広域ネットワークにおける
アプリケーションのモビリティに関する研究

2006 年度

福田 浩章

論文要旨

現在，情報システムの普及，および高機能な計算機(端末)の低価格化に伴い，我々は複数の端末を利用して作業を行う機会が増加している．特に近年，特定の個人や組織が所有している端末だけでなく，不特定多数が利用可能な端末がいたるところに整備されつつある．しかし，セキュリティなどの理由から，一般ユーザがこれらの端末に新規にアプリケーションをインストールすることができない．そのため，多数の端末が配置されていても，そこで利用できるのは，ウェブブラウザなどの非常に限られたアプリケーションだけであり，日頃使い慣れたアプリケーションを利用するユーザは個人の端末を携帯せざるを得ない．すなわち，設置された端末の有効利用ができていない．

また，アプリケーションの個数も年々増加し，個々のアプリケーション自身も多機能化していく一方で，ユーザが実際に使用しているのはその中の一部であることも少なくない．そのため，不特定多数が利用する端末で，一般ユーザが自由にアプリケーションをインストールできたとしても，場当たり的にインストールした結果，管理が煩雑になるばかりでなく，端末リソースを浪費することになる．しかし，多くのアプリケーションは動的にリンクするライブラリなどを利用するものの，実質的に一つの実行プログラムであり，分割不能であるため，必要な機能だけを選択してインストールすることはできない．

一方，無線LANを代表とするネットワークインフラの普及により，不特定多数が利用可能な端末や個人の端末をネットワークに接続する環境が整備されつつある．しかし，不特定多数に接続を許可すると，そこが不正アクセスの温床になることは容易に想像できる．そこで，利用を許可されたユーザ(正規ユーザ)とそれ以外のユーザ(不正ユーザ)を特定し，正規ユーザが使用する端末(正規端末)だけをネットワークに接続させる仕組みや，端末の識別子であるIP/MACアドレスを偽造した不正端末の利用を防止する必要がある．

これらの背景を受け，本論文では端末の種類(ハードウェアやOS)に依存することなく，個々のユーザが任意のアプリケーションを必要な機能だけで実行し，端末リソースの有効活用を実現する手法について述べる．本論文では，アプリケーションの起動を契機に，必要な部分だけをネットワークを介して取得しながら実行する枠組みを提案する．そして，この実行方式をアプリケーションの管理へ応用し，インストール，実行，アップデートといった一連の作業を効率化するシステムの提案と実装を行う．これらのシステムにおいて，アプリケーションの取得にはJavaの動的ローディング機構を拡張し，実際に必要な機能(プログラム)だけを取得するため，端末リソースの有効活用が実現できる．

また，これらのシステムで必須となるネットワーク接続については，認証処理を行うことで正規ユーザを識別し，正規端末だけをネットワークに接続させるとともに，アドレス偽造による不正利用を防止する手法を提案し，システムを実現する．

これらの仕組みにより，正規ユーザであれば使用する端末に依存することなくネットワークに接続し，使い慣れたアプリケーションを必要な機能だけで利用することが可能になる．

Abstract

As a result of the diffusion of information system with the appearance of the low cost but high performance computers, two or more computers are available for each of us nowadays. Increasingly, computers are installed in public spaces for temporary use in addition to the computer that belongs to a person or an organization. However, users have no right to install their required applications in such computers because of the security reasons. Therefore, limited applications are available in such a computer, for example web browser, even if the number of public computers is increasing. If users require specific applications, they may need to carry their own personal computer with them at all times. In a word, the computers that are installed in public spaces are not effectively used.

In addition, the increase of the number is not only the applications but also the functions within an application. Besides, users usually use only a part of functions in spite of a large number of functions. It can cause a waste of computer resources and difficulties of application management for individual users to install there required application in public computers even if it is possible. However, it is impossible to install nothing but parts of an application separately because of its monolithic structure even though the application may use dynamic link libraries.

On the other hand, the spread of the network infrastructure such as Access Points of wireless network allows both public computer users and personal computer users to be able to connect to the network. It is, however, easy to imagine that there will be a large number of illegal accesses from the network if the network opens to everyone without any authentications. Therefore, the system requires to define users if they are authorized or not, and prevent malicious computers to access with spoof IP/MAC addresses.

Based on the background which is described above, this thesis presents the way of achieving effectual use of computer resources by running applications that each user requires with a need functions without relying on computer environment (hardware or OS). In addition, we would like to provide a framework that executes the application while downloading necessary functions from network. We would also like to apply this execution method to the application management system including install, execute and update the application. Furthermore, the reduction of starting time is explained and required resources of well designed applications by using the framework is compared with the traditional method that downloads entire program of the application before the execution. In order to download required functions (programs), we extend the dynamic loading mechanism of Java language. In addition, we provide a system that can distinguish authorized users and prevent illegal accesses from malicious computers by introducing user authentication to provide network connectivity the framework requires. Moreover, we confirm the efficiency of the system from benchmark tests and the working result.

These framework/system enables an authorized user to execute his/her required

applications while downloading necessary functions on demand from network with any available computers.

目 次

第1章 序 論 **1**
1.1 背 景 1
1.1.1 ユビキタス環境 1
1.1.2 多機能なアプリケーション 1
1.1.3 ネットワーク接続環境 2
1.2 アプリケーションの種類とモビリティ 2
1.2.1 アプリケーションの分類 2
1.2.1.1 クライアントサーバ型アプリケーション 2
1.2.1.2 スタンドアロン型アプリケーション 3
1.2.2 モビリティの定義 3
1.3 本研究の目的と概要 4
1.3.1 XFW 4
1.3.2 SmartMobile 5
1.3.3 AppliStore 5
1.4 論文の構成 5

第2章 関連研究 **7**
2.1 ネットワーク接続方式 7
2.1.1 管理者とユーザの要求 7
2.1.2 不特定多数のユーザを対象にしたネットワーク接続方式 8
2.2 端末に依存しないアプリケーションの実行方式 9
2.2.1 シンクライアント 10
2.2.2 モバイルエージェント 10
2.2.3 Java Web Start 12
2.3 本研究の位置づけ 14
2.3.1 ネットワーク接続方式の比較 14
2.3.2 アプリケーション実行方式の比較 14

第3章 アドレス偽装を防止したネットワーク接続方式 **19**
3.1 概要 19
3.2 不正利用防止法 20
3.2.1 接続方法 21
3.2.2 接続維持と切断の方法 22
3.2.3 アドレス偽造の防止法 23
3.3 XFWの設計と実装 24

3.3.1 XFW 25
3.3.1.1 HttpRedirector 25
3.3.1.2 Authentication Server 25
3.3.1.3 Authenticator 25
3.3.1.4 CommandExecutor 25
3.3.1.5 KeepAlive Checker 26
3.3.2 認証処理 26
3.3.2.1 HttpRedirectorとブラウザ間 26
3.3.2.2 Authentication Serverとブラウザ間 27
3.3.2.3 Authenticatorと外部認証システム 27
3.3.3 セッションIDの変更方法 27
3.3.3.1 Javascriptが有効な場合 28
3.3.3.2 Javascriptの動作判定と無効な場合の対応 28
3.3.3.3 管理者が設定するパラメータ 30
3.4 運用と評価 31
3.4.1 XFW単体の負荷テスト 31
3.4.2 大学での運用 33
3.4.3 評価 33
3.4.3.1 セキュリティ 33
3.4.3.2 性能 36
3.4.3.3 ユーザの利用方法 36
3.4.3.4 導入コスト 36
3.4.3.5 XFWの問題点 37
3.4.3.6 Opengateとの比較 37
3.5 まとめ 37

第4章 オンデマンドローディングを支援するフレームワーク 39
4.1 概要 39
4.2 GUIアプリケーションの特徴 40
4.2.1 処理の手順 40
4.2.2 機能とリスナーオブジェクトの関係 40
4.2.3 開発と実行時の問題点 41
4.3 アプローチ 41
4.3.1 依存関係の役割 42
4.3.2 SmartAPIの使用法 42
4.3.2.1 オブジェクトの生成に関するAPI 43
4.3.2.2 メソッドの実行に関するAPI 43
4.3.2.3 依存関係の作成に関するAPI 44
4.3.3 ローディング機構の実現法 44
4.3.3.1 インタフェース機能の実現法 44
4.3.3.2 オンデマンド生成機能の実現法 45
4.4 ランタイムの実装と動作 46

4.4.1 リフレクションAPI . . . 46
4.4.2 変換後のJavaコード . . . 47
4.4.3 ランタイムの動作 . . . 48
4.4.3.1 イニシャライズ部分の実行 . . . 48
4.4.3.2 オンデマンドなオブジェクト生成 . . . 49
4.5 アプリケーションの作成と実行 . . . 50
4.5.1 アプリケーションの作成法 . . . 50
4.5.2 アプリケーションの実行 . . . 50
4.5.3 SmartMobileを使ったアプリケーション . . . 52
4.5.3.1 実行例 . . . 52
4.5.3.2 評価 . . . 52
4.6 応用例 . . . 55
4.7 まとめ . . . 56

第5章 オンデマンドローディングのアプリケーション管理への応用 57
5.1 概要 . . . 57
5.2 デプロイ作業と問題点 . . . 58
5.2.1 一般的なデプロイ作業 . . . 58
5.2.1.1 デプロイ作業の要素 . . . 58
5.2.1.2 デプロイ作業の問題点 . . . 59
5.2.2 AppliStoreのデプロイ作業 . . . 59
5.2.2.1 開発者の役割 . . . 60
5.2.2.2 AppliStoreの役割 . . . 61
5.3 オブジェクトの生成処理 . . . 61
5.3.1 オブジェクト生成とクラスファイルの関係 . . . 61
5.3.2 オブジェクトとアプリケーション設計 . . . 63
5.3.3 クラスローダの拡張 . . . 63
5.4 設計と実装 . . . 64
5.4.1 アーキテクチャと各要素の関係 . . . 64
5.4.2 セキュリティ . . . 65
5.4.3 デプロイファイルと管理方式 . . . 65
5.4.3.1 デプロイファイルと配置 . . . 66
5.4.3.2 アプリケーションの管理方式 . . . 68
5.5 AppliStoreの動作 . . . 68
5.5.1 アプリケーションの公開と更新 . . . 69
5.5.2 アプリケーションの実行 . . . 70
5.5.3 アップデート . . . 71
5.5.4 AppliStoreの制約 . . . 73
5.6 アプリケーションの実行と評価 . . . 74
5.6.1 サンプルアプリケーション . . . 74
5.6.2 実行例 . . . 74
5.6.3 評価 . . . 76

5.6.3.1 公開時間 76
5.6.3.2 起動と機能実行時間 77
5.6.3.3 クラス数とデータ量 77
5.6.3.4 アップデート 77
5.6.3.5 総合評価 78
5.7 既存技術との比較 78
5.8 まとめ 79

第6章 総 括 81
6.1 まとめ 81
6.2 今後の展望 82

謝辞 85

図 目 次

2.1 シンクライアント 10
2.2 モバイルエージェント 11
2.3 Java Web Start 13

3.1 プライベートネットワークを使った接続法 20
3.2 XFWを利用した接続方法 21
3.3 セッションIDの変更方法 23
3.4 XFWのアーキテクチャ 24
3.5 リダイレクトレスポンス 26
3.6 認証用コンテンツに必要な部分 26
3.7 Authenticatorインターフェース 27
3.8 Top.html 28
3.9 Success.html 29
3.10 Script.html 29
3.11 Jscheck.html 30
3.12 1秒あたりのリクエスト処理数 32
3.13 同時接続数の推移 35
3.14 延べユーザ数の推移 35

4.1 GUIアプリケーションとMVCモデル 40
4.2 Javaの標準APIを使用したイニシャライズ部分の実装 41
4.3 SmartAPIを使用したイニシャライズ部分の実装 43
4.4 SmartAPIを使った依存関係の作成 43
4.5 ローディング機構の実現法 45
4.6 変換後のJavaコード 47
4.7 オブジェクトの生成とメソッドの実行法 48
4.8 アプリケーションとランタイムの状態 49
4.9 EventProducerクラスの定義 49
4.10 アプリケーションの作成と配置 51
4.11 変換ツールの使い方 51
4.12 アプリケーションの実行手順 51
4.13 TinyMessengerの実行 53
4.14 Mobicomを用いたアプリケーションの利用法 55

5.1 一般的なデプロイ作業 58

5.2 AppliStoreのデプロイ作業 60
5.3 オブジェクトの生成 62
5.4 アプリケーション設計とオブジェクト 62
5.5 クラスローダとオブジェクトの関係 63
5.6 AppliStoreのアーキティチャ 65
5.7 AppXの依存関係 66
5.8 デプロイファイル 67
5.9 ディレクトリ構造 67
5.10 アプリケーションの管理 68
5.11 アプリケーション公開時の動作 69
5.12 アプリケーションの実行 70
5.13 アプリケーションのアップデート 72
5.14 アップデート手順 72
5.15 実行環境 75
5.16 アプリケーションの実行例 75

表 目 次

2.1　ネットワーク接続方式の比較 14
2.2　アプリケーション実行方式の比較 15

3.1　管理者が設定するパラメータ 31
3.2　テスト環境 32
3.3　運用サーバの仕様 34
3.4　XFWで使用するソフトウェア 34
3.5　延べユーザ数と同時接続数の最大値 34

4.1　SmartAPI 42
4.2　SmartAPIと変換後のクラス 46
4.3　SmartMobileでの起動時間 54
4.4　Java Web Startでの起動時間 54

5.1　バージョン間の差分 71
5.2　ASServerを用いた公開時間 76
5.3　ASClientを用いた起動と機能実行時間 76

第1章 序 論

1.1 背 景

1.1.1 ユビキタス環境

近年，我々の日常的な作業はシステム化され，高機能な計算機(端末)が低価格化している．それによって我々は自宅やオフィスではデスクトップ型端末，外出先ではノート型端末を使用するように，複数の端末を使用して作業を行う機会が増加している．特に近年，特定の個人や組織が所有する端末だけでなく，不特定多数が利用可能な端末がいたるところに配置され，端末をいつでも利用できる環境が整備されつつある．我々ユーザが端末を利用するとき，実際には端末そのもの(ハードウェアやOS)ではなく，目的に応じて適切なアプリケーションを選択して利用している．したがって，単に不特定多数が利用できる端末が十分に配置されたとしても，個々のユーザの目的に適したアプリケーションがインストールされていなければ意味を成さない．しかし現状では，セキュリティなど管理上の理由から一般ユーザにはこれらの端末に新規にアプリケーションをインストールする権限が与えられていない．そのため，それらの端末で利用できるのはあらかじめインストールされたウェブブラウザなど非常に限定されたアプリケーションだけであり，日頃使い慣れたアプリケーションを利用するユーザは個人の端末を携帯せざるを得ない．そのため，単に端末の個数だけに着目し，配置するだけではユーザの要求を満たすことにはならない．

1.1.2 多機能なアプリケーション

端末の性能が向上し一般に普及するにつれて，同等の機能をもった数多くのアプリケーションが開発され，配布されている．また，同一のアプリケーションであってもバージョンが上がるにつれて数多くの機能が追加され，多機能化している．しかし，アプリケーション全体の機能数に対し，ユーザが実際に使用するのはその中の一部であり，ほとんどの機能は利用されていない．また，多くのアプリケーションは動的にリンクするライブラリを使用することはあるものの実質一つの実行プログラムであり，分割不能であるため，必要な機能だけを選択してインストールすることはできない．したがって，実際には使用されないものも含め，すべての機能を事前に端末にインストールしておく必要があり，このことは端末リソース(HDDやメモリ)の浪費につながる．特に，不特定多数が利用できる端末において，個々のユーザがアプリケーションをインストールする権限を与えられた場合，ユーザごとに使い慣れたアプリケーションは異なるため，ほとんど同一の機能を有する

複数のアプリケーションがインストールされることになる．この場合，特にリソースの浪費は顕著になり，端末の管理も煩雑になることが容易に想像できる．

1.1.3　ネットワーク接続環境

無線LAN，およびxDSLなどの普及により，端末をネットワークに接続するインフラが整備されつつある．それに伴い，アプリケーションにもネットワークの使用を前提とした機能が増加している．端末の配置と同様，ネットワークもいたるところで利用できる環境が整備されつつあるが，不特定多数のユーザにネットワーク接続を無条件に許可することはできない．仮に接続を許可した場合，不正利用の温床となることは容易に想像できる．現在，ネットワークに接続し，不特定多数で利用可能な端末では，ユーザが使用できるアプリケーションを限定し，新規アプリケーションのインストール権限を与えないことで不正利用を防止しているが，この方法ではユーザに任意のアプリケーションを利用させることができず，端末の有効活用が実現できない．一方，個人の端末をネットワークに接続させる場合，ユーザは自由に端末の設定が可能であるため，任意のアプリケーションを実行することができ，不正利用が可能になる．

したがって，利用する権利をもった正規ユーザが使用する端末(正規端末)のネットワーク接続だけを許可し，それ以外の端末(不正端末)のネットワーク接続を防止する必要がある．しかし，不正端末のネットワーク接続を防ぐ従来の手法のほとんどは，端末の識別にIPアドレスやMACアドレスを使用しているため，それらを偽造するアプリケーションを実行されると不正利用を防ぐ手段がない．また，ユーザも初心者から熟練者までさまざまであり，専用アプリケーションを利用した不正利用対策を行う場合，インストールや設定で混乱の原因になる場合が多い．管理者は不正利用を防止し，ユーザは煩わしい作業をすることなく簡単に接続できる方式が望ましい．

1.2　アプリケーションの種類とモビリティ

本節では，処理の主体という点からアプリケーションを分類しそれぞれの特徴を述べた後，本論文で述べるモビリティの定義を明確にする．

1.2.1　アプリケーションの分類

1.2.1.1　クライアントサーバ型アプリケーション

クライアントサーバ型アプリケーションでは，サービスの要求を行うクライアント，クライアントからの要求を処理した後，クライアントに結果を返すサーバ，これら2種類のアプリケーションが，要求の送信と結果の受信を繰り返すことによって処理を進める．HTTPプロトコルを使用したウェブサーバ，ウェブブラウザによる静的および動的なコンテンツ配信や，遠隔手続き呼び出しなどはここに分類され

る．クライアントサーバ型アプリケーションでは，クライアントとサーバはともに単独では動作せず，処理の主体はサーバである．また，多くの場合クライアントとサーバが動作する端末は分離され，ネットワークを介して要求と結果の送受信を行うため，ネットワークは必須となる．なお，Ajax[1]やFlex[2]を用いたウェブアプリケーションでは，サーバだけではなく受信した結果をクライアントのJavascript[3]またはFlashプラグイン[4]が処理するが，本質的にはクライアントサーバ型アプリケーションである．

1.2.1.2　スタンドアロン型アプリケーション

スタンドアロン型アプリケーションでは，基本的に単一の端末ですべての処理が行われる．具体例としては，エディタや表計算などのオフィスアプリケーションが該当する．近年，スタンドアロン型アプリケーションであっても，事前に端末にインストールされていないデータをネットワークを介して取得する機能が実装されているものも見受けられる．しかし，それらの機能はアプリケーションの主要な機能の副次的な存在であるため，クライアント–サーバ型アプリケーションには分類されない．一方，1.2.1.1項で述べたように，ウェブサーバとウェブブラウザによるコンテンツ配信自体はクライアントサーバ型アプリケーションである．しかし，そこで用いられるウェブブラウザは，サーバとデータの送受信を行う代わりにローカルディスクからデータを読み込み，表示することができるため，スタンドアロン型アプリケーションと位置づけることができる．

1.2.2　モビリティの定義

本節ではモビリティの定義を明確にする．一般にソフトウェアの分野においてモビリティとは，データやアプリケーションをいつでも，どこでも，どんなデバイスを使用しても利用できる事を指す．一方，類似する用語としてポータビリティがあるが，ソフトウェアの分野においてポータビリティが高いこととは，僅かな改造で異なるプラットフォーム(CPUやOS) で動作することを指す．これらに対し本論文では，スタンドアロン型の中で，ネットワークを介してプログラムを取得し，任意の端末で実行できるアプリケーションをモバイルアプリケーションと呼び，その性質をアプリケーションのモビリティと定義する．したがって本論文で定義するモバイルアプリケーションはポータビリティを有することになる．

既存の研究やシステムの中で，モビリティを有するアプリケーションは，モバイルエージェントとして実装したアプリケーションや，Java Web Start[5]を用いて実行するアプリケーションである．アプリケーションをモバイルエージェントとして実装した場合，モバイルエージェントの特徴である移動を行うことにより，任意の端末にアプリケーションを移動して実行することができる．また，Java Web Startでは，端末にインストールされたアプリケーションマネージャが，オンデマンドで必要なプログラムを取得して実行する．これらのアプリケーションは，アプリケーション自身，または実行環境に移動する仕組みや，ダウンロードから実行までシー

ムレスに行う仕組みを備えている．

一方，ネットワークインフラの普及により，HTTPやFTPを用いた外部プログラムを利用することにより，容易にアプリケーションを取得できる．しかし，これらの方法で取得したアプリケーションを任意の端末で実行できるとしても，それらのアプリケーション自身にモビリティがあるとは言えない．HTTPやFTPを用いた転送は，従来CD/DVDといった外部メディアで行っていたアプリケーションの配布を置き換えただけに過ぎない．

1.3　本研究の目的と概要

本論文では，スタンドアロン型アプリケーションのモビリティを実現するための手法について述べる．1.1.1節で述べたように，近年個人の端末を携帯して利用するだけでなく，不特定多数が利用できる端末がいたるところに設置されつつあるため，アプリケーションのモビリティを実現することによりユーザの利便性が向上する．しかし現状では，端末のセキュリティや管理の煩雑化といった問題が存在する．また，1.2.2節で述べたモバイルアプリケーションの場合，実際には使用されない機能 (プログラム) まで取得する必要はない．

そこで本論文では，端末の種類に依存することなく，個々のユーザが任意アプリケーションを必要な機能だけで実行する枠組み (SmartMobile[6]) を提案する．SmartMobileでは，アプリケーションの起動を契機に，ユーザが必要な機能をオンデマンドでネットワークを介して取得しながら実行する．そのため，使用しない機能の転送は不要であり，大規模なアプリケーションであっても最小限の機能だけで実行が可能である．そして，この実行方式をアプリケーションまたはライブラリの管理へ応用し，配布元の一元管理，および変更部分だけをシームレスにアップデートするシステム (AppliStore) の提案と実装を行う．そして，アプリケーションを実装し，これらのシステムで実行することによって有効性を確認する．

また，これらのシステムではネットワーク接続が必須となるが，1.1.3節で述べたように不特定多数のユーザにネットワーク接続を許可することはできず，不正利用も防止しなければならない．この問題に対し，本論文では一般的なウェブブラウザだけを使用して，正規ユーザの識別と正規端末のネットワーク接続および，アドレス偽造による不正利用を防止するシステム，XFW (Extensive Firewall) [7]の提案と実装を行い，負荷テストと運用実績を示す.

これらのシステムを導入することによって，ユーザは認証を受けた後ネットワークに接続し，任意の端末で必要なアプリケーションをオンデマンドで取得しながら実行することが可能になる．

次に，これらのシステムの概要を示す.

1.3.1　XFW

XFWでは，ほとんどの端末にはあらかじめインストールされているウェブブラウザだけを利用し，直感的なユーザ認証インタフェースを提供する．そして，外部

の認証機構と連携することでネットワークを利用できる正規ユーザを識別し，IPアドレスやMACアドレスといった端末の識別子を偽造することによる不正利用を防止する．そのため，ネットワークに接続するための専用アプリケーションを導入する必要もなく，初心者でも容易に利用することができる．

1.3.2　SmartMobile

SmartMobileでは，ユーザがアプリケーションの必要な機能を指定することによって，必要なプログラムだけをオンデマンドに取得して実行する．これを実現するため，SmartMobileではAPIとアプリケーションの実行環境を提供しており，開発者がAPIを用いて通常のプログラムには現れないオブジェクト同士の依存関係を定義し，それをもとに実行環境が機能の実行に必要なプログラム群を検出する．また，プログラムのオンデマンドな取得，および実行は，ネットワークとJavaの動的なクラスローディング機構の拡張によって実現する．SmartMobileを用いることによって，全体が大規模なアプリケーションであっても，実際に使用する機能だけで実行することが可能になる．なお，一度取得したプログラムは端末に保存されるため，例え別のユーザが実行する場合であっても，その端末で同じアプリケーションを実行する場合再利用される．

1.3.3　AppliStore

AppliStoreはアプリケーションの管理に着目し，アプリケーションとそれが依存するライブラリをまとめて配布する従来の方式に対し，依存ライブラリをURL[8]形式で定義することにより，配布元の一元管理を実現する．AppliStoreでは，オブジェクトの生成処理に直接割り込み，開発者がURL形式で定義するライブラリへのパスを解釈することにより，必要なプログラムをオンデマンドに取得することができる．また，アプリケーションやライブラリに変更がある場合には，変更部分だけをシームレスにアップデートする機能を備えている．そのため，開発者は自身が作成するプログラムの管理だけに集中すればよく，依存ライブラリの変更によるアップデート処理を自動化できる．なお，SmartMobileと同様に一度取得したプログラムは再利用される．

1.4　論文の構成

本論文の構成を次に示す.
2章では本論文で述べるそれぞれのシステムに関連する研究を述べ，本論文の位置づけを明確にする．そして，3章ではXFWが行うネットワーク接続方式について述べる．次に，4章でSmartMobileで実現するオンデマンドなアプリケーションの実行方式について述べ，5章ではその実行方式をアプリケーション管理へ応用したAppliStoreについて述べる．最後に6章で本論文の総括と今後の展望について述べる．

第2章　関連研究

本章では，本論文で実現するシステムと既存研究を比較し，本研究の位置づけを明確にする．次に，XFWに関連する既存研究としてネットワーク接続方式，SmartMobileとAppliStoreに関連する既存研究として端末に依存しないアプリケーションの実行方式の順に述べる．

2.1　ネットワーク接続方式

本節では，ネットワーク接続を提供する場合，管理者とユーザの立場からそれぞれの要求について述べた後，不特定多数を対象にした場合の優劣について述べる．そして，不特定多数を対象にしたネットワーク接続方式の研究，および既存システムについて述べる．

2.1.1　管理者とユーザの要求

ネットワーク接続を提供する場合，それを管理する管理者と利用するユーザにはそれぞれ次のような要求がある．
まず，管理者の要求として，

- 正規ユーザの識別
- アドレス偽造による不正利用の防止
- 通信の暗号化
- 多様な認証機構との連携
- 通信量の制御

が挙げられる．一方，ユーザの要求として，

- 接続手順の簡略化
- 情報漏洩の防止
- 接続用アプリケーションの排除

が挙げられる．

このように，管理者は利用者の制限や通信の暗号化など，ネットワークの頑健性を高めることを目的とし，ユーザは煩わしい作業を行うことなく瞬時にネットワークを利用できるなど，利便性を重視する．しかし，例えば通信を暗号化して頑健性を高める場合，個々のユーザが暗号化するためのアプリケーションを導入する必要があるように，頑健性と利便性は相反する性質を持つ．そのため，これらすべての要求をすべて満たすことは不可能である．1.1.3節で述べた不特定多数のユーザが対象の場合，たとえ初心者であっても容易に接続できる必要があるため，接続手順の簡略化や接続用アプリケーションの排除などの利便性を優先し，その上で可能な限り頑健性をの高い接続方式が望ましい．

2.1.2　不特定多数のユーザを対象にしたネットワーク接続方式

頑健性を高めるため，ネットワーク接続に認証機構を設けて提供するという研究やシステムは過去にも事例があり，いずれも目的に対して十分な効果をあげている[9, 10, 11, 12, 13]．しかしこれらのシステムでは，ユーザを認証するために，それぞれの接続機構に特化した専用アプリケーションを使用する必要がある．それらの専用アプリケーションは一般的な端末にはインストールされていないため，ユーザはあらかじめ別の環境でインストールしておかなければ利用できない．また，接続するための設定を行う必要があるため，初心者にとっては混乱の原因となる．したがって，これらのシステムの利便性は低い．また，専用アプリケーションを導入しているものの，通信の暗号化は行っていない．

