmu_update | memory_op | update_va_mapping_otherdomain | event_channel_op |
| set_gdt | multicall | iret | physdev_op |
| stack_switch | update_va_mapping | vcpu_op | hvm_op |
| set_callbacks | set_timer_op | set_segment_base | sysctl |
| fpu_taskswitch | event_channel_op_compat | mmuext_op | domctl |
| sched_op_compat | xen_version | xsm_op | kexec_op |
| platform_op | console_io | nmi_op | |
| set_debugreg | physdev_op_compat | sched_op | |
| get_debugreg | grant_table_op | callback_op | |



# ハイパーバイザコール呼出し手法

1. ハイパーバイザコール呼出
2. ハイパーバイザコール呼出し用に確保されたメモリを参照
3. 利用するハイパーバイザコールに該当するメモリ番地を参照
4. ハイパーバイザコール関数へジャンプ
5. ハイパーバイザコール実行



# ハイパーバイザコール実装例 (mini-os)

- trap_init()で割り込みの設定
  - trap_tableはtrap_info_t型の構造体の配列
- HYPERVISOR_set_trap_table()で スタブを利用
- スタブ内でhypercall_pageという ハイパーバイザコール用メモリ領域を参照
- hypercall_page以降にXenの ハイパーバイザコール関数が存在

trap.c

```
void trap_init(void)
{
    HYPERVISOR_set_trap_table(trap_table);
}
```

hypercall-x86_32.h

```
static inline int
HYPERVISOR_set_trap_table(trap_info_t *table)
{
    return _hypercall1(int, set_trap_table, table);
}
                    .
                    .
                    .
#define _hypercall1(type, name, a1)          ¥
({                                           ¥
    long __res, __ign1;                      ¥
    asm volatile (                           ¥
        "call hypercall_page + name * 32)“   ¥
        : "=a" (__res), "=b" (__ign1)        ¥
        : "1" ((long)(a1))                   ¥
        : "memory" );                        ¥
    (type)__res;                             ¥
})
```



# ゲストOSへの実装手順

1. 特権命令，Xenのサポートの必要な箇所の列挙
2. 必要なハイパーバイザコールの準備
   1. ハイパーバイザコール用メモリの確保
   2. スタブの作成
3. ハイパーバイザコールへの置き換え

# おわりに

- まとめ
  - 協調型仮想計算機システムの提案
  - 準仮想化環境でOSの実行方法
    - ハイパーバイザコールの利用
    - ゲストOSへの改良点
- 今後の予定
  - ゲストOSへハイパーバイザコールの実装
  - 準仮想化環境での起動と動作確認
  - 協調型仮想計算機システムについての考察

# ファイルの使用頻度に基づくバッファキャッシュ制御法の評価

片上　達也　　田端　利宏　　谷口　秀夫

岡山大学大学院自然科学研究科

## 1. はじめに

既存の多くのオペレーティングシステム（OS）は，バッファキャッシュを制御する単位をブロックとし，置き換えアルゴリズムとしてLRU 方式を用いている．しかし，LRU 方式では，特定の読み込みパターンにおいて著しく性能が低下するという問題がある．これに対して，我々は，ファイルの使用頻度に基づくバッファキャッシュ制御法を提案した[1]．ここでは，提案手法を評価し，提案手法の有効性を示す．

## 2. 評価

### 2.1 評価項目

LRU 方式でキャッシュヒット率を低下させる読み込みパターンについて，提案手法の性能を評価し，提案手法の有効性を示す． LRU で性能を低下させる読み込みパターンとして，具体的には，キャッシュのサイズより大きいファイルの読み込みの影響について述べる．

### 2.2 評価における計算式と閾値

文献[1]で述べた重要度の計算方法に基づき，重要度の計算式を式(1) とした．式(1) において，$I$ は重要度，$N_{open}$ はオープン回数，$F_{num}$ は参照数，$k$ は定数，$F_{size}$ はファイルサイズ，$B_{size}$ は 1 ブロックのサイズである．

$$I = \frac{N_{open} + F_{num} \times k}{\left[\frac{F_{size}}{B_{size}}\right] + 1} \qquad \text{式(1)}$$

なお，式(1) では，オープン回数と参照数の重要度計算への影響の違いを考慮し，定数 $k$ を導入した．本評価では，$k = 10$ とした．

文献[1]で述べた重み関数 $w$ を式(2) とした．式(2)において，$N_{open}$ はオープン回数，$Dec$ は重みである．

$$w(N_{open}) = Dec \times N_{open} \qquad \text{式(2)}$$

評価における各閾値を以下に示す．

(1) $Rc$：重要度を計算した直後から open または close システムコールが発行された回数の総和が $Rc$ になると，重要度を再計算

Evaluation of I/O Buffer Cache Mechanism Based on the Frequency of File Usage
Tatsuya Katakami, Toshihiro Tabata and Hideo Taniguchi
Graduate School of Natural Science and Technology, Okayama University

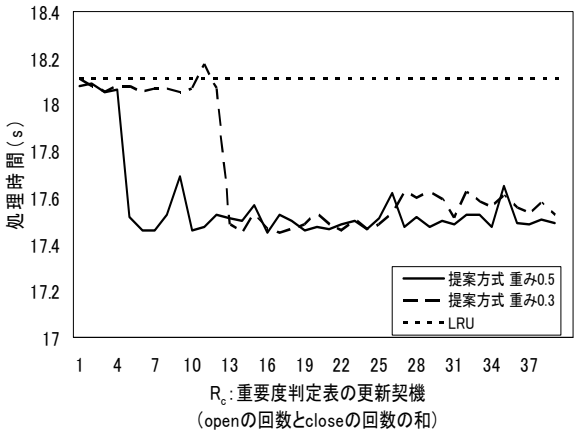



図 1　キャッシュのサイズより大きいファイルの読み込みの影響

### 2.3 評価結果

ファイル A，B，C，D，E に対して，A → B → A→ C → A → D → A → E という順で，ファイルをシーケンシャルに読み込む操作を 25 回繰り返すプログラムの処理時間とキャッシュヒット率を測定し，LRU 方式と比較する．評価で利用するファイルを以下に示す．

( 1 ) ファイル A：4.0 KB

( 2 ) ファイル B，C，D，E：3.0 MB（バッファキャッシュのサイズ）

評価結果を図 1 に示す．図 1 より，重みが 0.5 で $R_c$ が 6 のときに，提案手法は，LRU 方式と比較して，処理時間を最大 0.65 秒短縮する．これにより，提案手法は，サイズの大きいファイルの読み込み時の性能低下を抑制すると言える．

## 3. おわりに

ファイルの使用頻度に基づくバッファキャッシュ制御法の評価について述べた．具体的には，LRU 方式でキャッシュヒット率を低下させる読み込みパターンとして，キャッシュのサイズより大きいファイルの読み込みの影響を評価した．評価の結果，提案手法は，従来方式（LRU）と比較して，処理時間を最大 0.65 秒（4.4 %）短縮した．

## 参考文献

[1] 片上達也，田端利宏，谷口秀夫，渡辺弘志，乃村能成：ファイルの使用頻度に基づくバッファキャッシュ制御法，情報処理学会研究報告 2008-OS-108，Vol.2008，No.35，pp.115-122 (2008).

# ファイルの使用頻度に基づく バッファキャッシュ制御法の評価

片上達也‡　田端利宏‡　谷口秀夫‡

‡岡山大学 大学院自然科学研究科

# 目次



2

# 背景

メモリ：速い ⟷ ディスク：遅い

バッファキャッシュ制御はOSの重要な仕事

- 使用頻度の高いデータをメモリ上に蓄える機能
  - ブロック単位で制御（ファイルを意識しない）
  - LRU方式で制御するOSが多数

LRUの欠点：(1) 大ファイルに対するシーケンシャルな読み込み
(2) キャッシュミスを伴うループ読み込み

一方，応用プログラム（AP）では，ファイル単位でデータを意識

- AP毎に利用するファイルが異なる
- 同じファイルでも，AP毎に参照頻度や入出力パターンが異なる

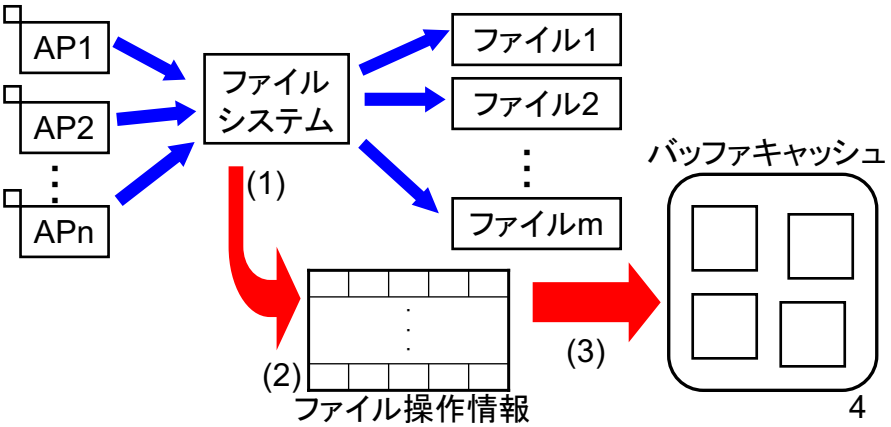
ブロック制御にファイルを意識することでAPの入出力性能を向上

3

# ファイルの使用頻度に基づく制御法

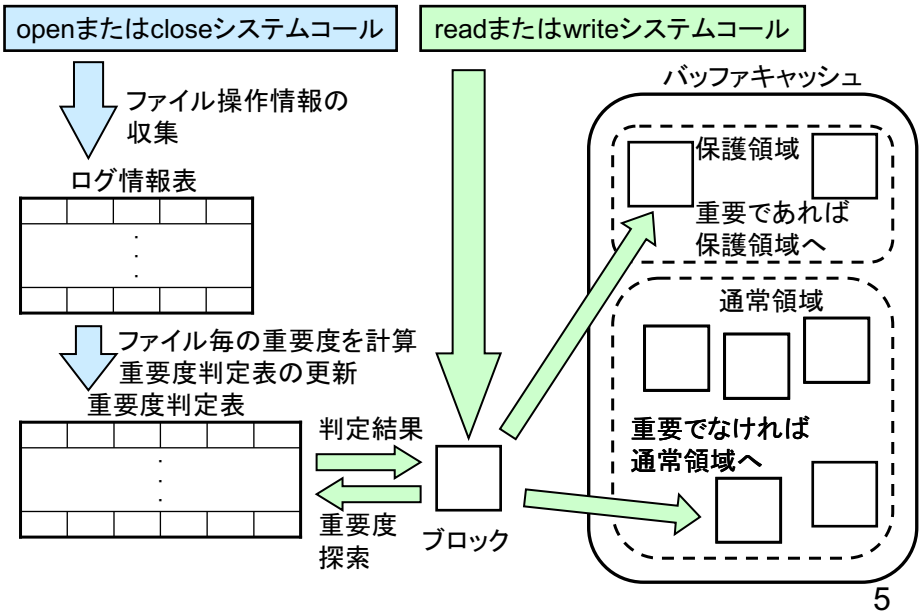
ファイルの使用頻度に基づくバッファキャッシュ制御法

(1) APの操作したファイルの情報（ファイル操作情報）を収集
(2) ファイル操作情報から，ファイル毎に重要度を算出
(3) 重要度に基づきバッファキャッシュを制御




4

# 基本方式




5

# 課題

(1) 重要度の計算方法
提案方式は重要度の計算法によって性能が大きく変化するため，重要度には，計算方法が必要

(2) 収集するファイル操作情報
キャッシュヒット率を向上させるために，ファイル操作情報として，適切な情報を収集する必要あり

(3) 重要度判定表の更新方法
計算した重要度を管理するために，重要度判定表の更新契機や，更新方法を決定する必要あり

6

# 課題1:重要度の計算方法

＜重要度計算に利用する情報（ファイル毎に計算）＞

(1) 参照数（現在ファイルをオープンしているプロセスの数）
参照数が大きいほど今後読み込まれる可能性が大

(2) オープン回数（ファイルをオープンした回数）
オープン回数が大きいほど今後読み書きされる可能性が大

(3) ファイルサイズ（アクセスしたファイルの大きさ）
サイズの大きいファイルを読み込むことによって発生するバッファキャッシュの占有を防止

重要度計算に与える影響

参照数＞オープン回数＞ファイルサイズ

今後読み書きされる可能性を考慮

7

# 課題2:収集するファイル操作情報

ファイル操作情報

| (1) 参照数 | (2) ファイルサイズ | (3) オープン回数 | (4) iノード番号 | (5) 最新オープン時刻 |
|---|---|---|---|---|
| | | | | |
| | | ・<br>・<br>・ | | |
| | | | | |

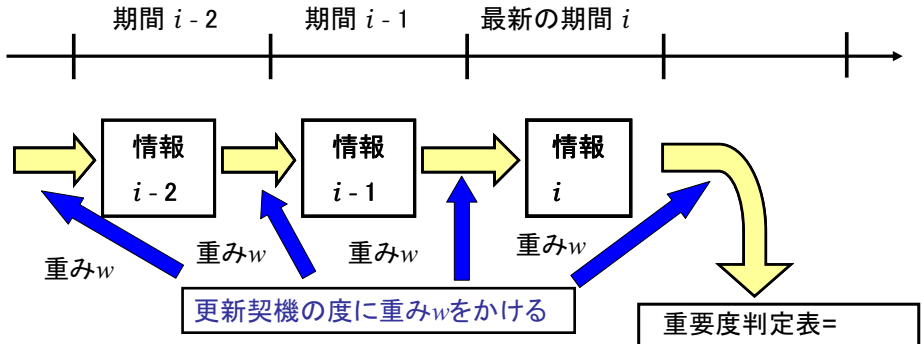
(1) 重要度計算に必要な情報
（参照数，オープン回数，ファイルサイズ）

(2) ファイルを特定するためのiノード番号

(3) 2つのファイルの重要度が等しい場合に，順位を付けるための最新オープン時刻

8

# 課題3:重要度判定表の更新方法




更新契機:
1: 前回の更新契機から一定回数オープン，クローズが行われたとき
2: ログ情報表に一定数のファイル操作情報が蓄積されたとき

9

# 期待される効果

(1) AP使用時の入出力性能の向上
APの入出力振る舞いに合わせたバッファキャッシュ制御により，AP使用時の入出力性能が向上

(2) サイズの大きいファイル利用時の運用中のサービスへの影響を抑制
重要度の算出において，ファイルサイズを意識

↓

サイズの大きいファイルによってバッファキャッシュが埋め尽くされることを防止し，運用中のサービスの入出力性能の低下を抑制

(3) バックアップによる運用中のサービスへの影響を抑制
バックアップ処理によって読み込まれるファイルは1度しかオープンされず重要度が低いために，保護されにくい

↓

運用中のサービスで利用されるファイルを構成するブロックが保護され，運用中のサービスの入出力性能の低下を抑制

10

# 評価条件

＜評価環境＞
(1) OS: FreeBSD 4.3-RELEASE
(2) CPU: Pentium4(1.95GHz)
(3) 実メモリ: 512MB（バッファキャッシュの大きさは3.0MB）
(4) VMIO機能: 無効（バッファキャッシュの効果を明確化）

＜評価において設定する計算式と閾値＞
重要度の計算式 （$I$は重要度，$N_{open}$はオープン回数，$F_{num}$は参照数，$k$は定数，$F_{size}$はファイルサイズ，$B_{size}$はブロックサイズ）

$$I = \frac{N_{open} + F_{num} \times k}{[\frac{F_{size}}{B_{size}}] + 1}$$ (本評価では，$k = 10$)

重み関数 （$Dec$は重み） $w(N_{open}) = N_{open} \times Dec$

閾値 (1) $R_c$ : 重要度再計算契機



# 評価項目

＜基本評価＞
評価1: キャッシュのサイズより大きいファイルの読み込みの影響
評価2: キャッシュの上限ブロック数を上回る数のファイルのループ読み込み

＜応用評価＞
評価3: カーネルのmake処理
評価4: バックアップ処理中のカーネルのmake処理

# 評価1

＜評価内容＞

ファイルA, B, C, D, Eに対して, A→B→A→C→A→D→A→E
という順でファイルを読み込む操作を25回繰り返す

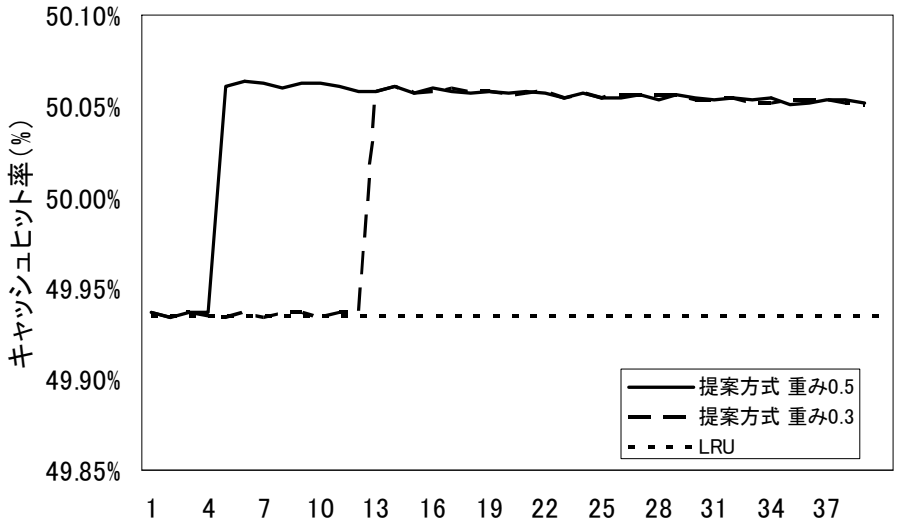
(1) ファイルA: 4.0 KB
(2) ファイルB, C, D, E: 3.0 MB（バッファキャッシュ
の上限値以上の値）

＜評価条件＞

(1) readシステムコール：4.0 KBずつ発行
(2) 重要ファイルの数：1（ファイルAを保護するため）

13

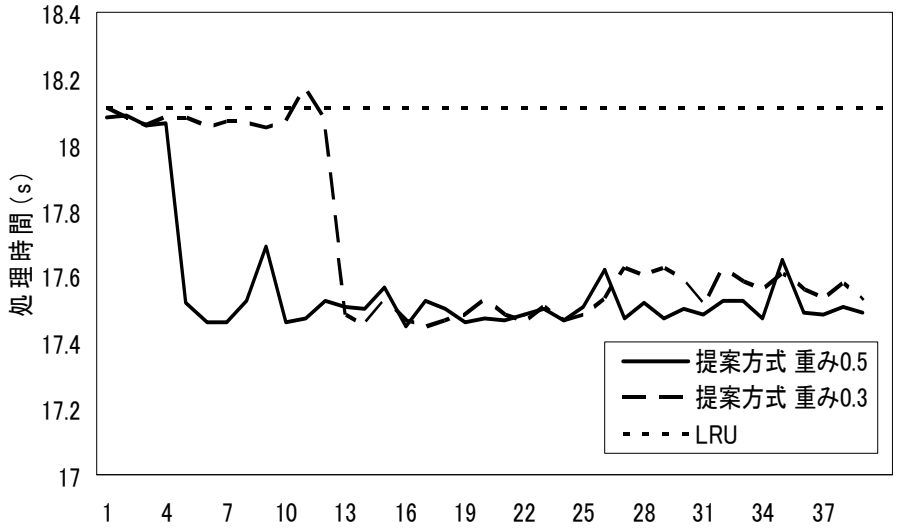
# 評価1におけるキャッシュヒット率




$R_c$：重要度判定表の更新契機
（openの回数とcloseの回数の和）

14

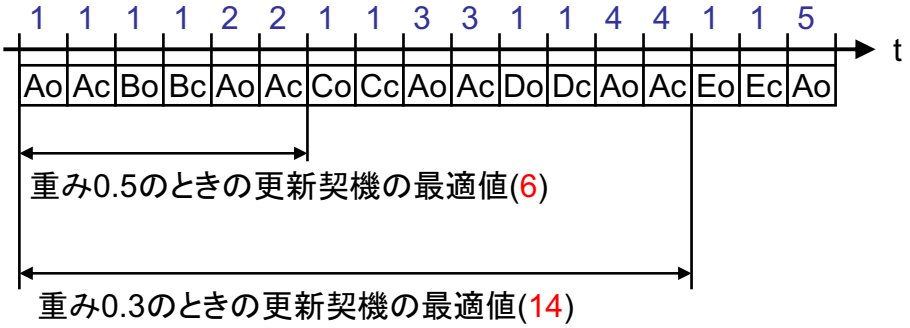
# 評価1における処理時間




$R_c$：重要度判定表の更新契機
（openの回数とcloseの回数の和）

15

# 評価1における重要度判定表の更新契機




アルファベット大文字：ファイル名
アルファベット小文字：openのときはo
closeのときはc
数字：同じ操作を行った回数

評価1における更新契機の最適値：
保護したいファイルが重みによって削除されない最小の値

16

# 評価2

＜評価内容＞

8 KBのファイル500個を読み込む動作を100回繰り返すプログラムの処理時間とキャッシュヒット率を測定

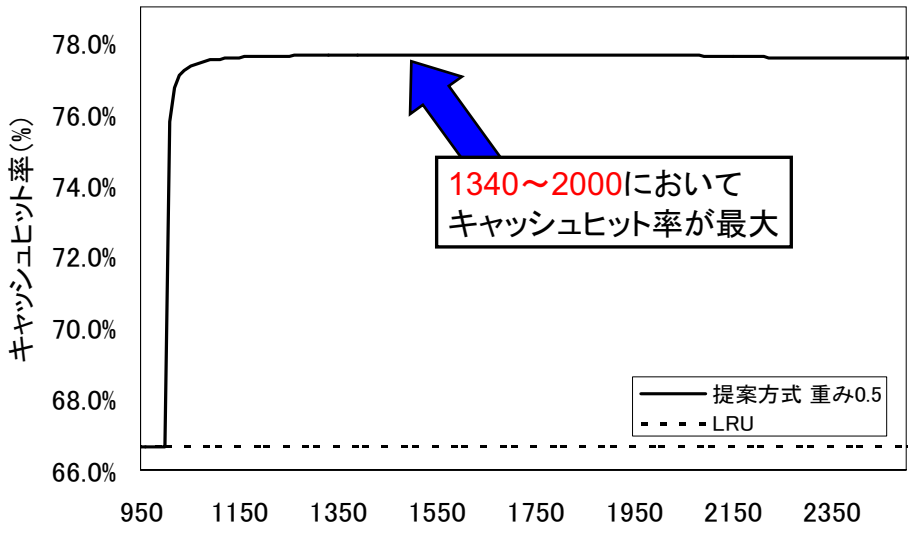
＜評価条件＞

(1) readシステムコール：4.0 KBずつ発行
(2) 重み: 0.5
(3) 重要ファイルの数：170（バッファキャッシュの上限値）

本評価におけるLRUのキャッシュヒット率は，66.7%と予測可能
(1) whileループの終了条件は，readシステムコールの戻り値が0のとき（8KBの読み込みに3回のreadシステムコール）
(2) 評価全体で50,000回のopenシステムコールと，150,000回のreadシステムコールを発行
(3) 先読みにより，50,000回の読み込みはキャッシュヒット

17

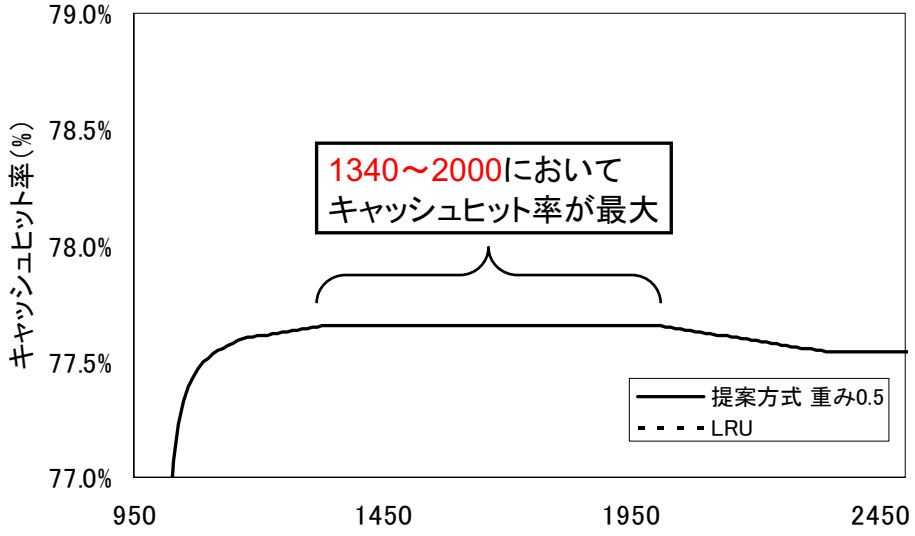
# 評価2におけるキャッシュヒット率




$R_c$：重要度判定表の更新契機
（openの回数とcloseの回数の和）

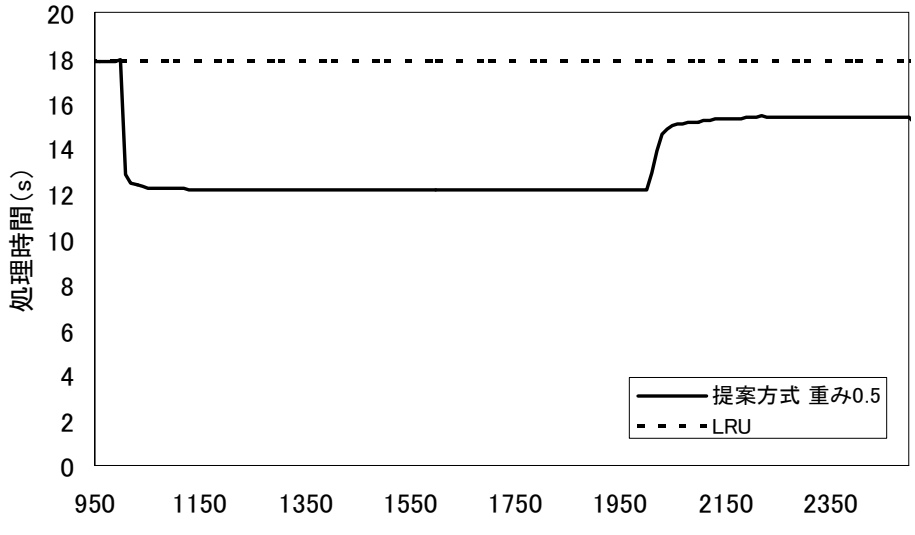
18

## 拡大:評価2におけるキャッシュヒット率




19

## 評価2における処理時間




20

# 更新契機の予測最適値

評価1および評価2より，導き出せる予測最適値
(1) 重みによって，重要なファイルが削除されない最小の値以上
(2) 重要なファイルが再度読み込まれる最小の値以下

これにより，以下の式が確からしいといえる

$$U_c[1/w] + 2F_p \leq R_{\max} \leq U_c([1/w]+1)$$

(1) $R_{max}$: 最もキャッシュヒット率を向上させる重要度判定表の更新契機
(2) $U_c$: 1度オープンされて次にオープンされるまでの間隔の保護したいファイルにおける最大値
(3) $w$: 重み
(4) $F_p$: 保護したいファイルの数

21

# 評価3

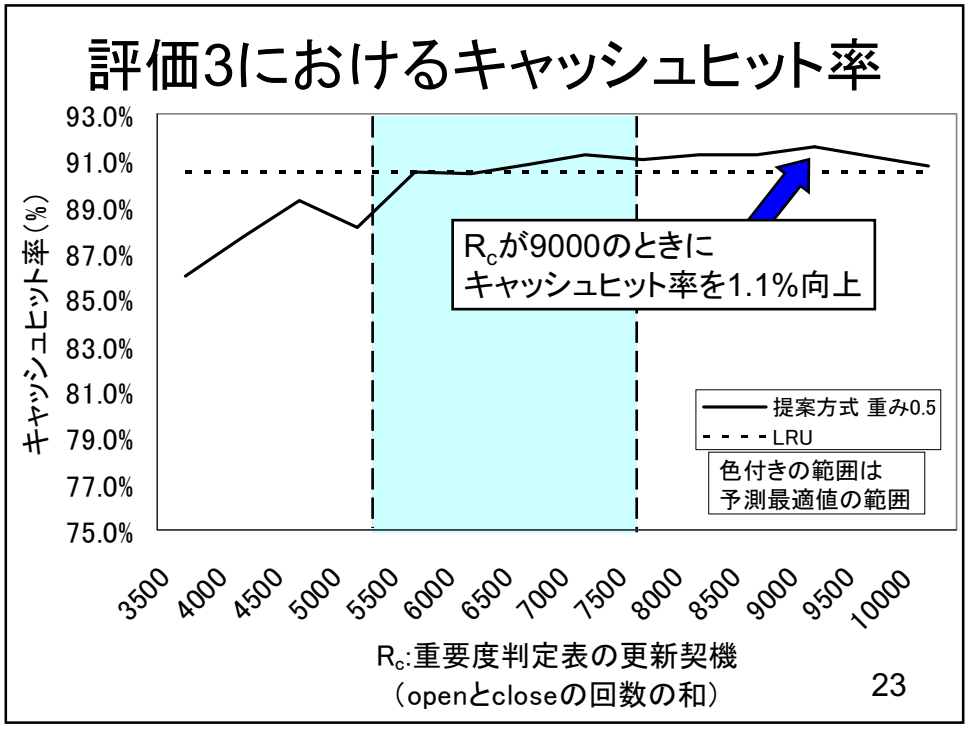
＜評価内容＞
カーネルのmake処理における性能を測定

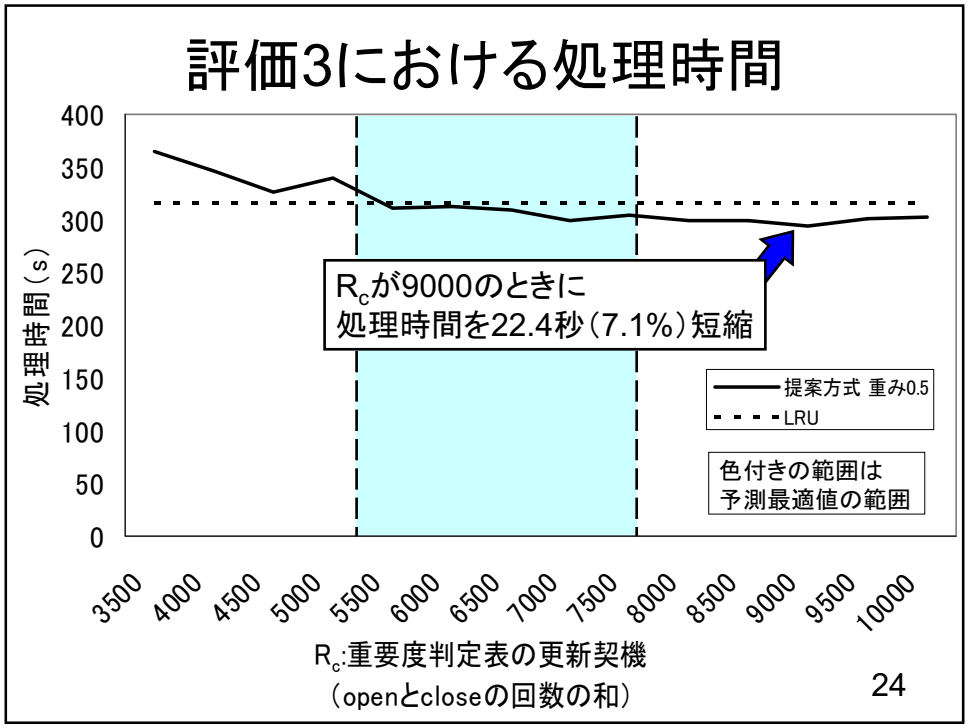
＜評価目的＞
(1) カーネルのmake処理の性能向上により，OS開発における作業時間を短縮可能
(2) 作業時間の短縮によるストレスの軽減

＜評価条件＞
(1) 重み: 0.5
(2) 重要ファイルの数：100（カーネルのmake処理において，重要度の上位から，バッファキャッシュの上限に達するまで）

評価2の結果より導き出した予測最適値は，5,200～7,800

22


評価3におけるキャッシュヒット率
93.0%
91.0%
89.0%
87.0%
85.0%
83.0%
81.0%
79.0%
77.0%
75.0%
キャッシュヒット率（%）
Rcが9000のときに
キャッシュヒット率を1.1%向上
提案方式 重み0.5
LRU
色付きの範囲は
予測最適値の範囲
3500
4000
4500
5000
5500
6000
6500
7000
7500
8000
8500
9000
9500
10000
Rc:重要度判定表の更新契機
（openとcloseの回数の和）
23


評価3における処理時間
400
350
300
250
200
150
100
50
0
処理時間（s）
Rcが9000のときに
処理時間を22.4秒（7.1%）短縮
提案方式 重み0.5
LRU
色付きの範囲は
予測最適値の範囲
3500
4000
4500
5000
5500
6000
6500
7000
7500
8000
8500
9000
9500
10000
Rc:重要度判定表の更新契機
（openとcloseの回数の和）
24

# 評価4

＜評価内容＞

rsyncで他計算機に/usr/src/以下のデータをバックアップしているときのカーネルのmake処理の性能を測定

バックアップ用計算機と測定計算機は，100MbpsのLANにより接続

＜バックアップ用計算機の構成＞

(1) OS: FreeBSD 4.3-RELEASE
(2) CPU: Celeron（2.8 GHz）
(3) メモリ: 768 MB

測定用計算機の環境は評価3までと等しく，本評価における閾値は評価3において処理時間を最短にした値とする

バックアップ処理中のカーネルのmake処理の性能

| 方式 | キャッシュヒット率（%） | 処理時間（s） |
|---|---|---|
| LRU | 87.6 | 427.2 |
| 提案手法 | 88.5 | 406.2 |

25

# おわりに

＜まとめ＞

(1) 提案手法では，サイズの大きいファイルの読み込みに対して，性能低下を抑制
(2) 提案手法では，キャッシュミスを伴うループ読み込みにおける性能低下を抑制
(3) カーネルのmake処理の処理時間を22.4秒（7.1%）短縮
(4) バックアップ中におけるカーネルのmake処理において，処理時間を21.0秒（4.9%）短縮

＜残された課題＞

(1) 閾値の自動学習化

26

# Cell/B.E. への MINIX 3 の 移植と SPE 管理方式の検討

野尻祐亮 †

† 東京農工大学 大学院（博士前期課程） 工学府

## 1 はじめに

近年は，サーバ向けや組込み向けを問わず，プロセッサのマルチコア化が進んでいる．マルチコアの形態のなかでも，Intel の Core 2 Duo / Quad のように，すべてのプロセッサコアが同じ仕様であるホモジニアス（対称型）マルチコアに対して，プロセッサダイ内に異なる仕様を持つプロセッサコアを配するヘテロジニアス（非対称型）マルチコアが注目されている．これは，汎用的なプロセッサコアを複数，搭載するよりも，目的の用途に特化したプロセッサコアを含めることで，高い性能を維持しつつコストや消費電力を下げることができ，効率が高いためである．しかし，ヘテロジニアスマルチコアプロセッサにおいては，ソフトウェアの開発が，単なるマルチコアでの並列化よりもさらに難しくなるという問題点がある．

- OS の機能（システムの制御，リソース管理，ユーザプログラムの管理）が 1 つのコアでしか実行されない．デバイスドライバも，特に工夫しない限り 1 つのコアでしか実行されない．
- プロセッサコアごとに異なる命令セットにしたがってプログラミングすることが負担である．
- 従来のほとんどの OS は，ヘテロジニアスマルチコアを扱うことを想定しておらず，各プロセッサコアの制御がアプリケーションに委ねられる．アプリケーションプログラマにとって負担である．

これらの問題点を解決するため，すべてのプロセッサコアを統合的に管理する新たな OS を実現する．その OS において，単なる並列化にとどまらない新規な点として，次の点を目指す．

- OS の機能の一部を，異なる命令セットを持つ別のプロセッサコアでも実行できるようにする．
- 上位層で実行するプログラムのために，プロセッサコアごとの命令セットの違いをできるだけ吸収する．

**Porting MINIX 3 to Cell/B.E. and Consideration of SPE Management Mechanism**
Yusuke NOJIRI†
†Graduate School of Engineering, Tokyo University of Agriculture and Technology

本研究においては，対象とするプロセッサを Cell Broadband Engine （以下 Cell/B.E.）とし，家庭用ゲーム機 PLAYSTATION 3 をターゲットプラットフォームとして開発を行う．

## 2 OS の設計方針

本研究においては， OS の設計としてマイクロカーネル方式を採用する．Linux はモノリシックカーネル方式であるため，本研究の目標に掲げた， OS の機能を分散させることが難しい．一方，マイクロカーネル方式では， OS の機能が小さなモジュールとして独立しており，次のような特徴を持つ．

- 各機能の互いの依存性が小さくなる．
- 各機能を並列に実行させることが容易である．

これらの特徴から， OS の機能をプロセッサコアにそれぞれ割り当てるモデルを考えると，マイクロカーネル方式はマルチコアプロセッサにふさわしい．

マイクロカーネル方式の OS を実現するに当たって，本研究においては MINIX をベースとすることにする．本稿では以降，MINIX をベースとして PLAYSTATION 3 向けに開発する OS を **PS3 MINIX** と称する．

## 3 MINIX 3 の Cell/B.E. への移植

本研究では，目標の OS を実現に MINIX を用いることにした．目標の OS の実現の段階としては，

1. MINIX 3 を Cell/B.E. の PPU 上で動作させる
2. PPU 上の MINIX から SPU も扱えるようにする

に分けられる．本章では，1. の MINIX を PPU で動作させることについて述べる．

本研究で移植する MINIX は，オリジナルの x86 版ではなく，PowerPC 版の **MinixPPC**[1] である．MinixPPC とは，Apple iBook G4 に移植したバージョンの MINIX である．PowerPC G4 ですでに動作しているとはいえ， Cell/B.E. の PPU で動作させるには少なからず変更が必要である．変更を行った主な点について次に述べていく．

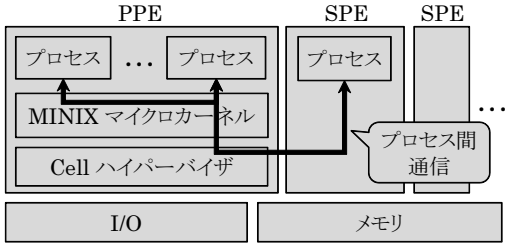



図 1: PPU/SPU プロセスとプロセス間通信

- **ハイパーバイザコールの利用**. PLAYSTATION 3 では， OS とハードウェアの間にハイパーバイザソフトウェアが介在している，ページテーブル操作，デバイスの利用，システム情報の取得など，ハイパーバイザを介さなければならない処理について，修正した.
- **PowerPC 64 ビットインプリメンテーションへの対応**. Cell Broadband Engine Architecture では，PPU 部分のアーキテクチャは 64 ビットインプリメンテーションの PowerPC であるが， MinixPPC は 32 ビット用に作られている．多くの部分は，PowerPC の 32 ビットモードの利用により，修正が不要となるが，なお 64 ビット対応が避けられない部分（例えば MMU 制御，コンテキスト保存・復帰）がある．そのため，64 ビットに対応するよう修正した.
- **デバイスドライバの作成**. PLAYSTATION 3 のハードウェアに対応したデバイスドライバが必要となる．フレームバッファドライバ，USB キーボードドライバ，内蔵ハードディスクドライバを作成した.

現時点の動作状況としては，ハードディスクまたは RAM ディスクからブートし，対話型のシェルが利用して，各種コマンドを実行できる．Cell の PPU への移植は一定の段階まで達している.

## 4 SPU 管理

本節では， PS3 MINIX における SPU の管理法について考える．Cell/B.E. におけるソフトウェア階層と，本節で考えるプロセスの管理法を図 1 に示す.

PPU では，通常のプロセッサと同様に複数のプロセスを動作させるのに対して，SPU には，1 コア当たり 1 プロセスを割り当てることにする．PPU, SPU で動作させるプロセスをそれぞれ PPU プロセス， SPU プロセスと呼ぶことにする.

複数のプロセッサコアを管理するためのデータ構造が必要である．各 SPU の状態を記憶し， SPU プロセスのディスパッチに利用できるようにする.

表 1: 各条件におけるループ速度

| OS | $\times 10^3$ ループ/秒 |
|---|---|
| OS なし | 57851 |
| Linux 2.6.23 | 57329 |
| PS3 MINIX | 57500 |

SPU ではプロセス間通信の部分を，他のプロセッサコアに対して働きかけるコードとする．具体的には，他のプロセッサとのメッセージデータの送受信，待機しているプロセスの起床，メッセージが到着するまでの自プロセスのブロックなどを行う．SPU においては，システムコールライブラリにもプロセス間通信コードを実装する.

プロセス同士の排他制御に関しては，現在の方式で特に問題はない．MINIX ではメッセージパッシングはランデブー方式で行っているからである．つまり，メッセージパッシング時にメッセージの送信側と受信側を同期させる.

SPU において外部割込みは，PPU からのメッセージ受信で受け取ることにする．これもまた MINIX の仕組みをそのまま利用したものである.

SPU を管理する機構に関しては，現在，詳細な設計を進めている．MINIX にヘテロジニアスプロセッサコアの管理機構を搭載し， OS サーバやデバイスドライバを SPU で実行できるようにする予定である.

## 5 評価

PPU 上において， OS のオーバヘッドがプロセスに与える影響を見積もるため，簡単なベンチマークを取る．各 OS: PS3 MINIX，Linux（Yellow Dog Linux 6.1，カーネル 2.6.23）において，C 言語で記述した空ループをユーザプロセスとして実行する．比較のため，OS のない状態でも同様のコードを実行する．それぞれの条件において，15 秒間のループ回数を計り，3 回の平均を算出する.

結果を表 1 に示す．他のプロセスやデバイスドライバの影響があるため厳密な比較はできないが，この結果からは，オーバヘッドの点において MINIX は Linux に大きく劣ることはないといえる.

## 参考文献

[1] Ingmar A. Alting: MinixPPC (2006). Master Thesis Computer Science.

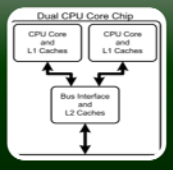
# Cell/B.E. への MINIX 3 の 移植とSPE 管理方式の検討

Summer Joint Symposium for Advanced System Software (JSASS) 2008
野尻 祐亮
東京農工大学
2008年9月1日



## 背景

### ↘近年のプロセッサ開発の潮流: マルチコア化




**ホモジニアスマルチコア**

- 同種のコアを統合
- 従来のソフトウェアとの互換性高


**ヘテロジニアスマルチコア**

- 異なるアーキテクチャのコアを統合
- 各コアを処理内容に応じて最適化(制御用、計算用など)→ 従来の性能限界を突破 / 高効率化

図: http://ja.wikipedia.org/wiki/%E7%94%BB%E5%83%8F:Dual_Core_Generic.png http://www.cellusersgroup.com/modules/product/toshiba-cell-chip-set.php

## ヘテロジニアスマルチコアでの課題

- OS やドライバが主のコアでしか動作せず
- 副のコアがアプリケーションからしか利用されず

**OS がヘテロジニアスマルチコアでの並列動作をサポートする必要**



## 研究概要

- ヘテロジニアスマルチコアプロセッサをサポートする OS の研究
  - OS の機能の一部を副のコアへ
    - ドライバ: フレームバッファ、データ暗号・復号化など
    - OS サーバ: ファイルシステム、プロトコルスタックなど
  - 副のコアの制御をアプリではなくOSで
- Cell/B.E. をターゲットとして MINIX 3 をベースにヘテロジニアス対応 OS を開発

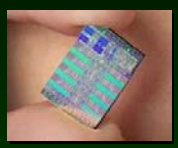
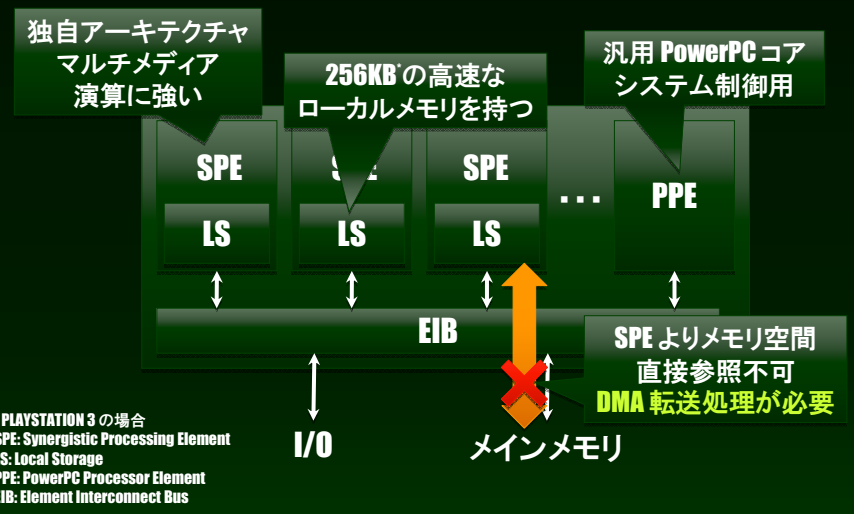



図: http://www.minix3.org/doc/Large_Logos.zip http://ja.wikipedia.org/wiki/%E7%94%BB%E5%83%8F:Cell-Processor.jpg

# Cell Broadband Engine




* PLAYSTATION 3 の場合
SPE: Synergistic Processing Element
LS: Local Storage
PPE: PowerPC Processor Element
EIB: Element Interconnect Bus



# 関連研究

- Cell broadband engine, SPE assisted user space device driver (町田, 2007)
  - Linuxドライバの一部の処理をユーザモードを経由しSPEで
  - ×設計の複雑さ、オーバヘッド
- タスクの動作特性に適合可能なヘテロジニアスマルチコアCPU向けOSの開発 (小林ら, 2007)
  - Linux上でSPE向けOSとしてSPE管理機構を実装
  - ×アプリケーションへの適用を想定、管理機構がユーザモード動作

# なぜ MINIX?

- マイクロカーネル設計: OS の機能を各コアに割り付けるモデルが成立しやすい
  - マイクロカーネル ... サーバやデバイスドライバがプロセスとして独立、プロセス間通信で互いに連携
  - モノリシックカーネル ... OS の機能が独立していない、すべてのデータをカーネル内で共有
- マイクロカーネルのヘテロジニアスマルチコアへの適用が未達成
  - Cell/B.E. では Linux の実装しかない



# PLAYSTATION 3 への移植

- MINIX 3 を PLAYSTAITION 3 へ移植
  - まず PPE での動作を目指す

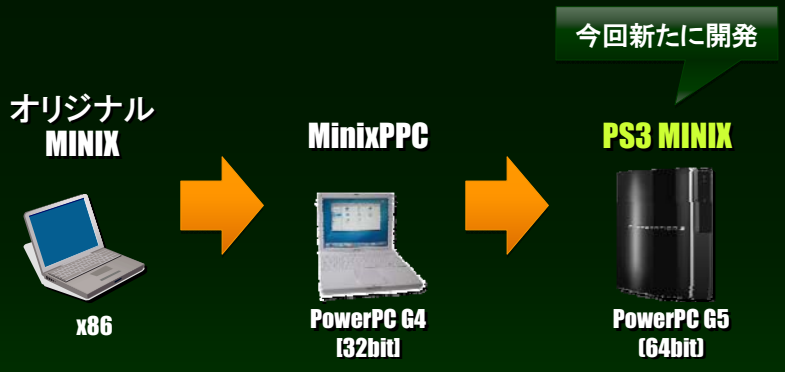



図: http://ja.wikipedia.org/wiki/%E7%94%BB%E5%83%8F:IBook_G4.jpg http://ja.wikipedia.org/wiki/%E7%94%BB%E5%83%8F:Playstation3vector.svg

# PS3 への移植における課題 (1)

- PowerPC 64ビット化
  - 32ビットモードを利用することで修正を最小限に
    - コンパイルを32ビットのままで
      → 変数の型の変化を考慮する必要なし
  - 修正が必要
    - MMU 管理
      - 実効アドレスは32ビットのままで互換性確保
      - リアルアドレスは64ビットまで扱えるよう拡張
    - プロセススイッチ、割込み
      - 64ビットモード→32ビットモードに切替え
      - CPU 状態の保存復元やメモリセグメントの設定を64ビット仕様に



# PS3 への移植における課題 (2)

- プラットフォーム仕様の違い
  - 画面出力、USB キーボード、HDD などのドライバが必要
- PS3 ハイパーバイザによる OS 仮想化
  - ページテーブル操作、デバイス利用、システム情報取得などにハイパーバイザコール使用の必要
- ブート方法
  - 複数の a.out ファイルからなるカーネル構成のため専用ブートローダが必要

# 開発のステップ

1. MINIX 3 を PPE で動作させる
   - ブートローダの作成
   - カーネルの動作検証・修正
   - 各種ドライバの作成
2. PPE 上の MINIX から SPE も扱えるようにする
   - SPE 管理機構、プロセス間通信の実装
   - ツール、ライブラリ類の整備
   - SPE 用プログラム作成



# 動作状況

- ハードディスク/RAM ディスクからブート
- USB キーボード使用可
- 対話型シェルで各種コマンドが動作

## 変更・追加箇所

- システムイメージローダ
  - kboot → システムイメージローダの2段ブート
- MMU 管理
  - ページテーブル、SLB の読書きを64ビット仕様に
- 画面表示
  - 左半分: tty、右半分: カーネルデバッグ出力
  - デバッグのほぼ唯一の頼みの綱
- 他 プロセススイッチ、割込み、クロック、メモリコピー、プロセス間通信など
  - 一部のデータ構造も64ビット化



## 簡単な評価

- Linux 上 / MINIX 上 / OS なしの状態における単位時間当たり空ループ回数の計測
  - 32bit ユーザプログラムとして実行 (Linux, MINIX の場合)
  - 3回試行、平均を算出

```
while (15秒間) {
    for {i = 0; i < 1000; i++} ;
    cnt++;
}
```

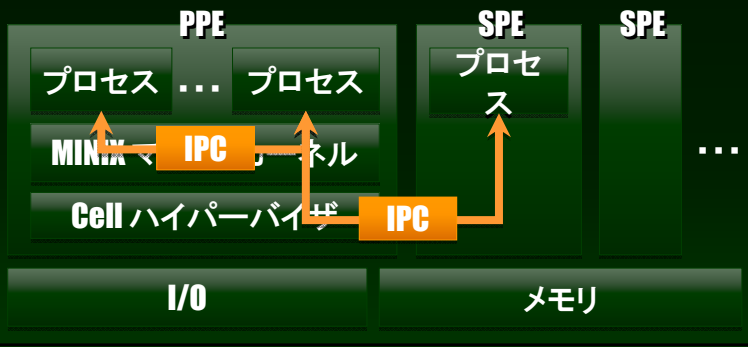
# 結果

- カーネルや他タスクなどによるオーバヘッドに遜色なし




# SPEの管理方針

- OS サーバやデバイスドライバ、アプリケーションをプロセス単位で SPE 上で動作させる
- PPE 上の MINIX IPC(プロセス間通信)
  → PPE – SPE 間の通信に置換え



## SPEを取り扱う上での課題

- SPE の割当て方法
  - 1つのプログラムに対し1コア固定割当て
- メッセージ交換
  - PPE に対して割込みをかけ、カーネルで実行
- リモートセグメント(カーネル、別プロセス、I/O)とのメモリ転送
  - ライブラリで極力ラッピング



## PPE/SPEでのプロセス

| PPE でのプロセス | | SPE でのプロセス |
|---|---|---|
| 共通 | 管理データ構造、PIDの割振りなど | 共通<br>SPE 状態データ、コア割当て情報も管理 |
| あり | プリエンプション | (とりあえず)なし |
| 直接発行 | システムコール | PPE への割込みとデータ転送で実現<br>ライブラリで隠蔽 |
| 直接扱う | 割込み | PPE の割込みハンドラ経由 |

## SPE のサポート

- SPE 用実行ファイル
  - ファイル形式: 既存の a.out 形式をベースに改造
  - 生成ツールの作成
- 各種ライブラリやカーネル API の SPE でのサポート
  - システムコール、メッセージ交換部分は PPE へ働きかけるコードに書換え
- 現況: SPE サポートに向けて詳細設計中



## まとめ

- ヘテロジニアスマルチコア向け OS の開発
  - OS サーバやドライバを副のコアで並列実行
  - マイクロカーネルの MINIX を応用
  - PPE でシェルが動作
- 今後の計画
  - SPE 管理機構の実装
  - ツールやライブラリの作成
  - 一部のプロセスを SPE へ移植
    - 例: フレームバッファドライバ、重いアプリ
  - 評価: プロセスをそれぞれ PPE と SPE で動作させたときの性能比較など

# センサネットワーク向けＯＳにおける処理時間の短縮を目的とした分散処理機構

李 冉
立命館大学大学院理工学研究科

## 1　はじめに

近年, 半導体技術や無線通信技術の発達により, 無線通信機能とセンシング機能を持ったセンサノードが開発されてきている. それに伴い, センサノードにより構成される無線センサネットワークへの関心が高まっている. センサネットワークは省電力性を実現するため, センサノードには低クロックの MPU を採用することが多い. そのため, センサノード上に実行待ちのタスクが多く存在する場合やデータの圧縮などの時間がかかる処理がある場合, 決まった時間内に処理を完了できない可能性がある. また, さまざまな環境情報を取得するため, 回収やメンテナンスが困難な場所にセンサノードを配置することが多くなり, このような場合はセンサノードのバッテリ駆動時間がセンサノードの寿命となる.

本研究では, 図 1 に示すように, センサネットワークをいくつかのグループに分割し, 各グループの 1 つをマスタノード, ほかのノードをスレーブノードとする. グループ全体を仮想的な 1 つのセンサノードとみなし, 多数のグループ化されたノードを用いてセンサネットワークを構成する. ほかのグループ (あるいは単体ノード) から受信できるのはマスタノードのみとする. マスタノードのバッテリ残量が少なくなれば, マスタノードのデータおよび環境情報をほかのスレーブノードにコピーし, そのノードを新たなマスタノードとする. それぞれのグループにあるセンサノードを用いて分散処理を行うことにより, タスクの合計処理時間を短縮することが可能となる. また, グループにあるすべてのノードが使用不可能になるまでシステムが動作するため, センサネットワーク全体の寿命を長くすることが可能となる.

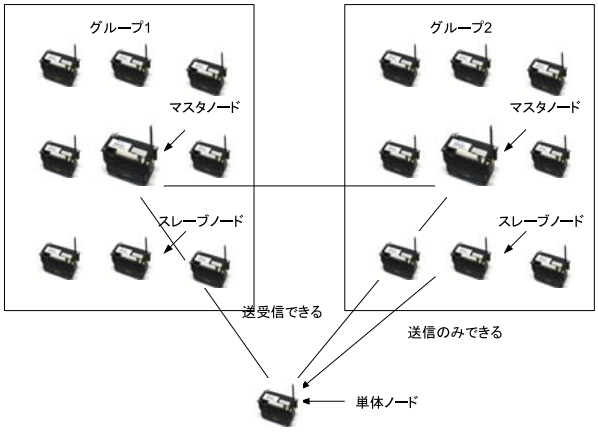



図 1　提案手法のイメージ図

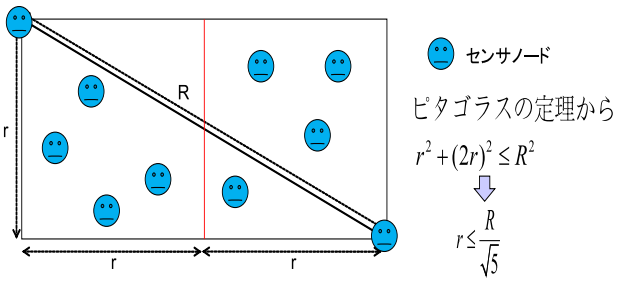



図 2　GAF におけるグループの大きさ

## 2　GAF

### 2.1　GAF の概要

本研究では, センサネットワークをグループ化する際, GAF(Geographical Adaptive Fidelity) [1] という手法を用いる. ここでは, まず, GAF について説明する. GAF はセンサネットワークをノードの位置に基づいて複数のグループに分割する. 各グループから予想稼動時間がもっとも長い 1 台のノードをマスタノードとして選出し, ほかのノードをスレーブノードとし, スリープさせる. スレーブノードは, マスタノードの予想稼働時間に応じた長さの時間だけをスリープした後起動し, 自身の予想稼動時間をグループにあるほかのノードにブロードキャストする. 各ノードは自身より予想稼動時間の長いノードの存在を知った場合, 自身をスリープさせる. これによって, グループ内のすべてのノードが使用不能になるまでシステムが動作し続ける. 本来の GAF では, スレーブノードを完全にスリープさせるが, 本手法では, スレーブノードを使用して並列処理を行うため, スレーブノードは無線デバイスのみ動作させる.

### 2.2　GAF におけるグループの大きさ

GAF では, 隣接グループ間での通信が必要となるため, グループの大きさは隣接グループ内のすべてのノードが互いに通信可能という条件を満たさなければならない. したがって, 図 2 が示すように, ノードの最大通信距離を R とすると, 隣接グループのもっとも遠い 2 点間の距離は R 以下である必要がある. したがって, 1 つのグループの最大面積は $R^2/5$ となる.

| ノード番号 | 電圧 | タスクの種類 | そのほか |
|---|---|---|---|
| 1 | 2.7 | データの送信 | |
| 2 | 3.0 | モニタリング | |
| 3 | 2.5 | データの圧縮 | |

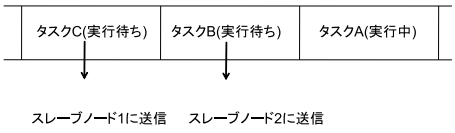



図 3　実行待ちのタスクの分散処理モデル

## 3　提案手法

本研究では, 次の 2 つの手法を用いてタスクの実行時間を短縮する.

1. 実行待ちのタスクを分散して処理する.

   マスタノード上にある実行待ちタスクの数が多く存在する場合, スレーブノードに転送して処理する.

2. 時間がかかる処理を分割して処理する.

