令和元年度

修士学位論文

複数計算機上に跨るソフトウェア実行環境の移送手法

Migration method of software environment spanning multiple computers

1225117　黒木 勇作

指導教員　横山 和俊

2020 年 2 月 28 日

高知工科大学大学院 工学研究科 基盤工学専攻

情報学コース

要 旨

複数計算機上に跨るソフトウェア実行環境の移送手法

黒木 勇作

クラウド環境は，地理的に離れた複数の計算機に資源を分散して配置した，広域分散システムとして実現されることが多い．広域分散システムでは効率的な利用のため，アプリケーションソフトウェア (以降 AP と略す) の実行環境を他の計算機に移送させることがある．移送させる際の代表的な手法として，仮想マシンのマイグレーションがある．この手法は実行環境となる仮想マシンイメージを移送させるものである．しかし，実行環境に必要のない資源も移送させるため，時間や移送サイズの面から非効率的な手法となる．そこで，本研究では資源利用情報を基に，ソフトウェア実行環境を特定する手法を提案する．提案手法では，AP のファイルアクセスを監視し，AP が利用するファイル群を特定する．また，AP が利用するソケット通信を監視し，単一計算機内に留まらず，計算機を越えて資源利用情報を特定し，複数計算機上に跨る AP 実行環境を追跡する．さらに，銀行のオンラインシステムにおけるバッチ処理を参考としたファイルアクセス情報に対して追跡を行うことにより，提案手法の性能を評価した．その結果，実用的な時間内で AP 実行環境の追跡を行えることがわかった．

キーワード　クラウドコンピューティング，AP 実行環境，マイグレーション

Abstract

Migration method of software environment spanning multiple computers

Yusaku Kurogi

Cloud Computing are often implemented as wide-area distributed systems. In a wide-area distributed system, the execution environment of "application software" (called AP) may be transferred to another computer for efficient use. A typical migration method is migration of virtual machines. However, it is an inefficient method that takes time to migrate unnecessary resources.

In this research, we propose a method for specifying a software environment based on resource usage information. The proposed method monitors file access, and specifies files which are migrate based of file sharing information. In addition, it monitors socket communication among APs, and identifies files to be migrated by tracking spanning multiple computers. We evaluate the performance of the proposed method using file access information based on batch processing in a bank's online system. As a result, it was found that the proposed method can track the AP execution environment within a practical time.

key words Cloud computing, AP environment, migration

目次

第 **1** 章　はじめに　**1**

第 **2** 章　研究背景　**4**

2.1　広域分散システム上の実行環境の移送 4

2.2　関連研究 5

2.3　これまでの研究内容 6

2.3.1　特定ステップ 7

2.3.2　転送ステップ 8

第 **3** 章　複数計算機上に跨るソフトウェア実行環境の移送手法　**11**

3.1　移送モデル 11

3.2　監視方法 12

3.2.1　同一計算機内のファイルアクセス 12

3.2.2　同一計算機内の AP プロセスの親子関係 13

3.2.3　外部計算機の AP プロセスへの通信 13

3.3　追跡方法 14

3.3.1　同一計算機内のファイルの追跡 14

3.3.2　同一計算機内の親子プロセスの追跡 16

3.3.3　複数計算機上に跨る資源の追跡 17

3.4　移送後の通信方法 18

第 **4** 章　提案手法の実現　**20**

4.1　監視フェーズ 20

4.1.1　同一計算機内のファイルアクセス 21

4.1.2　同一計算機内の AP プロセスの親子関係 24

4.1.3 外部計算機の AP プロセスへの通信 25
4.1.4 ホスト名と IP アドレスの対応関係 26
4.1.5 PID と実行ファイルパスの対応関係 26
4.2 追跡フェーズ 28
4.2.1 各リストの結合 28
a) PID とパスの置き換え 30
b) IP アドレスの付加 30
c) ソケット通信を行っている AP の組の決定 30
d) 依存関係リストの作成 32
4.2.2 実行環境を構成する資源の追跡 32
4.3 移送後の通信 32

第 5 章 評価 **35**
5.1 監視フェーズ 35
5.1.1 評価内容 35
5.1.2 評価結果 36
5.1.3 考察 37
5.2 追跡フェーズ 38
5.2.1 評価内容 38
5.2.2 評価結果 40
5.2.3 考察 41

第 6 章 おわりに **43**

謝辞 **44**

参考文献 **45**

図目次

1.1 広域分散システム 2
1.2 仮想マシンイメージの移送 3
1.3 AP 実行環境の移送 3

2.1 広域分散システムの例 5
2.2 各隔離方式における FREEBSD Kernel make 実行環境の総移送データ量 . 7
2.3 研究 [5] におけるファイルアクセスの監視 8
2.4 研究 [8] におけるプリコピー順序 10

3.1 複数計算機上に跨る AP 実行環境の例 12
3.2 1 つの AP が 1 つのファイルにアクセスしている例 14
3.3 複数の AP が 1 つのファイルにアクセスしている例 16
3.4 プロセスの親子関係 17
3.5 複数計算機上で通信する AP 18
3.6 移送元での名前解決 19
3.7 移送先での名前解決 19

4.1 ファイルアクセスの監視 24
4.2 execve により実行された AP プロセス 27
4.3 リスト結合の全体図 29
4.4 ソケット通信を行う AP の組 31
4.5 ソケット通信のリスト結合 31
4.6 実行環境を構成する資源の追跡 33
4.7 名前解決リストの移送 34

5.1 実行時間測定の方法 36
5.2 実行時間測定の方法 (execve) 36
5.3 システムコールのオーバヘッド 37
5.4 ファイルアクセス 39
5.5 共有ファイルにアクセス 40
5.6 実行環境の追跡時間 41

表目次

4.1 実装環境 20
4.2 C 言語におけるシステムコールラッパー関数 22
4.3 execve_list の結合 30
4.4 リストへの IP アドレスの付加 31

5.1 追跡対象 AP パラメータ 40

第 1 章

はじめに

近年，ネットワークの発達や通信技術の向上により，クラウドコンピューティングの利用が広がっている．クラウドコンピューティング環境は，図 1.1 に示すように，地理的に離れた計算機がネットワークによって互いに接続された，広域分散システムとして実現していることが多い．広域分散システムは複数台の計算機で，分散処理を行うことで負荷の分散を行っている．しかし，システムを継続して利用していると，図中の計算機のように，ある計算機に負荷が集中する場合がある．そのような場合は，計算機上の AP やその実行環境を他の計算機に移送させる必要がある．また，負荷分散だけでなく，近年増加している水害や地震による物理計算機への被害を踏まえ，複数計算機に分散してバックアップを配置させることがある．これらの理由から，地理的に離れた計算機間での AP 実行環境の移送技術が重要となっている．クラウドコンピューティング環境における移送の代表的な手法として，図 1.2 のように，仮想マシンイメージを移送させる方法がある．しかし，仮想マシンの移送は移送したい実行環境だけでなく，他の AP のソフトウェア資源も移送してしまう．その結果，移送する必要のない資源を移送することにより，移送に時間がかかり，ネットワークへの負荷も大きくなる．

この問題に対して，我々は，AP のファイルアクセス情報を基に AP 実行環境を特定する移送手法を提案している (図 1.3)．提案手法を用いることで，移送したい実行環境に必要な最小限のファイル資源を移送前に特定でき，移送時間の短縮を実現する．これまで我々は，AP によるファイルアクセスを監視することにより，単一計算機内のファイル資源の利用状況を特定し，AP 実行環境を追跡する手法を提案してきた．しかし，この手法は単一計算機内に限っての追跡となる．一般的に，広域分散システム上で用いられている AP 実行環境

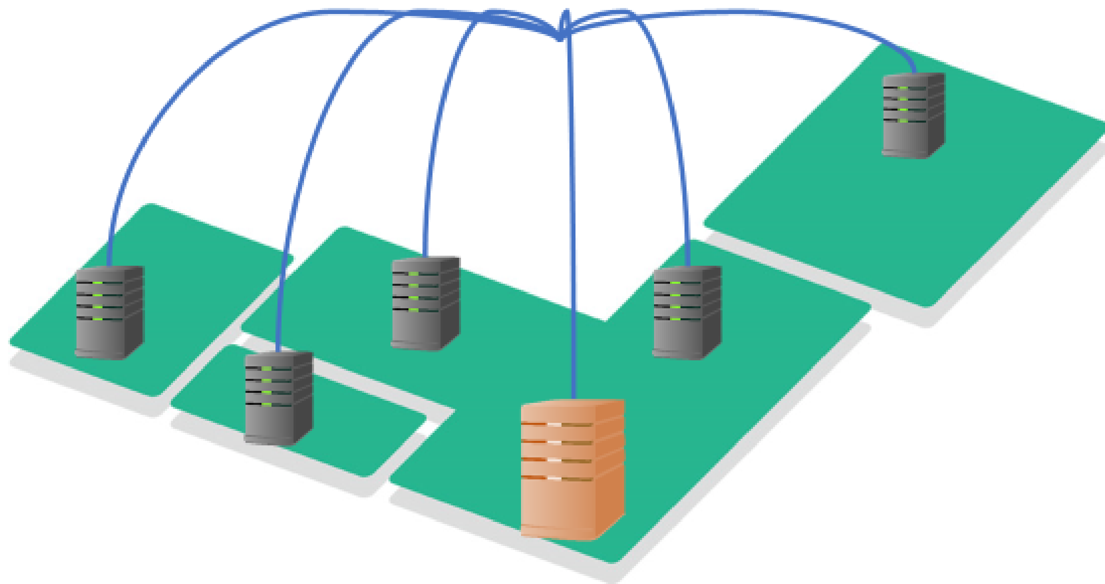


図 1.1　広域分散システム

は，複数の計算機の AP 同士が互いに通信を行いながら動作している場合がほとんどである．そのため前述の手法では完全に追跡することができないという問題があった．この問題に対し本研究では，AP 同士が通信を行う際のソケット通信に着目し，これを監視することにより，実行環境に必要なファイルと AP プロセスを，複数の計算機に跨って全て特定し，依存している全ての資源を明らかにする手法 (以降追跡と呼ぶ) を実現する．また，実行環境を移送する場合，移送元と移送先で所属する LAN が変わるため，そのままのネットワーク設定では通信できない．そこで本研究では，通信を行う際の名前解決を監視し IP アドレスとホスト名の対応関係を記録することで，記録した情報を基に移送後に通信を即座に復帰できるような手法を実現する．さらに，提案手法を計算機に実装した結果，通常と比較してどの程度オーバヘッドが発生するのかを明らかにする．また，AP 実行環境に対し提案手法を用いた追跡を行い, その有効性を明らかにする．