R. Beckらのシステム[14]では，認証にtelnet[15]を使用し，ユーザの識別子となるユーザID，パスワードを入力してユーザ認証を行った後，端末をネットワークに接続する方式を提供している．しかし，telnetを使用しているため，ユーザID，パスワードという極めて重要な情報が平文でネットワーク上を流れることになる．このシステムでは，ユーザが能動的に認証を行う必要はあるものの，認証機構にあらかじめインストールされているtelnetを利用しており，認証にもIDとパスワードという直感的で分かりやすい機構を導入しているため利便性は高い．しかし，先に述べたようにtelnetを使用しているため，頑健性は低い．

Opengate[12]では，XFWと同様にブラウザだけを利用した認証方法を提供しており，認証時にブラウザにダウンロードされるJavaアプレットが，サーバとのTCPコネクションを確立し，サーバがそれを監視することによって端末の切断を検知する．したがって，XFWと同様にIPアドレスやMAC アドレスを偽造したとしても不正利用することはできない．このシステムを利用する場合，ユーザが行うことはブラウザを起動するだけであり，利便性を保ちつつ頑健性も高いシステムである．しかし，この仕組みにはJavaが動作するブラウザを使用する必要があり，Javaが実装されていないPDAなどの携帯端末では利用することができない．

一方，近年では空港やホテルでもUniversal Subscriber Gateway[16]やPOPCHAT[17]，といった製品を使用して，ネットワーク接続が展開されている．これらのシステムでは，XFWと同様に一般的なウェブブラウザだけを利用して認証を行った後，端

末をネットワークに接続することができる．また，課金システムと連携することによって，一度認証を受けた端末であれば料金に対応した期間内は自由にネットワークに接続することができる．そのため，不正利用を目的としたユーザであっても，認証を受けた端末を使用することによってネットワークの利用が可能である．しかしこれらのシステムでは，端末の識別にMACアドレスを利用しているため，MACアドレスを偽造することによって容易に不正利用が可能である．したがって，利便性は高いが頑健性は低い．

Apresia[18]では，正規ユーザが利用する度に認証処理を行い，ping[19]を使用することによって正規端末の接続性を確認している．そのため，IPアドレスやMACアドレスを偽造することによる不正利用が可能である．また，近年コンピュータウィルスの蔓延によるセキュリティソフトの導入により，デフォルトではpingのエコーリクエストに対してエコーリプライを返さない．そのため，セキュリティソフトの設定を変更しない限り，数分間接続した後切断されてしまう．したがって，頑健性も低く，利便性も高いとはいえない．

LANA[10][11]では， 802.1Q[20]に対応したスイッチを使用し，VLANタグを使用してIPアドレス，MACアドレスの偽造を防止している．LANAの場合，スイッチにカスケード接続されたハブを使って接続する正規端末の識別に必ずLANAクライアントが必要であるため，無線LANアクセスポイントを接続する場合，すべての端末にLANAクライアントが必要になる．したがって，頑健性は高いが利便性は低い．

また，IEEE802.1X[21]に対応したスイッチ，または無線アクセスポイントを導入する場合，すべてのスイッチやアクセスポイントがこの規格に対応する必要があり，これを利用するソフトウェアも必要となる．現在一般的なユーザが最も使用しているWindowsXP，およびMacOSXでは標準でIEEE802.Xに対応したソフトウェアがあらかじめ導入されているが，設定はユーザ自身が行う必要がある．そのため，頑健性は高いものの利便性が高いとは言えない．また，インフラの導入コストも無視できない．

2.2　端末に依存しないアプリケーションの実行方式

本節では，1.2.1.2項で述べたスタンドアロンアプリケーションを任意の端末で実行するための既存システムや研究を述べる．1.2.2節で述べたように，端末に依存せずにスタンドアロンアプリケーションを実行するためには，アプリケーション自身あるいは実行環境にモビリティの性質を与える方式がある．または，アプリケーションおよび実行環境にモビリティを与えるのではなく，クライアントサーバ型アプリケーションを応用し，イベント情報や表示情報を送受信して処理を行うシンクライアント方式がある．次に，シンクライアント，モバイルエージェント，Java Web Startの順にそれらの方式を述べる．

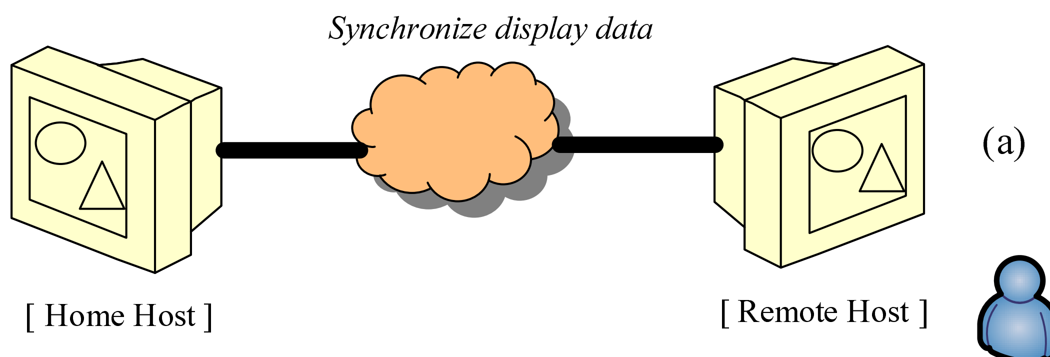

図 2.1 シンクライアント

2.2.1　シンクライアント

シンクライアントとは，アプリケーションのためのリソースを必要としないクライアントを指し，クライアントサーバ方式でサーバのアプリケーションを利用する．図2.1に示すように，この方式では利用するアプリケーションをサーバ([Home Host])にインストールしておき，サーバのディスプレイに表示される描画情報，およびマウスやキーボードの操作によって生じるイベント情報をサーバとクライアント([Remote Host])でやり取りすることで処理を進める．また，クライアントとサーバでやり取りするデータには共通のフォーマットやプロトコルは存在せず，一般に独自プロトコルを実装したアプリケーションによって実現される[22, 23, 24, 25, 26, 27].

この方式では，キーボードやマウスのイベント情報通知アプリケーションをクライアント端末にインストールする必要がある．また，クライアント端末から送信されるイベント情報を受信し，実行中のアプリケーションに通知するアプリケーションをサーバ端末にインストールしておく必要がある．これらのアプリケーションによってサーバ端末のディスプレイ情報がクライアント端末に描画され，ユーザはクライアント端末でサーバ端末のアプリケーションをあたかもクライアント端末で実行しているかのように使用することが可能である．したがって，不特定多数が利用する端末であっても，新規にアプリケーションをインストールする必要もなく，ユーザはサーバに接続して使い慣れたアプリケーションを利用することができる．

一方，この方式ではどんな処理を行う場合でもクライアントとサーバ間で通信が必要であるため，クライアントとサーバ間のネットワーク接続は常に安定している必要がある．したがって，LAN(Local Area Network)環境などネットワーク帯域が広く，安定した環境での用途に適している．また，あらかじめサーバにインストールされていないアプリケーションは利用できない．

2.2.2　モバイルエージェント

モバイルエージェント[28, 29, 30, 31, 32]とは，ネットワークに接続した端末間を移動しながら処理を進めるプログラムのことを指す．図2.2に示すように，モバイ

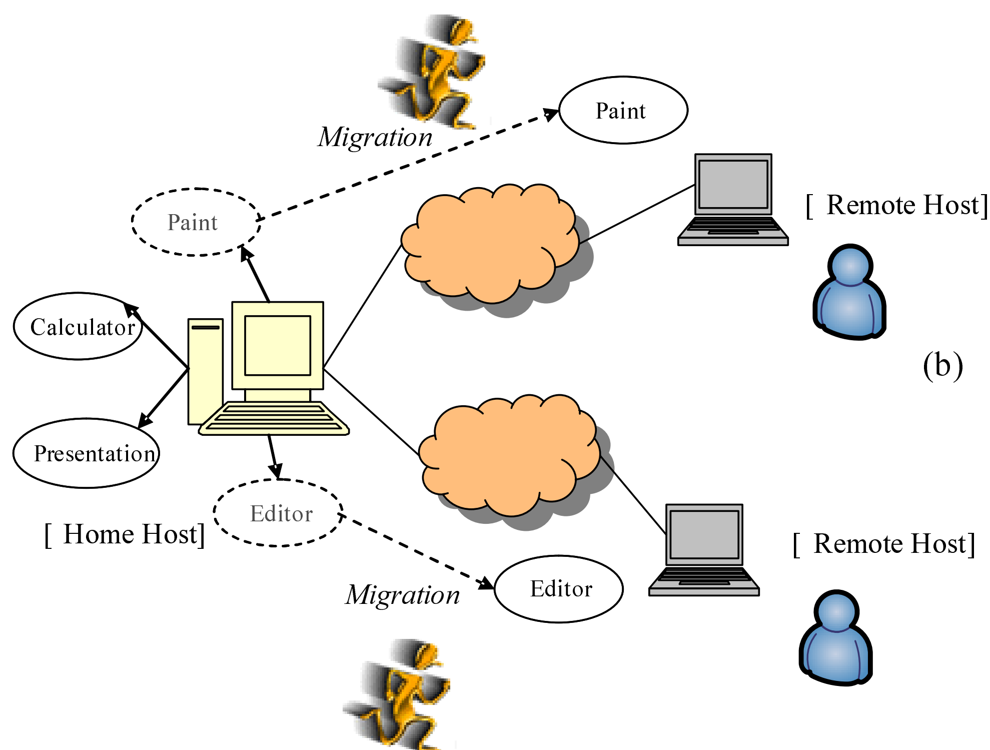

図 2.2 モバイルエージェント

ルエージェントの移動の際には，プログラムコードだけでなく，実行状態も移動するという特徴がある．そのため，移動先の端末で移動前の実行状態から処理を再開することができる．モバイルエージェントには，分散処理における通信遅延および通信回数の削減，耐故障性，負荷分散などの数多くの利点が挙げられ，次世代の分散処理システムとして注目を集めている．

ネットワーク上を移動するプログラムとしては，モバイルエージェントが始まりではなく，Process Migration，Remote Evaluation というプログラムの移動技術が従来から研究されていた．これらの技術は，負荷分散や耐故障性の実現に主眼をおき，プログラムは受動的に移動を行っていた．一方，モバイルエージェントは積極的に移動を行い，より高度な処理を実現するという概念を含んでいる．

モバイルエージェントの移動や実行を行うには，エージェントプラットフォームと呼ばれるソフトウェアを移動前と移動先の端末であらかじめ動作させておく必要がある．これらはエージェントの実行を管理，および制御するとともに，エージェントの送受信を行うソフトウェアである．エージェントプラットフォームを含むモバイルエージェントシステムに関する研究は，1990年代中頃に登場した商業用モバイルエージェントシステムであるTelescript[33]以降本格化する．そして，1997年頃からはJava 言語を利用したモバイルエージェントシステムが数多く研究開発されている[34][35]．また，モバイルエージェントの応用分野も，従来の遠隔情報収集や負荷分散だけでなく，通信ネットワーク管理やワークフローシステムなどにも利

用され，広がりを見せている．

アプリケーションをモバイルエージェントとして実装し，エージェントプラットフォームが動作している端末間の移動を行うことによって，任意の端末でアプリケーションを継続的に実行することができる．例えばJavaを利用したモバイルエージェントシステムの1つであるAgentSpace[36]では，モバイルエージェントの実装にAWT(Abstract Windowing Toolkit)のFrameクラスを継承しており，GUIアプリケーションのモバイルエージェント化を容易に行うことができる．またMobiDoc[37]では，独立したアプリケーションをモバイルエージェントのコンポーネントとして作成し，MobileSpace[38]の階層構造によって上位コンポーネントが内包するコンポーネントを含んだまま移動することで，移動性を持つ複合ドキュメントを実現している．

モバイルエージェントの移動は，移動元端末から移動先端末へ送り出す(Push型)方式であるため，エージェントプラットフォームが端末にインストールされているだけでなく，動作している必要がある．また，ユーザが場所を移動し，移動先の端末で処理を継続する場合，移動先端末にアプリケーションを移動させておく必要があるため，移動後に使用する端末を移動前に特定しておく必要がある．

また，モバイルエージェントの課題としては，セキュリティやエージェントプラットフォームが異なる場合の相互運用が挙げられる．モバイルエージェントのセキュリティとは，一般に悪意のあるモバイルエージェントから移動先端末を守ること，悪意のある端末からモバイルエージェントを守ること，の2通りがある．相互運用に関しては，モバイルエージェント技術そのものが登場してからあまり時間がたっていないため，特定のエージェントプラットフォームで構築したシステムを異なるエージェントプラットフォーム上のシステムと連携や相互運用させることが困難である．なお，モバイルエージェントのセキュリティに関する研究としては[39, 40, 41, 42]などがあり，相互運用に関する研究には[43]などがある．現在，モバイルエージェントは主にFIPA(Foundation for Intelligent Physical Agents)やOMG(Object Management Group)といった業界団体において標準化が進められているが，標準化は難しいと言われている．

2.2.3 Java Web Start

Java Web Start(以下，JWSと呼ぶ)は図2.3に示すようにウェブを利用してJavaアプリケーションをユーザに配布するための仕組みであり，2000年にSun Microsystemsによって提案と開発が行われた．ユーザは端末にインストールされたアプリケーションマネージャを使用し，ダウンロードされたアプリケーションを自由に利用できる．

JWSを利用するためにクライアント端末で必要になるのは，JWSをJavaの実行環境であるJRE(Java Runtime Environment)とともにインストールすることである．一方，サーバでは特別なサーバプログラムを導入する必要はなく，ウェブサーバの新しいMIME[44]タイプとして，JNLP(Java Network Launching Protocol)[45]ファイルを追加しておけばよい．また，サーバ側でアプリケーションを配布するのに最低

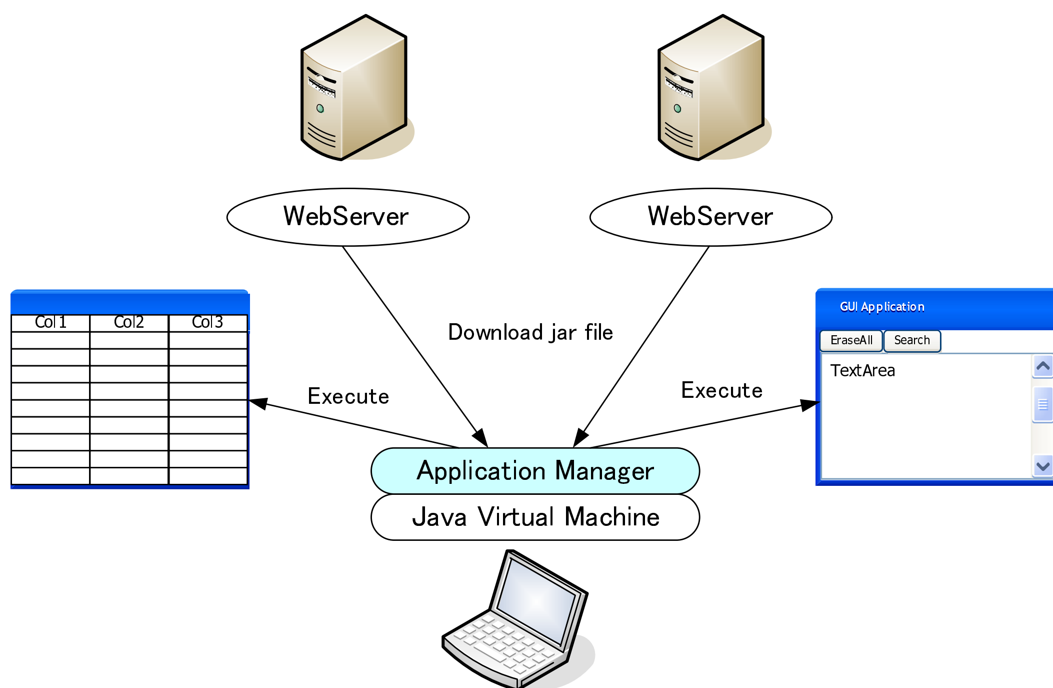

図 2.3 Java Web Start

限必要な作業は，

- Javaアプリケーションを作成してJARファイル形式にする
- JNLPファイルを作成する
- アプリケーション起動用のHTML文書を作成する

ことである．

ユーザがアプリケーションを実際に利用する手順は，ウェブブラウザからHTML文書にアクセスし，リンクをクリックするだけである．すると，JWSが自動的に起動し，JNLPファイルの記述にしたがって，JAR形式のアプリケーションをダウンロードして実行する．一度ダウンロードしたJavaアプリケーションは，次回からはウェブにアクセスする必要はなく，クライアント端末だけで実行することができる．また，JWSにはアプリケーションがアップデートされた場合に差分だけをダウンロードしてアップデートする機能や，通常は必要ないヘルプ機能などを分割し，別のJARファイルにすることで起動時の転送量を減らす機能がある．

JWSを利用することによって，ユーザはJWSがインストールされた任意の端末で使い慣れたアプリケーションを利用することができる．しかしJWSでは，アプリケーションを複数のJARファイルに分割できるものの，JARファイル分割はライブラリ単位や，パッケージ化のポリシーなど，開発者の視点から行うものであり，1.1.2節で述べた多機能なアプリケーションを実行する場合，ユーザが実際に使用する機能と分割されたJARファイルは必ずしも一致しない．また，リンクをクリッ

クするだけでアプリケーションを実行できるものの，JARファイルを完全にダウンロードしてからでなければ起動できない．

2.3　本研究の位置づけ

本節では，これまで述べた既存研究と比較することによって，本研究で実現するシステムの位置づけを明確にする．次に，ネットワーク接続方式，アプリケーションの実行方式の順に述べる．

2.3.1　ネットワーク接続方式の比較

表2.1に本論文で実現するXFWと，ネットワーク接続に関する既存研究の比較結果を示す．比較項目は，利便性を表す項目として，ネットワーク接続用専用アプリケーションの有無とし，頑健性を表す項目としてIPアドレスやMACアドレス偽造による不正利用が可能か否かとする．

2.1節で述べたように，一般に利便性と頑健性は相反する性質である．そのため，不特定多数が利用する駅や空港のホットスポットでは利便性を重視し，一般的なウェブブラウザだけで認証処理を行うことができるAPresiaやPOPCHATといったシステムを導入している．

一方，大学や企業など特定多数が利用する環境では，ユーザのサポートを簡単に行うことができるため，アドレス偽造による不正利用を防止したLANAやPortGuardを導入している．

XFWは，通信の暗号化は行わないものの，ApresiaやPOPCHATと同様のウェブブラウザによる直感的な認証処理に加え，LANAやPortGuardと同様にアドレス偽造による不正利用対策を施したシステムである．したがって，管理者にとって重要な頑健性と，ユーザにとって重要な利便性を兼ね備えたシステムであると位置づけることができる．なお，表2.1 に示すように，類似システムとしてOpengateがあるが，Opengateとの差異については3.4.3.6項で述べる．

2.3.2　アプリケーション実行方式の比較

2.2節で述べた各システムと，本論文で実現するSmartMobileやAppliStoreのアプリケーション実行方式を比較して表2.2に示す．各システムの比較は次に示す5項目を基準に行う．

表 2.1 ネットワーク接続方式の比較

	アドレス偽造不可	アドレス偽造可
専用アプリケーションなし	XFW, Opengate	Apresia, POPCHAT, USG
専用アプリケーションあり	LANA, PortGuard	————

表 2.2 アプリケーション実行方式の比較

	ネットワーク接続性	状態の保持	部分実行	転送単位	管理機能
Thin Client	広帯域で安定	あり	—	独自データ	なし
Mobile Agent	切断可能	あり	不可能	直列化したデータ	なし
JWS	切断可能	なし	可能	JARファイル	あり
SmartMobile	切断可能	なし	可能	クラスファイル	なし
AppliStore	切断可能	なし	可能	クラスファイル	あり

- ネットワーク接続性
  ネットワーク帯域の広さと状態.
- 状態の保持
  アプリケーションで扱うデータや実行状態を保持できるか否か.
- 部分実行
  アプリケーションの中でユーザが実際に使う機能だけで実行できるか否か.
- 転送単位
  転送するアプリケーション(プログラム)の粒度.
- 管理機能
  アップデートやダウングレードといったアプリケーション管理機能の有無.

次に，これらの比較項目にもとづき，各システムの特徴を述べる.

(1) シンクライアント
2.2.1節で述べたように，シンクライアントはサーバのディスプレイに表示される描画情報やイベント情報をクライアントと送受信するため，広く安定したネットワーク帯域が要求される．この方式では，アプリケーションの実体はサーバに存在し，描画情報を受信するアプリケーション固有のフォーマットでデータがやり取りされるため，データや実行状態は保持され，部分実行という概念はない．また，クライアントで利用するアプリケーションの実体はサーバにインストールされたアプリケーションであるため，個々のアプリケーションの追加や削除，アップデートといった管理は従来のアプリケーション管理と同じくサーバ管理者が行う必要がある.

(2) モバイルエージェント
モバイルエージェントの移動は，移動前端末での直列化，移動先端末への転送と復元により実現される．アプリケーションをモバイルエージェントとして実装した場合，直列化処理によってデータを含めたアプリケーションの状態が保存されたあと転送され，移動後は移動先端末のリソースだけで動作するため，転送時を除いてネットワークへの接続は不要である．また，モバイルエージェントの実装では，多くの場合エージェントプラットフォームで規定

されたインタフェースに従う必要があり，部分的に移動させる機構は持ち合わせてはいない．そのため，部分実行は難しく，転送単位はアプリケーション全体を直列化したデータとなる．また，多くのエージェントプラットフォームでは，エージェントのライフサイクルを管理する機構はあるものの，エージェントのバージョン管理機構は備えていないため，アップデートなどの管理はエージェントプラットフォームではなく，各アプリケーション開発者がすべて行う必要がある．

(3) Java Web Start
2.2.3節で述べたように，JWSでは起動時のJARファイル転送以外ネットワークの状態に依存せず動作する．そして，開発者が１つのアプリケーションを明示的に複数のJARファイルに分割することによって部分実行も可能である．また，複数のバージョンを管理し，バージョン間の差分だけをアップデートする管理機能を備えている．しかし，JWSはアプリケーションを配布する仕組みであるため，アプリケーションで扱うデータや実行状態の保存は行わない．そして，アプリケーションの転送はJARファイル単位で行われる．

(4) SmartMobile
SmartMobileはJWSと同様にアプリケーションを配布する仕組みであり，一度ダウンロードした機能はネットワークの状態に依存せずに動作する．また，アプリケーションの状態は保持せず，バージョン管理などの機構も持ち合わせていない．しかし，JWSのJARファイル分割が開発者の視点で行われ，JARファイル単位で転送が行われるのに対し，SmartMobileではアプリケーションが１つのJARファイルで構成されているとしても，ユーザが必要な機能(プログラム)をオンデマンドにクラスファイル単位で取得するという特徴を持つ．

(5) AppliStore
AppliStoreはSmartMobileにアプリケーション管理機能を加えたシステムであり，それ以外の特徴はSmartMobileと同様である．また，AppliStoreのアプリケーション管理機能はJWSの差分だけアップデートする機能に加え，1.3.3節で述べたようにアプリケーションが依存するライブラリの所在をURL形式で定義することを可能にしている．この結果，アプリケーションやライブラリは必ず開発者が管理する場所から取得させることが容易になるため，バージョン管理の複雑化，煩雑化を防ぐことができる．

これまで述べたように，SmartMobileやAppliStoreはJWSと同様にアプリケーションを配布するシステムであり，状態の保持は行わないため，使用する端末を変更しながら継続して処理を行う用途には適さない．それらの用途にはシンクライアントやモバイルエージェントを利用するべきである．また，SmartMobileおよびAppliStoreでは，ユーザの視点から実際に使用する機能だけでアプリケーションを実行する機構を備えているため，大規模かつ実際にはほとんど使用されない機能を多く含んだアプリケーションの実行に適している．なお，SmartMobileやAppliStoreを利用する場合，モバイルエージェントやJWSと同様にプログラムの転送時以外はネット

ワークの状態に依存せずローカル環境でアプリケーションを実行するため，無線LANなどの不安定なネットワーク接続環境でも十分に機能する．

なお，AppliStoreのアプリケーション管理機能と既存システムとの差異については，5章で詳細に記述する．

第3章　アドレス偽装を防止したネットワーク接続方式

3.1　概要

現在ノート型端末やPDAといった携帯型端末の低価格化によって，個人の端末を携帯して利用するユーザが増えている．これにともなって駅や空港といった不特定多数が出入りする場所でも，共用の固定端末だけでなく個人の端末をネットワークに接続するために無線LANや情報コンセントが急激に設置され始めている．このように，多数のユーザを対象にネットワーク接続サービスを導入する場合，ネットワーク管理者，およびユーザの立場から次の要求を満たす必要がある．

まず，管理者は不特定多数のユーザに対して無条件にネットワーク接続を許可することはできない．さもなければ不正利用の温床になることは容易に想像できる．したがって，正規ユーザが使用する正規端末のネットワーク接続を許可し，不正端末のネットワーク接続を防ぐ必要がある．また，IPアドレスやMACアドレスといった端末の識別子を偽造することによって不正にネットワークを使用する行為も防がなければならない．つまり，ネットワーク管理者はネットワーク接続の頑健性を高める必要がある．

一方，不特定多数のユーザが対象の場合，ユーザも初心者から熟練者まで様々であるため，ネットワーク接続をするために煩わしい作業が伴うと，混乱やトラブルの原因となる．したがって，ユーザにとっては誰でも直感的に利用できるといった利便性が最も重要となる．

しかし，一般に頑健性と利便性は相反する性質であるため，両者を同時に満たすネットワーク接続方式を提供することは容易ではない．また，利用する端末が個人の所有物である場合はもちろんのこと，共用端末であってもアプリケーションの追加など，一般ユーザが自由に設定できる端末を使用する場合，管理者が頑健性を高めることは非常に難しくなる．したがって，現状の固定端末では，使用できるアプリケーションを限定し，新規アプリケーションのインストール権限を与えないことで不正利用を防止しているが，この対策は端末の利用法を限定していることになる．また，個人の端末を接続させる場合，利便性を重視してアドレス偽造による不正利用を黙認するか，頑健性を重視して不正利用を防止するためのアプリケーションを導入している．

このような背景を受け，XFWでは頑健性を保ちつつ利便性を重視したネットワーク接続方式を実現する．XFWでは，一般的な端末のほとんどにあらかじめインストールされているウェブブラウザ(以下，ブラウザと呼ぶ)を利用し，ユーザIDとパスワードによるユーザの識別を行って正規端末をネットワークに接続する方法を提