   時間がかかる処理であれば, そのタスクを分割し, スレーブノードに処理させることによって全体の処理時間を低減する.

1 番の手法では, マスタノード上にノード状態管理テーブルを作成し, スレーブノードの電力残量などの状態を格納する. 実行待ちタスクを送信する際, 状態管理テーブルをチェックし, もっとも電力が残っているスレーブノードからタスクのデータを送信する. たとえば, 図 3 が示すように, 状態管理テーブルに 3 台のスレーブノードの情報が入っている場合, 実行待ちタスク B と C をそれぞれスレーブノード 2 と 1 に送信する.

2 番の手法は予想実行時間が分かるようなタスクに適している. この手法では, 次のような手順を踏むことにより, タスクの実行時間を短縮する.

- ユーザがアプリケーションを作成する際, あらかじめ分割タスクの種類を決める.
- 分割タスクが待ち状態になってもスレーブノードに送信しない.
- マスタノードの状態管理テーブルをチェックし, スレーブノードが所持するタスクの種類を確認する.
- タスクがもっているデータをスレーブノード用に分割する.
- 分割されたデータをスレーブノードに送信する.

## 4　おわりに

本稿では, 複数のセンサノードを 1 つのグループにまとめ, タスクの全体処理時間を短縮する手法について述べた. 本手法は, 実行待ちのタスクを分散して処理する方法と大きいタスクを分割して処理する方法を用いて処理時間を短縮する. 今後は, MANTIS OS [2] を用いてこれらの機能の実装と評価を完了させる予定である.

## 参考文献

[1] Ya Xu, et al:”Geograhy-informed Energy Conservation for Ad Hoc Routing,” in proc. of MobiCom’01, pp.70-84, 2001

[2] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, R. Han, “MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms,” ACM/Kluwer Mobile Networks & Applications, Special Issue on Wireless Sensor Networks, vol. 10, no. 4, August 2005.

# センサネットワーク向けＯＳにおける処理時間の短縮を目的とした分散処理機構

立命館大学大学院
理工学研究科
基本ソフトウェア研究室
Ｍ１　李　冉

1

## 発表内容

- 研究背景
- センサノードの概要
- センサノード向けOS の特徴
- 提案手法の概要
- 本研究の問題点
- おわりに

2

## 研究背景

- センサノード上に実行待ちのタスクが多く存在する場合
  - 決まった時間内に処理を完成できない可能性がある
  - より多くの電力が消費される
  - センサノードの寿命が短くなる
- 単一のセンサノードにおけるタスクの実行時間を短縮する必要がある
- 多くのセンサノードを投入し，タスクを分散処理することで処理時間を短縮することが可能

3

## センサノードの概要

- 環境情報を取得するセンサと無線通信できる小型のデバイスから構成される端末
- 複数のセンサノードからセンサネットワークを構成
- 特徴
  - 安価
  - 制約の厳しい計算機資源
  - 例：MICAz　(Crossbow)
    - 7.37MHzのCPU
    - 4kbのSRAM
  - 普通の電池で数ヶ月動作

4

## センサノード向けOS

- 小さいフットプリント
  - MANTIS OS (コロラド大) < 500B
  - Tiny OS (UCB) with Blink アプリケーション 683B
- 消費電力を抑えるスケジューリングを使用
  - イベント駆動型 (like TinyOS)
    - タスクは run to completion
  - プリエンプティブ・マルチスレッド型 (like MANTIS OS)
- 本研究はプリエンプティブ・マルチスレッド型を使用

5

## OSレベルで行う必要性

- ユーザが分散処理を意識せずにアプリケーションを作成することが可能
- システムコールを利用しないため, 余計なオーバーヘッドを抑止

6

## 提案手法の前提条件

- 複数のセンサノードをグループ化
  - GAFアルゴリズムを使用
- 1つのセンサノードをマスタノードとし，ほかのノードをスレーブノードとする
- マスタノードは常に動作
- ほかのグループから受信できるのはマスタノードのみとする

7

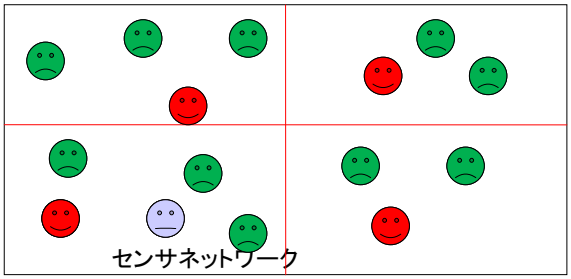
## 提案手法

- 実行待ちのタスクを分散して処理
  - マスタノード上にある実行待ちタスクの数が多く存在する場合，スレーブノードに転送して処理
- 時間がかかるタスクを分割して処理
  - 時間がかかる処理はタスクを分割しスレーブノードに処理させることで全体の処理時間を低減

8

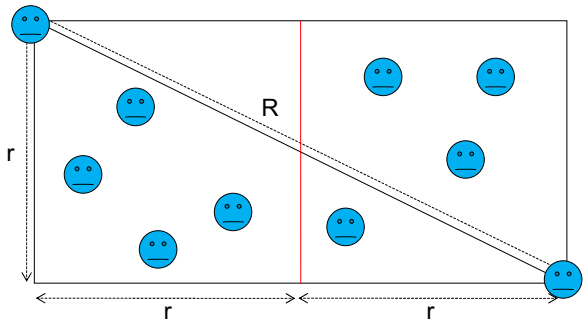
# GAFアルゴリズムの概要

- 前提条件：ノードの位置情報が既知
- センサネットワーク内のセンサノードを動的にグループ化することが可能
- ノードを高密度に配置することにより，システム全体の動作時間を延ばす
- 欠点：より多くのセンサノードが必要




# GAFにおけるグループ（セル）の選択方法

- センサネットワーク全体をいくつかのセルに分割
- セルの一辺の長さを r とする
- センサノードの最大通信半径を R とする
- セルの最大面積は$R^2/5$




ピタゴラスの定理から

$$r^2 + (2r)^2 \le R^2$$

$$\Downarrow$$

$$r \le \frac{R}{\sqrt{5}}$$

## GAFを用いる必要性

- グループを動的に選択することが可能
- グループ内にあるすべてのノードが使用不能になるまでシステムが動作する

**センサノードを最大限に利用**
**コストの削減が可能**

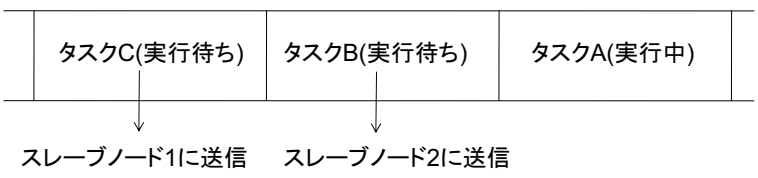
11

## 実行待ちのタスクの分散処理モデル

- マスタノードにスレーブノードの管理テーブルを作成
  - 指定周期でスレーブの状態を管理テーブルに格納

| ノード番号 | 電圧 | タスクの種類 | そのほか |
|---|---|---|---|
| 1 | 2.7 | データの送信 | |
| 2 | 3.0 | モニタリング | |
| 3 | 2.5 | データの圧縮 | |

- 寿命が長い順にスレーブノードにタスクを送信
  - 上記の場合タスクBをノード2に送信し，タスクCをノード1に送信




  - 同じタスクであれば，クエリで処理データのみを送信

12

## タスクの分割処理

- 予め分割タスクの種類を決める
- 分割タスクが待ち状態になってもスレーブノードに送信しない
- 状態管理テーブルをチェックし，スレーブノードが所持するタスクの種類を確認
  - 現在のタスクと異なる場合
  - ⇨ タスクのバイナリコードをスレーブノードに転送
- タスクがもっているデータをスレーブノード用に分割
- 分割されたデータをスレーブノードに送信

13

## 検討課題

- 実行待ちタスクが少ない場合，通信時間を含めたオーバーヘッドが発生する
- マスタノードの状態管理テーブルを定期的に更新必要があるため，オーバーヘッドが発生する

14

# おわりに

- 多数のセンサノードを用いたタスクの分散処理モデルについて述べた
  - センサネットワークをグループに分割
  - グループ内にあるノードを連携してタスクを処理
- 今後の予定
  - MANTIS OSを用いて実装を進め，システムの有用性を検証

15

# 情報漏洩防止のためのリムーバブルメディアのアクセス制御手法

岩永 真幸 †　　毛利 公一 ††
† 立命館大学大学院理工学研究科　†† 立命館大学情報理工学部

## 1　はじめに

近年，情報の漏洩事件が頻繁に発生している．現在発生している情報漏洩事件は，社員が自宅にデータを持ち帰り家の計算機から漏洩させたり，権限のあるユーザが外部に意図的に漏洩させるといった，内部の人間による情報漏洩が多くを占めている [2]．その中でもリムーバブルメディアを用いた情報漏洩が，特に問題となっている．しかし，このような情報漏洩は，内部の人間によって発生しているため，過剰な制御を行うと業務に支障を与えてしまう可能性がある．企業は，情報漏洩を防止するためのリムーバブルメディアを開発しており，リムーバブルメディア内のデータを暗号化したり，認証機能を搭載している．しかし，暗号化では利便性が低下し，業務に支障を与える．また，内部の人間による情報漏洩を防止することが困難である．このように，従来のセキュリティ技術は，内部の人間による漏洩を業務に支障を与えずに防止することが困難である．このような背景から，我々は，リムーバブルメディアを用いる状況や実際に発生しているリムーバブルメディアを用いた情報漏洩について調査し，通常の業務に影響を与えずに情報の漏洩を防止するシステムの開発を行っている．

## 2　情報漏洩のパターン

内部の人間によるリムーバブルメディアを経由した情報漏洩を防止するために，実際にどのような情報漏洩が発生しているのか考察を行った．その結果，情報漏洩事件は，ユーザが必要のない操作を行うことで情報漏洩が発生している場合が多く存在することがわかった．よって，必要のない操作による情報漏洩を防止する機能を実現することで，業務に支障を与えることなく情報漏洩を防止することが可能となる．以下に現在発生しているリムーバブルメディアを用いた情報漏洩パターンを述べる．

### 2.1　利用する必要のないユーザによる漏洩

実際に発生している情報漏洩は，そもそもデータを扱う必要のないユーザが，リムーバブルメディアにデータをコピーして発生している場合が多い．内部の人間であるが，そのデータを利用する必要がない悪意あるユーザが，外部に情報を漏洩させるためにリムーバブルメディアを用いている．

### 2.2　必要のない操作による漏洩

前節と違い，データを利用する必要のあるユーザが情報を漏洩させる場合も存在する．しかし，その場合もデータを利用する必要があるユーザであるが，業務上必要のない操作を行っている場合が多い．この場合，悪意のある場合とない場合が存在する．悪意があるユーザは，データの読込みや書込みを行う必要があるが，持ち出してはいけない重要なデータをリムーバブルメディアに書き込んで漏洩させる．悪意のないユーザは，自宅で作業を行うためにリムーバブルメディアにデータをコピーして，重要なデータを持出す．その後，自宅の計算機からウイルスなどによって漏洩させてしまったり，リムーバブルメディアの紛失によって，情報漏洩が発生する．

### 2.3　時間外の利用

持ち出してはいけない重要なデータを持ち出す場合，人が少ない時間に行われる場合が多い．この時間は，殆どの場合が業務時間外である．

## 3　提案手法

### 3.1　OS による制御

リムーバブルメディアへのアクセスは，さまざまなアプリケーションで行われる．アプリケーションの中には，セキュリティの低いものも存在する．しかし，重要なデータはどのようなアプリケーションを用いたとしても漏洩を防止する必要がある．そこで，データの保護を OS で行う．すべてのアプリケーションは，デバイスにアクセスするためには，OS を利用する．よって，OS 内で適切な制御を行えば，ユーザがどのようなアプリケーションを利用しても情報が漏洩することはない．

### 3.2　ファイル単位のアクセス制御

リムーバブルメディアへのアクセスを制御する際に，無差別に制御を行うと全てのデータがリムーバブルメディアを利用できなくなるため，業務へ影響を与えてしまう．また，データごとに制御を行えるような機能が必要となる．また，データごとに利用者や許可される操作が異なる．よって，データごとに適切な制御を決定できるような機能も必要である．そこで，ファイルごとに保護ポリシを設定可能とすることで，ファイル単位でアクセス制御を行う手法を提案する．

### 3.3　アクセス制御の実現方式

リムーバブルメディアへのアクセスは，大きく分けて書込みと読込みが存在する．実際に発生しているリムーバブルメディアを用いた情報漏洩事件は，ほとんどがそれへのデータ書込みによって発生している．そのため，リムーバブルメディアへの読込みを制御すると，リムーバブルメディアを用いて計算機へデータを持ち込めなくなる．よって，リムーバブルメディアへのアクセス制御は，書込みのみでよいと考える．

### 3.4　コンテキストの利用

2.1 節と 2.3 節で述べたユーザや時間による情報漏洩を防止するために，コンテキストとしてユーザと時間を利用する．保護ポリシに，ユーザや時間によってリムー

バブルメディアへの書込みの可否を記述可能とする．これにより，コンテキストによって，書込みの制御を行うことが可能となる．このコンテキストを，write システムコールの中で参照し，保護ポリシと比較する．このような仕組みで，コンテキストを利用したアクセス制御を実現する．

## 4 実装方式

以上のような手法を実現するために，プライバシアウェア OS *Salvia*[1] を利用する．*Salvia* を利用することにり 3 章のファイル単位の制御，OS による制御，コンテキストの利用が実現されている．しかし，*Salvia* はリムーバブルメディアの制御を行っていないため機能を追加する必要がある．現在は，システムコールを read, write，send_local，send_remote システムコールに分類している．本手法では，リムーバブルメディアへの書込みを制御するため，write システムコールの制御を行う．しかし，書込み先のデバイスの確認は，現在の *Salvia* では行っていない．そこで，write システムコールの中で書込み先がリムーバブルメディアであるかを識別する必要がある．このとき，write システムコールの中だけで識別することは困難である．そこで，write システムコールの中で書込み先の i ノードから，書込み先のメジャー番号とマイナー番号を取得することが可能であることを利用する．OS の中で現在接続されているリムーバブルメディアのマイナー番号とメジャー番号を管理し，write システムコールの中で書込み先のメジャー番号とマイナー番号を比較することによって書込み先がリムーバブルメディアであるかを確認する．リムーバブルメディアの情報の管理手法は，リムーバブルメディアの種類によって区別する必要がある．

- 静的デバイス

  起動時から接続されているデバイスを静的デバイスと呼ぶ．静的デバイスの記憶装置は，HDD，CDD や FDD などである．これらは，固定されたメジャー番号が与えられ，メジャー番号からどの記憶デバイスであるかを認識することが可能である．

- 動的デバイス

  起動後も自由に，接続と切り離しが可能なデバイスを動的デバイスと呼ぶ．動的デバイスの記憶装置は，メジャー番号やマイナー番号から識別することができず，接続時に動的に番号が割り振られる．そのため，接続されるたびに，接続されたデバイスが記憶装置であるかを確認する必要がある．確認の結果，記憶装置である場合は，デバイスのメジャー番号とマイナー番号を取得する．デバイスの情報の取得は，適切なタイミングで行わなければいけない．デバイスが利用可能になる時点より遅れて取得したり，情報の取得洩れが発生すると保護すべきデータが書き込まれてしまう．よって，リムーバブルメディアが接続された時に実行されるプログラムである Probe 関数の中で取得する．また，デバイスが切り離された時に実行される Remove 関数の中でデバイス情報を削除する．

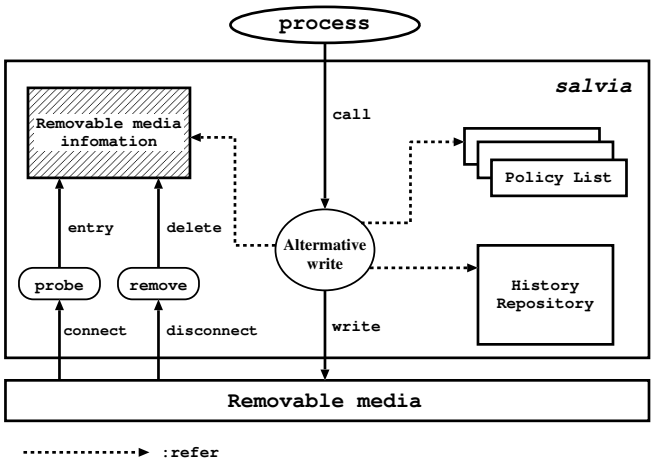



図 1 リムーバブルメディアへのアクセスを検出するための機構

このような仕組により現在接続中のリムーバブルメディアの情報を管理することが可能となり，漏れなく制御することが可能となる．このような機能を実現するためのシステムの構成を図 1 に示す．

## 5 おわりに

本論文では，リムーバブルメディアを経由した情報漏洩の防止について述べた．リムーバブルメディアによる情報漏洩は頻繁に発生しており，問題となっている．しかし，リムーバブルメディアを用いた情報漏洩は，権限を持つユーザが起こすため防止することが困難である．そのような背景から，リムーバブルメディアを用いる状況を考察し，通常の業務に影響を与えずに情報の漏洩を防止する手法を提案した．

## 参考文献

[1] 鈴来和久，一柳淑美，毛利公一，大久保英嗣: Privacy-Aware OS *Salvia* におけるデータアクセス時のコンテキストに基づく適応的データ保護方式，情報処理学会論文誌: コンピューティングシステム，Vol.47，SIG 3（ACS 13），pp.1 - 15.

[2] 独立行政法人国民生活センター: 個人情報流出事故に関する事業者調査結果，http://www.kokusen.go.jp/news/data/n-20050325 1.html

# 情報漏洩を防止するための リムーバブルメディアの アクセス制御手法

立命館大学院 2回生
毛利研究室
岩永 真幸

## もくじ

- 背景
- 研究の目的
  - リムーバブルメディアの利便性を損なうことなく、リムーバブルメディアを経由した情報漏洩を防止する
- 実際に発生しているリムーバブルメディアを用いた情報漏洩
- 提案手法
- 実装
- おわりに

2

# 背景

- リムーバブルメディアを用いた情報漏洩が多く発生している
  - コンパクトであるため持ち出しが容易
  - 数キロバイトのデータを持ち出せる
  - 内部の人間が漏洩させることが多い
- リムーバブルメディアの利用を禁止したり、過剰な制御を行うと利便性が失われる
  - 必要なアクセスは許可する
  - 情報漏洩を発生させるようなアクセスを禁止する

3

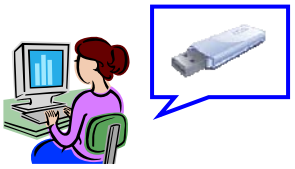
# 関連技術

- 認証型
  - リムーバブルメディアに対してアクセスするには、認証が必要
- 暗号型
  - リムーバブルメディア内のデータが暗号化される
- サーバ型
  - データがサーバに存在する
- コピー禁止型
  - HDD内にデータがコピーされない

4

# 研究目的

- リムーバブルメディアの利便性を損なうことなく、リムーバブルメディアを経由した情報漏洩を防止する
- リムーバブルメディアを用いた情報漏洩を考察
  - 情報漏洩は、不適切なアクセスによって発生しているものが多数を占めている
- 適切な状況でのアクセスは許可し、不適切な状況でのアクセスを禁止する



5

# 実際に発生している情報漏洩

- アクセスする必要のないユーザによる漏洩
  - 内部の人間であるが、そのデータを利用する必要がないユーザによる持ち出し

  ➡どのユーザもアクセスが許可されている
- 持ち出してはいけないデータの持ち出し
  - 悪意あるユーザが、持ち出して漏洩させる
  - 自宅で作業するために持ち出し、盗難や紛失、自宅計算機のウイルスによって漏洩させる

  ➡アクセスが必要だからといって、リムーバブルメディアへのアクセスも許可されている
- 時間外のアクセス
  - 深夜にデータを持ち出し

  ➡業務時間外にアクセスが許可されている

6

# 提案手法

- 書込みアクセス制御
  - データの漏洩は、書込みによって発生する
- HDDとリムーバブルメディアへのアクセスの区別
  - セキュリティの観点からHDDへの書込みと
    リムーバブルメディアへのアクセスは異なる
- OSによる制御
  - どのような書込みによる漏洩も防止する
  - アプリケーションに依存しない
- ファイル単位のデータ保護
  - 重要なデータとそうでないデータを区別するために、
    ファイルごとに保護を行う
- コンテキストの利用
  - ユーザと時間をコンテキストとして利用する

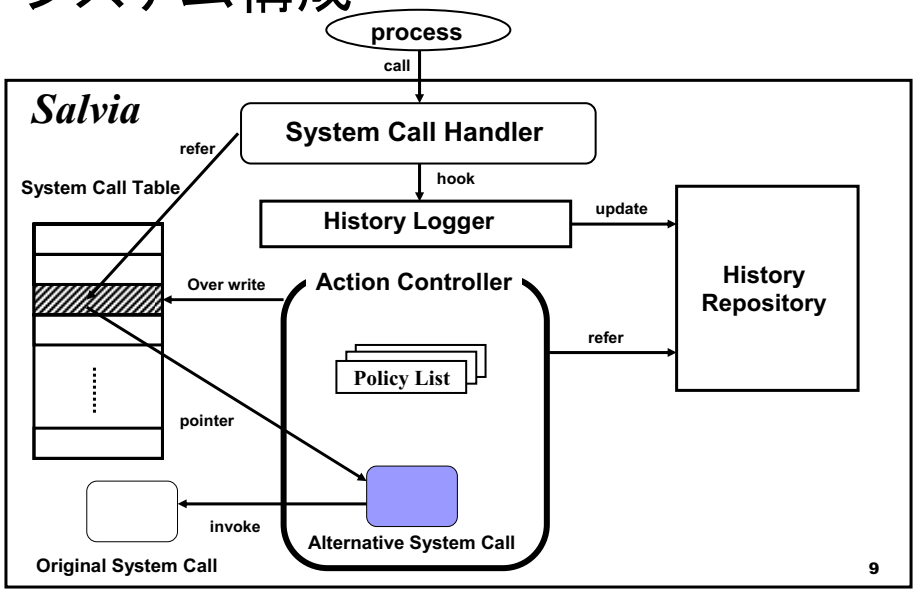
7

# Salviaの概要

- OSレベルのデータ保護
  - すべてのアプリケーションに対して統一的な保護を実現
- システムコールをデータの伝搬範囲に着目して分類し、アクセス制御を行う
  - ファイルに対する読出し(read)
  - ファイルに対する書込み(write)
  - 同一計算機上でのプロセス間通信(send_local)
    - パイプ、共有メモリ、ソケットなど
  - 他の計算機への送信(send_remote)
- コンテキストの利用
  - ユーザや計算機の状況に応じた保護を実現
    - ユーザ、時刻、位置、IPアドレスなど
- ファイル単位のデータ保護
  - ファイルごとにデータ保護ポリシを設定
  - データ提供者の意思を反映したデータ保護を実現

8

# システム構成




9

# 保護ポリシ

```
<default_access>
   <read>deny</read>
   <write>
      <write_access update="deny">deny</write_access>
   </write>
</default_access>
```

```
<ACL>
  <context>
     <user>
        <user_id type="real">1001</user_id>
     </user>
  </context>
  <access>
     <read> allow</read>
   </access>
 </ACL>
```

10

# 実装する機能

1. 書き込み先の判定
   - 書き込み先がリムーバブルメディアかを判定する
2. 保護ポリシとコンテキストの比較
   - 端末コンテキストの取得
     - ユーザコンテキストと時間コンテキストは、すでに利用可能
   - 保護ポリシの拡張
3. システムコールの可否判定

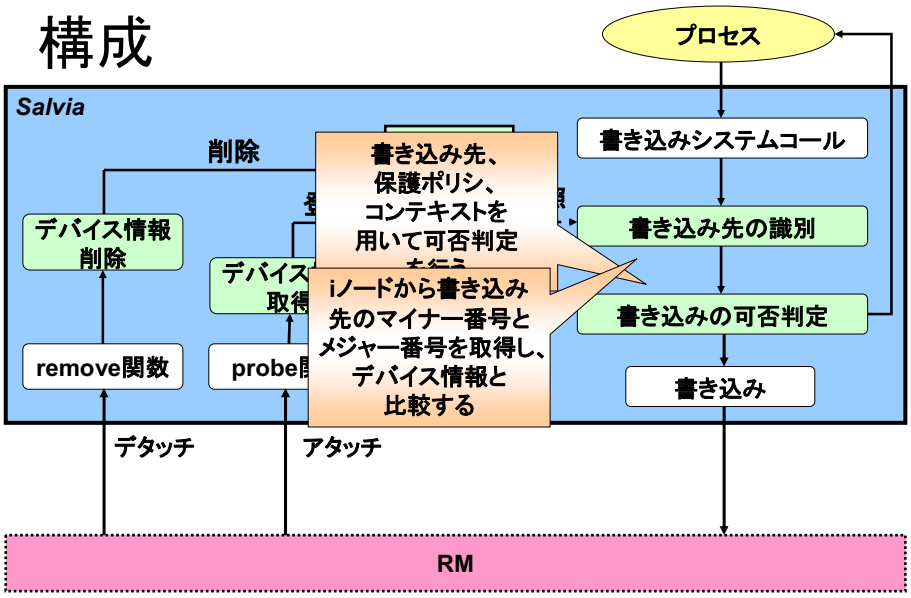
11

# 書き込み先の判定

- writeシステムコールの中だけで書き込み先を認識することは困難である
- 書き込み先のiノードから、書き込むデバイスのメジャー番号とマイナー番号を取得できることを利用する
- メジャー番号で特定できるデバイス
  - メジャー番号を確認することで書き込み先を特定する
- 特定できないデバイス
  - 現在接続中のリムーバブルメディアのメジャー番号とマイナー番号を管理することで書き込み先を判定する

12

# 構成




13

# 拡張保護ポリシ

```
<default_access>
   <write>
      <write_access update="allow">allow</write_access>
      <removable>deny</removable>
   </write>
</default_access>
```

```
<ACL>
   <context>
      <user>
         <user_id type="real">1005</user_id>
      </user>
   </context>
   <access>
      <write>
         <removable>allow</removable>
      </write>
   </access>
</ACL>
```

14

# 今後の予定

- プロトタイプの検証
  - 接続したリムーバブルメディアの情報が登録されているか
  - リムーバブルメディアへの書込みが制御されるか
  - HDDへの書込みは許可されるか
- 今後の課題
  - コンテキストに対応させる
  - writeシステムコールのオーバヘッドの計測

15

# おわりに

- 背景と研究の目的
  - リムーバブルメディアの利便性を損なうことなく、リムーバブルメディアを経由した情報漏洩を防止する
- 実際に発生しているリムーバブルメディアを用いた情報漏洩事件についての考察
- 手法と設計
  - 書込みアクセス制御
    - HDDとリムーバブルメディアの区別
  - OSによる制御
  - コンテキストの利用
  - ファイル単位の保護

16

# ヒステリシス署名によるログファイル改竄検出機構を備えたファイルシステムの提案

鶴田浩史[†]　齋藤彰一[†]　松尾啓志[†]

# Proposal of File System with Falsification Detection of Log File by Hysteresis Signature

Koji Tsuruta,[†] Shoichi Saito[†] and Hiroshi Matsuo [†]

## 1. はじめに

コンピュータに関する犯罪などによる訴訟が増えている．訴訟に対して，操作記録やログなどのデジタルデータが証拠として扱われる．しかし，デジタルデータは加工が容易である．そのため，不正を行った痕跡を消去することも容易になるので，証拠性が失われる．そこで，デジタルデータの証拠性を確保する技術であるデジタルフォレンジックが必要になる．デジタルフォレンジックは，法的な証拠性を明らかにする手順や技術のことである．証拠性を明らかにするために，証拠となるデジタルデータをハードディスクなどから捜し出す必要がある．

本稿では，デジタルフォレンジックによるログファイル改竄を検出する手法を提案する．

## 2. 提案手法

本稿では，ログファイルの証拠性を保証するために，署名を用いたファイルシステムを提案する．ログファイルは，不正が行われたことを追跡するための重要な記録となる．ログファイルは，管理者が管理するが，管理者がログファイルを改竄しない保証はない．そのため，ログファイルが改竄されないシステムが必要になる．しかし，ログファイルが記録されているディスクを持ち出して書き換えられた場合，ログファイルの正当性を証明することができない．そのため，ログファイルの正当性を証明するシステムが必要になる．このようにすることで，ログファイルの改竄防止やログファイルの改竄検出が行えるようになる．

### 2.1 追記のみのファイルシステム

ログファイルが改竄されないシステムとして，ログファイルを追記のみにするファイルシステムを提案する．ファイルシステムを用いるのは，複数のファイルを一括して扱うことや実装が容易であるためである．また，ログファイルを追記のみにすることによって，ログファイルを改竄されなくなる．しかし，ディスクを持ち出した場合，ログファイルが改竄されるという問題がある．そのため，ログファイルが改竄されていないことを証明するシステムが必要になる．

### 2.2 署名システム

ログファイルが改竄されていないことを証明するシステムとして署名システムを提案する．ログファイルに電子署名を用いることで，改竄を検出することが可能になる．しかし，ログファイルは長期間に渡って保管される場合が多いので，長期間署名を保証できるヒステリシス署名を用いる．

ヒステリシス署名は，過去の署名を連鎖構造で構築し，これを用いて連鎖構造のある新たな署名を作成する方法である．

### 2.3 署名を用いた追記のみのファイルシステム

先に述べた 2 つの提案を組み合わせて，署名を用いた追記のみのファイルシステムを提案する．このシステムの流れを述べる．

**(1)** コンピュータを操作した記録をログファイルに追記する．

**(2)** 追記したログファイルをハッシュ関数にかけ，ハッシュ値を作成する．

† 名古屋工業大学
Nagoya Institute of Technology

**(3)** 作成したハッシュ値を耐タンパ領域に転送し，そこでヒステリシス署名を行い，署名を作成する．

**(4)** 作成した最新の署名を耐タンパ領域に保管し，署名ファイルに署名を追記する．

ログファイルに追記する機能，ハッシュ関数，署名ファイルに追記する機能はファイルシステムで行い，署名を行う機能と最新の署名を保管する機能は，耐タンパ領域で行う．

ログファイルに追記するには，アクセス箇所が，ファイルの末尾以外を指している場合，書き込みは不許可にし，ファイルの末尾を指している場合，書き込みは許可する．このようにすることで，ログファイルが改竄されなくすることができる．また，読み込みに対しては，アクセス箇所がファイルのどこを指していても許可する．

署名を作成するには，ログファイルをハッシュ関数にかけ，ハッシュ値を作成する．その後，秘密鍵で暗号化を行い，署名を作成する．前回の署名データのハッシュ値，ハッシュ値，署名を署名データとして，署名ファイルに追記する．また，新しく署名を作成するには，先に追記した署名データをハッシュ関数にかけ，ハッシュ値を作成する．その後，先に追記した署名データの中からハッシュ値を取り出し，2 つのハッシュ値を結合する．そして，秘密鍵で暗号化を行い，署名を作成する．その後，先に追記した署名データのハッシュ値，今回作成したハッシュ値，署名を署名データとして署名ファイルに追記する．このような手順で署名を作成する．

### 2.4 署 名 確 認

以上のようなシステムで作成した署名を確認する手順を述べる．

**(1)** ログファイルの正当性の確認

ログファイルが改竄されていないことを確認する．これには，確認するログファイルに対応する署名を確認することで正当性を確認することができる．

**(2)** 署名データの正当性の確認

ログファイルに対応する署名データを確認する．署名データの前回の署名データのハッシュ値とハッシュ値を結合した値を作成する．また，署名データの署名を復号する．そして，結合した値と復号した値を比較することで署名データの正当性を確認することができる．

**(3)** 署名データの相関関係の確認

署名データの時系列に沿った相関関係を確認する．ログファイル K に対応する署名データのログファイル K-1 のハッシュ値を取り出す．また，ログファイル K-1 の署名データをハッシュ関数にかけハッシュ値を作成する．この 2 つの値を比較することで，署名データが相関関係を持っているか確認することができる．

このような手順で確認を行い，すべての手順で確認することができる場合，ログファイルの正当性，署名データの正当性を確認することができる．また，どれかの手順で確認することができない場合，ログファイルか署名データが改竄されたことを検出する．

**表 1** 時間測定結果

| | |
|---|---|
| 最長 | 20.2211 秒 |
| 最短 | 17.0899 秒 |
| 平均 | 18.6730 秒 |

## 3. 実装と実装に対する検討

このシステムを FUSE ファイルシステムにプロトタイプとして実装した．FUSE は，ユーザ空間で実装できるファイルシステムで，システムコールに対応した関数を作成することでファイルシステムを構築することができる．

プロトタイプを作成するにあたって，署名に対する時間とその時間に対する考察として検討を行った．

1GB のログファイルの署名に要する時間を測定した．20 回試行し，測定した結果は，表 1 のようになった．結果を見て分かるように最短でも 17 秒前後要するため，システムが破綻する可能性がある．そのため，ログファイルの大きさを 10MB で分割することで署名に要する時間を短縮することができると考えられる．また，分割したログファイルは書名データにより相関関係を持たせる．

## 4. まとめと今後の課題

ヒステリシス署名を用いたログファイル改竄検出機構を備えたファイルシステムを提案した．署名を行うことで，ログファイルの改竄を検出することが可能になる．また，署名データを確認することで，ログファイルの正当性と署名データの正当性を確認することができた．

今後の課題として，プロトタイプを作成することと，ログファイルがリネームしたときの挙動を考察することを予定している．

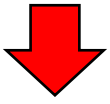
# ヒステリシス署名による ログファイル改竄検出機構を備えた ファイルシステムの提案

名古屋工業大学
鶴田　浩史

## 背景



- コンピュータに関する犯罪などによる訴訟が増加
  - 不正アクセス
  - 個人情報や機密情報の漏洩
- 証拠となるデータが必要
  - 操作記録、ログ
- デジタルデータは加工が容易



- デジタルデータの証拠性を確保する技術が必要
  - デジタルフォレンジックにより証拠性を確保

## デジタルフォレンジック

3

- 法的な証拠性を明らかにする手順・技術
  - データが捏造されたものかどうかを検証する技術
  - データの同一性を保全する技術
- 作業の痕跡や履歴を捜査・発見
  - ハードディスクから証拠となるファイルを探す
  - ログファイルからアクセス記録を割り出す
  - ディスクを復元する

## ログファイルの重要性

4

- 不正を追跡するための重要な記録
- 法的な証拠となる
- 管理者がログファイルを管理
- 問題点
  - 管理者がログファイルを改竄する危険性がある

提案1

ログファイルの改竄がされない
システムを提案

## 追記のみのファイルシステム(提案1)

5

**追記のみのログファイル**

- カーネルレベル
- ファイルシステムの利用
  - 複数のファイルを一括して取り扱う

➢改竄されない

- 問題点
  - ディスクが持ち出されると改竄される危険性がある

提案2

ログファイルの改竄が
されていないことを証明するシステムを提案

## 署名システム(提案2)

6

**ヒステリシス署名**

- ログファイルを署名
- 長期間保証

➢ログファイルの改竄を検出

提案手法

ヒステリシス署名を用いた
追記のみのファイルシステムの提案

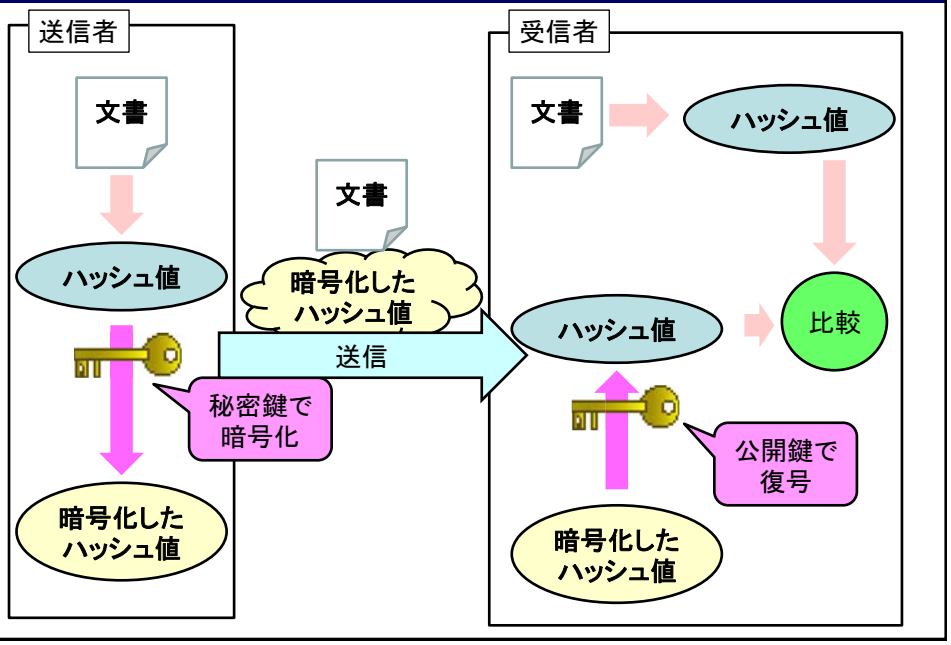
## デジタル署名

7

- デジタル文書の正当性を保証するための署名
  - 電子署名を実現するための方式の一つ
  - 公開鍵暗号方式を応用
  - 文書の送信者の証明
  - 文書が改竄されていないことを保証

## デジタル署名の流れ

8



## ヒステリシス署名

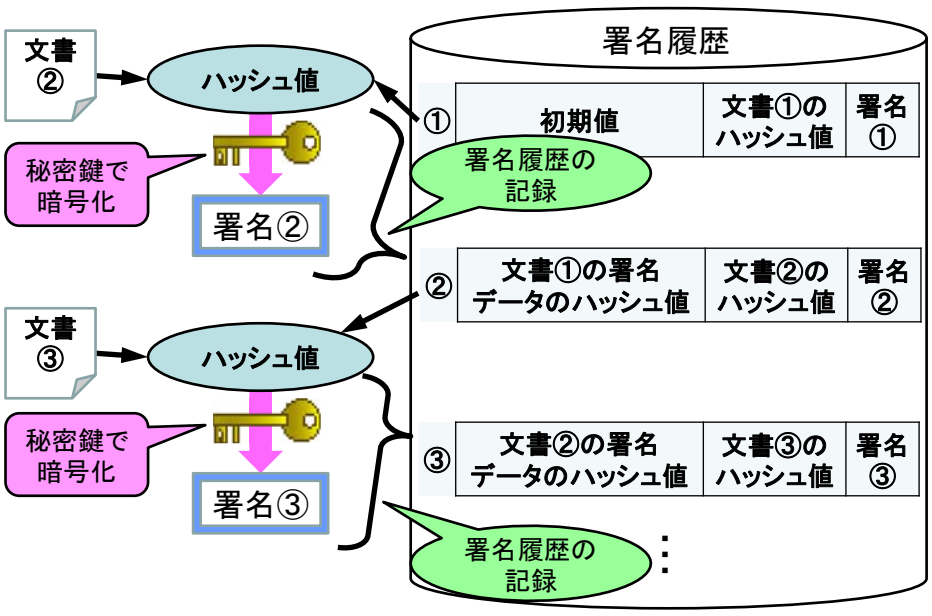
9

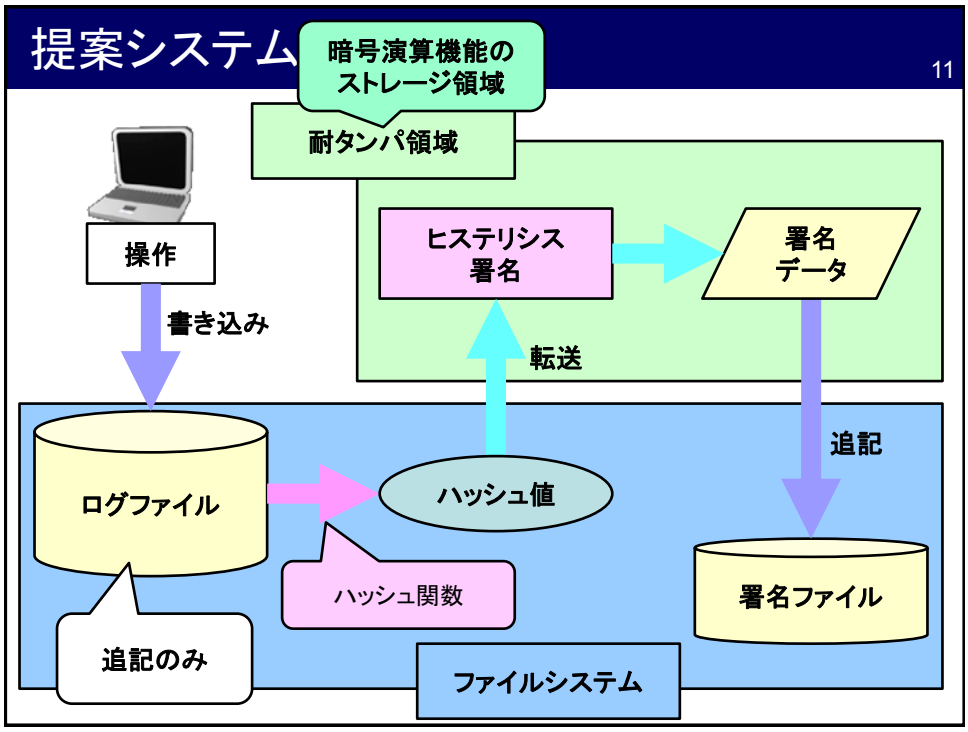
- ヒステリシス署名とは?
  - 電子署名技術の1つ
  - 過去の署名データの連鎖構造の構築
  - 連鎖構造のある新たな署名データを作り出す

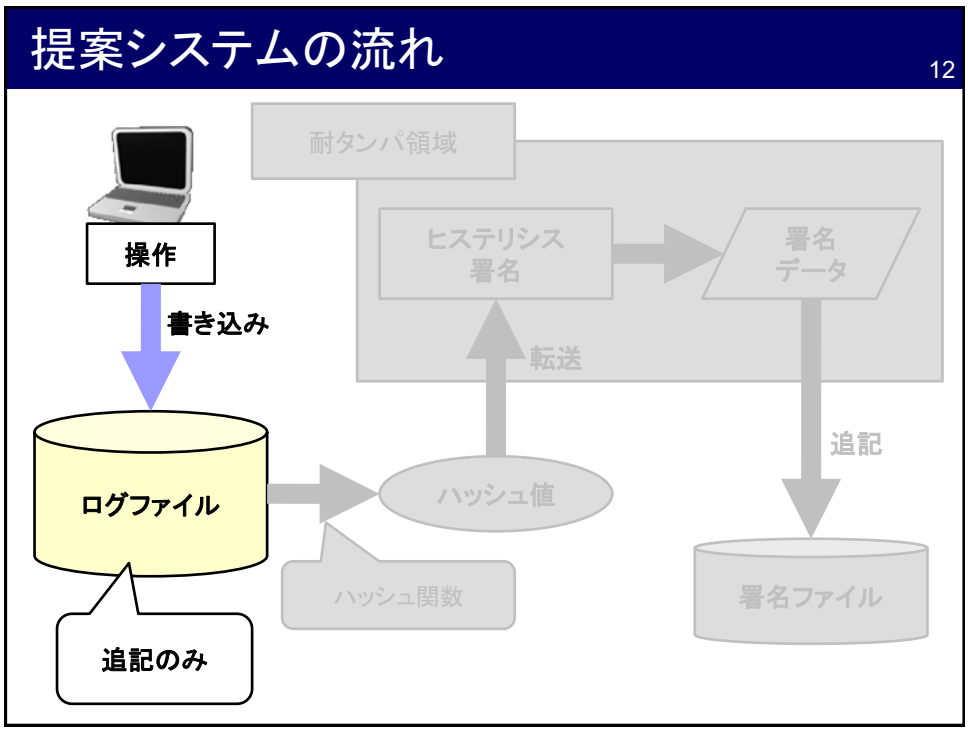
- 単独に電子署名を行うよりも安全性、有効性を長期間維持可能

## ヒステリシス署名の流れ

10




提案システム
11
暗号演算機能の
ストレージ領域
耐タンパ領域
操作
書き込み
ヒステリシス
署名
署名
データ
転送
追記
ログファイル
ハッシュ値
ハッシュ関数
署名ファイル
追記のみ
ファイルシステム


提案システムの流れ
12
耐タンパ領域
操作
書き込み
ヒステリシス
署名
署名
データ
転送
追記
ログファイル
ハッシュ値
ハッシュ関数
署名ファイル
追記のみ

## 追記手法

13

**ログファイル**

Aug 22 19:09:52 osorezan sshd[2241]: Received signal 15; terminating.

Aug 25 10:35:38 osorezan sshd[2151]: Server listening on :: port 22.

Aug 25 10:35:38 osorezan sshd[2151]: error: Bind to port 22 o
failed: Address already in use.

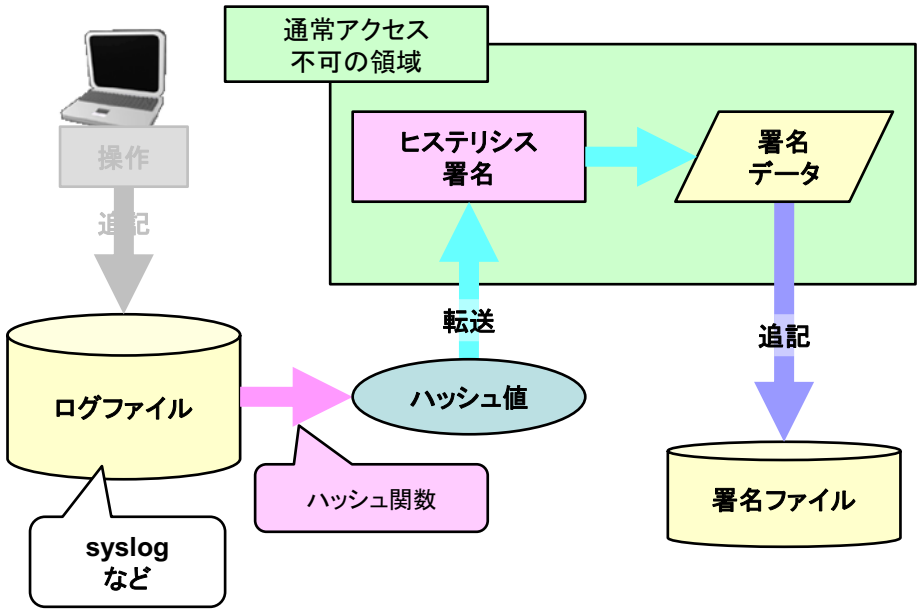
不許可

許可

- 書き込み
  - アクセス箇所がファイル末尾を指すとき***許可***
- 読み込み
  - アクセス箇所がどこを指していても許可

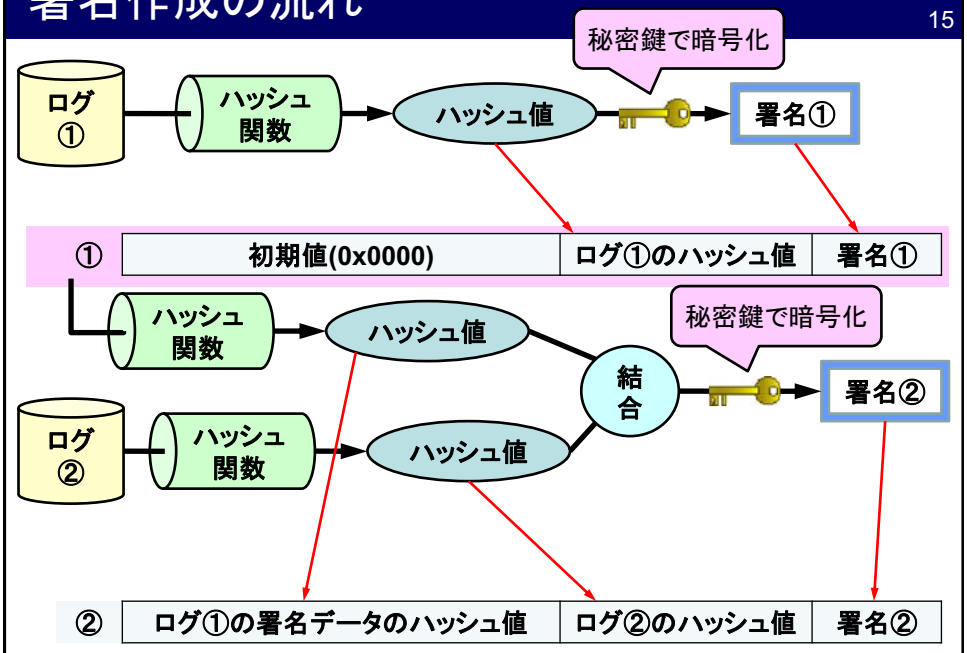
## 署名の流れ

14



## 署名作成の流れ

15




## 署名確認

16

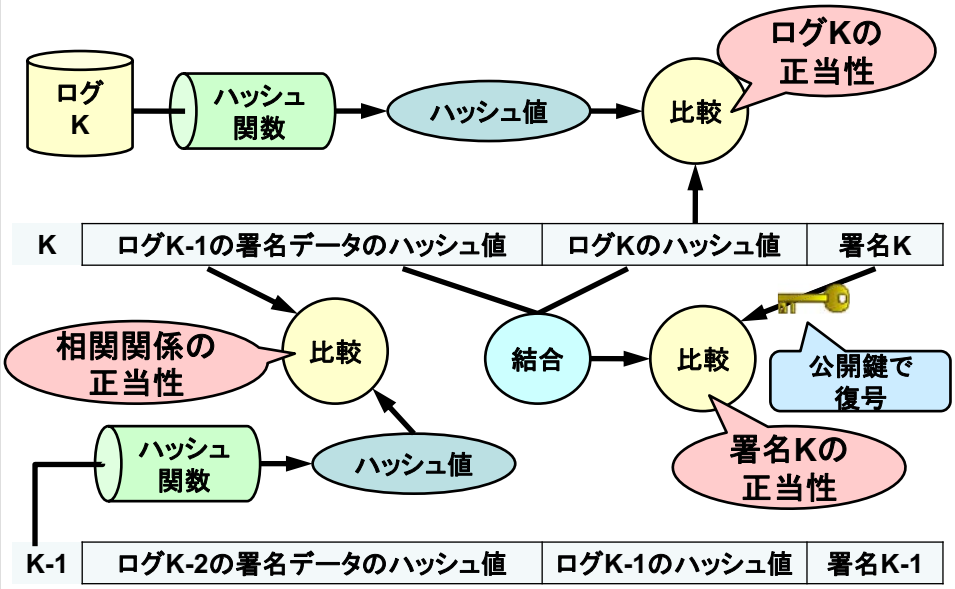
- ログファイルの証拠性を検証する手順
  - ログファイルの改竄の確認
    - ログファイルの正当性
  - 署名ファイルの署名の確認
    - 署名データの正当性
  - 署名データ間の相関関係の確認
    - 相関関係の正当性

➢すべて確認できる場合、改竄なし

➢どれか一つでも確認できない場合、改竄あり

## 署名確認の流れ






## プロトタイプ



- FUSEファイルシステムに実装
  - 複数のファイルを一括して扱う
  - 実装が容易
- FUSE
  - ユーザ空間で実装できるファイルシステム
  - システムコールに対応した関数を作成

➢検討事項
  - 署名に要する時間
  - 時間測定に対する考察

## 時間測定

19

- 1GBのログファイルの署名にかかる時間の測定
  - 20回試行

| | |
|---|---|
| 最長 | 20.2211秒 |
| 最短 | 17.0899秒 |
| 平均 | 18.6730秒 |

HDD: SATA，CPU: Athlon64 X2 4200+(2.2GHz)，OS: Vine Linux 4.2，Memory: 2GB

- 10MBで分割して署名する方法を実装する
  - 自動分割で署名
  - 署名に要する時間は0.2秒

## 自動分割

20

- 10MB程度のログファイルを作成
  - 自動的に新しいログファイルを作成
  - ログファイルの署名データ間に相関関係を持たせる

**ログ1**

Aug 22 19:09:52 osorezan sshd[2241]: Received signal 15; terminating.

Aug 25 10:35:38 osorezan sshd[2151]: Server listening on :: port 22.

Aug 25 10:35:38 osorezan sshd[2151]: error: Bind to port 22 on 0.0.0.0 failed: Address already in use.