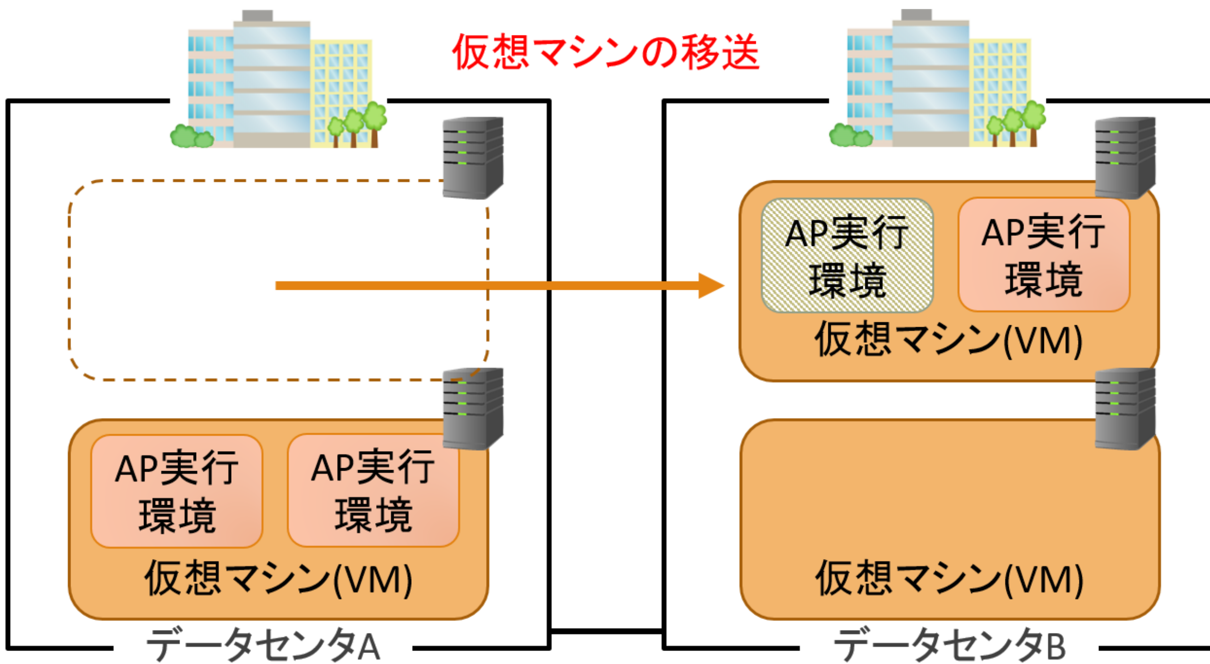


図 1.2　仮想マシンイメージの移送

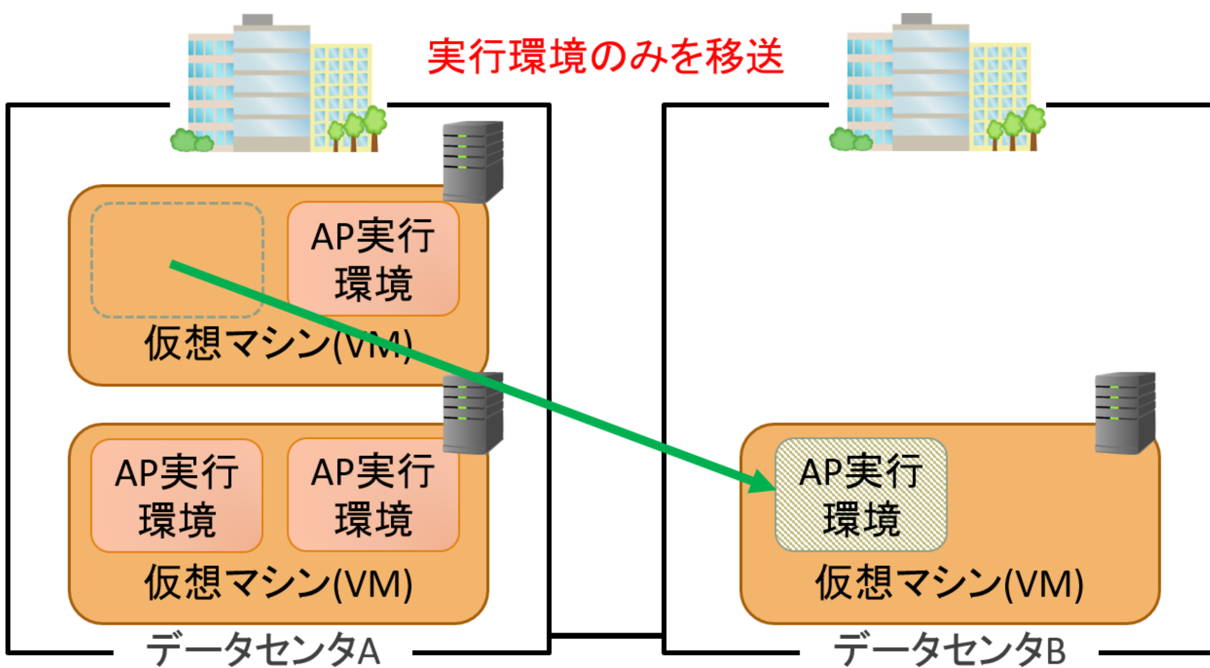


図 1.3　AP 実行環境の移送

第 2 章

研究背景

2.1 広域分散システム上の実行環境の移送

本研究で想定している広域分散システムの例を，図 2.1 に示す．図中の各データセンタは地理的に離れて存在している．これらのデータセンタは，図中の色付き線で示したような専用線で接続されている．また，各データセンタ内には複数の計算機があり，データセンタ内のそれぞれの計算機もネットワークで接続されている．内部ネットワークとなるデータセンタ内の回線速度は数 Gbps といった比較的高速なネットワークであることが多いが，外部データセンタへの回線速度は専用線を用いても数十 Mbps 程度である場合が多い．こうした広域分散システムで，仮想マシンを移送させる場合を考える．仮想マシンは，その OS イメージとファイル資源を全て含めると，総サイズが数 GB となる場合が多い．回線速度が数 Gbps となるようなデータセンタ内では，仮想マシンの移送も短時間で実現できる．一方で，回線速度が数 Mbps となるような，異なるデータセンタへの仮想マシンの移送は，資源の移送に非常に長い時間を要する．また，移送のために専用線を圧迫することにより，同じ専用線を利用して通信している他の実行環境の処理に，遅延が発生するなどの影響を与える可能性がある．以上の理由から，仮想マシンの移送に関する研究は通常，同ネットワーク内で行われることを前提としている．しかし，広域分散システムでは移送先が外部データセンタである場合がほとんどである．よって地理的に離れたデータセンタ間での移送を効率的に行うための手法が必要となる．

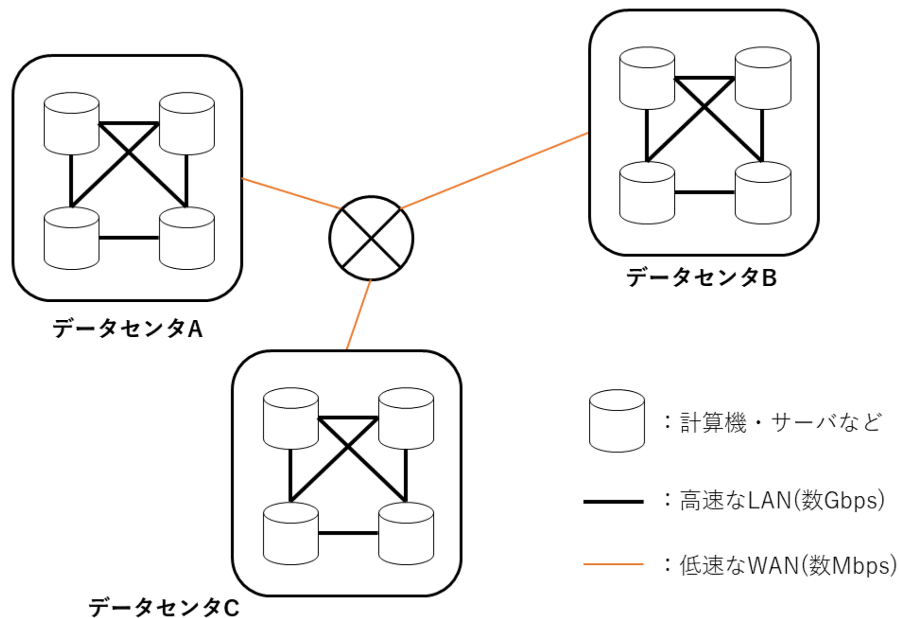


図 2.1　広域分散システムの例

2.2　関連研究

広域分散システムでの，仮想マシンや実行環境の移送に対する研究は大きく 2 つに分けられる．

1 つ目が，移送データ量の削減や，移送時間・マシン停止時間の短縮を主目的としたものである．代表的な手法として，プリコピー型マイグレーションがある．プリコピー型マイグレーションは，ライブマイグレーションの際に，大部分の資源をあらかじめ移送先にコピーしておくことにより，マシン停止時間を短縮するための手法である．プリコピー型マイグレーションを対象とした研究に，文献 [1] がある．文献 [1] ではメモリ単位でのプリコピー型マイグレーションを実現している．通常，プリコピー型のメモリマイグレーションはメモリの先頭から順に転送を行うが，文献 [1] では，更新頻度の低い順に転送を行うように転送順序を操作することにより，プリコピー型マイグレーション時に大きな問題となる再送の発生を抑制している．文献 [1] は固定サイズであるメモリ単位での移送を対象としている．そのため，可変サイズとなるファイル移送には適さない．

2 つ目が，異なるネットワーク環境となる移送先での，正常な通信を主目的としたものである．近年では，代表的な手法として，Software Defined Networking(SDN) を利用したものが多い．よく知られた SDN である OpenFlow[2] を用いた研究に，文献 [3] がある．文献 [3] では仮想マシンのライブマイグレーション時に，移送元と移送先の双方に展開した OpenFlow ネットワークを利用することで，従来の類似手法と比較した，通信途絶の短縮を実現している．文献 [3] は OpenFlow により柔軟なネットワークを実現しているが，実際の計算機に適用させる場合，OpenFlow に対応した環境を用意する必要がある．

2.3　これまでの研究内容

これまでに我々は，実行環境の移送における移送データ量の削減手法を提案している．[4][5][6][7][8].

研究 [4] では，資源の競合を防ぐために，その資源の隔離を保証しながら移送データ量を削減するための手法を提案している．具体的には，FREEBSD Jail における OS 仮想化，chroot におけるファイルシステム仮想化による資源の隔離方式を用いた場合，仮想マシンイメージの移送と比較し，どの程度移送データ量を削減することができるのかを明らかにしている．FREEBSD の Kernel make に必要な資源の移送データ量を比較した結果，図 2.2 に示すように，最大約 93%の移送データ量を削減している．

しかし，研究 [4] の手法では，隔離空間を超えたプロセス間通信は隔離されないため，プロセス間通信を行う AP を隔離・移送できない．そこで，AP 実行環境の特定と転送順序の操作を利用して，実行環境の移送データ量を削減する手法を提案している．我々の研究では，移送する実行環境を特定・追跡する部分を「特定ステップ」，特定・追跡した実行環境を，移送する部分を「転送ステップ」と呼び区別している．

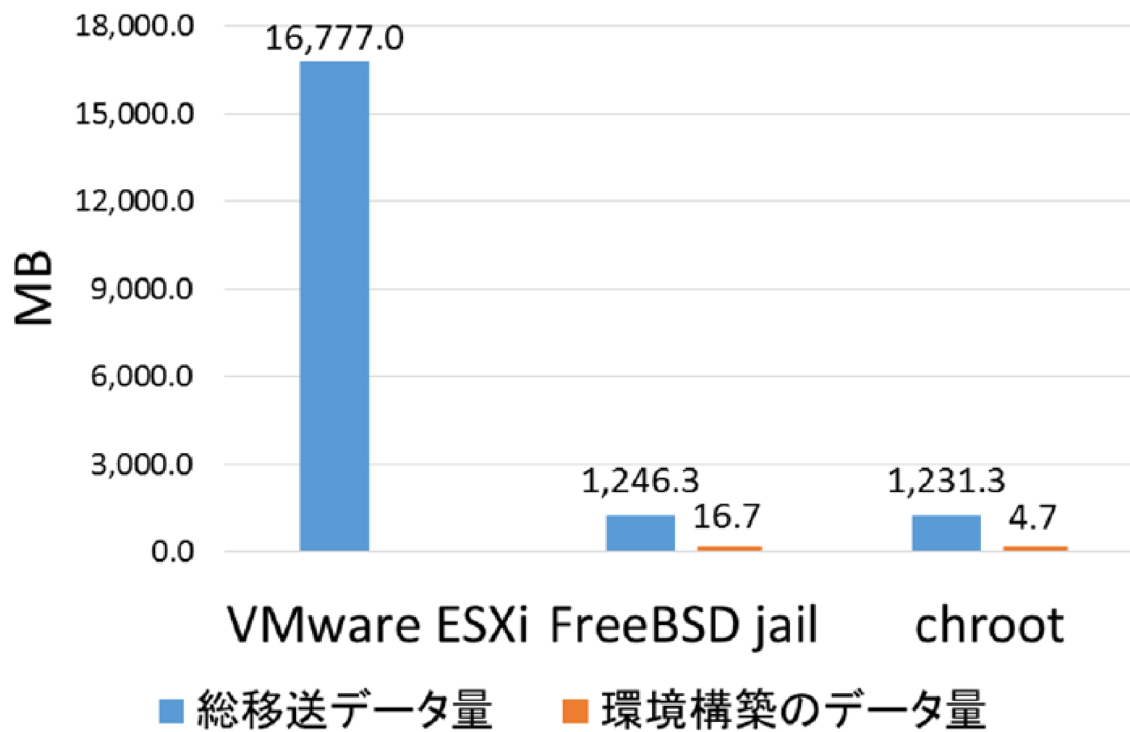


図 2.2　各隔離方式における FREEBSD Kernel make 実行環境の総移送データ量

2.3.1　特定ステップ

特定ステップでは，移送したい AP 実行環境を，移送前に特定・追跡するステップである．研究 [5] では，図 2.3 に示すように，AP がファイルにアクセスする際に発行する open システムコールを，C ライブラリ内で監視・記録することで，OS に依存しないユーザレベルでの特定手法を実現した．また，実行環境のファイルの依存関係を，独自のアルゴリズムを用いることで追跡し，線形時間での追跡を実現した．

また，研究 [6] では，研究 [5] における手法を，単一計算機内のプロセス間通信を含め，ネットワークを介して通信する他の計算機に拡張したものを提案している．研究 [6] は fork/execve/accept/connect の各システムコールを監視することで，AP が利用するソケット通信を監視し，同一計算機内のプロセスの親子関係や，別計算機との依存関係を追跡する．この手法は，fork/execve を OS カーネル内で実装しているため OS に依存するという点や，実際の AP 実行環境を対象とした評価が不十分であった点が課題となった．

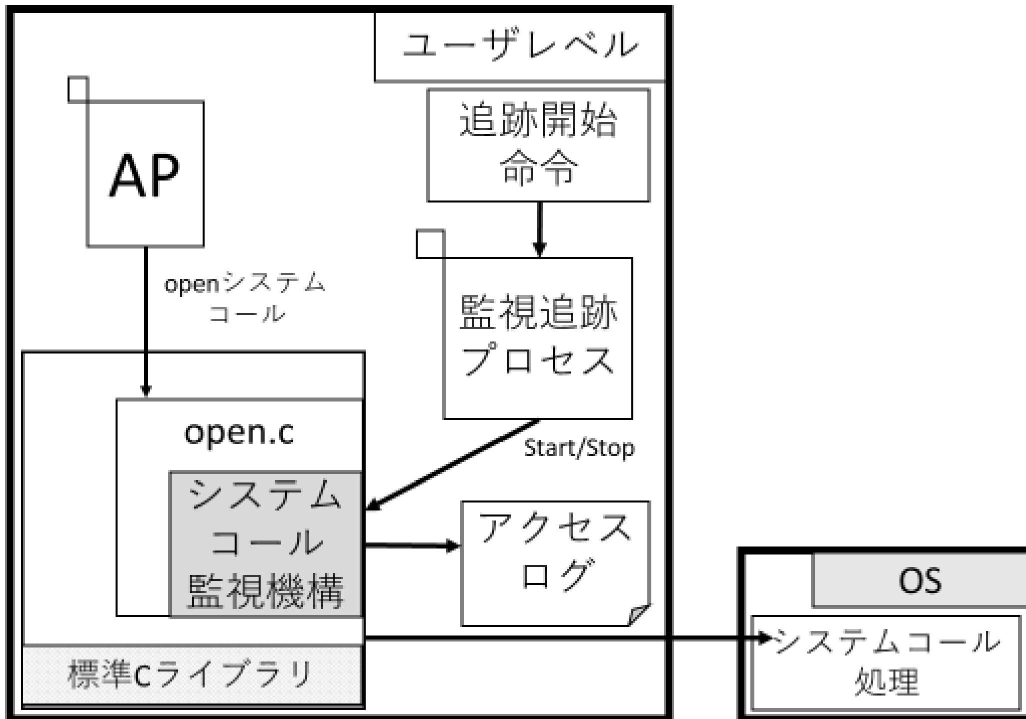


図 2.3　研究 [5] におけるファイルアクセスの監視

2.3.2　転送ステップ

転送ステップでは，特定ステップで特定した AP 実行環境をファイル単位で移送先に転送するステップである．研究 [7] では，文献 [1] が仮想マシン・メモリページを対象としたものであるのに対し，ファイルを対象とした移送を行っている．研究 [7] では，ファイルサイズとファイル更新頻度から独自の期待値を算出し，ファイルごとに期待値を付与する．この期待値を基にファイルの転送順序を操作し，プリコピー時の再送の発生を抑制している．その結果，文献 [1] に比べ，プリコピー時の転送量を最大 93%，サービス停止時の転送量を最大 19%削減している．研究 [7] は，プリコピー時の再送抑制に着目していたため，サービス停止時間の短縮が不十分であった．そこで，研究 [8] では，サービス停止時間の短縮に着目した．具体的には，図 2.4 に示すように，ファイルのアクセス種別 (Read-only，Read-Write)

と，ファイルサイズ・ファイルのアクセス頻度を基に，プリコピーするファイルの転送順序を操作することにより，ファイルのアクセス頻度昇順での転送が，既存手法に比べ約 68%，サービス停止時間の短縮ができることを明らかにした．