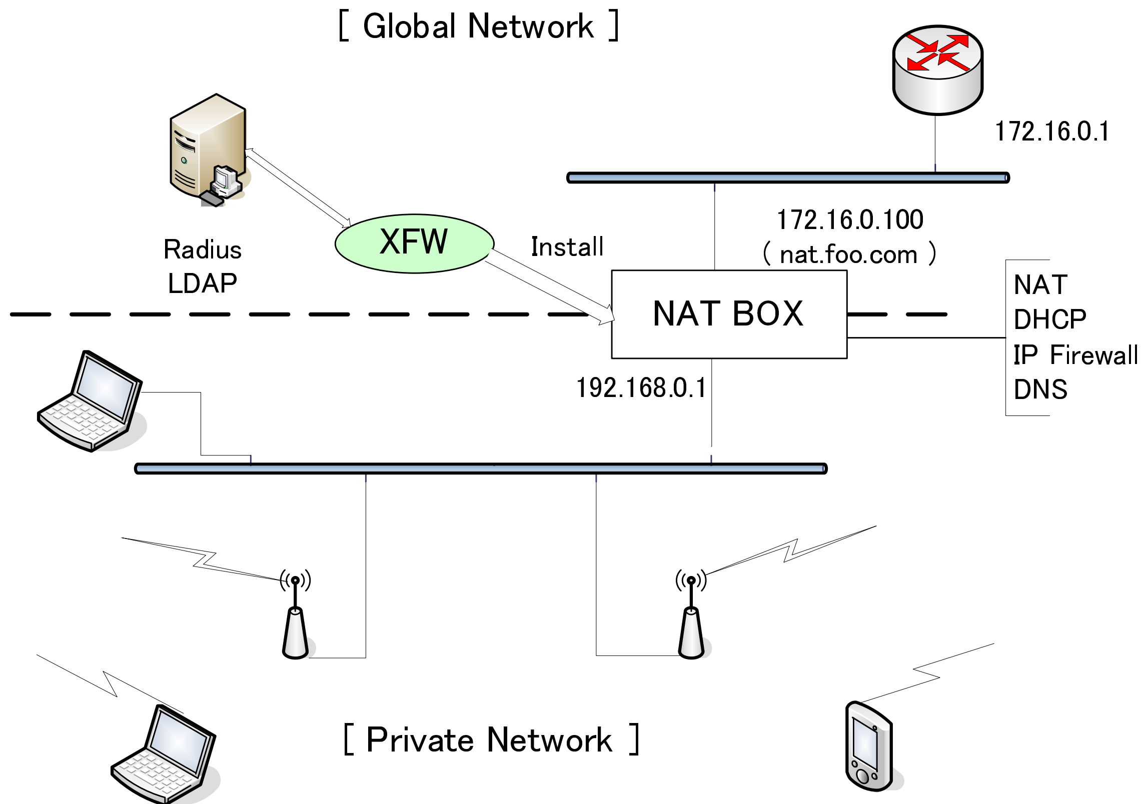

図 3.1 プライベートネットワークを使った接続法

供する．さらに，専用アプリケーションを導入することなく，ブラウザだけを使ってアドレス偽造を検知し，不正利用を防止する．

以下3.2節では，頑健性と利便性の両立する不正利用防止法について述べる．次に3.3節ではXFWの具体的な設計と実装について述べ，3.4節でXFWの運用と評価について述べる．最後に，3.5節ではまとめと今後の課題について述べる．

3.2　不正利用防止法

多数のユーザにネットワーク接続を提供する場合，図3.1に示すようなプライベートネットワークを利用して接続させることが多い．この環境では，情報コンセントに接続するか，無線アクセスポイントの電波を受信した端末は動的なIPアドレスの割り当てを受け，NAT BOX *を経由してグローバルネットワークに接続できる．

*NAT機能を提供するハードウェア

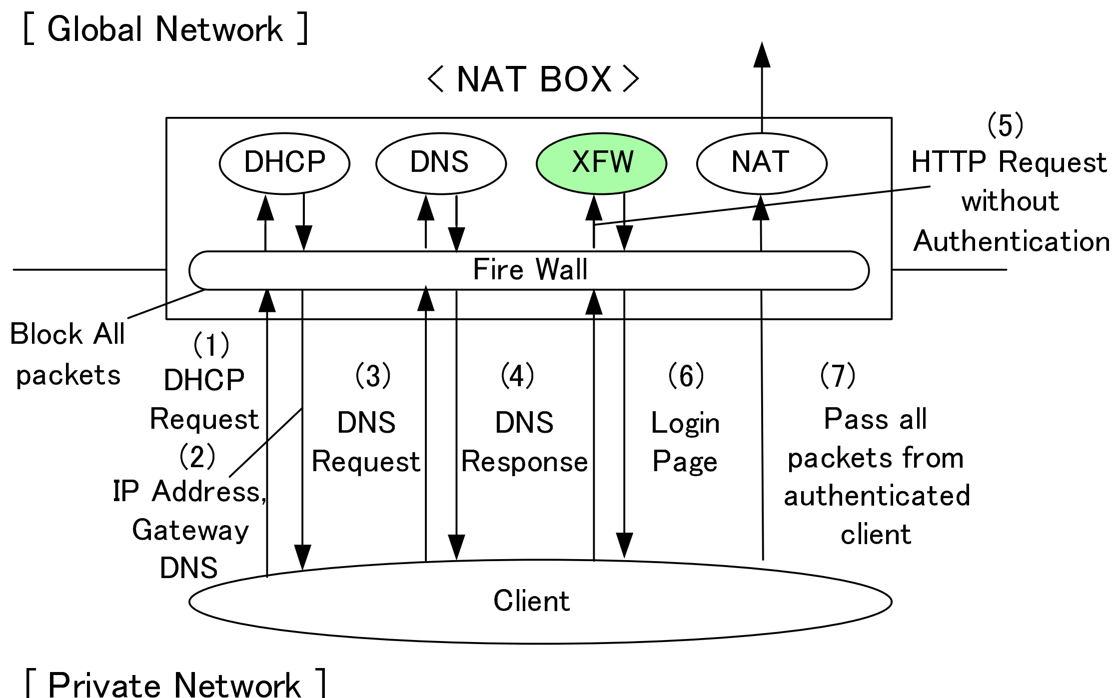

図 3.2 XFWを利用した接続方法

NAT BOXでは一般的にDHCP †やDNS‡，NAT §，Firewall ¶，といった機能を提供している．XFWはこのNAT BOX上で動作してユーザ認証を行い，Firewallの機能を利用して正規端末の接続を許可するとともに，それ以外の不正端末を使用したアクセスをすべて切断する．また2.1節で述べたように，Radius[46]やLDAP[47]といった多様な認証機構と連携することで正規ユーザの識別を行う．

次に，XFWを導入した場合のネットワーク接続方法と切断方法を示し，最後にアドレス偽造による不正利用を防ぐ方法について述べる．

3.2.1　接続方法

XFWでは図3.2の番号で示す手順でアクセス制限を行う．XFWを利用する場合，事前にFirewall機能でDNSとHTTP以外のパケットを破棄する設定にする必要がある．

(1) プライベートネットワークにつながった情報コンセントに接続するか，無線アクセスポイントから電波を受信し，端末がIPアドレスを取得するために

†端末に対して，IPアドレスやゲートウェイなど，ネットワーク情報を自動的に割り振るためのプロトコル

‡ホスト名とIPアドレスの関係を管理する仕組み

§グローバルネットワークとプライベートネットワークの接合部分において，IPアドレスの変換を行う仕組み

¶送信元または送信先IPアドレスやTCP，UDPなどのプロトコルに制限をかけ，不正なアクセスからホストを守る機能

DHCPリクエストを送信する.

(2) DHCPサーバが端末にIPアドレス，デフォルトゲートウェイ，DNSサーバ情報などを提供する.

(3) ユーザがブラウザを起動する．ブラウザはHTTPリクエストを出すサーバ名とIPアドレスの対応を解決するためにDNSリクエストを送信する.

(4) DNSサーバからIPアドレス情報を受け取る.

(5) ブラウザが，HTTPリクエストを送信する．Firewallの機能によって，HTTPリクエストは破棄せず，強制的にXFWにフォワードする.

(6) XFWはユーザが要求しているコンテンツに関係なくID，パスワードを要求する認証用コンテンツを返す.

(7) ユーザ認証に成功すると，XFWはその端末が正規端末であると認識して，使用しているIPアドレスとMACアドレス情報を取得し，その端末が送信するパケットを通過させるためにFirewallの設定を変更する．その結果，正規端末からのパケットだけがFirewallを通過する．すなわち，正規端末だけがネットワーク接続を許可される.

3.2.2 接続維持と切断の方法

3.2.1節の方法で正規端末の接続を許可し，その後ユーザが利用を止めたあと再びFirewallの設定を元に戻す必要がある．この処理を行わないと，利用を止めた正規端末と偶然同じIPアドレスを取得した端末が，認証を受けずにネットワークに接続できてしまう．このため，正規端末が接続しているかどうかを常に確認(生存確認)し，利用の終了を検出する必要がある.

正規端末の生存確認をするために，XFWでは正規端末からXFWに対して生存を通知するためのセッションID を一定間隔で送信させる．XFWでは，このセッションIDとIPアドレス，MACアドレスを利用して次のように端末の切断を検出し，実行する．なお，端末からXFWに送信されるセッションIDは3.2.3節で述べるアドレス偽造防止法にも利用され，送信する間隔はXFWで制御することができる.

(1) XFWは，正規端末のIPアドレスとMACアドレス，セッションIDを関連付け，セッションIDを受け取るたびに最終受信時刻を更新する.

(2) XFWは一定間隔でセッションIDの最終受信時刻をチェックし，一定時間更新のない正規端末を切断したと判断する．そして切断対象の端末が使用するIPアドレスとMACアドレス情報を抽出する.

(3) (2)で抽出した端末のIPアドレスとMACアドレス情報をもとに，Firewallの設定を変更して対象となる端末を切断する.

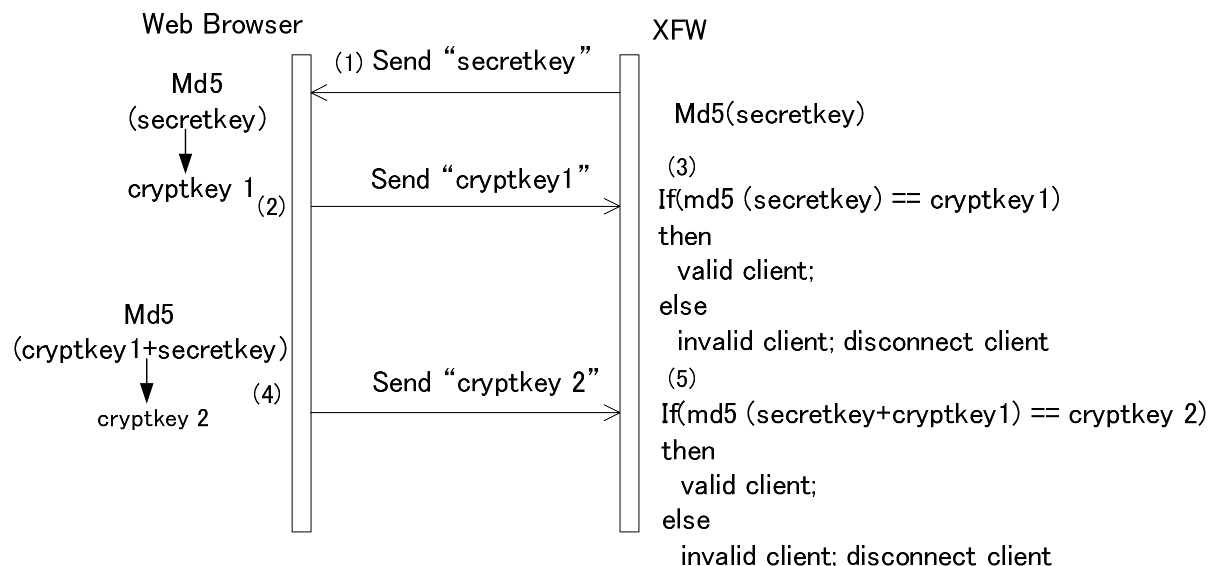

図 3.3 セッションIDの変更方法

3.2.3　アドレス偽造の防止法

XFWでは，IPアドレスとMACアドレス，セッションIDを使用して正規端末を識別しており，3つの値が完全に一致しない場合は不正利用とみなして即座に端末を切断する．しかし，3.2.1節で述べたように，不正端末であってもプライベートネットワークには接続できてしまうため，正規端末との通信を傍受され，セッションIDを盗まれると不正利用されてしまう．実際にWindows2000やWindowsXPでは，デフォルトでネットワークインタフェースのごとにMacアドレスを変更する機能を備えている．また，セッションIDのやりとりをHTTPSで行うことも考えられるが，事前の評価の結果毎秒20 リクエスト程度しか処理できない．そこでXFWでは，図3.3で示すとおり，秘密鍵とMD5ハッシュアルゴリズムを利用し，ワンタイムパスワードと同様な方法で正規端末とXFWでやり取りするセッションIDを毎回変更する．

(1) XFWは正規端末に対して秘密鍵を発行し，送信する．この間の通信にはHTTPSを使用するため，安全に端末に秘密鍵を送ることができる．

(2) 正規端末では受け取った秘密鍵のMD5ハッシュ値を計算し，その値を保存するとともに，HTTPを使ってXFWに送信する．

(3) XFWでも正規端末と同様に，その正規端末に発行した秘密鍵のMD5ハッシュ値を計算し，受け取った値と比較する．そして2つが同値の場合，その端末を正規端末であると認識し，受け取ったセッションIDと時刻を更新する．

(4) 正規端末は一定時間待った後，(2)で保存した値と秘密鍵を結合し，結合した値のMD5ハッシュ値を計算する．そして(2)と同様にその値を保存し，XFWに送信する．

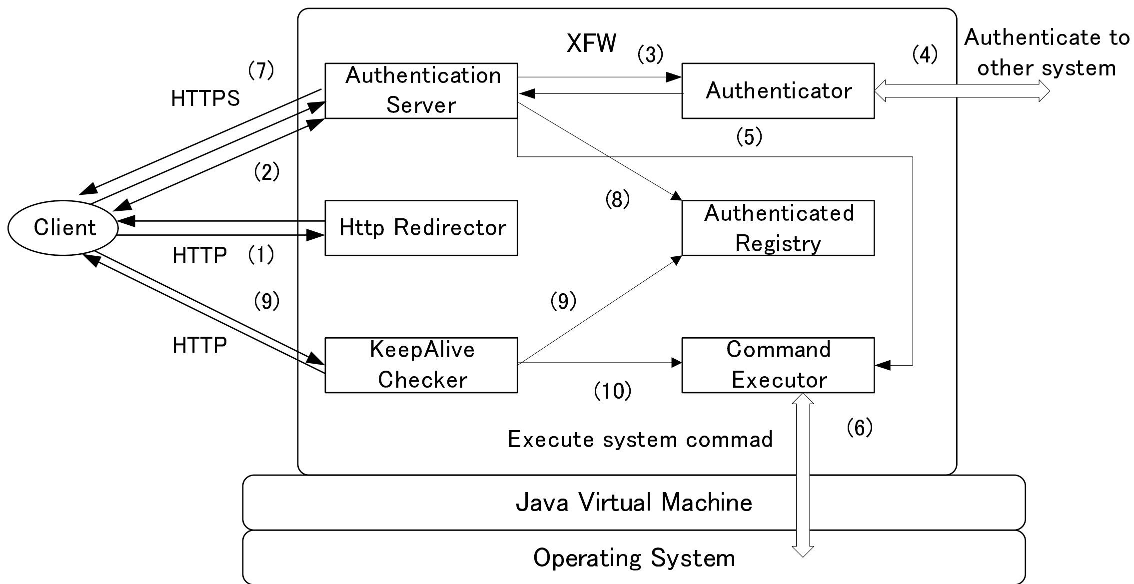

図 3.4 XFWのアーキテクチャ

(5) XFWでも正規端末と同様に，前回受け取った値と秘密鍵を利用してハッシュ値を計算し，受け取った値と比較することによって正規端末を識別する．

(6) 以後，(4)と(5)を繰り返し，正規端末の接続を維持する．

この方式によって，仮にIPアドレスとMACアドレスを偽造され，セッションIDを盗まれたとしても，盗んだセッションIDは既に無効であるため正規端末と不正端末を識別することができる．

ワンタイムパスワードでは，事前に秘密鍵を交換する必要があるが，上述の方式では認証終了時にXFWから秘密鍵が交付されるので，事前の鍵交換が必要ない．また，後述するとおり，XFWではこの方式をJavascriptが動くブラウザだけで実現している．

3.2.1節で述べたように，XFWを使用してネットワークに接続するためには，端末がDHCPでIPアドレスを取得したあと，少なくとも1回はHTTPリクエストを送信する必要がある．しかし現実には，ネットワークに接続するユーザの多くがウェブブラウジングを行うため，大部分のユーザに認証画面を表示することができる．

3.3　XFWの設計と実装

本節では，はじめにXFWを構成する各モジュールの機能と，それぞれの関係について述べる．次に，ユーザ認証時にブラウザとXFW，XFWと外部認証サーバでやり取りされるデータについて述べる．そして，セッションIDを変更する方法について述べ，最後にXFWを動作させるために管理者が設定するパラメータを示す．

3.3.1 XFW

XFW本体は，それが動作するOSに依存せず動作を保障するためにJava言語を使用する．図3.4に示すように，XFWは大きく分けて6つのモジュールで構成される．次にそれぞれのモジュールとその役割について述べる．

3.3.1.1 HttpRedirector

HttpRedirectorは，受け取ったHTTPリクエストに対して常にAuthentication Serverへリダイレクト応答を返す(図3.4の(1))． Firewallのフォワード機能によってフォワードされたHTTPリクエストは，すべてこのモジュールが処理する．

3.3.1.2 Authentication Server

HttpRedirectorがリダイレクト応答を返した結果，ブラウザがAuthentication Serverにリクエストを送信する．Authentication Serverはリクエストを受け取ると，ユーザ認証用コンテンツを返す(図3.4の(2))．また，ユーザが送る認証情報(IDとパスワード)を取得し，認証処理をAuthenticatorに依頼する(図3.4の(3))．認証に成功した場合，Firewallの設定を変更するためにCommand Executorに処理を依頼する(図3.4の(5))．そして，ユーザ認証とFirewallの設定変更に成功すると，秘密鍵を発行し，ユーザに認証成功を示すコンテンツを返す(図3.4の(7))．それと同時にAuthenticated RegistryにIPおよびMACアドレス情報と，秘密鍵を登録する(図3.4の(8))．一方，認証に失敗，または設定変更に失敗した場合は，認証失敗と再認証を促すコンテンツをユーザに返す.

3.3.1.3 Authenticator

Authenticatorは認証情報を受け取り，さらに外部の認証システム(Radius, LDAP etc.)と連携してユーザ認証を行う(図3.4の(4))．そしてAuthentication Serverに認証結果を返す．このモジュールと他のモジュール(Authentication Serverなど)は，Javaのインタフェースで結合しており，実装クラスを入れ替えることで，さまざまな外部認証システムとの連携が可能になる．

3.3.1.4 CommandExecutor

Authentication ServerまたはKeepAlive CheckerがFirewallの設定を変更するとき，端末を接続または切断するためのコマンドを作成し，外部プログラムを呼び出す(図3.4の(6))．Javaの実行環境からはFirewallの設定を直接書き換えられないため，外部プログラムとして実行する必要がある．

```
HTTP/1.1 307 Temporary Redirect
Connection: close
Location: https://nat.foo.com/login.html
```

図 3.5 リダイレクトレスポンス

```
<form action="/auth.html" method="post">
<input type="text" name="loginId" value=""/>
<input type="password" name="passwd" value=""/>
<input type="submit"/ name="Send">
</form>
```

図 3.6 認証用コンテンツに必要な部分

3.3.1.5 KeepAlive Checker

KeepAlive Checkerは，セッションIDを受け取り，そのセッションIDを送った端末が正規端末であることを確認した後，最終受信時刻の更新処理をAuthenticated Registryに依頼する(図3.4の(9))．一方，不正端末であると判断した場合，CommandExecutorに切断処理を依頼するとともに，Authenticated Registryのエントリーを削除する(図3.4の(10))．また，このモジュールはAuthenticated Registryから一定間隔で登録されているすべてのセッションIDと最終受信時刻を取得する．そして，一定時間更新されていない端末は切断したと判断し，不正端末の場合と同様に該当する端末を切断する．

3.3.2 認証処理

認証処理では，HTTPRedirectorとブラウザ，Authentication Serverとブラウザ，そしてAuthenticatorと外部認証システム間で次に示すデータがやり取りされる．

3.3.2.1 HttpRedirectorとブラウザ間

HttpRedirectorは受け取ったリクエストに対して図3.5で示すHTTPレスポンスを返す．ここでは説明のため，図3.1で示したようにNAT BOXにはグローバルIPアドレス“172.16.0.100”が割り当てられ，DNSサーバによって“nat.foo.com”として名前解決ができるものとする．この結果，ブラウザは“Location”ヘッダで示したURLにリクエストを再送する．

```
public interface Authenticator {
    public boolean authenticate(String loginId, String passwd);
}
```

図 3.7 Authenticatorインターフェース

3.3.2.2　Authentication Serverとブラウザ間

HttpRedirectorのレスポンスによって，ブラウザはAuthentication Serverに認証用コンテンツをリクエストする．認証用コンテンツは管理者が自由にカスタマイズ可能だが，HTTPのPOSTメソッドを利用して“/auth.html”にID(“loginId”)とパスワード(“passwd”)パラメータを送信できるものでなくてはならない．具体的には図3.6で示す部分を含む必要がある．

ユーザがログインIDとパスワードを入力し，送信ボタンを押すと，HTTPのPOSTメソッドでデータが送られ，Authentication Serverが認証情報を取得する．この間のデータはHTTPSで暗号化されるため，安全にやり取りできる．

3.3.2.3　Authenticatorと外部認証システム

Authenticatorは図3.7で示すインタフェースで定義される．したがって，このインタフェースを実装する具象クラスを入れ替えることで，さまざまな認証方式が実現できる．現在の実装では，Radiusプロトコルを使用して外部Radiusサーバと連携するRadiusAuthenticatorを実装している．

3.3.3　セッションIDの変更方法

3.2.3節で述べた方法を実現するためには，

- 生成した秘密鍵を安全に端末に送り，かつ端末から漏洩しない仕組み．
- 端末でMD5ハッシュ値の計算と，値の保持．

という用件を満たす必要がある．HTTPだけを使った通信では，端末に送られた秘密鍵を保存しておくことが難しい．唯一クッキーを使うことで保存することはできるが，通信は暗号化されていないため，ネットワークを盗聴されると簡単に盗まれてしまう．その結果，セッションIDを毎回変更したとしても端末の偽造が可能になる．

そこでXFWでは，HTMLのFRAMEとJavascriptを利用して，HTTPだけを使って上で述べた用件を満たす．Javascriptは通常のコンピュータに限らず，PDAなどの携帯端末でも動作するが，ユーザによってはセキュリティ等の理由で無効にしている場合も考えられる．これに対応するために，XFWは端末ごとにJavascriptが有効か無効かを判定する．次にJavascriptが有効な場合にセッションIDを変更する方法について述べ，Javascriptの動作判定と無効な場合の対応について述べる

```
<HTML>
<FRAMESET ROWS="0%,0%,*">
<FRAME SRC="" NAME="$SECRET">
<FRAME SRC="script.html" NAME="script">
<FRAME SRC="success.html" NAME="success">
</FRAMESET>
</HTML>
```

図 3.8 Top.html

3.3.3.1　Javascriptが有効な場合

図3.8にFRAMEのトップページ(top.html)，図3.9と図3.10にFRAME内部で使用するsuccess.htmlとscript.htmlのコンテンツを示す．図3.8に示すように，top.htmlではFRAMESETタグで3つのFRAMEを定義しているが，最初のFRAME指定は“SECRET”部分に秘密鍵を埋め込むために使用し，2番目のFRAMEにはMD5ハッシュ値の計算をするJavascriptと，値を保存するためのformを持つscript.htmlを指定する．これらのフレームの幅はいずれも“ 0% ”指定であるため，ユーザが目にすることはない．

3番目のFRAMEは“ 100% ”指定で，認証成功を示すsuccess.htmlを指定する．3.3.2.2項で述べたように，XFWではユーザ認証時の通信にHTTPSを使用するため，安全に秘密鍵を端末のブラウザに送ることができる．script.htmlでは，図3.10の(1)〜(4)に示す手順でセッションIDを変更する．

(1) top.htmlに埋め込まれた秘密鍵を取得する．

(2) Javascriptを用いて，(1)で取得した秘密鍵のMD5ハッシュ値を計算する．

(3) (2)で求めたハッシュ値をformのhiddenタグに保存する．

(4) (2)で求めたハッシュ値をsuccess.htmlのGETパラメータ“ sessionId ”として付加し，3番目のFRAMEをリロードする．

この方式では，秘密鍵が埋め込まれるtop.htmlとscript.htmlはHTTPSで一度だけ送信されるため，安全な秘密鍵のやり取りができる．また，ユーザは認証後の画面を開き続けておくだけで接続を維持することができる (ブラウザは最小化して構わない).