10MB

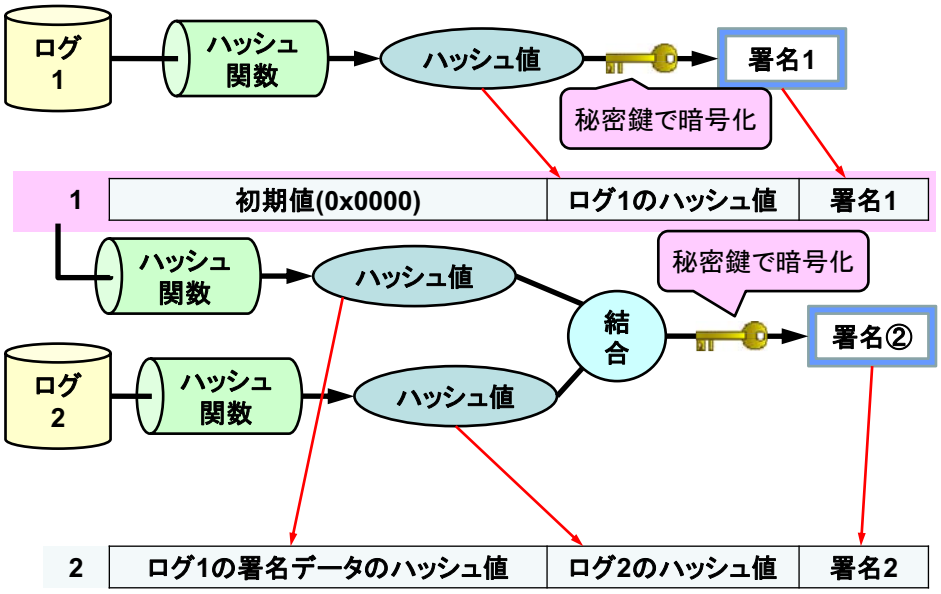
**ログ2**

Aug 25 10:35:58 osorezan last message repeated 3 times

Aug 25 10:35:58 osorezan gdm[2454]: pam_unix(gdm:session) session opened for user tsuruta by (uid=0)

## 分割したログファイルの相関関係






## まとめと今後の課題



- ヒステリシス署名によるログファイル改竄検出機構を備えたファイルシステムを提案
  - 署名データの安全性と有効性を長期間維持可能
  - ログファイルの改竄検出可能
  - 自動分割することで署名に要する時間を短縮

### 今後の課題

- 提案システムのプロトタイプの作成
  - 自動分割により作成したログファイルの命名方法

# 分散ハッシュテーブルを用いた匿名通信方式の提案

近藤正基† 齋藤彰一† 松尾啓志†

# Proposal of Anonymous Communication Method using DHT

Masaki Kondo ,† Shoichi Saito† and Hiroshi Matsuo †

## 1. はじめに

近年, 高度情報化社会の高まりによって, 商業, 行政, 医療, 家庭など, あらゆる分野においてコンピュータとインターネットの利用が一般化している．このため, 医療相談や社内告発といった行為をインターネットを介して行うことが考えられる．このような行為は,「だれ」が「どこ」へ通信を行ったかが分からないような, 強固な匿名性が必要である．

また匿名性を確保するさまざまな匿名通信手法が研究されている．中でも代表されるものに Crowds, オニオンルーティングがある．しかし既存の手法は匿名性を確保するものの，経路が固定的であり可用性が低い．また，大規模なネットワークに対して考慮されておらずスケーラビリティが乏しい．

そこで我々は, 分散型の P2P システムを用いて，多重暗号化と分散ハッシュテーブルである Chord[1] のアルゴリズムを組み合わせることにより, 柔軟な経路選択を行う匿名通信路の提案をする．

## 2. 提案手法

本提案手法は多重暗号化を用いて経路を秘匿し，かつ Chord の ID 空間を利用することで構成される．更に，**受信エリア**と呼ぶ Successor にメッセージを送る区間を設けることで, 可用性とスケーラビリティを確保したまま，高い匿名性を確保する手法を提案する．

### 2.1 構成要素

Chord に加えた提案方式の構成要素について述べる．

**送信者** $Ns$**:** 匿名通信の始点ノード

**受信者** $Nr$**:** 匿名通信の終点ノード

**受信エリア** $A_i(i = 0,\ 1,\ 2,\ ...,\ n)$**:** Chord の ID 空間上で $Ns$ が任意に指定した $n$ 個の ID 部分空間．$A_i(i = 1...n)$ のどれかに $Nr$ が含まれる必要がある．

**受信エリアの最小 ID** $ID_{min}(A_i)$**:** 受信エリア $A_i$ の始点 ID

**受信エリアの最大 ID** $ID_{max}(A_i)$**:** 受信エリア $A_i$ の終点 ID

**受信エリアの始点ノード** $A_is$**:** $ID_{min}(A_i)$ を管理するノード

**受信エリアの終点ノード** $A_it$**:** $ID_{max}(A_i)$ を管理するノード

**受信エリア内ノード** $A_iN_k(k = 0,\ 1,\ ...,\ m_j)$**:** 受信エリアに存在するノード．$A_iN_1$ の Successor が $A_iN_2$ となる

**ヘッダ用暗号化データ** $P_{node}(Y)$**:** データ Y をノード $node$ のヘッダ用鍵 $P_{node}$ で暗号化したデータ

**データ用暗号化データ** $K_{node}(Y)$**:** データ Y をノード $node$ のデータ用鍵 $K_{node}$ で暗号化したデータ

**送信メッセージ** $Ms$**:** $Ns$ が $Nr$ へ送る送信時のメッセージ

**メッセージヘッダ** $H_s$**:** 送信メッセージのヘッダ部

**通信内容** $V$**:** メッセージに含まれる通信内容部

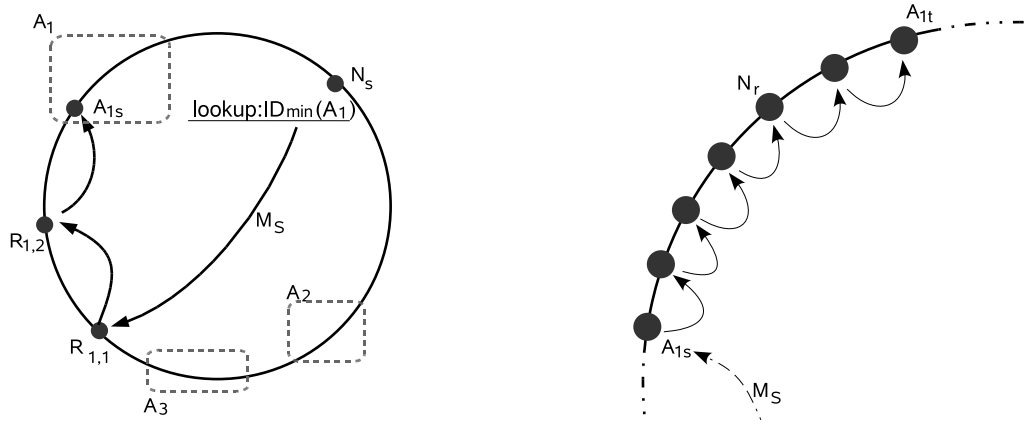
**中継ノード** $R_{i,j}(j = 0,\ 1,\ 2,\ ...,\ l)$**:** メッセージを受信エリア $A_i$ へ Chord アルゴリズムに従い中継するノード

また，文字列 A と B の結合を A‖B で表す．

### 2.2 提案方式

本提案手法は, 多重暗号化したヘッダ部とデータ用鍵で暗号化したデータ部からなるメッセージを，Chord を基盤とした経路制御方式で配送する．また多重暗号

† 名古屋工業大学
Nagoya Institute of Technology




(a)受信エリア外　　(b)受信エリア内

**図 1**　提案方式の経路制御

化であるが，1 度目の通信は鍵管理サーバなどから鍵を得ることで行うとする．2 度目の通信以降は 1 度目の通信にテンポラリな共通鍵を付加し，その鍵を用いる．以下に，提案手法のメッセージ生成と送信と返信の各フェーズについて述べる． 最後に全体の動作の概要を示す．

#### 2.2.1　メッセージ生成フェーズ

本手法のメッセージは，ヘッダ部とボディ部に分かれる．ヘッダ部は，受信エリア群の始点ノード ID を多重暗号化して作成する．以下, メッセージのヘッダ部とボディ部に分けて生成方法を述べる．

**(ヘッダ部)**

**(1)**　送信者は受信エリア $A_i(i \geq 1)$ を決める．受信エリアは最小 ID と最大 ID を指定して定める ($Nr$ がいずれかの受信エリアに含まれている必要がある)．

**(2)**　選択した受信エリアを任意の順に並び替える．このとき，送信者 $Ns$ から通信順に $i = 1, 2, 3...$ とする．またヘッダ部の多重暗号化のために $A_it$ の暗号鍵 $P_{Ait}$ を得る．

**(3)**　各エリアの最小 ID を次に示す再帰的計算により多重暗号化する．

$$H_i = P_{i+1}(ID_{min}(A_{i+2}) \parallel H_{i+1})$$

以上によりヘッダ部 $Hs$ $(= H_0)$ が求まる．

**(ボディ部)**

**(1)**　ボディ部 $(B_s)$ を次に示すとおり暗号化を行う．

$$B_s = K_{Nr}(V)$$

**(2)**　$B_s$ を受信者が位置する受信エリア以前に通信を行うエリアの $A_it$ の暗号鍵 $P_{Ait}$ を用いて多重暗号化を行う．

$$B_i = P_{i+1}(ID_{min}(B_s))$$

**(3)**　ヘッダ部 $H_s$ と結合する．これを送信メッセージ $M_s$ とする．

#### 2.2.2　送信フェーズ

提案手法の経路制御は主に受信エリア外と受信エリア内に分けられる．メッセージ送信フェーズの動作を以下に示す．

**(受信エリア外)**

**(1)**　送信者 $Ns$ は生成した $Ms$ のヘッダ部に従って, $ID_{min}(A_1)$ へ Chord のアルゴリズムに従い, メッセージを送信する．

**(2)**　$M_s$ を受け取った中継ノード $R_{i,j}$ は, ヘッダ部に従って, 以下のいずれかの処理を行う．

- 自身が $ID_{min}(A_i)$ を管理するノードでなければ，Chord のアルゴリズムに従いメッセージを $R_{i,j}(j = j + 1)$ に送信する．
- 自身が $ID_{min}(A_i)$ を管理するノードであれば, ヘッダ部の復号を試みる．復号できなければ受信エリアであることを示すフラグ (以下, 受信フラグ) を立て, Successor にメッセージを送信し，受信エリア内処理 (2) を実行する．このときヘッダ部から $ID_{min}(A_i)$ を指した情報を削除する．

**(受信エリア内)**

**(1)**　受信エリアフラグが立った状態で Predeseccor からメッセージを受けとった場合は, 中継ノードは以下のいずれかの処理を行う．

- ヘッダ部の復号を試み, 復号できなければ Suc-

cessor へメッセージを送信する．

- ヘッダ部の復号を試み, 復号できれば次の宛先 ($ID_{min}(A_{i+1})$) を得る．ここで, このノードは初めて自分が $A_it$ であることを知る．またヘッダ部を復号できた際，同様の鍵を用いてデータ部も復号を行う．$A_it$ は新たに initiator となって $R_{i+1,1}$ に $M_s$ を送る (受信エリア外処理 (2) へ戻る)．
- ヘッダ部の復号を試み, 復号できたとき，次の宛先 ID が無ければメッセージの送信を終了する．

**(2)** (1) の動作を行った後, データ用秘密鍵を用いてメッセージのデータ部の復号を試みる．このとき復号が可能ならば, メッセージ受信となる．

#### 2.2.3 動作の概要

図 1 に動作を示す．まず，送信者 $Ns$ は最初の受信エリアの始点ノード $A_1s$ にメッセージ $Ms$ を Chord のアルゴリズムに基づいて送信する (図 1(a) 参照)．$A_1s$ からは Successor にメッセージを送信する (図 1(b) 参照)．受信エリア内のノードは，自身の通信用鍵を用いてヘッダ部の復号を試みる．そして, 復号できなければ Successor に送信する．もし復号できた場合は, 次のエリアの始点ノードの $ID_{min}(A_2)$ を知ることができるので, initiator となって Chord のアルゴリズムに従いメッセージを送信する．これを繰り返し行うことで，ヘッダ内に受信エリアがなくなるまでメッセージを送信する．

また, 受信エリア内のノード ($A_iN_k$) はメッセージ中継後に, データ部をデータ用鍵を用いて復号を試みる．復号できた場合, 当該ノードが受信ノード $Nr$ であることが判明する．返信は, 送信メッセージのデータ部に含まれていた返信用ヘッダを用いて同様に送信する．

中継ノード故障などの場合, 受信エリア外では Chord のアルゴリズムに従い各々が自立的に次の中継ノードを選ぶ．また, Chord のアルゴリズムは Successor と常に接続していることから, 受信エリア内では Successor に正しくメッセージを送信できる．しかし，受信エリアの終点ノード ($A_it$) が故障などにより利用できない場合が考えられる．この場合に対処するため，ヘッダ用復号鍵を分割して前後のノードに渡しておく．終点ノード ($A_it$) が離脱した際は前に位置するノードが後ろに位置するノードに分割した鍵を渡すことで，$A_it$ の離脱に対処する．

**表 1** 既存手法との比較

| 手法 | Crowds | オニオンルーティング | 提案手法 |
|---|---|---|---|
| 送信者匿名性 | ○ | ○ | ○ |
| 受信者匿名性 | × | ○ | ○ |
| つながりの匿名性 | △ | ○ | ○ |
| 可用性 | △ | × | ○ |
| スケーラビリティ | × | △ | ○ |
| 応答時間 | ○ | △ | △ |
| 鍵管理コスト | ○ | △ | × |

## 3. 既存手法との比較と考察

オニオンルーティングと Crowds との，送信者の匿名性, 受信者の匿名性, 送受信者のつながりの匿名性, 可用性とスケーラビリティを比較する．それぞれの比較をまとめたものを表 1 に示す.

### 3.1 考　　察

Crowds，オニオンルーティングともに，経路が固定的であるためそれらと比較して提案手法は高い可用性があると言える．また Crowds はグループのメンバシップを専用のサーバによって集中的に管理すること，オニオンルーティングは経路を生成する際にフラッディングを用いることによりスケーラビリティが低いといえる．対して提案手法は，Chord を用いることによりスケーラブルな通信を行うことができると考えられる．

## 4. ま と め

本論文では, オニオンルーティングなどで扱われる多重暗号化に分散ハッシュテーブルである Chord の経路選択を用いる手法を提案した．IP アドレスを Chord の ID に写像し, メッセージの宛先を ID で管理することで高い可用性とスケーラビリティを満たすことができる．

今後は, 各種攻撃に対する防御策と不正に利用された場合の不正者の特定法を検討する．その上で高い匿名性を確保できるよう改良を行う．また, 大規模な実験環境を用いた性能評価実験を行う計画である．

## 参 考 文 献


1) Stoica，I., Morris, R., Karger, D., Kaashoek, F. and Balakrishnan, H.: Chord; A Scalable Peer-To-Peer Lookup Service for Internet Applications, Proc. 2001 ACM SIGCOMM Conference, pp.149-160(2001).
2) 近藤 正基, 齋藤 彰一, 松尾 啓志: DHT を用いた双方向匿名通進路の提案,情報処理学会研究報告, 2008-CSEC-42，pp.195-202(2008).

# 分散ハッシュテーブルを用いた匿名通信方式の提案

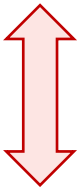
○近藤　正基
齋藤　彰一
松尾　啓志
(名工大)

## 発表の流れ

1. 背景
2. 従来研究とその問題点
   - オニオンルーティング
3. 提案手法の説明
   - Chordとオニオンルーティングを用いた匿名通信路
   - 受信エリア・復号バックアップノード
4. 提案手法の考察，評価
   - 従来研究との比較
5. まとめ

## 研究の背景

- 実社会
  - 匿名を前提とした仕組み
    - 内部告発，医療相談，投票など



ギャップ

**匿名通信**
**が必要**

- インターネット
  - IPアドレスと時間から個人の特定が可能

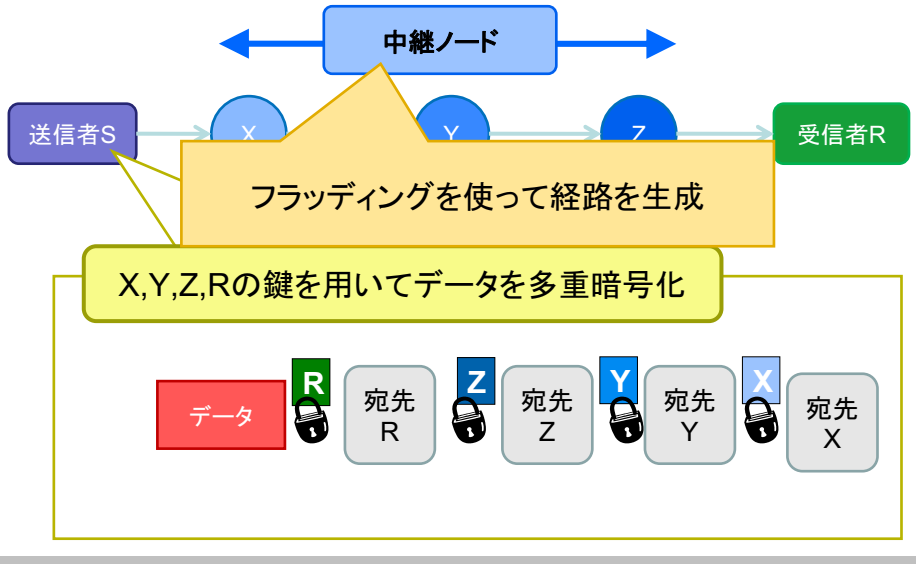
3

## 従来研究

- ブロードキャストを用いる手法
- いくつかの計算機で中継する手法
  - Crowds
    中継の有無を確率で決定する手法
  - オニオンルーティング
    多重暗号化により経路を秘匿する手法

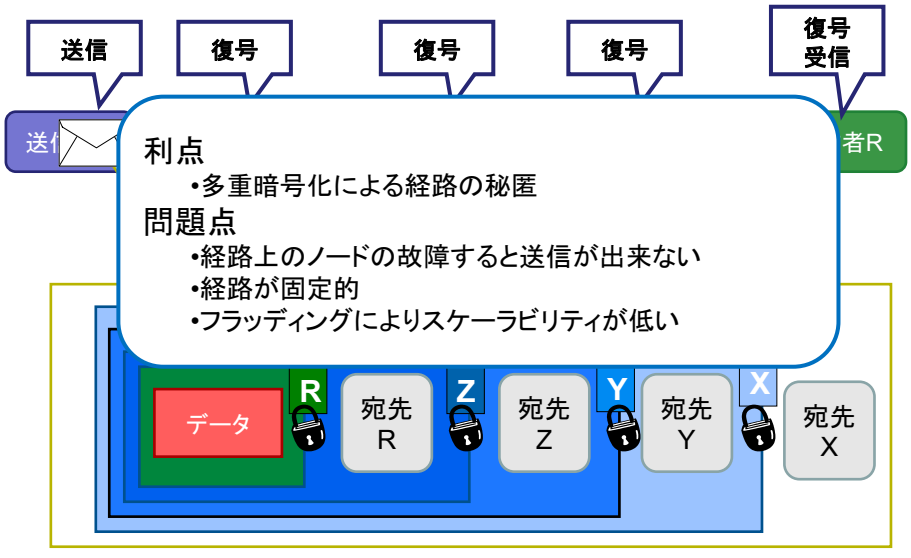
4

# オニオンルーティング




5

# オニオンルーティング




6

## 従来手法まとめ

| | Crowds | オニオンルーティング |
|---|---|---|
| 送信者の匿名性 | ○ | |
| 受信者の匿名性 | × | |
| つながりの匿名性 | △ | |
| 可用性 | △ | × |
| スケーラビリティ | × | △ |
| 応答時間 | ○ | △ |

**利点**
- 高い匿名性

**問題点**
- 大規模なネットワークに対応不可
- ノードの離脱,故障に弱い

7

## 提案手法1

既存手法の問題点

- 可用性とスケーラビリティが低い

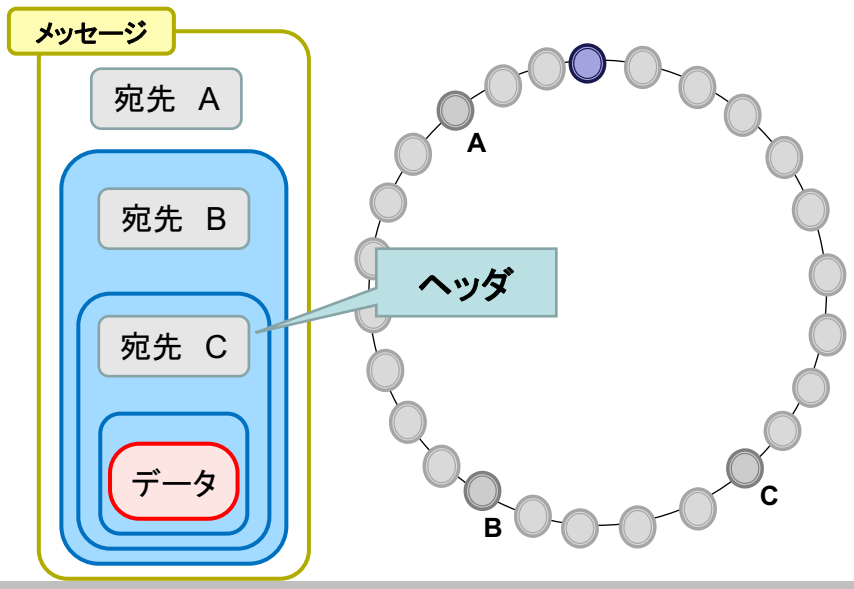
**分散ハッシュテーブル**
- IPアドレスではなくIDによるノード管理
- 各ノードの独立した経路制御
- Chord

**提案手法1**

**Chordを用いたオニオンルーティング**

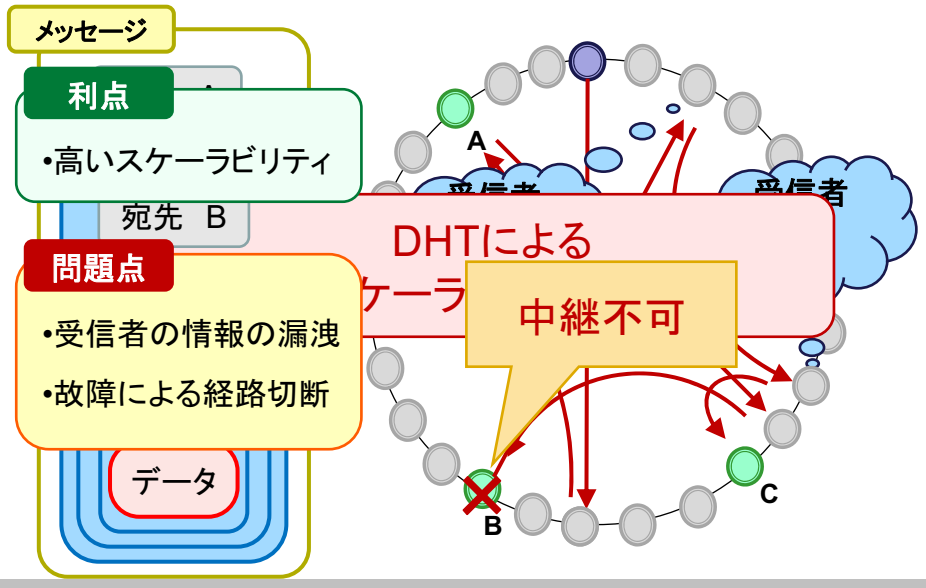
8

## 提案手法1の動作




9

## 提案手法1の動作




10

## 提案手法1の問題点と改良

- Chord中継ノードへの受信者の情報の漏洩

  通信ヘッダに受信者が現れることが原因

  提案

  通信宛先ノードと受信者を分離する

- 復号ノードの故障による経路の切断

  復号ノードが一つしかないことが原因

  提案

  復号ノードのバックアップを作成する

11

## 提案手法2

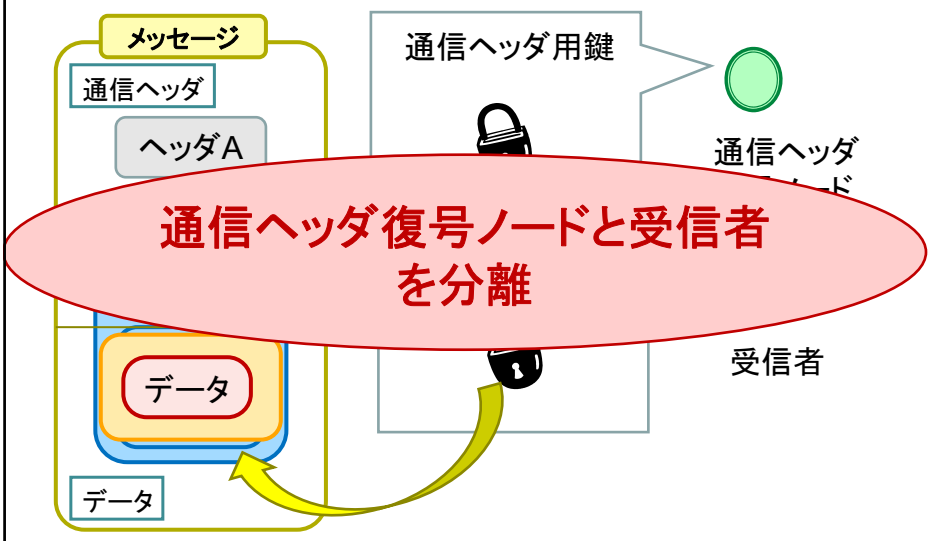
•通信ヘッダとデータを異なる鍵で暗号化
  - 通信ヘッダ復号ノードと受信者を分離

•通信ヘッダを使用しない経路制御区間受信エリア
  -受信者を含むように受信エリアを設定
    -受信者が通信宛先ノードでなくてもよい
  -受信エリア間で多重暗号化

•復号のバックアップノードを生成
  - 通信ヘッダ復号鍵を分割して他ノードへ譲渡

12

## 通信ヘッダとデータを異なる鍵で暗号化




13

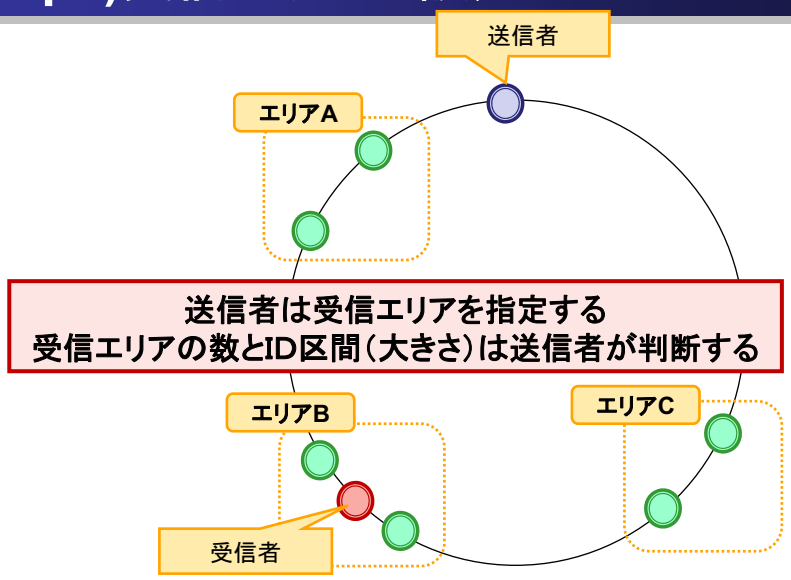
## 提案手法2

- 通信ヘッダとデータを異なる鍵で暗号化
  - 通信ヘッダ復号ノードと受信者を分離
- 通信ヘッダを使用しない経路制御区間受信エリア
  - -受信者を含むように受信エリアを設定
    - -受信者が通信宛先ノードでなくてもよい
  - -受信エリア間で多重暗号化
- 復号のバックアップノードを生成
  - 通信ヘッダ復号鍵のレプリカを譲渡

通信全体の流れに沿って説明

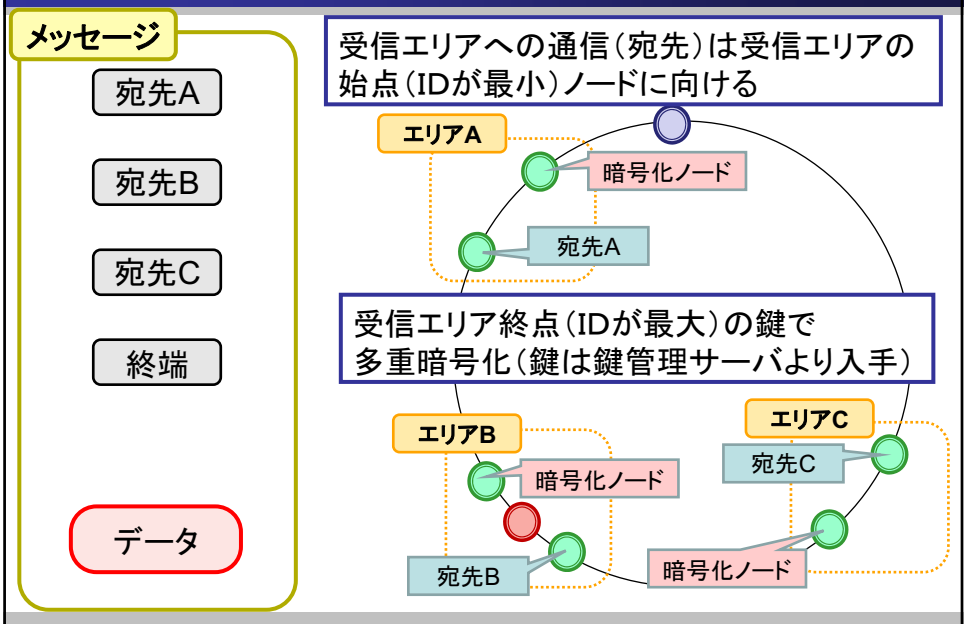
14

## (Step1)受信エリアの設定

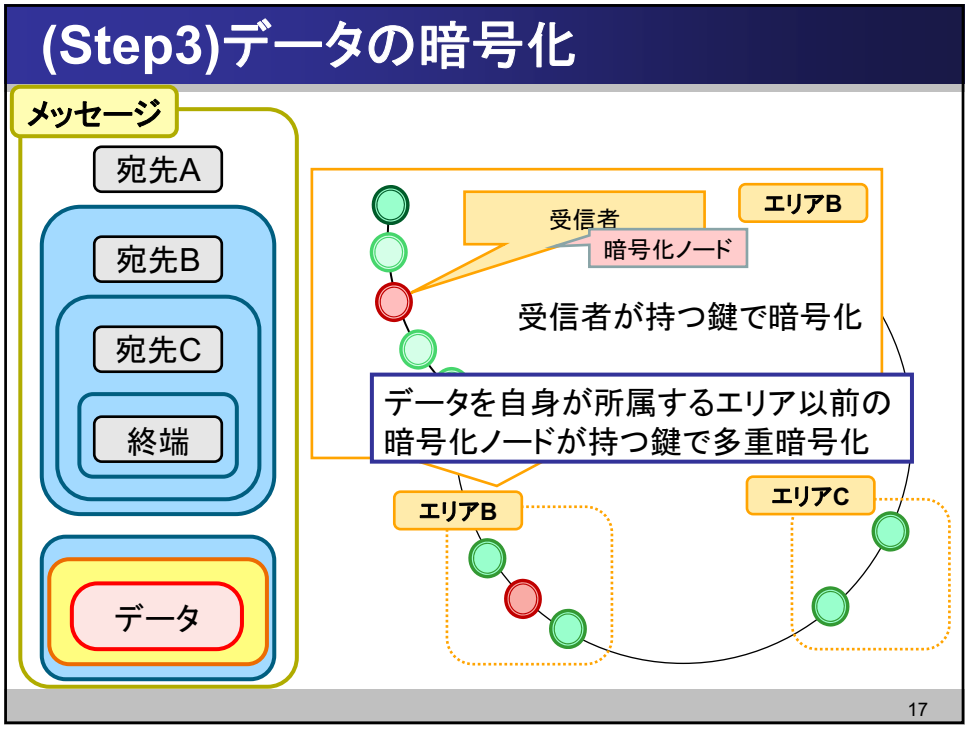


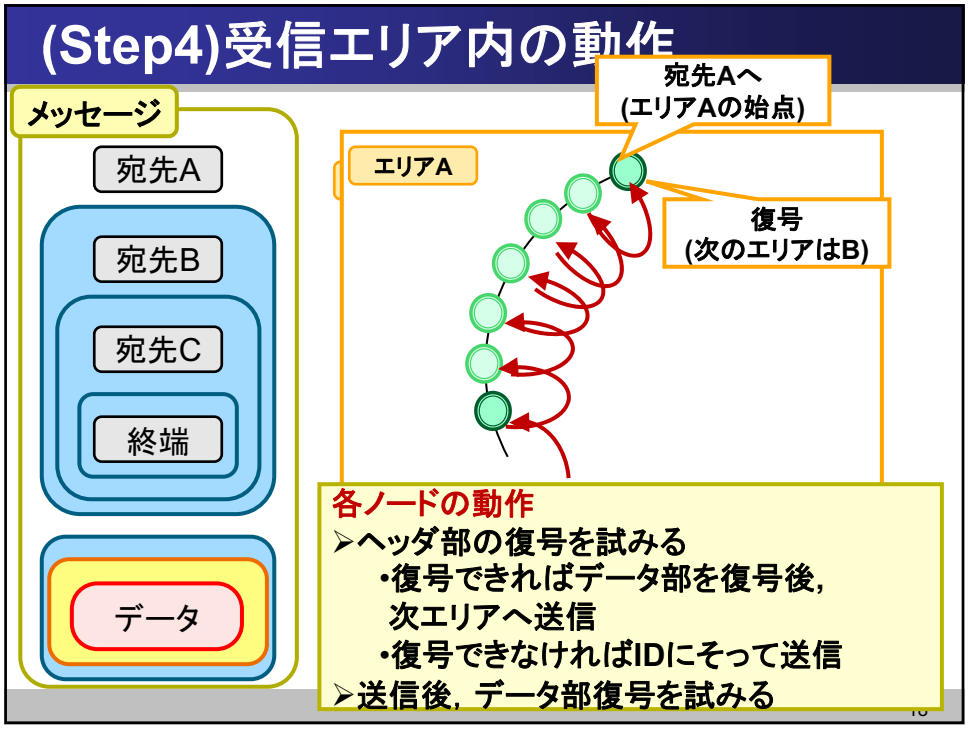

15

## (Step2)通信ヘッダの作成

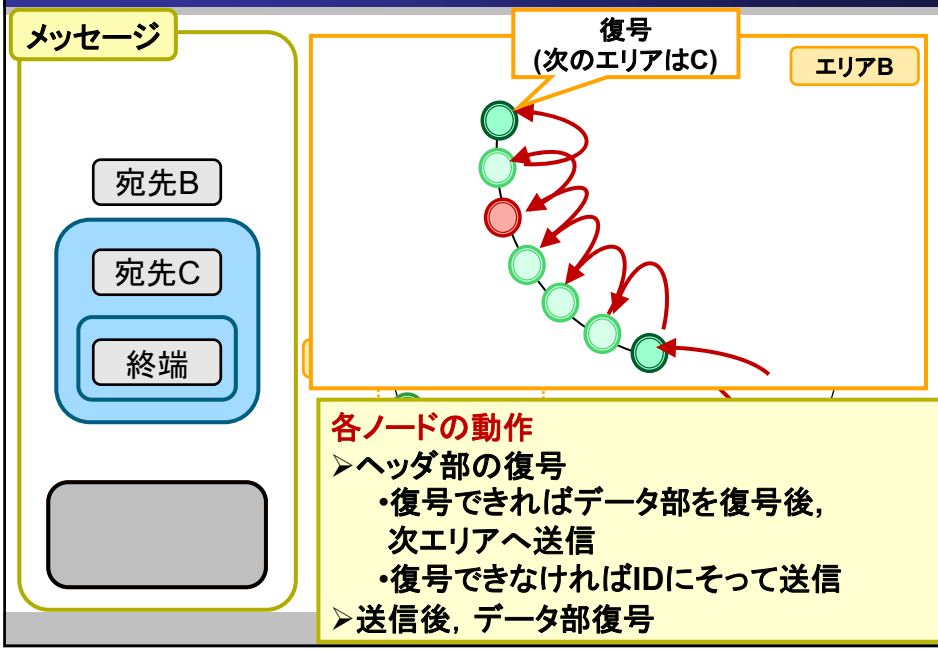



16


(Step3)データの暗号化
メッセージ
宛先A
宛先B
宛先C
終端
データ
受信者
暗号化ノード
エリアB
受信者が持つ鍵で暗号化
データを自身が所属するエリア以前の
暗号化ノードが持つ鍵で多重暗号化
エリアB
エリアC
17

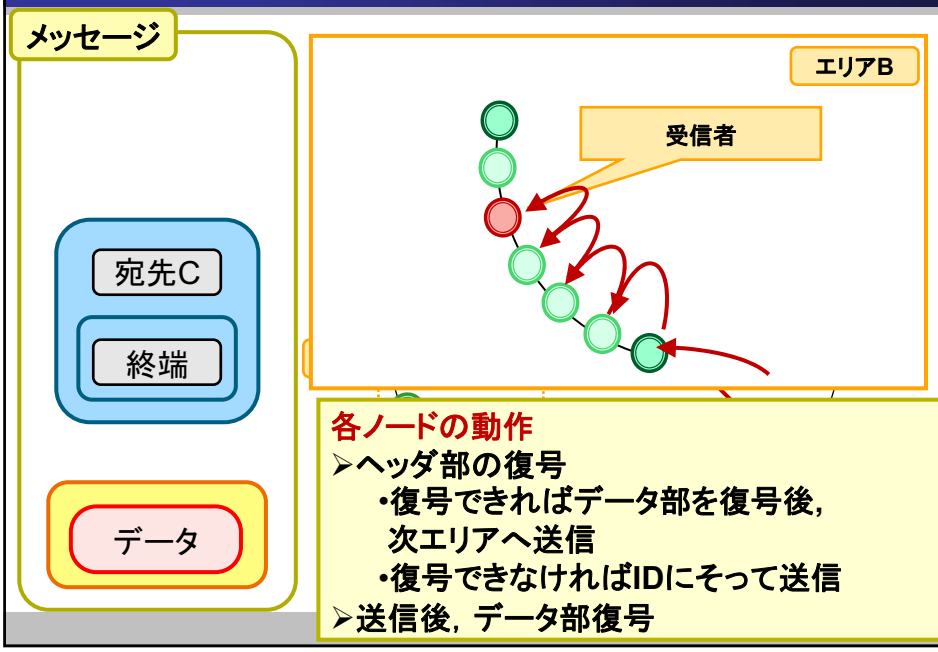

(Step4)受信エリア内の動作
メッセージ
宛先A
宛先B
宛先C
終端
データ
宛先Aへ
(エリアAの始点)
エリアA
復号
(次のエリアはB)
各ノードの動作
➢ヘッダ部の復号を試みる
•復号できればデータ部を復号後,
次エリアへ送信
•復号できなければIDにそって送信
➢送信後，データ部復号を試みる

## (Step5)受信者を含む受信エリアでの動作

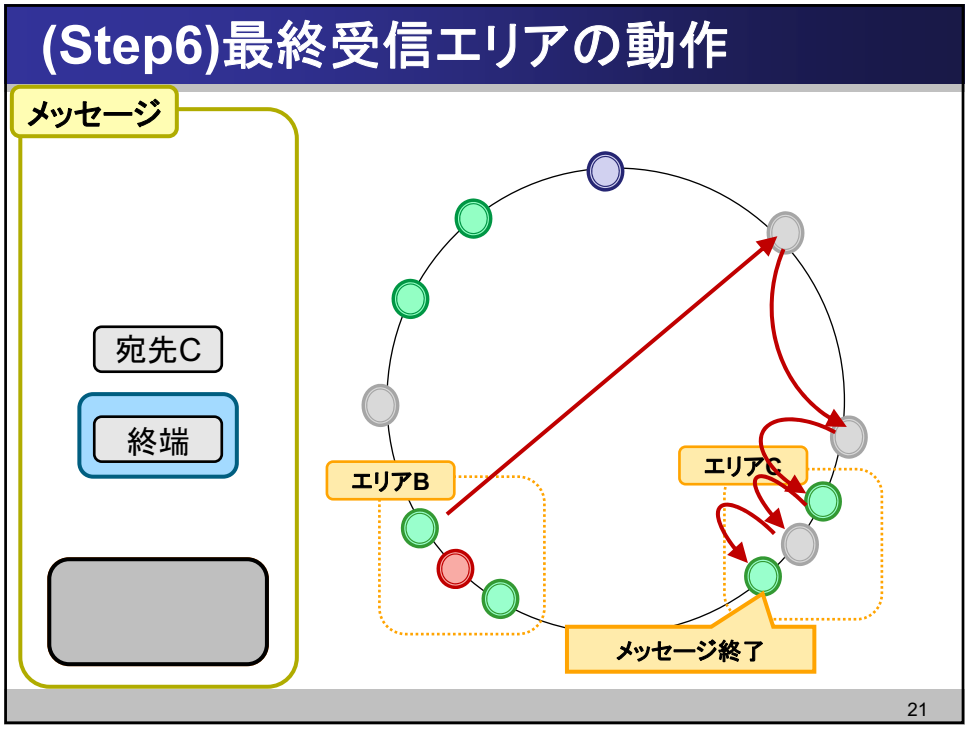


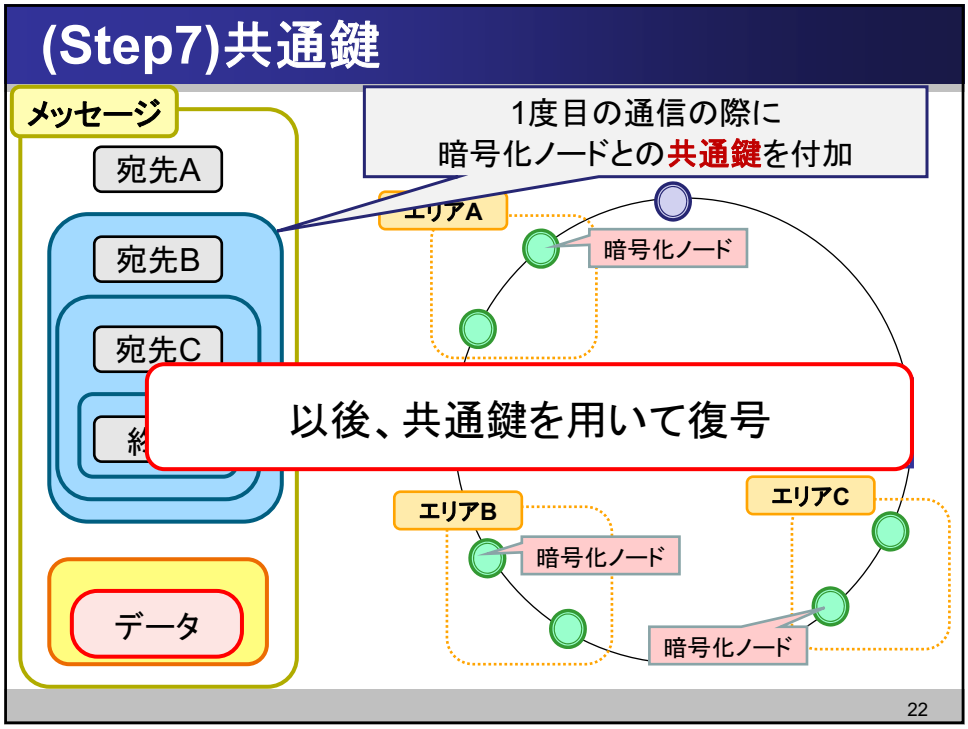

各ノードの動作
- ヘッダ部の復号
  - 復号できればデータ部を復号後，次エリアへ送信
  - 復号できなければIDにそって送信
- 送信後，データ部復号

## (Step5)受信者を含む受信エリアでの動作

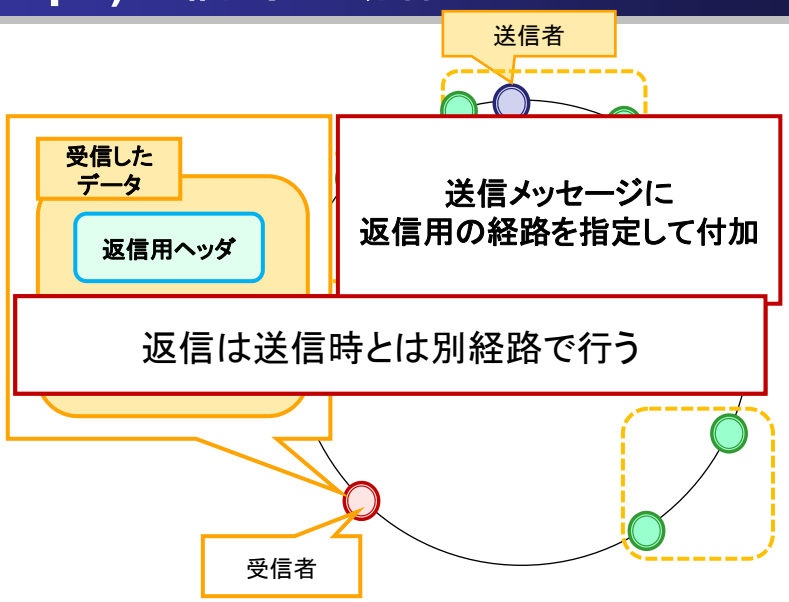



各ノードの動作
- ヘッダ部の復号
  - 復号できればデータ部を復号後，次エリアへ送信
  - 復号できなければIDにそって送信
- 送信後，データ部復号


(Step6)最終受信エリアの動作
メッセージ
宛先C
終端
エリアB
エリアC
メッセージ終了
21


(Step7)共通鍵
メッセージ
1度目の通信の際に
暗号化ノードとの共通鍵を付加
宛先A
宛先B
宛先C
データ
エリアA
暗号化ノード
以後、共通鍵を用いて復号
エリアB
暗号化ノード
エリアC
暗号化ノード
22

## (Step8)返信時の動作



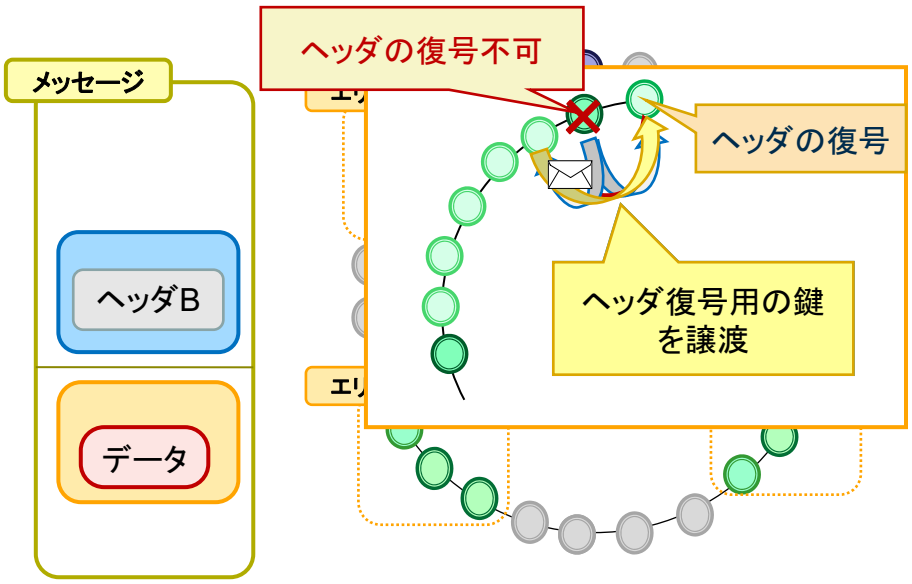

23

## 提案手法2

- 通信ヘッダとデータを異なる鍵で暗号化
  - 通信ヘッダ復号ノードと受信者を分離
- 通信ヘッダを使用しない経路制御区間受信エリア
  - -受信者を含むように受信エリアを設定
    - -受信者が通信宛先ノードでなくてもよい
  - -受信エリア間で多重暗号化
- 復号のバックアップノードを生成
  - 通信ヘッダ復号鍵を分割して他ノードへ譲渡

24

## バックアップノードの生成




25

## 提案手法2まとめ

**匿名性の向上**

鍵の分割と受信エリアの作成と多重暗号化

可用性とスケーラビリティを有した
匿名通信路

鍵の分割とDHTの経路制御による鍵の分割、譲渡

➡ 復号ノードが故障の際のバックアップ

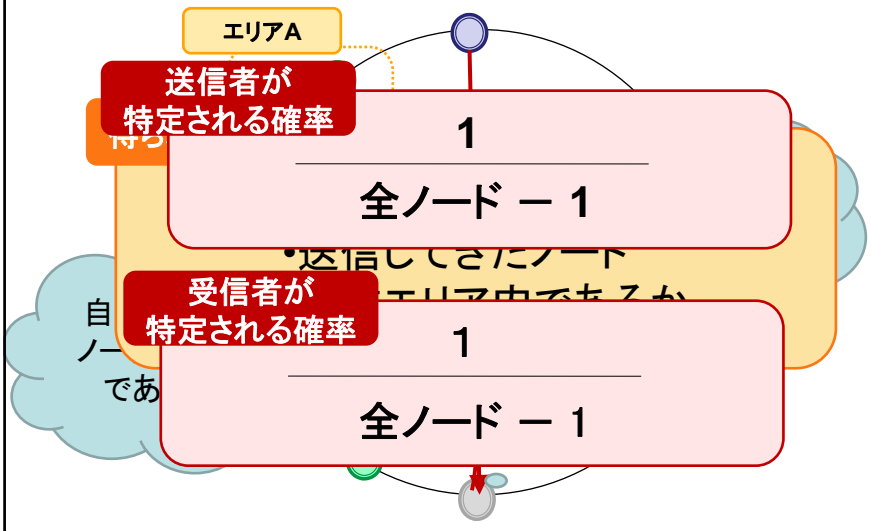
26

## 匿名性の考察

- 送信者および受信者の検出攻撃への耐性
  - 単独で行った場合
  - 複数のノードが結託を行った場合

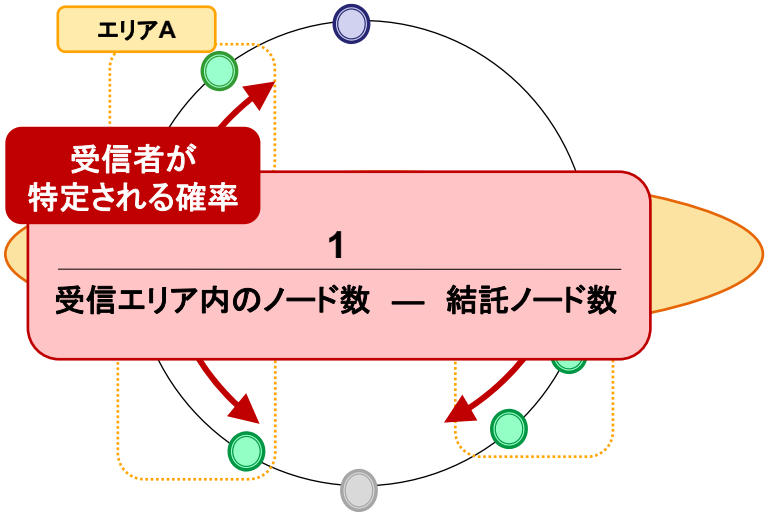
27

## 匿名性考察(単独)




28

## 匿名性考察(結託あり)




29

## 匿名性まとめ

- 送信者が特定される確率

$$\frac{1}{\text{全ノード数} - 1}$$

- 受信者が特定される確率

単独

$$\frac{1}{\text{全ノード数} - 1}$$

結託

$$\frac{1}{\text{受信エリア内ノード数} - \text{結託ノード数}}$$

30

## 従来手法比較

| | Crowds | オニオン ルーティング | 提案手法 |
|---|---|---|---|
| 送信者の匿名性 | ○ | ○ | ○ |
| 受信者の匿名性 | × | ○ | ○ |
| つながりの匿名性 | △ | ○ | ○ |
| 可用性 | △ | × | ○ |
| スケーラビリティ | × | △ | ○ |
| 応答時間 | ○ | △ | △ |
| 鍵管理 | ○ | △ | × |

31

## まとめ

**可用性とスケーラビリティを有した匿名通信路の提案**

提案1

- Chordを用いたオニオンルーティング

提案2

匿名性の向上:

- 受信者が通信ヘッダに表れない

可用性の向上:

- 受信エリアの復号鍵のバックアップ

**今後の課題**

- 実装の詳細の検討
- 提案手法でのパフォーマンスの評価
- 悪用された場合の送信者の特定手法の検討

32

# PerlMachine の設計と実装

浅野 一成 †

† 東京農工大学大学院工学府情報工学専攻

## 1 はじめに

プログラミング言語 Perl(以下 Perl) は，豊富なライブラリ群や特定のパラダイムに縛られないプログラミングスタイルを特長とし，Windows や Linux 上のバッチ処理や Web アプリケーションなどに広く用いられている．本報告では，そのような特長を持った Perl による高水準言語マシンの設計と実装について報告を行う．

## 2 先行事例と課題

前章で述べた Perl の利点を活かし，2001 年にユーザレベルのソフトウェアをすべて Perl で記述しようとしたプロジェクト Perl/Linux[1] が立ち上げられた．Perl/Linux では，init スクリプト以降のユーザプログラムは，すべて Perl で実現されており，Perl で書かれたシェル (psh), UNIX 互換のユーティリティ群 (PPT) などが作成された．

しかしながら，計算機資源をつかさどるカーネルには Linux が利用されており，カーネル層を含めると，ソフトウェア層の大半は C 言語で構成されることになる．そのため，Perl/Linux を Perl による高水準言語マシンと見なすことは難しい．

## 3 目標と設計方針

PerlMachine では，Perl 処理系を PC/AT 互換機上のハードウェア上で直接動作させ，カーネルレベルを含めたソフトウェア階層すべての部分を Perl で記述することを目標とする．最終的な目標として，以下に挙げる資源管理を，Perl で記述されたオペレーティングシステム (PerlOS) 上に実装する．

- デバイスドライバ
- スレッドライブラリ
- ファイルシステム
- TCP/IP プロトコルスタック
- ウィンドウシステム

PerlMachine のコアとなる言語処理系は，ハードウェアアーキテクチャレベルで独自実装せず，Perl.org[2] で利用されている処理系を利用し，PC/AT 互換機のベアマシン上で実行する．カーネル層では PerlOS が動作し，元来 C や C++で実装が行われていた計算機資源の管理を Perl で行う．

## 4 設計

PerlMachine の構成を図 1 に示す．

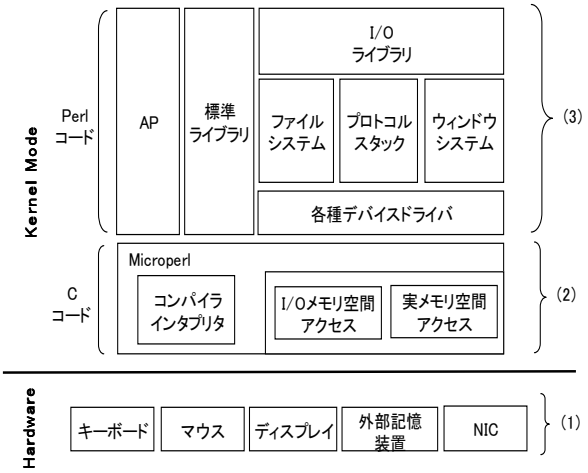



図 1: システムの全体構成

PerlMachine では前述のとおり，ソフトウェア階層をすべて Perl で記述することを目標としており，C 言語の部分は Perl 処理系，処理系拡張部分とプロセッサのコンテキスト退避部分のみとなっている．

PerlMachine におけるプログラムの実行単位は，処理系のメモリ上の実体である，インタプリタインスタンスとなる．インタプリタインスタンスは，スタックマシンのスタックや Perl の変数テーブルなどから構成される．インスタンスがメモリ上に複数ある場合，個々のインスタンスの内容は独立している．

処理系がハードウェア資源を要求するような VM 命令を実行する場合，いったん実行インスタンスがデバイスドライバ実行用のインスタンスに切り換えられ，Perl で記述されたデバイスドライバが実行される．機能ごとにインスタンスを切り分けることによって，デバイスからの外部割込み処理に対応できる．

†Kazunari ASANO
†Faculty of Engineering, Tokyo University of Agriculture and Technology

## 5 実装

Perl 処理系には，Perl.org で配布されている処理系をコンパイルするための Perl 処理系である Microperl[3] を用いた．Perl の組込み関数にはハードウェアアクセス機能が無いため，Microperl にハードウェアアクセスのための拡張を XSUB で施した．XSUB は C で処理系を拡張するための Perl 標準の機能である．

それらの XSUB を用いて，デバイスドライバ (フロッピーディスク，ハードディスク，キーボード/マウス) とファイルシステム (FAT12/16/32, ext2) の実装を行った．

## 6 おわりに

本報告では，Perl による高水準言語マシンの設計と実装について述べた．これまでにシングルタスク環境において，複数インスタンスのシーケンシャルなプログラム実行が実現された．また，インスタンスを用いたデバイスドライバやファイルシステムの試作を行った．

今後の予定として，PerlOS のマルチスレッド環境の実現が挙げられる．現在はシングルインスタンスをシーケンシャルに動作させているが，マルチスレッド環境では複数インスタンスがプリエンプティブに実行される．スレッド実行のモデルとして，1 インスタンスをネイティブスレッドの 1 スレッドに割り当てる方式を予定している．

## 参考文献

[1] Rafael George, Don Mahurin, Peter Willis : Perl/Linux, http://perllinux.sourceforge.net/

[2] The Perl Foundation Site Information and Contacts, The Perl NOC : Perl.org, http://www.perl.org/

[3] Simon Cozens : Microperl, The Perl Journal Volume 5, Number 3 (#19), Fall 2000

# PerlMachineの設計と実装

浅野　一成*
*東京農工大学大学院工学府情報工学科



## 発表の流れ

- 本研究の背景
- 既存システムの問題点
- 本研究の目標と設計方針
- PerlMachineの構成
- PerlMachineにおけるプログラムの実行モデル
- Perl処理系の拡張
- デバイスアクセスと割込み処理
- PerlMachineの実行環境
- 現在の実行モデルと実行フロー
- 今後の予定
- まとめ

# 本研究の背景

- **プログラミング言語Perlの特徴**
  - 軽量なスクリプト言語
    - プログラマを特定のプログラミングスタイルに縛らない
  - 豊富なライブラリ群とCPAN
    - ライブラリが豊富さから，テキスト処理だけなく，Webアプリケーション・管理スクリプト・GUIアプリケーション・各種フレームワークの登場
- **Perl/Linux**
  - Perlの利点を活かし，ユーザアプリケーション層すべてをPerlで実現しようとしたプロジェクト
    - http://perllinux.sourceforge.net/
  - カーネルにLinuxを利用し、initスクリプト以降のアプリケーションプログラムをすべてPerlで記述



# Perl/Linuxの問題点

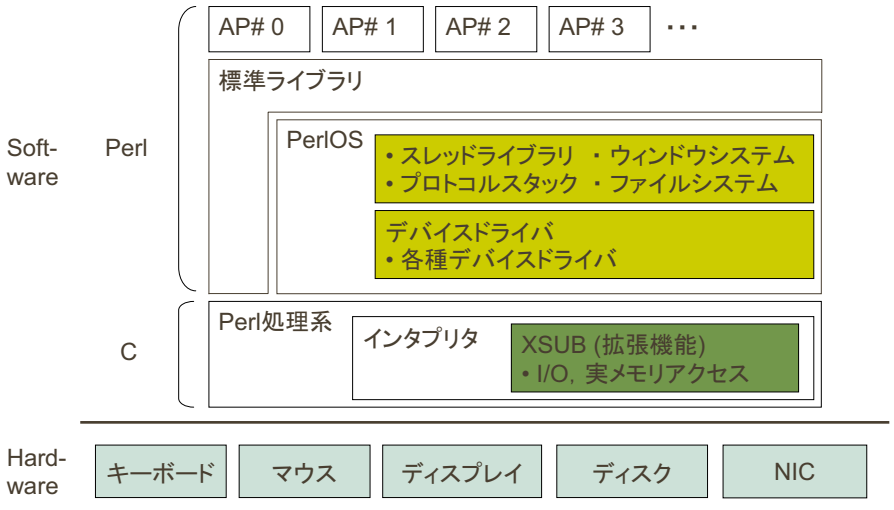
- **実現できていること**
  - Perl処理系単体のLinuxカーネル上への導入
  - ユーザアプリケーションのPerlによる実装
    - Perlによるシェル（Perl Shell）
    - UNIXユーティリティのPerlへの移植（Perl Power Tool）
    - CPANモジュールを利用したアプリケーションプログラム
- **高水準言語マシンとしての課題**
  - システムソフトウェアはPerlで記述されていない
    - カーネルはLinuxであるため、Cやアセンブリ言語で記述されている
    - Perlによる高水準言語マシンを実現できていない

# 本研究の目標と設計方針

- **PerlMachineの実現**
  - PC/AT互換機のベアマシン上にPerlの言語処理系を搭載した高水準言語マシンの実現
    - ソフトウェアで実装された言語処理系を用いた高水準言語マシンを目指す
    - ベアマシン上には様々なプラットフォーム上で動作実績のあるPerl処理系を搭載する
- **PerlOSの実現**
  - PerlMachine上の計算機資源管理を行う，Perlで記述されたオペレーティングシステムの実現
    - アプリケーションだけでなく，システムソフトウェア層にあるプログラムもPerlで記述する
    - 可能な限りPerlで記述し，Perlの仕様で実現できない部分にはPerl処理系の拡張機能を用いる



# PerlMachineの構成



## PerlMachineの構成

- **ターゲットアーキテクチャ**
  - PC/AT互換機 x86
- **Perl処理系**
  - Perlからハードウェアアクセスを行うための機能の追加
- **ハードウェア資源管理**
  - PerlOS
    - ディスクドライバ，PS/2装置ドライバ，NICドライバ，USBドライバ
    - スレッドライブラリ，ウィンドウシステム，TCP/IPプロトコルスタック
    - ファイルシステム

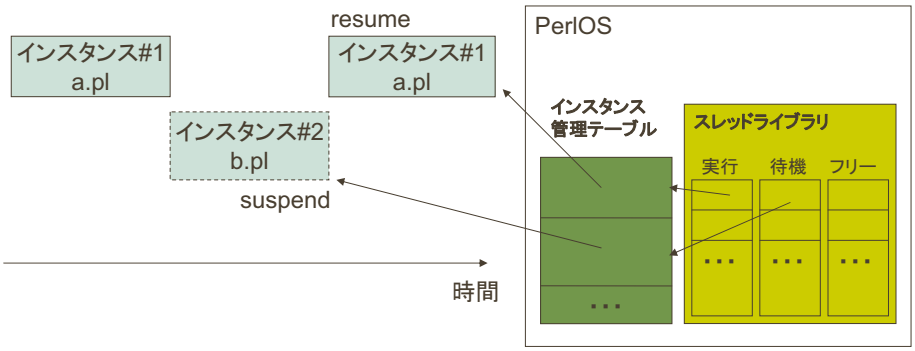


## PerlMachineのプログラム実行単位

- **プログラムの実行単位**
  - インタプリタインスタンス
- **インタプリタインスタンス**
  - Perl処理系が1つのインタプリタごとに保持するインタプリタグローバルな変数群の実メモリ上における実体
  - インスタンスが複数あった場合，メモリ内で各々独立
- **インタプリタインスタンスが保持する情報**
  - 処理系内部のメモリ管理エリア
  - 入力プログラムから生成された構文木
  - スタックマシンで利用されるスタック群
  - Perlプログラムにおける変数管理エリア

