ソケット通信に着目した計算機にまたがるソフトウェア実行環境の特定方式

平成 30 年度

学士学位論文

ソケット通信に着目した計算機にまたがるソフトウェア実行環境の特定方式

Identification method of software execution environment spanning computers using monitoring of socket communication

1190352　　西 拓人

指導教員　　横山 和俊

2019 年 2 月 28 日

高知工科大学 情報学群

要 旨

ソケット通信に着目した計算機にまたがるソフトウェア実行環境の特定方式

西 拓人

クラウドサービスは，広域分散システムで実現されることが多い．このようなシステムでは，応用プログラム (以降，AP と略す) や実行環境を他のデータセンタに移動 (マイグレーション) することで負荷を軽減することがある．従来では，このマイグレーションの際に AP の動作に障害が起こらないよう，仮想マシンイメージをそのまま移送する手法がとられていた．しかしこの手法では，移送する必要のないデータも移送することになり，非効率である．これを解消する手法として，AP が利用するファイル資源を特定し，AP が動作するために必要な最小限のファイル資源だけを移送するものがある．この手法の先行研究では，open システムコールの発行を監視し，AP のファイルへのアクセス状況を特定する手法が提案されている．しかし，ファイルアクセスを監視するだけでは，AP が動作するために必要なすべてのファイル資源を特定することができない．そこで本研究では，AP のファイルアクセスに加え，AP 同士の依存関係も特定できる手法を提案する．AP 同士の依存関係を追うために新たに監視対象としたのは，プロセスの親子関係とネットワーク通信である．これにより，計算機にまたがるソフトウェア実行環境の特定を実現した．ネットワーク通信監視のオーバーヘッドを計測した結果，単純なソケット通信のプログラムでも約 2%となった．また，監視情報からソフトウェア実行環境を求める機能の処理時間を評価した結果，実用的な時間で処理できることを確認した．

キーワード　オペレーティングシステム，クラウドコンピューティング，マイグレーション

Abstract

Identification method of software execution environment spanning computers using monitoring of socket communication

The cloud computing environment is realized as a wide area distributed system. In order to efficiently use these environments, it is frequent to migrate the application program and its execution environment to another computer. A typical method of migration is to migrate the virtual machine. In this method, it is necessary to migrate all the file in the virtual machine. Therefore, we propose a migration method of software execution environment based on access information to the file. However, with the monitoring of file access alone, it is not possible to identify all the resources required by the application program.

In this research, we propose a method to specify dependency between application programs. Specifically, we focus on not only file access but also parent-child relationship of process and network communication, and propose a method of identifying dependency software execution environment across multiple computers.

key words Operating system, Cloud computing, Migration

目次

図目次

表目次

第 1 章

はじめに

クラウドサービスは，広域分散システムを構築し実現されることが多い．これは，地理的に離れた複数のデータセンタ間を Wide Area Network(WAN) で接続し，分散処理を行うシステムである．このようなシステムでは，特定のデータセンタに負荷が集中した場合や，サービスのバックアップを取りたい場合などには，AP とその実行環境を他のデータセンタに移送することで負荷を軽減する必要がある．そのため，ソフトウェア実行環境を別計算機に効率よく移送するための技術が求められている．従来では，この移送の際に，仮想マシンイメージをそのまま移送する手法がとられていた (図 1.1)．この手法を研究している文献 [1] では，ライブマイグレーションの際に多くのメモリページを転送することになる問題に対し，移送の両ホストに共通して存在するメモリページの転送を行わないことで移送量を削減する手法を提案している．また，文献 [2] では，移送時に仮想マシンを停止させなければならなず，それがサービス利用者に不快感を与え，利用者減少に繋がることに対し，仮想マシンサーバと利用者のクライアントマシンとのコネクションを，アプリケーション層プロトコルのメッセージを監視することで，利用者に仮想マシンのダウンタイムを感知されにくくする手法を提案している．しかしこの手法では，AP が動作するために必要なファイル資源だけでなく，OS のデータなどの移送する必要のないデータも移送することになる．その結果，データセンタ間の低速なネットワークを長時間占有してしまうことになる．さらに，移送時間が長いということは，サービスの停止時間も長くなることになり，サービス利用者に不利益をもたらす．これを解消する新たな手法として，AP が利用するファイル資源を特定し，AP が動作するために必要な最小限のファイル資源のみを移送するものがある (図 1.2)．このとき，ファイル資源へのアクセス情報を基に，移送するファイル資源の順序を適切に定

めることで，移送時の仮想マシンの停止時間を短縮することができる．

先行研究 [3] では，上記の移送方式において，open システムコールを監視し，AP のファイルへのアクセス状況を特定する手法が提案されている．また，ファイルへのアクセス種別 (Read Only か，Read Write か) を考慮し，必要最低限のファイルのみを移送対象とすることを実現している．しかし，この機能では AP とファイルの依存関係しか追うことができず，AP 同士の依存関係，たとえば，他の計算機に存在する AP とネットワークを介して通信した場合，その関係性を追うことができない．そこで本研究では，AP のファイルアクセスに加え，AP 同士の依存関係も特定できる手法を提案する．AP 同士の依存関係を追うために新たに監視対象としたのは，プロセスの親子関係とネットワーク通信である．これにより，計算機にまたがるソフトウェア実行環境の特定を実現する．評価としては，AP 実行時の監視機構のオーバーヘッド，監視情報から移送対象となるソフトウェア資源を特定する機構の処理時間を計測する．