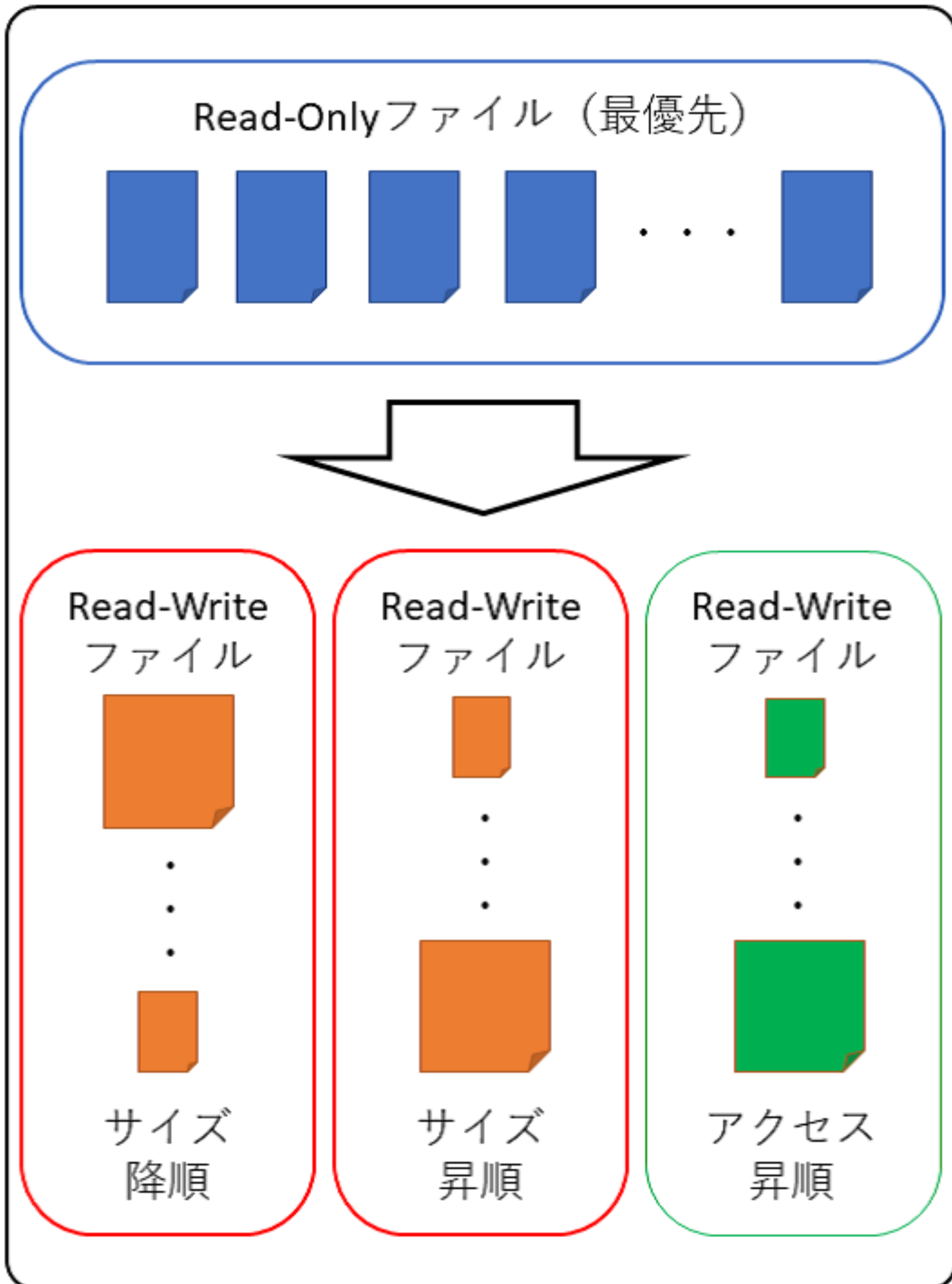


図 2.4　研究 [8] におけるプリコピー順序

第 3 章

複数計算機上に跨るソフトウェア実行環境の移送手法

本研究では特に特定ステップにおいて，複数計算機上に跨る AP 実行環境の特定・追跡を行い，実際の AP 実行環境を意識した環境における評価を行う．これにより実際の AP 実行環境での，提案手法の有効性を明らかにする．また，実行環境の移送後に体系の異なるネットワークにおいて，名前解決情報を利用して，正常に通信を復帰できるような手法を実現する．

3.1　移送モデル

本研究で想定する複数計算機上に跨る AP 実行環境の例を図 3.1 に示す．このような実行環境は通常，複数の AP が同じファイルを共有したり，互いに通信をしながら動作している．例えば，図中の AP1 は 2 つのファイルにアクセスしているが，そのうち 1 つは AP2 もアクセスしている．よってこのファイルは共有しているということになる．また，AP1 は AP2 と計算機内で親子関係にあるプロセスである．AP2 は他の計算機内の AP3 と，AP3 は AP4 と，それぞれソケットを利用して通信を行っている．よって青色で示した範囲は全て AP1 から見た実行環境である．本章では以下，この実行環境を対象とした監視・追跡のモデルを説明する．

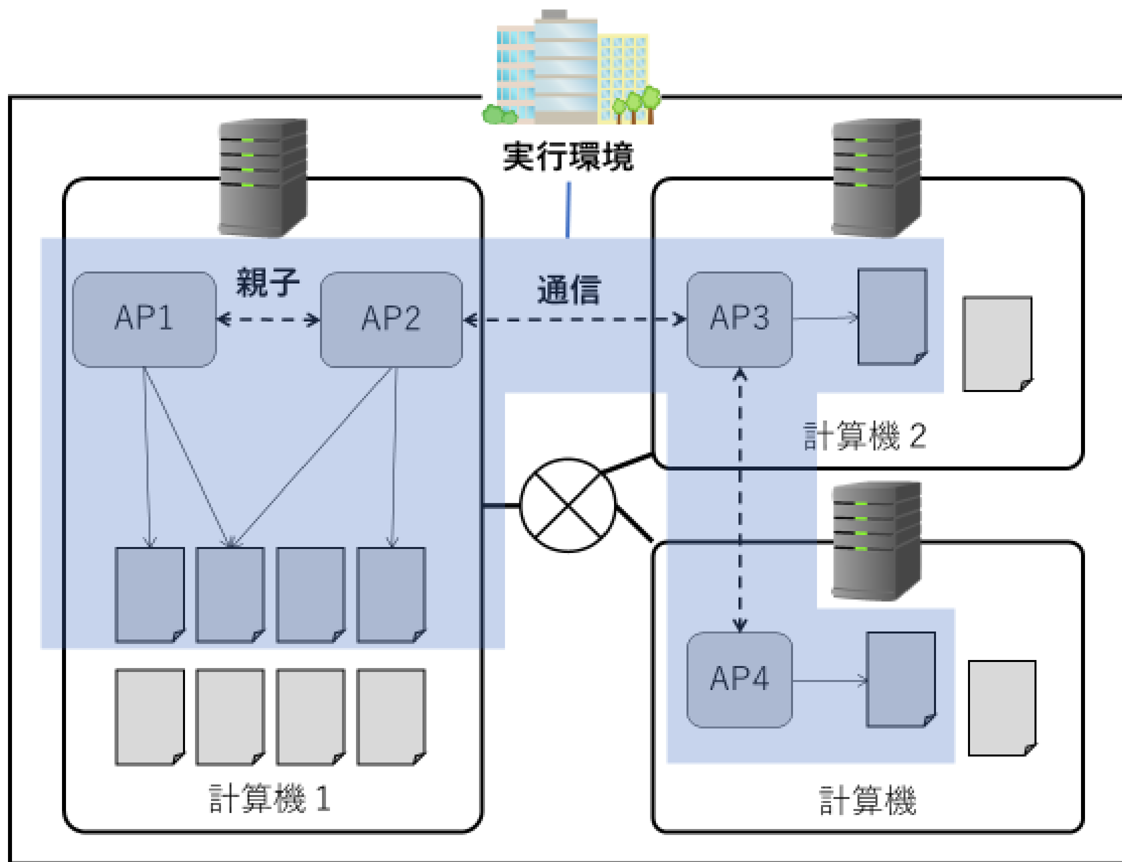


図 3.1　複数計算機上に跨る AP 実行環境の例

3.2　監視方法

AP がファイルアクセスする場合や他の AP と通信を行う場合，システムコールを発行する．提案手法は，このシステムコールを監視することにより，AP 実行環境に必要な AP やファイルを特定する．具体的に，提案手法で監視するシステムコールは open/openat, close，fork/vfork，accept，connect，execve である．また，名前解決の際に利用される gethostbyname，getaddrinfo 関数を同様に監視する．

3.2.1　同一計算機内のファイルアクセス

ある AP の，同一計算機内のファイルへのアクセスは open，close を監視して特定する．ここでは，既に open しているディレクトリ階層を基準に open 処理を行う openat も同様に

監視する．ファイルから何らかのデータを読み取る際には read，ファイルに何らかのデータを書き込む際は write がそれぞれ逐一利用されるが，同一ファイルに対しての読み書きの回数に関わらず，open と close は 1 セットずつ行われる．また，open されたファイルは，正常に AP が動作している限り必ず close される．そのため，close が発行されたタイミングで open されていたファイルを記録する．

3.2.2 同一計算機内の AP プロセスの親子関係

ある AP プロセスと親子関係にある，同一計算機内の AP プロセスは，生成される際の fork システムコールを監視して特定する．レガシーなシステムでは，execve システムコールにより別プロセスに書き換えることを前提に，親子でメモリページを共有しプロセス生成のオーバヘッドを軽減する，vfork システムコールが利用される場合がある [9]．よって vfork システムコールも監視対象とする．

3.2.3 外部計算機の AP プロセスへの通信

ある AP プロセスが，ネットワーク上の他の計算機内で動作する AP プロセスと通信を行う場合，ソケット通信を利用する．ソケット通信を行っている AP の組み合わせを，ソケット通信の受信側では accept システムコール，送信側では connect システムコールを監視して特定する．また，この時ホスト名から，DNS や hosts を利用して名前解決を行い IP アドレスを取得する際に，gethostbyname，getaddrinfo 関数が利用される．これを監視することで，ホスト名と IP アドレスの組み合わせを特定する．この組み合わせを名前解決リストとして記録する．記録した情報は，移送後に IP アドレスを書き換え，体系の異なるネットワークにおいて通信を継続させるために利用する．

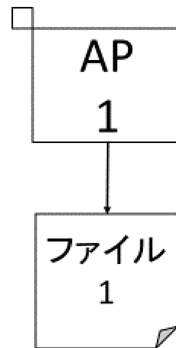


図 3.2　1 つの AP が 1 つのファイルにアクセスしている例

3.3　追跡方法

3.3.1　同一計算機内のファイルの追跡

通常，AP はファイルを単独で占有して利用しているのではなく，同一計算機内の他の AP とファイルを共有している．そのため，ある AP を移送する場合，その AP が利用しているファイル群と，各々のファイルを共有している AP を全て把握する必要がある．提案手法では，ファイルのアクセス種別に着目した追跡アルゴリズムを考えている．この追跡アルゴリズムについて説明する．

ファイルへのアクセス種別は，大別すると，ファイルから何らかのデータを読み取る場合と，ファイルに何らかのデータを書き込む場合の 2 種類に分けられる．ここでは書き込みを行わない場合を read-only ファイル，書き込みを行う場合を read-write ファイルと定義する．提案手法では，AP のファイルの共有状況と，アクセス種別によって AP とファイルの移送方法を変更する．

初めに，図 3.2 のように，1 つの AP が 1 つのファイルにアクセスしている場合である．このとき図中の AP1 を移送対象とする場合を考える．この場合，ファイル 1 に対するアクセスが read-only であるか，read-write であるかに関わらず，AP1 の実行にはファイル 1 が必要である．また，ファイル 1 は AP1 以外の AP には一切利用されていない．よって図 3.2 のような場合は，AP1 とファイル 1 を両方移送対象とする．

次に，図 3.3 のように，複数の AP が 1 つのファイルにアクセスしている場合である．図中のファイル 2 は複数の AP に共有されていると言える．ここで図中の AP1 を移送する場合を考えるが，各 AP からのファイルのアクセス種別によって次の 4 パターンを考える．

(パターン 1-1)　全ての AP が read-only の場合

全ての AP が read-only の場合，ファイル 2 の内容が書き換えられることはない．そのため，ある AP の動作により他の AP に影響を与えることはない．従って，AP1 とファイル 2 のみを移送する．ただし，移送してファイル 2 が移送元から失われた場合はその他の AP の動作に影響を与える．そのためファイル 2 のコピーを移送元に残しておく．

(パターン 1-2)　AP1 が read-write，その他の AP が read-only の場合

AP1 が read-write の場合，ファイル 2 の内容は AP1 の動作により決まることになる．そのため，ファイル 2 の内容を read-only で利用している他の AP は全て AP1 の動作に依存しているということになる．従って，AP1 とファイル 2 に加え AP2 から APi を移送対象とする．

(パターン 1-3)　AP1 が read-only，その他の AP の 1 つ以上が read-write の場合

AP2 から APi までのいずれかの AP が read-write の場合，ファイル 2 の内容はその AP の動作により決まることになる．そのため，ファイル 2 の内容を read-only で利用している他の AP は全て read-write でアクセスしている AP の動作に依存しているということになる．従って，AP1 とファイル 2 に加え，AP2 から APi を移送対象とする．

(パターン 1-4)　全ての AP が read-write の場合

全ての AP が read-write の場合，各 AP の動作が他の AP の動作に依存しているということになる．従って，AP1 とファイル 2 に加え，AP2 から APi を移送対象とする．