# PerlMachineのプログラム実行モデル

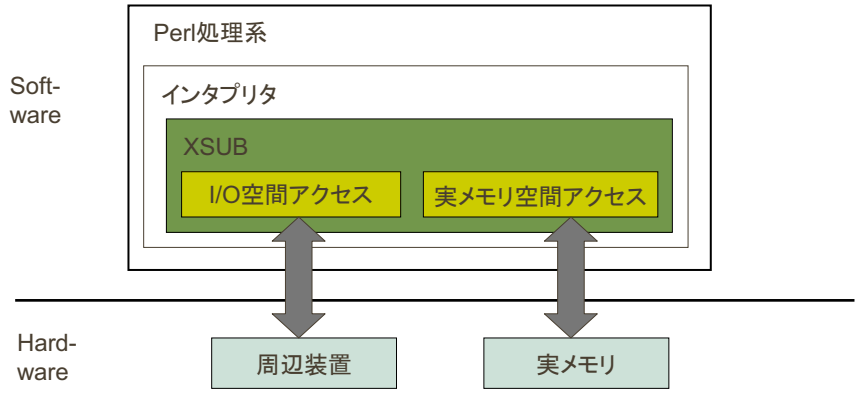
- **プログラムの実行モデル**
  - 1インタプリタインスタンスを1仮想CPUとして実行
  - 各インスタンスはプリエンプティブに動作
    - インタプリタインスタンスはスレッドライブラリで管理される





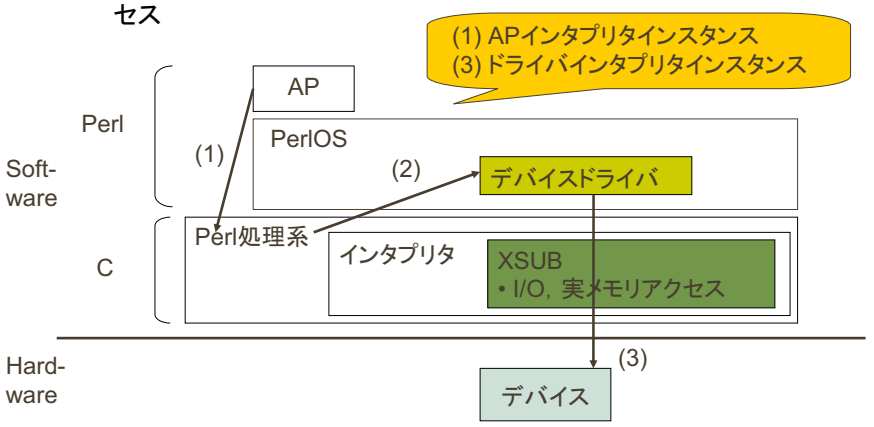

# Perl処理系の拡張

- **I/O，実メモリ空間へのアクセス**
  - 組込み関数では不可能であるため，処理系標準の拡張機能(XSUB)を利用



## デバイスアクセスを伴う処理系の実行フロー

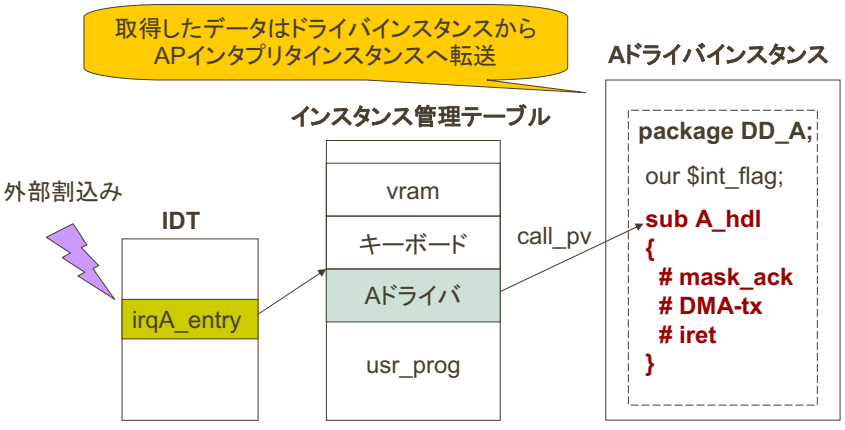
- **Perl処理系からデバイスアクセスまでの流れ**
  - 処理系はPerlOS内部のデバイスドライバを経由
  - (1) APで構文木実行, (2) ドライバインスタンス実行, (3) デバイスアクセス






## 外部割込み処理の実行フロー

- **外部割込みの実行フロー**
  - 処理系の実行インスタンスを対象となるデバイスドライバのインスタンスに切替え



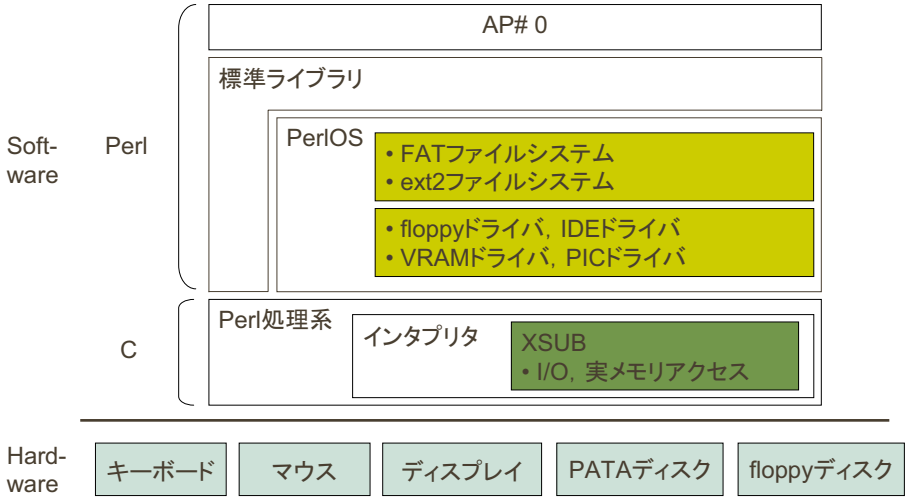
# PerlMachineの実行環境

- **開発環境**
  - LinuxとVMwarePlayer
- **実機動作確認環境**

| | |
|---|---|
| アーキテクチャ | PC/AT互換機 |
| CPU, メモリ | Intel Celeron 1.7GHz, 1GB |
| Perl処理系 | Perl (v5.10.0) |
| libc | newlib1.6 |
| ブートローダ | GNU GRUB |
| フロッピーディスク | 3.5インチ1.44MB |
| ハードディスク | LBAモード/UltraDMAモード2 |
| キーボード・マウス | PS/2接続 |



# 現在まで行ったPerlMachineの実装



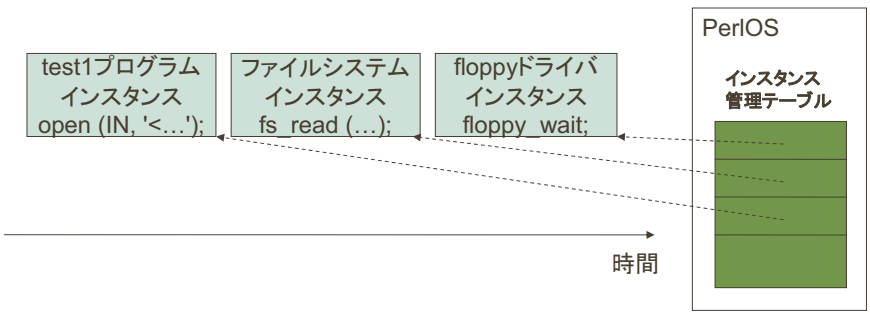
# 現在まで行ったPerlMachineの実装

- **ベアマシン上でのPerl処理系の実行**
  - Perl処理系の最新リリース版に対応
- **Perlで書かれた各種ドライバ**
  - 読込み専用ファイルシステムドライバ
    - VFAT12/16/32, ext2
  - 読込み専用フロッピーディスクドライバ
  - 読込み専用IDEディスクドライバ
  - 割込みコントローラドライバ
  - キーボード，マウスドライバ
  - VRAMドライバ



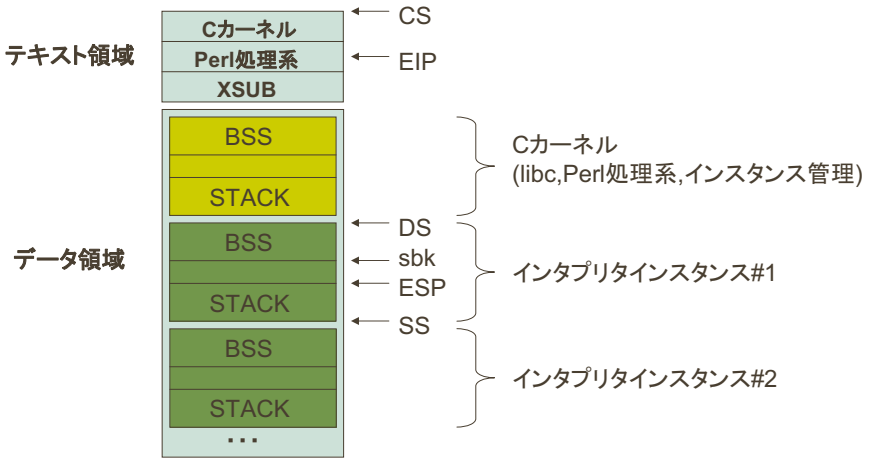
# 現在のプログラムの実行モデル

- **シングルタスク環境**
  - 単一のメモリ空間上で複数のインタプリタインスタンスがシングルタスクで動作
  - 各インスタンスはプログラムをシーケンシャルに実行



# メモリイメージの全容

- **メモリイメージ**
  - 単一メモリ空間上にインスタンスごとに論理的なセグメントを配置






# ベアマシン上のPerl処理系の実現

- **最新バージョン(5.10.0)に対応**

use 5.10.0

```
192.168.5.212 - Poderosa
ファイル(F)  編集(E)  コンソール(C)
改行 LF  エンコーディング utf-8
1 192.168.5.212
use 5.10.0;
$| = 1;
say 'say hello';
foreach my $num (1 .. 4)
{
   given ($num)
   {
      when (2)
      { say 'when1:2'; }
      when (4)
      { say 'when2:4'; }
      default
      { say "default:$num"; }
   }
}
```

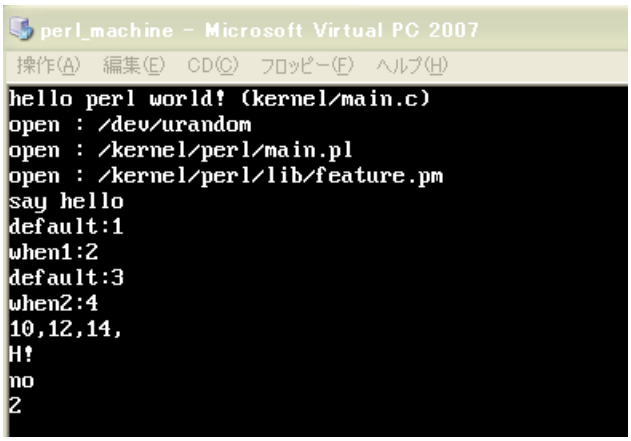
```
sub hoge
{
   state $num = 10;
   print "$num,";
   $num += 2;
};
&hoge,&hoge,&hoge,say;

my @alf = ('a' .. 'l');
if (@alf ~~ /h/)
{ say 'H!'; }
unless (@alf ~~ /z/)
{ say 'no'; }

my $aaa;
my $bbb = 2;
my $aaa = $aaa // $bbb;
say $aaa;
main.pl (1,1 先頭) [utf-8:unix]
```

# ベアマシン上のPerl処理系の実現

- **シミュレータ上での実行結果**






# I/O，実メモリ空間アクセスAPI

- **I/O空間へのアクセス**

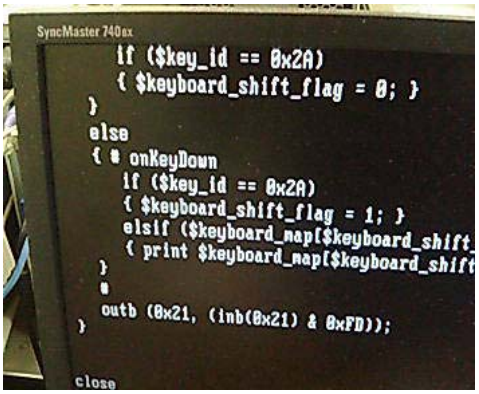
| 機能 | サブルーチン名 | 引数 | 返り値 |
|---|---|---|---|
| ポート番号から値を1バイト受信する | in[bwl] | ポート番号 | 値 |
| ポート番号に値を1バイト送信する | out[bwl] | 値、ポート番号 | なし |

- **実メモリ空間へのアクセス**

| 機能 | サブルーチン名 | 引数 | 返り値 |
|---|---|---|---|
| ベースアドレスから値の取得する | read[bwl] | ベースアドレス | 値 |
| ベースアドレスに値を1バイト格納する | write[bwl] | 値、ベースアドレス | なし |

# 割込み処理を伴う組込み関数の実行

- **組込み関数open・read・close・printの実行**
  - 各種ドライバ，ファイルシステムはPerlで実装




```
# 実行プログラム
$| = 1;
my $file_path =
'/kernel/perl/keyboard.pl';
print "open¥n";
open (IN, "<$path");
my @buf = <IN>;
print join ('', @buf);
close (IN);
print "close¥n";
```



# 組込み関数printの実行フロー

- **画面出力の場合（print "open¥n"）**
  - 処理系内部
    - pp_print関数
    - do_print関数
  - libc内部
    - printf関数
    - バッファ処理
    - libcのwrite関数
    - libcの_write関数

  APインスタンス

  - PerlOSのvramドライバ内部
    - vramインタプリタインスタンスの取得
    - vram_write_charサブルーチン
    - XSUBのwritew関数
      - ここでデバイスに文字の書込みを実行

  ドライバ
  インスタンス

# ファイル読込みの実行フロー

- **ファイル読込みの場合 ($buf = join ('', <IN>))**
  - 処理系内部
    - PerlIO_fread
  - libc内部
    - libcのfread関数
    - libcの_read関数

  APインスタンス

  - PerlOSのファイルシステムドライバ内部
    - fs_read関数
    - ファイルシステムインスタンスの取得
    - fat_readサブルーチン
      - FATを手繰ってファイル名からLBAの取得
    - floppy_readサブルーチン
      - LBAを元にディスク上のデータ取得
      - 読込み結果を_read関数で取得したポインタに書込み

  ドライバ
  インスタンス

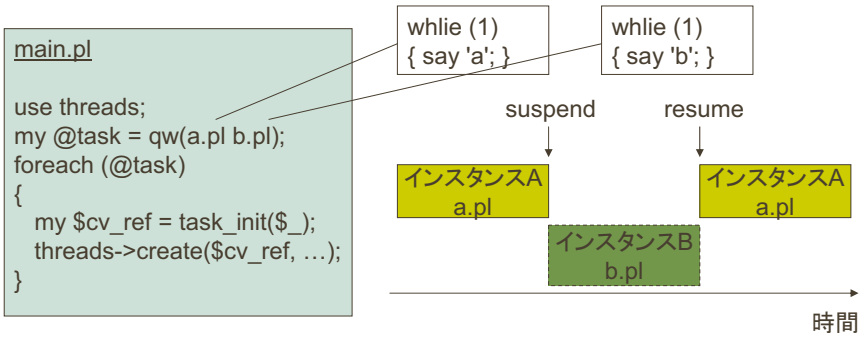


# まとめ

- **PerlMachine**
  - PC/AT互換機のベアマシン上にPerl処理系を搭載した高水準言語マシン
  - ユーザアプリケーションだけでなく，システムソフトウェアもPerlで記述される
- **PerlOS**
  - PerlMachine上で動作する，Perlで記述されたオペレーティングシステム
  - 1インタプリタインスタンスが1CPUにおけるプログラム実行の最小単位
- **各種ドライバの実装**
  - Perlで文字型・ブロック型デバイスのドライバや，ファイルシステムドライバが実現できることを示した

# 今後の予定

- **マルチタスク環境**
  - シングルコアにおけるマルチタスク環境
  - 複数インスタンスの並列実行
  - Perlによるマルチスレッドライブラリ(ithreads)を提供






# マルチタスク環境実現に向けて

- **実現までのステップ**
  - Cカーネルのマルチタスク対応
  - Perlのスレッドライブラリ(ithreads)が依存するnativeのpthreadライブラリの作成
    - libcのマルチスレッド対応
  - pthreadライブラリによるithreadsの実現
    - 並行実行
    - 共有変数
  - コンテキストスイッチを残し，その他処理をPerlで実現

- ご清聴ありがとうございました

# 携帯向けScheme処理系のSun SPOTへの応用

松崎 泰裕†
東京農工大学大学院情報工学専攻 †

## 1 背景と目的

筆者はこれまで携帯電話や計算機のJava VM上で稼動するScheme言語処理系YaScheme[1]を研究・開発してきた．本処理系は，Schemeの継続オブジェクトを転送することによって携帯電話間や計算機間でプログラムをマイグレーションできることを特徴としている．筆者は，この機能がセンサネットワークやロボットなどの組み込み機器にも有用であると考えた．

センサネットワークは携帯電話と同様に各ノードにCPUやストレージなどの資源があり，多くのノードが集まって処理を行うものである．プログラムのマイグレーションを用いることにより，データの集計や収集などの処理においてより柔軟な処理が可能となる．またロボットにおいても，複数のロボットが協調作業を行う場合や，仕事の譲渡をする際などにマイグレーションの活用ができると考えている．他にも，これら組み込み機器で計算を行う代わりに一旦より能力の高い計算機などへマイグレーションして計算を行うといった事例も考えられる．

本研究ではSun SPOTとMYUを用いてロボットを製作し，YaScheme処理系をSun SPOTへ移植し，センサネットワークやロボットへの応用を考える．

## 2 Sun SPOTとMYU

Sun SPOTはJavaプログラムが動作するセンサネットワークデバイスである[2]．GPIOが出力4ポート，汎用6ポート，アナログ入力4ポート搭載されている．一方MYUはロボット工作キットである[3]．3個のモーターとモータードライバを搭載しており，通常はそれらをPICから制御する．

本研究ではYaSchemeのロボットへの応用を考えるために，MYUをSun SPOTで制御できるように改造する．MYUのモータードライバには正回転と負回転の二つの入力があり，通常はそれぞれがPICの出力ポートへ接続されている．そこでMYUのPICを取り去り，モータードライバの入力とSun SPOTのGPIOの出力ポートを接続する．モータードライバとGPIOの電圧は共に3Vであるため，変換回路などは不要である．Sun SPOTのGPIOはJavaで制御可能であるため，これによりSun SPOTからのMYU制御が可能となる．製作したロボットの概観は図1のとおりである．

## 3 Sun SPOTへの対応

### 3.1 処理系コア

Sun SPOTのJavaプロファイルは携帯電話と同じJ2ME CLDC 1.1であり，基本的には携帯電話用の本処理系がそのままで動作する．しかし上位のプロファイルや，細かい制限などは携帯電話と異なるため，Sun SPOT用に修正を行う．

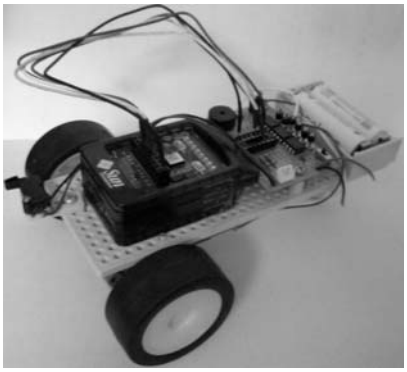


図 1: MyuSPOT 概観

### 3.2 Scheme手続き

本処理系で動くSchemeプログラムからSun SPOTやMYUの制御をできるように，表1のAPIを提供する．

通信はConnector-open手続きにプロトコルを指定して接続オブジェクトを作り，そこから入出力ポートを得てデータを送受信する．このポートはScheme標準手続きなどで読み書きできる．更にマイグレーションについてはmigrate-spot手続きを提供し，この手続き呼び出しの継続が転送される．

MYUのモーター制御をするために，GPIOのPWM制御をする手続きを提供する．その他にも，LEDの点灯や，各種センサの値を取得するための手続きを提供する．

### 3.3 マイグレーション

携帯電話は端末同士の直接通信ができず，またHTTPで自発的にしか通信ができないという大きな制限があった．しかしSun SPOTはノード間でZigBeeにより直接通信が可能である．そこでマイグレーションの通信処理をSun SPOT用に改変する．

Sun SPOTではZigBee上に実装されたRadiogramとRadiostreamというプロトコルが利用できる．RadiogramはUDPのようにパケットを指定した宛先へ送信やブロードキャストをできるが，データサイズは1KB程度までである．一方Radiostreamは任意の量のデータを送受信できるが，TCPとは異なり両者が通信先を指定する必要がある．そこでマイグレーションを行う際には，図2のように，まずRadiogramでマイグレーション要求を送ることでアドレスを通知し，次にRadiostreamで実際のマイグレーションデータを転送する．

携帯電話では通信の制限により一つの端末で同時に複数のAPが動くことは考えていなかった．しかしSun SPOTでは自由に通信できるため，あるAPの実行中に他のマイグレーション要求が来る場合がある．そのような場合に対応するために，マイグレーション専用のスレッドで受信処理をし，継続を受け取った場合には新しくスレッドを作成してその継続を起動することで，一つのノードで複数のAPを同時に処理できるよ

---

Scheme Interpreter for Sun SPOT

† Yasuhiro Matsuzaki
Department of Computer and Information Sciences, Tokyo University of Agriculture and Technology

表 1: 手続き一覧

| 手続き名 | 説明 |
|---|---|
| dgcon-receive | Radiogram パケット受信 |
| dgcon-openInPort | Radiogram データ読込 |
| dgcon-openOutPort | Radiogram パケット作成 |
| dgcon-send | Radiogram パケット送信 |
| strcon-openInPort | Radiostream 受信 |
| strcon-openOutPort | Radiostream 送信 |
| migrate-spot | マイグレーション |
| set-pwm | GPIO を PWM 制御 |
| front | MYU 前進 |
| back | MYU 後退 |
| left | MYU 左旋回 |
| right | MYU 右旋回 |
| set-LED | LED を点灯/消灯 |
| set-LED-color | LED 色を指定 |
| get-tilt-x | X 軸加速度センサ取得 |
| get-tilt-y | Y 軸加速度センサ取得 |
| get-tilt-z | Z 軸加速度センサ取得 |
| : | |

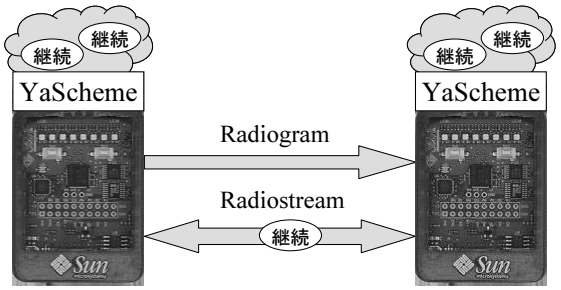



図 2: マイグレーション

うにする．

本処理系は携帯電話や計算機などでも動作するため，Sun SPOT とそれら計算機の間でも通信やマイグレーションの利用が考えられる．一般の計算機は ZigBee プロトコルを扱うことができないが，Sun SPOT を USB で接続して Basestation にすることで，計算機上で動く本処理系から ZigBee による通信ができるようになる．一方携帯電話は HTTP による通信にしか対応していないため，HTTP によりサーバーの CGI や Servlet を起動し，サーバーが ZigBee により通信をする．

## 4 評価

Sun SPOT の実機上で本処理系を動作させ，メモリ使用量を測定した．測定直前に Scheme 上から Java の GC を起動し，その後に全メモリ容量と使用量を取得することでメモリ使用量を算出した．尚，Sun SPOT の搭載するメモリは 512 KB，Java VM から使用可能な量は 459 KB である．Java VM 起動直後のメモリ使用量は約 102 KB，本処理系の起動直後は約 133 KB，長さ 1000 のリストを作成した時の消費メモリは約 12 KB となった．本処理系が消費している分は約 30 KB であり，本処理系の上で稼動するアプリケーションの余地が十分残っていると言える．

表 2: 実行速度

| | Sun SPOT | | PC(3GHz) |
|---|---|---|---|
| コード | 本処理系 | Java | 本処理系 |
| tak 18 12 6 | 127s | 259ms | 246ms |
| 単純ループ | 1.24ms | - | 2.14$\mu$s |
| 関数付ループ | 1.56ms | 3.52$\mu$s | 2.79$\mu$s |
| 継続ループ | 1.73ms | - | 3.39$\mu$s |
| do 構文ループ | 1.50ms | 2.73$\mu$s | 2.59$\mu$s |

次に Sun SPOT の実記上で実行速度を測定した．参考のために，Sun SPOT 上の Java と，PC 上の本処理系についても測定した．結果は表 2 のとおりとなった．ループは 1 ループあたりである．結果を見ると，複雑な計算には時間がかかっているが，センサネットワークで行われるような比較的簡単な計算や，簡単なロボットの制御には問題ない処理速度であると言える．

更にマイグレーションにかかる時間についても測定した．Sun SPOT 間で同期しているタイマは無いため，二つの Sun SPOT 間をマイグレーションで往復する時間を測定した．結果は平均で 2479ms となり，片道はこの半分の約 1375ms である．この速度は，多くのノードをマイグレーションして回る場合や，計算機間を頻繁に往復をするには難しい速度である．片道の時間の内訳を調べたところ，約 700ms がシリアライズ処理，残りが通信処理であった．シリアライズ処理はアルゴリズムの改良によりまだ高速化の余地があると考えている．また通信処理は，初期化に多くの時間がかかっており，同じ計算機への通信オブジェクトのキャッシュをすることで往復の高速化が可能である．

## 5 まとめと今後の課題

本論文では，マイグレーションに対応した携帯電話向け Scheme 言語処理系の，Sun SPOT と MYU を用いたロボットへの応用について述べた．現在までに Sun SPOT 上での Scheme プログラムの実行と，Sun SPOT や MYU の制御，Sun SPOT 間でのマイグレーションを実現した．しかしマイグレーションに時間がかかっており，改善の余地があるため，マイグレーション機能の改善が今後の課題である．またブロードキャスト通信による継続のマイグレーションや，端末を越えた束縛を持つ部分的なマイグレーションなど，更なる機能の考察も今後の課題である．

## 参考文献

[1] 松崎泰裕，並木美太郎: プログラムをマイグレーションできる携帯電話向けの Scheme 言語処理系，情報処理学会「コンピュータシステムシンポジウム 2007」論文集，Vol.2007, No.14, pp.59-68 (2007).

[2] Sun Microsystems: Sun Small Programmable Object Technology, http://www.sunspotworld.com/ (2007).

[3] Studio MYU: ミュウロボ, http://studiomyu.com/ (2008).

# 携帯向けScheme処理系の Sun SPOTへの応用

JSASS 2008年09月02日

東京農工大学 工学府 情報工学専攻
松崎 泰裕
並木 美太郎

## 背景(1/2)

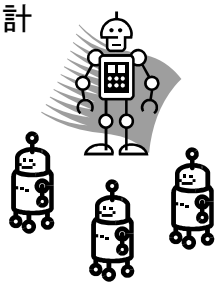
Yet another? Ya suhiro?
YaScheme

- 携帯電話向けのScheme処理系を開発
  - 計算機間の継続マイグレーション対応が特徴
- センサネットワークやロボット
  - 何台かが協調作業する計算機
  - 本処理系が有用・応用可能

2

# 背景(2/2)

- 例えば
  - センサネットワーク
    - マイグレーションしながら各所の温度等データを検索や取捨選択しながら集計
    - 死の瀬戸際に仕事譲渡
  - ロボット
    - 協調作業などで仕事を継続で譲渡
    - 外出先から家のロボットを操作
    - ~~地球から火星のロボットを操作~~？



3

# 目的

- SunSPOTとMYUでロボットを製作
- 処理系YaSchemeをSunSPOTへ移植
- センサネットワークやロボットへの応用を考える
  - マイグレーション
  - その他
    - RPCなど

4

# SunSPOTとは？

- センサネットワークデバイス
- ARM180MHz RAM512KB Flash4MB
- 3軸加速度/温度/照度センサ
- LEDx8 無線ZigBee　GPIOピン
- Javaが動く！(J2ME CLDC 1.1)

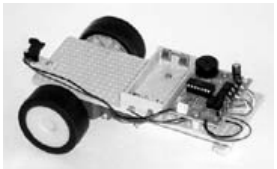
→YaSchemeを比較的容易に移植可能

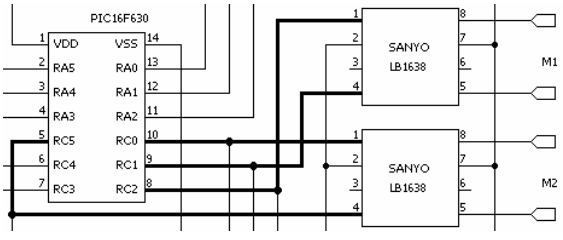
※他機器ではMindStormsも試したが、JavaVMの制限により移植困難

5

# MYUとは？

- ロボット工作キット
- PICで制御
- 3個のモーターとドライバ
- スイッチ１個　入力ピン数個

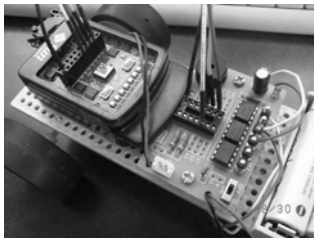





6

## MYU+SunSPOT

- 幸い両方とも電圧が3V
- PICを取り去り、SunSPOTのGPIOと接続
  - GPIOは出力x4、汎用x6、アナログ入力x4
  - 出力ピンをモータードライバの入力ピンへ
- SunSPOTからMYUを制御



7

## SunSPOTへの処理系移植(1/2)

- 基本的にそのままで動作可能
  - 携帯と同じJ2ME+CLDC
  - 環境依存の起動クラスやI/Oのみ変更
- SunSPOT用Scheme手続き
- GPIOのための手続き
  - モーター用PWM制御
    - (set-pwm pin value)
    - (define (front) (set-pwm 0 255)
      (set-pwm 2 255))

8

# SunSPOTへの処理系移植(2/2)

- ZigBeeのための手続き
  - J2ME準拠APIのラップ
    - (let* ((cnct (Connector-open "radiogram://10AB"))
      (out (dgcon-openOutputPort cnct))
      (write '(1 2 3) out)
      (dgcon-send cnct))
  - マイグレーション用
    - (migrate-spot "10AB")
- その他の手続き
    - (set-LED-color 2 #x00FF00) (set-LED 2 #t)
    - (get-tilt-x)

9

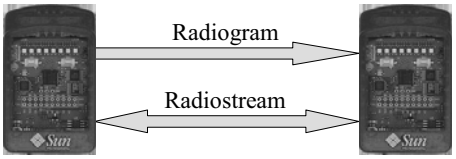
# マイグレーションへの対応(1/2)

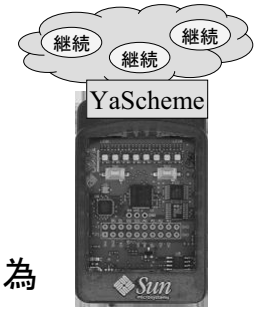
- SunSPOT間通信はZigBee
  - JavaからDatagramとStreamが利用可能
- Radiogram
  - 宛先指定、Broadcastにも対応
  - パケットサイズは1KB程度まで
- Radiostream
  - 両者が通信先とポートを合わせて通信

10

# マイグレーションへの対応(2/2)

- まずRadiogramでマイグレーション要求
- 次にRadiostreamで継続データを転送



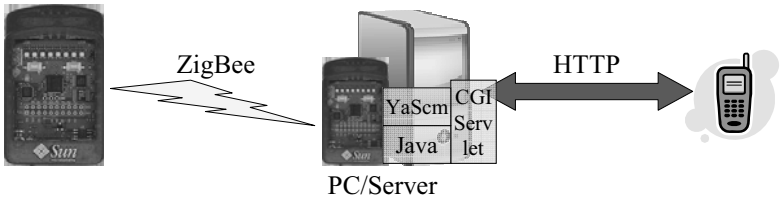




- 複数の継続を受け入れ
  実行することに対応
  - 携帯のように通信制限が無い為

11

# PC等他の機器との通信

- 1つのSPOTをPCへUSBで繋ぎBaseStation化
  - PC上のJavaからZigBeeを利用可能
- 携帯はHTTPからCGIやServletで起動




12

# メモリ使用量

- SunSPOT搭載メモリ 512KB
- JavaVMが利用可能な量 459KB

| | 使用中メモリ量 |
|---|---|
| JavaVM起動直後 | 102KB |
| YaScheme起動直後 | 133KB |
| 1000個のセル作成時 | +12KB |

- 処理系の使用量は30KB程度
- アプリケーションの余地は十分

13

# 処理速度(1/2)

※ループは1ループあたり

| コード | SunSPOT 本処理系 | SunSPOT Java | PC (3GHz) 本処理系 |
|---|---|---|---|
| (tak 18 12 6) | 127[s] | 259[ms] | 246[ms] |
| 再帰ループ | 1.24 [ms] | - | 2.14[μs] |
| 関数付ループ | 1.56 [ms] | 3.52 [μs] | 2.79[μs] |
| 継続ループ | 1.73[ms] | - | 3.39[μs] |
| do構文ループ | 1.50[ms] | 2.73 [μs] | 2.59[μs] |

- リアルタイム制御は難しい性能

14

# マイグレーション速度(2/2)

- マイグレーションにかかる時間
  - 往復にかかる時間を測定

```
(time
  (begin
    (migrate-spot Spot-Sub)
    (migrate-spot Spot-Main)))
```

- 平均 2749 ms → 片道 約1375 ms
  - 内 約700 msがシリアライズ処理(片道)
  - 残りが通信処理

15

# 今後の課題

- I/O中のマイグレーションなどの考慮
- ブロードキャスト
- 部分的なマイグレーションや
  端末を越えた束縛
- RPC
- セキュリティ

17

# リアルタイム仮想化ソフトウェア基盤におけるタイマ割込み通知機構

金城 聖 † 永島 力 † 毛利 公一 ††
† 立命館大学大学院理工学研究科 †† 立命館大学情報理工学部

## 1 はじめに

近年，携帯電話や家電製品，車載システムなどに利用されている組込みシステムは，従来の組込みシステムと異なり，応答性や信頼性だけでなく，機能性も求められるようになった．

従来の組込みシステムでは，ITRON や VxWorks などに代表されるリアルタイム OS (以下，RT-OS と記す) が利用され．機器制御を主要処理とした最小限の機能提供によって応答性と信頼性を保証してきた．そのため機能性の確保は考慮されていない．また，Windows や Linux などの高機能 OS では，複雑な情報処理を主要処理としている．そのため，RT-OS のような十分な応答性や信頼性は保証されていない．その理由としては，複雑な情報処理を目的とする機能は構造が複雑になり，処理の予測性が低下するからである．

以上のことから，単一 OS での応答性と信頼性の確保と機能性の実現はトレードオフの関係にあるといえる．このトレードオフを解消するために，我々は，RT-OS と高機能 OS を同時に運用可能とするリアルタイム仮想化ソフトウェア基盤 (以下，RT-VMM と記す) の研究・開発を行っている．RT-OS と高機能 OS を 1 台の計算機上で共存動作させることで信頼性・応答性の確保を RT-OS で行い，機能性の確保を高機能 OS で行うことが可能となる．

RT-VMM では，RT-OS が動作するゲスト計算機 (RT ドメイン) に対して安定したタイマ割込みを通知することが重要な課題となる．本稿では，RT-VMM における RT ドメインに対して安定したタイマ割込みを通知する手法について提案する．

## 2 OS 共存手法の比較

既存の OS 共存手法には，ハイブリッド方式 [1][2] と仮想化方式の 2 種類がある．仮想化方式は，さらに仮想化の方法によって区別可能で，ソフトウェア仮想化方式とハードウェア支援方式の 2 種類に分けられる．

- ハイブリッド方式
  マルチコアプロセッサ上のプロセッサコア毎に異なる OS を動作させることでを OS の共存動作を実現する方式である．I/O デバイスの管理については，デバイス毎の固定割り当てによって実現している．そのため I/O 処理は単体の OS を動作させた場合と遜色なく実行することが可能となる．しかし，搭載可能な OS の数はプロセッサコア数によって制限される．また，デバイスの共有ができないことや，一つの OS が利用可能な I/O デバイス数が，単体の OS を動作させた場合よりも少数に限られることがある．

- ソフトウェア仮想化方式
  ソフトウェア仮想化方式は，VMM を利用した方式である．ゲスト OS は仮想計算機 (以下，VM と記す) 上で動作する．ゲスト OS から利用可能な計算機資源は仮想化されるため，他のゲスト OS への動作に与える影響が小さい．そのため，資源利用の点で安全な動作環境が保証されている．しかし，I/O 処理や割込みが全て仮想化されるため，オーバヘッドや割込み周期のジッタが発生するなどの課題がある．

- ハードウェア支援方式
  ハードウェア支援方式は，ソフトウェア仮想化方式のうち，I/O デバイスの仮想化をハードウェアによる仮想化支援機能を利用する方式である．ハードウェア支援機能を利用することで，ソフトウェア方式より高速な I/O 処理が可能となる．しかし，支援機構を有するハードウェアが少ないため移植性が低い．また，現状では組込み機器への対応が想定されていないなどの利用環境面での問題がある．

RT-VMM では，RT-OS の特徴である信頼性・応答性と高機能 OS の特徴である機能性を同時に保証することを目的とする．また，各 OS の干渉による影響を最小限に抑える必要がある．そのため，本稿で提案する手法では，安全な動作環境が利用でき，独立した VM が構築可能であるソフトウェア仮想化方式を基礎技術に利用する．

## 3 リアルタイム仮想化ソフトウェア基盤

RT-VMM は，RT-OS と高機能 OS の共存動作を実現するシステムソフトウェアである．図 1 に示すように，RT-VMM の構成要素は，RT-OS が動作する VM である RT ドメイン，高機能 OS が動作する VM であるゲストドメイン，ドメイン管理用のインタフェースを提供する特権ドメイン，さらに，ドメインに対して仮想化された計算機資源を提供し，実計算機資源を隠蔽する RT ハイパーバイザから構成される．

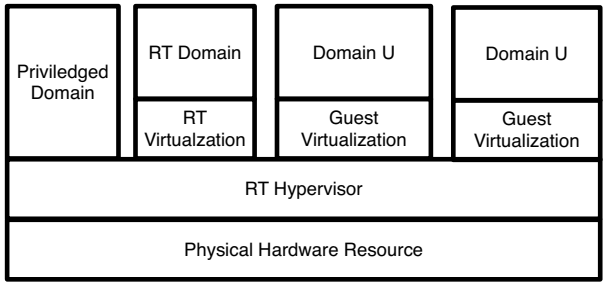



図 1　リアルタイム仮想化ソフトウェア基盤

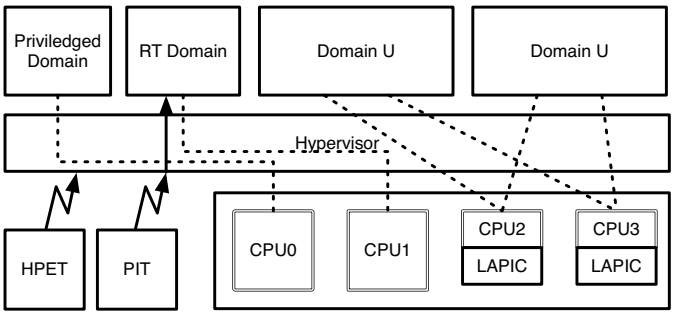



図 2　システム構成

RT ドメインでは，RT-OS の動作に特化した VM を実現する必要がある．RT-OS の動作では，タイマ割込みの周期が安定していることが重要となる．そのため RT ドメインでは安定したタイマ割込みを保証する必要がある．

本ソフトウェアは，既存の VMM である Xen [3] に改良を加えることで実現する．開発の効率の点から，既存の VMM を利用することで既存の資産を容易に利用することが可能となる．

## 4　Xen におけるタイマ仮想化手法

### 4.1　タイマ割込み通知機構の概要

Xen が管理するタイマデバイスには，プラットフォームタイマとローカルタイマがある．プラットフォームタイマには，PIT，HPET，ACPI PMT などの時間管理デバイスが利用され，ローカルタイマにはプロセッサコア毎に搭載されているローカル APIC(以下，LAPIC と記す) が利用される．

Xen 上で動作するゲスト OS へのタイマ割込みは，仮想タイマ割込みによって実現されており，LAPIC タイマ割込みを契機に利用している．Xen の内部では，LAPIC タイマ割込みを受信すると，LAPIC タイマ割込みハンドラ内で，遅延処理を実現するソフト割込みが生成され，以後の処理をソフト割込みハンドラ内で行うように実装されている．また，その処理は時限タイマというヒープ構造で定義されたテーブルで管理されており，LAPIC タイマ割込みが発生する毎に，ソフト割込みハンドラ内で時限タイマを探索し，期限切れの処理を実行する．仮想タイマ割込みをゲスト OS に通知する処理は，この時限タイマに登録されている．

### 4.2　RT-VMM 実現における課題

仮想タイマ割込みは LAPIC タイマ割込みを契機とし，ソフト割込みを利用した遅延処理で実現されている．そのため，ソフト割込みによる仮想タイマ割込み通知の遅延と，ヒープ探索による仮想タイマ割込み遅延が発生する．

以上のことから，Xen では仮想タイマ割込みの通知周期にジッタが発生することがわかる．RT-VMM では，RT ドメインに対して安定した周期でタイマ割込みを通知することが重要である．そのため，タイマ割込み周期のジッタ発生を防止することが必要である．

## 5　提案手法

### 5.1　概要

仮想タイマ割込み周期におけるジッタの発生を防止するために，未使用タイマデバイスを利用する．図 2 に示すように，未使用タイマデバイスが生成するタイマ割込みを RT ドメインに通知することで，仮想化による周期のジッタを解消することが可能となる．また，プロセッサを RT ドメイン対して固定的に割り付けることで，ドメインのスケジューリングによるタイマ割込みの通知遅延も防ぐことが可能となる．ゲストドメインに対しては，Xen が標準で提供する LAPIC 由来の仮想タイマ割込みを利用する．上記のような構成を実現することで，RT-OS と高機能 OS の共存が可能である．

### 5.2　機構

本手法を実現するためには，以下に示す機構を Xen に追加実装する必要がある．

**RT ドメイン用タイマ初期化機構**　RT ドメイン対して割り付けられているプロセッサコアにタイマ割込みを通知するようにタイマデバイスを初期化する．

**RT ドメイン用タイマ割込みハンドラ**　初期化機構によって設定されたタイマデバイスからのタイマ割込みを処理する．このハンドラ内で現在稼働中の RT ドメインに対してタイマ割込みを通知する機構を呼び出す．

**RT ドメイン用タイマ割込み通知機構**　ハイパーバイザから RT ドメイン上のゲスト OS に対してタイマ割込みを通知する．

RT ドメイン用タイマ割込みの発生から RT ドメインに通知されるまでの処理の流れは次のようになる．

1. RT ドメイン用に初期化されたタイマがタイマ割込みを生成する．
2. RT ドメイン用タイマ割込みハンドラが起動し，タイマ割込み受信を確認．
3. その後に RT ドメイン用タイマ割込み通知機構を経由して RT ドメインにタイマ割込みを通知する．

### 5.3 評価方法

本手法の評価方法として，次の場合におけるタイマ割込み周期の計測を検討している．

- 実機上 RT-OS と RT ドメイン上 RT-OS の比較
  実機と RT ドメインの比較によって，ハイパーバイザを仲介することによる遅延を計測する．
- RT ドメイン上 RT-OS と Xen 標準のゲストドメイン上で動作する RT-OS の比較
  専用タイマデバイスを利用する RT ドメインと Xen 標準のソフト割込みを利用したゲストドメインにおけるタイマ割込み周期のジッタを計測する．
- ゲスト OS 起動下での上記 2 パターンの計測
  RT ドメイン以外のゲストドメイン上でゲスト OS が動作しているときの RT ドメインの負荷耐性を評価する

## 6 おわりに

本稿では，RT-OS と高機能 OS の共存によって，信頼性・応答性・機能性を実現する RT-VMM を提案し，そのタイマ管理手法について述べた．今後の課題としては，上記手法の実装と評価，さらに RT ドメイン上で動作するプロセスの実時間処理の保証があげられる．

## 参考文献


[1] 遠藤 幸典, 菅井 尚人, 山口 義一, 近藤 弘郁: “シングルチップマルチプロセッサ上のハイブリッド OS 環境の実現–システムアーキテクチャ–,” 第 66 回情報処理学会全国大会講演論文集, Vol.1, pp. 9–10 (2004).

[2] 菅井 尚人, 遠藤 幸典, 山口 義一, 近藤 弘郁: “シングルチップマルチプロセッサ上のハイブリッド OS 環境の実現–OS 間インタフェースの実装–,” 第 66 回情報処理学会全国大会講演論文集, Vol.1, pp. 11–12 (2004).

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield:“Xen and the Art of Virtualization,” In Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, pp. 164–177 (2003).

# リアルタイム仮想化ソフトウェア基盤におけるタイマ割込み通知機構

Joint Symposium for Advanced System Software [JSASS] 2008
立命館大学大学院 理工学研究科
毛利研究室
◎金城 聖, 永島 力, 毛利 公一

## 発表内容

- はじめに
- OS共存手法の比較
- リアルタイム仮想化ソフトウェア基盤
- 既存の仮想計算機モニタのタイマ仮想化
  - 既存手法の概要
  - RTドメイン実現における課題
- タイマ割込み通知機構
  - 機構
  - 評価方法
- 今後の課題
- おわりに

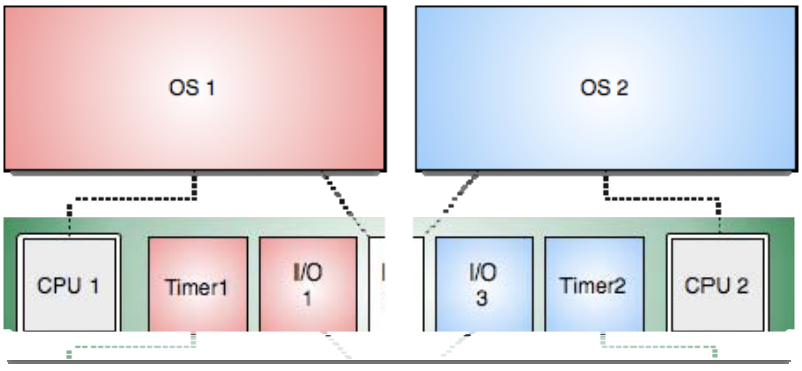
# はじめに

- 組込みシステムにおいて高い機能性の要求が増大
  - 携帯電話・車載システム・情報家電など
- 従来の組込みシステムへの要求は高い応答性
  →RT-OS(iTRON, VxWorks)を一般的に利用
- 機能性への要求の実現
  →高い応答性を保証しつつ，機能性を実現することが必要
- 解決策
  - RT-OSに対しての機能追加が必要
    →機能追加による構造の複雑化
    →リアルタイム性能の低下
  - OS共存による機能性の実現
    - RT-OSと高機能OSの共存
    - RT-OSで応答性を保証，高機能OSで機能性を保証



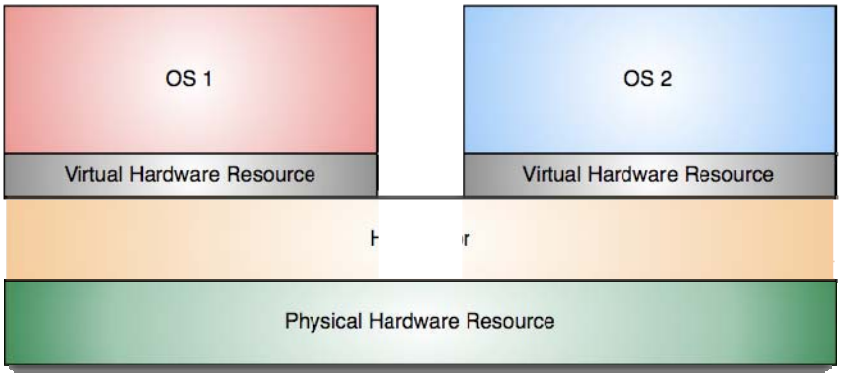
# OS共存手法(1/4)：ハイブリッド方式

- マルチコアプロセッサ環境上で実現
  - プロセッサコアごとに異なるOSが動作
- 共有と分割による資源管理



# OS共存手法(2/4)：仮想化方式

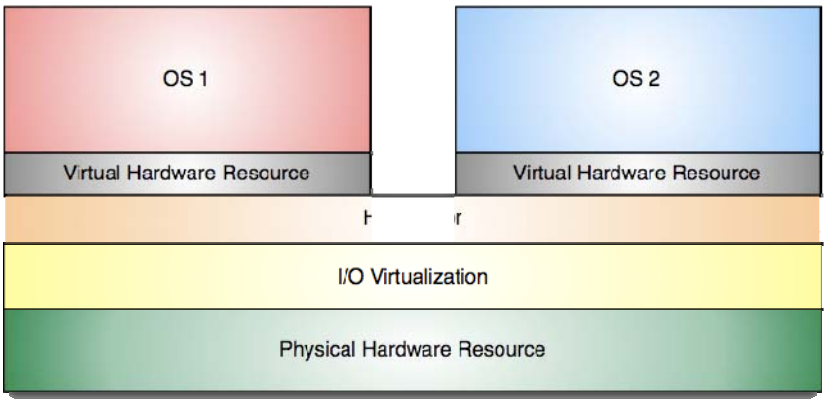
- 仮想計算機モニタ(VMM)を利用したOS共存方式
- 実計算機資源の隠蔽
- VMMによる計算機資源の仮想化






# OS共存手法(3/4)：ハードウェア支援方式

- ソフトウェアによる仮想化とハードウェアによるI/O仮想化支援を利用した方式



# OS共存手法(4/4)

- 方式の比較

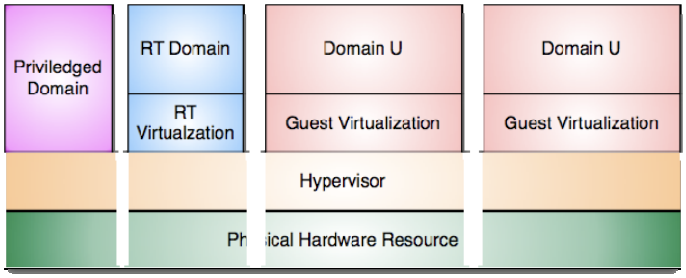
| 方式 | メリット | デメリット |
|---|---|---|
| ハイブリッド方式 | 資源分割固定割り当てによりI/O処理が高速 | プロセッサ数による制限<br>OSの対応が必要<br>利用可能資源が少ない |
| 仮想化方式 | 他OSへの動作の影響が小さい | I/O仮想化処理が低速<br>タイマ仮想化による割込み遅延 |
| ハードウェア支援方式 | 他OSへの動作の影響が小さい<br>I/O仮想化処理が高速 | 対応ハードが少ない<br>タイマ仮想化による割込み遅延 |

- 本研究では，仮想化方式が提供する安全な動作環境を利用し，RT仮想化ソフトウェア基盤を実現する．



# RT仮想化ソフトウェア基盤

- RT-OSと高機能OSをVMM上で動作
  - RTドメイン
    - RT-OSが動作
    - 仮想CPUと物理CPUの対応付けを固定
  - ゲストドメイン
    - LinuxやWindowsなどの高機能OSが動作
- RT-OS用の仮想計算機環境を提供
  - 安定したタイマ割込みの保証が必要
- Xenをベースに実現



# RT仮想化ソフトウェア基盤

- RT-OSと高機能OSをVMM上で動作
  - RTドメイン
    - RT-OSが動作
    - 仮想CPUと物理CPUの対応付けを固定
  - ゲストドメイン
    - LinuxやWindowsなどの高機能OSが動作
- RT-OS用の仮想計算機環境を提供
  - 安定したタイマ割込みの保証が必要
- Xenをベースに実現

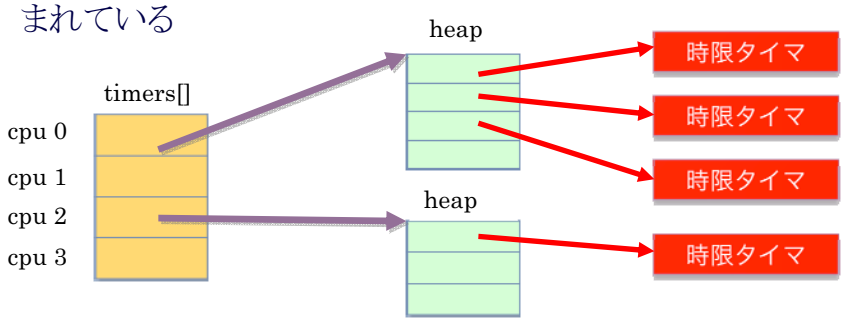



# Xenが管理するタイマ

- Linuxの割込みと遅延処理(ソフト割込み)に類似
- Xenが利用するタイマデバイスは2種類
  - プラットフォームタイマ
    - PIT, HPET, ACPI PM Timerのいずれか1つが利用可能
    - デフォルトではHPETを利用
  - ローカルタイマ
    - CPUのローカルAPIC(LAPIC)タイマを利用する
- 割込みハンドラ
  - プラットフォームタイマ割込みハンドラ
    - 起動時からの経過時刻の計測に利用
  - ローカルタイマ割込みハンドラ
    - 時限タイマの起動を行うためのタイマソフト割込みを要求する

# 時限タイマ

- 時限タイマの内容
  - 時限処理を実行する期限
  - 実行する処理
- 時限タイマの探索にはヒープ探索が利用されている
- 時限処理には，ゲストOSへのタイマ割込みに関する処理が含まれている






# 割込みハンドラ

- プラットフォームタイマ割込みハンドラ
  - jiffies変数を1ずつ進める
    - jiffies変数：起動時からのプラットフォームタイマ割込みの回数を記録する変数
- ローカルタイマ割込みハンドラ
  - タイマに関するソフト割込み(タイマソフト割込み)を要求
- タイマソフト割込みハンドラ
  - 上で要求されたタイマソフト割込みを処理する
  - 時限タイマの期限をチェックし，期限切れの処理を実行する

## ゲストOSへの周期タイマ割込みの通知

- ゲストOSが受信可能な割込み
  - 実ハードウェアから転送された割込み
  - Xenが生成した仮想割込み
- 仮想割込みを利用したタイマ割込み(仮想タイマ割込み)の通知
  - 時限タイマ処理に仮想タイマ割込み処理が登録されている

- ゲストOSに仮想タイマ割込みを通知する時限タイマはタイマソフト割込みハンドラ内で実行される
- 仮想タイマ割込みの契機には，ローカルAPICタイマ割込みが利用される



## 既存のタイマ割込み通知手法の課題

- 仮想タイマ割込みの通知には，ソフト割込み(遅延処理)が利用されている

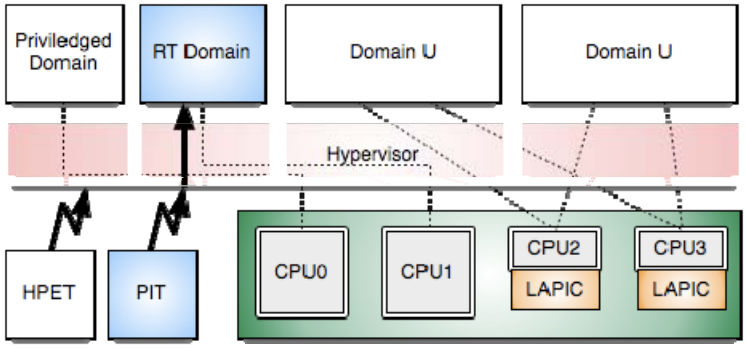
  →通知の遅延が発生する
- 時限タイマの実行にヒープ探索が利用されており，期限の早いものから順に実行する

  →探索による遅延が発生する

- タイマ割込みの周期に揺らぎが発生する
- リアルタイムドメインへの安定したタイマ割込みが困難

# PIT割込み通知機構(1/3)

- 未使用のハードウェアタイマを利用することで実現
  - PITのタイマ割込みをRTドメイン用のタイマ割込みとして利用
  - RTドメインへのプロセッサ割当てを固定
- 非RTドメインは現状のLAPICタイマとソフト割込みを利用



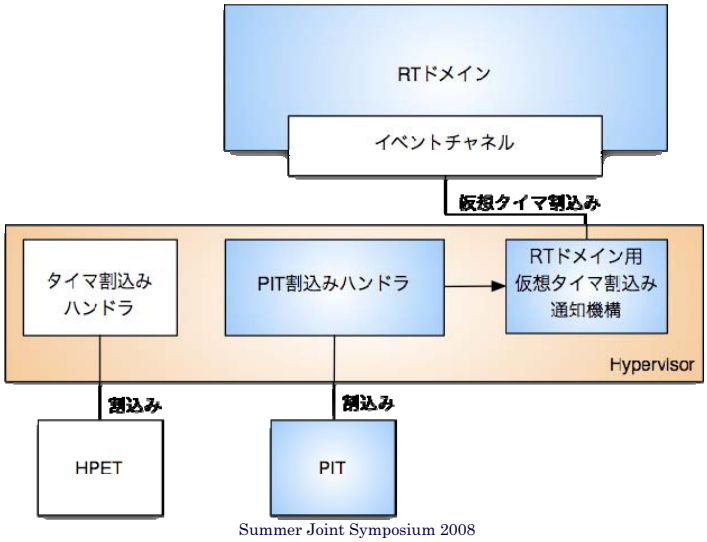



# PIT割込み通知機構(2/3)

- PIT初期化機構
  - RTドメインが使用するCPUに割込みを通知するようにPITを初期化
- PIT割込みハンドラ
  - VMMのプラットフォームタイマ割込みとは異なるハンドラ
  - RTドメインに仮想タイマ割込みを通知する
- RTドメイン用仮想タイマ割込み通知機構
  - RTドメインに対してのみPIT割込み契機の仮想タイマ割込みを通知する機構