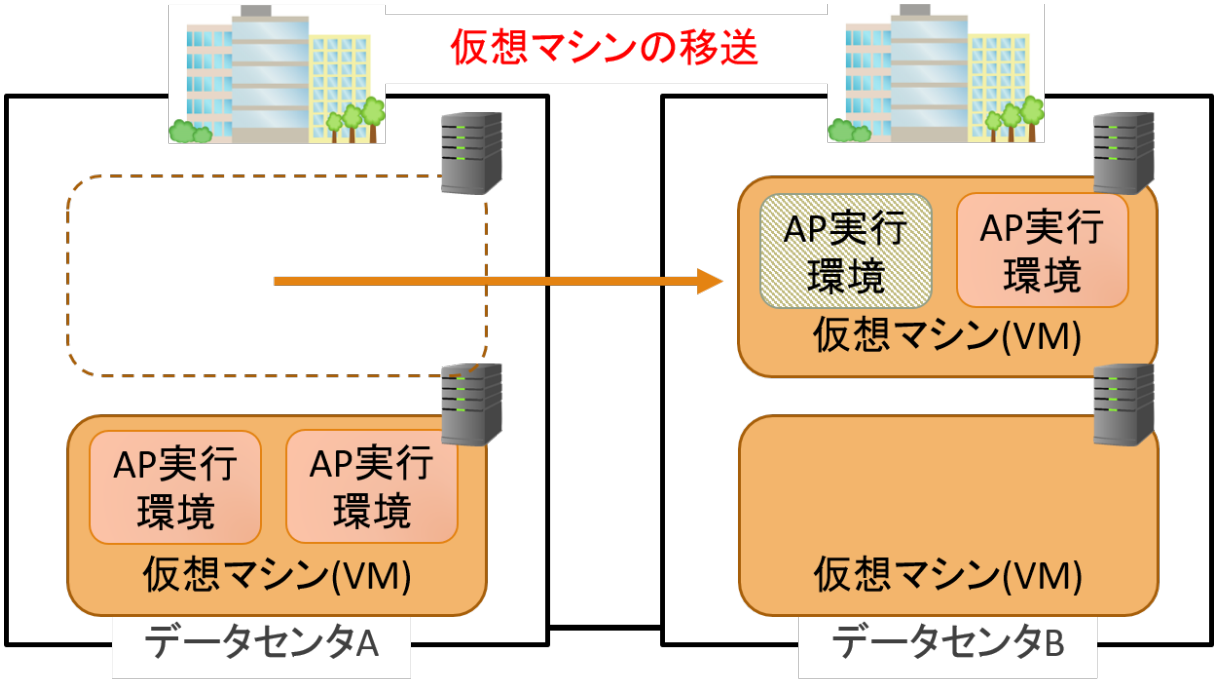

図 1.1　仮想マシンイメージの移送

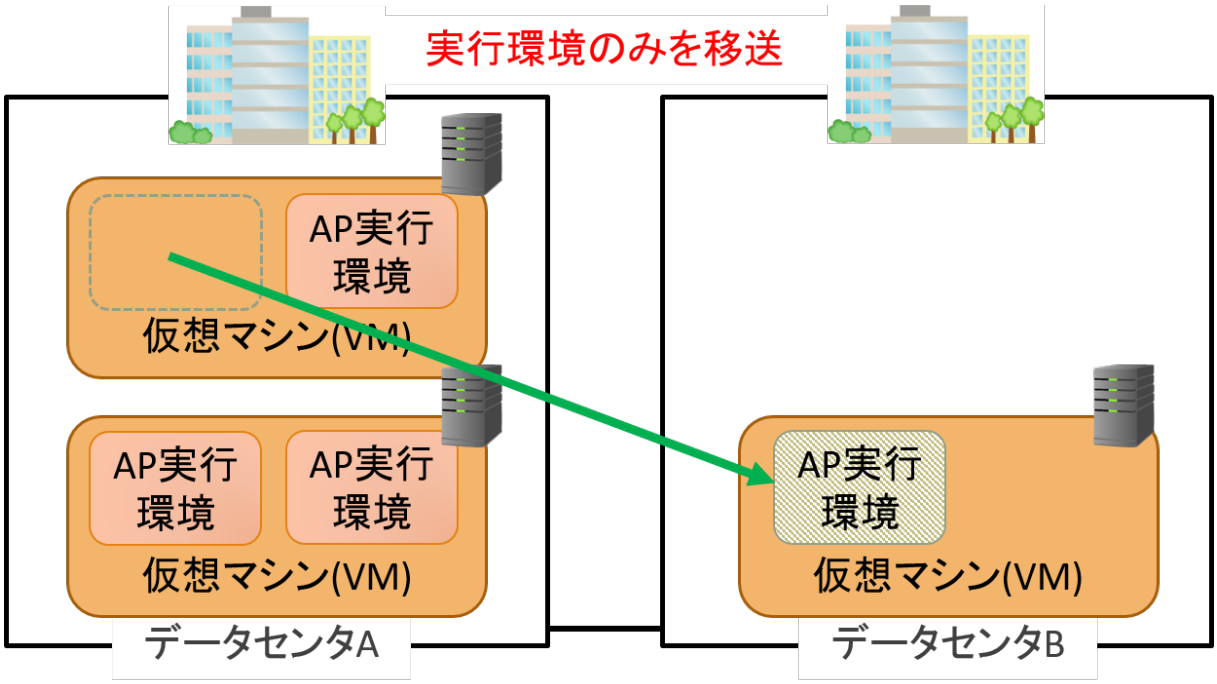

図 1.2　ソフトウェア実行環境の移送

第 2 章

先行研究

第 1 章で述べた通り，広域分散システムにおいてソフトウェア実行環境を移送するための技術が重要視されている．このような技術の代表的な手段として，仮想マシンイメージすべてを移送するものがある．この手法の問題点として，AP が動作するために必要なファイル資源以外のデータも移送してしまうことが挙げられる．データセンタ間を繋ぐネットワークは低速な WAN 環境であるため，仮想マシンイメージのようなサイズの大きいデータを移送すると，ネットワークを独占してしまい，広域分散システム全体に影響を及ぼす．また，移送時間が長いということは，仮想マシンの停止時間も長くなり，サービス利用者に不利益をもたらす．

これらの問題を解決するために，AP が利用するファイルの共有関係に着目して，AP を実行するために必要な最小限のファイル資源だけを移送する手法を提案している研究がある [3]．この手法のソフトウェア実行環境の特定方式は，監視機能と追跡機能で構成されている．監視機能では，open システムコール内部で扱われている情報をカーネル内で取得し，AP が利用しているファイルを把握する．監視機能の概要を図 2.1 に示す．AP が実行した open システムコールは，システムコールラッパーに渡され，システムコールが発行される．監視機構は，open されているファイルの情報等を，カーネル内での open システムコール処理中に取得し保存する．追跡機能では，監視機能で得られた監視情報を基に，移送したい AP と一緒に移送しなければならないファイル資源群を特定する．たとえば，あるファイルに AP1 が Read Write でアクセスしていた場合，この 2 つのファイル資源を一緒に移送すれば，AP1 は移送後も問題なく動作する．また，AP2 があり，これもファイルに Read Write でアクセスしていた場合，AP2 も一緒に移送しなければ，移送後の AP1 は AP2 に

よるファイルの変更を受けることができず，不具合を起こすため，3 つすべてのファイル資源を一緒に移送しなければならない．このように，監視機能と追跡機能によって，ファイル資源の関係性を求め，ソフトウェア実行環境を特定する．

この先行研究の問題点として，カーネルのメモリ消費と追跡機能の計算量が挙げられる．監視機構が open システムコール内から情報を取得した際，その情報はカーネルのメモリ内に蓄えられる．ファイルアクセスの情報は監視時間が経過するにつれて膨大になっていくため，カーネルが使用すべきメモリを大量に消費してしまう．また，追跡機能において，移送対象となるソフトウェア資源を特定するために監視情報に線形探索をかけるような処理をしている．新しく AP が移送対象となるたびに線形探索を行うため，追跡機能の計算量が $O(n^2)$ となっている問題がある．

これらの問題を解決した先行研究がある [4]．この先行研究の監視機能の概要を図 2.2 に示す．この先行研究の監視機能では，AP が発行した open システムコールが，カーネル内に処理が移る前にシステムコールラッパーを介していることに着目し，監視機構をこのシステムコールラッパーに実装している．このようにして，カーネルのメモリを圧迫する問題を，監視機構をユーザレベルで実装することで解消した．また，追跡アルゴリズムを見直し，計算量を $O(nlogn)$ まで改善した．

ここまでの研究で，AP のファイルへのアクセスを監視してファイルの共有関係を特定することはできたが，ファイルアクセスの監視だけでは，ソフトウェア実行環境を完全に特定することはできない．たとえば，移送対象の AP が子プロセスを生成し，その子プロセスに別の AP を展開し実行していたなら，展開された AP も移送対象としなければならないが，open 関数の監視ではこれを追うことができない．また，AP が他の計算機の AP とネットワークを介して通信していた場合，これらの AP は互いに依存関係を持つため，どちらかを移送したい場合には，もう一方も考慮に入れなければならない．