3.3.3.2　Javascriptの動作判定と無効な場合の対応

XFWでは，図3.11に示すjscheck.htmlを利用し，Javascriptの動作判定を行う．jscheck.htmlは，ユーザ認証成功直後に送信される．Javascriptが有効なブラウザは，

```
<HTML>
<HEAD><TITLE> Authenticate Success</TITLE></HEAD>
<BODY>
  認証成功.<BR/>この画面は閉じないでください.<BR/>
</BODY>
</HTML>
```

図 3.9 Success.html

```
<HTML>
<HEAD><SCRIPT type="text/javascript">
var secret, sessionId;
var url = "http://nat.foo.com:8000/success.html?sessionId=";
function keepalive() {
   setTimeout("keepalive()",5000);
   secret = parent[0].name;   (1)
   var currentKey = secret + document.forms[0].sessionId.value;
   sessionId = MD5_hexhash(currentKey);   (2)
   document.forms[0].sessionId.value = sessionId;   (3)
     var sendUrl= url + sessionId;
     parent.success.location.href = sendUrl;   (4)
}
function MD5_hash (data) {
  // encrypt data
  ----------------------------
  return crypted data;
}
</SCRIPT></HEAD>
<BODY onload="keepalive()">
<FORM NAME="form">
<INPUT TYPE="hidden" name="sessionId" value=""/>
</FORM>
</BODY>
</HTML>
```

図 3.10 Script.html

```
<HTML>
<HEAD>
<TITLE> Javascript Check</TITLE>
<meta http-equiv="Refresh" content="10 ;
URL=http://nat.foo.com:8000/nojsauth.html?sessionId=xxxx"/>
<SCRIPT type="text/javascript">
  function checkjs() {
      location.href="top.html?sessionId=xxxx";
    }
</SCRIPT>
</HEAD>
<BODY bgcolor="#FFFFCC" text="#000000" onload="checkjs()">
  ------------------------------------------------------------------
</BODY>
</HTML>
```

図 3.11 Jscheck.html

onloadイベントによって図3.11の“checkjs()”関数を実行し，3.3.3.1項で述べた方法でセッションIDを毎回変更しながら接続を維持する．一方Javascriptが無効なブラウザは，関数を実行せずに図3.11の<meta>タグで記述されたURLにリクエストを出す．このときリクエストするnojsauth.htmlにも同様の<meta>タグ記述が存在するため，ブラウザは定期的に同じセッションIDをXFWに送信して接続を維持する．

図3.10と図3.11の例では，3.3.1.5項で述べたKeepAlive Checkerがnat.foo.com の8000番ポートで処理をするが，次に述べる管理者が設定するパラメータによってKeepAlive Checkerの動作を変更可能である．

3.3.3.3　管理者が設定するパラメータ

表3.1にXFWを動作させるために管理者が設定するパラメータを示す．管理者は実行する環境に合わせてこれらのパラメータを設定する．特にXFWの動作を決定するのがKEEPALIVE_INTERVAL，CLOSE_INTERVALであり，KEEPALIVE_INTERVALの値が小さければ小さいほど，正規端末のブラウザは頻繁にセッションIDを送信するようになり，負荷が増大する．また，CLOSE_INTERVALは必ずKEEPALIVE_INTERVALよりも大きい値に設定する必要があり，さもなければ接続中の端末はすべて切断されてしまう．RADIUS_INFOに関しては，アドレス，キー，ポートをコロン(:)区切りにし，さらにその組合わせをセミコロン(;)区切りにすることによって，複数のRadiusサーバと連携することが可能である．その場合XFWは，先頭に指定されているサーバから順に認証を始め，どこかのサーバで認証が成功した時点で認証成功と判断する．

表 3.1 管理者が設定するパラメータ

設定するパラメータ名	パラメータの意味
HTTP_PORT	HttpRedirectorが 使用するポート
HTTPS_PORT	Authentication Serverが 使用するポート
KEEPALIVE_PORT	KeepAlive Checkerが 使用するポート
KEEPALIVE_INTERVAL	クライアントがセッションID を送信する間隔(秒)
CLOSE_INTERVAL	KeepAlive Checkerが 切断したと判断する間隔(秒)
CHECK_INTERVAL	KeepAlive Checkerが ポーリングチェックする間隔(秒)
RADIUS_INFO	Radiusサーバのアドレス, ポートと認証のためのキー

3.4　運用と評価

XFWの基本性能を示すために，XFW単体での負荷テストを行った．また現在大学のキャンパスでXFWを導入し，無線と有線のネットワーク接続サービスを運用している．次にそれぞれについて述べる．

3.4.1　XFW単体の負荷テスト

XFW単体での性能を測定するため，我々はWeb Application Stress Tool[48] (以下，Stress Toolと呼ぶ) を使って負荷テストを行った．XFWの方式では，接続するクライアントが増えた場合，最も性能に影響が出るのがセッションIDを処理する部分(以下，接続維持部分と呼ぶ)であるため， XFWへの同時接続数を10〜100まで10ずつ増やし，接続維持部分で１秒間あたりにXFWが処理するリクエスト数を測定する．3.3.3.2項で述べたように，XFWではJavascriptが無効なブラウザを使用された場合でも，Javascriptの動作判定を行ったあとセッションIDを固定して端末を接続させることができるため，測定にはセッションIDを毎回変更する場合と固定の場合の2種類で行う．また，比較対象として，最も処理が軽いHTTPRedirectorがリダイレクトする部分(以下，リダイレクト部分と呼ぶ)と，最も処理が重いSSL[49]を使った認証部分でも同様に1秒間あたりのリクエスト処理数を測定する．使用するコンテンツの大きさは接続維持部分で298(byte)，認証部分で2.42(kbyte)である．表3.2にテスト環境を示し，図3.12に測定結果を示す．

図3.12を見ると，同時接続数が30〜100までの間はリダイレクト部分も接続維持

表 3.2 テスト環境

CPU	Pentium4 2.8(GHz)
メモリ	1 (Gbyte)
ネットワーク帯域	1000(Mbps)
Java VM	J2SDK-1.4.2_04

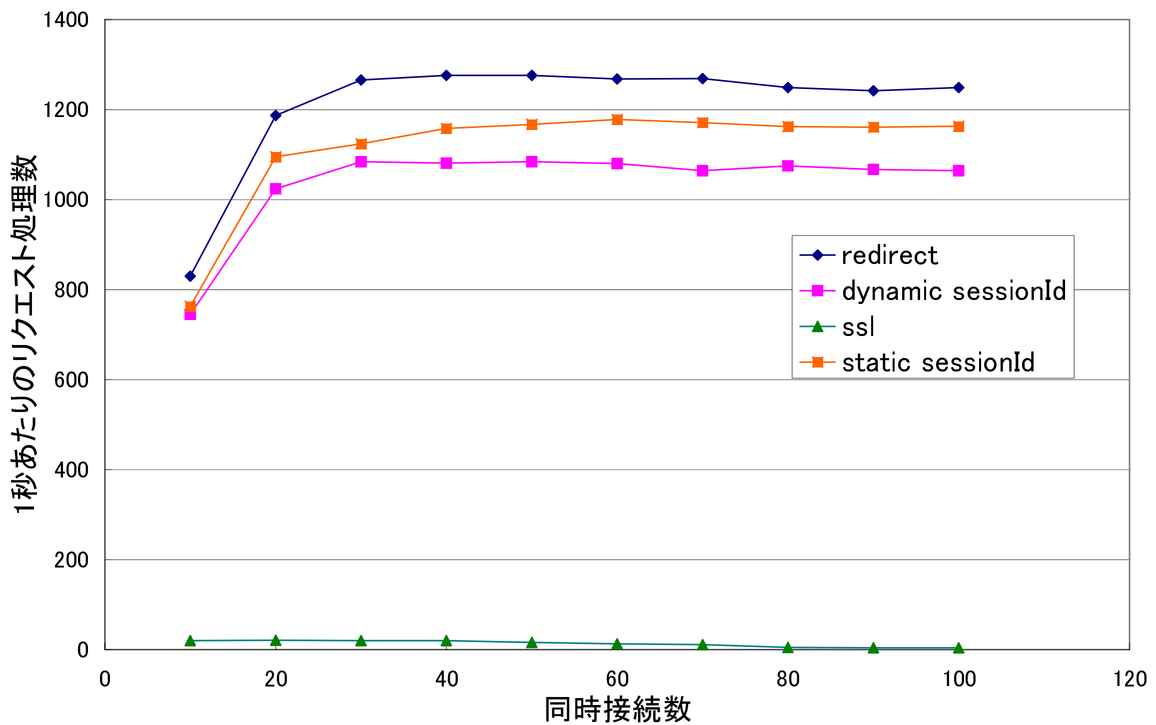

図 3.12 1秒あたりのリクエスト処理数

部分もそれほど変化が見られない．つまり，テスト環境においてXFWが1秒間に処理できるリクエスト数はリダイレクト部分で約1,260，接続維持部分ではJavascript対応のためセッションIDを固定した場合約1,160，毎回変更した場合約1,070である．セッションIDを毎回変更する場合，固定した場合と比較すると約10%，リダイレクト部分と比較すると約15%性能が低下する．しかし仮に10秒間隔でセッションIDをやり取りする場合，理論上約10,000ユーザが接続可能であり，実用上問題ない．また，同時接続数が10〜30までの間は，表3.2で示したテスト環境ではCPU使用率が飽和(100%にならない)しないため，同時接続数が増えるにしたがって1秒間に処理できるリクエスト数も増加していく．

一方，認証部分では同時接続数が40までは毎秒約20リクエストを処理できているが，50以降は接続エラーが頻発し，最終的に同時接続数100の場合は毎秒4リクエストしか処理できなかった．接続維持部分と認証部分のコンテンツサイズを考慮しても，この違いはSSLの処理コストであると考えられる．ただし，認証処理は接続維持と異なり接続時に1度だけの処理のため，ユーザがランダムに利用を開始する現実の環境においては実用上の問題は起こらない．仮に問題が起きた場合には複数のXFWで負荷分散すればよい．なお，現実の環境での同時接続数は，3.3.3.3項で述べたように表3.1のKEEPALIVE_INTERVALの値を小さくするほど増加する．

3.4.2 大学での運用

大学では表3.3で示す仕様のサーバに，表3.4で示すOSやソフトウェアを使ってXFWを導入し運用を行っており，Windows，MacOS，Unix互換OSのクライアントで動作することを確認している．また，運用実績として1日あたりの最大同時接続数と延べユーザ数を1年間測定し，それぞれの推移を図3.13と図3.14に示す．そして，それぞれの最大値を表3.5に示す．

XFWはシステムのログとして，端末の接続と切断時にログインID，使用するIPアドレス，MACアドレス，時刻を記録するため，iptablesのログとあわせて不正利用の発見やトラブルに対応している．

3.4.3 評価

ここでは，セキュリティ，性能，ユーザの利用方法，導入コストの点からXFWを評価する．そしてXFWの問題について述べる．

3.4.3.1 セキュリティ

XFWはIPアドレスとMACアドレスに加え，毎回異なるセッションIDを使うことで正規端末を識別する．したがって，IPまたはMACアドレスを偽造された場合でも，不正端末のネットワーク接続を防ぐことができる．しかし，不正端末であってもIPアドレスを取得することはできるため，プライベートネットワーク内部での不正アクセスは防ぐことができない．これを防ぐためには無線LANアクセスポイン

表 3.3 運用サーバの仕様

CPU	Pentium4 2.8(GHz)
メモリ	1 (Gbyte)
ネットワーク帯域(global)	1000(Mbps)
ネットワーク帯域(private)	1000(Mbps)

表 3.4 XFWで使用するソフトウェア

実行に必要な設定項目	使用したソフトウェア
Operating System	Linux(Fedora core3)
NAT	iptables
Firewall	iptables
DHCP	ISC-DHCP
Authentication Server	OpenRadius
Java VM	J2SDK-1.4.2_08

表 3.5 延べユーザ数と同時接続数の最大値

延べユーザ数(人/日)	1,541
同時接続ユーザ(人)	186

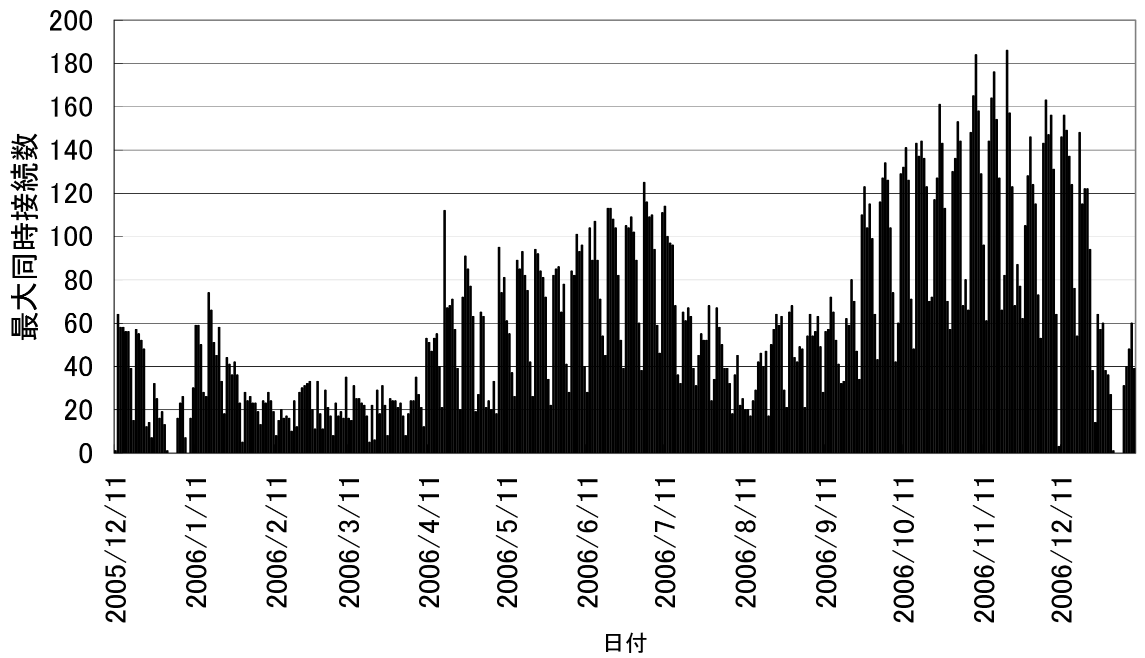

図 3.13 同時接続数の推移

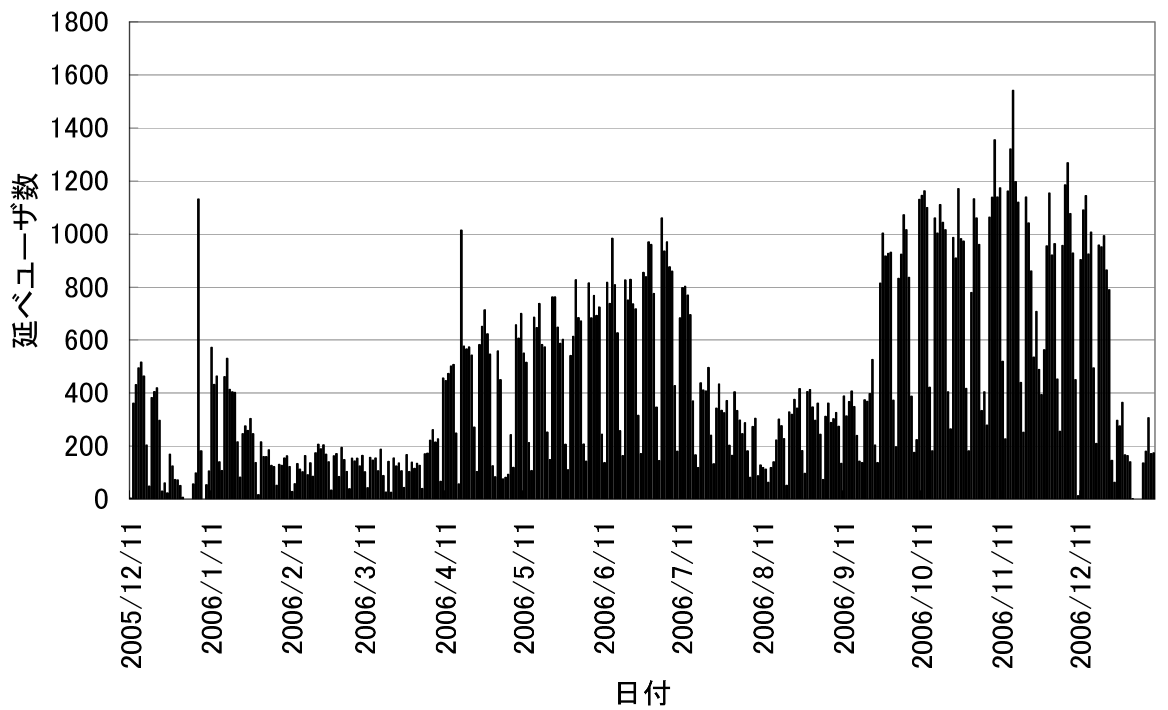

図 3.14 延べユーザ数の推移

ト，あるいはネットワークスイッチレベルでのアクセス制限が必要でり，そのようなアクセス制限にXFWは適さない．また，3.3.3.2項で述べたように，Javascript が無効な場合，IPアドレスとMACアドレスを偽造され，セッションIDを盗まれると不正に接続できてしまう．したがって，Javascriptが無効である場合の接続を許可するかどうかは，利用する環境に応じて管理者が決定できる仕組みにする必要がある．

3.4.3.2　性能

接続中の端末が送信するパケットは，セッションIDの送受信を除きXFWを経由せずNAT BOXあるいはルータでルーティングされる．したがって3.4.1節で述べたように，接続する端末が増えた場合XFWにかかる負荷は接続維持部分の処理だけであり，図3.12の結果から実用上問題ない．そして図3.13と図3.14，および表3.5で示したように，実際に大学で運用を行っているが，測定開始当時と現在の最大同時接続数，および延べユーザ数を比較すると，ともに2.5倍から3倍程度増加しているが，性能の低下は見受けられない．仮に負荷が高く性能が低下する場合には，表3.1で示したKEEPALIVE_INTERVALの値を増やすことで負荷を減らすことができる．しかしその場合，端末が使用を止めてから最長で設定した秒数だけそこで使用していたIPアドレスからのパケットは通過してしまう．そのため，その間に偶然同じIPアドレスを取得した他の端末は，認証せずともネットワークを利用できてしまうことになる．したがって，負荷が高くなりKEEPALIVE_INTERVAL の値を増やす必要がある場合は，複数のネットワークとサーバを使用して分散処理をするほうが安全である．一方，接続を維持するための正規端末にかかる負荷は，管理者が設定した間隔ごとにJavascriptでMD5ハッシュ関数を一回実行するだけであり，その負荷は無視できるほど小さい．

3.4.3.3　ユーザの利用方法

XFWは端末のHTTPリクエスト送信を契機にユーザ認証を行い，正規端末を特定したネットワーク接続を可能にする．この処理に必要なものはブラウザだけであり，最近の端末には標準でインストールされている．そして認証処理はウェブインタフェースを使用し，HTTPSで通信するため，安全でわかりやすく，初心者でも十分に利用することができる．

3.4.3.4　導入コスト

XFW自体はJavaで実装されたソフトウェアであるため，Javaの実行環境があればOSに依存しない動作が保障される．したがって3.2.1節で示したように，動作させるOSで利用できるFirewallソフトウェアにHTTPリクエストをXFWにフォワードできる機能があり，XFWからFirewallのルールを変更できさえすれば導入可能である．実際にFreeBSD上でipfwを利用して動作を確認している．したがって，新規導入はもちろん，既存のNAT BOXに組み込むことも十分可能である．なお，本論文

ではXFWをNAT BOXに組み込んでいるが，NAT変換は必須ではないのでルータに組み込むことも可能である．

3.4.3.5　XFWの問題点

XFWは3.2.3節で述べた方法で毎回異なるセッションIDを用いてアドレス偽造を検知し，不正利用を防ぐことができる．しかし，悪意のあるユーザがIPアドレスとMACアドレスを偽装し，不正なセッションIDを送ることで接続中の正規端末を切断できてしまう．これに対しては正規端末が送るセッションIDがXFWが指定する間隔で送られることを利用し，不正な時刻に送られるセッションIDを無効とすることである程度防ぐことができるが，XFWの仕組みでは本質的に正規端末からのセッションIDと悪意のあるセッションIDを正確に区別することはできない．

3.4.3.6　Opengateとの比較

Opengate[12]では，XFWと同様にブラウザだけを利用した認証方法を提供しており，認証時にブラウザにダウンロードされるJavaアプレットが，サーバとのTCPコネクションを確立し，サーバがそれを監視することによって端末の切断を検知する．したがって，XFWと同様にIPやMAC アドレスを偽造したとしても不正利用することはできない．しかし，この仕組みにはJavaが動作するブラウザを使用する必要があり，Javaが実装されていないPDAなどの携帯端末では利用することができない．XFWではJavascriptを使用しているが，PDAなどの携帯端末にあらかじめインストールされているブラウザでも動作を確認しているため適用範囲が広い．また，OpengateではJavaアプレットが実行できないクライアントに対し，認証後にユーザが明示的に利用時間を指定して接続を維持する仕組みが用意されている．しかし，毎回時間を指定することはユーザにとって少なからず手間がかかり，設定する時間によってはユーザが利用を止めたあと偶然同じＩＰアドレスを取得した端末が認証をうけずに長い時間接続できてしまう．XFWでは，Javascriptが無効であってもXFW自身がそれを判断し，単純なHTTPだけで接続を維持するため，ユーザは接続維持や切断に関して特に意識する必要は無い．そしていずれの場合にも，正規のユーザが利用を止めたあと，同じIPアドレスを取得した別のユーザが利用できる時間は，最大でも管理者が設定した時間までである．

3.5　まとめ

本章では，多数のユーザを対象にネットワーク接続を提供する場合，利用するユーザを認証してアクセス制限を行うシステムであるXFWの提案と実装を行い，負荷テストや実際に導入して評価することで有効性を示した．XFWでは利用するユーザの利便性を重視し，一般的なブラウザだけでユーザ認証，およびIPアドレスやMACアドレス偽造による不正利用防止などの処理をすべて行うことができる．また，認証画面や認証後の画面をカスタマイズすることでユーザに必要な情報を

通知することができ，ユーザサポートのコストも軽減できると考えられる．ただし，XFWでは認証後に端末がやり取りするデータに関して暗号化などは行わないため，情報漏えいを防ぐためにはSSL/TLS[50]などの暗号化技術を別途導入する必要がある．また，オープンスペースでの無線LAN接続サービスを展開する場合には偽基地局による盗聴の問題もあり，これを防ぐには基地局と端末間で相互認証するなどの対策が必要である．

なお，XFWではプライベートネットワーク内でのウィルス感染や，ウィルスを出すクライアントを強制的に切断することができない．これらに関しては，VLAN等を導入し，一台のサーバで複数のプライベートネットワークを使用することで，ウィルスが広範囲に蔓延することをある程度防ぐことができる．また，Firewallのログを解析し，不正アクセスをしているクライアントを強制切断する仕組みが必要である．

第4章　オンデマンドローディングを支援するフレームワーク

4.1　概要

作業のシステム化に伴い，我々の日常的な作業は目的に応じたさまざまなアプリケーションを使用することでより効果的に処理できるようになった．また，ネットワーク環境も整備され，我々はいたるところでネットワークに接続することができる．しかし，多くのアプリケーションは，データの送受信などそれ自身がネットワークを使用することはあるものの，インストールと実行をネットワークを介してシームレスに行うことを前提としていない．そのため，ユーザはあらかじめ使用するアプリケーションを端末にインストールしておく必要がある．

また，現在のアプリケーションは多機能化し，ほとんどの場合ユーザが一度に使用するのは一部の機能に限られる[51][52]．しかし多くのアプリケーションは，実行に必要なプログラムあるいはライブラリが端末にすべてインストールされている必要がある．また，機能の増加に伴い実行に必要なプログラムやライブラリは増大せざるを得ない．

そこで本章では，アプリケーションの起動を契機に，個々のユーザが実際に利用する機能だけでアプリケーションを実行する方式を提案し，それを実現するためのフレームワーク，SmartMobileについて述べる．SmartMobileでは，開発用APIと実行環境を提供し，アプリケーションの実行とインストールをシームレスに行うとともに，ユーザの指示で必要な機能をオンデマンドにロード（以下，オンデマンドローディングと呼ぶ）する．本章では，導入例として一般的なユーザが最も使用するGUIアプリケーションを対象とし，Javaの動的なクラスローディング機構を拡張することによってオンデマンドローディングを実現していく．

2.2.3節で述べたように，Javaではネットワークを介してアプリケーションを配布するJava Web Startがあるが，配布されるのは実行に必要なクラスファイルをまとめたJARファイルであり，そこにはユーザが実際には使用しない機能のクラスファイルも含まれる．SmartMobileでは，ユーザが必要な機能のクラスファイルだけをオンデマンドにロードするため，効率よくリソースを使用するアプリケーションの開発と実行を可能にする．

以下4.2節では，GUIアプリケーションの特徴と，SmartMobileで注目する点について述べ，実行時の問題点について述べる．次に，4.3節ではSmartMobileのアプローチを述べ，4.4節で具体的な実装と動作について述べる．そして4.5節では実際のアプリケーションを構築して評価し，4.7節でまとめを述べる．

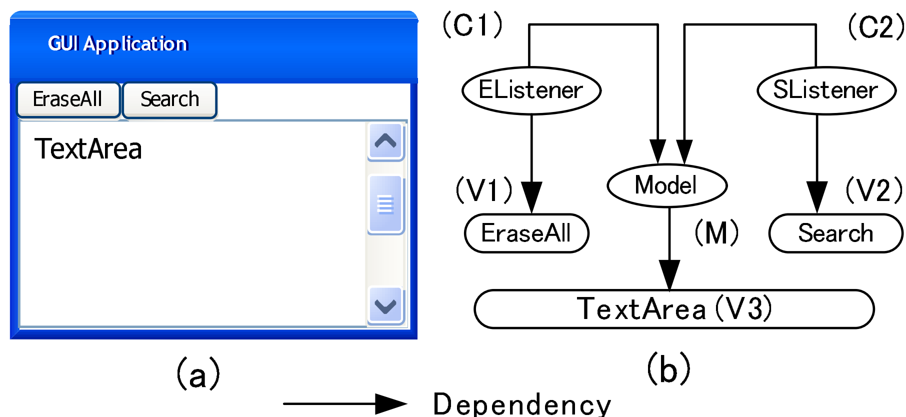

図 4.1 GUIアプリケーションとMVCモデル

4.2　GUIアプリケーションの特徴

本節では，GUIアプリケーションの処理手順を示し，SmartMobileで注目した個々の機能とオブジェクトの関係について述べる．そして，Javaによる開発および実行時の問題点を述べる．

4.2.1　処理の手順

図4.1に一般的なGUIアプリケーションの外観(a)と，それを構成するオブジェクトの構造(b)を示す．GUIアプリケーションはイベント駆動型のアプリケーションである．つまり，ユーザがイベントソースオブジェクト(以下ソースオブジェクトと呼ぶ)を操作し，そこで発生するイベントをイベントリスナーオブジェクト(以下リスナーオブジェクトと呼ぶ)が処理する．また，図4.1(b)に示すように，GUIアプリケーションはMVCモデルで構築されることが一般的であり，イベント処理の流れをMVCモデルに当てはめると，V1で発生したイベントをC1が処理し，Mを変更することによって対応するV3が変化する．

4.2.2　機能とリスナーオブジェクトの関係

アプリケーションが図4.1(b)で示す構造の場合，個々の機能はリスナーオブジェクト(C1やC2)に実装される．そして，C1で実現する機能がそれ自身で完結しC2に影響を与えない設計であれば，処理結果を反映させるためにMへの参照があればよく(C1はMに依存する)，C2の参照は不要である．また，C1はV1のイベントを処理するため，V1の存在は必須であるが，V1は実際にイベントが発生するまでC1およびそのクラスファイルも不要である(C1はV1に依存するが，V1はC1に依存しない)．このように，リスナーオブジェクトとその他のオブジェクトの間には，プログラム上に現れない依存の方向が存在する．本論文では，これをオブジェクトの依存

```
TextFrame tframe= new TextFrame ();
TextModel model = new TextModel ();
JButton button= new JButton (“doit”);       (a)
tframe.setModel (model);
tframe.add (button);

TextListener listener = new TextListener ();
listener.setModel (model);                  (b)
button.addActionListener (listener);
```

図 4.2 Javaの標準APIを使用したイニシャライズ部分の実装

関係と呼ぶ．

4.2.3　開発と実行時の問題点

JavaでGUIアプリケーションを開発する場合，図4.2に示すようにソースオブジェクトやリスナーオブジェクト，モデルオブジェクトを作成し，それらを結合する実装を行うイニシャライズ部分がある．図4.2(a)ではソースオブジェクトやモデルオブジェクト，図4.2(b)でリスナーオブジェクトの生成を行っている．

通常図4.2のプログラムをJava Virtual Machine(以下JavaVMと呼ぶ)が実行するとき，図4.2(b)で生成するTextListenerのクラスファイルが存在しないとエラーになるため，実行する端末には必ずそのクラスファイルが存在している必要がある．このため，ユーザが実際に必要ないリスナーオブジェクトのクラスファイルであっても，実行する端末にあらかじめ配置しておかなければならない．

4.3　アプローチ

4.2.3節の問題に対し，本研究ではユーザの指示によって必要なクラスファイルをネットワークを介して取得し，図4.2(b)で示したリスナーオブジェクトの生成と関連するメソッドを実行する方式を考案した．これを実現するには次の機能をもつローディング機構が必要である．

- インタフェース機能
  ユーザが必要なリスナーオブジェクトを指定し，生成を指示するためのインタフェースを提供する機能．

- オンデマンド生成機能
  ユーザがリスナーオブジェクトの生成を指示したとき，必要なクラスファイルを取得して，実際にオブジェクトを生成する機能．

開発者は，アプリケーション本来の機能に加え，ローディング機構を実装する必要がある．しかし，ローディング機構の実装は煩わしく，開発者の負担となる．

そこでSmartMobileでは，このようなアプリケーションの開発と実行を支援するために，SmartAPIと呼ぶAPIと，ランタイムと呼ぶ実行環境を提供する．ランタイムはオブジェクトの依存関係を利用してローディング機構を実現する．したがって，開発者はアプリケーション本来の機能を実装し，SmartAPIを用いてイニシャライズ部分の実装を行えばよい．次にオブジェクトの依存関係が果たす役割について述べ，SmartAPIの使用法について述べる．そして，ローディング機構の実現法について述べる．

4.3.1　依存関係の役割

ランタイムがオンデマンドにオブジェクトを生成するとき，そのオブジェクトが依存するオブジェクトは事前に生成しておく必要がある．例えば図4.1(b)では，4.2.2節で述べたように，C1を生成する前にそれが依存するV1とMは存在していなければならない．しかし，仮にその3つをユーザの指示でオンデマンドに生成する場合，V1とMを生成する前にC1を生成してもC1の機能を利用することができない．これを防ぐために，オブジェクトの依存関係を定義することによって，ランタイムはC1を生成する前にV1やMの存在を確認し，存在しない場合は生成する．