# PIT割込み通知機構(3/3)

- ソフトウェア構成






# 解決手法の評価方法

- タイマ割込みハンドラの評価
  - 実機上RT-OSとRTドメイン上RT-OSの比較
    - ハイパーバイザを仲介することによるオーバヘッドの計測
  - RTドメイン上RT-OSとゲストドメイン上RT-OSの比較
    - 提案手法と既存手法におけるタイマ割込み間隔のジッタを計測
  - ゲストOS起動下での上記2パターンの計測
    - 他のゲストOSが動作しているときのRTドメインの負荷耐性の評価
- 評価環境
  - RT-OSにはTOPPERS/JSP for IA-32を利用
  - 準仮想化を利用
- 評価用機構
  - ゲストOS用評価用PIT値取得ハイパーバイザコール
  - PIT値取得機構

## 今後の課題

- PIT割込み通知機構の設計
  - PIT初期化機構
  - PIT割込みハンドラ
    - プラットフォームタイマ割込みの処理に影響を及ぼさないようにする
  - PIT用仮想タイマ割込み通知機構
- 評価用PIT値取得機構の設計
  - PIT値取得ハイパーバイザコール
  - PIT値取得機構
- 上記機構の実装，評価



## おわりに

- 組み込みシステムにおける高機能化への要求増大

- RT-OSと高機能OSの共存による実現
  →RT仮想化ソフトウェア基盤の提案
- 安定したタイマ割込み通知機構
  - RTドメイン
  - 未使用のタイマデバイスを利用→PITを利用
- 今後の課題
  - 機構の設計，実装，評価

# 既存仮想計算機モニタのリアルタイム性に関する基礎性能評価と考察

永島 力† 金城 聖† 毛利 公一††
† 立命館大学大学院理工学研究科 †† 立命館大学情報理工学部

## 1 はじめに

近年，車載システムや携帯電話，情報家電などの組込みシステムには，これら製品を制御する機能に加えて，さまざまな機能が追加されている．一方で，組込みシステムには高いリアルタイム性能が組込みシステムには要求されている．これら，高い機能性の実現と高いリアルタイム性をもった処理の実行には背反する関係があり，両方を同時に実現することは出来ない．このため，組込みシステムにおいては，リアルタイムな処理を実行する計算機と高機能性を実現する計算機を複数搭載して実現しているケースがあり，新たな機能の追加による OS の追加によって拡大する製品のサイズやコストの増加などが問題となっている．この問題に対して，リアルタイム性能が求められる処理を行うリアルタイム OS (以下，RT-OS と記す) と増加する付加機能の処理を行う汎用 OS といった複数の OS を仮想化技術を用いて統合することで解決する方法が考えられている．しかし，従来の仮想計算機モニタ (以下，VMM と記す) では，ゲスト OS に対して仮想化された計算機資源を提供するため，ゲスト OS が実行する処理時間に大きな遅延や，遅延のゆらぎが発生する．そのため，リアルタイム性の保証が困難である．

このような背景より，我々は，RT-OS と汎用 OS が同時に動作可能であるリアルタイム仮想化ソフトウェア基盤 (以下，RT-VMM と記す) の開発を行っている．以下，本稿では RT-VMM を実現するにあたって，既存の VMM がもつリアルタイム性能に関する計測と評価について述べる．

## 2 Xen 上で動作する Linux のリアルタイム性に関する性能評価

Xen[1] 上で動作する OS のリアルタイム性について計測・評価するにあたって，Linux を用いて Xen がゲスト OS に対してどの程度のリアルタイム性能を提供しているのかを計測・評価した．

### 2.1 性能評価手法

性能評価に利用した評価環境には，表 1，表 2 に示しているように，OS に Linux(2.6.23.8) を，CPU は Core 2 Duo T5600 を利用した．ゲスト OS は，Virtual Machine Manager を利用して VM 上で構築し，OS にはドメイン 0 と同様に Linux(2.6.23.8) を利用した．

表 1 実計算機上の Linux 性能評価環境

| 評価環境 | Linux |
|---|---|
| OS | Linux 2.6.23.9 |
| CPU | Core 2 Duo T5600(シングルモード) |

表 2 仮想計算機上の Linux 性能評価環境

| 評価環境 | Linux on VM |
|---|---|
| ホスト OS | Linux 2.6.23.9 |
| CPU | Core 2 Duo T5600(シングルモード) |
| ゲスト OS | Linux 2.6.23.9 |
| CPU | Core 2 Duo T5600(シングルモード) |

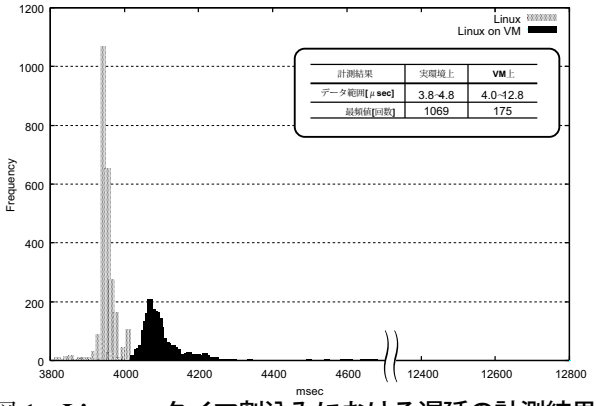



図 1 Linux タイマ割込みにおける遅延の計測結果

評価対象としてタイマ割込みハンドラの遅延と周期について計測した．ハードウェア割込みが発生した時点のクロックとタイマ割込みハンドラが開始された時点でのクロックを出力し，それをもとにタイマ割込みの遅延と周期について計測した．具体的には，ハードウェア割込みに対して，do_IRQ 関数が起動した際にタイマ割込みが発生したとしてクロックを取得し配列に格納する．次に，handle_IRQ_event によってタイマ割込みサービスルーチンである timer_interrupt が呼び出された際にタイマ割込み処理が開始されたとしてクロックを取得し格納する．この動作によって得られたクロックが一定の量になると出力し，前述のような処理を行うことでタイマ割込みの遅延と周期を計測している．

### 2.2 考察

今回の計測によって得られた計測結果を図 1，2 に示す．図 1 より，実環境上で動作している OS のタイマ割込みにおける遅延と比べ，VM 上で動作している OS の

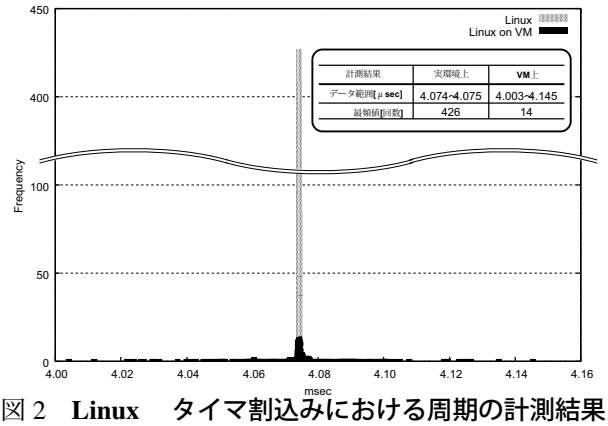



図 2　**Linux　タイマ割込みにおける周期の計測結果**

タイマ割込みの遅延値は大きく，ゆらぎも大きいことがわかる．つぎに，図 2 に示すように，実環境上で動作している OS のタイマ割込みにおける周期と比べ，VM 上で動作している OS のタイマ割込みの周期についてゆらぎの幅も大きく，タイマ割込みの周期が不安定であることがわかる．また，VM 上で動作している OS の方が実環境上で動作している OS より周期が短い場合もあるが，これはタイマ割込は一定の間隔で発生するため，前の周期が長くなった場合には次のタイマ割込み処理までの周期が短くなるためであると考えられる．

これらのことから，VM 上で OS が安定して一定の間隔で処理を実行するのは困難であり，また，周期の不安定さによる正確なスケジューリングも困難であると考えられる．

## 3　実環境上で動作する TOPPERS のリアルタイム性に関する計測

前述の Linux による Xen 上で動作する OS のリアルタイム性に関する性能評価によって，Xen 上で動作する OS に対して提供される割込みには実環境上で動作するものに比べて大きなゆらぎが生じていることが分かった．

以下では，Xen 上で動作している RT-OS についてどのようなゆらぎが生じるか計測・評価するために行った実環境上での TOPPERS[2] のリアルタイム性に関する計測について述べる．

### 3.1　性能計測

性能評価に利用した評価環境には，表 5 に示しているように，RT-OS に TOPPERS(jsp-1.4.2) を，PIT を用いて計測した．また，CPU については Core 2 Duo T5600 を利用した．具体的な計測方法としては，PIT が発生させた割込みに対し，タイマハンドラが開始した時点の PIT 値を取得し，PIT の最大値から取得した値との差分を計算することで遅延を計測した．

表 3　**実計算機上の TOPPERS 性能評価環境**

| 評価環境 | TOPPERS |
|---|---|
| OS | jsp-1.4.2 (i386 上で動作) |
| CPU | Core 2 Duo T5600 |
| PIT(AT 互換機) | 1.19318MHz |

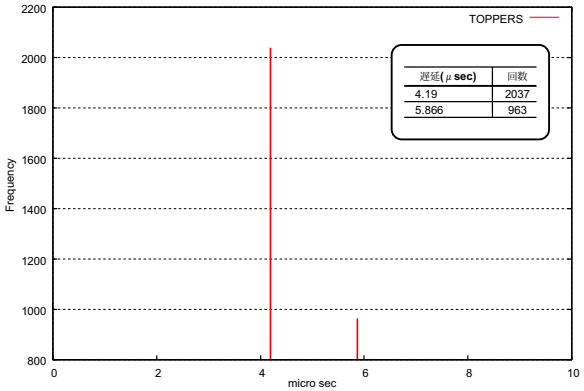



図 3　**実環境上で動作する TOPPERS のタイマ割込み処理における遅延**

### 3.2　考察

今回行った計測の結果を図 3 に示す．実環境上で動作する TOPPERS におけるタイマ割込み処理の遅延は，ほぼ一定でゆらぎが少ないことが分かった．これによって実環境上で動作する TOPPERS のタイマ割込み処理の周期にもゆらぎが少なく安定していると考えられる．

## 4　おわりに

本稿では，組込みシステムにおける近年の開発事情を背景とした，多機能化とリアルタイム性能の問題について述べ，我々が開発する RT-仮想化ソフトウェア基盤についてその大まかな構成と携帯端末への適用例を示した．そして，RT-仮想化ソフトウェア基盤を実現するにあたって，既存 VMM が VM 上の OS に提供するリアルタイム性能を既存の VMM 上における Linux のリアルタイム基礎性能について，また，実環境上で動作する TOPPERS の計測結果について述べた．

今後，VM 上で動作する TOPPERS の計測について計測を行い，Xen 上で動作する RT-OS の評価を行い，タスク遷移や，タスクの起床などによる計測・評価を行っていく予定である．

## 参考文献

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield : Xen and the Art of Virtualization, ACM Symposium on Operating Systems Principles, pp. 164–177, 2003.

[2] TOPPERS プロジェクト, http://www.toppers.jp/

www.asl.cs.ritsumei.ac.jp

# 既存仮想計算機モニタのリアルタイム性に関する基礎性能評価と考察

立命館大学　理工学研究科
毛利研究室
○永島 力，金城 聖，毛利公一

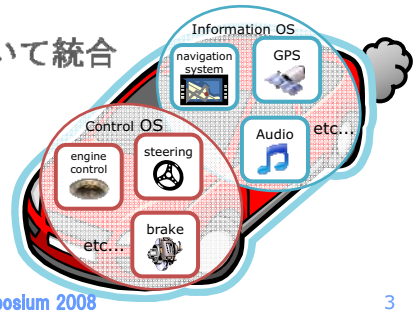
www.asl.cs.ritsumei.ac.jp

## 目次

- はじめに
- RT-仮想化ソフトウェア基盤
- 携帯端末への適用例
- 研究内容について
- Linuxにおける計測
- TOPPERSにおける計測
- おわりに

# はじめに

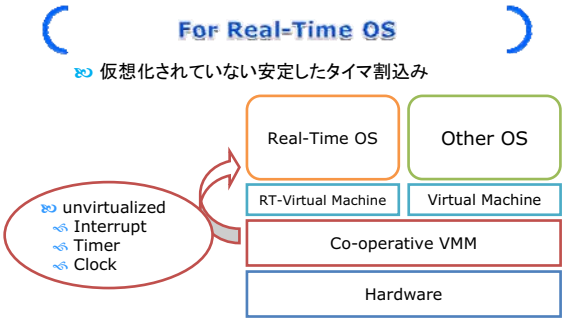
- 組込みシステムにさまざまな機能が追加されている
  - 車載システム，携帯電話，情報家電システム，etc…
- 一方で…
  - 高いリアルタイム性能が組込みシステムには求められる
  - 製品のサイズやコストの増加を防がなくてはいけない
- 複数のOSを仮想計算機を用いて統合
  - RT-OSと汎用OS
  - 資源を有効利用





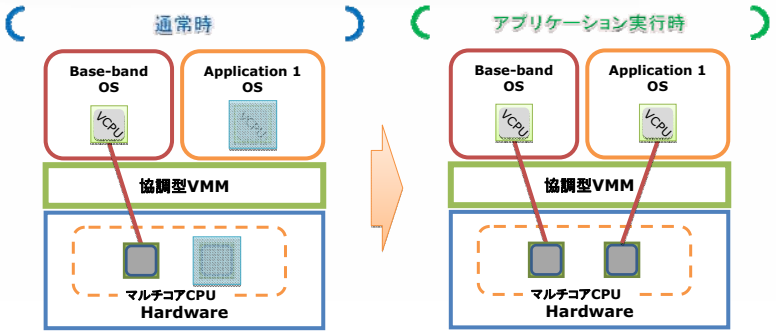

# RT-仮想化ソフトウェア基盤

- RT-OSと汎用OSがVMM上で動作可能
  - RTドメイン
    - 実CPUと物理CPUを固定で割当て
    - RT-OSが動作
  - ゲストドメイン
    - 多機能な汎用OSが動作



# 携帯端末への適用例

- 通話機能に加え，様々な機能が付加されている
  - 消費電力の増大
  - 製品サイズの拡大
- 複数のOSを統合
  - 余過剰分の資源を停止することで消費電力の低減




www.asl.cs.ritsumei.ac.jp



# 研究内容

- RT-仮想化ソフトウェア基盤の実現
- 既存のVMMがどの程度のリアルタイム性能をゲストOSに提供できるのか計測
- 計測対象
  - Linux
  - TOPPERS

www.asl.cs.ritsumei.ac.jp

www.asl.cs.ritsumei.ac.jp

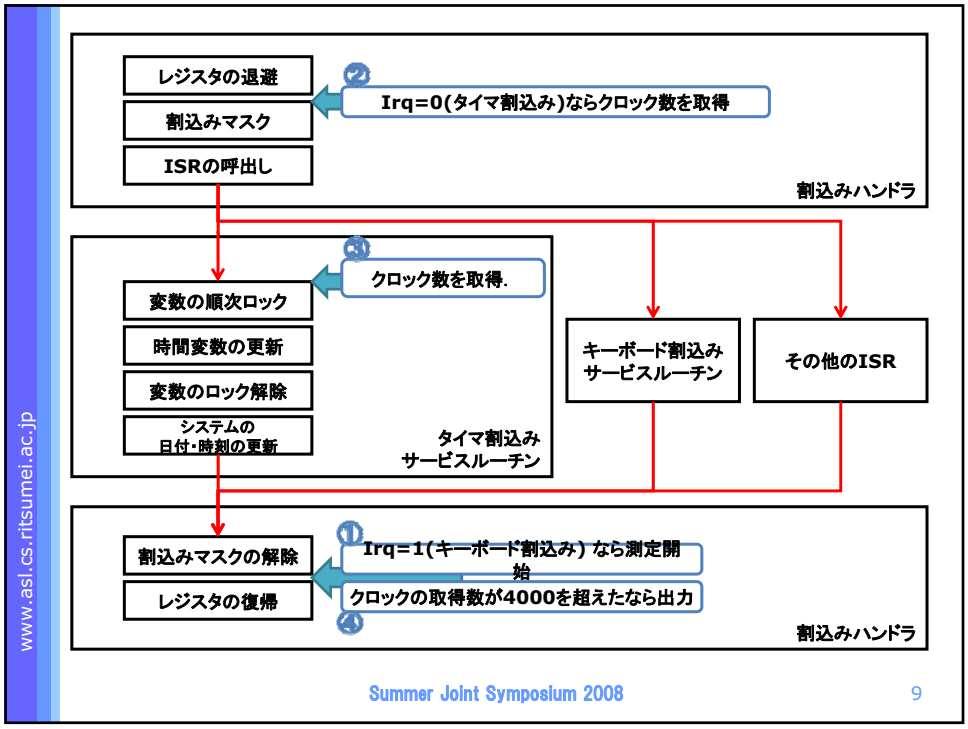
# LINUXにおける
# 割込みハンドラの計測



www.asl.cs.ritsumei.ac.jp

# 性能評価手法

- タイマ割込みについて計測（2,000回）
  - 処理要求が発生してからどれだけの遅延があるか
  - タイマ割込み処理は優先で処理される
- 遅延
  1. 割込みハンドラ起動時のクロックを計測
  2. 割込みサービスルーチン(ISR)実行開始時のクロックを計測
  3. 差分を取ることで遅延を算出
- 周期
  1. タイマISR実行時のクロックを計測
  2. 計測結果をもとに周期を算出








# 評価環境

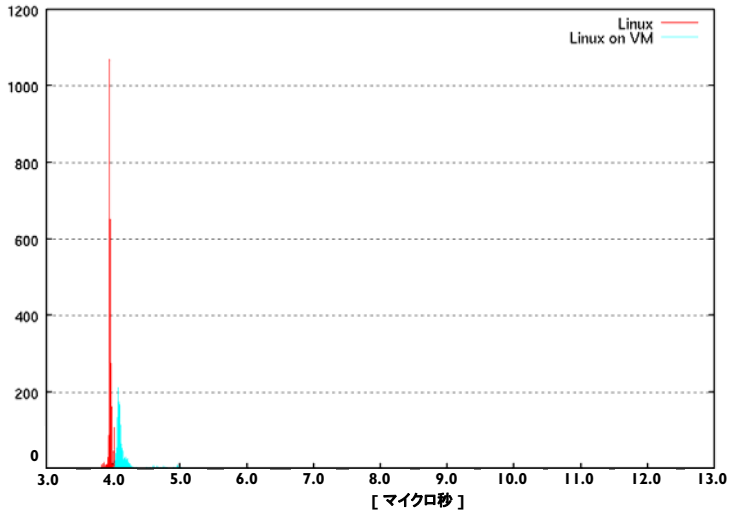
Linux

- OS
  - Linux 2.6.23.9
  - シングルユーザモード
- CPU
  - Core 2 Duo 1.83 GHz
  - シングルコアで動作

VM上のLinux

- OS
  - ホストOS
    - Linux 2.6.21-6.fc7xen
  - ゲストOS
    - Linux 2.6.23.9
    - シングルユーザモード
- CPU
  - Core 2 Duo 1.83GHz

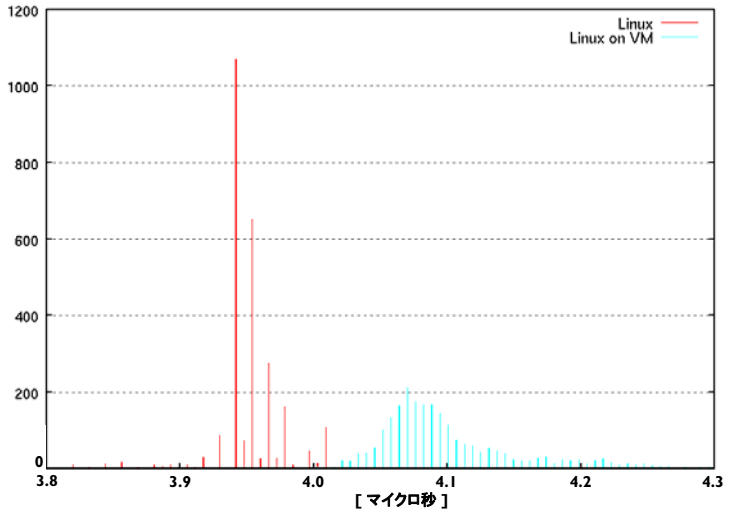
# タイマ割込みハンドラにおける遅延




www.asl.cs.ritsumei.ac.jp

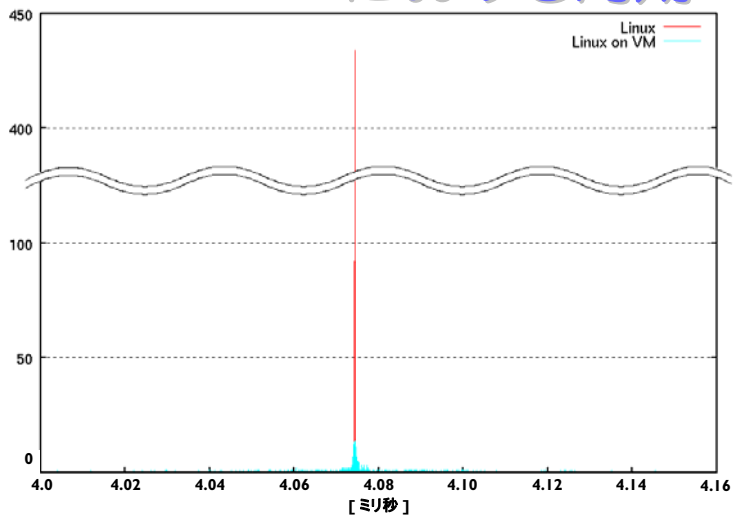


# 密集区間における タイマ割込みハンドラの遅延




www.asl.cs.ritsumei.ac.jp

# タイマ割込みハンドラにおける周期






# 考察

- Linuxにおける遅延
  - VM上では実環境上と比べ，遅延の値が増加
  - VM上では実環境上と比べ，ばらつきの幅が広範囲
- 周期
  - 実環境上と比べ，VM上では割込み周期が広範囲に分散
- 予想されるVM上におけるリアルタイムOSの動作
  - 安定して，一定の間隔での処理を行うことが困難
  - 周期の不安定さによる正確なスケジューリングが困難

www.asl.cs.ritsumei.ac.jp

# TOPPERSにおける タイマ割込みハンドラの計測



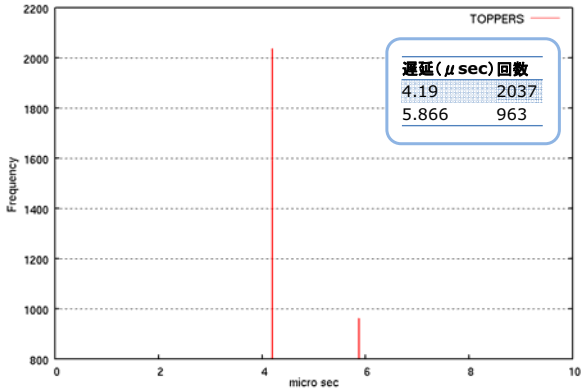
www.asl.cs.ritsumei.ac.jp

# 計測環境

- 計測環境
  - i386で動作
  - AT互換機
    - PITの割込み周期：1.19318MHz
    - PITのカウンタ値の最大値：1193
- 計測方法（3,000回）
  - タイマハンドラの開始時点でPITの値を計測
  - PITの最大値と取得値との差分を取ることで遅延を計測

# タイマ割込みハンドラの遅延

- ほぼ，一定の安定した遅延
  - ➥安定した割込み周期








# おわりに

- 本発表では
  - 組込みシステムにおいてさまざまな機能が追加
  - RT-仮想化ソフトウェア基盤
  - 既存VMM上におけるLinuxのRT基礎性能の計測・評価
    - VM上において，タイマ割込みハンドラの遅延と周期に大幅なばらつきが生じた
    - VMMがゲストOSに安定したタイマ割込みの提供ができていない
  - 既存VMMにおけるTOPPERSのRT基礎性能の計測・評価
    - 遅延が安定している
      - ➥タイマ割込みの周期も安定
- 今後の予定
  - TOPPERSの計測
    - Xen上でのタイマ割込みについて
    - タスク遷移やタスクの起床にかかる遅延時間

# VoXY：Webブラウザ上での仮想計算機システム

高橋　一志　†

# VoXY : Virtual Computer System on a Web-browser

Kazushi Takahashi †

## 1. はじめに

VDI:Virtual Desktop Infrastructureとは，サーバ側にある計算機上のデスクトップ環境を，ネットワークを通じてクライアント側へ転送するインフラストラクチャのことである．VDIにより転送されたデスクトップは仮想デスクトップと呼ばれる．VDIと呼ばれる技術には，古典的なthin clientや，後述するWebVDIなどがあげられる．

thin clientは，”組織に所属する人々”といった限定的な人々に向かって計算機資源を提供するシステムとしては枯れた技術である．このことは，多くの組織への導入実績や，多くのthin client製品が開発されていることからも見て取れる．

一方，WebShakaが開発したYouOS[3]はWebブラウザ上で仮想デスクトップが動作するシステムである．これはデスクトップ環境を転送するthin clientとは異なり，Webアプリケーションとしてまったく新しく実装されたGUIアプリケーションである．われわれは，このようにWebブラウザ上で動作する仮想デスクトップを新たにWebVDIと定義する．

この二つの技術は一見すると似ている．しかし，これらは大きく異なる．現状，thin clientの運用形態は，プライベートなネットワーク内で組織に所属する従業員が使用するといった，クローズドな運用に限定されている．これは，専用クライアントやアプライアンスのインストール，特殊なプロトコルを使用するにあたっての通信インフラの整備が必要であることに起因する．一方，WebVDIは使う人間を限定しない．クライアントはデフォルトでインストールされているWebブラウザのみで使用できる．プロトコルにはよく利用されるHTTPのみを使用しているため，WebVDIは新たにソフトウェアを導入せずとも利用が可能である．

先に挙げたVDIの現状について二つの着目すべき点がある．一点目は，既存のWebVDIが提供する仮想デスクトップは，GUIのWebアプリケーションという形態を取っているため，既存のアプリケーションの再利用が不可能である点．二点目として，VDIの代表格であるthin clientは，専用クライアントの導入が必要であり，手軽に使用できない点である．

われわれは上記の着眼点を元に，既存のアプリケーションが再利用可能な，新たなWebVDIを構築する．また，最終的な目標として，ユーザへは独立した一つの計算機環境を提供できる一方，サービスの提供基盤たるサーバの安全性は最大限に確保できるWebVDIを目指している．

このWebVDIを実現するため，われわれは仮想マシンモニタ(VMM)とVNCに着目し，VMM上の仮想計算機の画面データをWebブラウザ上に転送することのできるVNC proxy．VoXYを開発した．

## 2. VoXY（Vnc Over http proXY）

VoXYのシステム概念図を**図1**に示す．VoXYは大きく分けて，VMMの管理やユーザ管理などを行うVoXY管理サーバと，Webブラウザ上で動作する，JavaScriptで実装されたVoXY client，VoXY clientとVoXY管理サーバを接続するための，VoXY cgi-scriptの3つのコンポーネントに分けられる．仮想マシンの画面データ転送には，VNCで利用されているRFB protocolを使用する．仮想マシンの画面データをRFB protocol[1]として出力できるVMMは多い．代表的なものにXen,VMware[2]などが挙げられる．

WebVDIという中央集権的なアーキテクチャを考慮し，VoXYは以下の特長を有している．VoXYはマルチユーザに対応しており．複数人のユーザそれぞれに

† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

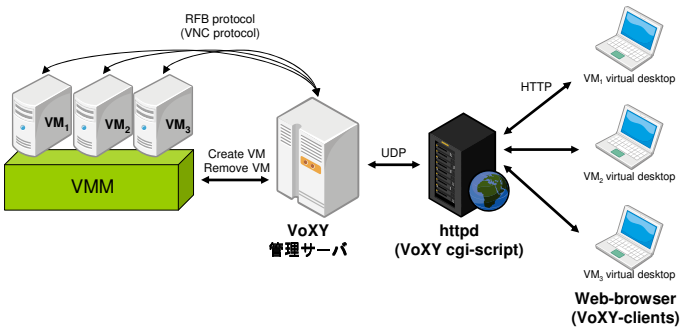



**図 1** VoXY アーキテクチャの概念図
Fig. 1 VoXY architecture

1 台の仮想マシンを割り当てたり，複数人のユーザに対して 1 台の仮想マシンを共有させることが可能である．httpd は cgi-script の実行に対応している httpd であればどれでも使用することができる．

### 2.1 VoXY 管理サーバ

VoXY 管理サーバは，VMM を管理する VoXY VMM Manager と，VoXY Window Manager の 2 つのコンポーネントに分けられる．それぞれのコンポーネントの詳細を以下に示す．

**VoXY VMM Manager** VoXY VMM Manager は VMM に対して，仮想マシンの起動，新規作成，削除などの命令を発行する．仮想マシンの画面データは RFB protocol で遣り取りを行う．VMM に対する，仮想マシン自体の制御☆には VMI を使用する．これにより，多くの VMM を統一したインターフェイスで操作することが可能である．

**VoXY Window Manager** VoXY Window Manager は主に，ユーザの管理と画面データの管理を行う部分である．VoXY VMM Manager が管理する仮想マシンに対してユーザを割り当てたり，画面データの取得などを行う．

### 2.2 VoXY cgi-script

VoXY cgi-script は，httpd 上で動作する VoXY client と VoXY 管理サーバ間の中継を行うための CGI スクリプトである．通信は UDP を使って独自のプロトコルで行う．

### 2.3 VoXY client

VoXY client は Web ブラウザで動作するクライアントである．これは，仮想マシンの作成，削除といった VMM のインターフェイスと，仮想マシンの画面データの表示や，仮想マシンに対して送信する Keyboard や Mouse のイベント取得などを行う．通信は Ajax を用いて，非同期に httpd 上にある VoXY cgi-script へとデータを送る．

### 2.4 画面描画手法

仮想計算機の画面全体は VoXY 管理サーバによって，適当な大きさを持つ複数の正方形ブロックに分割して管理される．VoXY 管理サーバはブロック内の一ドットでも変更されると，そのブロック全体をサーバの HDD に書き出す．VoXY client は，VoXY 管理サーバとの通信によって変更されたブロックのファイル名を取得して，IMG タグを変更する．その後，Web ブラウザ側がブロックのファイルをダウンロードし，Web ブラウザに表示される．

☆ パワーオン，リセット，仮想マシンの作成や削除といった操作．

## 3. 評　　価

評価は，ドローソフトとワープロソフトを利用し，VoXY-client が httpd から画面データをダウンロードして，実際に画面が更新されるまでの時間を測定した．結果，ドローソフトウェアがデータ全体の 99%が 50ms 以内の遅延時間で収まっており，ワープロソフトはほぼ 100%が 50ms 以内の遅延時間で画面更新を行えていることがわかった．

## 4. おわりに

本稿では，VMM を制御する機能と，VNC のプロトコルを HTTP に変換する二つの機能を持つ VoXY を開発した．VoXY が目指すものは，ユーザが Web ブラウザ上から，柔軟に計算機資源を借り出し．そのまま利用できるような WebVDI を構築することである．

今後の課題は，VoXY が VMM をコントロールするためのインターフェイスへの対応，VoXY 管理サーバのユーザ数に対するスケーラビリティの測定やなどが挙げられる．

## 参　考　文　献

1) Richardson, T.: The RFB Protocol, http://www.realvnc.com/docs/rfbproto.pdf (2007).
2) VMware, Inc.: *VMware*. http://www.vmware.com/.
3) WebShaka, I.: YouOS, https://www.youos.com/ (2006).

# VoXY：Webブラウザ上での仮想計算機システム

高橋一志*

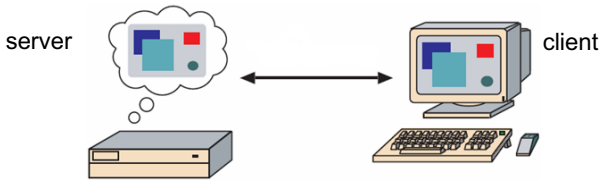
*東京大学大学院情報理工学系研究科

# 発表の流れ

- 研究の背景
- VoXY（Vnc Over proXY)
  - VoXYコンポーネントの紹介
  - Webブラウザにおける画面転送方式
  - VoXYプロトコルの紹介
- 評価
  - 評価手法
  - LISTENあたりの画面更新遅延時間
  - LISTENあたりのデータ転送量平均値
  - 評価のまとめ
- まとめ
  - 既存研究
  - 今後の課題
  - まとめ

# 背景
## ~ VDI：Virtual Desktop Infrastructure ~

- デスクトップをネットワークを通して転送する技術
  ➔ 再現されたデスクトップを“仮想デスクトップ”と定義
- 例）Thin-Client
  – X, Sun Ray, Windows Terminal, VNC
  – 情報漏洩対策，省エネルギーの観点から注目

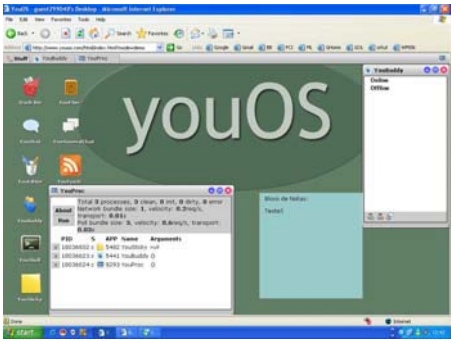



仮想デスクトップのイメージ図

# 背景
## ~ WebVDI ~

例) Thin-Client以外のVDI
  – YouOS
  – Webアプリケーション

- YouOS (WebShaka, Inc.)
  – Window Managerライクな Webアプリケーション
  – 類似システムは多数存在




※Webブラウザ上で“仮想デスクトップ”が動作する

**Web上で動作するVDI ➔ WebVDIと定義**

## 既存VDIの分析

- 既存のThin-Client
  - 利点：既存のソフトウェア資産が流用可能
  - 欠点：導入が困難
    - 専用の機器が必要
    - HTTPしか通さないネットワークでは使用が不可能
- 既存のWebVDI
  - 利点：導入が容易
    - Webブラウザ上で動作
    - HTTPしか通さないネットワークでも使用可能
  - 欠点：既存ソフトウェア資産の利用が不可能
    - JavaScriptで書き直す必要がある

## 本研究の目的

- 新たなWebVDIの提案

➔既存システムの特長を継承したシステム

- WebVDIの特長
  - Webブラウザ上で動作する（導入が容易）
- Thin-Clientの特長
  - 既存のソフトウェア資産を利用できる

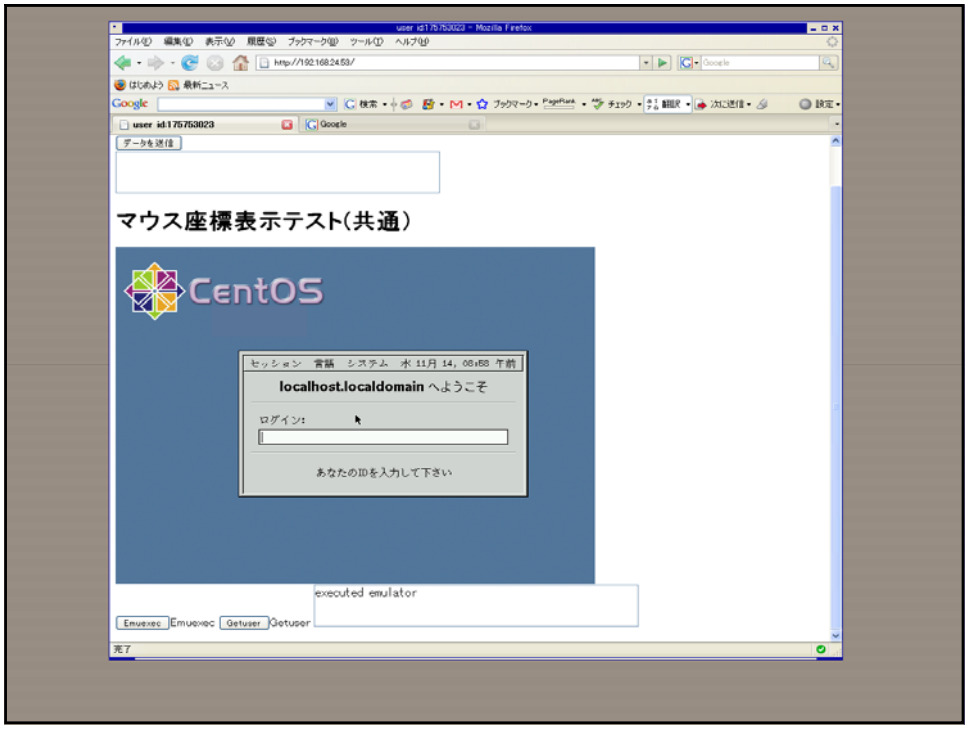
➔具体的な目標

- クライアントはWebブラウザを使用
  - JavaScriptのみを使用し，プロトコルにはHTTPを使用
- 画面転送方式での実現
  - デスクトップをそのままWebブラウザ上に転送

# 本研究のアプローチ

既存のThin-Clientプロトコルを
HTTPに変換するProxy

- VoXY：VNC proxy の開発
  - VNCプロトコルをHTTPに変換
- VNCを選んだ理由
  - 既存のThin-Clientプロトコルはいくつかあるけど…
  - 例）X, ICA, RDP, VNC, SLIM
  - ➔ VNC：Virtual Network Computing
    - サポートするプラットフォーム多数
      - RFBプロトコルとして仕様が完全に公開
    - 近年ではVMMでのKeyboard Video Mouseプロトコルとして使用

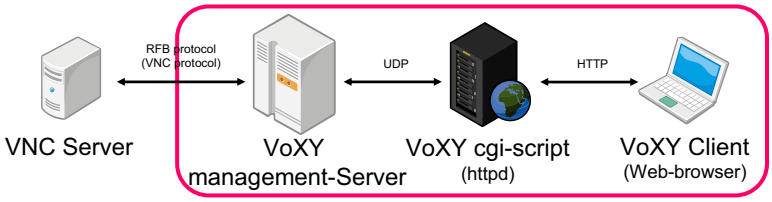


# 残りの発表の流れ

- 研究の背景
- VoXY（Vnc Over proXY）
  - VoXYコンポーネントの紹介
  - Webブラウザにおける画面転送方式
  - VoXYプロトコルの紹介
- 評価
  - 評価手法
  - LISTENあたりの画面更新遅延時間
  - LISTENあたりのデータ転送量平均値
  - 評価のまとめ
- まとめ
  - 既存研究
  - 今後の課題
  - まとめ

# VoXY（Vnc Over http proXY）

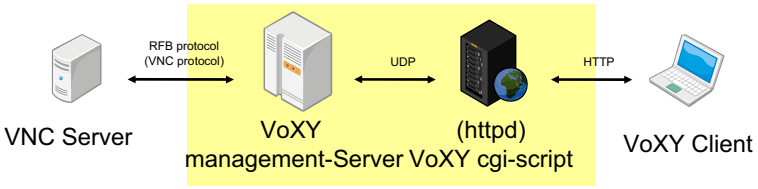
- VoXYシステム概念図




- VoXYの特徴
  - RFB protocolをHTTPに変換
    - クライアントにWebブラウザを使用可能
  - マルチユーザ，マルチサーバに対応
    - ➔ 複数のユーザに対応
  - 既存のhttpdが使用できる
    - ➔ 定評のあるhttpdに組み込むことが可能
  - management-server，VNC server，httpdを分離している
    - ➔ スケーラブルなシステム構築が可能

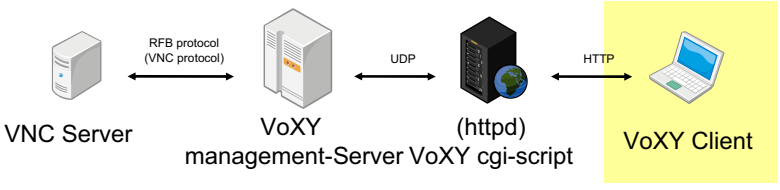
# VoXYコンポーネント(1/2)

- VoXY management-server
  - ユーザ管理と，VNC server管理
  - cgi-scriptとの通信
  - 実装はC++で4,000行ほど
- VoXY cgi-script
  - VoXY clientとmanagement-serverの中継
  - 実装はCommon Lispで400行ほど (clisp使用)



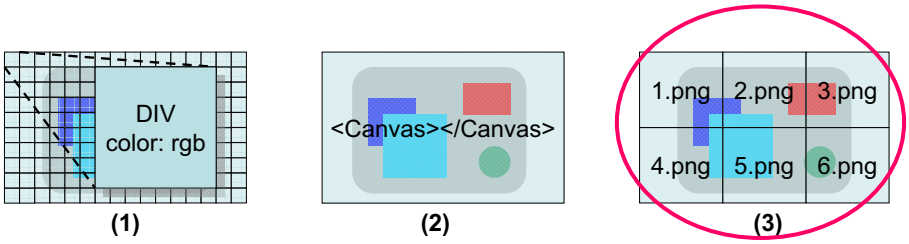

# VoXYコンポーネント(2/2)

- VoXY client
  - ユーザへのインターフェイス提供
    - 画面表示，ユーザイベント取得，cgi-scriptとの通信
  - コンパクトな実装 （JavaScriptで400行ほど）
  - 対応するWebブラウザ
    - FireFox，Internet Explorer，Opera，Safari

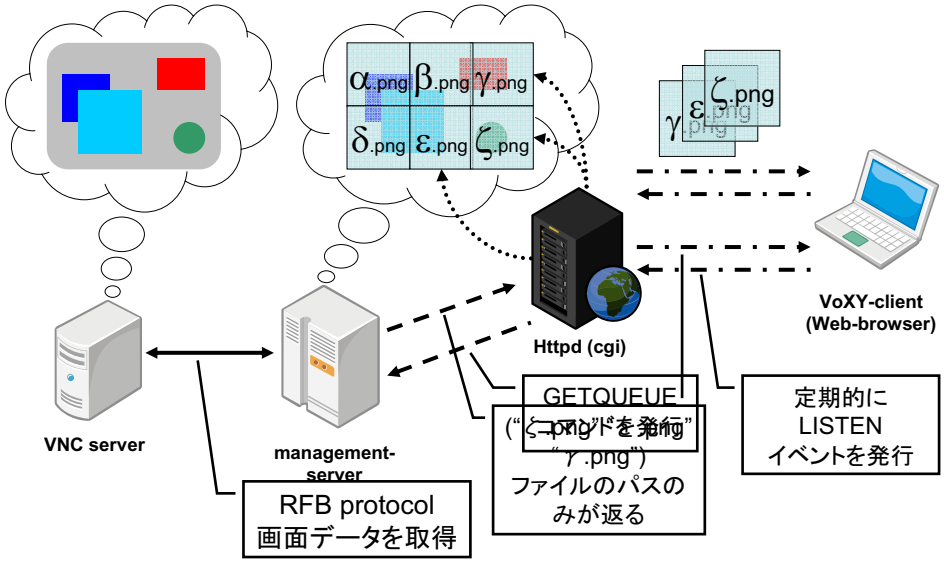


# JavaScriptでの画面描画方法

- VoXY Clientを実現するにあたっての課題
  - いかにしてJavaScriptで画面描画を行うか？
  1. 1x1pixelのDIVタグ方式
  2. Canvas等の拡張タグ方式
  3. DOMによるIMGタグの動的加工




# VoXYプロトコルの流れ



# 残りの発表の流れ

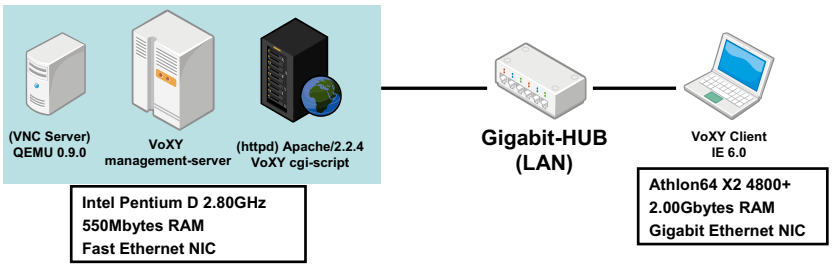
- 研究の背景
- VoXY（Vnc Over proXY）
  - VoXYコンポーネントの紹介
  - Webブラウザにおける画面転送方式
  - VoXYプロトコルの紹介
- 評価
  - 評価手法
  - LISTENあたりの画面更新遅延時間
  - LISTENあたりのデータ転送量平均値
  - 評価のまとめ
- まとめ
  - 既存研究
  - 今後の課題
  - まとめ

# 評価手法

- **定量的評価**
  - **アプリケーションを選定**
  - **ユーザが10分間使用**
    - **解像度は640x480(pixels)**
- **測定項目**
  1. **LISTENあたりの画面更新遅延時間**
  2. **LISTENあたりのデータ転送量平均値**

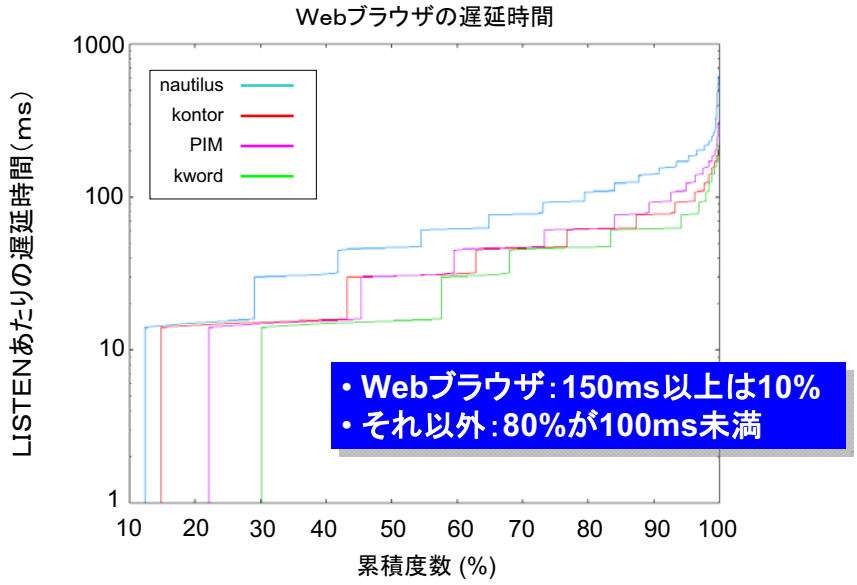
選定したアプリケーション

| 種類 | アプリケーション名 |
|---|---|
| Image Processing | KOffice Kontour |
| Web Browsing | Gnome nautilus 1.0.4 |
| Word Processing | KOffice – KWord 1.1.1 |
| PIM | AKddressBook |



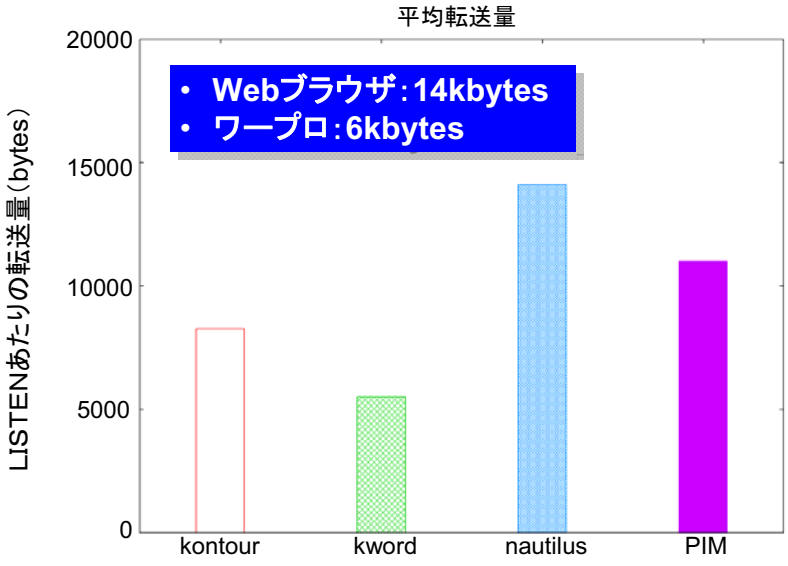

評価環境

# (1) LISTENあたりの画面更新遅延時間




- Webブラウザ:150ms以上は10%
- それ以外:80%が100ms未満

# (2) LISTENあたりの転送量平均




- Webブラウザ:14kbytes
- ワープロ:6kbytes

# 評価のまとめ

2つのデータに関する考察

- LISTENあたりの画面更新遅延時間
  - nautilus：150ms以上は10%
  - それ以外：80%が100ms未満
  - 人が遅延を感じ始める時間
    - Range：50ms – 150ms

B. Shneiderman, Designing the User Interface: Strategies for Effective Human-Computer Interaction

- LISTENあたりのデータ転送量平均値
  - LISTEN間隔は1秒に10回
  - nautilus: 14kbytes
    - (14336 * 8) * 10 = 1.093 (Mbps)
  - kword: 6kbytes
    - (6144 * 8) * 10 = 0.005 (Mbps)

例）NT〇 Bフレッツ実測値

- 19.027Mbps（下り）
- 19.027Mbps > 1.093Mbps > 0.005Mbps

# 残りの発表の流れ

- 研究の背景
- VoXY（Vnc Over proXY）
  - VoXYコンポーネントの紹介
  - Webブラウザにおける画面転送方式
  - VoXYプロトコルの紹介
- 評価
  - 評価手法
  - LISTENあたりの画面更新遅延時間
  - LISTENあたりのデータ転送量平均値
  - 評価のまとめ
- まとめ
  - 既存研究
  - 今後の課題
  - まとめ

# 既存研究(1/2)

## FLOZ:Free Live OS Zoo

- http://www.oszoo.org/wiki/index.php/Free_Live_OS_Zoo
  - Webブラウザ上でJava Applet版VNCを利用
    - HTTPとは別のTCPコネクションが必要
  - OSを選択してWebブラウザ上に画面転送

我々のシステムでは
Webブラウザ/HTTPだけで動作する

# 既存研究(2/2)

## AjaxVNC

- http://sourceforge.net/projects/ajaxvnc/
- VNC server + httpd
- VoXYと同様にHTTPのみを使用
  - Webブラウザ上にVNCの画面を表示できる

- 相違点

➔management-serverとhttpdを分離したアーキテクチャ

- VNC serverに選択の余地がある

例)Virtual Machine Monitorが使用可能

# まとめ

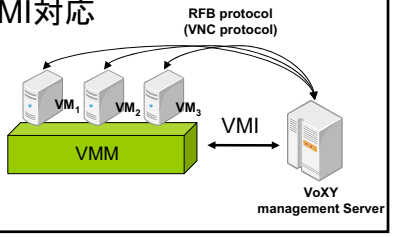
- **VoXY（Vnc Over http proXY）の開発**
  - 導入が容易
    - クライアントはWebブラウザ/HTTPのみで動作
  - VMMのクライアントとして利用可能
  - 既存のソフトウェアの再利用が可能
  - HTTPでThin-Clientを作成できる可能性
- VoXYの定量的評価
  - 十分に現実的な値で画面更新が可能
    - 描画遅延時間の大枠が150ms以内

# ここまでがSWoPP'08での話

～これからの展望～

# 今後の課題

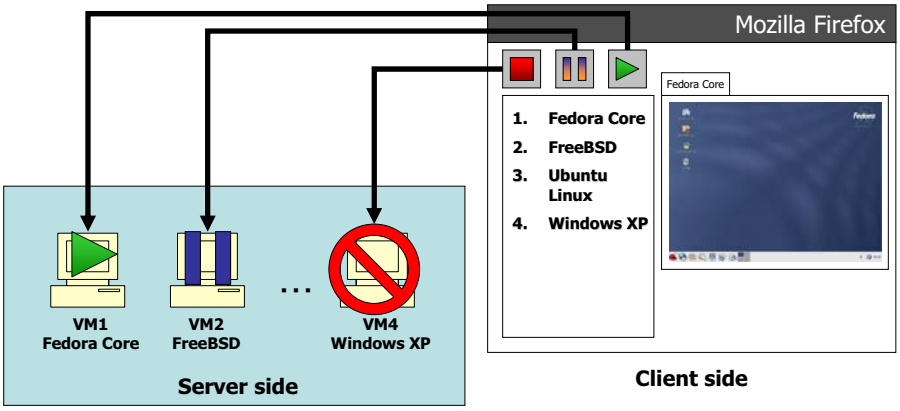
- 実装の洗練
  - LISTENの定期的なコマンド発行の廃止
    - Cometを使用
  - Webブラウザへの描画方法の洗練
    - 拡張タグ方式も試してみる
  - VoXY cgi-scriptの見直し
    - FastCGI, Apache module.. etc
- VMM対応の強化
  - VoXY management-serverのVMI対応
  - VMI：Virtual Machine Interface




# VMM＋VoXY

(イメージ図　本画像ははめ込み画像です)

**Ajaxにより実現されたブラウザ上のUIで**
**仮想マシンの停止，サスペンド，起動がダイナミックに行える**



# 最終目標

VoXYを用いた仮想計算機環境の貸し出しシステム

関連研究:

例1）Amazon EC2

- Webページ上で仮想サーバが購入可能
  - Xen上での仮想計算機の切り売り
  - LAMPがインストールされたサーバを即入手可能
  - 購入した仮想計算機へのアクセス手段
    - SSHでのコンソールアクセス

例2) Amazon S3

- Webページ上でオンラインストレージが購入可能
  - HTTP（REST/SOAP API）, BitTorrentでアクセス可能

# ご清聴ありがとうございました

# GPU と CPU による並列処理のためのスケジューリングの検討

岸本 和哉† 芝 公仁†
†龍谷大学理工学部

## 1 はじめに

近年，GPU の性能が著しく向上し，GPU が CPU と比較してベクトル処理や並列処理に特化したハードウェア構成を持っている．そのため，GPU を画像処理以外の汎用的な数値計算に使用する GPGPU(General Purpose GPU) と呼ばれる動きが活発となっている．また，GPU だけでなく，CPU においてもマルチコア化が急速に進んでいる．現在では，このように複数のプロセッサが一台の計算機上に存在するようになってきている．CPU・GPU のどちらか一方を用いるよりも両方を併用することで高い性能を発揮することができる．

本稿では，CPU と GPU を併用するにあたり，問題の分割や利用するプロセッサなどの分配を考慮し，複数のプロセッサを用いた効率的な問題分割，分配を行うスケジューリングについて述べる．

## 2 関連研究

GPGPU を使用した研究では，GPU を一つ使用したものや一般的な数値計算を GPU で処理させて CPU との比較を行ったりするものが多く見られる [1][2]．また，GPU を計算環境としてグリッドやクラスタに組み込んだヘテロな計算環境に関する研究なども行われている [3]．一部のこのような CPU と GPU を併用した並列処理において，問題をプロセッサの性能に応じて分割する方法を検討している．

## 3 CPU と GPU の並列処理

### 3.1 タスクの分割，分配

CPU と GPU で並列処理を行うにあたり，問題をタスクに分割しなければならない．そして，分割されたタスクを各プロセッサへ分配し，処理を行わせることになる．問題をタスクに分割し，プロセッサへ分配する方法として以下の 2 つが挙げられる．

**手法 1** 問題をプロセッサの数だけタスクに分割し，各々のプロセッサに割り当てる

**手法 2** 問題を多数のタスクに分割し，使用可能なプロセッサから順次タスクを実行し，タスクがなくなるまで処理する

Scheduling GPU and CPU for Parallel Processing
Kazuya Kishimoto† and Masahito Shiba†
†Faculty of Science and Technology, Ryukoku University

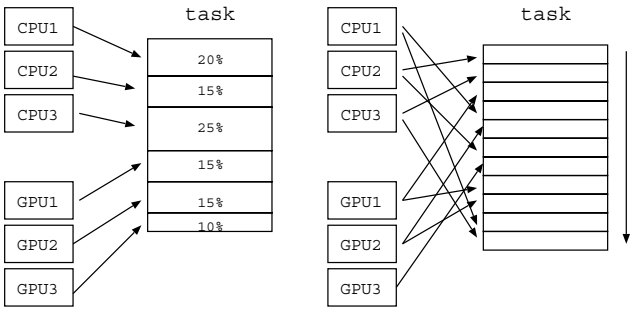



図 1 分配手法 (左図:手法 1 右図:手法 2)

図 1 に二つの手法の概要図を示す．手法 1 はタスクを一度にプロセッサ側へ分配する形となるため，割り当てるプロセッサに応じて，処理させるタスクの大きさをあらかじめ指定しておかなければならない．そのためにはプロセッサの性能からどの程度の大きさのタスクに分割するのかを調べて，求めておく必要がある．

手法 2 では問題をできるだけ多くに分割し，ひとつひとつのタスクを小さくする．処理を開始できるプロセッサから順次タスクを処理していき，タスクがなくなるまで繰り返し実行する．これにより，あらかじめプロセッサの数や性能に応じて分割する必要がなく，問題が分割できるものであるならば効率的に割り当てを行っていくことができる．そして，他プロセスが GPU を使用する場合でも即座に処理を移行することができる．

しかし，この方法ではタスクをプロセッサに割り当てる処理を何度も行うため，処理時間が長くなる可能性がある．

### 3.2 GPU とのインタフェース

グラフィックボードの現在のインターフェースは PCI-Express x16 が使用されていることが多い．PCIExpress 1.1，2.0 における x16 の転送速度はそれぞれ 4GB/s，8GB/s となっているが数年先に登場する PCIExpress 3.0 など，今後の展開を含め，転送速度の異なる規格が複数混在する環境を考慮していかなくてはならない．

表 1　実験環境

| | |
|---|---|
| CPU | Athlon 64 X2 4400+ (2.2GHz) |
| GPU | GeForce 8600 GTS |
| | Quadro FX 570 |
| Memory | 4GB |
| OS | Linux kernel 2.6.22-15 |
| CUDA | Ver 2.0 beta2 |

## 4　予備実験

### 4.1　並列処理実験

#### 4.1.1　実験内容

GPGPU のプログラミング言語として NVIDIA が開発した CUDA を用いた．CUDA は C，C++におけるライブラリとして使用でき，CUDA を用いることで容易にプログラムを作成することが出来る．本実験は CPU と GPU での並列処理の動作を確認したものである．実験内容は N × N 行列の積をもとめるもので，N の値を 100，200，300，400，500 で変化させ，以下の場合で実験を行った．