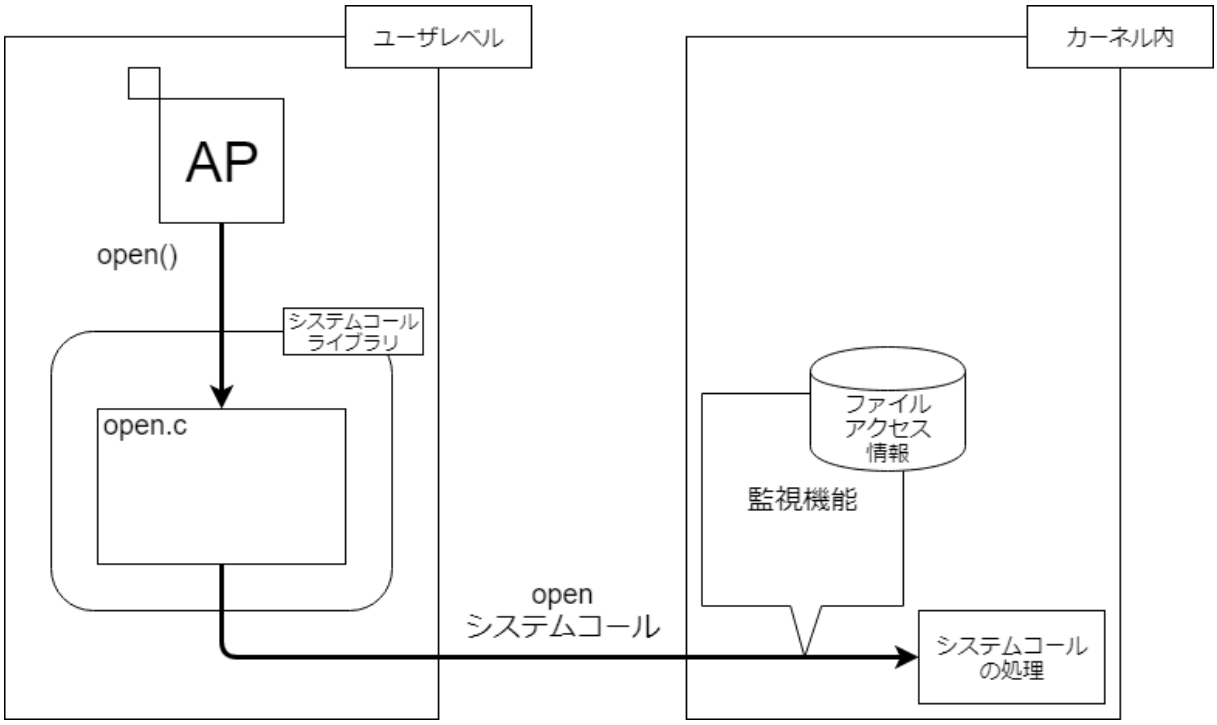

図 2.1　文献 [3] の特定方式の監視機能の概要図

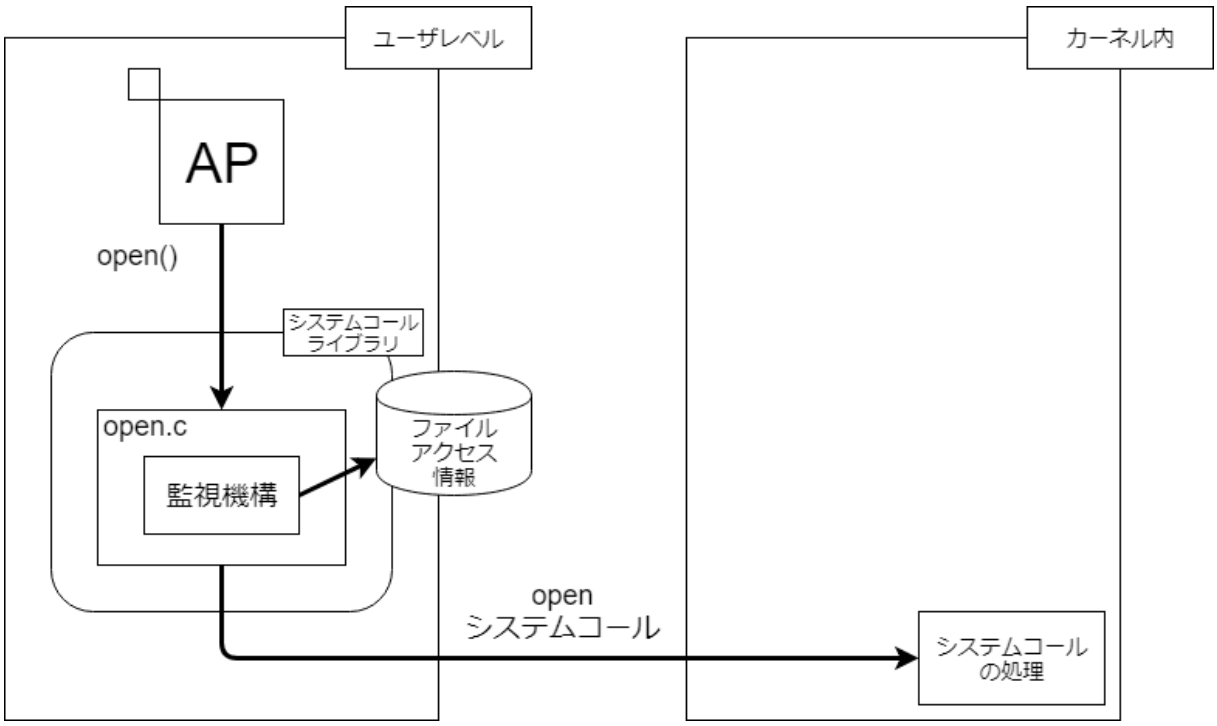

図 2.2　文献 [4] の特定方式の監視機能の概要図

第 3 章

提案手法

本研究で提案するソフトウェア実行環境の移送方式について説明する．この提案方式は，特定ステップと転送ステップの 2 つのステップから構成されている．特定ステップは，移送対象となるファイル資源を求めるステップである．ファイルアクセスする際や，他の AP とネットワーク通信する際に発行されるシステムコールを監視し，その監視情報を基に，移送したい AP と依存関係にあるすべてのファイル資源を洗い出す．転送ステップは，特定ステップで求めた移送対象のファイル資源群を目的の計算機に転送するステップである．このとき，すべてのファイル資源を一度に転送するのではなく，いくつかに分割して転送する．この転送処理は，AP を動作させながら行う「プリコピーフェーズ」と，AP を停止させてから行う「最終コピーフェーズ」に分かれている [5]．プリコピーフェーズで適切なファイル資源を転送することで，最終コピーフェーズで転送するファイル資源を可能な限り減らし，サービスの停止時間の短縮を図ることができる．

本研究では特定ステップに焦点を絞り，計算機にまたがるソフトウェア実行環境を特定する手法を提案する．提案手法の特定ステップは，監視フェーズと追跡フェーズの 2 つの手順で実現される．各フェーズについて説明を行う．

3.1　監視フェーズ

監視フェーズでは，AP によって発行されたシステムコールを監視機構が監視し，AP のファイル資源利用情報を取得する．監視フェーズで行う処理の概要を図 3.1 に示す．監視対象としたシステムコールを表 3.1 に示す．本研究では，先行研究で実現したファイルアク

セスに加え，AP 同士の関係性をも追う必要がある．これを，プロセスの親子関係とネットワーク通信を監視することで実現する．

ファイルアクセスは，プログラムがファイルを開くときに，開くファイルのパスを open() で指定する．したがって，open システムコールの処理内部からパスを取得する．プロセスの親子関係は fork システムコールを監視して，親プロセスと子プロセスの識別情報を組で取得する．ネットワーク通信はソケット通信から監視できる．したがって，ソケット通信の確立を行う accept システムコール，connect システムコールの処理内部から，通信の情報を取得する．上記 4 システムコールは，AP の識別情報を PID で取得している．PID はプログラム実行ごとに変わる値であるので，これを AP の識別情報として追跡に使うことはできない．そこで，exec システムコールを監視して，PID と実行されている AP のパスの対応表を得る．

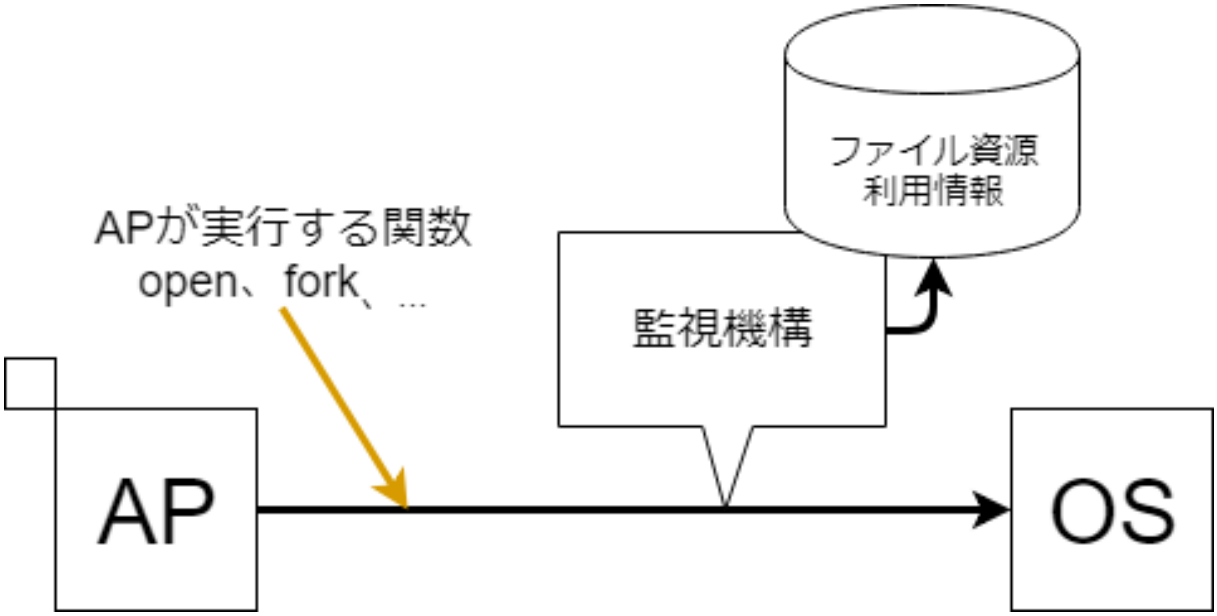

図 3.1　監視フェーズの処理の概要図

表 3.1　監視に利用した関数

関係性	システムコール名
ファイルアクセス	open
プロセスの親子関係	fork
ネットワーク通信	accept connect
PID とパスの解決	exec