各パターンをまとめ，以下の 2 通りのパターンで移送を考える．

(パターン 2-1)　全ての AP が read-only の場合

AP1 を移送し，ファイル 2 をコピーして移送する．

(パターン 2-2)　いずれかの AP が read-write の場合

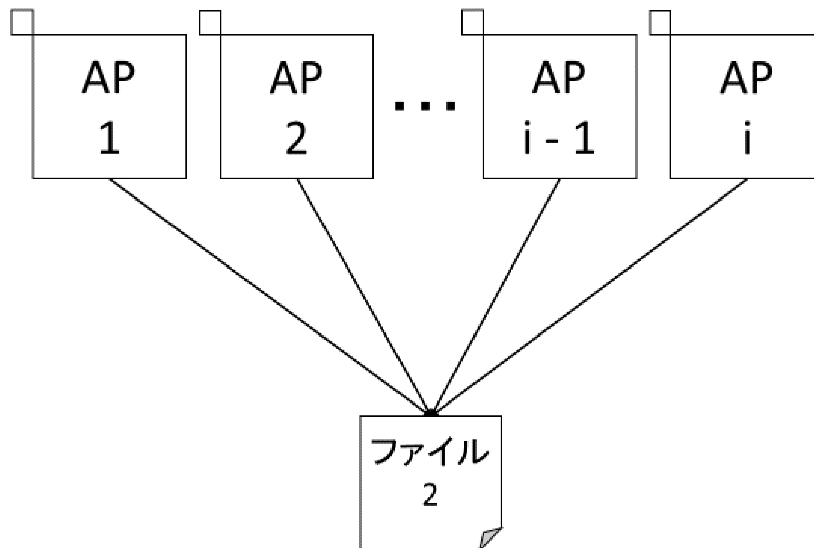


図 3.3 複数の AP が 1 つのファイルにアクセスしている例

AP1 から APi とファイル 2 を全て移送する．

3.3.2 同一計算機内の親子プロセスの追跡

プロセスは，並行処理や負荷分散などの目的で，自身の子プロセスを逐一生み出しながら動作している場合がある．例として，代表的な HTTP ウェブサーバである Apache の場合，数個〜数十個のプロセスを起動することで，同時に多数のリクエストへの応答を実現している [11]．従って，ある AP プロセスの親子関係を把握する必要がある．追跡の例を図 3.4 を用いて説明する．

図中の AP1 はファイル 1 にアクセスしている．同様に AP2 はファイル 2 にアクセスしている．また，AP2 は AP1 の子プロセスである．同様に AP3 は AP2 の子プロセスである．従って，AP1 から見て AP3 は孫プロセスとなる．このような環境で AP3 を移送対象とする場合を考える．AP3 は AP2 から生成されたプロセスである．従って，AP3 を移送するということは AP2 の実行環境が必要となるということである．よって AP2 を移送対象

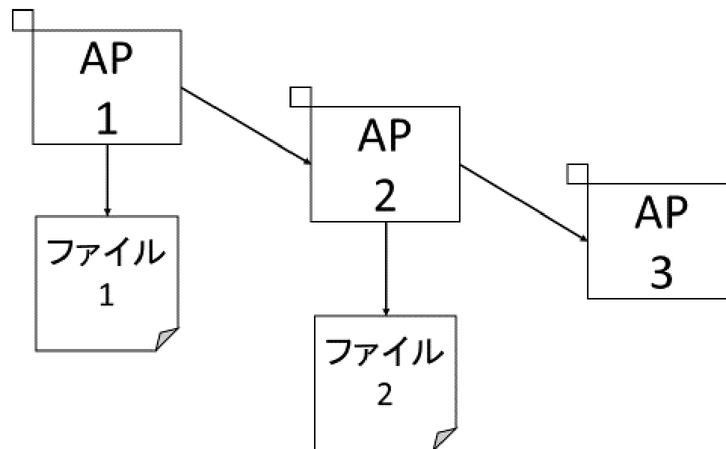


図 3.4　プロセスの親子関係

とする．また AP2 が利用しているファイル 2 も移送対象となる．同様に AP2 は AP1 から生成されたプロセスであるので，AP1 も移送対象となり，AP1 が利用しているファイル 1 も移送対象となる．このように，ある AP プロセスを特定して移送対象とする場合，プロセスの親子関係を追跡することで，その AP プロセスの実行環境を全て把握することが可能となる．

3.3.3　複数計算機上に跨る資源の追跡

AP は同一計算機内だけでなく，他の計算機に存在している AP と通信を行っている場合が多い．そのため，ある AP を移送する場合，計算機を超えて通信を行っている AP を把握する必要がある．追跡の例を図 3.5 を用いて説明する．

図中の計算機 1 では AP1 がファイル A とファイル B を利用している．同様に，計算機 2 では AP2 がファイル C とファイル D を利用している．また，計算機 1 の AP1 は計算機 2 の AP2 と通信を行っている．このような環境で AP1 を移送対象とする場合を考える．AP1 はファイル A とファイル B を利用している．よってファイル A とファイル B を移送対象とする．そして，AP1 は計算機 2 上の AP2 と互いに通信している．そのため，AP2 が存在しないと AP1 の実行に影響を与える場合がある．従って，AP2 を移送対象とする．同様に

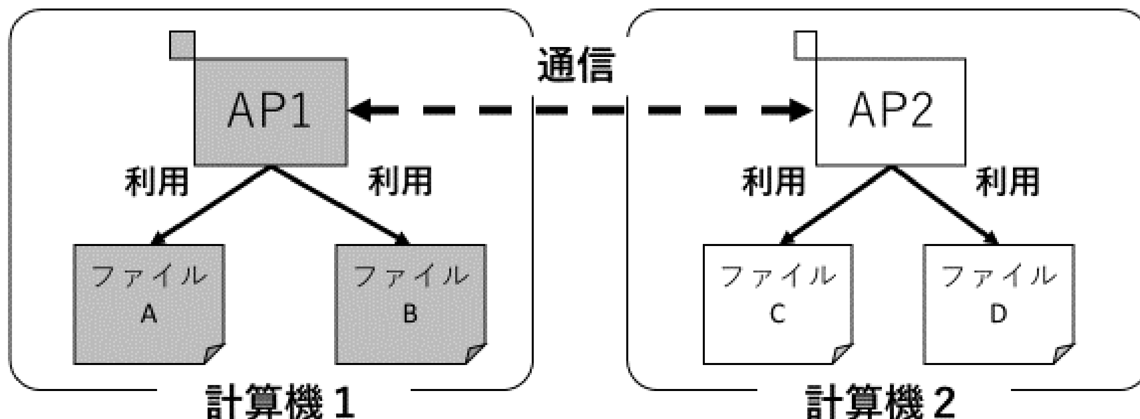


図 3.5　複数計算機上で通信する AP

AP2 が利用しているファイル C とファイル D も移送対象となる．提案手法では，このように計算機を超えて通信を行っている AP の組み合わせと，その AP が利用しているファイルを追跡し，複数計算機上に跨る AP 実行環境の追跡を可能としている．

3.4　移送後の通信方法

各 AP は移送後に，移送前とは全く異なるネットワーク内に配置されることになる．そのため，AP はそのままでは通信を続けることができなくなる．そこで提案手法では，通信先の計算機に通信を行う際に，AP が名前解決を行い IP アドレスを取得する際の DNS による変換処理に着目する．

DNS は，ホスト名と IP アドレスの対応関係を記述した対応表であるゾーンファイルを格納している．図 3.6 のように，ローカルの対応関係を記した DNS に対して名前解決を依頼することで，ホスト名から計算機の IP アドレスを取得している．しかし，AP を移送すると，それまで使用していた DNS とは別の DNS を使用することになる．例として，図 3.6 における AP 実行環境を，移送先に作成した新たなホストへ移送した例を図 3.7 に示す．移送先の DNS には新たに移送された AP の属するのホスト名と IP アドレスの対応関係を持っていない．よって，新たに移送された AP の属するホストに対して通信を行う場合，正常な通信を行うことができない．

3.4　移送後の通信方法

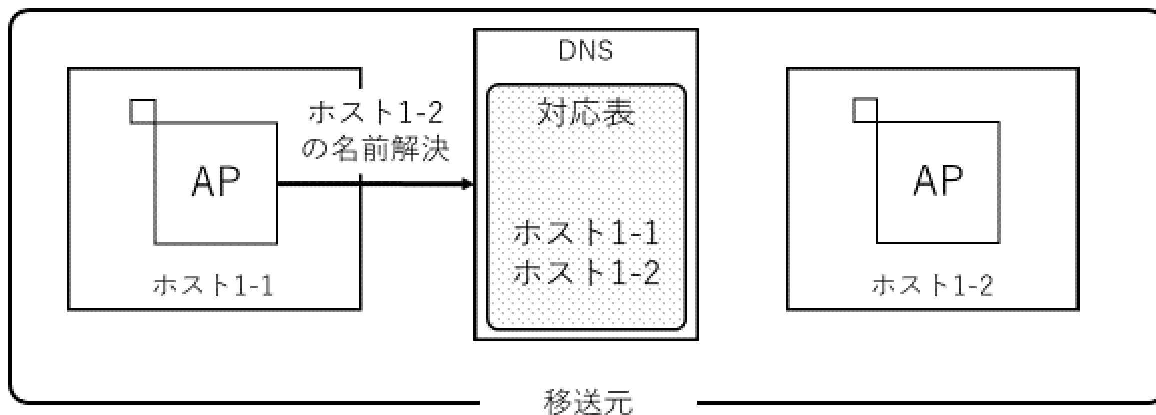


図 3.6　移送元での名前解決

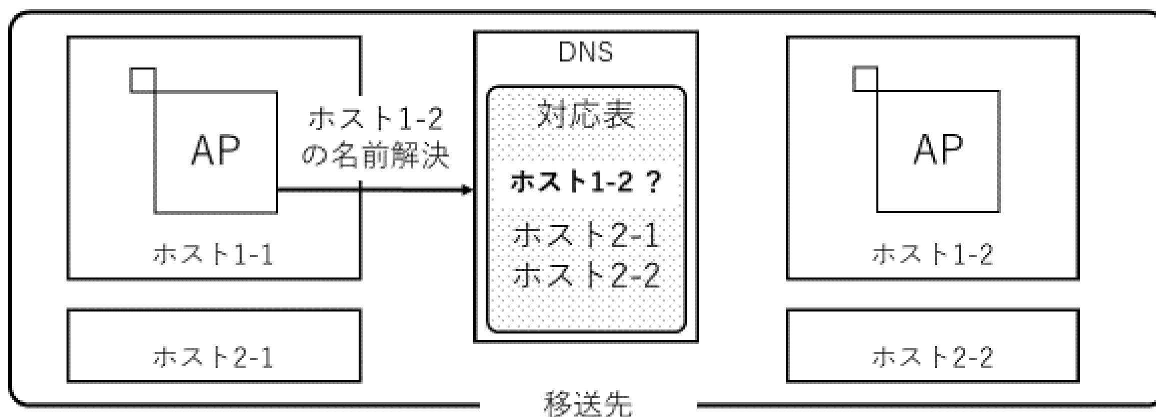


図 3.7　移送先での名前解決

そこで，提案手法では移送前に AP の名前解決を行う gethostbyname, getaddrinfo 関数を監視することで，通信するホスト名と IP アドレスの対応関係を記録し，実行環境の移送時に移送先の DNS へ対応関係を書き込むことで，移送後の正常な通信を可能とする．

第 4 章

提案手法の実現

本章では実際に計算機に実装する際に，提案手法をどのように実現させるのかを説明する．提案手法を実装する計算機の情報を表 4.1 に示す．以降この環境を実装環境と呼ぶ．また，本章の基準内容は特筆しない限り実装環境におけるものとする．提案手法では，システムコールを監視し情報を記録する部分を”監視フェーズ”，記録した情報を基に実行環境の追跡を行う部分を”追跡フェーズ”と呼ぶ.

4.1 監視フェーズ

監視フェーズでは各システムコールを監視することで，追跡を行うためのリストを作成する．前章で述べたように，監視するシステムコール・関数は open/openat，close，fork/vfork，accept，connect，gethostbyname，getaddrinfo，execve である．本節では各システムコール・関数を監視する際の情報取得の方法を説明する．

システムコールの実処理はカーネル内で行われる．しかし，カーネル内で監視を行う場合，OS のカーネルを make する必要がある．さらに，OS ごとにソースコードが異なるた

表 4.1 実装環境

OS	FREEBSD 11.2-RELEASE
プロセッサ	Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
メモリ	8GB
C ライブラリ	BSD libc

め移植性に問題がある．そこで本研究では，ユーザレベルでライブラリを持つ C 言語のシステムコールのラッパー関数内で監視を行う．各システムコールの C 言語におけるラッパー関数の関数名と，ソースファイルのパスを表 4.2 に示す．gethostbyname と getaddrinfo はシステムコールではないため，関数の実体が記されたソースファイルを示している．表中では C 言語ライブラリのルートディレクトリを libc としている．

ラッパー関数の内，vfork と execve については C 言語で記述されたソースファイルがデフォルトでは存在していないという問題がある．そこで自作した/libc/sys/execve.c と/libc/sys/vfork.c を配置し，その関数を経由してからデフォルトの関数を呼び出すようにライブラリをコンパイルして解決した．ただし，vfork は execve もしくは exit を呼び出すまでに他の関数を呼び出すとエラーとなるため [9]，デフォルトの__sys_vfork ではなく，通常の fork を経由するようにしている．これにより全てのシステムコールのユーザレベルでの監視を実現した．

各システムコールで特定した情報は CSV ファイルとして出力する．各システムコールの情報の記録方法を以下で説明する．

4.1.1 同一計算機内のファイルアクセス

AP がアクセスしたファイルを特定するため，open/openat/close 各システムコールを監視する．open/openat/close 各システムコールの定義を以下に示す．

```
int open(const char *pathname, int flags);
int openat(int dirfd, const char *pathname, int flags);
int close(int fd);
```

ここでは以下の情報を監視して，CSV ファイルとして記録する．

- open() を発行した AP プロセスの PID
- open するファイルのパス
- open する際のアクセス種別

open() を発行した AP プロセスの PID は関数内で getpid() を用いて取得する．open するファイルのパスは open の定義中の変数*pathname からコピーして取得する．最後に open する際のアクセス種別であるが，read-only を 0，read-write を 1 として記録する．この情報は open の定義中の変数 flags から取得する．open システムコールにおけるファイルアクセスの種別 (定義中の flags) は，その目的により必ず O_RDONLY(read-only)，O_WRONLY(write-only)，O_RDWR(read-write) のいずれかが利用される．提案手法では，open システムコールが利用される際のアクセス種別と，read および write が発行される際の組み合わせを次の 3 パターンで考える．

(パターン 1)　O_RDONLY でアクセスされ，read が行われる場合 もしくは read が行われない場合

(パターン 2)　O_WRONLY もしくは O_RDWR でアクセスされ，write が行われる場合

(パターン 3)　O_WRONLY もしくは O_RDWR でアクセスされ，write が行われない場合

表 4.2　C 言語におけるシステムコールラッパー関数

システムコール名	実体となる関数	ソースファイル
open	open()	libc/sys/open.c
openat	openat()	libc/sys/openat.c
close	close()	libc/sys/close.c
fork	fork()	libc/sys/open.c
vfork	__sys_vfork()	libc/i386/sys/Ovfork.S
accpet	accept()	libc/sys/accept.c
connect	connect()	libc/sys/connect.c
gethostbyname	gethostbyname()	libc/net/gethostbynamadr.c
getaddrinfo	getaddrinfo()	libc/net/getaddrinfo.c
execve	execve	libc/_execve.S