4.3.2　SmartAPIの使用法

表4.1に示すように，SmartAPIではオブジェクトの生成に“snew”と“dnew”の2種類，メソッドの実行に“<ADM>”と“<DFM>”の2種類，そして依存関係の作成に“<DEPENDS>”というAPIを提供している．開発者はこれらを使用してオブジェクトの生成時期(アプリケーション起動時またはオンデマンド)や，オブジェクトの依存関係を定義する．4.4.2節で述べるように，SmartAPIを利用したイニシャライズ部分は，実行前に通常のJavaコードに変換される．この変換処理のために，SmartAPIを使用したイニシャライズ部分は，パッケージ名まで含めてクラスを指定する必要がある．なお，SmartAPIを適用する部分はイニシャライズ部分だけでよく，各クラスの記述にはJava標準のAPI を使用すればよい．次に，図4.2をSmartAPIで書き換

表 4.1 SmartAPI

SmartAPI	APIの意味と用法
snew	起動時に生成するオブジェクトに使用
dnew	ユーザの指示で生成するオブジェクトに使用
<ADM>	依存関係が発生するメソッドの実行に使用
<DFM>	通常のメソッド実行に使用
<DEPENDS>	依存関係の作成に使用

```
foo.bar.TextFrame tframe = snew foo.bar.TextFrame ();
foo.bar.TextModel model = snew foo.bar.TextModel ();
javax.swing.JButton button = snew javax.swing.JButton (“doit”);
tframe <ADM>.setModel (model);
tframe <ADM>.add (button);

foo.bar.TextListener listener = dnew foo.bar.TextListener ();
listener <DFM>.setModel (model);
button <ADM>.addActionListener (listener);

listener<DEPENDS>model;
```

図 4.3 SmartAPIを使用したイニシャライズ部分の実装

えて図4.3に示し，それぞれのAPIについて述べる．

4.3.2.1　オブジェクトの生成に関するAPI

Javaの標準APIでオブジェクトの生成を行う場合，図4.2のようにnewを使用するが，SmartAPIではsnewまたはdnewどちらかを使用して記述する．このイニシャライズ部分は，通常のJavaコードに変換後ランタイムが実行する．その際ランタイムは，開発者がsnewを使って記述したオブジェクトだけ生成し，dnewを使って記述したオブジェクトは，アプリケーション実行中要求に応じてに生成する．

4.3.2.2　メソッドの実行に関するAPI

メソッドの実行はオブジェクトの依存関係と深く関係する．2つのオブジェクト(仮にObjAとObjBとする) 間に依存関係が存在するということは，ObjAがObjB

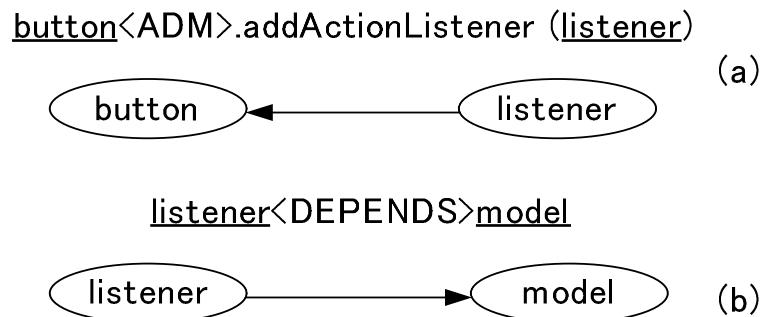

図 4.4 SmartAPIを使った依存関係の作成

の参照を保持する，またはObjBがObjAの参照を保持するということである．そして，例えばObjAにObjBの参照を保持させる場合，ObjAのクラスに定義された“setObjB”などのメソッドを実行する必要がある．しかし，メソッド呼び出しの記述を読み込んでも，ランタイムはどのメソッドの実行がオブジェクトの参照を保持するのかを知ることはできない．そこでSmartMobileでは，イニシャライズ部分で依存関係が発生するメソッド呼び出しを記述するとき，開発者が明示的にAPIを用いて依存関係を定義する必要がある(4.3.2.3項で述べるAPIとは用法が異なる)．図4.3(2)に示すように，メソッドの呼出しに<ADM>が追加された場合，引数のオブジェクトとの間に依存関係があることを示す．例えば図4.3の(*)では，図4.4 (a)に示すように，“listenerはbuttonに依存する”という意味になる．

一方，4.4.2節で述べる変換処理のため，依存関係が発生しないメソッド呼び出しには<DFM>を追加する必要がある．つまり，イニシャライズ部分のメソッド呼び出しを記述する際，開発者は<DFM>か<ADM>どちらかを必ず追加しなければならない．

SmartMobileのランタイムはこれらの記述をもとに，アプリケーション実行時に各オブジェクトの依存関係を認識する．

4.3.2.3　依存関係の作成に関するAPI

メソッドの実行を利用しない依存関係の作成には<DEPENDS>を使用する．図4.3(3)では，処理結果を反映するためにmodelの参照をlistenerに保持させている．したがって“listenerはmodelに依存する”が，仮に図4.3(3)の記述を<ADM>で置き換えると“modelはlistenerに依存する”という意味になり，依存する方向が反対になるため，ここで<ADM>を使用することはできない．また，4.3.2.2項で述べたように，<DFM>では依存関係を定義することはできない．

そこで，表4.1で示す<DEPENDS>を利用して依存関係を定義する．この例の場合，図4.3(4)のように記述することで，図4.4(b)に示すように，“listenerがmodelに依存する”，という意味になる．なお，すべてのメソッド呼び出しに<DFM>を追加し，<DEPENDS> だけを利用してすべての依存関係を定義することも可能であるが，メソッドの実行と依存関係の作成を別々に記述する必要があり，記述量が増える．

4.3.3　ローディング機構の実現法

図4.3のイニシャライズ部分を例にしたローディング機構の実現法を図4.5に示し，インタフェース機能と，オンデマンド生成機能の実現法について述べる．

4.3.3.1　インタフェース機能の実現法

4.2.1節で述べたように，ユーザは通常ソースオブジェクトを操作して機能を実行するため，リスナーオブジェクトの生成を指示する場合も，同様のGUIを用いるほうが分かりやすい．そこでSmartMobileでは，ソースオブジェクトを使ってユーザ

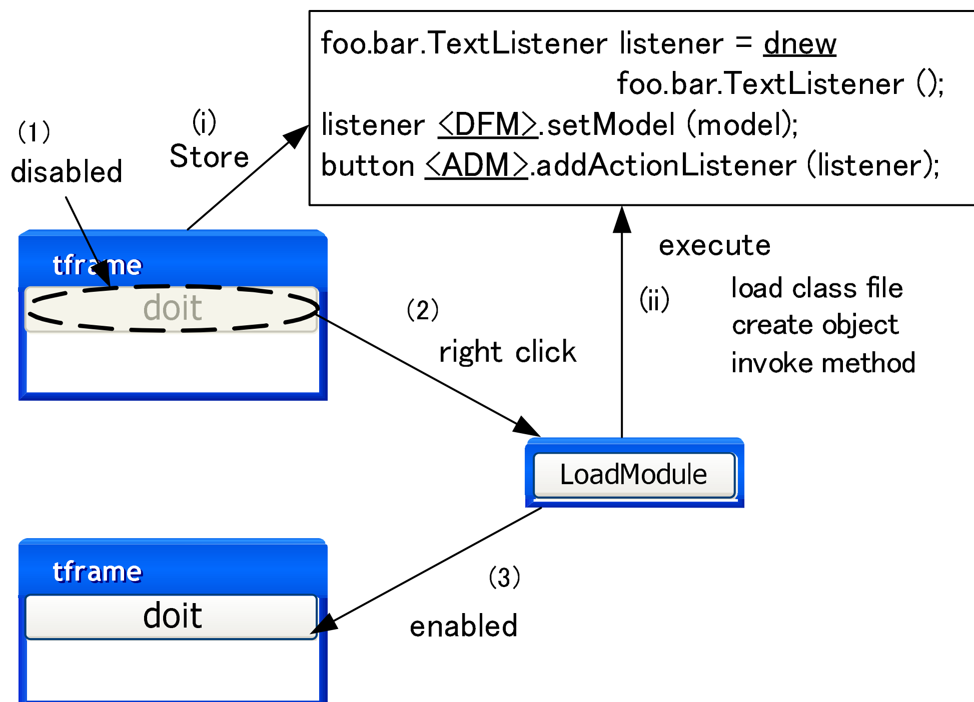

図 4.5 ローディング機構の実現法

にリスナーオブジェクトの生成を指示させる．

4.3.2.1項で述べたように，ランタイムはイニシャライズ部分を実行するとき，snewで記述されたオブジェクトの生成や，それに関するメソッドの実行および依存関係の認識を行うが，dnewで記述されたオブジェクトに関しては，依存関係の認識だけしか行わない．したがって図4.3(b)では，ランタイムはlistenerとbuttonの依存関係を認識するだけでlistenerの生成は行わず，それ(listener)が依存するbuttonを無効化(グレーアウト)する(図4.5(1))．

その後アプリケーション実行中，無効化したbutton上でユーザが右クリックしたとき，ランタイムはlistenerを生成するためのボタン(以下ロードボタンと呼ぶ)を表示する(図4.5(2))．そして，4.3.3.2項で述べる方法でlistenerを生成したあと，buttonを有効化する(図4.5(3))．そのため，ソースオブジェクトの生成はかならずsnewを用いて記述する必要がある．

4.3.3.2　オンデマンド生成機能の実現法

ランタイムはイニシャライズ部分を実行するとき，dnewで記述されたオブジェクトの生成や関連するメソッドを実行するための情報を記録しておく(図4.5(i))．そして，ユーザがロードボタンをクリックすると，記録した情報をもとに，listenerの生成と関連するメソッドを実行する(図4.5(ii))．

また，ランタイムはオブジェクトを生成するとき，必要なクラスファイルが存在

しない場合はネットワークを介して取得するため，実行する端末にはランタイムだけがあればよい．クラスファイルの取得先やアプリケーションの実行環境については4.5.2節で述べる．

4.4　ランタイムの実装と動作

ランタイムは，イニシャライズ部分を実行するとき，オブジェクトの生成やメソッドの実行を行うために，JavaのリフレクションAPI(以下リフレクションAPIと呼ぶ)[53][54]を使用する．本節では，はじめにリフレクションAPIについて述べる．次に，変換後のJavaコードとリフレクションAPIの関係について述べる．最後に，ランタイムの実行時の動作について述べる．以後説明のため，オブジェクトとは，クラスからインスタンス化した実体を指し，クラスを表現する場合には“クラス名”クラスと表記する．

4.4.1　リフレクションAPI

一般的なコンパイル型言語では，コンパイルした時点で判明している情報だけで動作する．そして，Javaはコンパイル型言語であるため，通常はコンパイル時に判明しているクラスだけで動作する．4.3節で述べたように，SmartMobileでは開発者が記述したイニシャライズ部分をランタイムが実行する必要があるが，Javaの標準APIだけではランタイムのコンパイル時に判明していないイニシャライズ部分を実行することはできない．そこでSmartMobileでは，リフレクションAPIを使用してランタイムがイニシャライズ部分を実行する．リフレクションAPIは，クラスの構成要素を問い合わせるだけでなく，クラス名やメソッド名といった文字列情報とクラスの型情報だけで実行時にクラスを動的にロードおよびインスタンス化することができ，さらにメソッドを呼び出すこともできる．実際に変換ツールは，イニシャライズ部分をリフレクションAPIを使用した通常のJavaコードに書き換える．

表 4.2 SmartAPIと変換後のクラス

SmartAPI	変換後のクラスまたはメソッド
snew	StaticInstance
dnew	DynamicInstance
<ADM>	AddMethodCall
<DFM>	MethodCall
<DEPENDS>	ObjectId.addDependId

```
Message tframeMessage = new StaticInstance (“foo.bar.TextFrame”);
ObjectId tframeId = RuntimeUtil.newInstance (tframeMessage);
Message modelMessage = new StaticInstance (“foo.bar.TextModel”);
ObjectId modelId = RuntimeUtil.newInstance (modelMessage);
Message buttonMessage = new StaticInstance (“javax.swing.JButton”);
buttonMessage.setArg (“doit”);
ObjectId buttonId = RuntimeUtil.newInstance (buttonMessage); ──▶(1)      (a)
Message addtframesetModelMessage
                          = new AddMethodCall (“setModel”);
addtframesetModelMessage.setArg (modelId);
RuntimeUtil.invoke (addtframesetModelMessage, tframeId); ──────▶(2)
Message addtframeaddMessage = new AddMethodCall (“add”);
addtframeaddMessage.setArg (buttonId);
RuntimeUtil.invoke (addtframeaddMessage, tframeId);

Message listenerMessage =
          new DynamicInstance (“foo.bar.TextListener”);──────▶(3)
ObjectId listenerId = RuntimeUtil.newInstance (listenerMessage);
Message listenersetModelMessage = new MethodCall (“setModel”);
listenersetModelMessage.setArg(modelId);                                    (b)
RuntimeUtil.invoke (listenersetModelMessage, listenerId);
Message addbuttonaddActionListenerMessage =
                     new AddMethodCall (“addActionListener”);
addbuttonaddActionListenerMessage.setArg (listenerId);
RuntimeUtil.invoke(addbuttonaddActionListenerMessage, buttonId);

listenerId.addDependId(modelId);
```

図 4.6 変換後のJavaコード

4.4.2 変換後のJavaコード

SmartAPIは，表4.2に示すクラスまたはメソッドを用いた通常のJavaコードに変換される．表4.2で示すクラスは，ObjectIdクラス以外すべてMessageクラスのサブクラスであり，Messageクラスはリフレクションの APIのラッパークラスとなっている．

図4.6は図4.3で示したイニシャライズ部分を変換した後のJavaコードである．Messageクラス(表4.2で示したサブクラスを含む)は，コンストラクタの引数に生成するクラス名，または実行するメソッド名をとる．また，オブジェクトの生成やメソッドの実行に必要な引数のために，setArgメソッドを備えている．オブジェクトの生成には，RuntimeUtilクラスのnewInstance(Message msg)メソッドが使用され(図4.6(1))，生成するオブジェクトの参照であるObjectIdオブジェクト(以下ObjectIdとする)が返り値となる．また，メソッドの実行にはRuntimeUtilクラスのinvoke(Message msg,

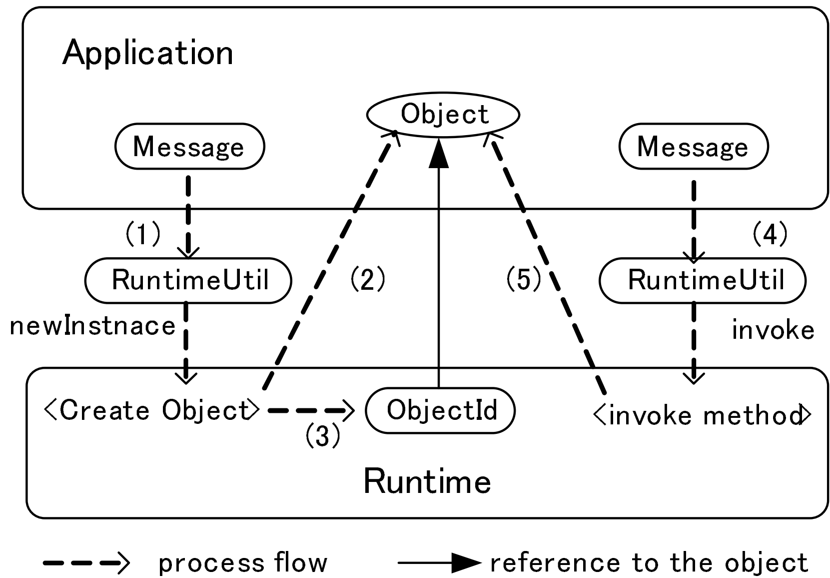

図 4.7 オブジェクトの生成とメソッドの実行法

ObjectId oid)メソッドが使用される(図4.6(2))．ObjectIdクラスは，他のObjectIdと依存関係を作成するためのaddDependId(ObjectId oid)メソッドを備えており，表4.2に示すように<DEPENDS> はaddDependIdメソッドとObjectIdを使って変換される．

4.4.3　ランタイムの動作

実際にランタイムが実行するのは，図4.6で示すイニシャライズ部分のコードである．次に，ランタイムが図4.6のコードを実行するときの動作と，オンデマンドにオブジェクトを生成するときの動作を述べる．

4.4.3.1　イニシャライズ部分の実行

ランタイムは，図4.7に示す手順で図4.6の(a)を実行する．ランタイムは，RuntimeUtilのnewInstance(図4.7(1))でオブジェクトを生成し(図4.7(2))，その参照であるObjectIdを生成する(図4.7(3))．また，invoke(図4.7(4))でメソッドを実行する(図4.7(5))．

一方，図4.6の(b)を実行するとき，図4.6(3)で示すTextListenerオブジェクトの生成にDynamicInstanceクラスが使用されているため，図4.7(2)で示すオブジェクトの生成は行わずに，DynamicInstanceオブジェクトを保持し，図4.7(3)で示すObjectId(この場合listenerId)だけを生成する．同様に，生成していないTextListenerオブジェクトに関連するメソッドも，実行(図4.7(5))せずに，MethodCallやAddMethodCallオブジェクトを保持する．

また，ランタイムはAddMethodCallクラスやaddDependIdメソッドをもとに，生成したObjectId を使用して図4.8の(DP)で示す依存関係を作成する．そして，未生成

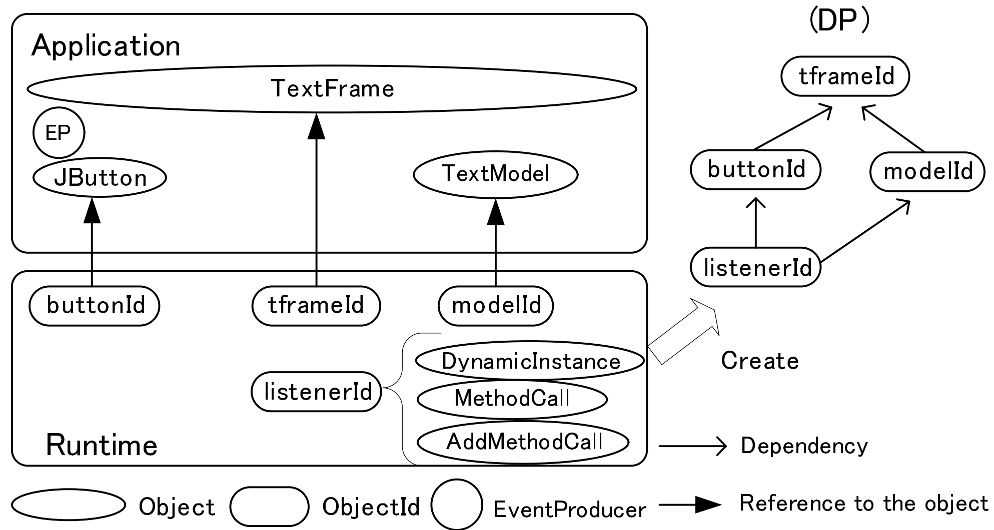

図 4.8 アプリケーションとランタイムの状態

のTextListenerオブジェクトが結合していないソースオブジェクト(この場合JButtonオブジェクト)に，図4.9で示すEventProducer を付加し，setEnabled(false) メソッドを実行して無効化する．EventProducerは，ソースオブジェクトに発生するイベントをランタイムに通知する役割を果たす.

この結果図4.8で示すように，ランタイムは4つのObjectIdとオブジェクトの依存関係，そしてTextListenerオブジェクトを生成するためのDynamicInstance，および関連するMethodCallとAddMethodCallオブジェクトを保持する.

4.4.3.2　オンデマンドなオブジェクト生成

アプリケーションを実行中，ユーザが無効状態のJButtonオブジェクト上で右クリックすると，EventProducerがランタイムに右クリックイベントを通知する．ランタイムは図4.8の(DP)で示した依存関係をもとに，JButtonオブジェクトに結合する

```
public class EventProducer implements ActionLister,
              MouseListener, KeyListener, Serializable{

   public void mousePressed(MouseEvent e){
        //catch right click event
        ......................
   }
}
```

図 4.9 EventProducerクラスの定義

TextListenerオブジェクトの存在を調べ，生成していなければロードボタンを表示する．そして，ユーザがロードボタンをクリックすると，ランタイムはイニシャライズ部分実行時に保持したDynamicInstanceオブジェクトなどを使用して図4.6の(b)に示したTextListenerの生成と関連するメソッドを実行し，JButtonオブジェクトのsetEnabled(true)メソッドを実行して有効化する．その結果，TextListener の機能が利用可能になる．

4.5　アプリケーションの作成と実行

本節ではSmartMobileを用いたアプリケーションを作成する手順と，実行する方法について述べる．SmartMobileでは，必要なクラスファイルをネットワークを介して取得するため，アプリケーションを実行する端末をRemoteとし，クラスファイルの取得先をStationとする．また，RemoteとStationで動作するランタイムをそれぞれRemoteランタイム，Stationランタイムとする．

4.5.1　アプリケーションの作成法

図4.10の(1)〜(4)にアプリケーションの作成とデプロイの手順を示す．これにはJavaCC[55]を用いて作成した変換ツール(MakeInitialize)を使用する．

(1) 開発者がアプリケーションで使用するクラスを実装し，SmartAPIを使用してイニシャライズ部分(initialize.src)を記述する

(2) MakeInitializeを実行する．図4.11に示すように，MakeInitializeでは，(a)イニシャライズ部分を記述したファイル名，(b)変換後のソースファイル名を必ず指定する必要があり，(c)ランタイムが実行するメソッド名，(d)パッケージ名は必要に応じて指定する．(c)の指定がない場合は“smmain”となる．

(3) (2)の結果，通常のJava言語に変換されたソースファイルと，アプリケーションの情報を記述した，XML形式のデプロイファイルが出力される．デプロイファイルにはアプリケーション名，クラス名，メソッド名が記述され，4.5.2節で述べるアプリケーションの実行に使用される．

(4) 開発者は，変換されたソースファイルをコンパイルした後，デプロイファイルと共にStationに保存する．

4.5.2　アプリケーションの実行

図4.12の(1)〜(5)にアプリケーションを実行するための手順を示す．

(1) ユーザがRemoteランタイムを使って，実行するアプリケーション名，Stationランタイムのアドレスとポート番号を指定する．

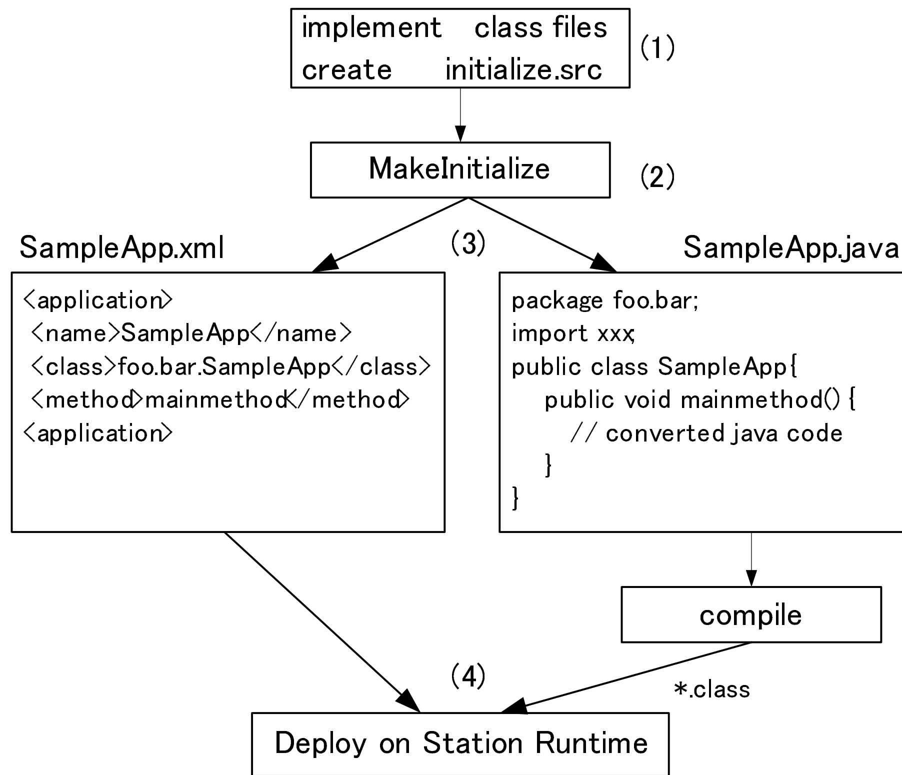

図 4.10 アプリケーションの作成と配置

```
>java MakeInitialize  initialize.src  SampleApp  mainmethod  foo.bar
                           (a)            (b)         (c)        (d)
```

図 4.11 変換ツールの使い方

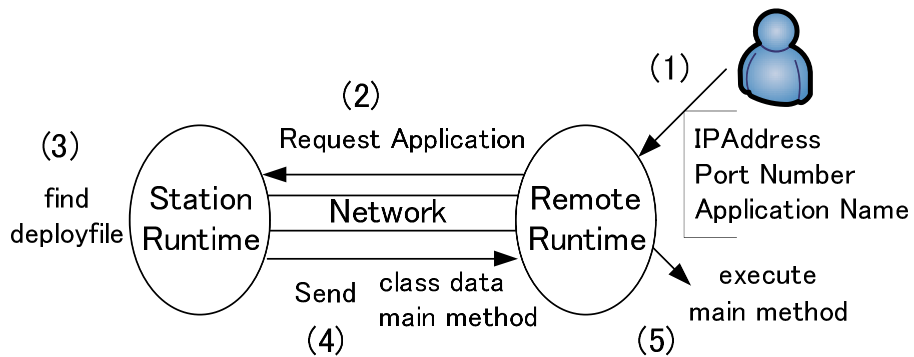

図 4.12 アプリケーションの実行手順

(2) RemoteランタイムがStationランタイムにアプリケーションの取得要求を出す.

(3) Stationランタイムが，デプロイファイルからイニシャライズ部分のクラスファイルと実行するメソッド名を取得する.

(4) Stationランタイムは(3)で取得した情報をRemoteランタイムに送る.

(5) Remoteランタイムは，クラスファイルとメソッド名を受け取り，リフレクションAPIを使用してそのメソッドを実行する.

以降Remoteランタイムは，4.4.3.2項で述べたように動作し，必要に応じてStationランタイムからクラスファイルを取得する．なお，StationランタイムのIPアドレスやポート番号の通知はSmartMobileの機能には含まれていないため，ユーザが別の手段(ウェブやメールなど)を用いて別途入手しておく必要がある.

4.5.3　SmartMobileを使ったアプリケーション

SmartMobileを使ったアプリケーションとして，簡易版インスタントメッセンジャー(TinyMessenger) を実装した．TinyMessengerでは，メッセージを送受信するという基本機能に加え，電子メールやデータの送信，メンバーの追加や検索などの拡張機能を実装している．拡張機能は別々のリスナーオブジェクトで実装され，イニシャライズ部分でdnewを使って生成される．また，各リスナーオブジェクトは，ユーザリストやメッセージの取得や変更を行うために，モデルオブジェクトへの参照をもつ.

4.5.3.1　実行例

図4.13にTinyMessengerの実行例を示す．(1)がSmartMobileのコンソールであり，ユーザは(2)にStationランタイムのIPアドレスなどを入力してTinyMessengerを起動する．(3)で示すように，TinyMessengerの中で，ロードしていない拡張機能のソースオブジェクトは無効状態になり，ユーザがその上で右クリックすることによって，Remoteランタイムが(4)のロードボタンを表示する．ユーザがロードボタンをクリックすると，Remoteランタイムが(3)のソースオブジェクトに対応するリスナーオブジェクトを生成し，ソースオブジェクトを有効化する.

4.5.3.2　評価

SmartMobileの基本性能を示すため，TinyMessengerの起動にかかる時間(以下起動時間と呼ぶ)とデータ転送量，および拡張機能の追加にかかる時間(以下追加時間と呼ぶ)を測定する．また，JWSを用いて起動時間とデータ転送量を測定し，比較することによってSmartMobileを評価する．1.3.2節で述べたように，SmartMobileでは一度取得したプログラムは再利用されるが，この場合すべてのプログラムをStationランタイムから取得している.