3.2　追跡フェーズ

追跡フェーズでは，監視フェーズで得られた情報から，ある AP を移送したいときに一緒に移送しなければならないファイル資源を特定する．この処理は大きく 2 つに分けられる．追跡フェーズの処理の概要を図 3.2 に示す．

はじめに，すべての監視情報を集約して，監視情報の網羅リストを作る．このリストは，LAN 内のすべての計算機で得られた各監視情報をひとつにまとめたものである．ファイルアクセスやネットワーク通信などの，関係性の種類にかかわらず，すべての監視情報を「ファイル資源同士の 1 対 1 の依存関係」として扱い，これをリストに網羅する．このリストを依存関係リストとする．依存関係リストに格納される監視情報は，LAN 内に存在する複数台のマシンで得られたものである．したがって，ファイル資源は IP アドレスと絶対パスの 2 つの識別情報ではじめて一意に識別することができ，依存関係リスト作成にはこれらの情報が必要である．そのために各監視情報は，IP アドレスとパスでファイル資源の情報を記録していなければならないが，一部の監視情報はこのどちらかが欠けているので，これを補完する処理を行う．

次に，作成された依存関係リストから，移送対象となるファイル資源群を特定する．依存関係リストは，ファイル資源同士のすべての依存関係を網羅したリストである．したがって，このリストに任意の AP を参照すれば，任意の AP と関係性を持っているすべてのファイル

資源を求めることができる．このような特定処理を行う．

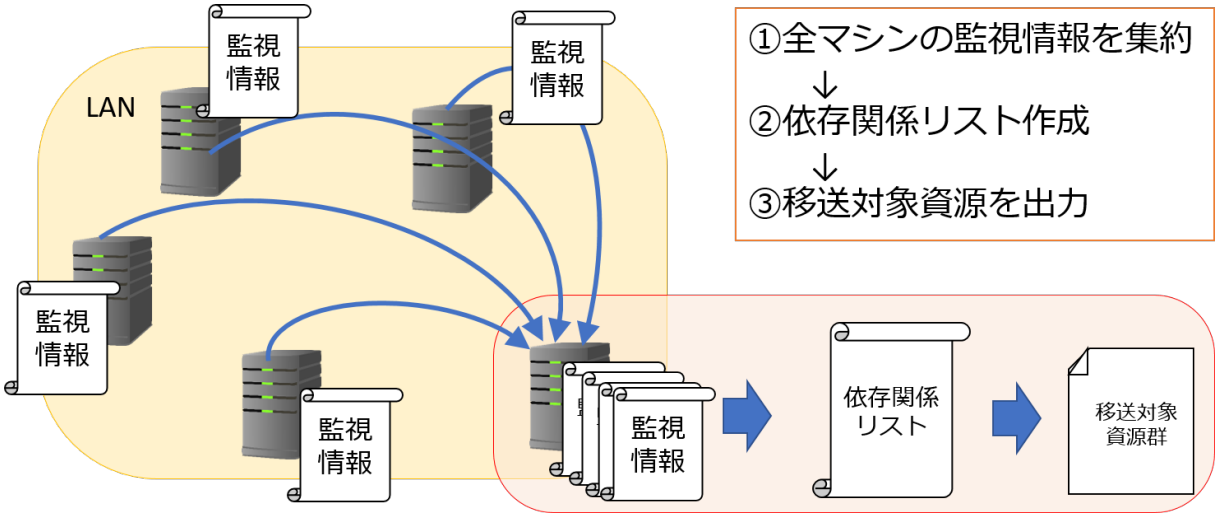

図 3.2　追跡フェーズの処理の概要図

第 4 章

実現方法

本研究の特定方式の実現方法を説明する．本特定方式は，FreeBSD 11.2-RELEASE にて実装を行った．

4.1 監視フェーズ

文献 [3] と文献 [4] で，ファイルアクセスのユーザレベルでの監視は実現できているので，本研究で新たに監視対象とするのはプロセスの親子関係とネットワーク通信である．監視機構を実装する関数と，その関数が定義されているソースファイルを表 4.1 に示す．カーネルのメモリの消費を抑えるため，可能な限りユーザレベルで実装するようにした．accept，connect は open 同様 C ライブラリ関数であるため，AP が accept() や connect() を実行した後は一度システムコールラッパーを介してシステムコールが発行される．監視機構は，このシステムコールラッパーに実装した．取得した情報は逐次 CSV としてファイルに出力した．fork，exec は直接カーネルへ処理が移るため，ユーザレベルでの実装はできなかった．また，取得した情報もカーネル内に蓄積されるため，これをユーザレベルで扱えるようにするためのユーザ定義システムコールを実装した．それぞれの情報の取得方法を説明していく．

4.1.1 fork

AP が fork() を実行したとき，カーネル内関数である sys_vfork() が呼び出される．この関数の定義の中に追記をし，必要な情報を取得する．取得する情報を以下に示す．

表 4.1 システムコール監視の実装

システムコール	監視する関数	関数が定義されているソースファイル
open	open()	/usr/src/lib/libc/sys/open.c
fork	kern_vfork()	/usr/src/sys/kern/kern_fork.c
accept	accept()	/usr/src/lib/libc/sys/accept.c
connect	connect()	/usr/src/lib/libc/sys/connect.c
exec	kern_execve()	/usr/src/sys/kern/kern_exec.c

- 親プロセスの PID
- 子プロセスの PID

これらの情報の取得方法を説明する．以下に，sys_vfork() の引数を示す．

```
int sys_vfork(struct thread *td, struct vfork_args *uap)
```

親プロセス，すなわち fork システムコール発行元の PID は，第 1 引数である thread 構造体 td のメンバの td_proc という構造体に，p_pid というメンバで格納されている．また，sys_vfork 関数は処理の最後に，td のメンバである td_retval という配列に子プロセスの PID を格納しているので，これを取得する．

取得した情報は，kern_fork.c 内にグローバル変数として構造体を定義し，この構造体のメンバに代入して保存した．しかし，このままでは取得した情報はカーネルのメモリ内にあり，ユーザレベルで扱うことができないので，これをユーザレベルに吐き出すためのシステムコールを実装した．監視情報の入った構造体の先頭番地から，システムコールの引数にとったユーザランドのメモリ番地へ，情報がなくなるまでコピーアウトする．その後，カーネルメモリ内の監視情報は削除して，メモリを開放する．これにより，監視情報を特定処理で利用できるようになる．

4.1.2 accept

ソケット通信は，サーバ側プログラムが通信を待ち受け，クライアント側プログラムがサーバに接続することで確立する．このとき，クライアント側は connect() でサーバを指定して接続要求し，サーバ側は accept() を実行して要求を受け付け，接続を確立させる．よって，accept システムコール，connect システムコールを監視して，ソケット通信の通信情報を取得する．取得する情報を以下に示す．

- サーバ側マシンの IP アドレス
- サーバ側 AP のポート番号
- サーバ側 AP の PID
- クライアント側マシンの IP アドレス
- クライアント側 AP のポート番号