4.1 監視フェーズ

パターン 1 では，read が行われる・行われないに関わらず O_RDONLY でアクセスされているのでファイルの内容が書き換えられることはない．このようなファイルは read-only ファイルとして記録する．パターン 2 とパターン 3 は O_WRONLY もしくは O_RDWR でアクセスされる場合である．すなわち write が行われファイルの内容が書き換えられる可能性のあるパターンとなる．ここで問題となる点が，パターン 3 のように O_WRONLY や O_RDWR でアクセスしていても write が行われない場合がある点である．open のみの監視の場合では，パターン 2 とパターン 3 を判別することはできない．そこで提案手法では，O_WRONLY や O_RDWR でアクセスされたファイルは，実際に書き込み処理が行われたかどうかに関わらず read-write ファイルとして記録する．

open が行われると，AP が正常に動作している限り必ず close が行われる．そこでファイルアクセスは以下の手順で記録する．

1. open() 内で上述した情報を格納するためのグローバル変数を用意
2. open() 内で用意したグローバル変数に情報を格納
3. close() が発行された際に，open() で情報を格納したグローバル変数の情報を，テキスト形式で CSV ファイルに出力

監視の様子を図 4.1 に示す．初めに，AP が open システムコールを発行した際，open.c 内で定義されているグローバル変数に，上述した情報を格納する．その後，AP でファイルに対して read や write などの処理が行われ，最終的に該当ファイルに対して close システムコールを発行する．このとき，close.c 内では open.c 内で定義したグローバル変数を extern を用いて利用する．extern はソースファイルを超えて変数を共有できるため，open を発行した AP が終了しない限りメモリ上に残り続ける．close システムコールの処理を行う際に，記録した情報を全てファイルに書き出すことにより，情報を保存する．

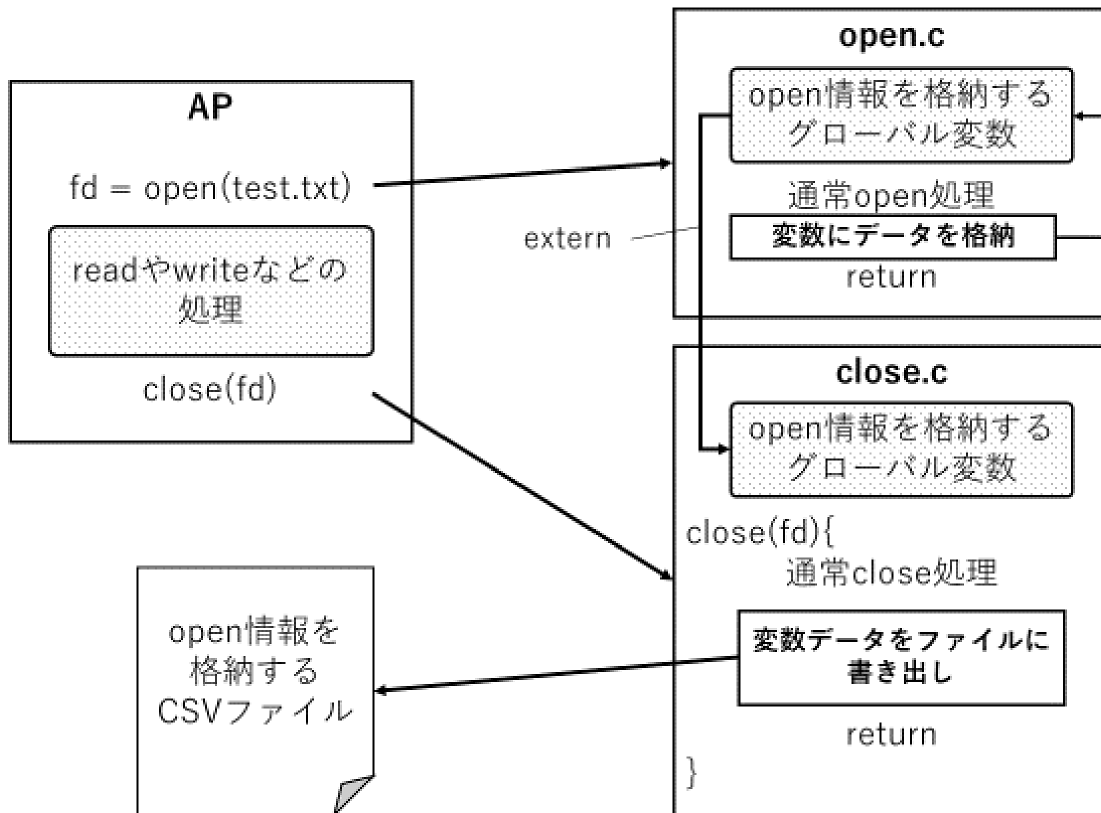


図 4.1 ファイルアクセスの監視

4.1.2 同一計算機内の AP プロセスの親子関係

AP プロセスの親子関係を特定するため，fork/vfork 各システムコールを監視する．fork/vfork 各システムコールの定義を以下に示す.

```
pid_t fork(void);
pid_t vfork(void);
```

ここでは以下の情報を監視して，CSV ファイルとして記録する.

- fork() を発行した AP プロセスの PID(親プロセスの PID)
- fork() によって生成された AP プロセスの PID(子プロセスの PID)

fork() を発行した AP プロセスの PID は，関数内で getpid() を用いて取得する．fork()

によって生成された AP プロセスの PID は fork システムコールの返り値から取得する．

4.1.3　外部計算機の AP プロセスへの通信

ネットワークを跨る AP プロセスの通信を特定するため，ソケット通信時に利用される accept/connect 各システムコールを監視する．accept/connect 各システムコールの定義を以下に示す．

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen );
int connect(int sockfd, const struct sockaddr
*serv_addr, socklen_t addrlen );
```

ここでは，受信側では accept，送信側では connect から以下の情報を監視して，CSV ファイルとして記録する．記録する内容は両システムコールともに同じとなる．

- accept()/connect() を発行した AP プロセスの PID(相手の IP アドレス)
- 通信先計算機の，接続している IP アドレス (相手のポート番号)
- 通信先計算機の，接続しているポート番号
- accept()/connect() を発行した AP が存在する計算機の IP アドレス (自分の IP アドレス)
- accept()/connect() を発行した AP が接続を受け付けたポート番号 (自分のポート番号)

accept()/connect() を発行した AP プロセスの PID は，関数内で getpid() を用いて取得する．通信先計算機の，接続している IP アドレス・ポート番号は accept()/connect() の定義中の構造体*addr/*serv_addr に格納されているのでコピーして取得する．accept()/connect() を発行した AP が存在する計算機の IP アドレスと，AP が接続を受け付けたポート番号は，ソケットディスクリプタ (accept()/connect() の定義中の sockfd) から取得する．ここではソケットディスクリプタからソケットの名前 (構造体 sockaddr) を取得する関数 getsockname() を利用する．

4.1.4 ホスト名と IP アドレスの対応関係

通信している計算機のホスト名と IP アドレスの関係を取得するため，名前解決時に利用される gethostbyname/getaddrinfo 各関数を監視する．gethostbyname/getaddrinfo 各関数の定義を以下に示す.

```
struct hostent * gethostbyname(const char *name);
int getaddrinfo(const char *hostname, const char *servname,
const struct addrinfo *hints, struct addrinfo **res;)
```

ここでは，以下の情報を監視して，CSV ファイルとして記録する.

- gethostbyname()/getaddrinfo() を発行した AP プロセスの PID
- 名前解決したホスト名
- 名前解決して得られた IP アドレス

gethostbyname()/getaddrinfo() を発行した AP プロセスの PID は，関数内で getpid() を用いて取得する．名前解決したホスト名は各関数の引数である*name/*hostname からコピーして取得する．名前解決した結果得られた IP アドレスは，gethostbyname() の場合，返り値である hostent 構造体の内部から取得する．getaddrinfo() の場合，addrinfo 構造体の内部に格納された sockaddr 構造体から取得する.

4.1.5 PID と実行ファイルパスの対応関係

AP プロセスの絶対パスを取得するために execve システムコールを監視する．execve システムコールの定義を以下に示す.