- CPU のみ
- GPU のみ
- CPU と GPU の並列処理
- CPU と 2 つの GPU の並列処理

#### 4.1.2　実験方法・結果

CPU のみ，GPU のみについてはそれぞれ行列積を行うプログラムを作成し，並列処理を行う場合は CPU で計算するスレッドと GPU で計算するスレッドに分け，データを分割し CPU と GPU に割り当てを行うプログラムを作成した．処理時間の計測は各々の場合で計算にかかった時間を計測し，並列処理の場合は CPU スレッドと GPU スレッドを比較して処理時間の長い方をその場合の処理時間としている．実験環境は表 1 の通りである．以後，実験に用いた GeForce 8600 GTS を GPU1，Quadro FX 570 を GPU2 と呼ぶ．

実験結果を図 2〜図 5 に示す．図 2 は CPU，GPU1，GPU2 をそれぞれ単体で処理させた場合の結果である．縦軸は処理時間，横軸は N × N 行列の N を表す．最も処理時間が短いのが GPU1 であり CPU と比較して約 6 倍の処理速度であった．GPU2 は CPU の約 3 倍の処理速度であり，CPU が最も遅い結果となった．

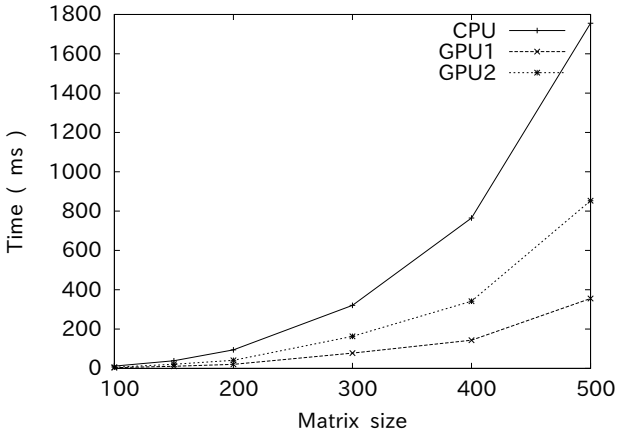



図 2　CPU と GPU1 と GPU2

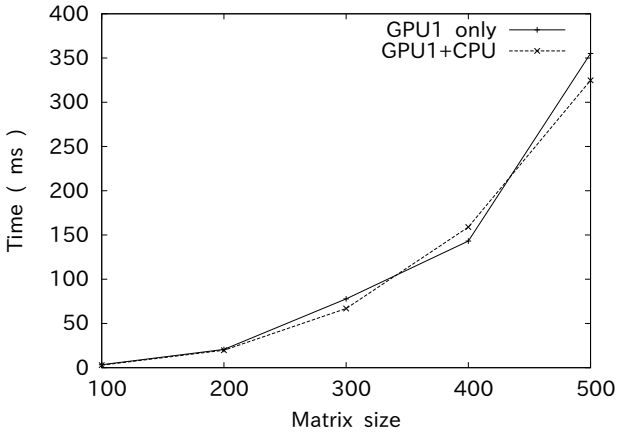



図 3　GPU1 と CPU+GPU1 の比較

図 3，図 4 は GPU 単体と CPU と GPU の並列処理とを比較した結果である．並列処理における処理量の割り当て率は，以前に割り当て率を変化させて行列積を処理させた実験を行った結果から GPU1 との組み合わせでは 8(GPU1):2(CPU) のとき，GPU2 とでは 7(GPU2):3(CPU）のときが最適な割り当て率であることが分かっている．この最適な割り当て率で処理させたときと GPU 単体で処理させたときの比較を行った．一部逆転している箇所があるが，GPU1 と CPU による並列処理では GPU1 のみと比較して約 10%，同様に GPU2 との並列処理では約 30%処理時間が短くなった．行列の N の値が大きいほど確実に処理時間を短縮できているものの，CPU と GPU の性能の差が大きいほどあまり効果は得られないという結果となった．

図 5 では CPU，GPU1，GPU2 の 3 つで処理している．縦軸は処理時間，横軸はデータの割り当て比を表している．割り当て比を変更しながら右に行くほど GPU1 の比率が大きくなり，CPU の割り当て比が小さくなる．

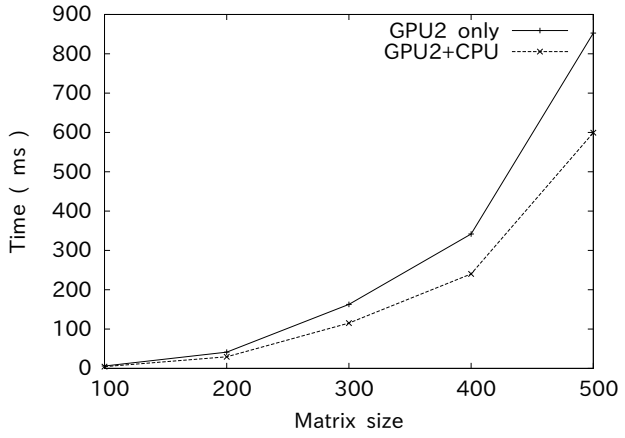



図 4　GPU2 と CPU+GPU2 の比較

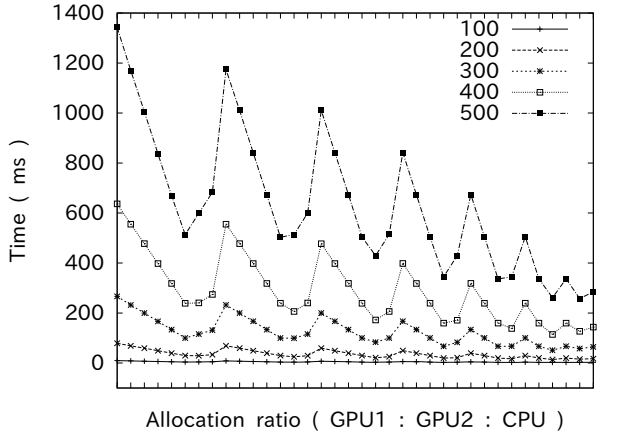



図 5　CPU と GPU1 と GPU2 の並列処理

6(GPU1):3(GPU2):1(CPU1) の割り当て率のときに最も処理時間が短くなる結果となった．CPU と GPU の一つずつの並列処理よりも GPU を複数利用することでより効果が得られることが確認できた．

### 4.2　CPU 資源の活用

#### 4.2.1　CUDA の API

上記の実験はデュアルコア CPU を用い，デュアルコアモードで実験を行った．しかし，同じ実験をシングルコアモードで行った場合，大きく処理能力が低下した．シングルコアモードで行った場合は複数の処理を一つのコアで処理しなければならないため，プロセスやスレッドに割り当てられる CPU 資源は少しでも無駄にできない．

そこで，CUDA において GPU で処理を行う過程で CPU への負荷が増加している処理に注目した．CUDA を使用するにあたり GPU で計算させるためにはデータを GPU のメモリへ転送する，またはデータを受け取る処理が必要である．CUDA にはそのデータ転送のための API が用意されており，その API を使用して CPU と GPU 間のデータ転送を行っている．

データ転送を行うための API は同期をとるものと非同期のものがあるが，必要な場合を除いて通常は同期をとる API を利用する．

同期をとる API は呼び出されたときに，データを送信，または受け取る場合でも GPU の処理が終了するまでビジーウェイトしてしまう．そのため，本来 GPU が処理している間は CPU に余裕ができるはずが，逆に負荷がかかってしまっている．データを送信する場合は GPU で処理が行われていることが少なく，すぐに転送処理が行える．しかし，データを受け取る場合は GPU に計算を指示してからすぐにデータを受け取る API が呼び出されることが多くなるため，GPU の処理が終わるまで CPU には負荷がかかっていることになる．

しかし，GPU の処理が終わっているときに呼び出せばすぐにデータの受け取りを行うため，GPU が処理を行っている間に呼び出さなければ CPU に負荷はかからない．そこで，GPU の処理関数を呼び出した後，GPU が処理している間に以下の関数を使用し，CPU への負荷の影響を確かめた．

**sleep(int sec) / usleep(int usec)**
　指定数秒（マイクロ秒）停止する

**sched_yield()**
　CPU 資源を譲渡する

#### 4.2.2　実験方法・結果

シングルコアモードを使用し，N × N 行列の積を CPU で求めるプログラムと GPU で何らかの処理をさせるプログラムを同時に実行させ，sleep() 及び sched_yield() を用いた場合とそうでない場合の CPU による行列積演算の処理時間への影響を調べた．GPU の方は常に処理している状態にし，データ転送の API を用いて待ち状態にしておくか，API を呼び出さず，sleep() または sched_yield() を呼び出すようにした．

sleep/usleep() は時間の指定が必要となるが，GPU の処理時間に合わせて設定する．常に CPU 処理時間よりも長くなるよう設定している．

sched_yield() はこの関数を呼び出したスレッドより低い優先度のスレッドには有効な手法ではなく，さらに，一度の実行ではすぐに処理が再開されてしまうため，GPU が処理を行っている間，くり返し呼び出しておかなければならない．

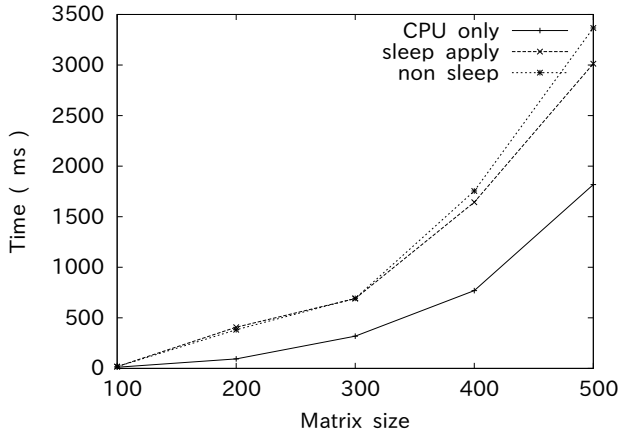



図 6　sleep() 適用

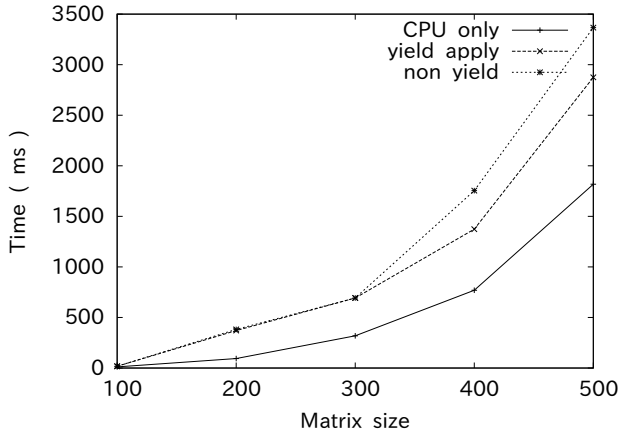



図 7　sched_yield() 適用

上記のことを考慮し実験を行った．sleep() 及び sched_yield() を使用した結果を図 6 と図 7 に示す．縦軸は処理時間，横軸は N × N 行列の N を表す．CPU での処理と GPU での処理の 2 つのプログラムが実行されていることを示すために，CPU で処理したプログラムのみを実行したときの結果も加えている．GPU の処理を並行して実行した場合は，CPU の処理だけ実行した場合の約 2 倍の処理時間であることから，CPU 資源がほとんど均等に CPU での処理と GPU での処理に分けられていることが分かる．

sleep()，sched_yield() ともに N の値が 100 から 300 までは効果はほとんど得られていないが，それ以降から徐々に影響が表われてきた．sleep()，sched_yield() を利用することで約 10〜20 %程度処理時間が短くなっている．このことから sleep()，sched_yield() が確かに行列積の計算に影響を与えていることが分かり，これらの関数を呼び出している間は別のスレッドに CPU 資源を渡すことが可能であることが確認できた．

しかし，これらの手法が有効であると分かったが sleep() を GPU の処理時間に合わせて設定することは困難であり，sched_yield() は CPU の利用を完全に停止するわけではない．さらに，これらの関数自身が CPU を使用するため効率がいいとは言えない．GPU による処理が行われている間，グラフィックカードを監視するなどのシステム側でサポートできる機能が必要である．

## 5　おわりに

本稿では CPU と GPU を併用した並列処理におけるスケジューリングの検討を行った．並列処理の実験においては CPU と GPU との組み合わせにより一応の効果は得られた．そして，CUDA の API による CPU への負荷が確認でき，sleep() と sched_yield() により API による負荷を改善し，CPU 資源を他のスレッドへ渡せることを示した．

今後は，資源の活用のためにシステム側でサポートするための機能を実装，評価し，分配手法 2 によるスケジューリングの実装を行う．

## 参考文献

[1] 大島 聡史，吉瀬 謙二，片桐 孝洋，弓場 敏嗣: ”CPU と GPU を用いた並列 GEMM 演算の提案と実装 (数値計算)”，情報処理学会論文誌．コンピューティングシステム Vol.47，SIG_12(ACS_15)，pp.317-328，(2006.9)

[2] 柏木 啓一郎，宮島 信也，柏木 雅英，内村 創: ”CPU および GPU を併用する FFT ライブラリの提案と評価”，情報処理学会研究報告．ハイパフォーマンスコンピューティング Vol.2007，No.80，pp.13-18，(2007.8)

[3] 小谷 祐基，伊野 文彦，萩原 兼一: ”グリッド環境において遊休 GPU を活用するための資源選択手法”，情報処理学会研究報告．ハイパフォーマンスコンピューティング Vol.2006 No.106，pp.37-42，(2006.10)

# GPUとCPUによる並列処理の　めの スケジューリングの検討

龍谷大学
理工学部

岸本　和哉
芝　公仁

1

## 目次

- 研究背景
- GPU資源の活用
- GPGPUについて
- 関連研究
- 実験
- スケジューリングの考察
- まとめ

2

# 研究背景

近年、GPGPUが注目され、様々な研究分野で取り入れられている

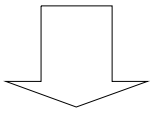
◇ GPGPU(General Purpose GPU)
・・・GPUをグラフィックのみならず汎用計算に用いる技術

現在のコンピュータ環境において、複数のCPU、GPUを搭載することが可能

3

# 目的

問題を分割しGPU、CPUで並列に処理することで、計算処理能力を向上させることができる(関連研究）



CPUとGPUにどのように問題を割り当てるかが重要

CPUおよびGPUに最適な割り当てを行う
スケジューリングの提案、及び実装

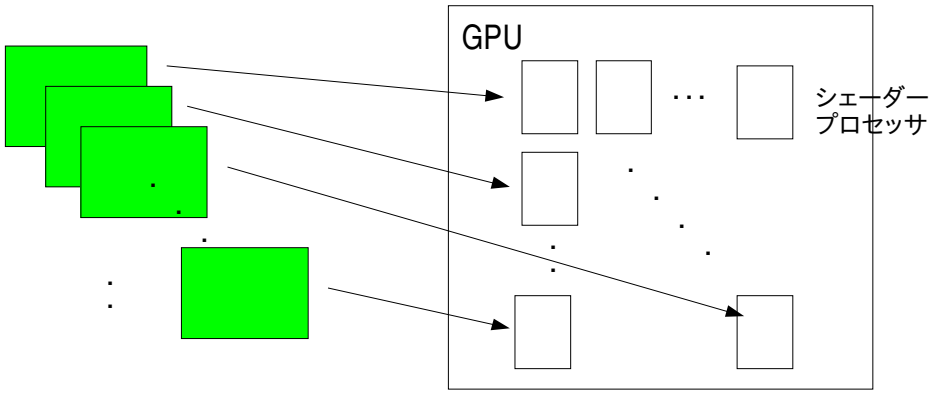
4

# GPGPU(1)

◇GPUを汎用計算機として利用

- 複数のシェーダープロセッサによる並列処理が得意
- 積和演算、シミュレーション、FFT

◇複数のシェーダープロセッサに仕事を割り当て、並列に実行させる




5

# GPGPU(2)

◇シェーダー言語によるプログラミング

シェーダー言語によるGPGPUプログラミングは比較的困難
（シェーダー言語の知識、関数の問題）

◇GPGPU用プログラミング言語

シェーダー言語を知らなくても、一般的なプログラミング言語による
GPGPUの　めのプログラミング環境が用意されている

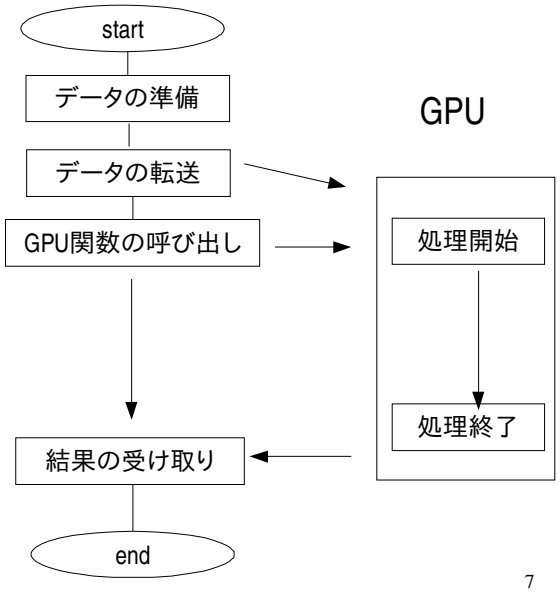
GPGPU環境

- •NVIDIA・・　Cg、CUDA
- •AMD（ATI）・・・　CAL
- •ウォータールー大学・・・Sh
- •スタンフォード大学・・・Brook

6

# CUDAの処理の流れ

- GPUへの「データの転送」
- GPU関数の呼び出し
- GPUからの「データの受け取り」



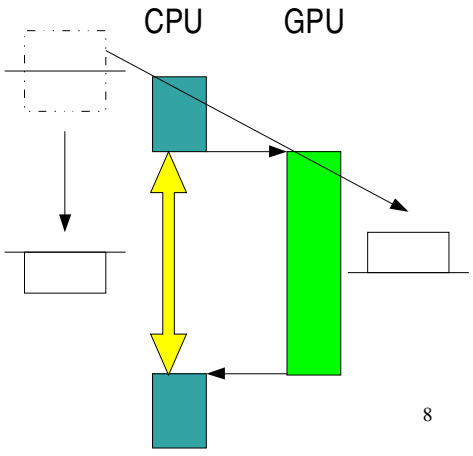

7

# CPUとGPUの並列処理

◇CPUとGPUの並列処理

プロセッサ単体で処理するより効率が上がる

問題を分割し、GPUとCPUに
それぞれ割り当てる




8

# 関連研究

- CPUとGPUを用い　並列GEMM演算の提案と実装（数値計算）

  （2006,大島ら)

  - GEMM演算をCPUとGPUで並列処理
  - HPLベンチマークへの適応
  - CPUとGPUの問題割り当てのチューニングの検討
- CPUおよびGPUを併用するFFTライブラリの提案と評価

  (2007,尾形ら)

  - CPUとGPUにおける性能モデルの構築
  - CPUとGPUへの最適な割り振りを推測
  - 実測値と推測との比較



# 演算資源

◇複数のGPU、CPUを利用し、並列処理を行う

＊CPU x n（コア数）

＊GPU x m

◇計算量に対し、プロセッサの数が多いと処理性能が低下する可能性がある　め、使用するCPU、GPUの選択なども必要となる

- 計算量の少ない場合は処理能力の高いGPU、CPUを使用する
- 計算量の多い場合はGPU、CPUに手分けして処理をさせる

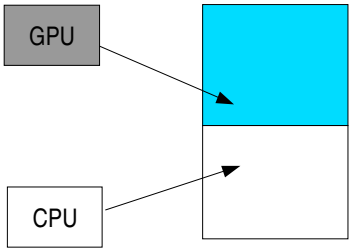
# 問題の分割

◇問題を2つ以上のタスクとして分割する

- プロセッサの数、能力
- タスクの大きさ

◇分割し　タスクのプロセッサへの配分

- プロセッサ数で分割し　タスクをそれぞれ配分する
- 複数に分割し　タスクを動的に各プロセッサへ配分する




11

# 問題の分割・配分方法（1）

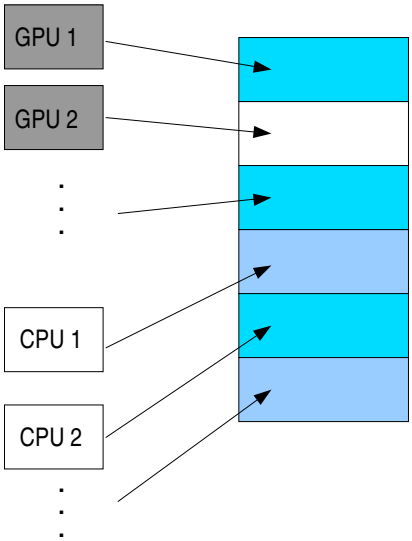
1、プロセッサの数だけ問題を分割

2、プロセッサの能力に応じて
タスクの割り当てサイズを決定

あらかじめ

- 性能比
- 割り当て率

を求めておくことが必要

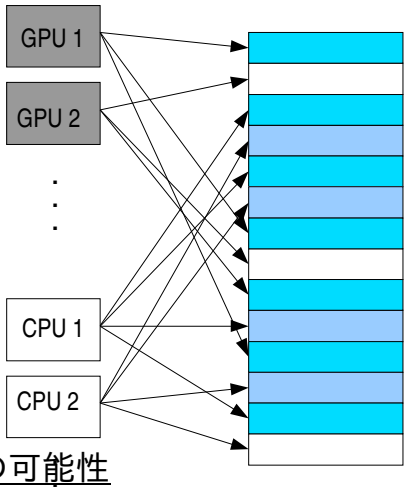



＜＜関連研究　1、2　はこのタイプ＞＞

12

# 問題の分割･配分方法（2）

- 複数のタスクに分割
- 使えるプロセッサは順次タスクを処理していき、タスクがなくなるまで繰り返す




- プロセッサ数
- 割り当て率

に関係なく全てのタスクを処理可能

頻繁なデータの移動による処理時間増加の可能性

⇨ 最適な割り当てを行うスケジューリングが必要

13

# 実験

◇目的

CPUとGPUによる並列処理の試験

◇実験内容

NxN行列の積を以下の場合で行う

N = 100 , 200 , 300 ,400 ,500

1.CPUのみの処理

2.GPUのみの処理

3.CPUとGPUでの並列処理（方法1）

→ CPU x 1 + GPU x 1

→ CPU x 1 + GPU x 2

14

# 実験環境

◇使用し　プログラミング言語

C言語 + CUDA

| | |
|---|---|
| CPU | Athlon 64 X2 4400+ (2.2GHz) |
| GPU | GeForce 8600GTS |
| | Quadro FX 570 |
| メモリ | 4GB |
| OS | Linux kernel 2.6.22-15 |
| ビデオドライバ | ver 177.13 |
| CUDA | ver 2.0 beta2 |

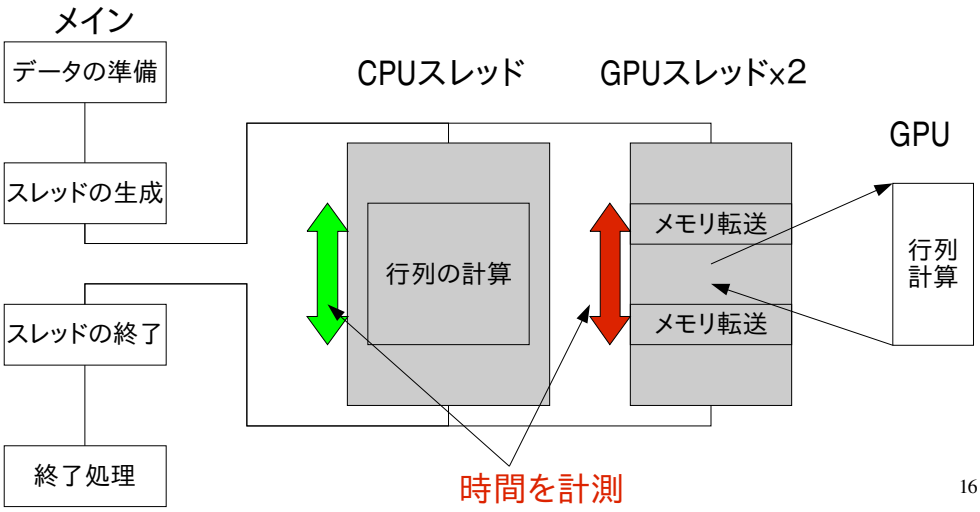
⇨ GPU1

⇨ GPU2

15

# 実験方法

◇ CPUでの処理　、　GPUでの処理　のスレッドを生成
◇ 矢印の部分の時間を計測




16

# 実験方法

◇処理時間

- 計算にかかる時間を計測
- CPUスレッド、GPUスレッドでの処理時間の長い方を使用する

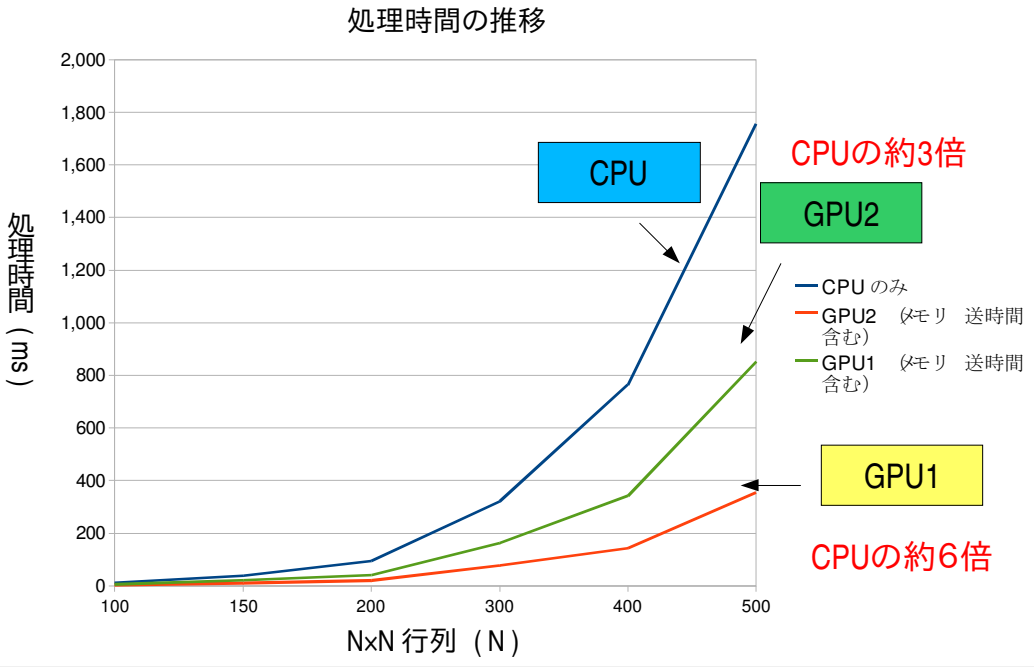
MAX ( CPU , GPU )

◇問題の割り当て率は　10%　ずつで計測

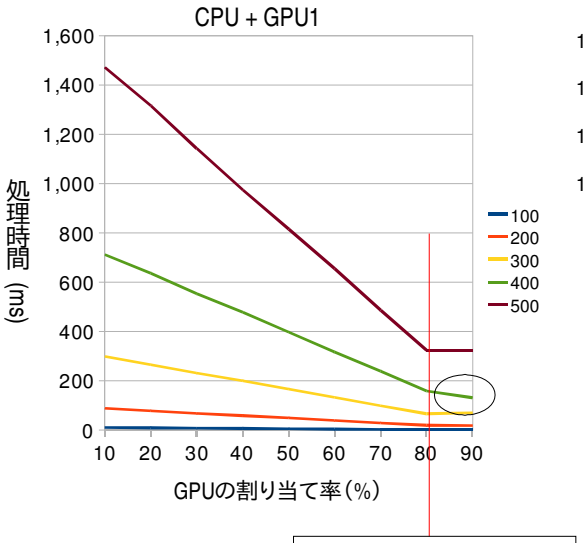
CPU: GPU　＝＞　9:1 8:2 7:3 6:4 5:5 4:6 3:7 2:8 1:9

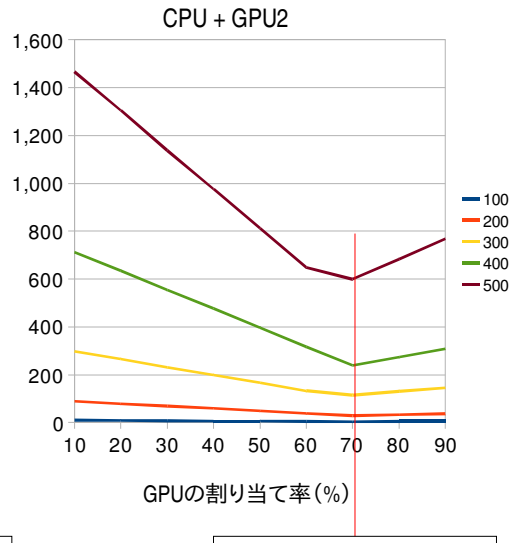
17

# 実験結果(CPUのみ、GPUのみ)



# 実験結果（CPU+GPU）



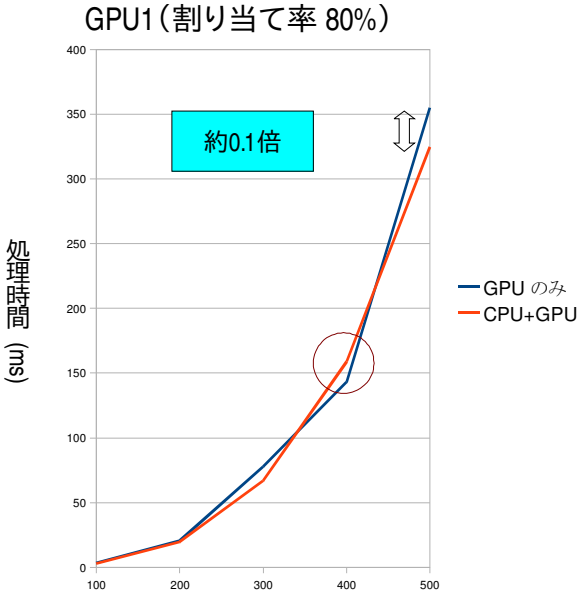



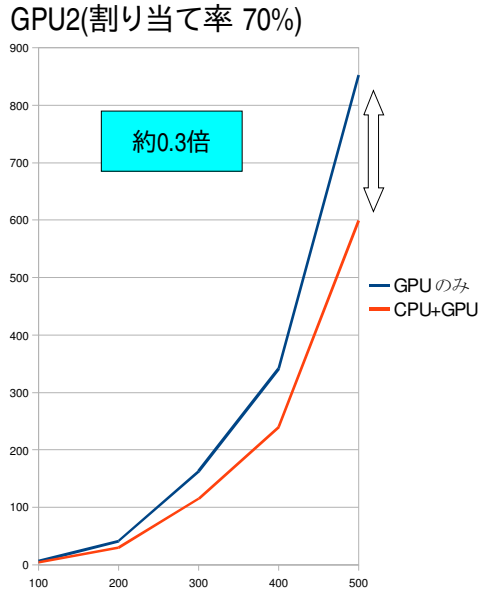

割り当て率：80 %

割り当て率：70 %

19

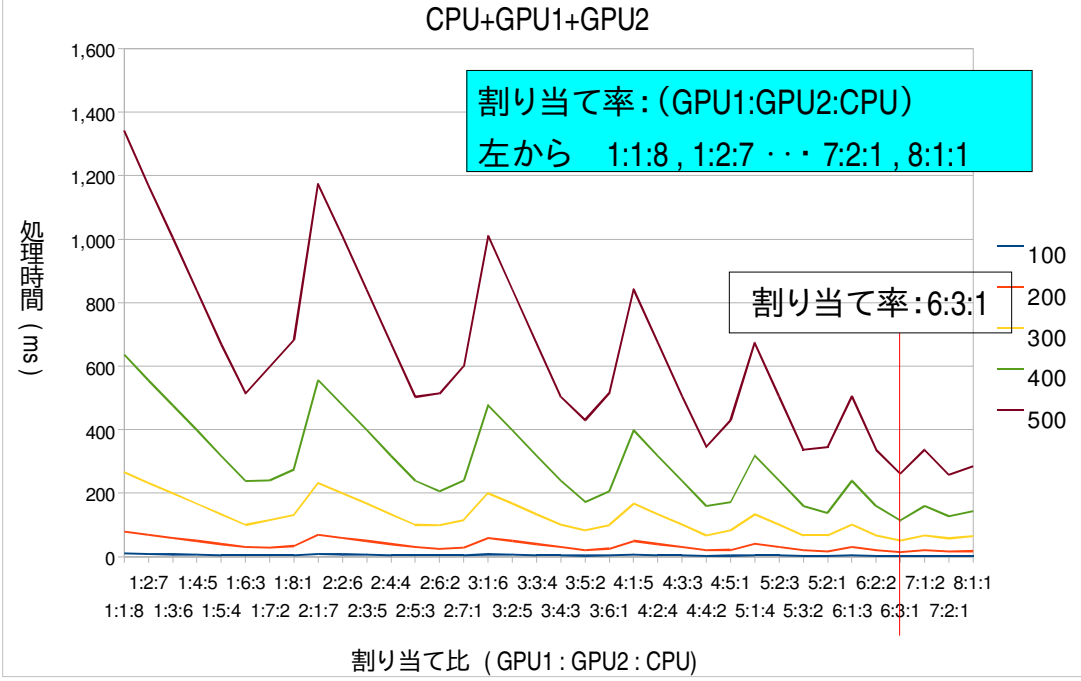
# 実験結果（GPUのみと並列との比較）




20

# 実験結果(GPU1,GPU2,CPU)




# 実験結果

◇CPU+GPU1

GPU1のみと比較して　約10%

◇CPU+GPU2

GPU2のみと比較して　約30%

- Nによって最適な割り当てが異なる(一部)
- グラフより、一部GPUのみの方が早いときがある (CPU +GPU1)
- Nの値が大きいときほど効果が得られる

22

# 実験結果

◇グラフ:処理時間の推移　より

GPU1　:　GPU2　:　CPU　の処理時間の比
6　　3　　1

実験結果(GPU1,GPU2,CPU)　の　割り当て率の比

GPU1:GPU2:CPU ＝ 6 ： 3 ： 1

プロセッサの速度比と同じ比率となっ

23

# コア数の違い

◇これまでの実験は デュアルコアCPU による処理

ー 各スレッドの処理をコアで分担

シングルコアで行っ　場合に処理性能が大きく低下する

24

# CUDAの特徴

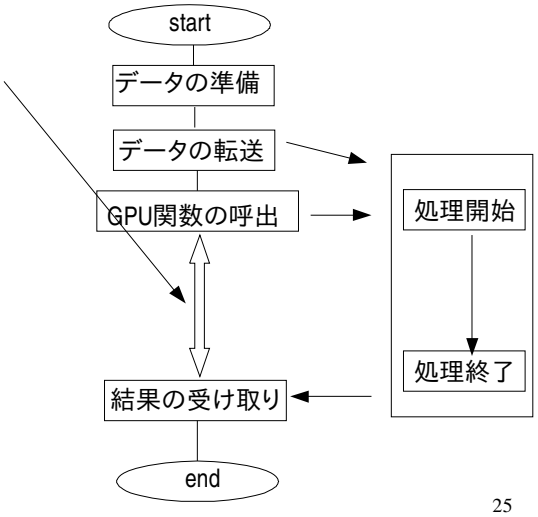
- GPUの処理は非同期で実行できる

「GPU関数呼出」から
「結果の受け取り」までの区間

- 何もしない
- データを受け取るまでにしておく処理

結果の受け取りを行う関数を使うと
GPUの処理が終わるまでビジー状態

CPU資源の無駄




25

# 資源の開放

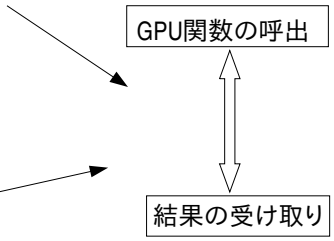
◇ sleep( ) / usleep( ) <unistd.h>

・指定秒数（マイクロ秒数）間休止する
・GPUの処理時間にあわせて調整が必要

◇sched_yield ( ) <sched.h>

・ CPUを明け渡す

・ 一度の実行ではすぐに処理が戻るので
ループさせる



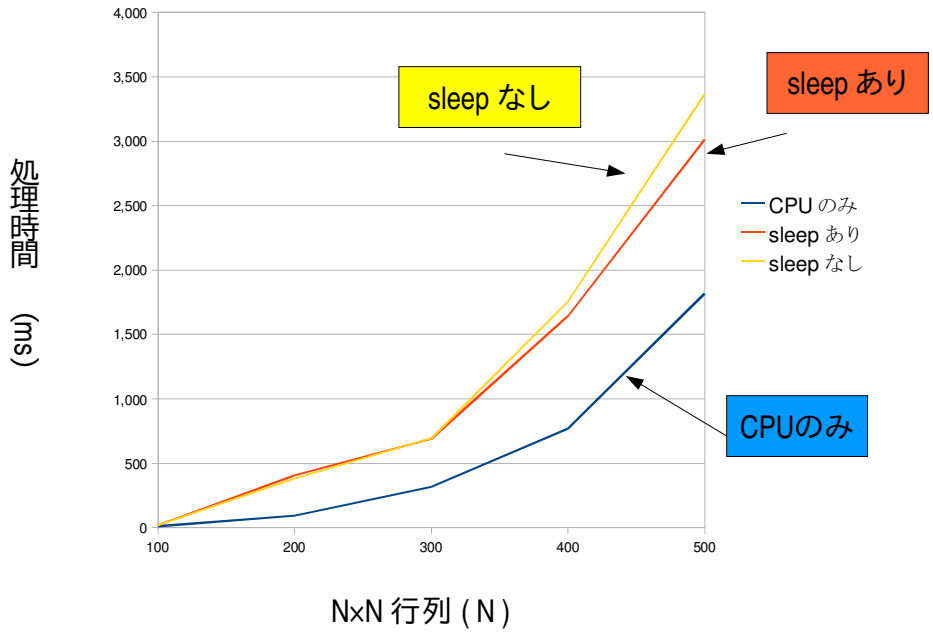

26

# 実験

- sleep , yield の効果を確認
  - CPUスレッド、GPUスレッドが動作
  - GPU処理開始から結果の受け取りの間に
    sleep , yield　を挿入
- ビジー状態によるGPU処理への影響
  - GPUのみによる計測
  - ビジー状態の間にGPUの処理に関係することをしているかどうかの確認

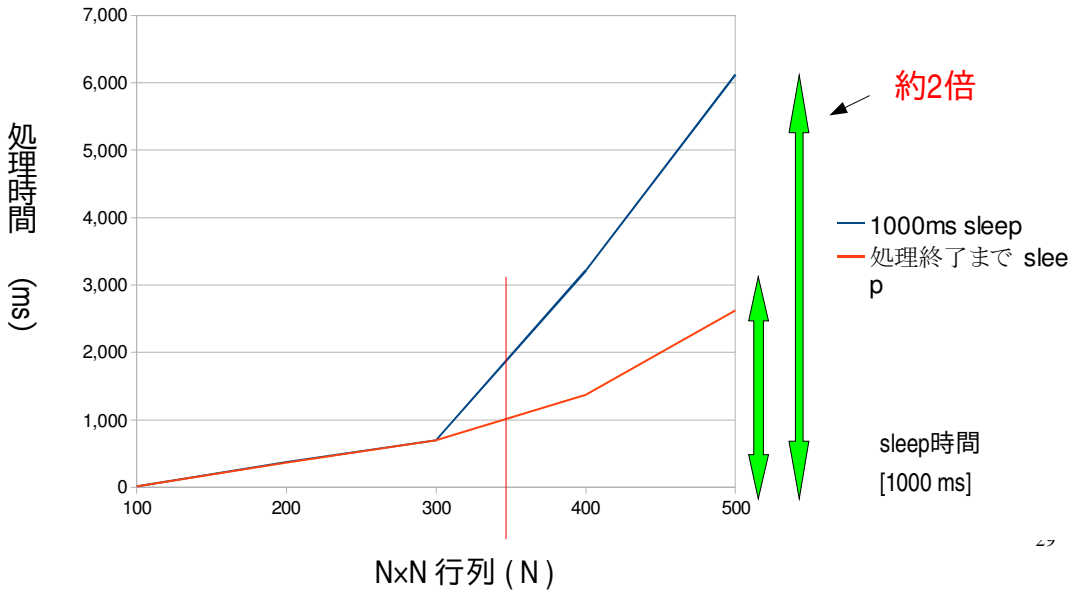
27

# CPU計算結果（sleep）

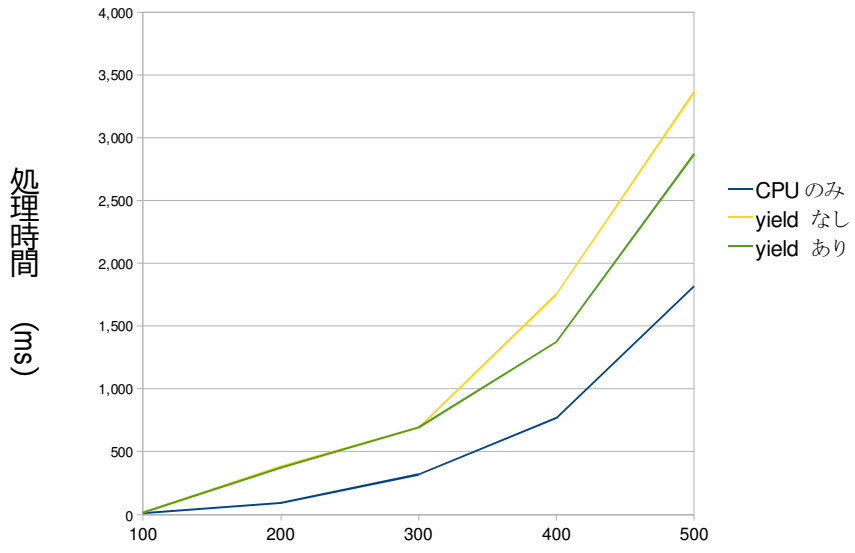



28

# sleepによる影響

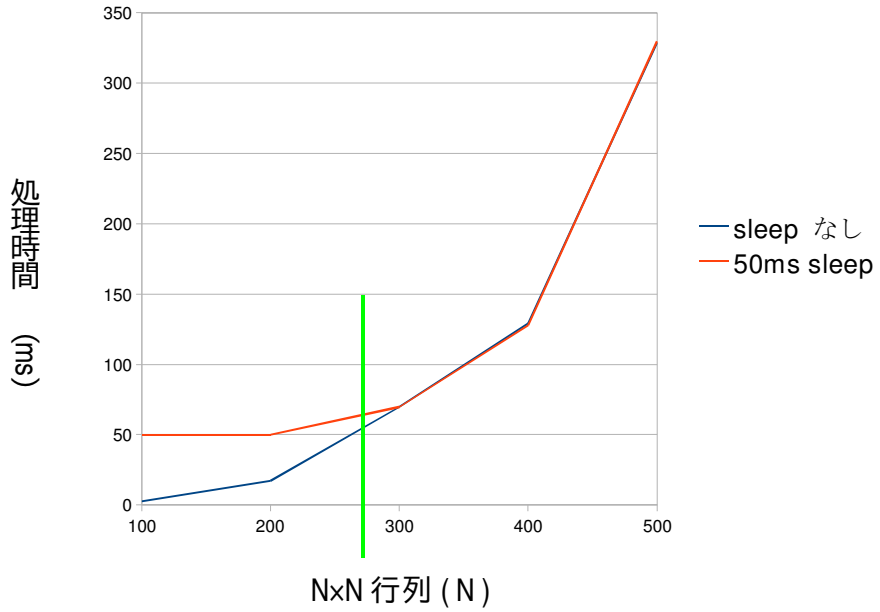



# CPU計算結果(yield)




30

# sleep挿入時のGPUの処理時間




31

# 実験結果

◇ sleep , sched_yield による効果を確認

― CPU資源の無駄使いが起きていることを表している

◇スリープの終了後にGPUの処理が復帰すると処理時間が2倍になる

◇ビジー状態時にGPUに関わる処理等を行っていない

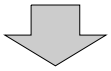
― GPUの処理とビジーに関係無い

32

# 実験結果

以上のことから

- CPU資源の削減が可能であり、必要であることが確認でき
- sleep や sched_yield を使用する場合は注意が必要
  - sleep : GPU処理の終了のあと、結果の受け取りに遅延が発生する場合がある
  - sched_yield : 完全にCPUの使用を止めることはできない



システム側、カーネルでサポートできる仕組みが必要

33

# まとめ

- CPUとGPUによる並列処理の有用性
- GPGPUにおけるCPU資源の問題性
- 今後の予定
  - CPU:GPU資源を考慮し　スケジューリングの実装
  - 2つ目の分配手法による並列処理の実験
  - 様々な問題における並列処理

34

# センサ・アクチュエータを用いた仮想ネットワークにおける複数プロセスを実行するノード上の資源管理機構のための考察

金丸 達雄 † 横田 裕介 †† 大久保 英嗣 ††
† 立命館大学大学院理工学研究科 †† 立命館大学情報理工学部

## 1 はじめに

スマートタウンやビルなどの大規模施設を運用する場合，施設運用サービスと業務サービスを実施する必要がある．施設運用サービスとは，異常検知 (火災検知，建築物のモニタ) や環境管理 (空調管理，消灯点灯管理，扉の開閉) といった，施設全体にわたって行う必要があるサービスである．業務サービスとは，業務用品管理 (商品温度の管理，商品数量の管理) と業務用品の情報提供 (商品の取り扱い履歴の提供，コマーシャル) といった，特定の業務を行うエリアに対して行う必要があるサービスである．

大規模施設において，このようなサービスを低コストで実現するためには，無線センサ・アクチュエータネットワーク上でサービス毎の仮想ネットワークを構成することが有効な方法である．

無線センサ・アクチュエータネットワークは，センサノードからの情報を提供するサービスや，センサノードから取得したデータに基づいたアクチュエータの制御サービスを提供する．これにより，遠隔からの機器の保守と環境の観測が容易に実現でき，管理コストの削減につながる．また，無線による通信を行うため，配線コストが削減できる．

サービス毎の仮想ネットワークとは，複数のサービスでノードや通信経路を共有して構成されるネットワークである．これにより，サービス別に独立した構成をする場合と比較して，センサ・アクチュエータの量を削減できる．また，サービス毎に独立している場合と比較して，広域から多様な情報を取得可能である．このため，効率的なセンサ・アクチュエータの制御が可能になる．

無線センサ・アクチュエータネットワーク上でサービス毎の仮想ネットワークを実現する場合，ノード上の資源管理 (センサやアクチュエータ, 通信タイミング, 周期毎の処理時間等の管理) が重要となる．これは，サービス間で利用される資源の競合を回避し，ノードの高信頼化を実現するためである．

本稿では，ノード上の資源管理機構を設計するため，センサ・アクチュエータを用いた仮想ネットワークについての考察を行う．以下，仮想ネットワークの概要，仮想ネットワークの例，仮想ネットワークにおける検討事項と対策について述べ，おわりに本稿をまとめる．

## 2 仮想ネットワークの概要

仮想ネットワークは，global network と local network に分類される．図 1 に概要を示す．global network は，ネットワーク全体に共通するサービス (施設運用など) を管理する．このネットワークは，仮想ネットワーク中に唯一存在し，すべてのノードが所属する．また，所属するプロセス (サービスを実現する処理) は，local network より優先される．これは，global network で扱うサービスは事故や故障の防止など緊急性の高いものであることが多いため，一般に優先度を高くすることが望ましいことが多いためである．local network は，特定の場所 (店舗，オフィス) に限定したサービスを実施する．施設内に複数存在し，業務サービスを実施するノードのみが所属する．なお，複数の local network 間では，ノードを共有しないものとする．

また，各仮想ネットワークは，1 つの基地局，複数のセンサ・アクチュエータノードを持つ．次にそれぞれについて説明する．

**基地局**　仮想ネットワークを管理する，計算能力の高いノード (PC など) である．仮想ネットワークに所属するセンサノードからセンシングデータを収集する．また，仮想ネットワークに所属するノードに対するプログラムの配布やアクチュエータの制御を行う．

**センサノード**　指定されたサンプリング周期毎にセンサからデータを取得し，隣接したノードもしくは基地局にセンシングデータを送信する．また，隣接ノードとの協調動作を行う．例として，基地局までの通信の負荷分散，アクチュエータの動作結果の観測とそれに基づいた制御が挙げられる．

**アクチュエータノード**　実世界に作用する機器であるアクチュエータを搭載したノード．図 2 に動作例を示す．アクチュエータは，基地局による制御に従って動作する場合と，周囲のノードから情報を取得し自律的に動作する場合がある．前者は，アクチュエータの動作を決定するための計算処理プロセスの負荷が高い場合や広域のセンサ情報が必要な場合に使用される．後者は，高速な応答が必要な場面に用いられる．

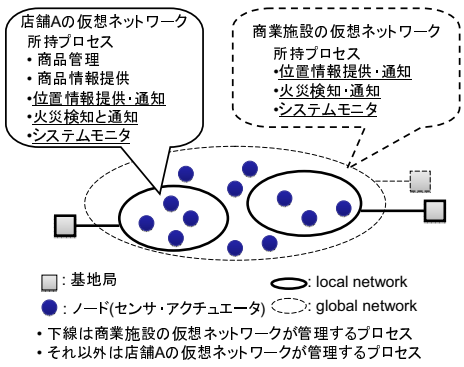



図 1　仮想ネットワークの構成例

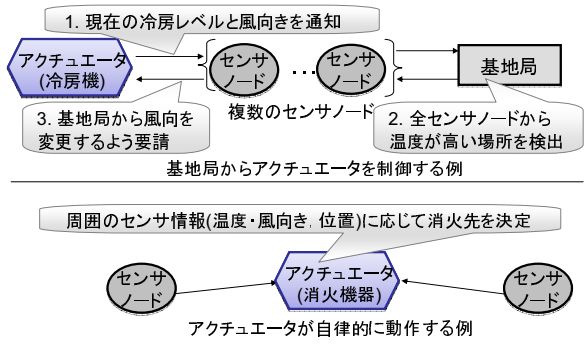



図 2　アクチュエータの動作例

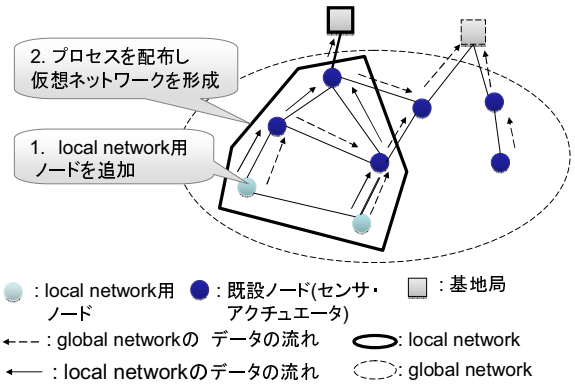



図 3　商業施設においてを新規に業務を実施する例

## 3　仮想ネットワークの例

本章では，商業施設において新規に業務を実施する例と仮想ネットワークを利用したアクチュエータの制御例について示す．それぞれの概要を図 3, 4 に示す．

商業施設において，新規に業務を開始する場合，global network に local network 用のサービスを実現するノードを追加する．次に，local network の基地局から当該サービスを実行するためのプロセスを配布し，global network のプロセスと同時に動作させる．このように複数のサービスでノードを共有し，ノード上でそれぞれのサービスに対応するプロセスを並列に実行することで，センサ・アクチュエータノードの量を削減できる．

次に仮想ネットワークを利用したアクチュエータの制御例について述べる．この例では，施設内に客足感知サービス (global network) と空調管理サービス (local network) が設置されている．客足感知サービスは，人感センサノードから構成されており，客足の情報を取得することができる．また，空調管理サービスは，温度センサノードから情報を収集し，基地局から空調機に指示を出すことで空調を行う．このとき，空調管理サービスに所属する基地局は，人感サービスに所属する基地局へ人の流れについての問い合せができる．これにより，自身の保持する温度情報とともに人の流れについて評価することが可能となり，効率的な空調が可能となる．このように，センサ・アクチュエータノード上に仮想ネットワークを実現することで，広域からの多様な情報の取得

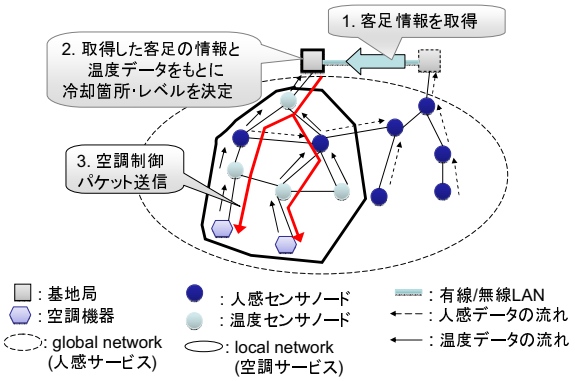



図 4　仮想ネットワークを利用したアクチュエータの制御例

が可能になる．結果，効率的な制御が可能となる．

## 4　仮想ネットワークにおける検討事項と対策

仮想ネットワークにおける検討事項として，プロセス・資源の管理，ノードの移動に起因する問題，事象検出のための問い合せ手法，基地局間での協調方法，消費電力の削減，セキュリティが挙げられる．これら 6 つの検討事項と対策について次に述べる．

### プロセス・資源の管理

プロセス・資源の管理における検討内容として，プロセスの優先度管理やプロセスへの資源割付けが考えられる．これは，local network, global network のプロセス間における優先度管理 (global の優先度が local よりも高い)，global network のプロセスが local network 用ノードを占有しサービスの実施を妨げるような場合への対処，プロセスが利用する資源の競合・不足への対処である．

これらの問題に対応するため，ノード上でプロセスと資源を検査する方法が考えられる．この検査は，資源の状態が変化する際に行う必要がある．これは，プロセスの実行前後，デバイスからの情報通知時，プロセスが利用する資源が変化した場合である．また，資源の検査を実現するため，プロセスが利用する資源を管理する必要がある．管理する資源として，サービス間で競合する可能性がある，アクチュエータ，センサ，処理時間，記憶領域，ノードの属性が考えられる．さらに，資源の特性がそれぞれ異なるため，資源の種類毎に管理方針を提供する必要がある．例として，アクチュエータには単一プロセスの割当てを行うことが考えられる．また，センサにはキャッシュを用い複数プロセスから読出しを行うことが考えられる．

### ノードの移動に起因する問題

移動体ノードを経由する通信，移動体ノードによる資源アクセス方法を考慮する必要がある．これは，固定ノード間のように常に経路が存在せず，ノードの移動時に通信路が切断されるためである．

この問題を解決するため，通信の性質に合わせた対処が必要となる．対処が必要な場面として，移動体ノードから情報を収集する場合と移動体ノードが情報を収集する場合がある．

移動体ノードから情報を収集する場合，基地局へセンシングデータを送る必要がある．データを送信するノードを A，ノード A が通信を行う際の 1hop 先のノードをノード B とする．このとき，ノード B が固定ノードの場合と移動体ノードの場合の 2 つの場合が考えられるが，前者の場合は常に安定した通信が可能なため特に考慮する必要はない．後者の場合，状況によっては安定した通信が困難な場合があるため，そのような場合の対処について以下で考察を行う．このような場合，通信元のノードから基地局までの経路が成立する確率を短時間で判断する必要がある．これは，ノードの移動でトポロジが変更される前にデータを送信するためである．そのため，図 5 のように，ノード B の先に固定ノードが存在する場合のみ，ノード B を経由した通信を許可する．これは，ノード B が，ノード A と隣接する固定ノードの両方と接続し続ける確率を短時間で判断できるためである．また，両方のノードと接続する確率を判定する手法として，現在の RSSI(Received Signal Strength Indication) と単位時間当たりの RSSI の減衰値を比較する方法が考えられる．

移動体ノードが情報を収集する場合として，移動するアクチュエータを考える．このとき，アクチュエータの作用できる範囲は，ノードの一般的な通信距離 (30〜70m) より小さいものとする．アクチュエータが広域の情報を必要とする場合，基地局から送られてくる定期的な制御パケットを利用する．一方，周囲の事象に対し即座に対処する必要がある場合 (消火など) は，1hop 以内のノードからデータを収集する．これは，アクチュエータが作用する範囲の環境情報を収集するためには，最初の仮定より 1hop 以内のノードからデータを取得すれば十分であり，また短時間での情報収集が可能となるためである．

### 事象検出のための問い合せ手法

従来のセンサネットワークシステムにおいて用いられてきた，具体的なセンサの種類を指定する問い合せ方法では，アクチュエータを稼動させるための条件となる事象を検出する問い合せの記述が困難な場合が考えられる．アクチュエータノード自体の移動および周囲のセンサノードの移動により，アクチュエータノード近辺のセンサノードの配置が動的に変わる場合がある．このような場面では，周囲のセンサノードが持つセンサの種類が一定しないことになる．

例として，火災検知が挙げられる．この例では，温度センサの値を基に火災を認識し，消火を行うアクチュエータを想定する．従来の問い合せ手法では，煙センサを搭載したノードが火災を検知した場合でも，温度センサを持つノードが周囲になければ，アクチュエータは火災を認識できず消火ができない．

このような問題は，問い合せシステムを拡張し，事象を検出するための抽象度の高い問い合せ記述を可能にすることで解決できると考えられる．火災検知の例では，アクチュエータが「火災検知に利用可能な情報」の問い合せを隣接ノードに送信することで煙センサから情報が取得できる．これは，煙センサを持つノードが火災検知に利用可能なセンサとして，自身の煙センサを管理しているためである．