これらの情報の取得方法を説明する．以下に，accept() の引数を示す．

`int accept(int s, struct sockaddr *addr, socklen_t *addrlen)` この accept() はシステムコールラッパーであり，定義の記述は，カーネルへ処理を渡すだけの単純なものである．カーネルから処理が返ってくると，引数 addr のメンバに通信相手の情報が格納されている．よって，カーネル処理の後に監視機構を追記し，クライアント側の IP アドレスとポート番号を取得する．accept() を実行した AP の PID は，getpid() を用いて取得する．getpid() は，実行中のプロセスの PID を返す関数である．サーバ側の IP アドレスとポート番号は，getsockname() を用いて取得する．getsockname() は，通信が確立しているソケットのファイルディスクリプタ (以降，FD と略す) を渡すと，そのソケットに関連付けられている通信情報を，sockaddr 構造体の変数として返す．渡すソケット FD は，accept の引数にある s である．これは，accept() を実行した AP が生成したソケットの FD であるため，サーバ側のソケット通信のための情報が得られる．監視情報は，accept 監視機構で情報が得られたら逐次 CSV ファイルに書き込む．

4.1.3 connect

connect はサーバ側とクライアント側が逆なだけで，accept とほぼ同じ処理で監視を行う．取得する情報を以下に示す．

- クライアント側マシンの IP アドレス
- クライアント側 AP のポート番号
- クライアント側 AP の PID
- サーバ側マシンの IP アドレス
- サーバ側 AP のポート番号

以下に，accept() の引数を示す．

`int connect(int s, const struct sockaddr *addr, socklen_t addrlen)` 引数 addr が，接続先のサーバを指定するための変数であるので，サーバ側の IP アドレスとポーと番号はこの変数から取得する．引数 s は，connect() を実行した AP が生成したソケットの FD であるため，accept と同様の方法で，クライアント側の IP アドレスとポート番号を取得する．クライアント側の PID も，accept と同様の方法で取得する．監視情報の保存も，accept と同様である．

4.1.4 exec

プロセスにプログラムが展開されるとき，exec() が実行される．したがって，ある PID のプロセスでどんなプログラムが実行されているかを知るには，exec システムコールを監視すればよい．exec() が実行されたとき，カーネル内関数である execve() が呼び出される．この関数の定義の中に追記をし，必要な情報を取得する．取得する情報を以下に示す．

- exec を実行したプロセスの PID
- 展開されたプログラムのファイルのパス

これらの情報の取得方法を説明する．以下に，sys_execve() の引数を示す．

```
int sys_execve(struct thread *td, struct execve_args *uap)
```

exec() で実行するプログラムのファイルパスは引数 execve_args のメンバ fname に格納されているので，これを取得する．exec() を実行したプロセスの PID は，引数 td のメンバである td_proc という構造体の，p_pid というメンバに格納されている．よって，これを取得する．取得した情報は，fork 同様ユーザ定義のシステムコールでユーザランドへコピーアウトする．

4.2　追跡フェーズ

追跡フェーズでは，第 3.2 節で述べた通り，依存関係リストを作成し，そこから移送対象資源を特定する．それぞれの処理を説明する．

4.2.1　依存関係リストの作成

依存関係リストは，依存関係にあるファイル資源の 1 対 1 の組の網羅リストである．また，ファイル資源を LAN 内で一意に識別するために「マシンの IP アドレス」と「パス」の 2 つの情報が必要である．したがって，依存関係リストは以下の情報を持っていなければならない．

- ファイル資源 A の出身マシンの IP アドレス
- ファイル資源 A のパス
- ファイル資源 B の出身マシンの IP アドレス
- ファイル資源 B のパス

しかし，第 4.1 節の open，fork，accept，connect の監視情報を見ると，依存関係リストの作成に必要な情報が一部揃っていない．したがって，監視情報を補完する処理が必要になる．依存関係リストの作成フローを図 4.1 に示す．この図に沿って説明する．

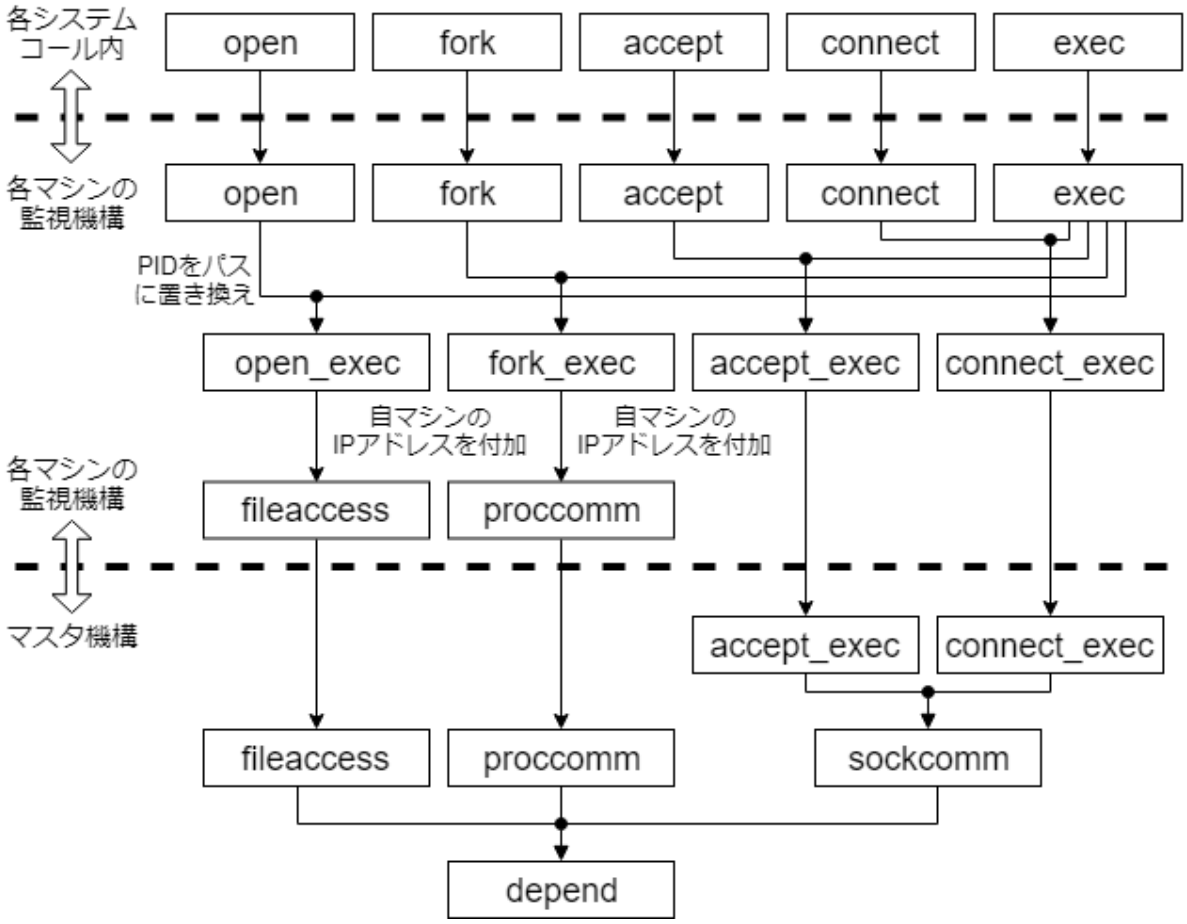

図 4.1　依存関係リストの作成フロー

(1) PID とパスの置き換え

ファイルアクセス，プロセスの親子関係，ネットワーク通信の監視において，AP の識別情報は PID で取得している．しかし，PID はプログラムの実行の度に変化するものであり，PID だけでは移送対象となるファイル資源を定められない．よって，exec の監視情報を基に，open，fork，accept，connect の監視で AP の識別情報として格納している「PID」を，すべて対応するパスに置き換える．置き換えるためには，プロセスとその上で動いているプログラムの対応表が必要だが，これは exec を監視することで得られている．各監視対象関数内からは直接 AP のパスを取得することができなかったため，このような方法をとった．