```
int execve(const char *filename, char *const argv[],
char *const envp[]);
```

(1)〜(4) までの各システムコール・関数は，それを発行した AP プロセスの実行ファイルのパス名を直接知る手段が存在しない．そこで，execve システムコールを監視することで

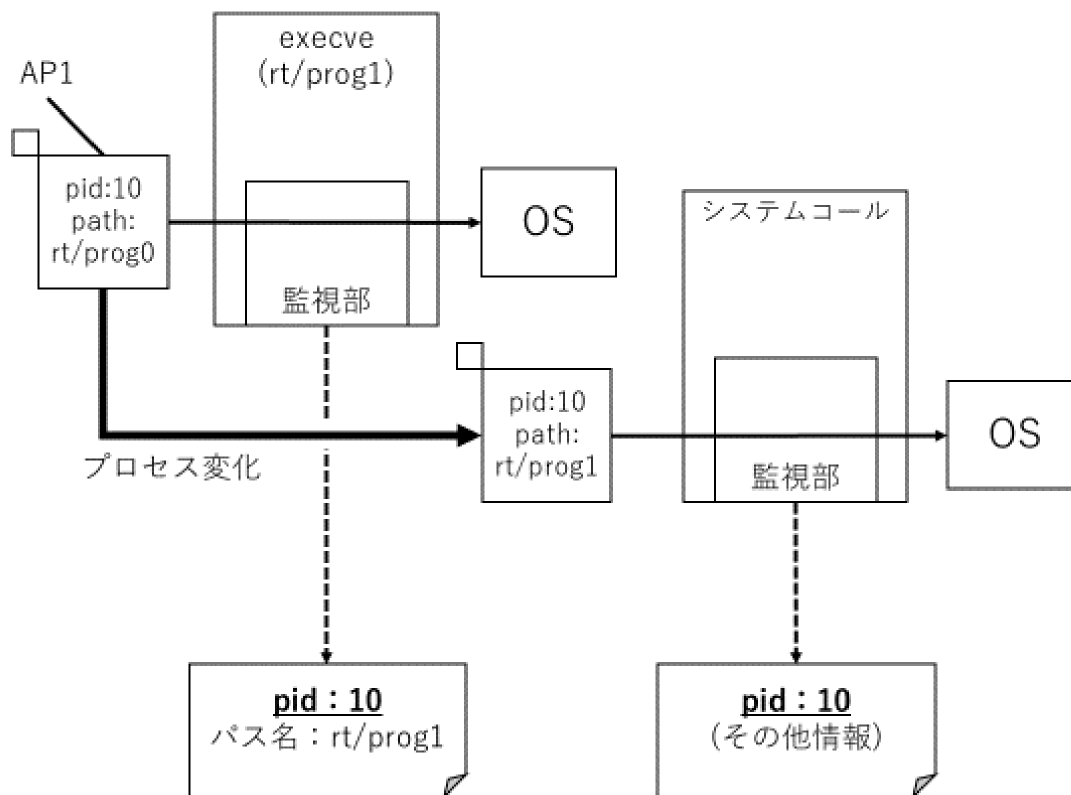


図 4.2　execve により実行された AP プロセス

PID と実行ファイルのパス名の関係を記録しておき，追跡時に，各システムコール・関数の発行時に記録した AP プロセスの PID を基に，実行ファイルのパス名を特定する．具体例を図 4.2 を用いて説明する．図中左上部の AP1 プロセス (pid は 10 とする) は execve システムコールを発行し，パス rt/prog1 という実行ファイルを実行した．このとき execve システムコールの監視部は”pid:10”，”実行ファイルパス名:rt/prog1” という対応関係を記録する．その後 AP1 が他のシステムコールを発行した場合，そのシステムコール監視部が pid を記録する．ここでは”pid:10” という情報を記録している．追跡時にこのようにして記録した情報を突き合わせて，システムコールを実行した AP1 の実行ファイルは，パス”rt/prog1” であると特定する．

ここでは，以下の情報を監視して，CSV ファイルとして記録する．

- execve() を発行した AP プロセスの PID
- execve() によって指定されたファイルのパス名

execve() を発行した AP プロセスの PID は，関数内で getpid() を用いて取得する．execve() によって指定されたファイルのパス名は，execve() の引数である*filename からコピーして取得する．

4.2　追跡フェーズ

追跡フェーズでは，各システムコールの監視によって得られたリストを利用して実行環境の追跡を行う．リスト名は”[システムコール名]_list.csv”として統一している．open/openat はまとめて open_list.csv とし，fork/vfork はまとめて vfork_list.csv としている．

実行環境の追跡は以下の手順で行う．

1. 各リストを結合し，全ての資源情報をまとめた”依存関係リスト”を作成する
2. 依存関係リストに目的の AP 名を与えて，実行環境を構成する AP・ファイル資源を出力する

各手順について以下で説明する．

4.2.1　各リストの結合

各リストの結合の全体図を図 4.3 に示す．リストの拡張子は省略する．

リストは最上段が各システムコールの監視によって得られたリストのオリジナルとなり，最下段が依存関係リストとなる．依存関係リスト名は trace_list としている．各階層ごとにリストを結合していく．

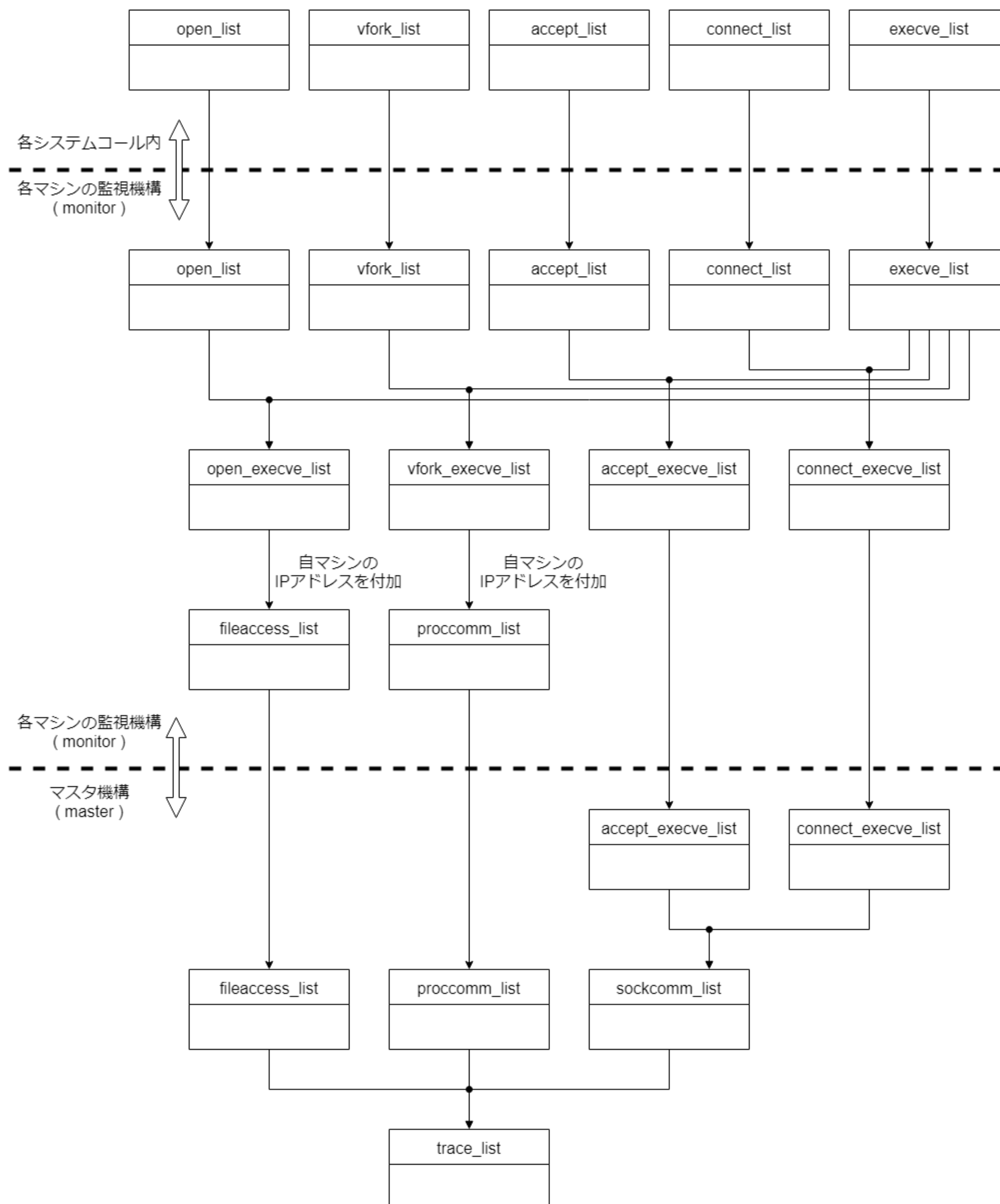


図 4.3　リスト結合の全体図

a) PID とパスの置き換え

前節で説明したように，各システムコール内で記録した AP プロセスの PID を，execve で取得したパス名に置き換える．結合するリストを表 4.3 に示す．

b) IP アドレスの付加

依存関係リストを作成する際に，各計算機のリストを 1 つのマシンに集約する必要がある．そのため，リストを生成した計算機を判別するために，open_list と fork_list に，生成した計算機の IP アドレスを付加する．各リスト名を表 4.4 に示す．

c) ソケット通信を行っている AP の組の決定

accept，connect の監視情報単体では，通信相手の IP アドレスとポート番号は分かっても，通信先の AP が何であるかはわからない．しかし依存関係リストを作成するためには，通信相手の AP 名が必要である．そこで，accept と connect の監視情報を相互に参照して，ソケット通信を行っている AP の組を求める．具体例を，図 4.4 のような環境で監視を行った際の監視・追跡を想定して説明する．図 4.4 のように 2 台の計算機が通信したとき，受信側では accpet_list に，送信側では connect_list に 1 行ずつ監視情報が記録される．ここでは図 4.5 に示すリストが得られる．基本的にポート番号は，枯渇しない限り再利用されないため，本研究では同 IP アドレス内でのポート番号はユニークな値であるとしている．その

表 4.3　execve_list の結合

結合するリスト 1	結合するリスト 2	生成するリスト
open_list	execve_list	open_execve_list
vfork_list	execve_list	vfork_execve_list
accpet_list	execve_list	accept_execve_list
connect_list	execve_list	connect_execve_list

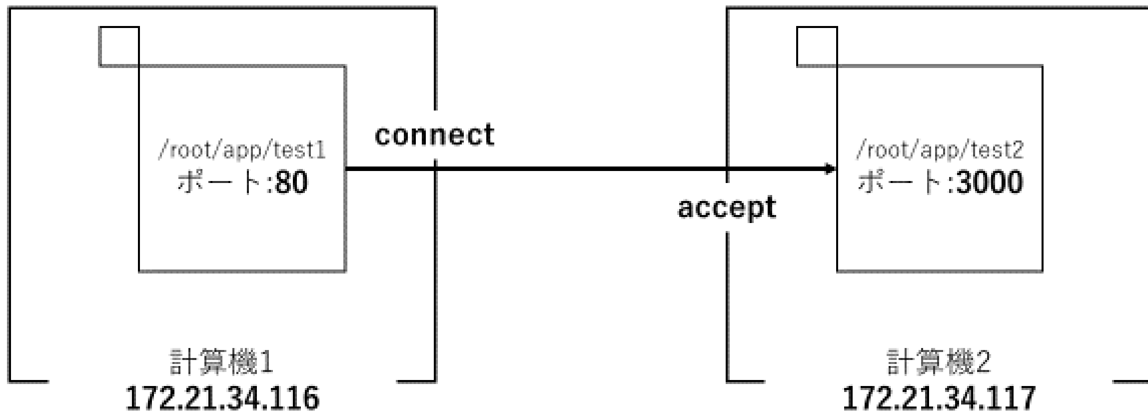
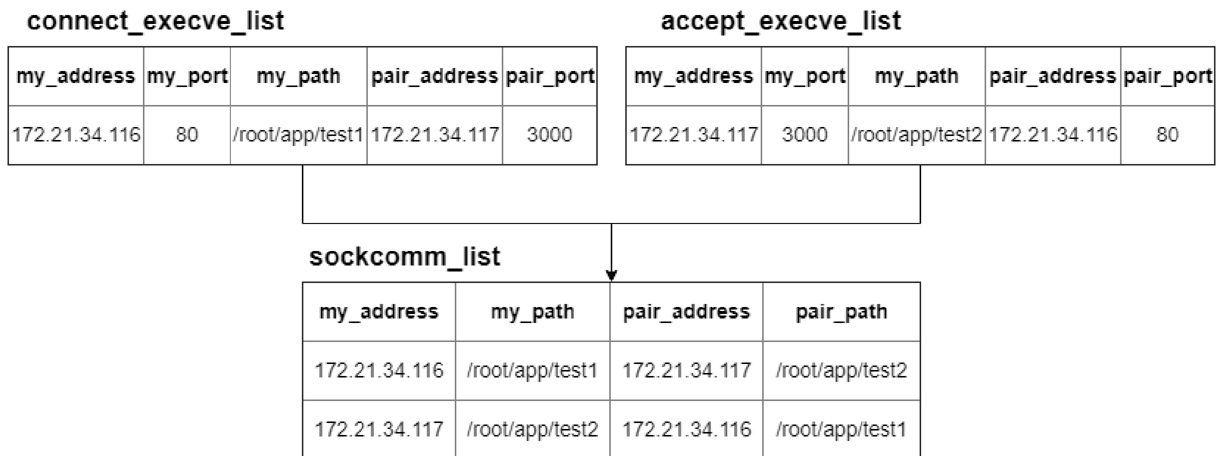


図 4.4　ソケット通信を行う AP の組



connect_execve_list

my_address	my_port	my_path	pair_address	pair_port
172.21.34.116	80	/root/app/test1	172.21.34.117	3000

accept_execve_list

my_address	my_port	my_path	pair_address	pair_port
172.21.34.117	3000	/root/app/test2	172.21.34.116	80

sockcomm_list

my_address	my_path	pair_address	pair_path
172.21.34.116	/root/app/test1	172.21.34.117	/root/app/test2
172.21.34.117	/root/app/test2	172.21.34.116	/root/app/test1

図 4.5　ソケット通信のリスト結合

ため，accept_list と connect_list において IP アドレスとポート番号が一致したものは，同じ AP として扱う．よって，その組を互いに突き合わせて 1 組の通信を特定する．結合したリストは sockcomm_list とし，パスを記述する．

表 4.4　リストへの IP アドレスの付加

元のリスト	生成するリスト
open_execve_list	fileaccess_list
vfork_execve_list	proccomm_list

d) 依存関係リストの作成

a，b，c の結果得られたリストは，全計算機内のファイルアクセスを記録した fileaccess_list，全計算機内の AP プロセスの親子関係を記録した proccomm_list，計算機を超えて通信を行っている AP プロセスの関係を記した sockcomm_list である．これらのリストを結合して依存関係リストである trace_list を作成する．

4.2.2　実行環境を構成する資源の追跡

依存関係リストは，AP やファイルの 1 対 1 の関係をリスト形式で記録したものである．しかし，このままでは，図 3.5 中の AP1 とファイル D のような，間接的な関係を把握することができない．そこで依存関係リストに AP 名を与えることで，直接的な資源だけでなく間接的な資源も含めた，全ての AP・ファイルを出力するようにする．図 4.6 の例を用いて説明する．ここでは依存関係リストに AP1 を与える．まず直接関係のある AP2 と File1 が把握できる．これらは移送対象とする．次に，新たに移送対象とした AP2，File1 と直接関係のある資源を調査する．すると次は File2 が把握できる．ここで新たに把握した File2 は移送対象とする．これを，新たな移送対象が現れなくなるまで繰り返し，移送対象が現れなくなった場合に，移送対象リストの調査を終了する．このように直接関係している資源を追うことにより，AP1 と間接的に関係のある全ての資源を把握することができる．図の例では最終的に AP2，File1，File2 を移送対象とする．