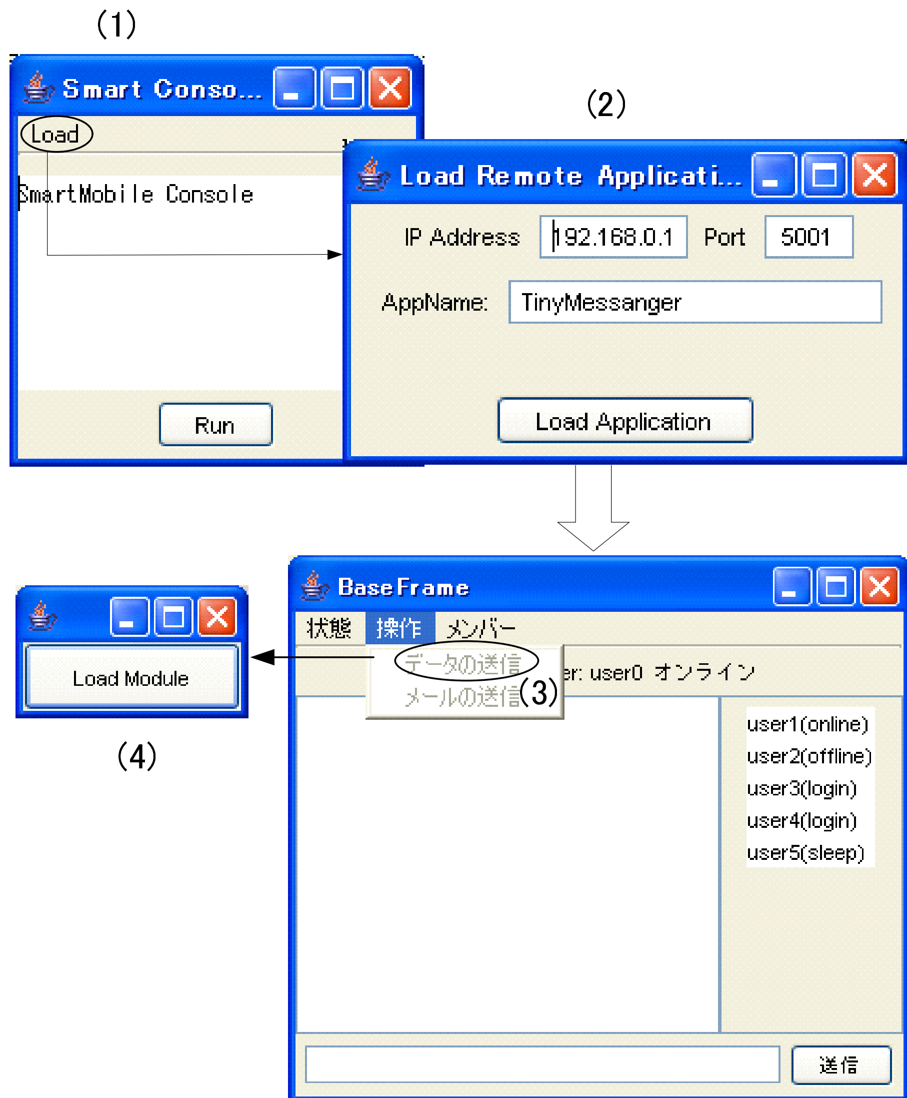

図 4.13 TinyMessenger の実行

表 4.3 SmartMobileでの起動時間

サイズ(kbyte)	12	24	32	52	72	76	98	102
時間(ms)	411	481	540	791	1013	1092	1883	1913

表 4.4 Java Web Startでの起動時間

サイズ(kbyte)	1	36
時間(ms)	1,782	2,373

なお，測定にはj2sdk1.4.2_08がインストールされ，100Mbpsで接続された2台の端末(Pentium4 1.6GHz，1Gbyteメモリ)を使用する．

SmartMobileとJWSを用いた起動時間およびデータ転送量を表4.3と表4.4に示す．表4.3の左端は基本機能だけを起動した場合で，右端に近づくにつれて1つずつ拡張機能を追加(dnewをsnewに変更)して起動している．表4.3に示すように，TinyMessengerはすべての機能の実行に102kbyteのクラスデータを必要とするが，基本機能だけであれば2つのクラスファイル，合計12kbyteで起動でき，起動時間も411ms(TCPコネクションの確立に90ms，クラスファイルの要求と転送に81ms，Stationランタイムのクラスファイル検索とデータ化に20ms，Remoteランタイムのクラスファイル書き出しと18個のオブジェクト生成，18回のメソッド実行に220msの合計)程度であった．起動時間はデータ転送量に比例して増加し，すべての拡張機能を加えて起動すると1,913msであった．また，SmartMobile本来の利用法である基本機能だけを起動し，必要な機能を追加する方式では，12kbyteのクラスデータを必要とする機能の追加時間を測定したところ100msであった．基本機能の実行に必要なクラスデータと，追加機能に必要なクラスデータはともに12kbyteであるにもかかわらず，起動時間と追加時間に差があるのは，TCPコネクションの再利用，および生成するオブジェクトの個数やメソッド実行の回数に違いがあるためである．

一方JWSでは，jarコマンドで36kbyteに圧縮した後転送して実行するが，基本機能だけしか使用しないときには2つのクラスファイルを圧縮した7kbyte以外，つまり29kbyte のクラスデータが無駄になる．また，起動時間も2,373msでありSmartMobileですべての拡張機能を含んで起動するよりも約400ms多くかかっている．しかし表4.4で示すように，JWSでは1kbyteという小さなアプリケーションであっても起動に1,782msかかっており，SmartMobileに比べてデータ転送量の増加に対する起動時間の増加率が小さい．

以上のことから，起動に必要なデータ量が多い場合SmartMobileはJWSに比べて起動時間がかかる可能性があるが，常には必要でないリスナーオブジェクトの生成にdnewを用いることでデータ転送量を減らして起動時間を短縮でき，必要なときにシームレスに追加できるためリソースの有効活用が実現できる．また，4.3.3.1節で述べたように，SmartMobileを用いる場合でもソースオブジェクトの生成は起

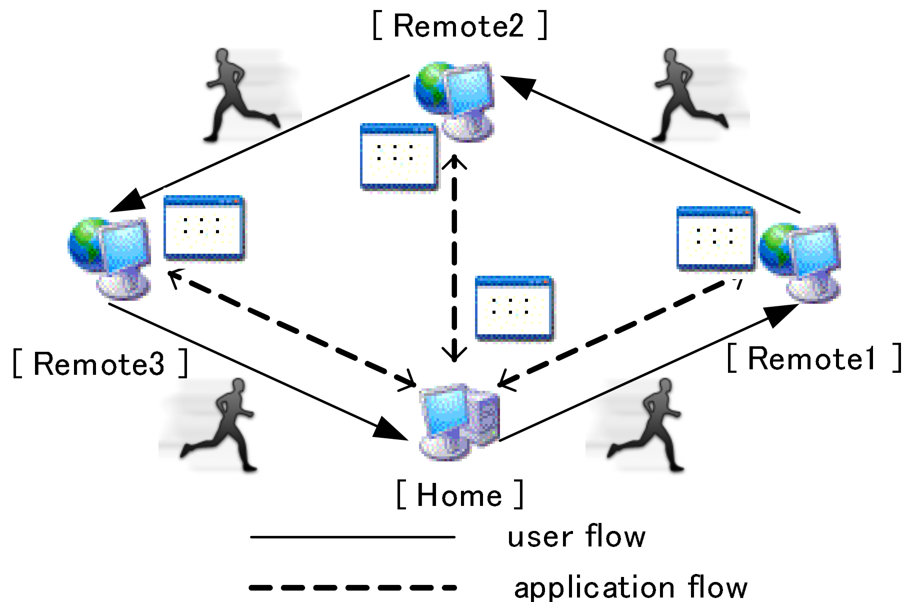

図 4.14 Mobicomを用いたアプリケーションの利用法

動時に行う必要があるが，ソースオブジェクトには一般的にRemoteに配置されたJavaの標準ライブラリを使用するため，それを用いる限りデータ転送量は増加せず，大規模なアプリケーションでもSmartMobileは有効に機能する．また，開発者はオブジェクトの依存関係を意識した設計と，SmartAPIでイニシャライズ部分を記述するだけで蓄積された開発経験を有効に活用できる．

4.6　応用例

これまで述べたように，SmartMobileは，アプリケーションの起動を契機に，ユーザが必要な機能をオンデマンドにロードすることによって端末リソースの有効活用を実現するシステムであり，個々のアプリケーションで利用するデータには関与していない．そのため，データの操作が伴うアプリケーションを使用し，端末に依存することなく継続的に作業を行うためには，SmartMobileを用いて任意の端末で必要なアプリケーションを取得するだけでなく，扱うデータを保存，転送し，取得したアプリケーションで再び読み込む必要がある．

そこで本研究では，SmartMobileで導入したオブジェクトの依存関係と，モバイルエージェントの技術を応用することによって，使用中のデータと状態を保持したまま，ユーザが必要な機能だけを部分的に移動するアプリケーションの提案，およびそれを実現するフレームワークとしてMobicom[56]を実現している．

図4.14で示すように，Mobicomではユーザが自由に使用できる端末をホームとし，ネットワークで繋がったそれ以外の端末(出張先や取引先などの端末)をリモートとする．そして，ユーザはホームで操作(文書の作成など)を中断し出張先や取引先に

移動する場合でも，リモートでホームからデータと状態を保持したアプリケーションを移動して操作を継続し，使用後にホームへ移動する．このように，ユーザが移動して複数のリモートで操作を継続する場合でも，端末に依存せず常にホームからアプリケーションを移動して使用する．また，アプリケーションはホームでは常にすべての機能が利用可能であるが，リモートではユーザが必要な機能をホームからオンデマンドに移動して使用する．

この動作を実現するためには，ユーザがリモートでアプリケーションの移動を指示するとき，すべての機能が揃ったホーム上のアプリケーションから，実際にユーザが必要な機能だけを分離する必要がある．また，アプリケーション使用後に，リモートからホームへ移動した後，リモートに移動していない機能と，リモートから移動した機能を再結合する必要がある．

そこでMobicomでは，SmartAPIの拡張を行い，イニシャライズ部分においてオブジェクトの参照を削除するメソッドも開発者に記述させる．そしてアプリケーションが移動するとき，オブジェクトの依存関係と参照を削除するメソッドを利用し，リモートで不要なオブジェクトをランタイムがホームで分離した後，データと状態を含んだオブジェクトの直列化，転送，復元を行うことによってアプリケーションの部分的な移動を実現している．また，リモートからホームへアプリケーションを移動するときには，リモートから移動したオブジェクトと，ホームから移動していないオブジェクトに対し，イニシャライズ部分に記述された参照を保持するメソッドをランタイムが実行することによってオブジェクトを再結合する．

4.7　まとめ

本章では，アプリケーションの導入コスト軽減と端末リソースの有効活用を実現するフレームワークとしてSmartMobileの提案と実装を行い，それを用いたアプリケーションを構築した．そして，アプリケーションのインストールと実行がシームレスであり，かつ少ないリソースで動作することを確認した．アプリケーションの各機能が別々のオブジェクトで構成され，お互いが依存しない設計の場合，SmartMobileを使用することで効率よく端末リソースを使用することができる．なお，本論文ではGUIアプリケーションを対象としたが，インタフェース機能の実現法を変えることで，アプリケーションの種類によらずSmartMobileを導入することができる．また，SmartMobileの仕組みを応用することで，リソース制約のある携帯端末(PDAや携帯電話など)で実行するアプリケーションの実現や，クラスファイル単位でのソフトウェアアップデートに応用できる．

今後の課題として，現在のSmartAPIはクラスの指定にパッケージ名が必須であり，依存関係が発生しないメソッドにも<DFM>を追加する必要がある．今後これらの制約を排除し，よりJavaの文法に近づけることで開発者の負担を軽減できる．また，転送データの圧縮やsnewで記述されたクラスのクラスデータを一度に転送することによって転送データ量を削減し，起動時間の向上が期待できる．さらに，ユーザプロファイルを導入し，頻繁に利用する機能を起動時にロードすることによってユーザの利便性を向上することも可能となる．

第5章　オンデマンドローディングのアプリケーション管理への応用

5.1　概要

AppliStoreは，SmartMobileで導入した動作方式をアプリケーションの管理へ応用する仕組みである．アプリケーションを実行することだけに着目した場合，SmartMobileを導入することによって大規模で多機能なアプリケーションを実際に必要な機能だけで実行することが可能であり，十分効果的であることは4 章で述べたとおりである．しかし，ユーザがアプリケーションを使用するまでには，開発者がアプリケーションの開発，公開，配布，および変更時の更新作業(以下，デプロイ作業と呼ぶ)を行わなければならない．デプロイ作業の中でも，アプリケーションの更新は開発者にとって負担となる作業であり，第三者のプログラムをライブラリとして利用している場合，自身のプログラムの変更時だけでなく，ライブラリの変更時にも更新作業を行う必要がある．SmartMobileはアプリケーションの配布作業は支援するものの，更新に関する機能は持ち合わせないないため，依然として開発者の負担は軽減されない．むしろ，SmartMobileを用いるにはSmartAPIの使用が必須であるため，開発時の負担は増えることになる．また，SmartAPIを用いていない既存ライブラリを利用する場合，オンデマンドなローディング機能を利用できない．そのため，既存ライブラリを多用する大規模アプリケーションの場合，SmartMobileは効果が半減してしまう．

そこでAppliStoreでは，オブジェクトの生成処理に直接割り込むことにより，専用のAPIを用いることなくユーザが必要な機能だけをオンデマンドに取得するというMobileStart[57]の成果を受け，依存するライブラリをURL形式で記述する機能を提供する．そして，アプリケーションを実行するとき，開発者がURL形式で定義した情報をもとに，ネットワークを介して必要なプログラムを取得する．そのため，開発者は自身のプログラムの配布や更新に集中すればよく，依存ライブラリの再配布を行う必要はない．また，ライブラリの更新作業も自動化される．さらに，専用APIを用いないため，ライブラリを多用する大規模アプリケーションであっても十分機能する．

以下5.2節では，一般的なデプロイ作業と問題点を述べ，AppliStoreのデプロイ作業を述べる．次に5.3節では，オブジェクトの生成処理について述べ，必要な機能だけを取得する方法を示す．そして5.4節でAppliStoreの設計と実装を述べ，5.5節でアプリケーション実行時の動作とAppliStoreの制約について述べる．5.6節では実際にアプリケーションを実行することでAppliStoreを評価し，5.7節でデプロイメントに関する既存技術との比較を行った後，5.8節でまとめを述べる．

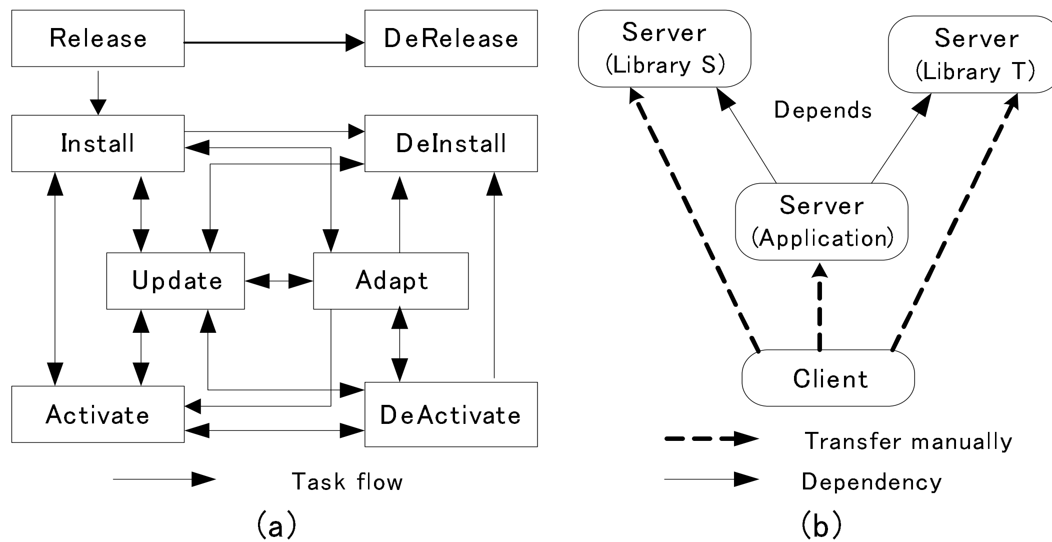

図 5.1 一般的なデプロイ作業

5.2　デプロイ作業と問題点

本節では，アプリケーションの一般的なデプロイ作業と問題点について述べる．次に，AppliStore のデプロイ作業を述べ，違いを明確にする．なお，アプリケーションを配布する端末をサーバとし，実行する端末をクライアントとする．また，サーバとクライアントで動作するAppliStoreのシステムをそれぞれASServer，ASClient とする．

5.2.1　一般的なデプロイ作業

図5.1に一般的なデプロイ作業の要素と流れを示し，問題点を述べる．

5.2.1.1　デプロイ作業の要素

図5.1(a)[58]に示すように，デプロイ作業は一般的にリリース，インストール，アクティベート，ディアクティベート，アップデート，アダプト，デインストール，デリリースの要素に分けられる[58][59].

(1) リリース(release)
必要なプログラムをパッケージ化(package)しサーバ上に配置すること，またアプリケーションの内容や取得法をユーザに広告(advertise)することである．

(2) インストール(install)
パッケージ化されたアプリケーションをサーバからクライアントに転送(transfer)し，クライアントの環境(OSやライブラリなどの違い)にアプリケーションを適応させること(configure)である．

(3) アクティベート (activate)
クライアントでアプリケーションを起動することである．

(4) ディアクティベート (deactivate)
ディアクティベートはアクティベートと反対に，アプリケーションを終了することである．

(5) アップデート (update)
　アップデートはインストール済みアプリケーションの別バージョンをインストールするという位置付けであり，必要なライブラリは既に存在することが多い．アップデートは開発者がアプリケーションを変更した場合に生じる．

(6) アダプト (adapt)
アップデートと同様，インストール済みのアプリケーションを変更することである．しかし，アップデートがサーバで生じるイベント(アプリケーションの変更など) を契機に行われるのに対し，アダプトはクライアントで生じるイベント(環境の変更など)を契機に行われる．

(7) デインストール (deinstall)
クライアントからアプリケーションを削除することである．

(8) デリリース (derelease)
開発者がアプリケーションの開発およびサポートを終了し，ユーザに広告することである．

5.2.1.2　デプロイ作業の問題点

図5.1(a)で明らかなように，一般的に各要素の処理が完了するまで次の作業を行うことができないため，インストールやアップデートを完了するまでユーザはアプリケーションを実行できない．そして，ユーザが実際に使用する機能を事前に知ることは不可能であるため，実際には使用されない機能(プログラム)まですべてインストールせざるを得ず，無駄が多い．

また，第三者が開発したプログラムをライブラリとして使用する場合，図5.1(b)に示すように開発者が第三者のライブラリを含めて自身のプログラムを配布するか，ユーザ自身がライブラリを取得して，インストールを行うことになる．いずれの場合でも，ライブラリのアップデートは開発者またはユーザの負担となり，致命的な問題でアップデートが必要な場合でも，その手間のためそのまま使い続けることも少なくない．しかし一方で，共有ライブラリのアップデートによって，動作しなくなるアプリケーションが出る場合もある．

5.2.2　AppliStoreのデプロイ作業

AppliStoreを用いたデプロイ作業を図5.2に示す．図5.2(a)のように，AppliStoreでは5.2.1.1項で述べた要素の中で(デ)リリース，アダプトを除くその他の作業をネッ

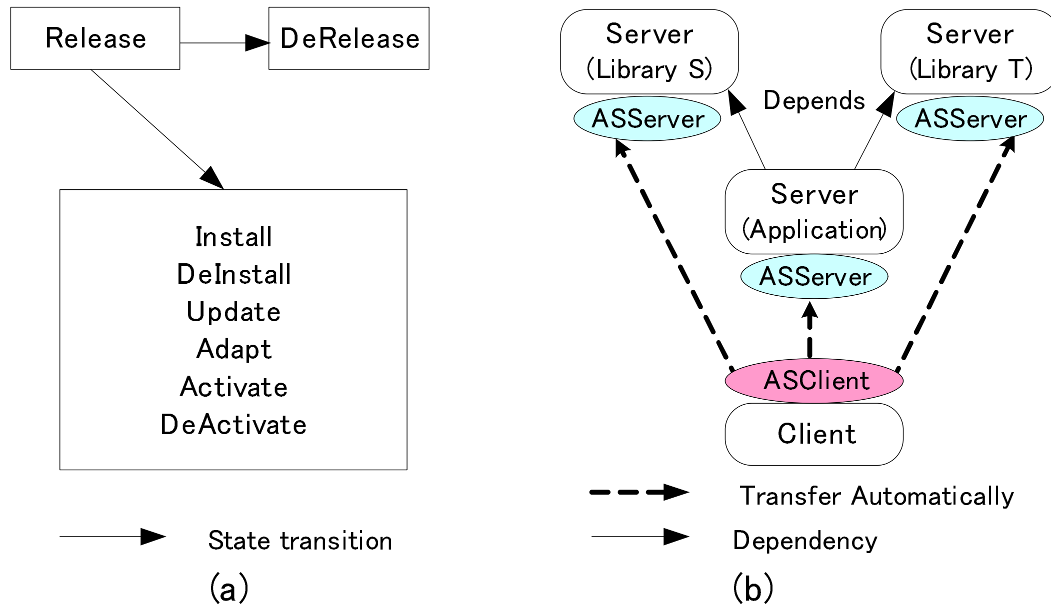

図 5.2 AppliStoreのデプロイ作業

トワークを介してシームレスに行う．図5.2(b)に示すように，この方式ではアプリケーションの起動を契機に，ASServerによって配布されるアプリケーションやライブラリを，ASClientが取得しながら実行する．そのため，図5.1(b)で示したように，開発者またはユーザが依存ライブラリの取得や更新をする必要はない．したがって，ユーザはアプリケーションとそれが依存するライブラリを含め事前にインストールすることなくアプリケーションを実行でき，使用しない機能のプログラムは取得しないですむためリソースを有効に活用することができる．また，開発者は自身のプログラムだけ管理および配布すればよく，依存するライブラリは参照を定義するだけでよい．

次に，この方式に必要な，開発者とAppliStoreの役割を述べる．なお，ASServerとASClientはともに複数のアプリケーションを配布および実行できるが，説明を模式化するため複数のASServerがそれぞれ１つずつアプリケーションまたはライブラリを配布し，ASClientで実行するものと仮定する．

5.2.2.1　開発者の役割

開発者は(デ)リリースを行う．SmartMobileと同様，アプリケーションの実装にはJavaを使用するため，開発者はjarコマンドを使用してパッケージ化を行い，アプリケーションの情報を記述したデプロイファイル(5.4.3.1項で詳しく述べる)とともにサーバに配置する．また，ASClientを使用してASServerにアプリケーションを要求するための情報(以下，アクセス情報と呼ぶ) をユーザに広告する．広告にはAppliStore以外の手段(ウェブやメールなど)を用いる．また，クライアントの環境に応じて柔軟に対応できるアプリケーションの実装，すなわちアダプトも開発者の役

割となる．なお，リソースの有効活用を実現するためにはオブジェクト指向の概念に基づき，アプリケーションの各機能を別々のクラスに分けて実現する必要がある．

5.2.2.2　AppliStoreの役割

AppliStoreでは，アプリケーションを実行するためにユーザのクライアントにASClientを提供し，アプリケーションまたはライブラリを配布するために開発者のサーバにASServerを提供する．ユーザがASClientからアプリケーションを起動すると，ASClientはASServerを通じてサーバから必要なクラスデータを取得しながらインストール(アップデート)とアクティベートをシームレスに行う．また図5.2(b)のように，第三者のライブラリを使用する場合，ライブラリを配布するASServerのアクセス情報を開発者がデプロイファイルに記述しておくことによって，ASClientが必要なクラスデータを取得する．このときASClientが取得するのは開発者がサーバに配置したjarファイルそのものではなく，実際に必要なクラスデータだけであり，複数のアプリケーションで共有しないため，個別にアップデート可能である．

なお，ASClientが取得したクラスデータをクライアントに保存するか否かはユーザが指定でき，保存されたクラスデータは同一アプリケーションの2回目以降の実行に使用され，転送データの削減に効果を発揮する．しかし，クラスデータを保存する場合ユーザが明示的にクラスデータを削除しない限り，デインストールは完了せず，保存しない場合と比較してアップデートの動作が異なる(5.5.3節で詳しく述べる)．

5.3　オブジェクトの生成処理

本節ではまず，実際に必要なクラスデータか否かの判定に用いる，Javaのオブジェクト生成とクラスファイルの関係について述べる．次に，一般的なユーザが最も使用するGUIアプリケーションを例に，アプリケーション設計とオブジェクトの関係を述べる．最後に，5.2.2節で述べた方式に必要なクラスローダの拡張について述べる．

5.3.1　オブジェクト生成とクラスファイルの関係

図5.3(1)のように，Javaではオブジェクトの生成に“new”を使用する．Javaの実行環境(以下，JVMと呼ぶ)は，実際にオブジェクトを生成するときにローカルディスクのクラスファイルからクラスデータを読み込んでオブジェクトを生成する．図5.3の例では，(1)が実行されるときに“MyObj.class”ファイルを検索し，そのクラスデータを用いてMyObjオブジェクトを生成する．言い換えると，(1)を実行しない限り“MyObj.class”ファイルは必要ない．

```
public class JavaClass {
    public void doit() {
        MyObj obj = new MyObj ();  ──────▶  (1)
        obj.mymethod ();
        ...............
    }
}
```

図 5.3 オブジェクトの生成

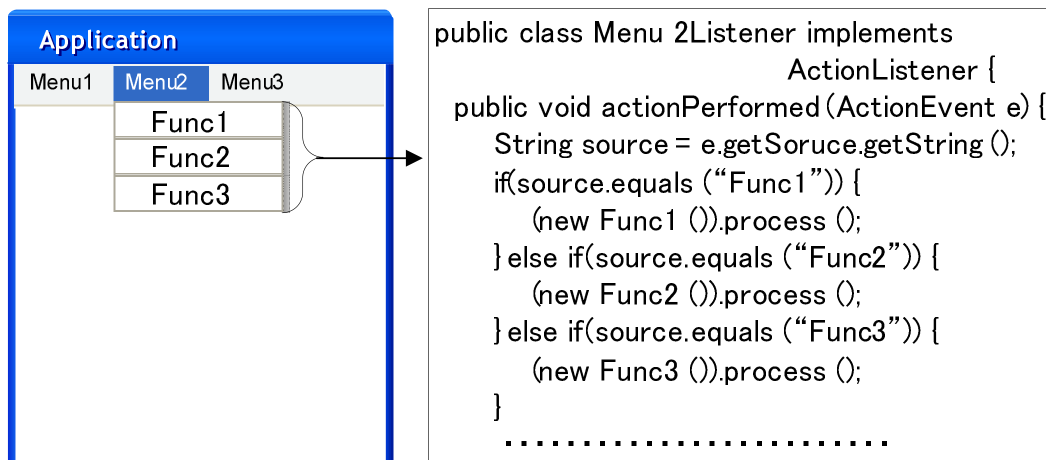

図 5.4 アプリケーション設計とオブジェクト

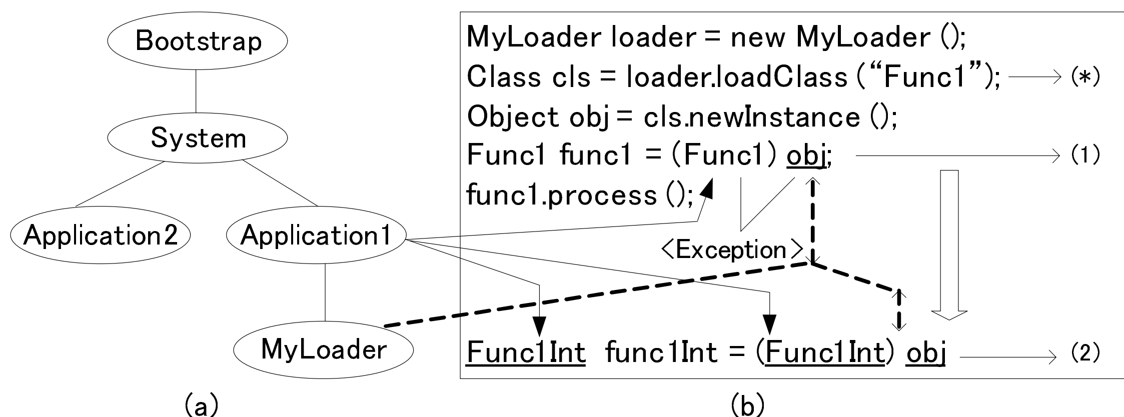

図 5.5 クラスローダとオブジェクトの関係

5.3.2　オブジェクトとアプリケーション設計

4.2節で述べたように，GUIアプリケーションの処理形式はイベント駆動型であり，ユーザがボタン等のソースオブジェクトを操作することでイベントが発生し，イベントリスナーオブジェクトがそのイベントを処理する．図5.4では1つのメニューに属する複数のボタン(ソースオブジェクト)に，同一のリスナーオブジェクトを登録し，リスナーオブジェクトがソースオブジェクトを判定してイベントに対応するオブジェクトを生成した後，処理を委譲している．このように，オブジェクト指向(型言語)では役割ごとにクラスを定義し，1つのコンポーネントとして扱って処理を進めることが一般的である．

アプリケーションがこのように設計されている場合，5.3.1節で述べたように実際に機能が使用されるまでそれに対応するオブジェクトおよびクラスファイルは不要であるが，どの機能を使用するかはユーザによって異なり，開発者が予測することはできない．したがって，現状のインストールやアップデートにはアプリケーション実行前にすべてのクラスファイルをクライアントに配置しておく必要がある．

SmartMobileでは，ユーザの指示でリスナーオブジェクトを生成することによって，オンデマンドに機能を取得していたが，それにはSmartAPIが必須であった．AppliStoreでは5.3.3節で述べるクラスローダの拡張を行うことによって，Javaの標準APIだけでオンデマンドなローディングを可能にする．

5.3.3　クラスローダの拡張

図5.5(a)に示すように，JavaではBootstrapクラスローダをルートにして，複数のクラスローダ(System，Application1, Application2など)が木構造を形成している．そして，それらのクラスローダはロードするクラスに応じて使い分けられる．また，新たにクラスローダを定義することができる．したがって，図5.5(b)の(*)のように記述することによって，必要なクラスデータをオンデマンドに(外部デバイスやネットワークから)取得してFunc1のクラスデータをロードすることができる．しかし

Javaでは，異なるクラスローダでロードされたオブジェクトは同じクラスデータから生成されても別オブジェクトとして扱われる．したがって，図5.5(b)の(1)では，“obj”オブジェクトはMyLoaderでロードされるが，それをキャストするときには暗黙のうちにJVMのクラスローダ(以下，Application1と呼ぶ)が使用される．そのため，ローカル環境にFunc1のクラスデータが存在しなければランタイムエラーが発生する．また，仮にMyLoaderがobjをロードするとき，クラスパス上にFunc1のクラスデータを配置したとしても，キャストに使用されるのはApplication1であるため例外が発生する(objをロードしたクラスローダと異なる).