(2) IP アドレスの付加

依存関係リストを作成するために，LAN 内のすべてのマシンの監視情報をいずれか 1 台のマシンに集約する必要があるが，open と fork の監視情報は，そのまま集約するとどのマシンの情報かが分からなくなり，依存関係リスト内で AP を一意に識別できなくなる．そこで，この 2 つの監視情報には，出身マシンの IP アドレスを付加してから集約する．

ここまでの処理ができたら，監視リストを 1 箇所に集約する．

(3) ソケット通信の通信組を求める

accept，connect の監視情報それぞれ単体では，通信相手の IP アドレスとポート番号は分かっても，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 IP アドレスとパスが必要である．そこで，accept と connect の監視情報を相互に参照して，ソケット通信の通信組を求める．処理の概要を図 4.2 に示す．2 台のマシンが 1 回通信を確立したとき，accept と connect で 1 行ずつ監視情報が記録される．この組を検索し，それらの IP アドレスとパスで組にし，新たにリストにしたものを作成する．

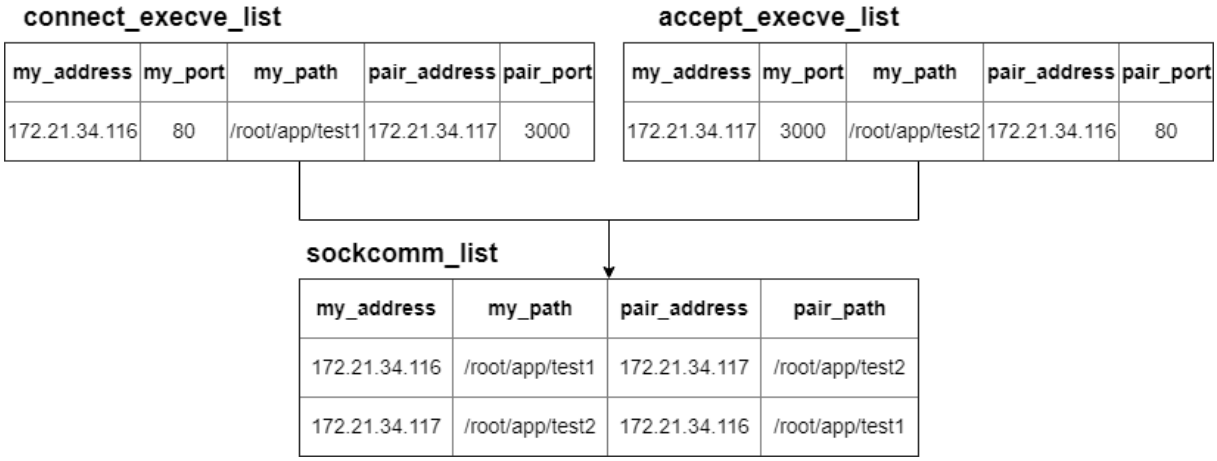

図 4.2　ソケット通信の通信組の求め方

(4) 依存関係リスト作成

ここまでの処理で，すべてのファイル資源の識別情報に IP アドレスとパスが揃ったので，すべての監視情報を，依存関係リストとして 1 つのリストに合体させる．

4.2.2　移送対象資源の特定

依存関係リストは，ファイル資源同士の「直接的な」関係の網羅リストである．しかし実際のソフトウェア実行環境では，関係性的に遠く離れたファイルとファイルが間接的に係わっている場合もあることが考えられる．このような関係性を依存関係リストから追跡し，移送対象資源を特定する方法を，図 4.3 のような例を用いて説明する．

移送したい AP，ここでは AP1 を起点 AP とする．これを依存関係リストに参照することで，AP1 と「直接」依存関係にあるすべての資源が得られる．得られたファイル資源は移送対象に含める．ここでは，AP2，File1 が得られた資源である．これらのファイル資源で再度依存関係リストに参照すると，AP2 と File1 と「直接」依存関係にあるすべての資源が得られる．得られた資源 AP1，File2 のうち，File2 はまだ移送対象に含まれていないため，含める．この File2 で再度依存関係リストに参照すると，File2 と直接依存関係にある AP2 が得られるが，これは既に移送対象に含まれているためここで処理を終了，現時点で移送対象に含まれているファイル資源群が，求めたいソフトウェア実行環境である．

このように，ファイル資源で依存関係リストに参照し，直接的な依存関係を求める処理を再帰的に行うことで，間接的なものも含めて依存関係を求めることができる．

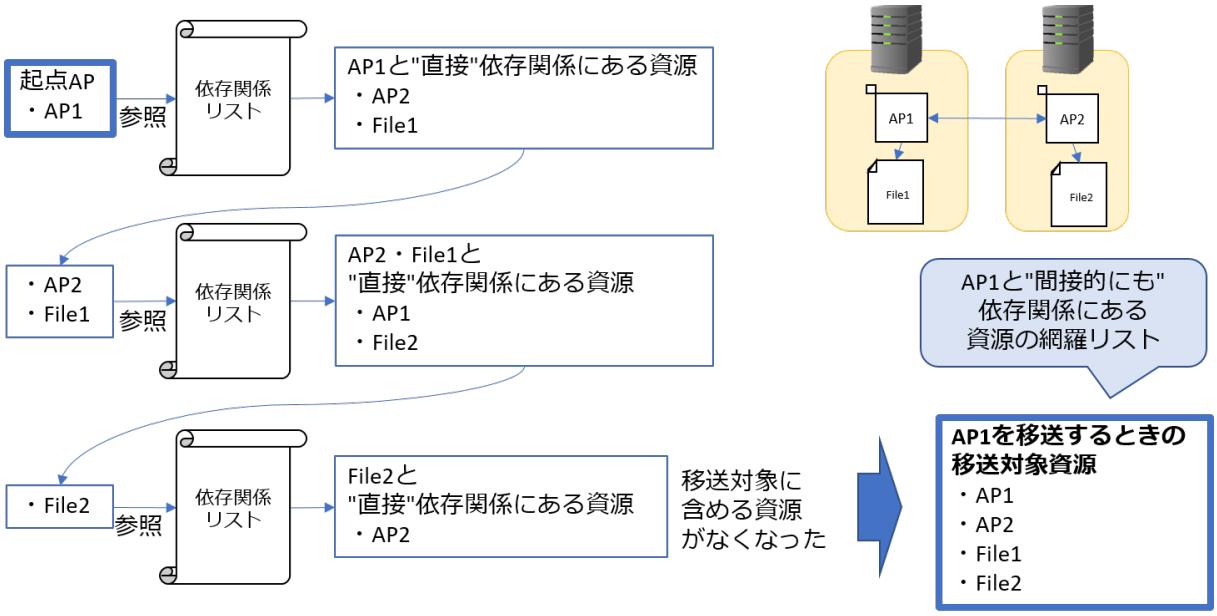

図 4.3　依存関係リストから移送対象資源を特定する方法

第 5 章

評価

ネットワーク通信の監視を実現するために，システムコールラッパーである accept() と connect() に処理を追加した．これにより，AP が他の計算機の AP と通信を確立する度に監視機構が動作するが，これが AP の動作に大きな影響を与える可能性がある．そこで，accept，connect の監視機構のオーバーヘッドを計測し，ソケット通信確立の処理に対してどれほどの影響があるかを評価する．

また，特定フェーズの処理に時間がかかると，その分転送処理に移るまでの時間が延びることになるため，ソフトウェア実行環境の移送処理全体に影響を与える可能性がある．そこで，依存関係リストから移送対象資源を求める処理の実行時間を計測し，移送処理全体に対する影響を考察した．

5.1 評価環境

評価の環境を表 5.1 に示す．

表 5.1 評価環境

OS	FreeBSD 11.2-RELEASE
CPU	Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
メモリ	8 GB