4.3　移送後の通信

監視フェーズで記録した，gethostbyname，getaddrinfo の情報を使用する．移送時の流れを図 4.7 に示す．図のような場合，監視フェーズを経て各計算機から IP アドレスとホスト名の対応関係を記録したリストが得られる．それを統合したものが図中の名前解決リストである．AP を移送時にこの名前解決リストを同時に移送する．このとき，ファイル内の IP アドレスを全て移送先の IP アドレスに置換する。移送後，DNS のゾーンファイルに対応関

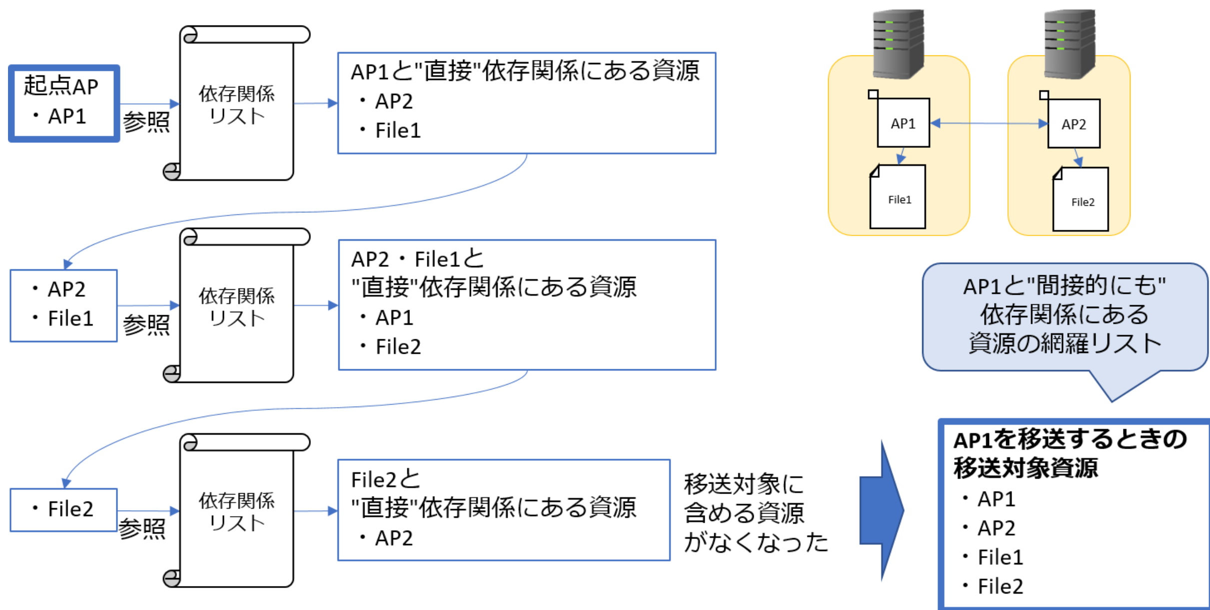


図 4.6　実行環境を構成する資源の追跡

係を記録する．これにより移送後，各計算機のホスト名を解決することが可能となる．

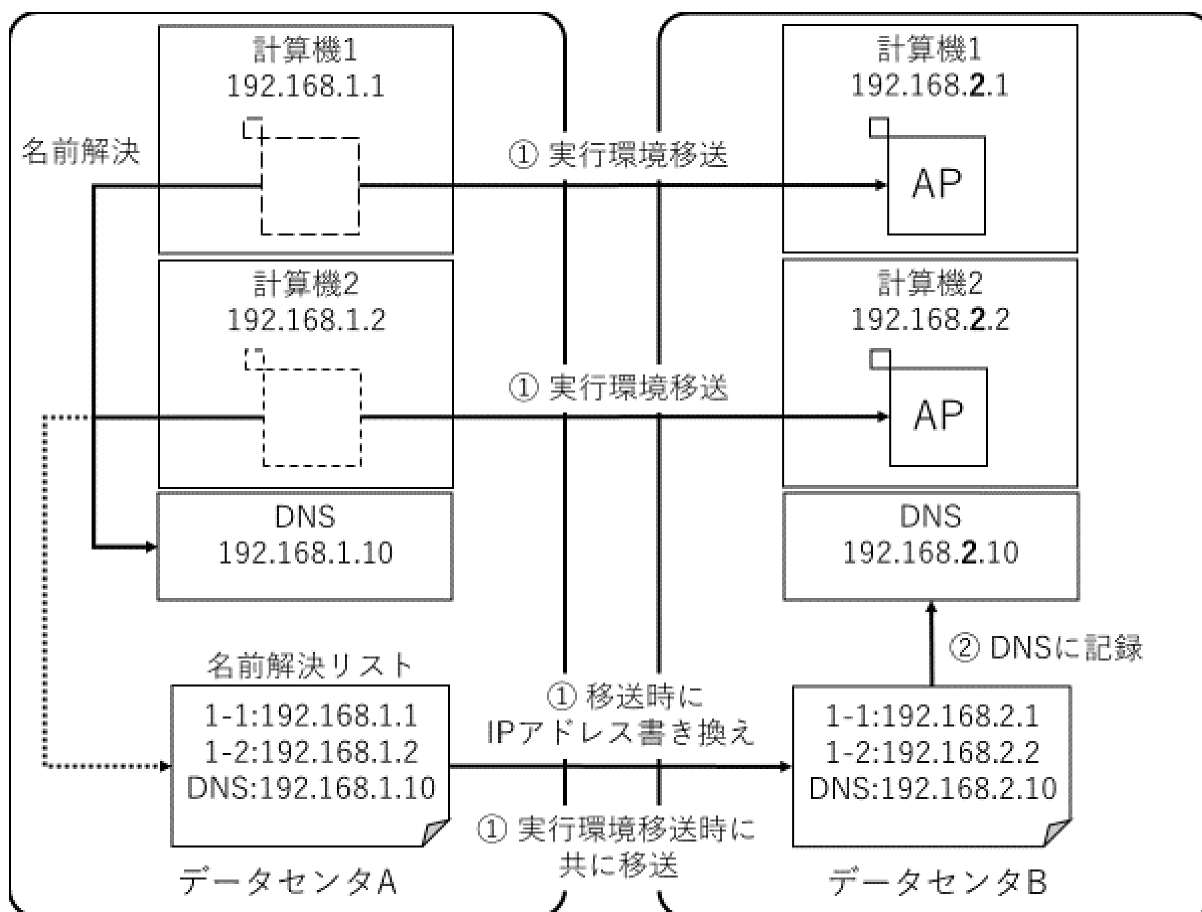


図 4.7　名前解決リストの移送

第 5 章

評価

5.1　監視フェーズ

5.1.1　評価内容

監視機構を実装した際に，その有無によりシステムコールのオーバヘッドがどの程度生まれるのかを確認するため評価した．評価環境は実装環境 (前章の表 4.1) と同じものを利用した．監視機能を実装するにあたり影響を受けるシステムコールは前章で説明したように，open/openat，close，fork/vfork，accept，connect，execve である．これらのうち，open+close，fork，connect，execve の各システムコール実行時間を，監視を行っていない状態と，監視を行っている状態でそれぞれ 5 回ずつ測定した際の平均値を実行時間とした．open() は必ず close() とセットで発行されるので，これらはセットで実行時間を測定した．また，accept システムコールは接続待ち時間が発生することで正確なオーバヘッドの評価が困難であるため，ここでは connect システムコールのみの評価とした．各システムコールの実行時間の測定方法を図 5.1 に示す．現在の時刻をマイクロ秒精度で取得する gettimeofday() を用いて，システムコール処理直前と直後の時刻をそれぞれ取得し，その差をとることで実行時間とする．しかし，execve() はシステムコール実行成功時に別プロセスに変化するため，この方法では実行時間が測定できない．そこで図 5.2 に示すように，測定用 P1 と測定用 P2 の 2 つのプログラムを用意して測定を行う．P1 で execve() により P2 を実行するが，P1 での execve() 実行直前に gettimeofday() を実行し，取得した時刻を execve() で実行する P2 に，引数として与える．P2 は実行直後に gettimeofday() を実行し，

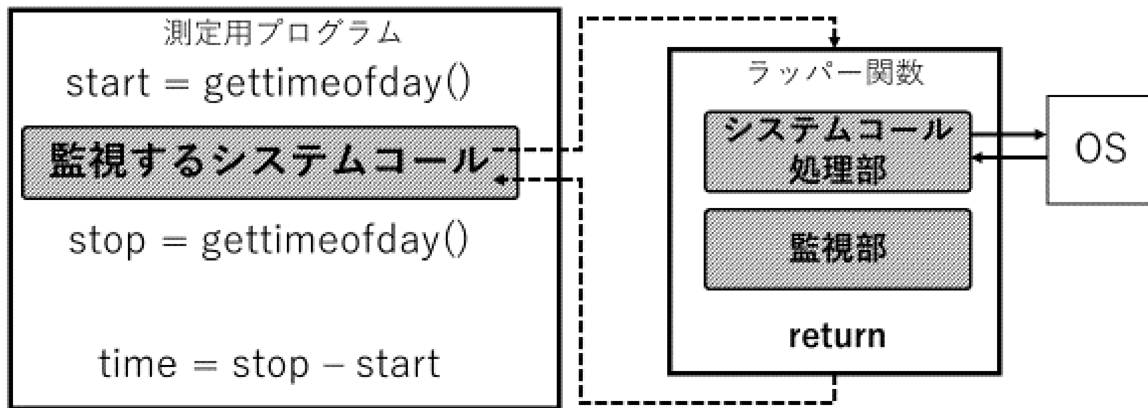


図 5.1 実行時間測定の方法

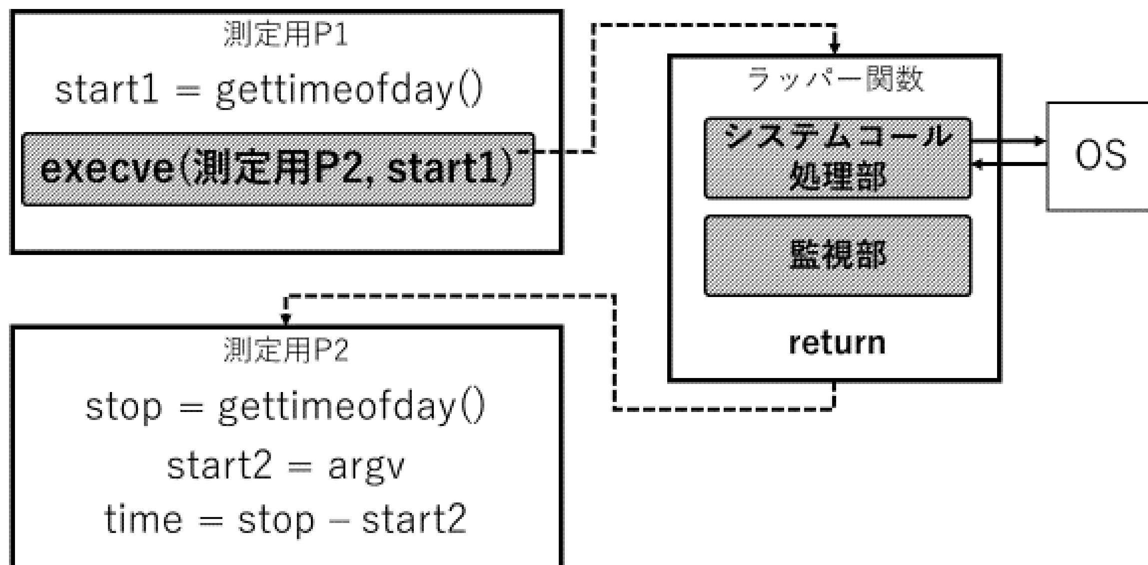


図 5.2 実行時間測定の方法 (execve)

時刻を取得する．最後に P1 から引数として受け継いだ時刻と，P2 で取得した時刻との差をとって，execve() の実行時間とする．

5.1.2 評価結果

各システムコールの実行時間を図 5.3 に示す．図中の白いグラフがシステムコールを監視していないときの実行時間，色付きのグラフがシステムコールを監視しているときの実行

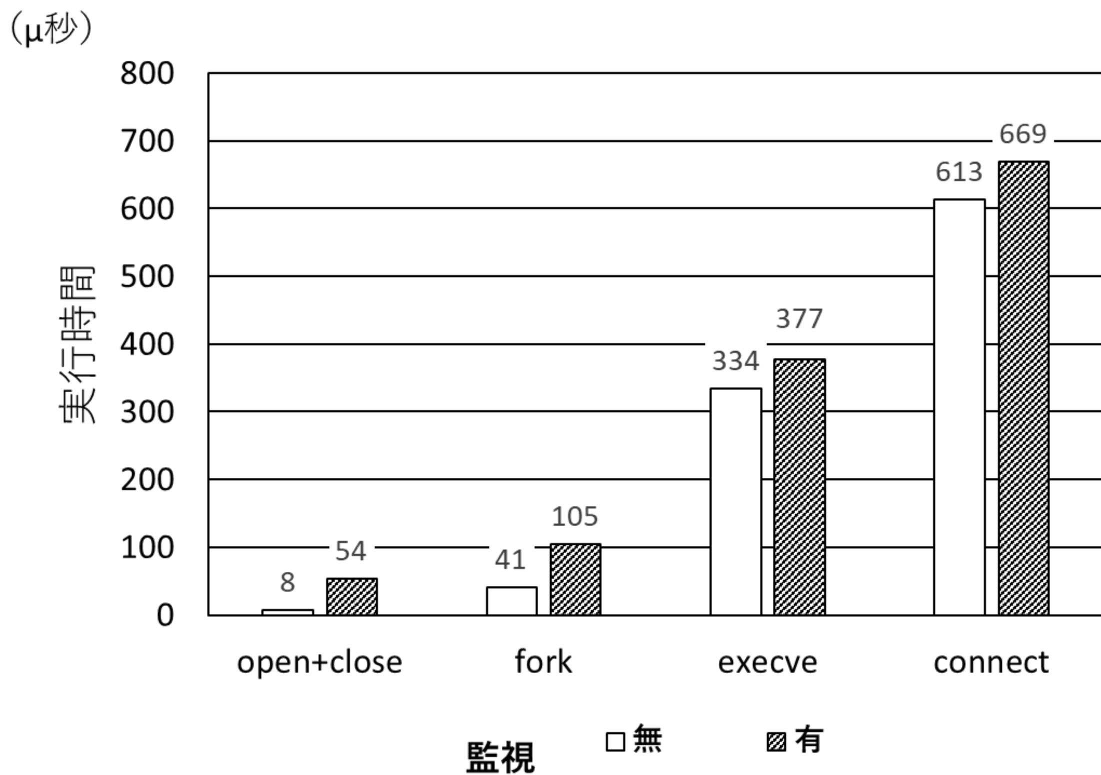


図 5.3　システムコールのオーバヘッド

時間である．グラフ縦軸はシステムコール実行時間である．各グラフの上部にそのシステムコールの実行時間をマイクロ秒単位で記している．すなわち各システムコールのオーバヘッドは，open+close は 46 マイクロ秒，fork は 64 マイクロ秒，execve は 43 マイクロ秒，connect は 56 マイクロ秒である．

5.1.3　考察

各システムコールのオーバヘッドは概ね 40〜60 マイクロ秒程度となり，システムコールによって特にオーバヘッドが増加したということはなかった．そのため，元々の実行時間が比較的短く，発行回数が多い open システムコールと fork システムコールは，オーバヘッドが大きく影響すると考えられる．実際に実行時間は open+close システムコールは約 6.7 倍，fork システムコールは約 2.6 倍の実行時間となった．一方，元々の実行時間が比較的長く，

発行回数の少ない execve システムコールと connect システムコールは，オーバヘッドの影響は小さいと考えられる．

監視部分の各システムコールに共通するとして，記録した情報をファイルに出力する際の処理があるが，今回はシステムコール実行の度にユーザレベルでファイルに出力しているため，二次記憶装置へのアクセスがオーバヘッドの大きな要因となっていると考えられる．よって対策として，メモリにデータをキャッシュし，二次記憶装置へのアクセスを減らすことを検討できる．

5.2　追跡フェーズ

5.2.1　評価内容

実行環境の追跡処理の評価を行った．

今回追跡処理を行う AP 実行環境の対象として，実際の銀行のオンラインシステムにおけるバッチ処理を想定した．想定する環境のパラメータの参考として文献 [12] で用いられているものを利用した．文献 [12] は，銀行のオンラインシステムにおけるオンライン処理とバッチ処理の負荷分散を目的とした手法を提案・実装しているが，その評価のため，同環境を FreeBSD に実装している．本研究ではそのうちバッチ処理に着目し，文献 [12] における実装パラメータを用いた際のファイルへのアクセスと AP の通信を監視したものを，依存関係リストとして出力した．このリストに対し追跡処理を行ったものを本研究での評価とした．

実行環境の動作は次の通りである．

1. AP とファイルを一定数用意する．
2. 各 AP が図 5.4 のように順にファイルに 1 度アクセスする
   この動作により，各ファイルはどれかの AP から一度アクセスされたことになる．
3. 各 AP は図 5.5 のように 2 でアクセスしたファイル以外のファイルの一定数にアクセスする
   この動作により，全体のファイル数に対して一定の割合のファイルは 2 種類以上の AP

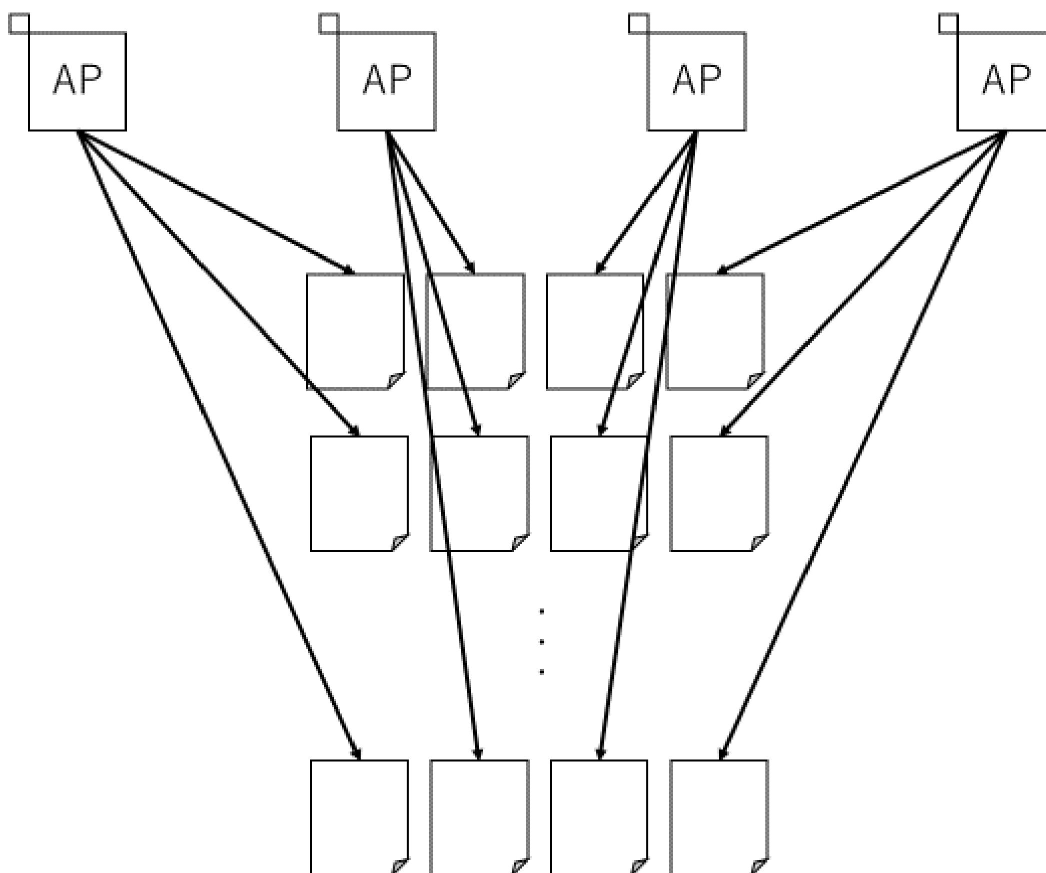


図 5.4　ファイルアクセス

からアクセスされたことになる．そのようなファイルを共有ファイル，全体のファイル数に対する共有ファイルの割合を共有率と呼ぶ.

4. 共有ファイルにランダムな AP がランダムに一定回数アクセスする

   共有ファイルに対してランダムな AP が複数回アクセスすることにより，1 ファイルを共有する AP の種類数を増加させる.

評価における AP 実行環境のアクセスパラメータを表 5.1 に示す.

表 5.1 に示すようにパラメータを変え，それぞれの場合において依存関係リストを作成した．これらのリストに対しそれぞれ追跡処理を行った際の追跡時間を評価した.

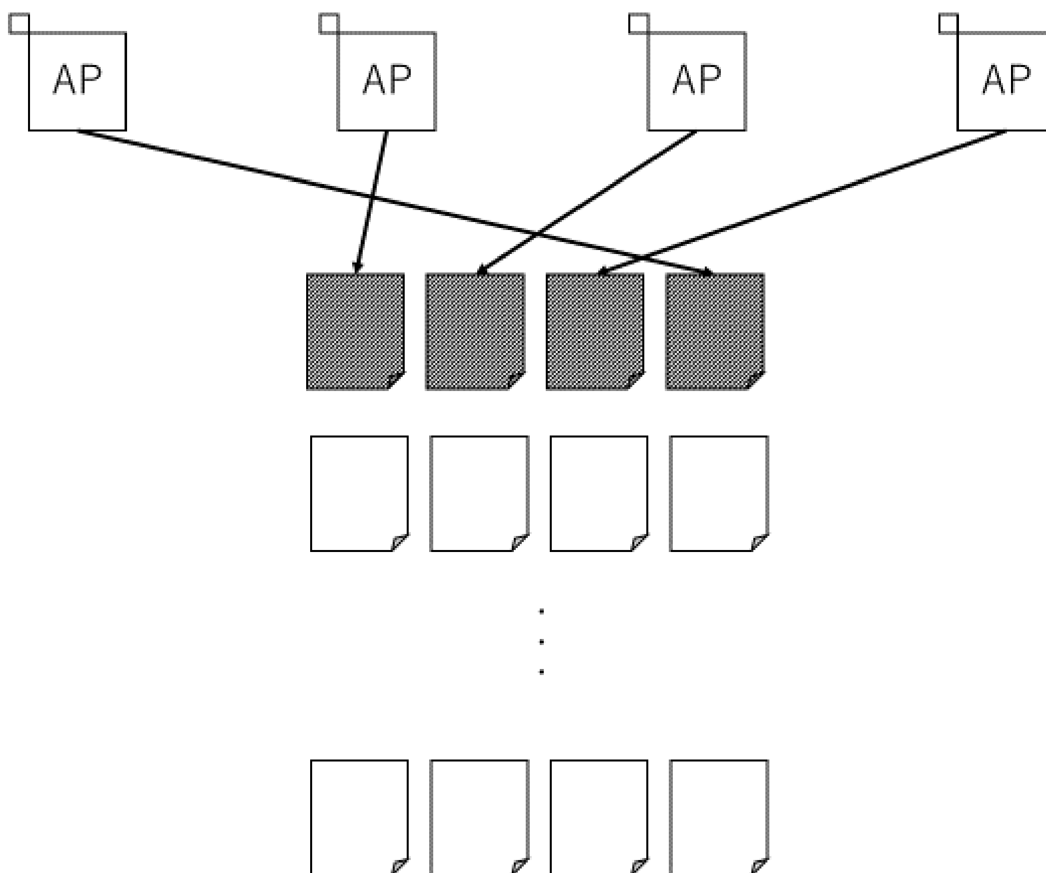


図 5.5　共有ファイルにアクセス

5.2.2　評価結果

ファイル共有率別の追跡時間の評価結果を図 5.6 に示す．グラフの縦軸が追跡時間 (単位: 秒)，グラフの横軸が AP 数である．また，グラフ右側の数値は AP 数が 3000 の場合の追跡時間を秒単位で示している．

表 5.1　追跡対象 AP パラメータ

AP 数	10，20，50，100，200，500，1000，2000，3000
ファイル数	AP 数の 100 倍
共有率 (%)	20，40，60，80，100
共有ファイルへのランダムアクセス回数	3000

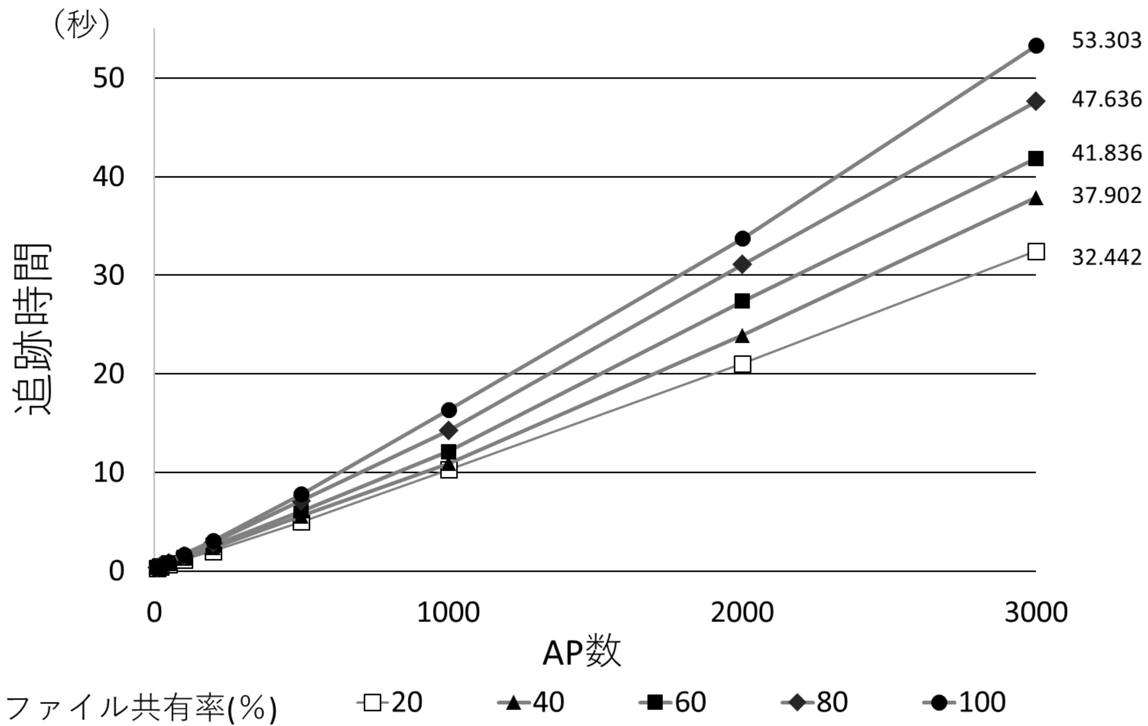


図 5.6 実行環境の追跡時間