これを回避するには，Func1クラスのインタフェースとしてFunc1Intをローカル環境に配置し，図5.5(b)の(2)のようにApplication1でFunc1Intをロードし，MyLoaderでロードしたobjをキャストする．この場合，Func1Intとobjをロードするクラスローダは異なるが，インタフェースと実装クラスという関係であるため(同一のクラスデータから生成されない)，例外は発生しない．また，この処理を行うことによってJVMはすべてのオブジェクトをApplication1でロードしたものとみなすことができ，例外も発生しない．

しかしこの方法では，MyLoaderで生成するオブジェクトには必ずインタフェースが必要になり，あらかじめクライアントに配置しておかなければならない．そのため，MyLoaderで生成するオブジェクトが多い場合それぞれに専用のインタフェースを用意するか，1つのインタフェースに制約された実装が必要になる．また，クラスデータを取得するプログラム自身もクライアントに配置しておく必要がある．

したがって，各アプリケーションでインタフェースやクラスデータを取得するプログラムを用意することなく，クラスローダの制約に違反せずにアプリケーションを起動するには，アプリケーションとは独立してクラスデータの取得やオブジェクトを生成する実行環境，すなわちASClientが必要になる．

5.4　設計と実装

本節では，AppliStoreの設計について述べた後，リリース時に開発者が作成するデプロイファイルついて述べ，アプリケーションの管理方式について述べる．

5.4.1　アーキテクチャと各要素の関係

ユーザがASClientからアプリケーションを起動すると，ASClientはアプリケーションごとに専用のクラスローダ(以下，ASLoaderと呼ぶ)を割り当て，Javaの標準クラスローダに代わってそれぞれのASLoaderにオブジェクトの生成処理を委譲する．ASLoaderは，アプリケーションのクラスデータが配置されたサーバからASServerを経由してクラスデータを取得する．また図5.6のように，あるアプリケーション(AppX)が第三者のライブラリ(Library)を使用する場合，デプロイファイルに記述されたアクセス情報をもとに，ASLoaderがそのライブラリを公開しているサーバからクラスデータを取得する．この結果，5.3.3節で述べたクラスローダの制約に違反せず，実際に必要なクラスデータだけでアプリケーションを起動することがで

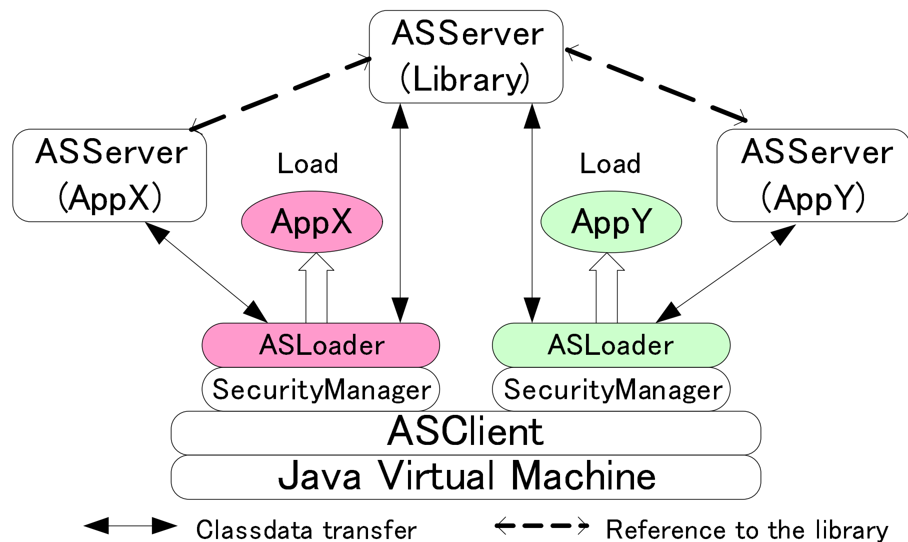

図 5.6 AppliStoreのアーキティチャ

き，第三者ライブラリを再配布する必要がない．また，クラスデータを取得するプログラムを用意し，前もってクライアントに配置しておく必要もない．

5.4.2　セキュリティ

AppliStoreの方式では，クラスデータ(プログラム)をサーバから転送して実行するため，悪意のあるプログラムや改ざんによるクライアントリソースへのアクセスや，ネットワークの使用を制限するなどのセキュリティを考慮する必要がある．AppliStoreでは，ASClientから起動されるアプリケーションをSandboxで実行する[60][61]．AppliStoreのSandboxはJavaのセキュリティマネージャ(SecurityManager)を使用し，アプリケーションのファイルアクセスやネットワーク使用を制限する．図5.6で示したように，ASClientはSecurityManagerがインストールされたASLoaderをアプリケーションに割り当てる．ただし，ネットワークやファイルへのアクセスを要求するアプリケーションに対しては，起動時にユーザの指定によってSecurityManagerをインストールせずにASLoaderを割り当て，アプリケーションにすべてのアクセス権を与えることができる．このSecurityManagerのポリシーを変更することによって，アプリケーション単位でアクセス権をより詳細に制御できるが，現在はアプリケーションごとにアクセス権の有無だけを設定する機能を実装している．

5.4.3　デプロイファイルと管理方式

本節では，アプリケーションAppXの配布を例に，デプロイファイルの作成と管理方式について述べる．図5.7で示すように，AppXはappx.com上のappx.jarと，lib1.comで配布されるライブラリLib1によって実行可能となる．また，Lib1の実行

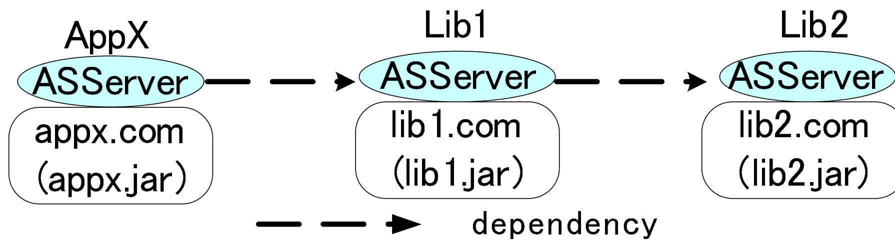

図 5.7 AppXの依存関係

にはlib1.com上のlib1.jarの他に，lib2.comで配布されるライブラリLib2を参照し，Lib2はlib2.com上のlib2.jarだけで実行可能であるとする．なお，AppXの実行にはfoo.AppXクラスのinitメソッドを実行する．

5.4.3.1　デプロイファイルと配置

図5.8にAppXのデプロイファイルを示す．デプロイファイルはファイル名を“アプリケーション名.xml”とし，図5.8(i)部分で示すアプリケーション全般の情報と，(ii)部分で示すバージョンごとに異なる情報で構成され，(ii)部分を複数記述することで異なるバージョンを追加することもできる．

図5.8の(i)には次の(1)〜(4)を，(ii)には(5)〜(9)の要素を記述する．

(1) アプリケーション名

(2) ベースディレクトリ

(3) 現在のバージョン

(4) TTL(Time To Live)

(5) 実行クラス名

(6) 実行メソッド名

(7) JVMのバージョン

(8) ローカルのクラスライブラリ指定

(9) リモートのクラスライブラリ指定

また，(8)と(9)の内部には(*)で示すjarファイル名を含め，(9)のリモートライブラリ指定は次の(a)〜(d)を用いてURL形式で記述する．

(a) 参照サーバ名

(b) ポート番号

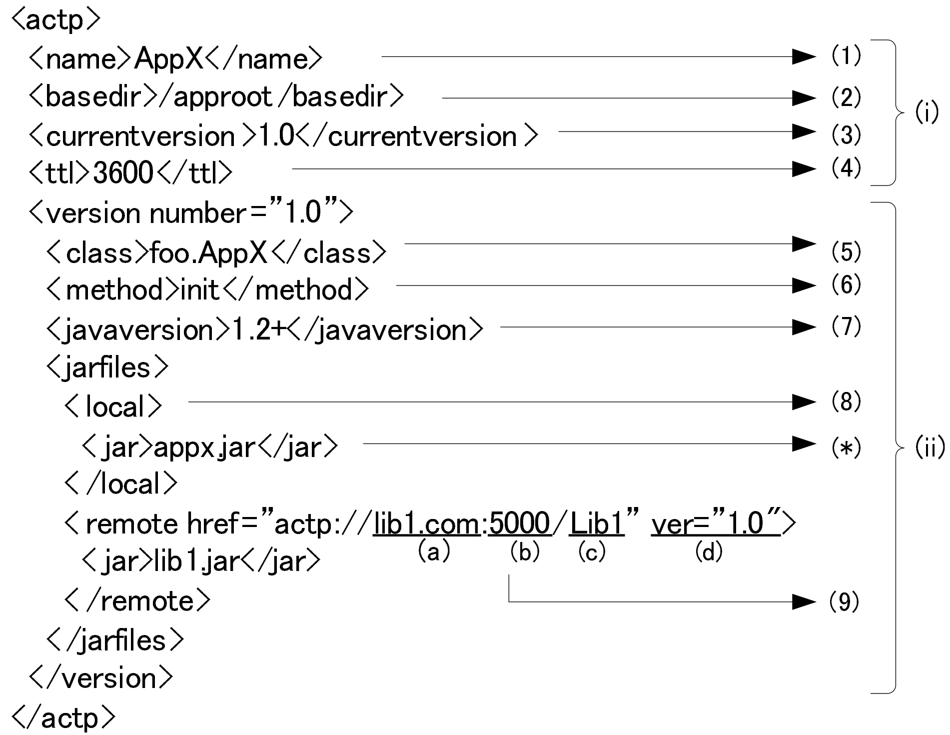

図 5.8 デプロイファイル

(c) ライブラリ名

(d) 使用するバージョン

なお，ライブラリはそれ自身でアプリケーションとして動作しないため，(5)と(6)を省略することができる．

次にデプロイファイルの内容に基づき，デプロイファイル，jarファイルを図5.9のように配置する．

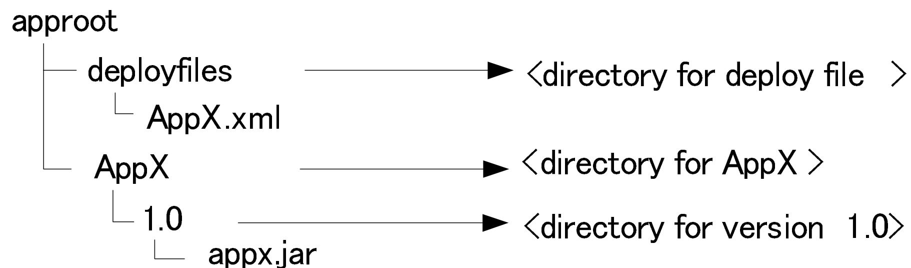

図 5.9 ディレクトリ構造

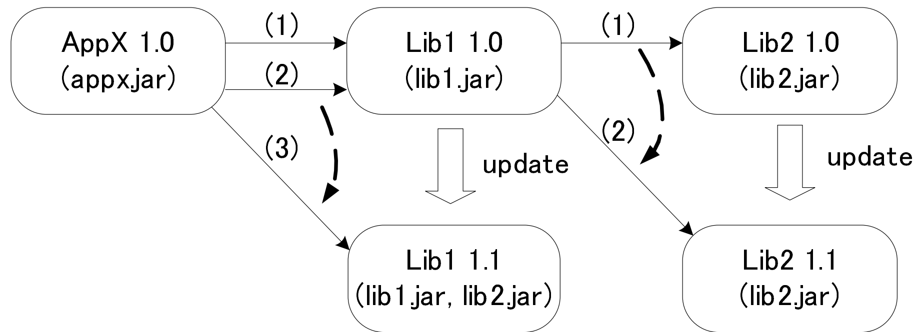

図 5.10 アプリケーションの管理

5.4.3.2　アプリケーションの管理方式

ここでは，アプリケーション(ライブラリ)を配布する際，開発者が負う責任の範囲を明確にする．図5.7の例では，AppXの実行にLib1，Lib1の実行にLib2を必要としている．AppXの開発者(仮に開発者Aとする)は自身のプログラム(appx.jar)の管理と，AppXが直接参照するLib1のバージョン指定を行うだけでよく，間接的に使用するLib2の参照先やバージョンはLib1の開発者(仮に開発者Bとする)が責任を持つ．例えば図5.10のように，AppX，Lib1，Lib2ともにバージョン1.0で動作中(図5.10(1))，Lib2の開発者(仮に開発者Cとする)がプログラムを修正し，バージョンを上げて(1.1にして)公開すると，開発者BはLib1の1.0がLib2の1.1でも正しく動作することを確認した後，Lib1のデプロイファイルで参照するLib2のバージョンを1.0から1.1に変更する(図5.10(2))．このとき，Lib1のプログラム(lib1.jar)に変更を加えていなければLib1自身のバージョンを変更する必要はなく，Lib2の更新による開発者Aの作業は発生しない．ただし，Lib2の公開停止や開発者BがLib2のプログラム(lib2.jar)を拡張するなどの理由でlib2.jarをLib1の一部として再配布する場合，lib1.jarに変更がなくても開発者BはLib1のバージョンを更新する必要がある(図5.10(3))．

このように，開発者の責任は，自身のプログラムとそれが直接使用するライブラリのバージョン指定および，アプリケーションの動作であり，間接的に使用するライブラリの存在やバージョンは隠蔽される．

5.5　AppliStoreの動作

本節ではAppliStoreの動作を，開発者が行うアプリケーションの公開と更新，ユーザによるアプリケーションの実行，アップデートの順で述べる．そして，AppliStoreの制約について述べる．なお，例題アプリケーションとして図5.7で示したAppXおよび，参照するLib1，Lib2を用い，配布に用いるASServerをそれぞれAS_AppX，AS_Lib1，AS_Lib2とする．

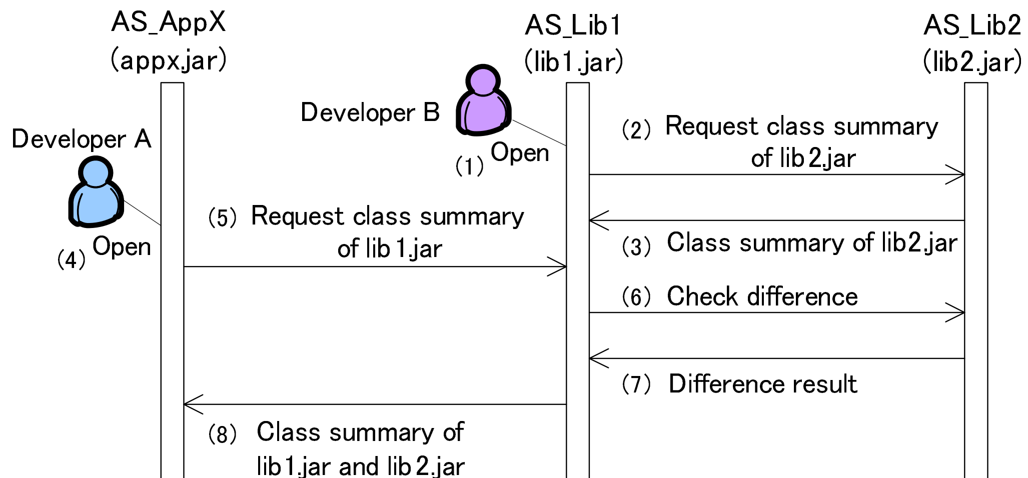

図 5.11 アプリケーション公開時の動作

5.5.1　アプリケーションの公開と更新

図5.11の(1)〜(8)にLib1およびAppXを公開するときのASServerの動作を示す．なお，Lib2は事前に公開されているものとする．

(1) 開発者BがAS_Lib1を操作してLib1を公開する．

(2) AS_Lib1はLib1のデプロイファイルからAS_Lib2のアクセス情報を抽出して接続した後，AS_Lib2に対してLib1が参照するLib2(lib2.jar)に含まれるクラス名一覧を要求する．

(3) AS_Lib2はlib2.jarのクラス名一覧を抽出した後，2つ以上のクラスが同一パッケージに含まれる場合，クラス名を“*”で置き換え，図5.8(4)のTTLとともにAS_Lib1に送信する．例えば“lib2.foo.Class1”と“lib2.foo.Class2”というクラス名は“lib2.foo.*”にまとめられる．AS_Lib1はTTLが過ぎるまでクラス名一覧とAS_Lib2へのアクセス情報を関連付け，保存する．

(4) 開発者AがAS_AppXを操作してAppXを公開する．

(5) AS_AppXは(2)のAS_Lib1と同様にLib1(lib1.jar)に含まれるクラス名一覧をAS_Lib1に要求する．以後，AS_Lib1がクラス名一覧要求を受け付けた時刻が，(3)で受信したTTLを超えるか否かでAS_Lib1の動作が異なる．TTLを超えない場合，(6)，(7)の処理を行わず，直接(8)の処理を行う．

(6) AS_Lib1はAS_Lib2に差分情報を要求する．

(7) AS_Lib1は差分情報を受信し，(3)で保存した情報と有効期限を更新する．

(8) AS_Lib1は(3)と同様にlib1.jarのクラス名一覧を抽出し，保存してあるlib2.jarのクラス名一覧も含めてAS_AppXに送信する．この結果，AS_AppXには参照

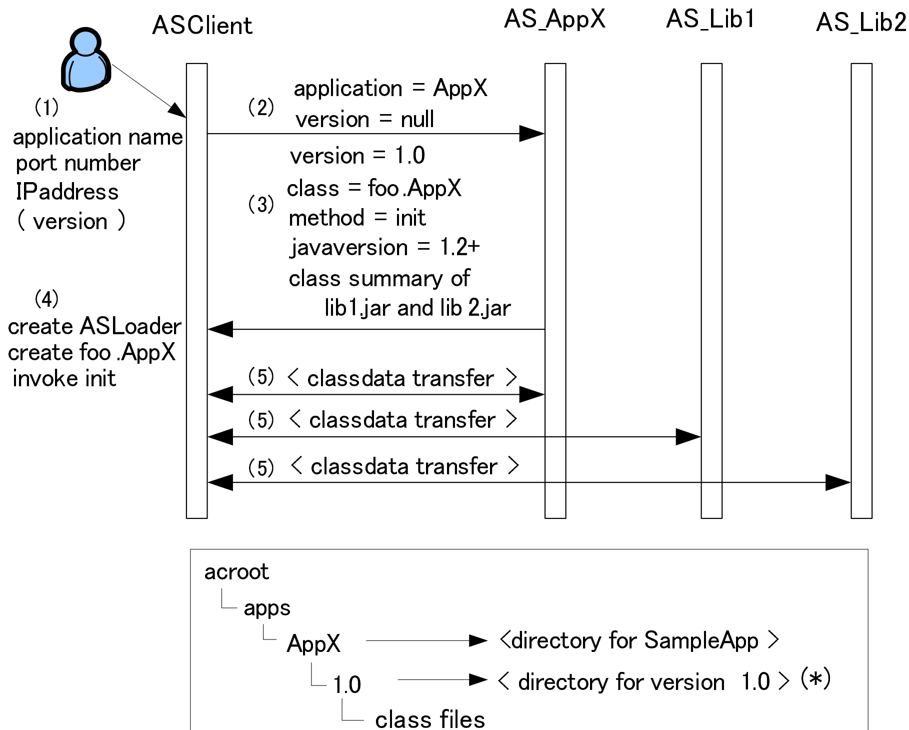

図 5.12 アプリケーションの実行

ライブラリ (lib1.jar と lib2.jar) に含まれるクラス名と，取得先のアクセス情報が保存される．

(6) の差分情報とは，新規に追加または削除されたクラスファイル，または更新されたクラスファイル情報のことであり，更新が行われた否かはクラスファイル単位でハッシュ値を比較することによって検出する．なお，(6) と (7) でやり取りする差分情報は，Lib1が直接参照するLib2(lib2.jar) の変更によるものではなく，Lib2が外部ライブラリの参照を変更するときに生じるため，この例では差分は生じない．lib2.jar に変更がある場合には，5.4.3.2項で述べたようにLib2のバージョンを更新する必要があるため，Lib1が更新後のLib2を使用するにはLib1のデプロイファイルを変更して再公開する必要がある．

5.5.2 アプリケーションの実行

アプリケーションを実行する手順を図5.12の (1)～(5) に示す．

(1) ユーザは開発者が広告するアクセス情報(アプリケーション名，IPアドレス，ポート番号，バージョン) をASClient に入力してアプリケーションを起動する．アクセス情報はバージョン以外必須であり，それらはASClient に保存される．

(2) ASClient はAS_AppX に接続してアプリケーション名とバージョン(任意) を送信

表 5.1 バージョン間の差分

	追加クラス	変更クラス	削除クラス
AppX	-	Main.class, Execute.class	Sub.class
Lib2	Util.class	Mod1.class,Mod2.class	-

する．

(3) AS_AppXはデプロイファイルから対応するバージョンの実行クラス名，実行メソッド名，JVMのバージョン，そして図5.11(8)で保存したクラス名一覧とアクセス情報(TTLを過ぎた場合は図5.11(6)と(7)の更新処理を行う)をASClientに送信する．(2)で送信されるバージョンが空の場合は現在のバージョン(図5.8(3))が使用される．

(4) ASClientは専用のASLoaderを割り当て，クラス名一覧とアクセス情報を設定した後，オブジェクト(foo.AppX)を生成してメソッド(init)を実行する．また，バージョンを指定せずに起動した場合，現在のバージョンがアクセス情報に追加され，2回目からはそのバージョンが使用される．

(5) ASLoaderは“foo.AppX”をはじめとするオブジェクトの生成処理に割り込み，アクセス情報をもとに必要なクラスデータを各ASServerに要求する．

なお，ASLoaderに設定されるクラス名一覧にはAppX(appx.jar)に内包されているものは含まれず，クラスデータの取得先が特定できない場合，その取得要求はAS_AppXに出される．また，ユーザがクラスデータの保存を指定した場合，取得したクラスデータは図5.12の階層構造でクライアントに保存される．図5.12中の“acroot”は，ASClientがインストールされているディレクトリを指す.

5.5.3　アップデート

本節では，クライアントで実行するアプリケーションをアップデートする手順と，AppliStoreの動作を述べる．例として，図5.13で示すように，AppXを1.0から1.1にアップデートする場合を想定する．なお，ユーザはAppXの1.0を使用した経験があり，そのときLib1の1.0，Lib2の1.0を参照していたものとする(図5.13(a))．そして，現在Lib1の1.0はLib2の1.1を参照し，AppXの1.1はLib1の1.0，Lib2の1.1で動作するものとする(図5.13(b))．また，AppXとLib2において，バージョン間で追加，変更，削除されたクラスを表5.1に示す.

アップデートを行うにはまず，開発者が図5.8で示したデプロイファイルに1.1用の(ii)部分を追加し，jarファイルを適切な位置に配置する．そして，図5.8(3)で示した現在のバージョンを1.1に変更した後，AS_AppXを操作してアプリケーションを再公開する．

次に，アップデート時のASServerとASClientの動作を図5.14の(1)〜(8)に示す.