5.2　ネットワーク通信監視機構のオーバーヘッド

accept は，/usr/src/lib/libc/sys/accept.c の，監視のために追記した箇所の先頭と末尾それぞれでタイムスタンプをとり，その差分をオーバーヘッドとした．結果を図 5.1 に示す．結果は 5 回の平均で 34.4 μ 秒であった．accept の監視処理の中で connect の監視は関与しないため，connect の監視の有無による差は誤差の範囲となった．

connect は，単純なソケット通信のプログラムを作成して実行し，accept，connect それぞれの監視が有効になっている場合となっていない場合のプログラム実行時間を計測し，それらの差分をオーバーヘッドとした．結果を図 5.2 に示す．結果は，connect のみ監視有無を切り替えた場合は，30.8 μ 秒であった．accept，connect のオーバーヘッドともに，単純なソケット通信のプログラムでも全体の実行時間の約 2%と十分小さい値であり，AP のソケット通信処理にさほど影響しないといえる．

また，図 5.2 で accept 監視の影響がない様に見えるのは，クライアント側から見て，ソケット通信確立の処理の中に accept 監視機構の処理を挟まないからである．/usr/src/lib/libc/sys/accept.c 内の監視機構は，カーネルから処理が帰ってきたあとに記述されているが，クライアント側マシンに対する通信確立の信号送信は，このカーネル処理の中で済んでいる．したがって，クライアント側からは，accept では監視処理が実行されていないように見える．

5.3　移送対象資源の特定処理の実行時間

依存関係リストから移送対象資源を求める処理の実行時間を評価するため，依存関係リストのログ数を図 5.3 のように変化させて，実行時間を計測した．1 つの AP がファイルアクセスを 5 回，fork を 1 回，ソケット通信を 1 回行ったときに生成される依存関係リストのログ 7 行を 1 セットとし，マシンは 5 台あると想定して，このセット数を増やした．結果は図 5.4 の通りであり，50000 セットでも 7.79 秒となった．コールドマイグレーションでの広域分散環境での移送を考えた場合，回線種別にも依るが数時間かかることが想定される．それ

5.3　移送対象資源の特定処理の実行時間

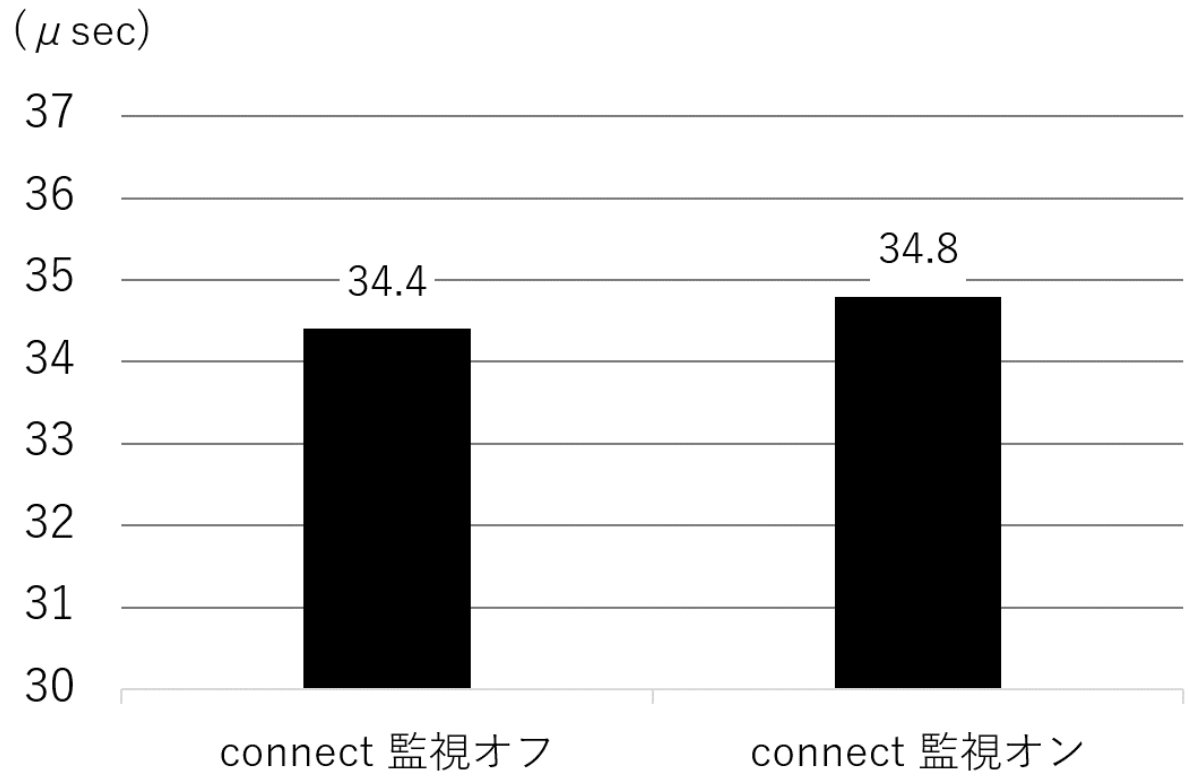

図 5.1　accept 監視機構の処理時間

に対して，8 秒という処理時間は，十分無視できるものと考えられる．また，グラフを見るに，計算量は準線形以下となっており，移送対象資源特定のアルゴリズムも実用に足るものだといえる．