5.2.3 考察

評価結果は，AP 数に線形比例して追跡時間が増加するものとなった．また，ファイル共有率が増加すると，追跡時間が少し増加している．例えば，AP 数が 3000 の場合，共有率の違いにより，追跡時間に最大約 21 秒の差が生まれた．このことから，追跡するファイル数が増加すると追跡時間も増加するということがわかる．このような結果から，例えば，多数の AP が少数のログファイルに対して書き込みを行うような形態の AP 実行環境に対しては，提案手法は非常に有効に働くのではないかと考えられる．一方で，少数の AP が多数のファイルに対してアクセスを行うような AP 実行環境の場合，追跡する AP が 1 つ増加する度に，特定するファイル数が大きく増加することが考えられ，追跡時間の増加に繋がるのではないかと考えられる．

文献 [12] では，実際の銀行オンラインシステムにおけるバッチ処理は，最大でも同時に実行される AP 数は 800 程度であるとされている．本手法では，AP 数が 800 の場合，追跡

時間は最大でも 15 秒程度となった．文献 [12] によると，バッチ処理の場合 1 ファイルあたり 1 度に書き込まれるデータ量は 1〜100MB 程度である．よって AP 数が 800 で共有率が 100%の場合，ファイル数は 80000 となり，最小でも 80GB 程度のデータ量となる．このとき 100Mbps のネットワーク環境を想定すると，移送に約 107 分を要するため，15 秒での実行環境の追跡は移送時間と比較すると十分に実用的な時間であると考えられる．以上から，提案手法を用いることで，銀行のオンラインシステムのような大規模な AP 実行環境においても，十分に実用的な時間で実行環境の追跡を行うことができるといえる．

以上の追跡時間は，サービスの開始後，移送の時点で追跡を行う場合に，特に有効であるといえる．しかし，動作中の AP のうち，通信時間を追跡時間以上に要するものについては，その時間に依存してしまう可能性があるため，その有効性が十分に発揮されないと考えられる．また，サービス開始前に十分追跡できる時間がある場合は，その有効性が十分に発揮されないと考えられる．

第 6 章

おわりに

クラウド環境は，地理的に離れた複数の計算機に資源を分散して配置した，広域分散システムとして実現されることが多く，負荷分散やバックアップのため，AP 実行環境を他計算機に移送させることが頻繁に発生する．そのような理由から効率的な移送手法が求められている．代表的な手法である仮想マシンの移送は，移送する必要のないデータも移送してしまい移送に時間がかかる．今日ではネットワーク技術は日々発展しており，今後，様々な新技術の発展と共に，通信技術も向上すると考えられる．しかし，同時に扱うデータのサイズや量も膨大なものとなり，依然として効率的な移送手法は需要があると考えられる．本研究で我々は，ファイルの資源利用情報を基に，ソフトウェア実行環境を特定する手法を提案した．提案手法では，AP のファイルアクセス情報を基に，複数計算機上に跨る AP 実行環境の特定・追跡を行った．これにより，実行環境に必要最小限の資源をファイル単位で特定することができた．また，提案手法を実装し，銀行のオンラインシステムにおけるバッチ処理を模したファイルアクセス情報に対して追跡を行うことにより，提案手法の性能を評価した．その結果，実用的な時間内で AP 実行環境の追跡を行えることがわかった．

本研究では，移送後の異なるネットワークに対応するため，最も単純な DNS を用いた．しかし近年では，ソフトウェアレベルで非常に高機能なルーティングやネットワーク制御を可能とする Software-Defined-Network(SDN) が用いられることが多い．こうしたネットワークではコントローラと呼ばれる管理機能でネットワークを制御している．そのため名前解決だけで完全に対応できるとは限らない．そこで今後の課題として，一般的なネットワークだけでなくこうしたネットワークに対して移送した際に，低コストで通信を復帰できるような手法の確立が必要であると考える．

謝辞

本研究，本稿，および本研究内容梗概の作成にあたり，研究の提案，進行，発表指導および発表主査，論文の添削や作成指導を，担当指導教員である高知工科大学情報学群の横山和俊教授にして頂きました．横山和俊教授には深く感謝申し上げます．また，本研究発表の副査を同学群の敷田幹文教授，植田和憲講師にして頂きました．先生方にはお時間忙しいところ貴重なご意見や，ご指導を頂きました．同じく感謝申し上げます．さらに研究の参考として論文や研究成果をご参考にさせていただいた同大学大学院，横山研究室卒業生の畑翔太さん，共に研究に取り組んでいただいた同大学同研究室卒業生の澤田優真さん，西拓人さん，同大学同研究室の有薗里奈さん，また，協力をいただきました同研究室同期生の福永昂輝さんをはじめ，同研究室の方々に深くお礼申し上げます．

参考文献

[1] 中井新太郎, 川島龍太, 斎藤彰一, 松尾啓志, “更新履歴に基づいたメモリページ転送順序スケジューリングによる仮想マシンライブマイグレーションの高速化”, 情報処理学会論文誌, Vol.56, No.2, pp.516-524 (2015).

[2] Open Networking Foundation, “Open Networking Foundation is an operator led consortium leveraging SDN, NFV and Cloud technologies to transform operator networks and business models”, ⟨`https://www.opennetworking.org/`⟩ (参照 2020/01/21)

[3] 藤本大地, 近堂徹, 前田香織, 大石恭弘, 相沢玲二, “OpenFlow による移動透過な広域ライブマイグレーションシステムの提案と実装”, インターネットと運用技術シンポジウム 2015 論文集, pp.36-43 (2015).

[4] 谷村直哉, 横山和俊, “資源の利用状況に着目したプログラム実行環境の移送方法の選択方式”, 高知工科大学 平成 26 年度学士学位論文 (2015).

[5] 大西史洋, 黒木勇作, 横山和俊, 谷口秀夫, “プログラム実行環境移送のための資源追跡機能のユーザレベルでの実現”, 情報処理学会第 81 回全国大会, 第 3 分冊, pp.319-320 (2018).

[6] 黒木勇作, 西拓人, 横山和俊, 谷口秀夫, “複数計算機上に跨るプログラム実行環境の特定手法の提案”, 情報処理学会第 81 回全国大会, 第 3 分冊, pp.265-266 (2019).

[7] 畑翔太, 横山和俊, “ファイルの更新期待値を用いたソフトウェア実行環境のプリコピーマイグレーション方式”, 高知工科大学 平成 28 年度修士学位論文 (2017).

[8] 黒木勇作, 横山和俊, “サービスの停止時間を短縮するソフトウェア実行環境のプリコピー移送方式”, 高知工科大学 平成 29 年度学士学位論文 (2018).

[9] IEEE and The Open Group, “vfork”, ⟨`https://pubs.opengroup.org/onlinepubs/009695399/functions/vfork.`

html⟩ (参照 2020/01/21)

[10] IEEE and The Open Group, “open”,
⟨https://pubs.opengroup.org/onlinepubs/009695399/functions/open.html⟩ (参照 2020/01/21)

[11] The Apache Software Foundation. “prefork - Apache HTTP Server Version 2.4”,
⟨https://httpd.apache.org/docs/2.4/en/mod/prefork.html⟩
(参照 2020/01/22)

[12] 田辺雅則, 横山和俊, 長尾尚, 谷口秀夫, “オンライン処理とバッチ処理の処理負荷を分散制御する入出力制御方式の実装と評価”, 情処論文誌, vol.61, No.2, pp275-276 (2020).