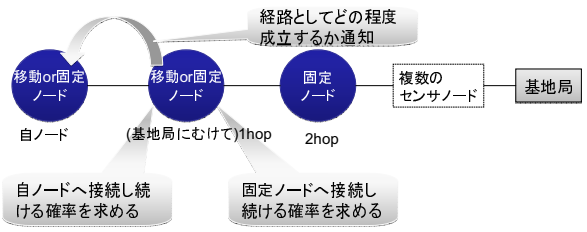



図 5　移動体を経由した基地局への通信

このような問い合せを実現するためには，抽象度の高い問い合せ記述を実行時に解釈し，その時点におけるセンサノードの配置状況に応じた具体的なセンサノードを指定した問い合わせに書き換える機構が必要になる．火災検知の例では，火災検知という事象をアクチュエータ周辺の温度センサや煙センサからのデータ取得とその値の変化に基づく状況判断という処理に解釈することになる．ノードが持つセンサの種類とノードの位置情報，および火災などの事象の記述とセンサデータの処理方法の対応付け規則 (温度や煙センサからのデータの閾値など) をノード上で管理することで，このような問い合せを実行する機構を構築することができると考えられる．

### 基地局間での協調方法

基地局間で協調動作を行う必要がある場合，各基地局が持つ機能を発見し，その利用方法を取得する必要がある．これは，Jini や UPnP などのサービス発見機構を用いることで解決できる．

### 消費電力の削減

複数の仮想ネットワークに所属するノードは，単一のサービスだけを実行する場合と比較して消費電力が増加する．これは，ノードの処理時間や通信量が増加するためである．この問題に対処するため，仮想ネットワーク間でルーティングや通信周期を最適化することが考えられる．

### セキュリティ

センサデータやアクチュエータの不正な利用を防ぐ必要がある．この問題を解決するために，楕円曲線暗号系などの計算量が少ない認証機構やハードウェアのサポートを利用する方法が考えられる．また，認証結果を反映するため，ノードの資源へのアクセス管理と仮想ネットワークへの参入を管理する機構が必要である．

## 5　おわりに

本稿では，センサ・アクチュエータを用いた仮想ネットワークにおける複数プロセスを実行するノード上の資源管理機構のための考察について述べた．今後，資源管理機構について既存研究の調査と設計を行う．また，想定する仮想ネットワークが妥当か検討する．これは，仮想ネットワークの階層についての検討であり，サービスによる階層以外に物理的な通信を管理する階層が必要であると考えている．

# センサ・アクチュエータを用いた仮想ネットワークにおける複数プロセスを実行するノード上の資源管理機構 のための考察

立命館大学 理工学研究科
基本ソフトウェア研究室
M1 金丸 達雄

## 発表内容

- はじめに
- 想定する大規模施設
- 大規模施設のコスト削減方法
- 仮想ネットワークの概要
- 仮想ネットワークにおけるプロセス
- 仮想ネットワークの構成
- センサ・アクチュエータノード上の仮想ネットワークにおける検討事項
- 検討事項への対処方法
- 今後の予定
- おわりに

2

## はじめに

- 大規模施設ではコストを抑えることが重要
  →無線センサ・アクチュエータネットワークを用いて
  コスト削減(後述)
- センサ・アクチュエータネットワークを利用する場合
  ノード上の資源管理が重要
  - ノードの信頼性を向上させ業務の停止を防ぐため
  - 管理対象としてアクチュエータやセンサ，通信タイミング，
    周期毎の処理時間を想定
- 資源管理機構を検討するため大規模施設における
  センサ・アクチュエータネットワークについて考察

3

## 想定する大規模施設

- 大規模施設で必要なサービス
  - 施設運用のためのサービス
    - 異常検知：火災検知，ネットワークや建築物のモニタ
    - 環境管理：空調管理，消灯点灯管理，扉の開閉
  - 施設の業務サービス
    - 業務用品管理：商品温度の管理，商品数量の管理
    - 業務用品の情報提供：商品取扱い履歴の提供，
      コマーシャル
- これら複数のサービスを低コストで
  提供したい

4

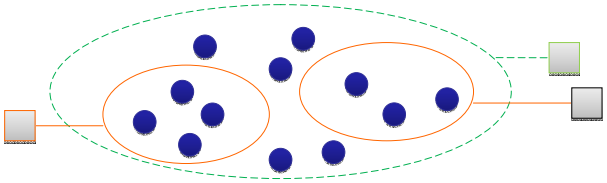
## 大規模施設のコスト削減方法

- 無線センサ・アクチュエータネットワーク上で
  サービス毎の仮想ネットワークを利用
- 無線センサ・アクチュエータネットワーク
  - センサノードから情報を収集し，アクチュエータノードを
    制御するサービスを提供
  - 無線による配線コストの削減
  - 無線ネットワークとセンサで機器・環境管理が容易に
- サービス毎の仮想ネットワーク
  - 複数のサービスで実際のノードやネットワークを共有し，
    センサ・アクチュエータの量を削減
  - 業務別ネットワークと比較し，広域から多種の情報を取得
    することで効率的な制御が可能

5

## 仮想ネットワークの概要(1/2)

- 仮想ネットワークは1つの基地局，複数のセンサ・ア
  クチュエータノードを保持
- 仮想ネットワークは2種類に分類
  - global network :
    全体に共通するサービス(施設運用)
  - local network :
    特定の場所に限定したサービス(店舗等)



: 基地局　　: local network

: ノード(センサ・アクチュエータ)　　: global network

6

## 仮想ネットワークの概要(2/5)

- global network
  - 施設に唯一存在
  - 全てのノードが所属
  - 所属するプロセス(後述)はlocal networkより優先
    - 事故などを防ぐため施設管理を優先
    - プロセスの配布はノードの能力や属性を参照
- local network :
  - 施設内に複数存在
  - 業務サービス実施のためのノードのみ所属
  - 複数のlocal networkでノードを共有しない

7

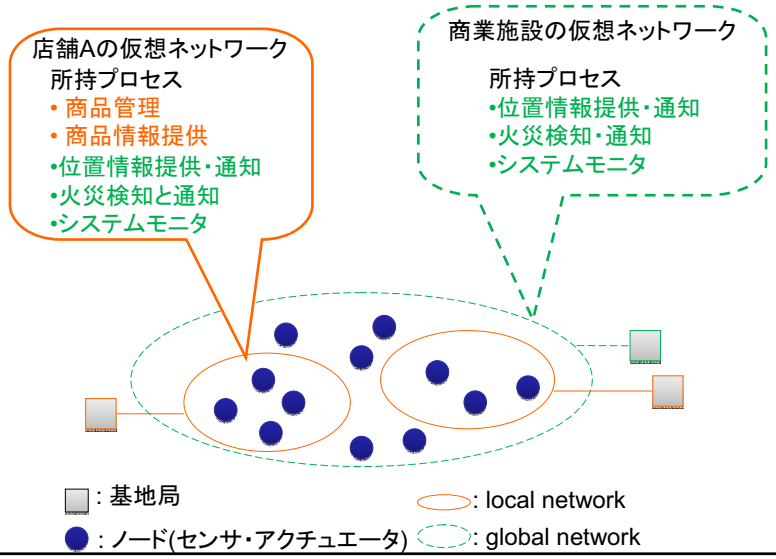
## 仮想ネットワークの概要(3/5)
## - 仮想ネットワークにおけるプロセス -

- 施設のサービスを実現する処理
  - 情報の提供
  - アクチュエータの制御
  - パケットの転送
- 1つの仮想ネットワークに所属
  - 基地局がノードに無線通信で配布
- 1つのノード上で複数のプロセスが動作

8

## 仮想ネットワークの概要(4/4)
## - 商業施設における仮想ネットワークの例 -

• 緑は商業施設の仮想ネットワークが管理するプロセス
• 橙は店舗Aの仮想ネットワークが管理するプロセス




9

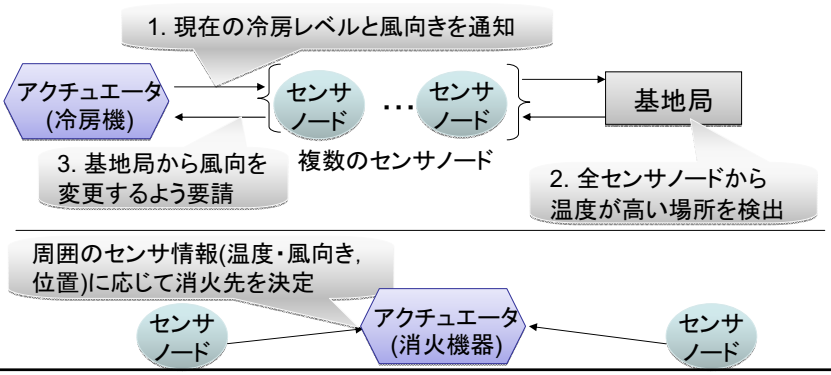
## 仮想ネットワークの構成(1/5)

- 基地局：仮想ネットワークの管理
  - 仮想ネットワーク中のセンシングデータの収集
  - 所属する仮想ネットワークの上のノードに対し プログラムを配布
  - アクチュエータに制御パケットを送信
- センサノード：センシングと経路の形成
  - センシングデータを提供 (基地局へは常に提供，要求に応じ隣接ノードにも提供)
  - 隣接ノードとの協調や制御 (アクチュエータの制御・結果観察，負荷分散)

10

## 仮想ネットワークの構成(2/5)

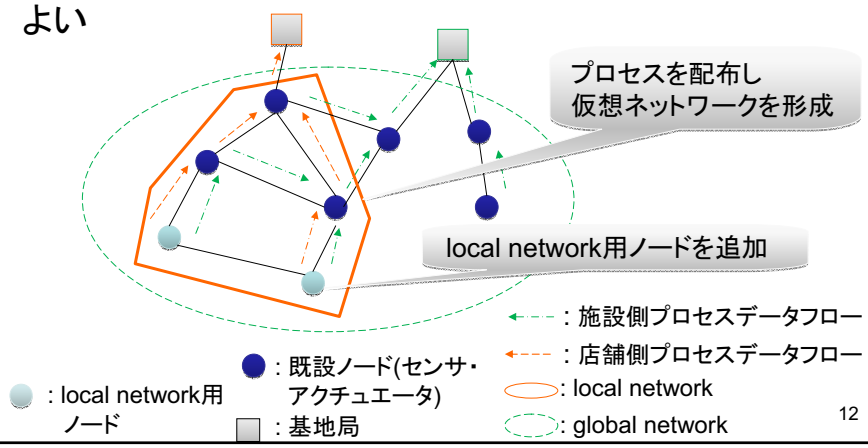
- アクチュエータ： 実世界へ作用する機器, 2つの動作モードを持つ
  - 基地局による制御：基地局からの制御パケットで動作
    (ノードの動作決定のための処理量が大きい場合に利用)
  - 自律的動作：周囲のノードから情報を取得し，判断することで動作
    (応答性の向上を目的として利用)




11

## 仮想ネットワークの構成(3/5)
## - 商業施設において新規業務を実施する場合 -

- ノードを共用し経路を形成することで
  業務毎の専用ネットワークが不要
- local network用ノードをネットワークに追加すれば
  よい

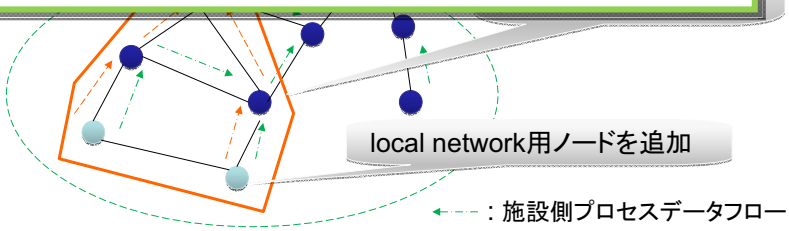



12

## 仮想ネットワークの構成(3/5)
## - 商業施設において新規業務を実施する場合 -

- ノードを共用し経路を形成することで
  業務毎の専用ネットワークが不要
- local network用ノードをネットワークに追加すれば

*•複数のサービスで実際のノードを共有することで*
*センサ・アクチュエータの量を削減可能*

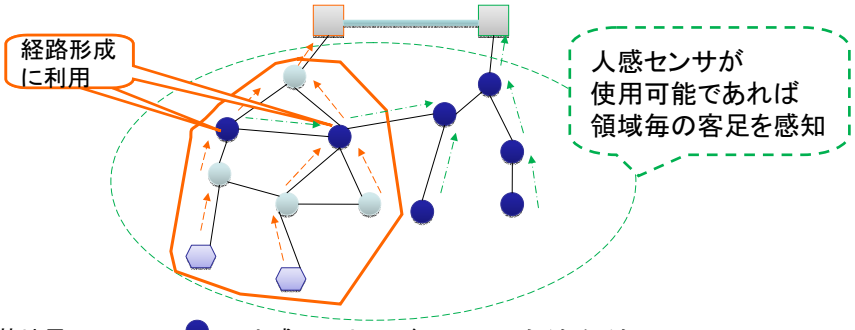



: local network用ノード
: 既設ノード(センサ・アクチュエータ)
: 基地局
: 施設側プロセスデータフロー
: 店舗側プロセスデータフロー
: local network
: global network

13

## 仮想ネットワークの構成(4/5)
## - 仮想ネットワークを利用したアクチュエータの制御(1/2) -

- 2種類のサービスから構成
  - 客足感知サービス(global network): 人感センサからデータを収集
  - 空調管理 サービス(local network) : 温度センサからデータ収集,
    空調機の風向・設定温度管理

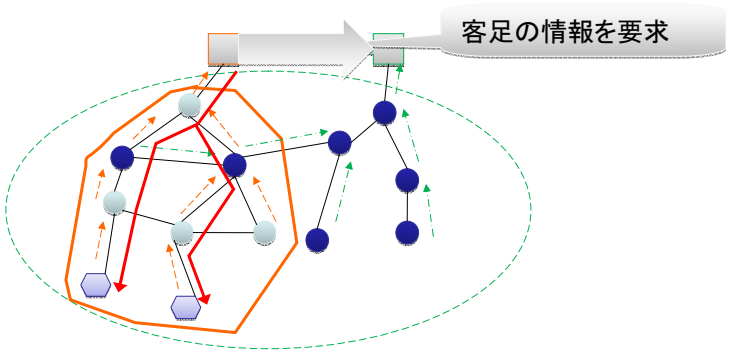



: 基地局
: 空調機器
: 温度センサノード
: 人感センサノード
: local network
: global network
: 有線/無線LAN
: 客足モニタリングデータフロー
: 空調モニタリングデータフロー

14

## 仮想ネットワークの構成(5/5)
## - 仮想ネットワークを利用したアクチュエータの制御(2/2) -

- 温度だけでなく人の密度を用いた効率的な空調
  - 客足が減少する領域の冷却能力を下げ
    客足が増加するにある領域を冷却能力を向上

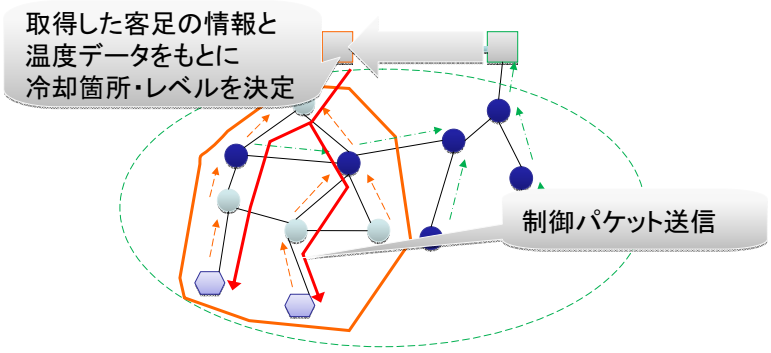



: 基地局　: 人感センサノード　: 有線/無線LAN
: 空調機器　: local network　: 客足モニタリングデータフロー
: 温度センサノード　: global network　: 空調モニタリングデータフロー

15

## 仮想ネットワークの構成(5/5)
## - 仮想ネットワークを利用したアクチュエータの制御(2/2) -

- 温度だけでなく人の密度を用いた効率的な空調
  - 客足が減少する領域の冷却能力を下げ
    客足が増加するにある領域を冷却能力を向上



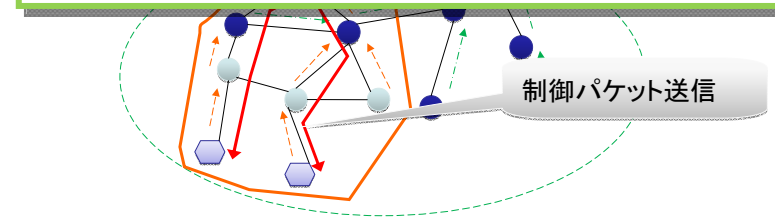

: 基地局　: 人感センサノード　: 有線/無線LAN
: 空調機器　: local network　: 客足モニタリングデータフロー
: 温度センサノード　: global network　: 空調モニタリングデータフロー

16

## 仮想ネットワークの構成(5/5)
## - 仮想ネットワークを利用したアクチュエータの制御(2/2) -

- 温度だけでなく人の密度を用いた効率的な空調
  - 客足が減少する領域の冷却能力を下げ
    客足が増加するにある領域を冷却能力を向上

- *広域から多種の情報を取得することで効率的な制御が可能*




: 基地局
: 空調機器
: 温度センサノード
: 人感センサノード
: local network
: global network
: 有線/無線LAN
: 客足モニタリングデータフロー
: 空調モニタリングデータフロー

17

## センサ・アクチュエータノード上の
## 仮想ネットワークにおける検討事項(1/2)

- ノードのプロセス・資源管理
  - local, global network間のプロセス優先度管理
    (global > local)
  - global networkのプロセスによるlocal network
    用ノードの占有(サービスが実施不能)への対処
  - プロセスが利用する資源の競合・不足への対処

- ノードの移動に起因する問題
  - 移動体を経由した通信，移動体への通信による
    資源アクセスの管理

18

## センサ・アクチュエータノード上の仮想ネットワークにおける検討事項(2/2)

- ノード間での抽象的な問合わせ方法とその対処
  - 例: アクチュエータによる「火災検知に利用可能な情報」の要求と取得したデータの処理方法
- 基地局間での協調方法
  - 各基地局が持つ機能の発見と利用方法
- 消費電力の削減
  - 複数の仮想ネットワークに所属するノードの負荷
- セキュリティ
  - ノードの不正な利用への対処

19

## 検討事項への対処方法(1/5) - ノードのプロセス・資源管理 (1/2) -

- ノード上で資源利用が変化する際に検査
  - プロセス実行前後
  - デバイスからの状態通知
  - プロセスの利用する資源が変化した場合
- プロセスが利用する資源を管理
  - アクチュエータ，センサ，処理時間，記憶領域，属性
- 資源毎の管理方針の提供
  - 単一プロセスがアクチュエータを占有
  - センサはキャッシュを用い複数プロセスから読出し

20

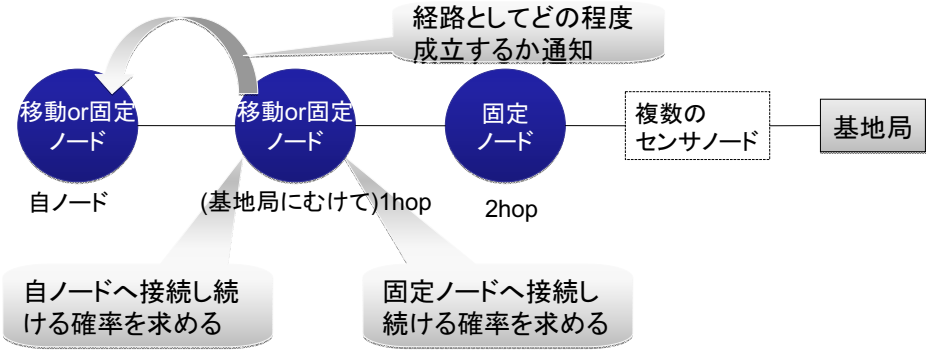
## 検討事項への対処方法(2/5)
## - ノードのプロセス・資源管理 (2/2) -

- 資源検査結果のプロセスへの反映
  - ルーティングなど一部の機能を実行
  - 競合する資源は使用しない
  - 実行可能になるまで待機
  - 基地局に競合を通知
  - 他のプロセスとのバルク処理
  - 隣接ノードにプロセスを渡し実行
    (以後，パケットは隣接ノードに転送)



## 検討事項への対処方法(3/5)
## - ノードの移動に起因する問題 (1/2) -

- 通信の目的，隣接ノードの性質に応じ対応
- 基地局への通信：1hop先に2hop先の固定ノードと自ノードが接続し続ける確率により管理
  - RSSIと単位時間あたりのRSSI減衰量で算出



## 検討事項への対処方法(4/5)
## - ノードの移動に起因する問題 (2/2) -

- 移動するアクチュエータが環境情報を収集し対処を行う場合を想定
  - アクチュエータが事象に対し作用できる範囲はノードの通信距離(30～ 70m)より小さい場合を想定
- 広域の情報を収集する場合：基地局に問合わせ
- 周囲の事象に対し即座に対処する場合(消火など)：1hop以内のノードからデータを収集
  - 1hopより外の事象をセンサが通知しても対応できない
  - 短時間で情報収集が可能
    - 短時間での情報収集が不要であれば基地局に問合わせ

23

## 検討事項への対処方法(5/5)
## - その他問題点への対策 -

- ノード間での抽象的な問合わせ方法とその対処
  - メタデータを利用
- 基地局間での協調方法
  - サービス発見機構の利用(Jini, UPnP)
- 消費電力の削減
  - ノードの稼動期間を管理し，プロセスの分配を最適化
  - 仮想ネットワーク間でルーティングや通信周期の最適化
- セキュリティ
  - 計算量が少ない認証機構を利用し認証(楕円曲線暗号系など)
  - ハードウェアのサポートを利用
  - 認証機構を用いた資源アクセスの管理

24

## おわりに

- センサ・アクチュエータを用いた仮想ネットワークについて述べた
  - ノード上で複数のプロセスが稼働，プロセスは1つの仮想ネットワークに所属
  - 複数のサービスがノードを共用することで，新規に必要な機器のコストを削減
- 想定するネットワークの検討事項とその対策について述べた
  - ノードのプロセス・資源管理，ノードの移動に起因する問題，ノード間での抽象的な問い合わせ方法とその対処，基地局間での協調方法，消費電力の削減，セキュリティ
- 今後ノード上のプロセス・資源管理機構について考察
  - 既存のセンサノード上の資源管理機構，信頼性向上のための機構について調査
  - 資源管理機構の管理範囲の明確化
  - 仮想ネットワークの階層についての検討

25

## 付録：センサ・アクチュエータネットワークの運用例

- EcoBino：空調管理サービス
  - 店舗全体で15%～25%の電力削減効果
  - CO2の排出量に換算すると、年間10トン～16トンの削減に相当
  - 温度センサ，空調機を利用
- Smart Attire：衣類の情報管理
  - 着用者に服の形状が向いているか提示
  - 加速度センサや圧力センサを利用

## 付録：仮想ネットワークの階層についての検討

- サービスによる階層だけでなく物理的な接続について考慮
- 認証済み/認証中/認証不能といった認証による分類についても考慮
- 機器の管理のための階層の考慮
  - 遠隔から任意の機器を操作

## 付録：認証による資源利用の考察

- 認証により資源が利用可能かを管理
  - ファイルシステムの属性管理などを参考に
- ノード上では次の内容について検討が必要
  - 一時的な認証と恒久的な認証が存在
    - ノードの移動能力を考慮
  - ノードの識別方法
    - ノードがIDを持たないデータセントリックな環境への対策
  - 認証実施箇所の限定
    - 攻撃を行うノードがどのように参入するかを考察

# DTN におけるデータとノードの特性を考慮した選択式ルーティングプロトコル

陶山 優一† 横田 裕介†† 大久保 英嗣††
† 立命館大学大学院理工学研究科 †† 立命館大学情報理工学部

## 1 はじめに

近年, 断続的な接続性や通信遅延が発生する環境においてメッセージ配送を可能にする技術として Delay/Disruption/Disconnect Tolerant Network(DTN) が注目されている [1] . DTN は, 災害情報ネットワーク, 山間部や海洋部などにおける環境観測ネットワーク, 車車間ネットワークといった無線端末を含んだ比較的不安定なアドホックネットワークへの適用が求められている. アドホックネットワークでは, 従来の有線ネットワークと比較し, 様々な問題が発生する.

まず, アドホックネットワークは, 通信帯域や電力資源, 記憶容量の制約が厳しく, また近隣ノードとの電波干渉が発生する. このため, 保存するデータやトラフィックなどのネットワークリソースの消費を抑える必要がある.

次に, 送信するデータのプライオリティを考慮する必要がある. これまでの有線ネットワークでは, ネットワークが安定しており, 高速にデータを送ることが可能であったため, データのプライオリティを考慮する必要性がなかった. しかし, アドホックネットワークでは, 通信帯域の制限や電波干渉, ネットワークの切断などにより, 即座にデータを送ることができない場合が多々発生する. このような場合, 人命に関わるデータやイベント検知データのような緊急性の高いデータを優先的にサーバに送信する必要がある.

さらに, アドホックネットワークでは, 適用されるシステムによって様々な移動パターンを持つ. また, 大規模なネットワークの場合, エリアによってノードの移動パターンが異なる場合が考えられる. このような環境において, 最適なルーティングパフォーマンスを得るためには, 事前にシステム設計者がノードの移動特性を把握し, 最適なルーティングプロトコルを実装する必要がある.

DTN では, 様々なルーティングプロトコルがこれまでに提案されている [2, 3, 4] . しかし, 単一のルーティングプロトコルでこれらの要求を満たすことは困難である. このため, 本研究では, ネットワーク上のノードとデータの特性を考慮し, 動的にルーティングプロトコルを選択する手法を提案する. これにより, 複雑なネットワーク環境においても高いルーティングパフォーマンスを得ることが可能になる.

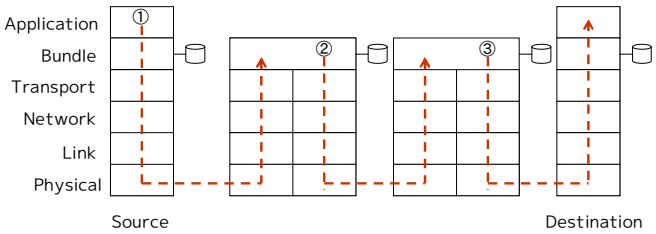



図 1 DTN における転送方法 (文献 [1] より抜粋)

## 2 DTN における転送方法

これまで広く用いられてきた TCP/IP では, アプリケーションデータをセグメントに分割し, 宛先ノードで再結合することによりデータの送信を行う. しかし, DTN では, アプリケーション層とトランスポート層の間にバンドル層と呼ばれるストレージを持つ層を用意し, 送信されたセグメントデータは中継ノードに着く度, アプリケーションデータに復元され, バンドル層のストレージに一時保存される. このようにアプリケーションデータ単位で転送していくことにより, たとえばデータ送信中に経路が途中で切断された場合でも, 送ることができるノードまで送っておき, 経路が回復次第, 転送を途中から行うことが可能となる. DTN における転送方法の概要を図 1 に示す.

## 3 DTN Routing Protocol

DTN では, いくつものルーティングプロトコルがこれまでに提案されてきている. 本章では, 既存のルーティングプロトコルをデータ配置方式の面と転送方式の面で分類し, 比較を行う.

### 3.1 データ配置方式での分類

データ配置方式の面では, Replication と Forwarding に分類することができる. Replication では, データを転送する際, データを中継したノードはそのデータを自身のストレージに複製する. 一方, Forwarding では, 転送したデータは削除し, エッジノードのみがデータを所持する. それぞれのデータ配置方式の概要を図 2 に示す.

### 3.2 データ転送方式での分類

データ転送方式の面では, Epidemic と Utility-Based に分類することができる. Epidemic では, フラッディング

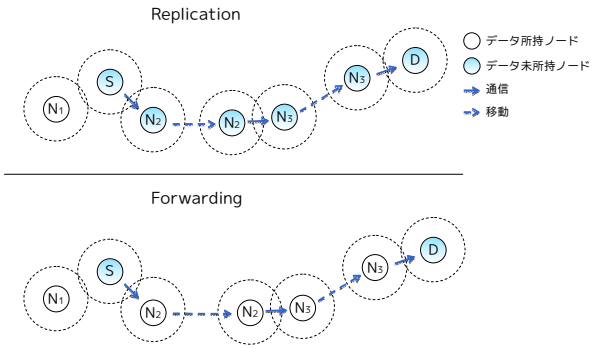



図 2　データ配置方式における分類

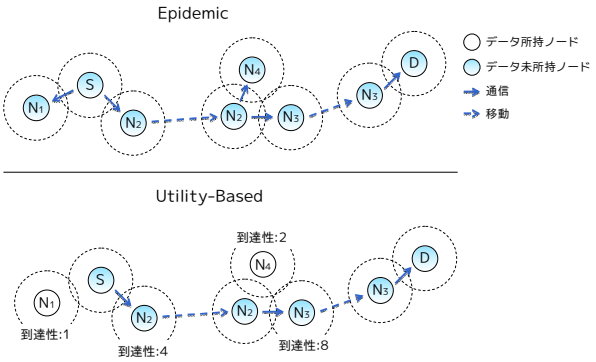



図 3　データ転送方式における分類

を用いて転送を行う. 一方, Utility-Based では, 各ノードから目的ノードまでの到達性を元に転送するノードを選択する. 到達性の算出方法は, ルーティングプロトコルによって様々であるが, ノードの移動履歴を元に算出するものが一般的となっている. それぞれのデータ転送方式の概要を図 3 に示す.

### 3.3　既存ルーティングプロトコルの比較・問題点

まず, Replication 方式では, 多くのノードがデータおよびその送信タスクを持つ. このため, Forwarding 方式と比較し, データの配送確率が向上し, データの伝送遅延も軽減される. しかし, 多くのノードのストレージを消費し, データ転送によるトラフィックも増大するため, ネットワークリソースの消費が大きいことが問題点として挙げられる. 一方, Forwarding 方式では, エッジノードしかデータおよびその送信タスクを持たないため, ネットワークリソースの消費は小さいが, データの伝送遅延は大きくなる.

次に, Epidemic 方式では, フラッディングを用いることから, Utility-Based 方式で必要な到達性を求めるためのオーバヘッドがかからないことが利点である. しかし, ノードの移動パターンに規則性がある場合, Utility-Based 方式を用いた場合より, 配送確率が低下する. 一方, Utility-Based 方式では, ノードの移動パターンに規則性がある場合, 到達性を元に転送することにより, 高い配送確率を得ることができるが, その到達性を求めるためのオーバヘッドが問題となる.

データ配置方式の分類では, データの伝送遅延とネットワークリソースの消費がトレードオフの関係になっているため, どちらか 1 つのルーティングプロトコルでは, 1 章で述べた要求を満たすことができない. また, データ転送方式の分類では, それぞれ適したノードの移動パターンが異なるため, システム開発者はその都度, システムにあわせて用いるルーティングプロトコルを実装しなければならない. さらに, 同一システム内においても, エリア毎にノードの移動パターンが異なる場合が想定される. このような場合, エリア毎に異なるルーティングプロトコルを実装することは困難である. 以上の点から, 状況に応じてルーティングプロトコルを切り替えることが求められる.

## 4　選択式ルーティングプロトコル

本研究では, 送信するデータ毎に, ノードの特性とデータのプライオリティから用いるルーティングプロトコルを動的に切り替える選択式ルーティングプロトコルを提案する. 本提案手法におけるルーティング処理は, Selection 処理, Transfer 処理, ACK 処理の 3 つに分類される. 以下, 各処理内容について述べる.

### 4.1　Selection 処理

Selection 処理では, 用いるルーティングプロトコルの選択を行う. ルーティングプロトコルの選択は, 次の 3 つのパラメータを元に行う.

- データのプライオリティ
  アプリケーションから提供されるものと想定する.
- 目的ノードへの到達性 (移動パターンに規則性があると判断した場合のみ)
  周辺ノードの移動履歴から目的ノードへの到達性を評価する.
- ネットワーク上のノードの移動パターン
  一定周期毎に固定ノードのみ周辺の移動特性を評価する. 移動ノードは, 固定ノードからそのエリアの移動特性を取得する.

### 4.2　Transfer 処理

Transfer 処理では, 送信データに付随するメタデータに基づき, データの転送を行う. メタデータには, PacketID, TimeStamp, RoutingProtocol の種類, データのプライオリティ, その他の情報 (Hop 数, TTL など) を持つ.

RoutingProtocol は, Replication か Forwarding のみの記述となる. Epidemic か Utility-Based の選択は, それぞれのノードが持つ現在のエリアの移動特性から選択する.

ノードのストレージは, それぞれ制限があるものとし, 容量を超える場合, TimeStamp とデータのプライオリティを基に置換する.

### 4.3 ACK 処理

ACK 処理では, 送信データに対する ACK の送信を行う. DTN では, 中継ノード間ではその都度コネクションを確立し通信を行うが, 通信しているセグメントに対する ACK は中継ノードに対して返信されるため, データの発信ノードと宛先ノード間はコネクションレスな通信となる. また, データが到達した後, ACK を返そうとしても発信ノードまでの経路が存在する保証はない. このため, ACK に関しても送信データと同様, DTN のルーティングプロトコルを用いて送信を行う.

また, 本提案手法では, 転送に用いたルーティングプロトコルによって ACK を返す必要のあるノードの数が異なる. このため, Forwarding ACK と Replication ACK の 2 種類に分類し, ACK の転送量の調節を行う.

## 5 おわりに

本稿では, 送信するデータ毎に, ノードの特性とデータのプライオリティから用いるルーティングプロトコルを動的に切り替える選択式ルーティングプロトコルについて述べた. 本提案手法により, データのプライオリティに応じた通信制御を行うことが可能となる. また, システム開発者は, システム毎に異なるルーティングプロトコルを実装する必要がなくなり, エリアによって様々な移動パターンを持つような環境においても対応することが可能となる.

今後の予定として, 引き続き関連研究の調査を行い, 到達性および移動特性の評価手法の検討, 詳細な各処理方法の検討を行い, 終わり次第, シミュレーションによる評価を行いたいと考えている.

## 参考文献


[1] Disruption Tolerant Networking, http://www.darpa.mil/sto/solicitations/DTN/DTN_industryday.pdf(2004).

[2] A. Balasubramanian, B. N. Levine, and A. Venkataramani: DTN routingas a resource allocation problem, Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications, August, 2007.

[3] E. Jones, L. Li, and P. Ward. Practical Routing in Delay-Tolerant Networks. In Proc. ACM Chants Workshop, pages 237–243, August. 2005.

[4] J. Burgess, B. Gallagher, D. Jensen, and B. N. Levine. MaxProp: Routing for Vehicle-Based Disruption-Tolerant Networks. In Proc. IEEE Infocom, April, 2006.

# DTNにおけるデータとノードの特性を考慮した選択式ルーティングプロトコル

立命館大学 理工学研究科
大久保・横田研究室
M1 陶山 優一

1

## 発表内容

- はじめに
  - アドホックネットワークにおける問題点・要求
- 既存技術
  - DTN(disruption-tolerant networks)
  - DTN Routing Protocolの分類
  - 既存技術の問題点
- 選択式ルーティングプロトコル
  - Selection処理
  - Transfer処理
  - ACK処理
- おわりに

2

# はじめに

- アドホックネットワークの特徴
  - ノードの移動によるネットワークの切断
    - 分断されたネットワークにデータを送りたい
    - DTN技術
  - 限られた通信帯域・電力資源・記憶容量，電波干渉
    - ネットワークリソースの消費を低減したい
  - データによってプライオリティが異なる
    - 例)警備システム:平常時の観測データ，不審者発見時の観測データ

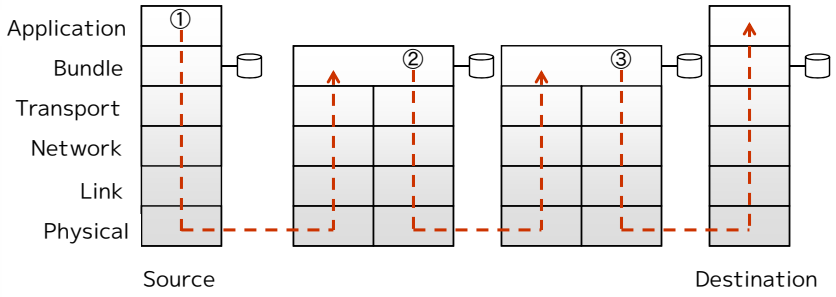
    →プライオリティの高いデータの伝送遅延を小さく

ネットワークリソースの消費を抑えつつ，プライオリティの高いデータの伝送遅延を小さく

3

# DTN(disruption-tolerant networks)

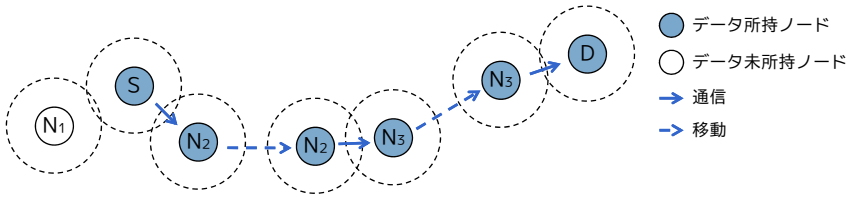
- DTNとは
  - 大きな伝送遅延,データリンクの切断・接続が頻繁に発生するネットワーク環境でのメッセージ配送を可能にする技術
  - ストアアンドフォワード方式で転送



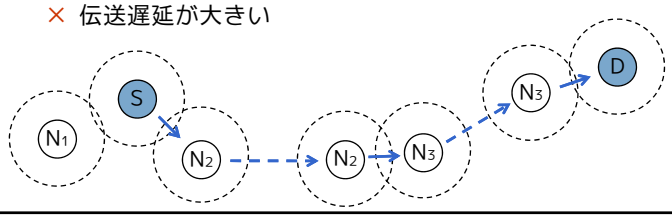

Disruption Tolerant Networking
http://www.darpa.mil/sto/solicitations/DTN/index.htmより一部引用

# DTN Routing Protocolの分類
## データ配置方式

- Replication：データを中継したノード全てがデータを所持
  - ○ 伝送遅延が小さい
  - × ネットワークリソースの消費が大きい



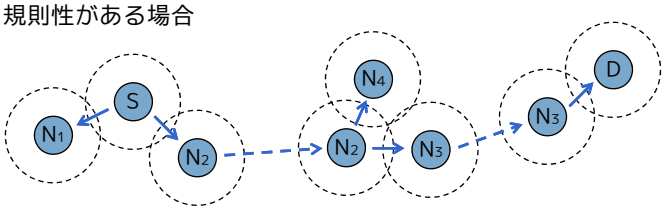

- Forwarding：エッジノードのみデータを所持
  - ○ ネットワークリソースの消費が小さい
  - × 伝送遅延が大きい



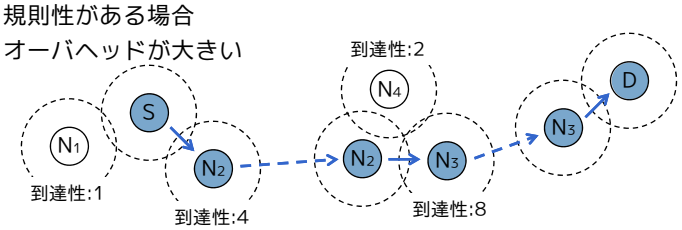

5

# DTN Routing Protocolの分類
## 転送方式

- Epidemic：フラッディングを用いて転送
  - ○ オーバヘッドが小さい
  - × 規則性がある場合




- Utility-Based：目的ノードまでの到達性(過去の移動履歴などから算出)に基づき，転送
  - ○ 規則性がある場合
  - × オーバヘッドが大きい




6

## 既存技術における問題点

- 要求：ネットワークリソースの消費の低減・プライオリティが高いデータの伝送遅延の低減

- ネットワークリソースの消費を抑える → 伝送遅延大
- 伝送遅延を小さく → ネットワークリソースの消費大

→単一プロトコルでは要求に適応できない

- 適用するシステムによってノードの移動特性は様々

→適用対象に応じたルーティングプロトコルを実装しなければならない

- エリア毎に移動特性が異なる場合も

7

## 選択式ルーティングプロトコル

- 送信するデータ毎に，ノードの特性・データのプライオティから用いるルーティングプロトコルを動的に選択

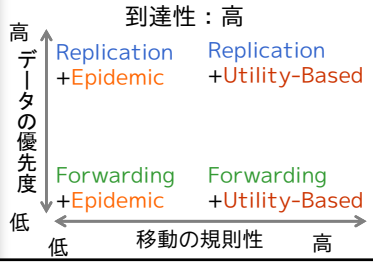
ネットワークリソース消費の低減，
プライオリティの高いデータの伝送遅延の低減を可能に

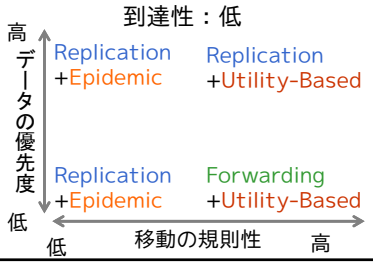
- Selection処理
  - 用いるルーティングプロトコルを選択
- Transfer処理
  - 選択したルーティングプロトコルでのデータ送信
- ACK処理
  - ACKのキャッシュ・送信

8

# Selection処理

- 3つのパラメータに基づき，用いるルーティングプロトコルを選択
  - データの優先度
    - アプリケーションから提供
  - 目的ノードへの到達性(規則性がある場合のみ)
    - 各ノードの過去の接続履歴から評価
  - ネットワーク上のノードの移動特性
    - 一定周期毎に固定ノードが周辺の移動特性を評価
    - 移動ノードは近隣の固定ノードから取得




9

# Transfer処理

- PacketID, TimeStamp, RoutingProtocolの種類，データのプライオリティ，その他の情報(Hop数，TTLなど)をメタデータとして所持
- Epidemic or Utility-Basedはデータを持つノードが持つローカルの移動特性から選択

| PacketID | TimeStamp | RoutingProtocol | Priority | その他 |
|---|---|---|---|---|
| 1001 | 2008-01-01 23:03:20 | Replication | 5 | |
| 2001 | 2008-01-02 01:00:40 | Forwarding | 1 | |
| ⋮ | ⋮ | ⋮ | ⋮ | |

- 蓄積するデータはTimeStamp，プライオリティを基に置換

10

# ACK処理

- Acknowledgement
  - アプリケーションデータレベルのACK
- ルーティングプロトコルによって2種類のACKに分類
  - Forwarding ACK
  - Replication ACK
- ACKの返信先
  - Forwarding ACK：ソースノード+転送中のエッジノード
  - Replication ACK：ソースノード+中継ノード
- 返信先の数：Forwarding ACK < Replication ACK
- ACKの種類に応じて，頒布するACKの量を調整

11

# おわりに

- まとめ
  - DTNにおけるデータとノードの特性を
    考慮した選択式ルーティングプロトコルの提案

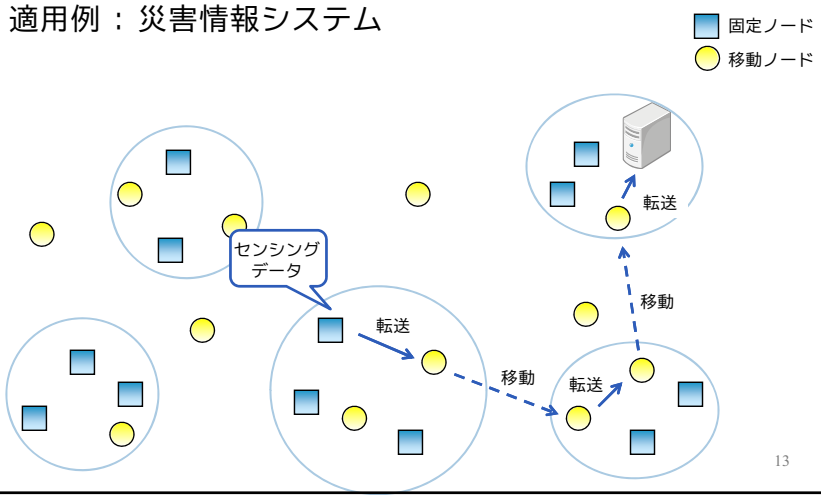
  → ネットワークの移動特性を意識することなく，
    適切なルーティングプロトコルを選択可能に

  → プライオリティの高いデータの伝送遅延を短縮＆
    プライオリティの低いデータのネットワーク
    リソース消費を低減
- 今後の予定
  - 関連研究の調査
  - 到達性・移動特性の評価方法の検討
  - 詳細な各処理方法の検討
  - シミュレーションによる評価

12

# 想定環境

- 中～大規模なネットワーク
- 固定ノードと移動ノードが混在
- 適用例：災害情報システム




13

# センサネットワークを用いたデバイス制御における動的な構成変化に対応するフレームワーク

植田 裕規 †
† 立命館大学大学院理工学研究科

## 1 はじめに

近年，半導体技術や無線通信技術の発達により，センシング機能を持ったセンサノードを複数用いて構成されるセンサネットワークへの関心が高まっている．センサネットワークは，様々な種類が存在し，これらを組み合わせることで多種多様な環境情報の取得が可能となる．一方，空調機器や照明などの外部から制御可能なデバイスやネットワーク上に存在するある特定の機能を持ったオブジェクトが互いに連携し，ユーザに対して新たなサービスを提供する UPnP や Jini などの技術がある．

本研究では，センサネットワークと多種デバイスを組み合わせた環境を想定する．このような環境において，センサネットワークから取得した環境情報をイベントとし，ユーザの環境に合わせたサービスを提供するシステムを研究の対象とする．環境情報の取得にセンサネットワークを利用することで，センサ単体による点の観測ではなく，面や空間の観測が可能であるため，温度の平均の値をイベントとしたり，局所的な環境の変化によるデバイスの誤動作を防ぐことが可能になる．

本研究で対象とするシステムにおいて，ユーザの望むサービスを提供するためには，何らかの形式で要求を記述しシステムに登録する必要がある．しかし，システムのネットワーク上には，センサやデバイスが多数存在することが考えられるため，ユーザがこれらの機能や位置のすべてを把握して記述することは困難である．さらに，デバイスの追加や削除を考慮した場合，デバイスの把握がより困難となることが考えられる．以上のような問題を解決するために，センサネットワークを用いたデバイス制御における動的な構成変化に対応するフレームワークを提案する．本フレームワークを適用することで，対象とするシステムを利用するユーザの負担を軽減することが可能である．

以下，本稿では，2 章で既存研究と提案フレームワークについて述べる．3 章でそのアーキテクチャについて述べ，4 章で本フレームワークで用いる動作定義について述べ，5 章で本稿のまとめと今後の予定について述べる．

## 2 センサネットワークとデバイス制御

### 2.1 既存研究と機能要件

既存システムである Easy Living[1] では，サービスの記述において，デバイスとユーザの位置が重要であるとし，個々のデバイスに対してサービスを記述するのではなく，場所に対してサービスの記述を行う．これにより，ユーザの位置によって必要とするデバイスの切り替えを行うことが可能である．しかし，Easy Living は，環境情報の変化の通知など非同期なイベントの通知をサポートしていないため，“機能オブジェクトの処理が終了した場合，これをユーザに通知する” といった記述を行うことが困難である．これは，環境の変化によってデバイスを制御することを目的とした本研究において不適切である．したがって，本研究で対象とするシステムにおいて，環境の変化をイベントとして伝達する機構が必要となる．

一方，STONE[2] では，ワールドワイドなネットワーク環境を想定したサービスの合成を行うために，サービスリゾルバと呼ばれる名前解決機構とサービスグラフと呼ばれる機能オブジェクトの合成機構を用いて合成を行う．サービスの合成は，サービスグラフと各オブジェクトが持つインタフェースに基づいて行われる．したがって，サービスの合成を記述するエンドユーザは，インタフェースを利用するために，オブジェクトのデータフォーマットや通信プロトコルなどの標準化された形式の知識が必要となる．また，サービスの合成は，サービスグラフに依存しているため，ユーザ環境の変化により制御対象の変更が生じた場合，クライアントアプリケーションによってサービスグラフを変更する必要がある．

その他の既存システムに関する調査 [3] では，センサやデバイスを管理するためのディレクトリサービスの重要性を述べている．

このような点から，本フレームワークが満たす機能要件を次のように定めた．

- 空間に対する動作の記述
- システム構成の動的な変化に対応可能
- 非同期なイベントのサポート

本研究の目的は，エンドユーザのシステム利用時における負担軽減である．センサネットワークやデバイスの変化をエンドユーザに如何に意識させないかを目指す．これにより，システムの構成が変化した際に，ユーザはシステムにおいて予め定義されたサービスに対して，新たに変更を加える必要がなくなることが考えられる．このことは，システム構成の動的な変化に対応するという特色を活かすことに繋がると考えられる．また，これらの既存システムでは，センサ同士でネットワークを構築している環境に対応していない．本フレームワークでは，センサネットワークの利用を想定しているため，容易に対応することが可能である．

### 2.2 提案フレームワーク

本研究で提案するフレームワークは，家屋におけるホームネットワークのような小規模なローカルネットワークにシステムを構築することを想定している．このような環境において，窓やイスなどに配置されたセンサがセンサネットワークを形成しており，このセンサネットワークがイベントソースとなるものとする．また，室内に設置された空調機器や照明などユーザや空間に対して何らかの動作をするデバイスが制御対象となる．

本フレームワークは，システム構成の変化に対して動的に対応するために，ディレクトリサービスを用いてイベントソースと制御対象を抽象化する．ここで，システム構成の変化とは，故障や換装などによりセンサネットワークやデバイスが，システムから取り除かれたり新たに設置されることを意味する．イベントソースであるセンサネットワークを登録する際に，センサネットワークの取得可能な環境情報や設置場所などをディレクトリサービスに登録する．同様に，制御対象となるデバイスを登録する際には，デバイスの提供機能や設置場所などをディレクトリサービスに登録する．これらのディレクトリサービスに登録された情報を利用することでイベントソースの特定と制御対象の特定が行われる．本フレームワークは，センサネットワークやデバイスが登録するときに必要となるインタフェースを提供する．

また，本フレームワークは，空間に対する動作定義を可能とする．具体的には，部屋の名前，ユーザの位置，室内の平均温度などをイベントソースとして，“ユーザの入室した部屋の空調機器を操作し，室内平均温度を 28 °C に調節する” といった動作を定義することができる．なお，記述の詳細は 4 章で述べる．

## 3 フレームワークのアーキテクチャ

本研究で提案するフレームワークのアーキテクチャは，センシングデータをコンテキストへと変換することでデバイスを制御するという基本構想からなる．図 1 にその構成を示す．

フレームワークは 3 つの要素から構成される．すなわち，Device(Mobile Computer), Middleware(Gateway), Sensor Network である．これらの各要素の概要について述べる．

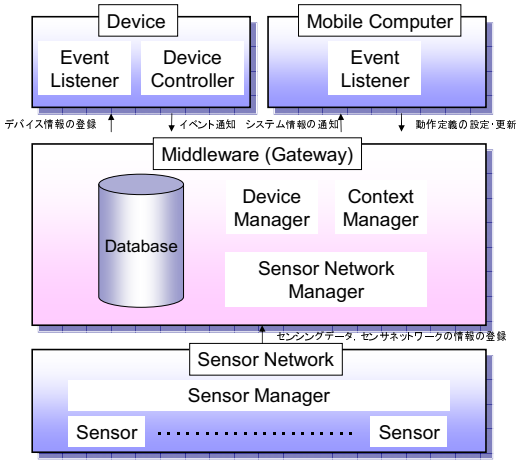



図 1 提案フレームワークのアーキテクチャ

**Sensor Network**

- Sensor
  センサノードを表し各ノードに搭載されるオペレーティングシステムやミドルウェアなどの制御モジュールを指す．本アーキテクチャでは，これに対して直接関与することはないものとする．
- Sensor Manager
  Sensor から環境情報を取得し，Middleware へと送信する機能を持つ．本フレームワークからは，後述する Sensor Network Manager にセンサネットワークの情報を登録するための API とインタフェースが提供される．

**Middleware**

- Database
  エンドユーザの設定した動作定義，デバイスの情報，センサネットワークの情報，実環境情報などを格納するためのデータベースを表す．各 Manager は，この Database から必要な情報の参照を行う．
- Context Manager
  エンドユーザの設定した動作定義の登録・削除や，動作定義に基づいたデバイスの制御などのコンテキストに関する管理を行う．また，Context Manager は，システムを利用するユーザを管理する機能を持つ．
- Device/Sensor Network Manager
  デバイス/センサネットワークの登録や削除

など，システム構成に関わる操作を行う機能を持つ．Device/Sensor Network Manager は，ディレクトリサービスを提供することでセンサネットワークやデバイスの抽象化を行い，これらを管理する．

**Device/Mobile Computer**

- Event Listener
  Middleware の Context Manager からのイベント通知メッセージを受信する機能を持つ．これによって，コンテキストの変化をデバイスが検知可能となる．フレームワークによって提供される API とインタフェースを使用することで，Device Manager に自身の提供する機能や位置に関する情報を登録し，Context Manager からのイベント通知メッセージを受信するまで待機する．
- Device Controller
  Event Listener によってコンテキストの変化が検知された場合，受信した制御メッセージに応じたデバイスの制御を行う．これにより実空間に対してデバイスを作用させる．

## 4 フレームワークにおける動作記述

提案するフレームワークでは，動作の記述方式に ECA(Event-Condition-Action) ルールを採用している．フレームワークのプロトタイプにおけるシナリオとして “イベントソースに照度とユーザの現在位置を利用し，ユーザの方向にカメラを向ける” というシナリオを考えている．このシナリオを基に具体的な動作記述例について述べる．

ECA ルールを用いた動作の記述例

DEFINE $Camera\_Move\ XXXX$
EVENT $Enter\_Room$
  WHEN $9:00-17:20$
  WHERE $User\_Location\_Room$
IF $AVG(light) = Bright$
THEN
  DO $Camera$
  WHERE $User\_Location\_Room$
  HOW $TRAIN$
  INFORMATION $User\_Location\_Coordinates$

ECA ルールを用いた動作の記述例における各項は次のように解釈される．

**DEFINE** ユーザ ID と登録したユーザ名を指定する．ユーザ名により動作定義の優先度を決定することで複数のユーザに対応させる．ここでの “XXXX” はユーザ名を表す．

**WHEN** デバイスを制御する時間帯を指定する．日付を指定することで揮発性を持つ動作定義を行い，指定しないことで不揮発性を持つ動作定義を行う．ここでは，不揮発性の午前 9 時から午後 5 時 20 分の間を指定している．

**WHERE** イベントが発生する場所を指定する．WHEN と WHERE の指定により，定義可能なイベントやデバイスがシステムより提示される．ここでは，ユーザが現在いる部屋を指定している．

**EVENT** 発生したイベントを指定する．このイベントは，イベント名とセンシングデータの対応付けを行う必要がある．ここでは，ユーザがある室内に入室したというイベントを指定している．

**IF** WHEN で指定したイベントが発生したときの状態を指定する．本フレームワークでは，センサ単体のほかに複数のセンサにおける平均値や合計値を状態として指定可能である．ここでは，入室した部屋の平均照度が予め定義されている”明るい”という条件に合致した場合を表している．

**THEN** デバイスの制御内容を指定する．DO では，制御対象であるカメラを指定している．そして，WHERE，HOW, INFORMATION とそれぞれ制御対象の場所，動作，制御に必要な情報が指定される．

これにより，“ユーザが入室したとき，部屋が明るい場合カメラをユーザに向ける” という制御が行えると考えられる．この記述において，IF 項と DO 項でセンサネットワークとデバイスの抽象化が可能であることがわかる．

## 5 おわりに

本稿では，センサネットワークを用いてデバイスを制御するためのフレームワークについて述べた．提案フレームワークは，ディレクトリサービスを用いることでイベントソースと制御対象となるデバイスを抽象化する．これにより，システム構成の動的な変化に対応する．

現在は，フレームワークを実現するためのプロトタイプの作成を行っている．今後も作成を継続し，想定している機能を実現することで，フレームワークの設計について考察・検討する予定である．

## 参考文献


[1] B.Brumitt, B.Meyers, J.Krumm, A.Kern, and S.Shafer, “Easy Living : Technologies for Intelligent Environments, ” Proc. of the 2nd International Symposium on Handheld and Ubiquitous Computing (HUC2000), pp.12-29, 2000.

[2] 森川博之, 南正輝, 青山友紀, “STONE:環境適応型ネットワークサービスアーキテクチャ, ” 電子情報通信学会技報, IN2001-12, 2001.

[3] Baldauf, M., Dustdar, S. and Rosenberg, F. “A survey on context-aware systems”, Int. J. Ad Hoc and Ubiquitous Computing, Vol. 2, No. 4, pp.263-277, 2007.