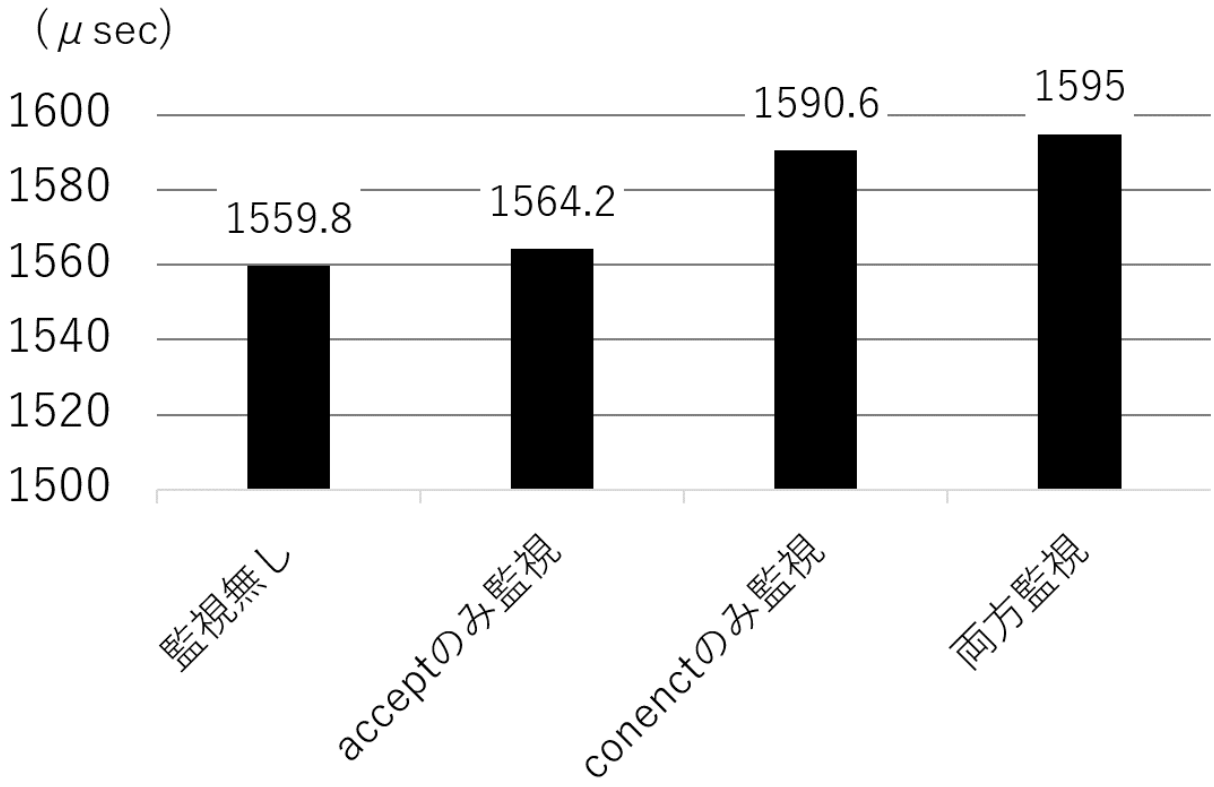

図 5.2　簡単なソケット通信のプログラムにおける connect 監視のオーバーヘッド

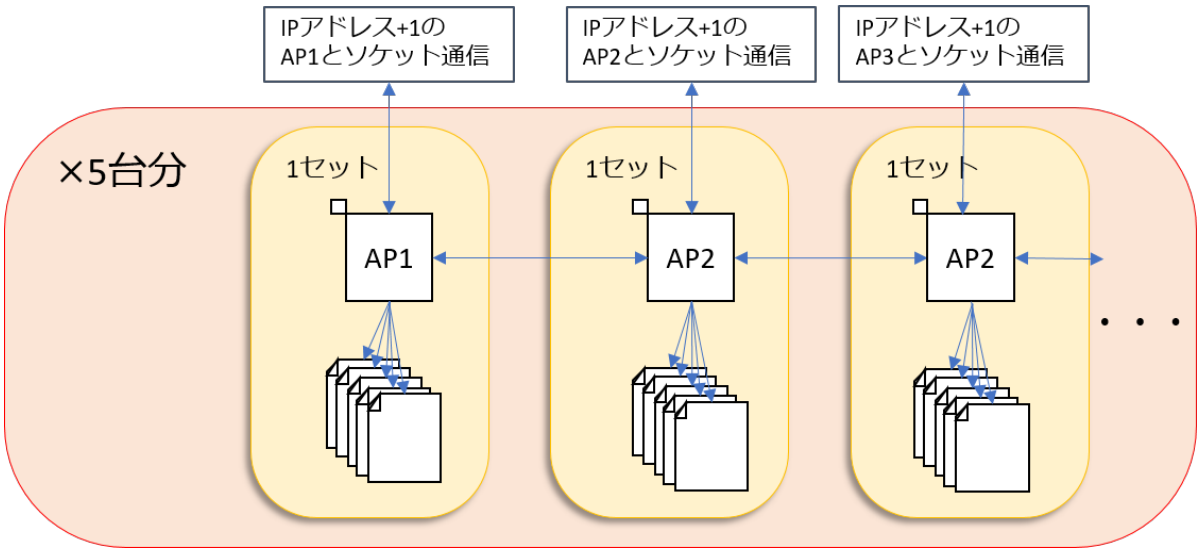

図 5.3　評価に用いる依存関係リストの内容

5.3　移送対象資源の特定処理の実行時間

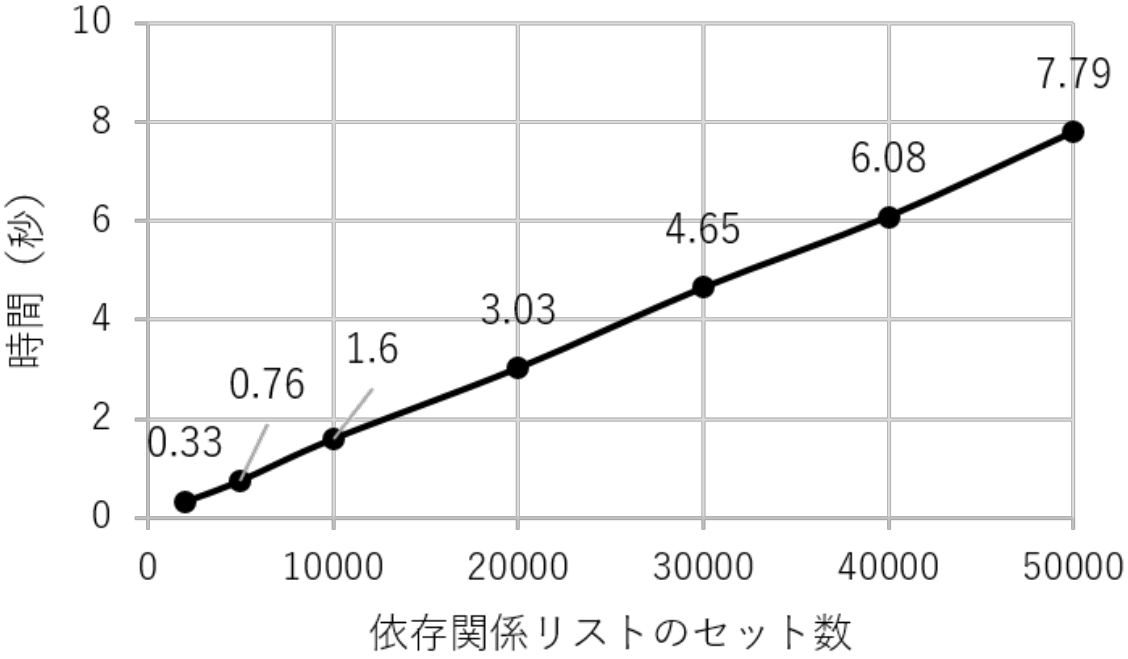

図 5.4　依存関係リストから移送対象資源を特定する処理の実行時間

第 6 章

おわりに

地理的に離れた複数のデータセンタ間を WAN で接続し，分散処理を行う広域分散システムでは，負荷分散やバックアップのために，ソフトウェア実行環境を別のデータセンタに移送することが頻繁に発生するため，効率的な移送技術が求められている．代表的な移送方法として，仮想マシンイメージをそのまま移送するものがあるが，これには移送する必要のないデータが含まれており，移送量が膨大になり，データセンタ間の低速なネットワークを占有してしまうという問題があった．また，移送時には仮想マシンは停止させなければならず，移送に時間がかかれば，その分サービスの停止時間も伸び，利用者に不利益をもたらしていた．そこで本研究では，ファイル資源の依存関係を特定し，AP が動作するために必要な最小限のファイル資源のみを移送する手法を提案した．提案方式では，ファイルアクセス，プロセスの親子関係，ネットワーク通信によるファイル資源同士の依存関係を追跡することで，計算機にまたがるソフトウェア実行環境の特定を実現した．評価としては，ネットワーク通信の監視機構のオーバーヘッドと，追跡処理の実行時間を測定し，十分実用に足る程度のものであることを示した．今後の課題として，fork，exec の監視機構の性能評価，追跡フェーズの依存関係リスト作成までの性能評価が挙げられる．また，他の計算機に存在する移送したいファイル資源が，自分が所有するものでない等の理由で移送できない場合が考えられる．そのような場合はその資源は移送せず，移送先で新たにネットワークを構築するなどの手立てが必要になる．

謝辞

本研究を進めるにあたり，卒業研究の指導教員としてご指導いただきました高知工科大学情報学群の横山和俊教授に心から感謝いたします．教授には，梗概の添削，卒業研究発表のアドバイスなど，様々なご指導をいただきました．卒業論文の副査を同学群の敷田幹文教授，植田和憲講師にしていただきました．卒業研究発表の場では，貴重なご意見をいただきました．同じく感謝申し上げます．さらに論文や研究成果をご参考にさせていただいた黒木勇作さん，大西史洋さんをはじめとする同研究室の方々に深くお礼申し上げます．

参考文献

[1] 上司 陽平，古藤 明音，河野 健二:Paas 環境におけるメモリ情報を用いた LiveMigration の負荷軽減，研究報告システムソフトウェアとオペレーティング・システム（OS），2015-OS-134，No.20，pp.1-7，2015.

[2] 石川 豊，山田 浩史，浅原 理人，花岡 美幸，河野健二:アプリケーション層プロトコルの状態を考慮した広域ネットワーク上での仮想マシン移送，情報処理学会論文誌コンピューティングシステム（ACS），Vol.4，No.2，pp.12-24，2011.

[3] 畑 翔太，谷村 直哉，横山 和俊:ファイルの共有関係に着目した移送するプログラムと実行環境の特定方法，情報処理学会研究報告，Vol.2015-DPS-164，No.11，pp.1-6，2015.

[4] 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，第 80 回全国大会講演論文集，Vol.2018，No.1，pp.319-320，2018.

[5] 黒木 勇作，大西 史洋，横山 和俊，谷口 秀夫，: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，研究報告高度交通システムとスマートコミュニティ（ITS），2018-ITS-73，No.16，pp.1-6，2018.