# センサネットワークを用いた デバイス制御における 動的な構成変化に対応するフレームワーク

立命館大学大学院 理工学研究科
基本ソフトウェア研究室
植田　裕規

## 発表内容

- はじめに
- センサネットワークとデバイス制御
  - 既存研究
  - 機能要件
  - 提案フレームワーク
- 提案フレームワークのアーキテクチャ
- フレームワークにおける動作記述
- おわりに

## はじめに（1）

- 半導体技術や無線技術の発達
  - センサノードを複数用いたセンサネットワーク
- 日常にあるデバイス
  - 空調機器，照明，カメラ，etc.
- サービスの連携技術&システム
  - Jini，UPnP，Easy Living，STONE，etc



## はじめに（2）

- センシングデータによるデバイス制御
  - 問題点
    - センサとデバイスの数が膨大
    - システム構成の把握が困難
    - エンドユーザの負担が大きい
      - センサやデバイスの追加，削除や名前などの把握

    システム構成の変化に対して
    動的に対応する必要がある
  - 研究の目的
    - エンドユーザの負担を減らし，より使いやすいシステムを構築
  - 研究のゴール
    - センサネットワークを用いたデバイス制御を行うための動的な構成変化に対応するフレームワークの実現

# 既存研究

- EasyLiving (Microsoft)
  - デバイスとユーザの位置が重要
  - ユーザの位置によるデバイスの切り替え
    - 場所に対するサービスの記述
  - 環境変化やオブジェクト処理などの通知が不可能
    - 非同期イベントの通知が困難
  - 1つの部屋のみの記述を想定
    - 複数の部屋を対象とした記述が困難
- STONE (東京大学)
  - ワールドワイドなネットワーク環境に対応
  - 機能オブジェクトの合成機構を用いたサービスの連携
    - ユーザ環境の変化に合わせた制御対象の変更が困難
  - オブジェクトのデータフォマットや通信プロトコルなどの知識が必要
    - 標準化された形式の専門知識が必要

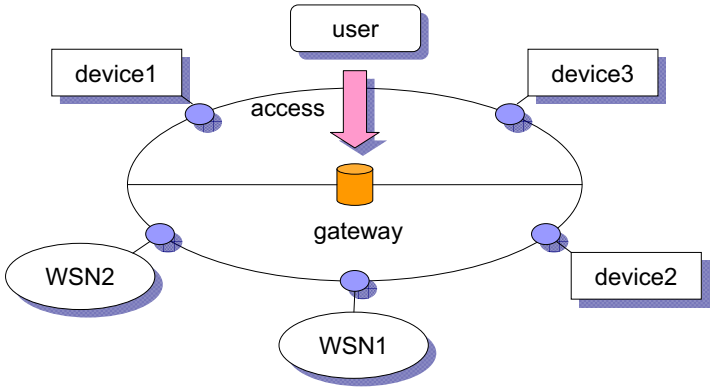


# 提案フレームワークの機能要件

- 空間に対するデバイス制御の記述が可能
  - 多数のデバイスやセンサネットワークに対応するため
  - 場所・時間に基づいた記述
- システム構成の動的な変化に対応可能
  - センサネットワークやデバイスの追加・削除・換装に対応
  - 動作定義の再定義の必要性を除去
- イベント通知機構のサポート
  - 環境変化やシステムの状態をデバイスやユーザにリアルタイムに通知するため
- センセネットワークの使用
  - センサ単体の利用に対して，適用範囲の拡大や空間の環境情報の取得が可能

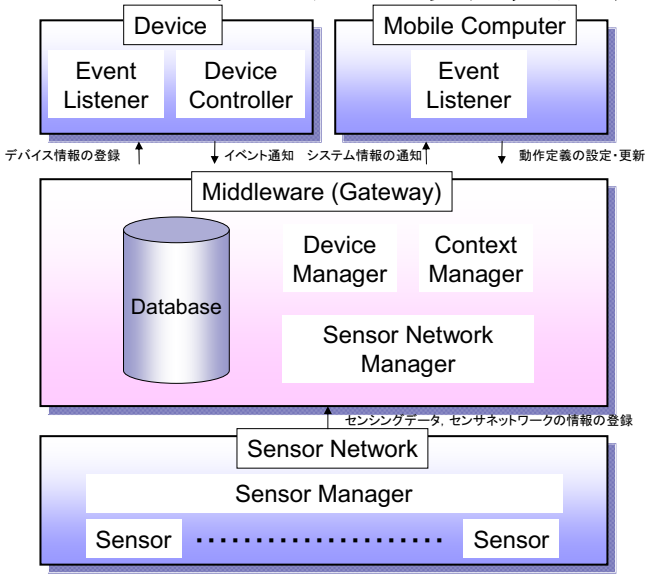
# 提案フレームワークについて

- ディレクトリサービスの提供
  - イベントソースと制御デバイスを抽象化
- 登録するためのAPIとインタフェース
- 動作記述の解析＆イベント通知エンジン






# フレームワークのアーキテクチャ



# フレームワークにおける動作記述

- ECA(Event-Condition-Action)ルールの採用
- 午前9時～午後5時20分ユーザが入室したとき，部屋が明るい場合カメラをユーザに向ける

```
DEFINE Camera_Move XXXX
EVENT Enter_Room
   WHEN 9:00 - 17:20
   WHERE User_Location_Room
IF AVG(light) > Bright
THEN
   DO Camera
   WHERE User_Location_Room
   HOW TRAIN
   INFORMATION User_Location_Coordinates
```



# 既存システムとの記述比較

- EasyLiving
  - GUIを用いたダイアログ選択方式
- STONE
  - データタイプやプロトコルなどの専門知識が必要

```
Function Name : Audio Input
Interface Name : IO-Direction/DataType/Protocols
State Name : Location, Load, etc
GUN : gershwin.mlab.t.u-tokyo.ac.jp
```

- Gaia (Illinois大学 Gaia project)
  - デバイス名を知る必要があり，抽象的に指定できない

```
Context(<ContextType>, <Subject>, <Relater>. <Object>)
Ex. Context(temperature, room 3231, is, 98F)
      Context(printer status, srgalw1 printer queue, is, empty)
      Context(social activity, room, 2401, is, presentation)
```

# おわりに

- センサネットワークによるデバイス制御
  - システム構成の変化に対して動的に対応する必要がある
- 提案フレームワーク
  - ディレクトリサービスの提供
  - 登録するためのAPIとインタフェース
  - 動作記述の解析＆イベント通知エンジン
- ECAルールベースの抽象的な動作記述

- 今後の予定
  - 動作記述ルールの検討
  - フレームワーク実現のためのプロトタイプの作成を継続



# 質疑応答

- Q:動作記述について他に良いものがあるのでは？
  - A:試行錯誤しながら設計しているため，今後変化する可能性がある．しかし，設計した記述が良いかどうかは評価しにくいため，既存の物と比較して何ができるか，何ができないのかを示すしかない．
- Q:提案フレームワークの特色は何か？
  - A:センサ単体ではなく，センサネットワークが利用されている点
  - コメント：センサネットワークを活かした記述が行える等を特色としては？どちらにせよ，センサネットワークを利用する理由が必要となる．

# センサネットワークを利用した屋内移動体測位システムの概要

高木 敬介 †
† 立命館大学大学院理工学研究科

## 1 はじめに

現在, 我々は, 移動体センシングプラットフォームの研究を行っている. 移動体センシングプラットフォームとは, センシング機能を持つ無線通信端末を人や物をはじめとする移動体に携帯させ, 位置情報や取得した環境情報に基づいたサービスを提供するための基盤技術である. 本技術を実現することで, 物体を追跡するトラッキングシステムや人を目的地まで誘導するナビゲーションシステムに応用可能となる. 移動体センシングプラットフォームでは, サービスを提供するために, 端末位置情報を取得する技術が必要である. 位置情報の取得は, 測定距離と固定端末の座標から, 測位手法を利用して移動体の座標を推定するという手順で行う.

従来の測位手法は, 端末移動による端末間距離の変化, 繰り返し距離測定を行う度に不規則に発生する誤差, 障害物による距離測定信号の回り込みを考慮していない. そのため, 物体が高速に移動する場合, 端末間の距離が大きい場合, 端末間に障害物が存在する場合には, 測位精度が悪化するという課題がある.

そこで, 本研究では確率密度関数に基づく高精度測位手法を提案する. 本手法は, 移動体位置の導出過程において, 距離情報に重みづけを行い, 確率的なアプローチで位置測定精度の向上を図る手法である. 測定距離を信頼する指標として，経過時刻, 移動体の速度, 測定距離によって決定され, 正規分布に従う確率密度関数を定義する. 領域に対する移動体の存在確率を確率密度関数の総和で表すことが特徴である. 本手法によって, 先に述べた測距誤差の測位精度に対する影響を削減することが可能である.

また, 提案手法の評価と提案手法を利用した位置測定システムの開発の効率化を実現するために, 位置測定アプリケーションの動作基盤となる測位フレームワークの開発を行う. 測位フレームワークは, 測位に必要な情報を柔軟に定義できるため, 様々な測位システム・測位手法に対応できることが特徴である.

本稿では, 2 章で測距誤差の原因, 3 章で提案手法の概要, 4 章で測位フレームワークの設計，5 章で本稿のまとめを述べる.

## 2 測距誤差の原因

測位を行うために利用する端末間の距離情報には, 様々な要因により誤差が含まれる. 提案手法では, 距離測定段階に発生する誤差による測位精度への影響を低減し, 高精度な測位を実現する. 一般的に, 測定距離と実距離の誤差が発生する原因として, 端末の移動による実距離と測定距離の誤差, 繰り返し距離測定を行う度に不規則に発生する偶然誤差, 障害物による誤差が挙げられる. 本章では, これらの測定誤差について述べる.

**端末の移動による実距離と測定距離の誤差**　移動体センシングプラットフォームでは, 端末が移動することを前提として位置測定を行なう. そのため, ある時刻に測定された距離情報の利用価値は, 時刻の経過や端末の移動により低下する. 端末間の測定距離がそれぞれ異なるタイミングで取得される場合や, 測距周期が大きい場合, 一部の利用価値の低い距離情報を利用せざるを得ない. したがって, 測定距離は, 移動体の速度と経過時間により補正されるべきである. ここで, 移動体の速さを $v$ , 測距周期を $f$ とするとき, 測距情報には, 最大 $vf$ の誤差が含まれ得る.

**繰り返しによる偶然誤差**　一般的に, 繰り返し距離測定を行うことで発生する偶然誤差は, 実距離に対して指数関数的に増加する性質がある. したがって, 端末間の距離が $d$ 離れているとき, 測定距離の確率分布関数は, 平均 $d$ , 分散 $(e^{ad})^2$ の正規分布に従うと仮定できる. ここで $a$ は, 測距方式によって決定される定数とする. そして, 測定距離を最も信頼できる値と見なせば, 測定距離に含まれ得る繰り返しによる偶然誤差を推測できるため, それが測位に与える影響を低減することができる.

**障害物による誤差**　端末間に存在する障害物によって測距信号が回折・反射して, 測定距離に誤差が生じることがある. 障害物による影響には, 測定距離が実距離よりも小さく検出されることはないという性質がある. したがって, 測定距離に障害物による影響が含まれるとき, 実距離は測定距離よりも大きくはならない. また, 障害物によって生じる誤差には, 繰り返し誤差よりも比較的大きいといえる. これは, 一般的に測距デバイスの距離分解能よりも, 回折・反射による測距信号の回り込みの方が大きいからである.

## 3 確率密度関数に基づく位置測定手法

提案手法は, 正規分布に従う確率密度関数を利用し, 2 章で述べた測距誤差が発生する原因を排除する. 確率密度関数は, 移動端末の座標である $(x, y)$ を確率変数とし, 固定端末と移動端末の組み合わせに対して定義する. ここで, 固定端末 $i$ から測距情報を取得したとき, 移動端末が 2 次元上の座標 $(x, y)$ に存在する確率密度関数を $Pi(x, y)$ とする.

このとき, $N$ 個の固定端末から得られる距離情報を利用して, 移動端末が座標 $(x, y)$ に存在する確率密度関数 $P(x, y)$ は, 式 (1) として得られる. したがって, 移動端末が存在する確率がもっとも高いといえる点は, $P(x, y)$ が最大となる座標 $(x, y)$ であるといえる.

$$P(x, y) = \frac{1}{N} \sum_{i=1}^{N} P_i(x, y) \tag{1}$$

この確率密度関数 $P_i(x, y)$ は, 測定距離に含まれる誤差によ

る測位精度への影響を低減するよう物理特性を考慮し, 距離を測定してからの経過時間, 端末の移動速度, 測定距離により定義される.

## 4 測位フレームワークの設計

3 章で述べた提案手法の評価と測位システムの開発の効率化のために, 位置測定アプリケーションの動作基盤となる測位フレームワークの開発を行う. 測位フレームワークは, 測位に必要な情報を柔軟に定義できるよう設計されており, 測位手法に依存しない独立したフレームワークであること, モジュールの変更により様々な測位システム・測位手法に対応できることが特徴である.

図 1 に測位フレームワークの構成を示す. 測位フレームワークは, 測定距離・端末・空間を管理する複数のテーブル, そのテーブルにアクセスするためのインタフェース, 3 章で述べた提案手法や multi-lateration [1] 等の測位手法を組み込むためのモジュールから構成される. 測位アプリケーションは, フレームワーク上で動作し, API を通して測位フレームワークを利用する. 次節から, 測位フレームワークの構成要素である情報管理テーブルと測位モジュールの概要について述べる.

### 4.1 情報管理テーブル

情報管理テーブルには, 距離情報, 端末情報, 空間情報を管理するための 3 つのテーブルが存在する.

**距離情報管理テーブル** 端末間の距離を保持するテーブルである. 本フレームワークを利用したアプリケーションは, 距離情報管理テーブルの距離情報を必要に応じて更新する. 同時に, フレームワークは, 距離情報を取得してからの経過時間を設定する.

**端末情報管理テーブル** 移動端末と固定端末の位置情報を保持するテーブルである. アプリケーションは, システム開始時に固定端末位置を設定する. そして, 測位フレームワークは, アプリケーションからの測位要求をトリガとして, 移動端末位置を更新する.

**空間情報管理テーブル** 移動端末や固定端末が存在する空間情報を保持するためのテーブルである. 本テーブルには, 測位手法が必要とする情報に基づいて, 空間制約情報・電波強度・空間の接続情報などを保持することを想定している. そのため, 保持する情報は, アプリケーションや測位モジュールにより定義され, アプリケーションは, 定義に基づいて空間情報を設定する.

### 4.2 測位モジュール

測位モジュールには, 移動端末の位置を導出するための操作である測位手法と, 測位手法が参照する空間情報のデータ型を定義する. 測位フレームワークは, アプリケーションの要求に応じて, モジュール内に組み込まれた測位手法を呼び出す. 測位モジュールでは, 測位フレームワークの要求をトリガとして, 次のような処理を行うことを想定している.

1. 測位条件の成立の判定を行う. モジュールからは, 4.1 節で述べた各情報管理テーブルにアクセスできる. 距離情報テーブルから, 利用可能なビーコンノード数が不足している場合, 測位不能であるという制限を設ける.

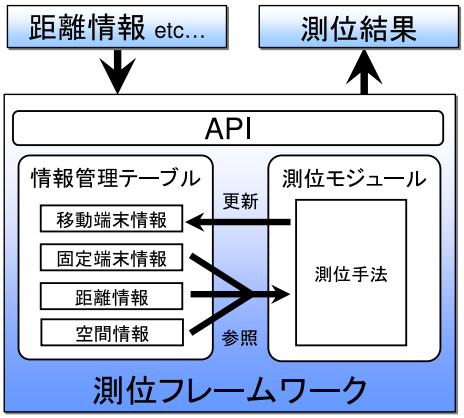



図 1 測位フレームワーク構成図

2. 利用可能な距離情報をテーブルから取得する. ここで, 利用可能な距離情報とは, フレームワークが測定距離情報に, タイムアウトを設定する必要がある場合に, 距離情報がフィルタリングされることなどを想定している.
3. 測位のための演算に必要なパラメータを生成する. 例えば, multi-lateration では, 位置情報と距離情報から測位演算に必要なパラメータである行列を生成する.
4. 測位演算を行い, 移動体位置や速度を導出する.
5. 移動体位置を情報管理テーブルに格納・更新する.

また, 測位手法によっては, 測位対象空間に制約を設ける場合や, 電波強度マップを参照する場合がある. その場合, 本モジュール内に空間情報のデータ型の定義が, 先に述べた, 空間情報管理テーブルに反映される. そのため, 測位モジュールは, アプリケーションが登録した空間情報にアクセスすることができる.

## 5 おわりに

本研究では, 従来の測位手法の課題であった, 距離情報に含まれる誤差による測位精度への影響を低減することを目的とした手法の提案を行なった. そして, 本手法の評価・測位システムの開発効率の向上のために, 測位システムの動作基盤となるフレームワークを開発しており, 本稿では, 測位フレームワークの設計について述べた. 移動体センシングプラットフォームでは, 移動端末が, 自律的に位置情報に基づくサービスを提供できることが望ましい. そのため, 移動端末上でフレームワークの動作が可能となるように, より詳細な設計を行い, 移植性の高い実装を行う.

今後は, フレームワークの実装, 提案手法の評価アプリケーションの開発, 提案手法の評価・改良, トラッキングシステムの開発を行う予定である.

## 参考文献

[1] Andreas Savvides, Chih-Cheih Han, Mani B. Strivastava, Dynamic Fine-Grained Localization in Ad-hoc Networks of Sensors, International Conference on Mobile Computing and Networking, pp.166—179, 2001.

# 確率密度関数に基づく高精度位置測定システムの開発

JSASS 2008

立命館大学大学院 理工学研究科 博士課程前期課程
情報理工学専攻 計算機科学コース

高木敬介

1

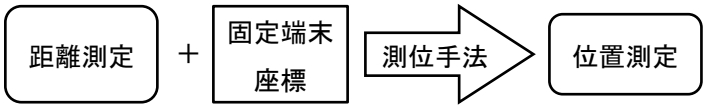
## 目次

- はじめに
  - 研究背景
  - 移動体センシングプラットフォームとは
- 誤差の原因
  - 端末の移動による実距離と測定距離の誤差
  - 繰り返しによる偶然誤差
  - マルチパスによる誤差
- 確率密度関数による高精度位置測定手法
  - 確率密度関数の決定
- 測位フレームワークの設計
  - 測位モジュール
- おわりに

2

## はじめに

- 移動体センシングプラットフォーム
- 位置測定

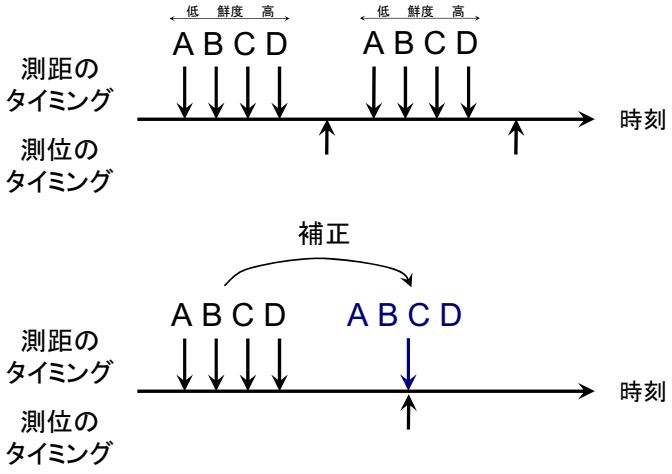



- 距離測定における誤差の原因
  - □端末の移動
    端末の移動による測定距離と実距離の差
  - □繰返しによる偶然誤差
    繰り返し測定するごとに発生する異なる誤差
  - □障害物による影響
    端末間に存在する障害物による測距信号の回り込み

3

## 課題1) 端末の移動による誤差

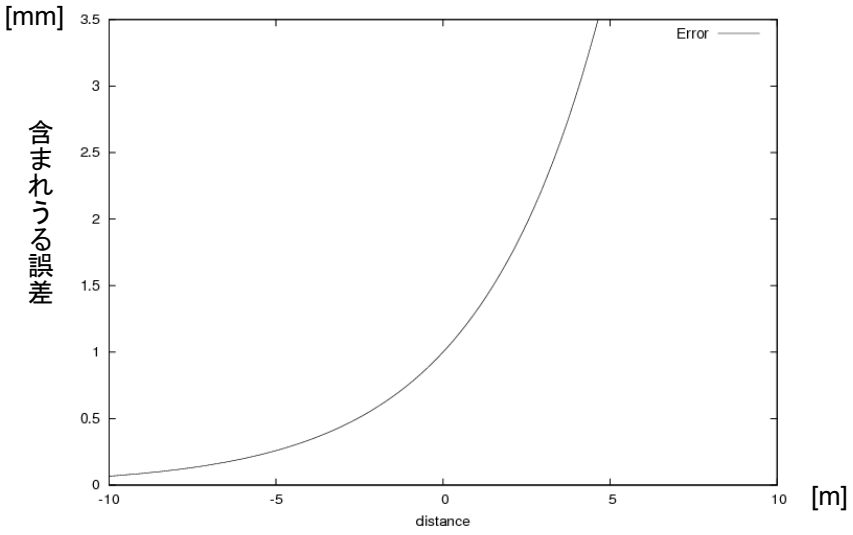
- 鮮度の低下




4

## 課題2) 繰り返しによる偶然誤差

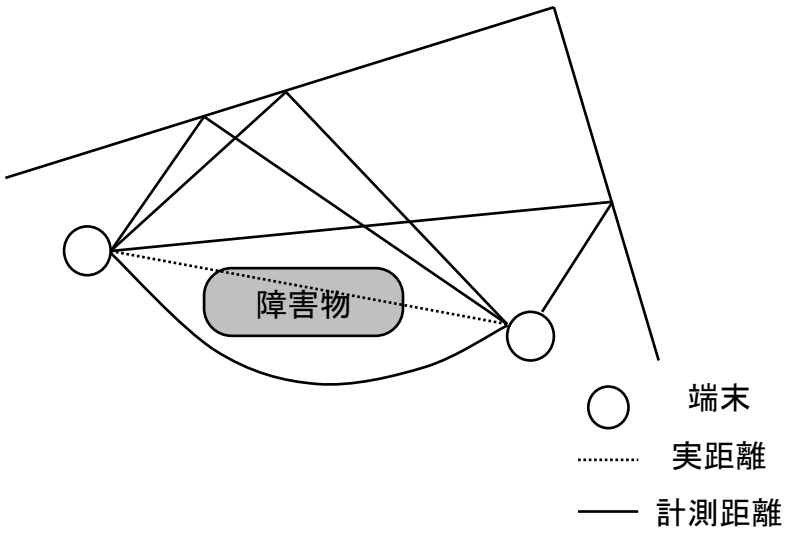
- 実距離に対して指数関数的に誤差が増加




5

## 課題3) 障害物による影響

- 実距離<計測距離




6

## 提案手法

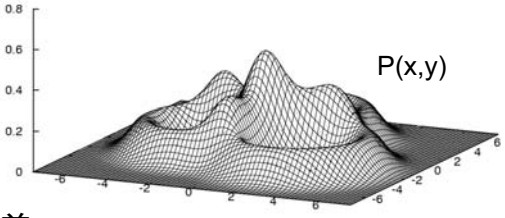
- 確率密度関数に基づく測位手法

$$P(x,y)=\frac{1}{N}\sum_{i=1}^{N}Pi(x,y)$$

一意に位置を導出することが困難である

確率的なアプローチ

- 確率密度関数
  - 移動体速度 ($v_x$,$v_y$)
  - 経過時間 t
  - 測定距離 d



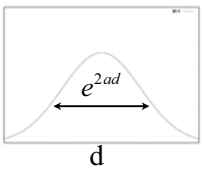

- 測定距離に含まれうる誤差
  - 物体の移動
  - 繰返しによる偶然誤差
  - 障害物による影響

7

## 確率密度関数 $P_i$( , ）の決定

- 偶然誤差と障害物
  - 測定距離 d
  - 正規分布
    - 平均　$\mu = \mathrm{d}$
    - 分散　$\sigma^2 = e^{2ad}$
  - 分散増加率 a

- 物体の移動
  - 移動体速度 ($v_x$,$v_y$)
  - 経過時間 t
  - 減衰定数 c



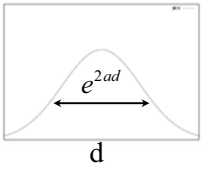

- 確率密度関数 $P_i$(x,y)

$$P(x)=\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right)$$

8

## 確率密度関数 $P_i$( , ) の決定

- 偶然誤差と障害物
  - 測定距離 d
  - 正規分布
    - 平均　$\mu = \mathrm{d}$
    - 分散　$\sigma^2 = e^{2ad}$
  - 分散増加率 a
- 物体の移動
  - 移動体速度 ($v_x$,$v_y$)
  - 経過時間 t
  - 減衰定数 c



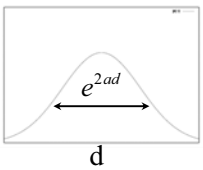

- 確率密度関数 $P_i$(x,y)

$$P_i(x,y) = \frac{1}{\sqrt{2\pi(e^{2ad})}} \exp\left( \frac{-(\sqrt{x^2+y^2} - d)^2}{2e^{2ad}} \right)$$

9

## 確率密度関数 $P_i$( , ) の決定

- 偶然誤差と障害物
  - 測定距離 d
  - 正規分布
    - 平均　$\mu = \mathrm{d}$
    - 分散　$\sigma^2 = e^{2ad}$
  - 分散増加率 a
- 物体の移動
  - 移動体速度 ($v_x$,$v_y$)
  - 経過時間 t
  - 減衰定数 c



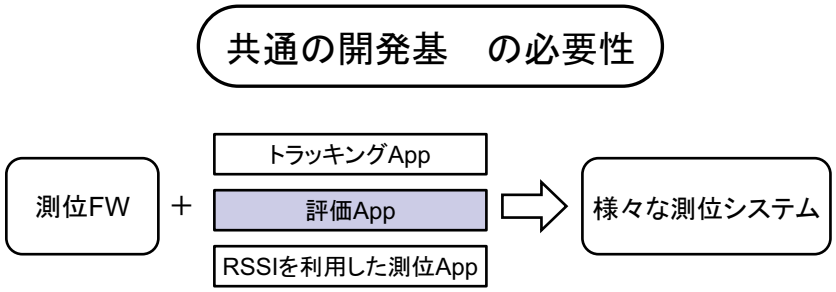

- 確率密度関数 $P_i$(x,y)

$$P_i(x,y) = \frac{1}{e^{ct}\sqrt{2\pi(e^{2ad})}} \exp\left( \frac{-(\sqrt{(x+v_x t)^2 + (y+v_y t)^2} - d)^2}{2e^{2ad}} \right)$$

10

## 開発基　の必要性

- 提案手法の評価システムが必要
- 今後の開発プロセス
  - 提案手法評価システム
  - トラッキングシステム
  - その他の位置測定システム



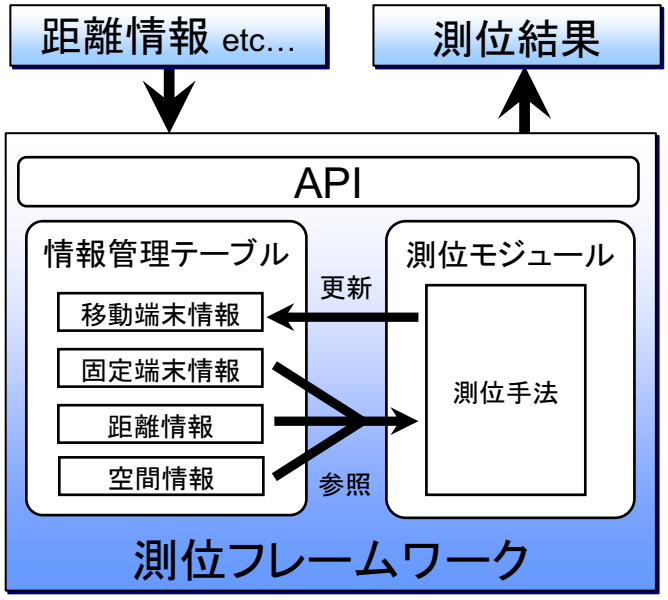

11

## 測位フレームワークの開発

- 位置測定アプリケーションの動作基
- 目的
  - 開発効率の向上
- 特徴
  - 多くの測位手法・測位システムに対応できる
    - モジュール
    - 移植性の高い実装
  - Ex) Multi-lateration・電　強度に基づく測位手法(RSSI)

12

## 測位フレームワークの構成




13

## 測位モジュール(1/2)

- 測位手法の実装
  Ex. multi-lateration / 提案手法
  - フレームワークから呼び出される
  - 距離情報・固定端末情報・空間情報を参照
  - 移動端末情報を更新
- 想定される処理内容
  - 測位手法上の制限 Ex. 利用可能な距離情報のフィルタリング
  - 利用可能な距離情報の取得
  - 測位　算
  - 移動体情報の更新

14

## 測位モジュール(2/2)

- 測位に用いる空間情報
  Ex. 空間制約 / 電　強度マップ / 隣接空間情報
  - 空間情報管理テーブルのデータ型
  - アプリケーションから参照可能

15

## おわりに

- 確率密度関数に基づく高精度位置測定手法
  - 測定距離に含まれる誤差を確率的なアプローチで低減
- 測位フレームワークの開発
  - 開発効率の向上
  - 測位手法・測位システムに　用的
- 移動端末上で動作が可能となるような実装

- 今後の予定
  - フレームワークの実装
  - 提案手法の評価・改良
  - トラッキングシステムの開発

16

# センサノード上の時系列データに対するルールベースの問合せ処理

冨森 英生 †
† 立命館大学大学院理工学研究科

## 1 はじめに

近年, 機器の小型化や低価格化が進み, 無線とセンサ機能を備えたセンサノードの開発が行われている. これらの技術の進展により, 複数のセンサノードを利用し構成するセンサネットワークが注目されている. センサノードを用いて気温や湿度といった周囲の環境情報を取得する場合, 観測対象の状態やユーザの要求に合わせてノードの動作を指定し, 取得する観測データを変更する必要がある. センサネットワークでは, 問合せ処理を行うことで, ノードの動作を変更しユーザの要求を満たす観測データの取得を行う.

バッテリーにより動作するセンサノードは, 長期間にわたり動作させるため, 電力消費を抑制する必要がある. センサノードの動作のうち, 特に無線によるデータの送受信は多くの電力を消費する. ノードを長期間動作させるため, 不要なデータを削減し, 必要なデータのみ送受信することによる電力消費の抑制が行われている. 問合せ処理では, 問合せに記述された条件により, ユーザが必要とするデータの判別が行われる. これを観測データの取得元であるノードにおいて行うことで, 不要なデータの送受信を効率よく削減することが可能である.

センサノードは分散して配置された各地点について継続して観測を行い, 取得した観測データは時系列を表わすものとなる. また, センサネットワークを用いて観測を行う対象は, 時間の経過により状態が変化する. データの時系列変化から観測対象の状態を判別し, それに応じて使用するセンサの種類や観測周期を変更することで, より詳細な情報をセンサネットワークから取得可能になる.

本研究ではネットワークに分散して存在する個々のノードが観測データの時系列変化から観測対象の状態を判別し, 動作を動的に変更することを目的としたルールベースの問合せ手法を提案する. 具体的には, ノードは取得した観測データを自身の記憶領域に蓄積する. ノードが得られたデータの変化を検出し, 事前に定められた条件に従い動作を切り替えることで, 観測方法や蓄積した時系列データに対する処理を切り替える.

以下, 本稿では, 2 章で問合せ処理における課題について述べ, 3 章で提案する問合せ処理, 4 章で実装と今後の検討事項について述べ, 5 章でまとめる.

## 2 観測対象の状態に応じた問合せ処理

### 2.1 問合せ処理の課題

TinyDB [1] など既存の問合せ処理では, ノードが取得した観測データをホストに集める集中管理型をとる. データの取得元となるノードはすべての観測データをホストに送信し, ホストが観測データに対する解析を行う. 観測方法を変更する必要がある場合, ホストはノードに対して新たな問合せを発行し, ノードの動作を切り替える. ホストがノードの動作を指定する集中管理型の場合, 観測を行うノードと, データ解析, 観測方法を決定するホスト間で通信が頻繁に発生し, 電力面で非効率である.

観測データに対する処理をホストで行わず, データを取得したノード自身が行うことで, ノードの自律的な動作が可能になる. ノードは観測データに重要な変化が検出された場合のみホストに送信することで, すべての観測データをホストに送信する場合に比べ通信を削減でき効率が良い. また, 取得した観測データに基づき, ノードが自身の判断で観測方法を切り替えることで, 状況に応じた適切な観測が可能になる.

### 2.2 状態に応じた観測

観測対象の状況は変化することから, ユーザはセンサネットワークに対し, 状況により異なった観測を行いたいという要求がある. ノードの電池寿命を延ばすため, 電力消費の大きい観測や通信を最小限に抑えたい場合がある一方で, 状況によって電力面で非効率でも現在の観測データをホストで即時に取得したい場合もある.

例えば防災を目的とした観測を行う場合, 図 1 に示すように, 安定した天候では観測周期は長いものでよく, 省電力のため観測結果はホストに送信しない, もしくは 1 日程度を単位として集約した結果のみ送信すればよい. 一方で, 雨量が多く災害の危険性が高い場合, 観測周期を短くし, 観測データは即座にホストに送信し災害予測のための高度なデータ解析に備えることが望ましい.

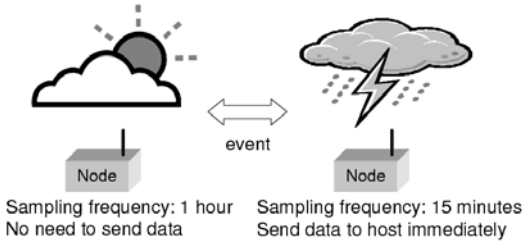


図 1 状態に応じた観測

このような要求に対して，ECA(Event-Condition-Action) に類似したルールを用いて動作を切り替える手法が提案されている [2, 3]. 観測対象の状態に応じてノードが取るべき動作をユーザが事前に定め, ノードに与えておくことで, ノードはホストと通信を行うことなく, 自身の判断で動作を切り替えることが可能になる. 時系列の変化は, ノードに観測データを蓄積し, そのデータを処理するルールを定義することで判別できる.

この例では, ユーザはルールに晴天・雨天の各状態を定義し, 観測周期などそれぞれの状態の観測方式を事前に定める. ノードはルールに基づき, 観測データから雨の降り始め, 降り終わりの状態変化を判別し, 省電力性重視またはデータの即時性重視といった状況に合わせた動作へ自身の判断で切替えを行う.

## 3 ルールベースの問合せ処理機構

本研究では，ルールベースの問合せを用いた動的なノードの動作切替えを実現する. 図 2 に示すように, 観測した時系列のデータ履歴を各ノードが蓄積・参照し, 状態に応じた重要な情報のみユーザに通知することで, 観測方式の切替えと通信の削減による電力消費の抑制に寄与できる. ノードの動作はユーザにより事前にルールとして定められ, すべてのノードに対して発行される. 各ノードはルールに基づき, 観測や通信などの動作を自身の判断で切り替える. ノードの動作を定めるルールは次の問合せにより構成される.

**状態ごとの観測問合せ**　ノードが取る動作状態を定義し, 各状態において実行する問合せとして, 実行周期, 使用するセンサデバイスの種別やデータ集約の有無, データ格納先を指定する. 格納先は自身のメモリへの蓄積, または無線によるホストへの送信を選択する. 通信周期を観測周期よりも長く設定した問合せを定義することで, 集約値のみホストに送信することが可能になる.

**イベント生成問合せ**　ノードは自身が取得・蓄積した観測データを参照し, 条件が満たされる場合はイベントを

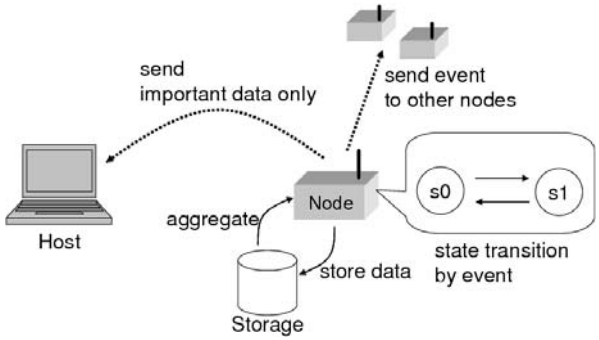


図 2 システム構成

生成する. ノードがおかれた状況の変化を表わすイベントは, ノード自身のみならず近隣のノードにも伝搬させ, 複数ノードによる協調観測に役立てる.

**状態遷移規則**　自身が生成または他のノードから受信したイベントに応じて, 規則に基づき適切なノードの動作状態へ遷移する. 規則は遷移元・遷移先・遷移のトリガとなるイベントの 3 種類の記述により定義される.

例えば, 図 1 の雨天への状態遷移を過去 3 時間の積算降水量が閾値を超えた場合に行うとすると, ルールは次のように記述できる. イベント生成問合せは, 観測問合せで取得された観測データがローカルストレージ上に格納されることをトリガとして実行される. 急激な状況の変化が起きうる場合は観測周期を短く，そうでない場合は長くとることによって対応する.

```
INITIAL STATE fine {
  SELECT nodeid, rainfall
  FROM sensors INTO local_storage
  SAMPLE PERIOD 1 hour
}

EVENT start_raining {
  ON UPDATE local_storage
  IF sum(rainfall) > threshold
    FROM local_storage
    WHERE time > now - 3 hour
  PROPAGATE local, remote all
}

TRANSITION fine -> rain {
  ON EVENT start_raining
}
```

雨天時は, 現在の観測データを表わす sensors テーブルを参照し, 雨量データをホストに送信する.

```
STATE rain {
  SELECT nodeid, rainfall
```

```
  FROM sensors
  INTO remote_host, local_storage
  SAMPLE PERIOD 15 minute
}
```

## 4 実装と今後の検討事項

### 4.1 実装

センサノードデバイス Sun SPOT 上に本問合せ処理機構を実装している. 現在までに, ルールを構成する観測問合せ・イベント生成問合せ・状態遷移規則を処理する基礎機能を実装しており, 3 章で示したルールを実行可能である.

現状の実装では問合せ機能の拡充を優先している. センサネットワークで重視される通信デバイス, 使用しないセンサの電力遮断などの省電力化機能は今後実装を進める. また, 受信待機電力を削減するための通信タイミング制御方式を検討する必要があると考えている. ノードの省電力化を行った上で, 既存の問合せ手法との比較や問合せ言語の記述能力を評価する予定である.

### 4.2 複数ノードの協調観測

イベントの発生は観測対象の状態変化を表わす. この情報はイベントを生成したノードのみならず, 他のノードにおいても有用である. 先述の防災観測の例では, 少数の雨量計, 多数の土壌水分観測計を用いるなど異なる機能を持つノードを混在させてネットワークを構成することが予想される. この場合, 雨量観測ノードが天候に関するイベントを土壌水分観測ノードに伝搬することで, 異種ノード間で協調した観測を実現できる.

センサネットワークを構成する複数のノードに対してイベントを効率よく伝搬するために, ノードの種類や伝搬範囲を考慮したルーティング手法を今後検討する. これらルーティングのための情報はユーザがルール内に記述し, イベントごとに適切な伝搬対象を設定可能にする.

## 5 おわりに

本稿では, センサノード上に観測データを蓄積し, ノードの動作を状況に応じて動的に切り替える問合せ手法について述べた. 今後は問合せ処理機構の実装を進め, イベント伝搬手法の検討を行う予定である.

## 参考文献


[1] Madden, S., Franklin, M., Hellerstein, J. and Hong, W.: TinyDB: an acquisitional query processing system for sensor networks, *ACM Transactions on Database Systems (TODS)*, Vol. 30, No. 1, pp. 122–173 (2005).

[2] Romer, K. and Mattern, F.: Event-based systems for detecting real-world states with sensor networks: a critical analysis, *Intelligent Sensors, Sensor Networks and Information Processing Conference, 2004. Proceedings of the 2004*, pp. 389–395 (2004).

[3] Zoumboulakis, M., Roussos, G. and Poulovassilis, A.: Active rules for sensor databases, *ACM International Conference Proceeding Series; Vol. 72*, pp. 98–103 (2004).

# センサノード上の時系列データに対するルールベースの問合せ処理

立命館大学大学院 理工学研究科
大久保・横田研究室
冨森 英生

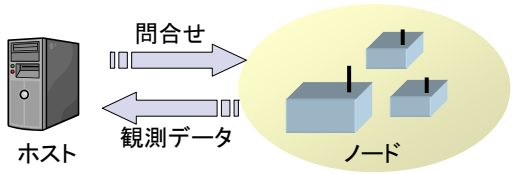


## 発表内容

- はじめに
- ノードの省電力化・役割分担
  - ノード上でのデータ量の削減
  - 状況に応じた動作への切替え
- ルールベースの問合せ処理
  - 問合せ例
- おわりに

# はじめに

- センサネットワークによる環境観測
  - センサノードは電池、温度センサなどを搭載した小型デバイス
  - 無線通信で観測データをホストに収集
- 問合せ処理による観測
  - ノードは問合せ内容に従い周期的に観測




3

# 従来の問合せ手法の課題

- 頻繁に発生する無線通信
  - ノードは観測データを選別せずにホストに送信
    - 重要性の低いデータも送信
    - 通信は電力消費が大きく観測寿命に影響
- ノードごとの動作を指定できない
  - 全ノードが同時に単一のクエリにしか対応できない
    - ノードの種類別に役割分担ができない
- ノードの動作が受動的であることに起因
  - ノードの動作をホストが集中管理

✓ 不要な動作が電力を消費

4

## ノードの自律的な動作による解決

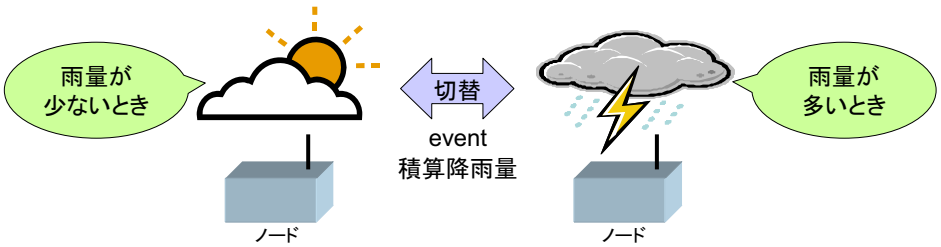
- ノードごとに適切な観測動作・通信データを選択させる
  - [観測方式] 使用するセンサデバイス・観測回数を最小限に
  - [データ処理] 重要性の高いデータのみホストに送信
- ただし重要なデータ・適切な動作は状況により異なる

- ノードの動作を状況に応じて
  自律的かつ動的に切り替えることで解決可能
  - 観測データに基づきノードが状況を判別
  - 判別結果に応じて観測・データ処理方法を切替え
  - ノードごとに役割分担して観測

5

## 状況に応じた観測

- 例: 防災(地滑りなど)を目的とする観測




観測周期: 1 時間
観測毎のデータ送信なし
1日単位で集約値を送信

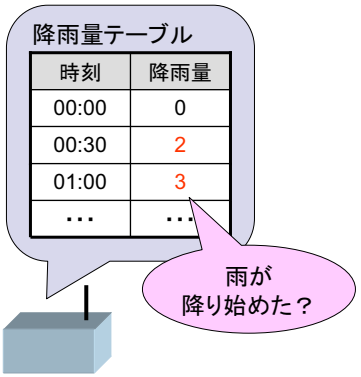
観測周期: 15分
観測毎にデータをホストに送信

状況に応じた動作で観測処理とデータ量を削減

6

# 時系列データによる状況判断

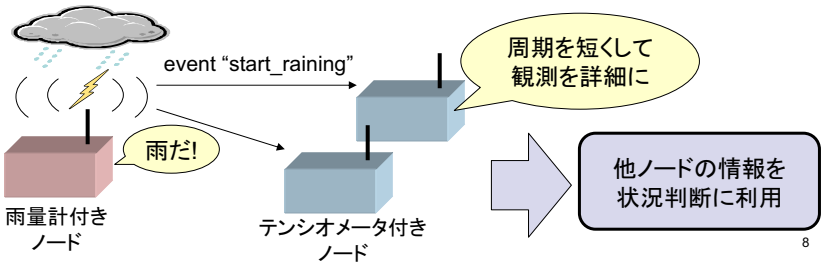
- 観測データをノード上に蓄積
- 過去の時系列データを参照して現在の状況を判断
  - 過去のデータの平均・合計，1日前の同時間帯との比較など
  - ホストを介さずノードが判断する
- 判断結果としてイベントを生成
  - 動作切り替えのトリガになる




7

# 他ノードの情報の利用

- 状況判断に他ノードの情報も利用したい
- 異なるセンサを搭載するノードが混在する環境を想定
  - 地滑り観測では，雨量計，テンシオメータ(土壌水分計)
  - 種類ごとに台数・観測周期が異なる
- 状況の変化を他ノードにイベントとして伝搬

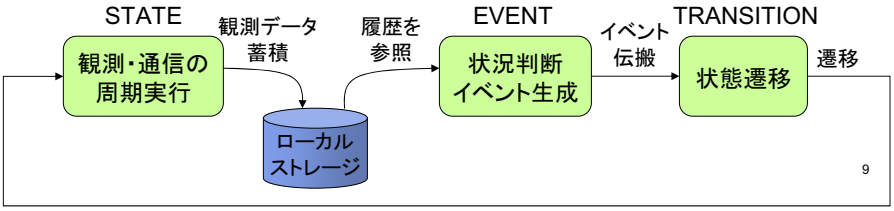



8

# 提案手法:
# ルールを用いた動作の切替え

- ノードの動作を状況に応じて動的に切り替える

- 次の3種類の処理内容をユーザがルールとして記述
  - [STATE] 観測, データ選択・集約手法を状態ごとに記述
  - [EVENT] 周囲の状況の変化を表わすイベントの検出方法
  - [TRANSITION] イベントによる状態の遷移規則
- イベントに基づく状態遷移の実現




9

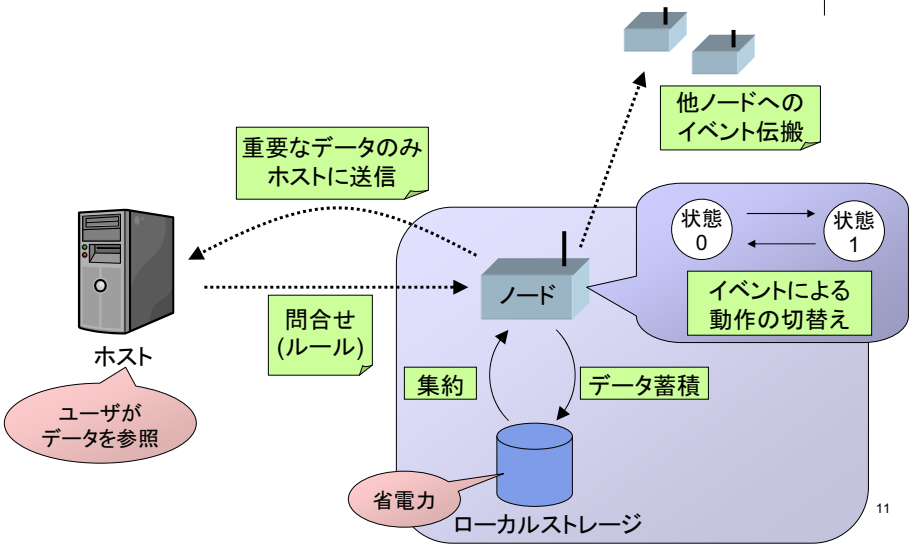
# 提案手法の特徴

- 従来手法では同時に単一のクエリにしか対応できない
  - 受動的なノードの問合せ処理
- 提案手法はノードごとに動作が異なる
  - 各ノードが状況に応じた問合せを実行
  - 自律的な動作

- 特徴
  - 観測データをローカルストレージに蓄積して
    ノード自身が状況判断
  - 異種ノード間で役割分担が可能
  - ノードの高機能、高効率化

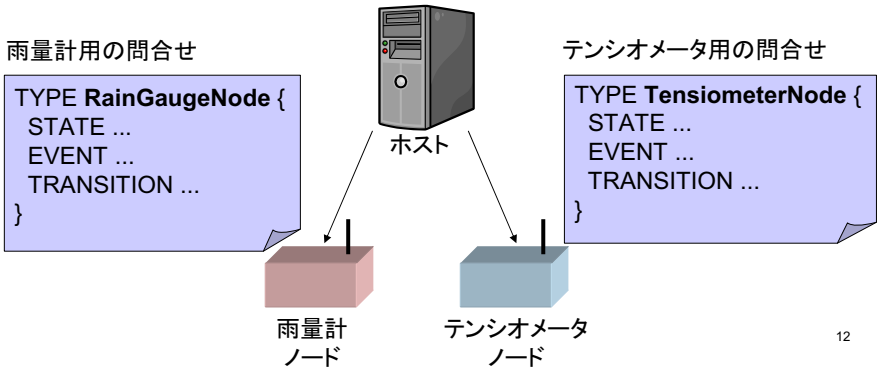
10

# システム構成



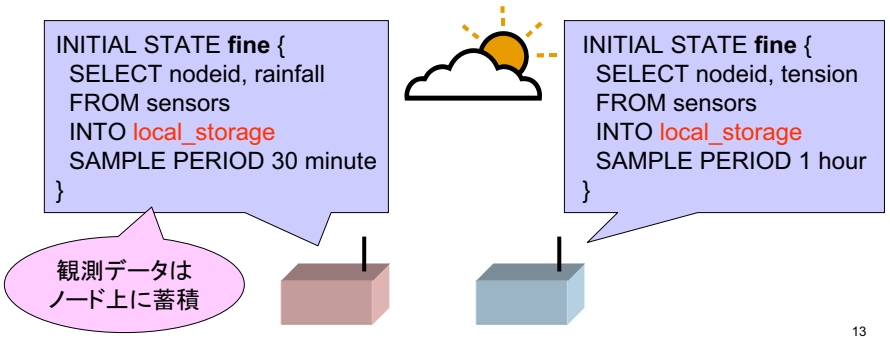

# ルールを用いた問合せの発行

- ホストからルールをノードに送信
  - ノードの種類別に, STATE, EVENT, TRANSITIONを定義



# 平常時の観測

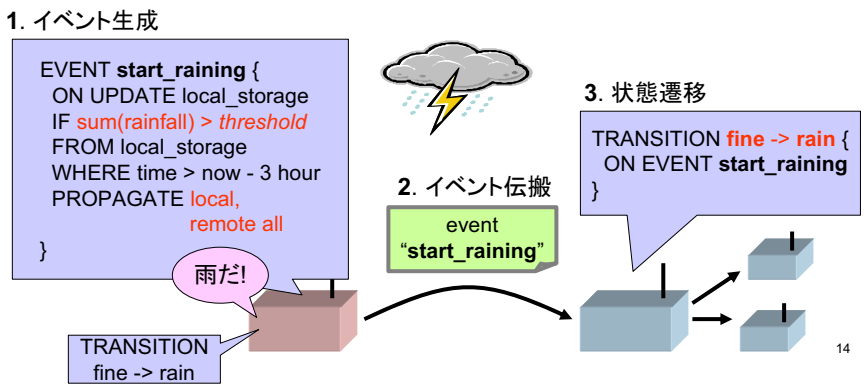
- 平穏な天候のとき，観測データはノードに蓄積
  - データの重要性は低い
  - 通信量を削減




13

# イベントの生成と状態遷移

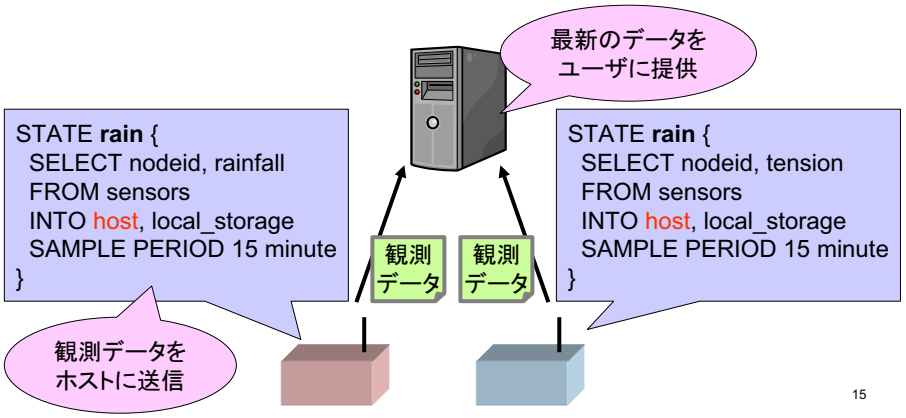
- 過去の雨量データの積算値を基に状況を判別
  - 閾値を超えるとイベント”start_raining”を生成し，他ノードに伝搬
  - 降雨時の観測状態へ遷移




14

# 状態に応じた観測動作への切替え

- 降雨時は観測毎にデータをホストに送信
  - ユーザはホストのデータを参照して防災に役立てる




15

# 実装

- Sun SPOT 上に実装を行っている
  - Sun Microsystems社製センサノード
  - ノード上でJava Virtual Machineが動作
  - Java言語でプログラムを記述

- 作成するノードの機能
  1. データ蓄積・問合せ処理
  2. ルールによる動作切り替え
  3. 省電力化のためのデバイス制御

16

## 実装と今後の検討事項

- 問合せ・ルール処理の基本機能を実装した
  - 観測問合せ・イベント生成問合せ・状態遷移規則
  - 通信デバイス・使用しないセンサの電力遮断などの省電力化機能は今後実装
- イベント伝搬手法
  - ノードの種類・伝搬範囲を考慮したルーティング手法を検討
  - ユーザがイベントごとに適切な伝搬対象を設定可能にする

17

## おわりに

- センサノードの省電力化・役割分担
  - 動作が受動的であることに起因
- ルールによる状況に応じた観測・データ処理
  - ノードに観測データを蓄積
  - 状態, イベント, 状態遷移による動作の切替え
  - 重要なデータのみホストに送信
- 今後の予定
  - 省電力化機能の実装
  - イベント伝搬範囲の制御・ルーティング手法の検討

18

# 編集後記

つい先日名古屋で皆さんと研究について議論し，懇親会でさまざまな情報交換をさせて頂いたはずでしたが，そう，もうあれから 1 年がたちました．時間の流れの速さが身にしみます．年に 1 度というのは，その数字だけを見れば多くはありませんが，ご参加頂いている皆さんの，普段の研究教育活動や学業などとあわせたとき，十分なインパクトがあるのではないかと思います．少なくとも私にとっては，良い意味でのプレッシャーとなっており，どんなクオリティの発表ができるか，いくつの発表ができるか，その質も量も気にしながら進めております．そういう意味では，JSASS が，毎年のマイルストーンになっています．学生諸君にとっても，毎年参加していると，この時期には JSASS があるものだという暗黙の了解ができているようで，超えるべきひとつのハードルとして考えてくれているようです．人間は誰でも，何らかの目的・目標がないと何もできないと思いますが，JSASS が目標のひとつになっているようで，それだけをとっても JSASS の意義はあると，そう感じています．もちろん，そこで繰り広げられる研究についての議論，研究者同士の交流については，言わずもがな…です．

さて，今年は，東京大学大学院情報理工学系研究科の秋葉原拠点で JSASS を開催させて頂きました．東京大学の笹田先生には，幹事団のお一人としてご参加いただき，特にローカルアレンジをしていただきました．会場の準備に加え，懇親会の準備もしていただき，多大なご尽力をいただきました．この場をお借りしまして，お礼申し上げます．

また，幹事団の一人として，龍谷大学の芝先生にもご尽力いただきました．JSASS の開催の案内，申し込みの取りまとめ，講演プログラムの作成など，すべてご担当いただきました．この場をお借りしまして，芝先生にも，お礼申し上げます．

さらに，開催当日には，東京農工大学の並木先生からオープニングとクロージングでお言葉を頂戴しました．ディスカッションでもスパイスの効いた助言をして頂くなど，全体的なとりまとめをして頂きました．また，本ポストプロシーディングでは，巻頭言も頂戴いたしました．この場をお借りしまして，お礼申し上げます．

最後になりましたが，ご発表頂いた皆様，議論にご参加頂いた皆様，ありがとうございました．特に学生諸君におかれましては，今回得られた助言や経験を，研究と人生の両方に活かして頂ければと思います．出力のないプログラムは無意味ですが，人についても同様なことが言えると思います．外に向いてアウトプットしていきましょう．JSASS がその一端を担えたとすれば，幸いです．

立命館大学　毛利 公一

今年も無事に JSASS を実施することができた．6 個のセッションで 18 件の発表があり，いずれも興味深いものであった．毎年 JSASS を楽しむことができているが，これは毎回新鮮さを感じることができているからだと思う．改めて第 1 回から JSASS のことを考えてみると，スコープがシステムソフトウェアであることは変わらないが，発表テーマは年々変わってきていることが分かる．システムソフトウェアのトレンドは大きく変化しつづけているため，これは当然のことかもしれないが，トレンドをきちんと分析しその中から課題や研究テーマを見つけている発表を毎年見ることができるのは嬉しいことである．また，JSASS では比較的大学院生の発表が多いが，研究室での研究活動の中心となる大学院生が多くの新しいテーマに取り組んでいることは素晴らしいことだと思う．

JSASS は，扱うテーマが for Advanced System Software ということであり，Advanced System Software に繋がりそうなものであれば広く受け入れる．完成する前のシステムについても議論することができ，このような場で得られるものは貴重であると思う．また，今後もシステムソフトウェアの研究が停滞することなく活発でありつづけ，JSASS のような場が必要とされることがつづいていくことを願う．

龍谷大学　芝 公仁

JSASS は過去に 3 度ほど出席させていただきましたが，オペレーティングシステムやシステムソフトウェアを中心とする話題に関してざっくばらんな議論をすることができる場所ということで大変楽しかった思い出があります．2008 年 4 月の IPSJ SIGOS 研究会で毛利先生とお会いしたとき，まだ今年の JSASS の会場が決まっていないということで，現在の笹田の職場である東京大学情報理工学系研究科 秋葉原拠点をご紹介したところ，使っていただく運びとなりました．あわせてローカルアレンジメントも仰せつかりました．

秋葉原拠点は同研究科創造情報学専攻の拠点であり，秋葉原駅前に建っている秋葉原ダイビルの 13F に位置します．創造情報学専攻は，ハードウェアアーキテクチャのような話から人工知能のような研究まで，横断的な情報分野に関する研究を行う専攻です．また，秋葉原という立地を活かした新しい情報分野の開拓を目指しています．

JSASS を東大秋葉原拠点で開催していただくことができたのは，本専攻の趣旨としても歓迎できることですが，個人的にも自分のホームでこのような楽しいイベントを開催していただき大変嬉しく思っております．

今年も多くの学生の方々の色々な分野の発表を聞くことができ，大いに刺激を受けました．質疑応答で色々とうるさいことを言ったような気もしますが，それだけ興味深いお話だったということで許して下さい．また，懇親会も大いに盛り上がり，とても楽しい時間を過ごすことができました．どうもありがとうございました．

頼りないローカルアレンジではありましたが，概ね問題なく会期を終了できたかと思います．協力いただきました皆様にはこの場をお借りして御礼いたします．とくに，JSASS の総まとめをされております立命館大学の毛利先生には深く感謝いたします．

また，来年以降も参加させて下さい．

東京大学　笹田耕一

先進的基盤ソフトウェア
Joint Symposium for Advanced System Software 2008 (JSASS2008)

2 巻 1 号（通号 2 号）オンライン版　2008 年 11 月 11 日発行


編　　集　毛利 公一, 芝 公仁, 笹田 耕一, 並木 美太郎
委 員 長　大久保 英嗣
発 行 者　JSASS 実行委員会
〒525-8577 滋賀県草津市野路東１－１－１
立命館大学情報理工学部 毛利研究室内
電話 077-561-5061
発 行 所　〒184-8588 東京都小金井市中町２－２４－１６
東京農工大学工学部 並木研究室
電話 042-388-7139