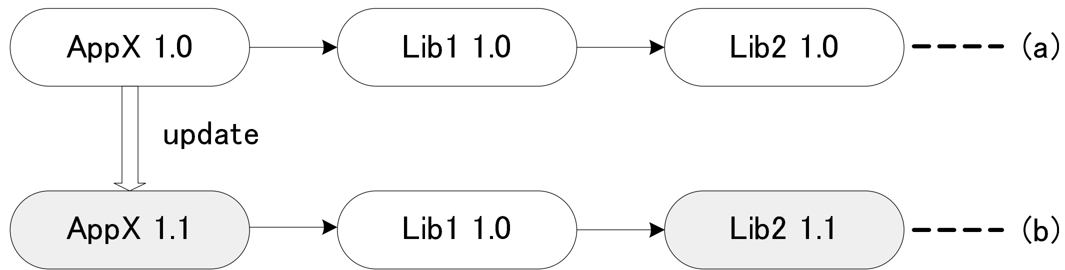

図 5.13 アプリケーションのアップデート

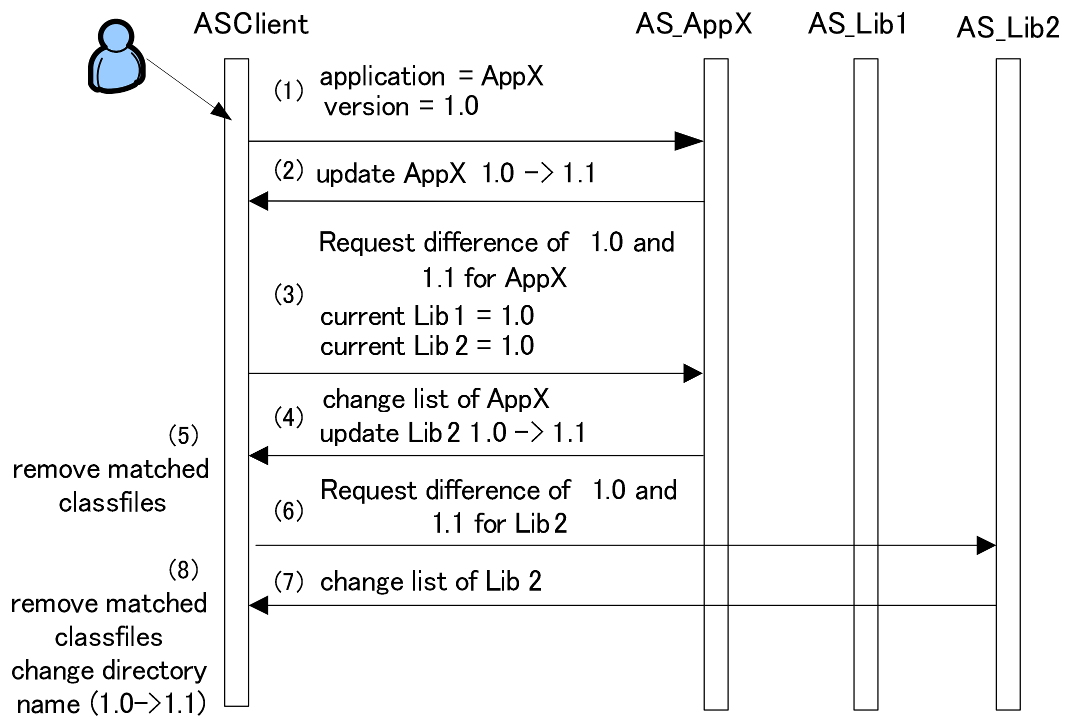

図 5.14 アップデート手順

(1) ユーザがASClientからAppXの1.0を起動する．

(2) AppXの現在のバージョンは1.1であるため，AS_AppXはASClientにアップデートメッセージを送る．ASClientはユーザにアップデート承認用のボタンを提示する．これ以降，クラスファイルが保存されていない場合といる場合では動作が異なり，前者の場合バージョン1.1で5.5.2節のように動作してアップデートが完了する．後者の場合だけ(3)以降の処理が行われる．

(3) ユーザが最後に起動したときのAppXとそれが参照したライブラリのバージョンをASClientがAS_AppXに送信する．この場合AppX, Lib1, Lib2のバージョンはともに1.0となる．

(4) AS_AppXは，自身が直接配布する1.0と1.1のjarファイルをクラスファイルのハッシュ値を利用して比較し，変更があるクラス一覧を抽出する．この場合変更されたクラスとして“Main.class”，“Execute.class” 削除されたクラスとして“Sub.class”名が抽出される．そして，ASClientから送られたLib1とLib2のバージョンとAppXの1.1が現在参照しているバージョンを比較し，変更がある場合には変更クラス一覧とともにLib2のアップデート要求をASClientに送信する．

(5) ASClientは，受信した“Main.class”，“Execute.class”，“Sub.class”がローカルに存在している場合，それらのクラスファイルを削除する．

(6) ASClientはLib2の1.0と1.1で変更されたクラス一覧をAS_Lib2に要求する．

(7) AS_Lib2は，Lib2の1.0と1.1で変更された“Mod1.class”，“Mod2.class”のクラス名をASClientに送信する．

(8) AppXの場合と同様，ASClientは“Mod1.class”，“Mod2.class”がローカルに存在する場合それらを削除する．そして，クラスファイルが保存されているディレクトリ名(バージョン名になっている)を変更する(1.0から1.1に変更)．これ以降はバージョン1.1で5.5.2節のように動作し，アップデートが完了する．

5.5.4　AppliStoreの制約

これまで述べたように，AppliStoreは配布するプログラムの動作を開発者が保障し，必要なライブラリへの参照を定義することによって，ユーザが実際に使用する機能(プログラム)だけをそれぞれのサーバから取得しながらアプリケーションを実行できるシステムである．また，開発者が動作を確認し，参照先を変更するだけで，ユーザに負担をかけることなく参照するライブラリのバージョンアップを行うことができる．しかし，アプリケーションが間接的に同じライブラリの別バージョンを使用する場合，問題が発生する可能性がある．例えばAppXでは，Lib1に加え新たにLib3を直接参照し，それらがLib2の異なるバージョンを参照する場合である．この問題はAppliStoreの問題ではなく，使用するライブラリの組み合わせの問題であるため，AppXの開発者が責任を負う必要があり，AppliStoreではAppXの(再)公開時に警告を表示して開発者に通知する．

また5.5.1節で述べたように，ライブラリのバージョン変更はそれを参照するアプリケーションの開発者がライブラリの更新情報を定期的に確認し，変更にはアプリケーションの動作を確認後，デプロイファイルを更新して再公開する必要がある．AppliStoreでは，仕組み上この作業の自動化も可能であるが，その場合アプリケーションの動作保障ができないためあえて行っていない．

さらに，デプロイファイルで設定するTTLにも配慮する必要がある．AppXの例では，仮にLib2のTTLが短いとしても，Lib1のTTLが長い場合AppXにとってLib2の更新頻度はLib1のTTLに依存するため，TTLは参照するライブラリとの兼ね合いを考慮して決定する．この問題は5.4.3.2項で述べた管理方式に起因し，開発者Aの責任範囲は直接参照するLib1に留まり，間接参照するLib2の存在は隠蔽できることの副作用である．これは，AS_AppXがLib2の情報を持たずにASClientが出すクラスデータ取得要求をASServer間でルーティングさせて解決できるが，この場合1回のクラスデータ取得が複数回リダイレクトされてしまうため，現在の実装ではASClientから1ホップでクラスデータを取得できることを優先させている．

5.6 アプリケーションの実行と評価

本節では，AppliStoreを用いてアプリケーションを実行し，動作を確認するとともに基本性能を測定する．

5.6.1 サンプルアプリケーション

サンプルアプリケーションとして株式自動売買システム(以下，Autradeと呼ぶ)を実装し，AppliStoreを用いて実行する．Autradeは，データの取得，複数の検証アルゴリズム実行，自動取引など，合計10個の機能を別々のクラスで実装しており，イベントに応じてそれに対応するオブジェクトを生成した後，処理を委譲する．またAutradeは，データの取得にHTMLParser[62], 結果のグラフ表示にJFreeChart[63]といった外部ライブラリを使用し，さらにJFreeChartはライブラリとしてJCommon[64]を使用する．これらの関係と実行環境を図5.15に示す．なお，実行に使用するのはj2sdk1.4.2_07がインストールされ，100Mbpsで接続された5台(サーバ4台，クライアント1台)の計算機(Pentium4 1.8G，1Gメモリ)である．

5.6.2 実行例

ASClientからAutradeを実行する様子を図5.16に示す．ユーザはASClient(図5.16(a))の追加ボタンから図5.16(b)を表示し，(2)アクセス情報および，セキュリティやクラスデータを保存するか否かを選択してASClientにアプリケーションを追加する．すると，(3)“Autrade”がメニューに追加され，ユーザはそのメニューをクリックしてAutradeを起動する．以降実行する機能に必要なクラスデータを，ASClientがASServerを通じて各サーバから取得する．

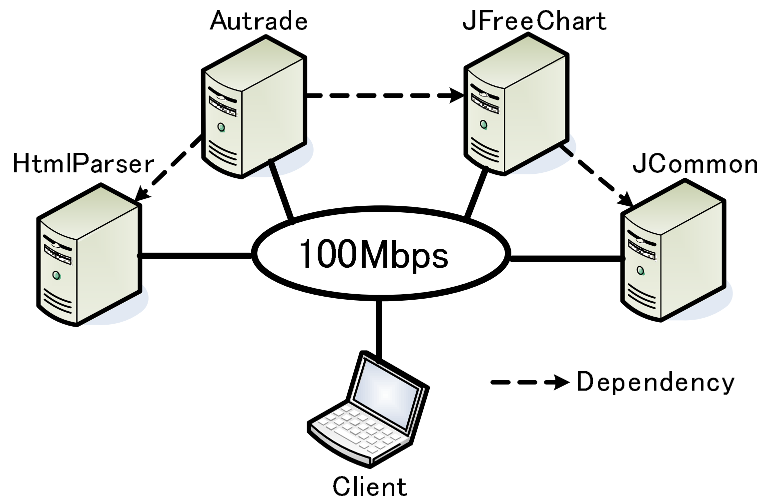

図 5.15 実行環境

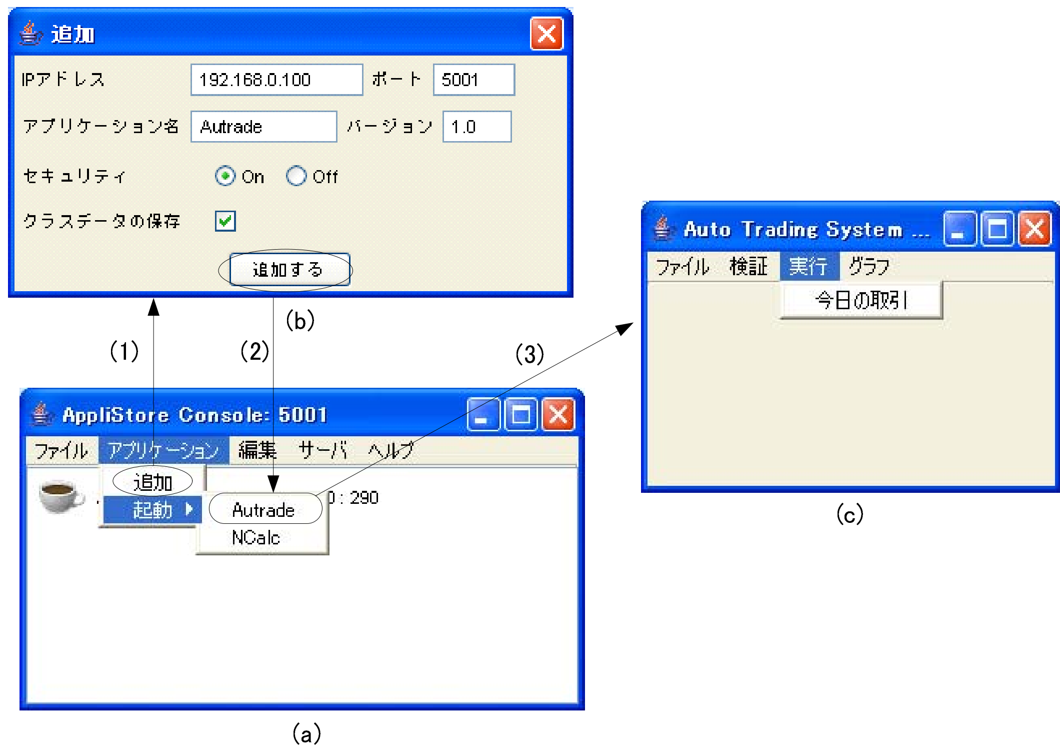

図 5.16 アプリケーションの実行例

表 5.2 ASServerを用いた公開時間

	JCommon	HTMLParser	JFreeChart	Autrade
時間(ms)	5	5	80	150
サイズ(kbyte)	306	617	1075	206
クラス数(個)	232	482	549	69

表 5.3 ASClientを用いた起動と機能実行時間

	起動	データ取得	グラフ表示	アップデート
時間[単体](ms)	100	491[240]	1141[841]	221
サイズ(kbyte)	21.3	206.8	678.4	21.3
クラス数(個)	13	73	160	13

5.6.3 評価

AppliStoreの評価として，Autradeおよび参照するライブラリの公開にかかる時間(以後，公開時間と呼ぶ)，公開データ量，クラス数を測定して評価する．次に，ASClientを利用してAutradeの起動にかかる時間(以後，起動時間と呼ぶ)と機能の実行にかかるオーバヘッド，アップデートを含めた起動時間および，それぞれで実際に必要なクラスデータ量とクラス数を測定し，起動と機能取得時間，クラス数とデータ量，アップデートの順に評価する．そして，最後に総合評価を述べる．

なお，オーバヘッドの測定は，Autradeを単体で起動する場合と，ASClientから起動する場合とでデータ取得機能とグラフ表示機能それぞれの実行時間の差分で測定する．また，アップデート時間はJFreeChartが参照するJCommonのバージョンを1.0.0から1.0.4に変更してJCommonとJFreeChartを再公開した後，ASClientからAutradeの起動にかかる時間とする．そして，このときのTTLは0とし，最も時間のかかる場合を想定する．

5.6.3.1 公開時間

公開時間の測定結果を表5.2に示す．JCommonとHTMLParserは5msという非常に短い時間で公開できた．一方JFreeChartおよび，Autradeはそれぞれ80ms，150ms程度時間がかかっている．JCommonのデータ量はAutradeの約1.5倍，クラス数は約3倍なのにもかかわらず，Autradeの公開時間がJCommonの約30倍であることを考えると，Autradeおよび，HTMLParser の公開時間に影響するのは，参照ライブラリのクラス情報取得時間である．したがって，参照関係が深く，広くなるほど公開時間は指数関数的に増加するが，TTLを適切に設定することで非同期に情報を更新することが可能であるため，ASServerは十分実用的な時間内でアプリケーション(ライブラリ)を公開できる．

5.6.3.2　起動と機能実行時間

起動と機能取得時間の測定結果を表5.3に示す．表5.3のように，AppliStoreを用いたAutradeの起動時間は100msであり，十分実用的な時間内で起動が可能である．また，機能の実行時間を見ると，Autradeを単体で動作させた場合と比較してデータ取得に251ms，グラフ表示に300ms程度多くかかっている．これはクラスデータの取得先決定や，TCPコネクションの確立，データ転送のオーバヘッドである．しかし，クラスデータの取得はユーザが最初に機能を実行するときだけである．また，ユーザはすべての機能を一度には実行しないため，オーバヘッドも分散され，実用上問題ない．

5.6.3.3　クラス数とデータ量

表5.2で示すように，Autradeと参照するライブラリのデータ量は，合計で2,204kbyte(非圧縮時は8,517kbyte)であり，クラス数は合計で1,332個であった．それに対し，AppliStoreを用いる場合，表5.3に示すように起動だけであれば13個のクラスファイル，合計21.3kbyte(非圧縮)のクラスデータで実行可能であり，必要なクラス数，データ量ともに全体の1%程度で済む．またデータ取得機能の実行では，HTMLParser全体で617kbyteのサイズ，クラス数482個のうち，実際に使用するのは206.8kbyte，73個のクラスであり，それぞれ全体の約30%のサイズ，約15%のクラス数で実現可能である．

同様にグラフ表示機能で実際に使用するのは618.4kbyte，160個のクラスであり，JCommon，JFreeChart全体(1,381kbyte，クラス数781個)のそれぞれ約50%のサイズ，約20%のクラス数であった．

ライブラリの中で実際に使用する部分は，それを使用するアプリケーションに依存するが，一般にライブラリは汎用的に作られているため，ユーザがアプリケーションの一部の機能しか使用しないのと同様に，開発者がライブラリのすべての機能を使うことは稀である．したがって，参照するライブラリの個数が多いほど，AppliStoreを用いることで削減できるリソースは大きくなる．

5.6.3.4　アップデート

表5.3の通り，JCommonのアップデートを伴ったAutrade起動時間は，伴わない場合の約2倍であった．これは5.5.3節で述べたように，ユーザがAutradeを起動したとき，TTLが0であるためバージョン変更によるASServerでの情報更新，そしてASClientとASServer間で変更クラス情報を送受信する影響である．これもアプリケーション(ライブラリ)公開時の問題と同様，参照関係が深く，広くなるほど更新時間が増加するが，TTLを適切に設定することで調整可能であるため，十分現実的な時間内でアップデートが可能である．また，クライアントにクラスデータを保存しない場合はクラスデータの変更情報をやり取りする必要はない．

5.6.3.5　総合評価

これまで述べた評価から，多数の機能があり，実際にはその中の一部しか利用しないクライアントアプリケーションの場合，AppliStoreを利用することで有効にリソースを活用することができる．特にAutradeのようなGUIアプリケーションでは，ソースオブジェクトの生成は起動時に行う必要がある．しかし，ソースオブジェクトの生成にはクライアントにすでに存在するJavaの標準ライブラリを用いるため，それらを使用する限り起動時のデータ転送量および，転送回数は増加しない．また，アプリケーション起動時の更新処理によって，ユーザは常に開発者が動作を確認した最新バージョンを利用でき，開発者も必要なライブラリの参照を定義するだけで再配布を行う必要はない．

一方，すべての機能を使用することが前提のサーバアプリケーションの場合，AppliStoreでも実行は可能だがクラスロードにオーバヘッドがあるため適していない．また，サーバアプリケーションは基本的に再起動を行わないため，起動時にアップデートを行うAppliStoreの方式は適さない．なお，AppliStoreの特徴を生かすためには，5.3.2節で述べた設計とTTLを適切に設定する必要がある．

5.7　既存技術との比較

2.2.3節で述べたように，Java Web Startは管理機能としてアプリケーションが更新されたときに差分だけをアップデートする機能を備えている．しかし，第三者のライブラリを使用するときにはアプリケーション開発者による再配布が必要である．

Maven[65]では，依存ライブラリを指定しておくことでリモートレポジトリから自動的に取得してインストール(アップデート)を効率化しており，One-Jar[66]では複数のJARファイルを入れ子構造にして1つにまとめ，確実にインストールを行う手段を提供している．これらはプロジェクト管理や，実行可能なアプリケーションの自己完結を目的としているため，ユーザが実際に使用するか否かにかかわらずすべてのjarファイルをダウンロード，または入れ子構造に含める．

AppliStoreでは，ユーザが使用する機能に必要なクラスデータをクラスファイル単位で取得しながら必要最小限のクラスデータでアプリケーションを実行することができる．また，開発者は必要なライブラリへの参照を定義するだけで再配布を行う必要はなく，開発者がアプリケーション(ライブラリ)の動作を保証する限り，ユーザは確実に最新で安定したアプリケーションを利用することが可能である．

一方，CCM[67]やEJB[68]，WebService[69]といった分散オブジェクト(コンポーネント)技術を導入し，遠隔オブジェクトに処理を依頼することで機能を分割することができる．これらの方法では，処理の結果だけをやり取りするため，クライアントには機能を呼び出すプログラムだけがあればよい．しかし，それらのプログラムは一般的にアプリケーションごとに異なるため，利用するアプリケーションの個数だけ事前にクライアントにインストールしておく必要がある．また，機能を実行するたびにネットワーク接続が必要になり，メソッド呼び出しなどの粒度の細かい処理を遠隔オブジェクトの機能として提供することは難しい．したがって，クライアントアプリケーションへの導入には適さない．

AppliStoreでは，ASClientがインストールされたクライアントで任意のアプリケーションを実行でき，処理結果ではなくプログラムを転送してクライアントで実行するため，一度転送した機能(プログラム)はネットワークの接続性に依存せずに実行可能である．

5.8　まとめ

本章では，SmartMobileで導入した動作方式をアプリケーションの管理へ応用したシステム，AppliStore の提案と実装を行った．そして，実際にアプリケーションをAppliStoreで実行して動作を確認し，起動時間，データ転送量，機能実行のオーバヘッドを測定して有効性を示した．

AppliStoreでは，アプリケーションの起動を契機に，必要なプログラムをネットワークを介して取得するとともに，依存ライブラリへの参照をURL形式で記述することができ，アプリケーションやライブラリ開発者の責任を明確にした上でユーザに安定したアプリケーションを提供することができる．そして，ユーザも煩わしいインストールやアップデートを気にする必要がない．しかし実際の運用では，アプリケーションの開発者または管理者は，依存ライブラリの変更時にはそれらの変更点や動作の確認し，安定して動作すること保証することが望ましい．

また，AppliStoreはアプリケーションがJavaで記述されていればその種類によらず適用することができるが，サーバアプリケーションのようにすべてのオブジェクトを生成しなければ動作しないものでは，従来の実現方法との相違はない．また，現在の実装では，クラスデータを取得するサーバへのホップ数を優先したため，公開するアプリケーション(ライブラリ)のTTL設定は参照先との兼ね合いを考慮して決定するという制約がある．

第6章　総 括

本章では，本論文の各章をまとめて総括するとともに，今後の展望について述べる．

6.1　まとめ

1章では，本論文の背景を述べ，モビリティの定義を行った．そして，本論文の目的と意義，実現したシステムの概要を述べた．

2章では，アプリケーションのモビリティを実現するための既存研究，およびそれらに必須のネットワーク接続方式に関する既存研究を述べ，本論文の位置づけを明確にした．

3章では，本論文で実現した端末のネットワーク接続方式，XFWについて詳細に述べた．XFWを導入することによって，一般的なウェブブラウザだけを使用し，ユーザ認証によるアクセス制限，およびIPアドレスやMAC アドレス偽造による不正利用を防止することができる．そして，負荷テストと運用実績を示し，有効性を確認した．

4章では，アプリケーションの起動を契機に，必要な機能をオンデマンドに取得しながら実行するという方式を提案し，それを実現するためのフレームワーク，SmartMobileの詳細を述べた．そして，アプリケーションを構築し，起動時間と転送量をJava Web Startと比較することによって有効性を示した．SmartMobileによって，たとえ大規模で多機能なアプリケーションであっても，ユーザが実際に使用する機能だけでアプリケーションを実行することが可能であり，端末のリソースを有効に活用することができる．また，実行中に必要になった機能はオンデマンドで取得して利用することができる．

5章では，4章で提案した実行方式をアプリケーションの管理へ応用したシステム，AppliStoreの詳細を述べた．AppliStoreは，SmartMobileで導入したオンデマンドに機能を取得する優位性を残したまま専用APIの制限を排除し，さらに依存ライブラリをURL形式で指定できるように拡張することで再配布を行うことなく一元管理を実現した．AppliStoreによって，開発者は自身のプログラム管理に集中すればよく，依存ライブラリの更新は自動化される．

これらの仕組みによって次のようなことが可能になる．

まず，不特定多数のユーザに対してネットワーク接続を提供する場合，不正利用を防止するために専用ソフトウェアの導入や，ネットワークに接続された共用端末を利用する方式，あるいは利便性を重視してアドレス偽造による不正利用を許容してウェブブラウザによる認証処理を行う方式がとられてきた．本論文で述べた

XFWを導入することにより，アドレス偽造による不正利用を防止しつつ，一般的なウェブブラウザだけで認証処理を行うことができるため，端末に依存することなく統一的なネットワークアクセス方法を提供することが可能になる．

次に，現在企業や大学はもちろん，駅や空港でも共用端末が設置されているが，ユーザは端末自身ではなく，目的に適したアプリケーションに依存しているため，共用端末でできることは限られている．例えば大学の共用端末では，教育に必要なアプリケーションはあらかじめインストールされているが，各ユーザが希望するアプリケーションが必ずしもインストールされているとは限らず，一般に新規にアプリケーションをインストール権限が与えられていない．仮にその権限を与えた場合，管理の煩雑化やリソースの浪費が容易に想像できる．

本論文のSmartMobileやAppliStoreを導入することによって，ユーザが希望するアプリケーションをネットワークを介してオンデマンドに取得し，実行することができるため使用する端末とアプリケーションを分離することができ，端末に依存せずに作業を行うことができる．

さらに，必要のない機能は取得する必要がないため，大規模アプリケーションで問題となる起動時間や端末リソースの浪費を軽減することが可能である．なお，ネットワーク接続は前述したXFWの導入により，直感的なインタフェースを介して認証処理を行うことで取得すればよい．また，AppliStoreを導入することにより，プログラムの一元管理が可能になり，アプリケーションやライブラリの管理コストが軽減される．

6.2　今後の展望

本論文で実現したSmartMobile，およびAppliStoreの動作方式は，アプリケーションを必要な機能だけで実行するという特徴を備えており，アプリケーションは単一の端末で動作する．そのため，PDAや携帯電話といったリソース制約の厳しい端末で動作するアプリケーションの基盤技術として応用が期待できる．例えばURC(Universal Remote Console)[70]では，ユーザが持っているデバイスによって周辺の機器の制御を可能にするために，対象の機器自体に固有のインタフェースを持たせるのではなく，ユーザの持つデバイスによって最適なインタフェースを提供することを目的としている．ユーザが携帯できる端末は一般にリソース制約が厳しいため，多数の機器を扱う場合それぞれの機器を制御するアプリケーションをすべてインストールすることは難しい．SmartMobileやAppliStoreを導入することにより，実際に使用する機能だけで実行が可能になるため，複数の制御用アプリケーションを同時に実行することが可能になる．さらに，RFID[71]などを利用した位置情報検出システムと連携することによって，ユーザが能動的に制御用アプリケーションを検出して取得するのではなく，システムからユーザの端末に利用可能な機器のアプリケーションを提示することも可能である．

また，SmartMobileやAppliStoreは一般にアプリケーションを配布するシステムであるため，それらで扱うデータには関与していない．一方，大学や企業など特定多数を対象にアプリケーションを配布する場合，正規端末だけにダウンロードを許

可する仕組みや，外部ファイルサーバと連携してデータを一元管理することによって，利便性が向上する．そのため，XFWで導入したアドレス偽造防止法を応用することで正規端末を識別し，データを扱うAPIを提供することで外部ファイルサーバと連携する機能を実装する予定である．

一方，本論文で実現したシステムの中で，XFWは特別なアプリケーションを導入することなく利用できるが，SmartMobileやAppliStoreはJava言語の動的なローディング機構を拡張しているため，Javaの実行環境が必要となる．また，Javaの実行環境をインストールした後，SmartMobileやAppliStoreのランタイムををインストールする必要がある．SmartMobileやAppliStoreの動作方式は動的なローディング機構を持つ言語，および実行環境であれば応用が可能であるため，Rubyや.NET Frameworkでの実装やインストーラを提供することによって，より多くのユーザに広めることができると考えている．

謝 辞

本論文をまとめるにあたり，多くの方に御指導，御助言，御協力をいただきました．まずはじめに，学部以来御指導，御教授を頂きました慶應義塾大学理工学部助教授 山本喜一先生に深く感謝いたします．山本先生の御指導がなければ，これほど深くソフトウェアの研究に携わることもなかったことでしょう.

また，本論文のために貴重な時間を割いて下さり，貴重かつ建設的な御意見を下さった副査の先生方，慶應義塾大学理工学部教授 寺岡文男先生，慶應義塾大学理工学部助教授 高田眞吾先生，慶應義塾大学理工学部教授 山口高平先生に深く感謝いたします.

本論文の第3章の研究は，2004年から筆者が所属している慶應義塾インフォメーションテクノロジセンターの研究開発業務の一環として行いました．その間，本研究に関する有益な討論，御助言を頂いた，細川達己氏，林貞孝氏，山方崇氏をはじめとする皆様に感謝いたします．また，本論文第4章と5章の研究において，アイデアや実装に関する数多くの有益な討論，御助言を頂いた国立情報学研究所アーキテクチャ科学研究系教授 佐藤一郎博士，慶應義塾大学理工学部助手 飯島正博士に深く感謝いたします．また，筆者が博士課程を志すきっかけでもあり，さまざまな場面で適切な助言を頂いた，マサチューセッツ大学ボストン校 計算機科学学部助教授 鈴木純一博士に心から感謝いたします.

そして，それぞれ異なる問題に取り組みながらも，共に研究していくことができた，松井望氏，辻将悟氏，清家宏之氏をはじめとする山本研究室の皆様には深く感謝いたします．特に清家氏には，研究のアイデアに関する議論に始まり，実装や論文の査読など，多くの点で助けていただきました．また，研究活動を続けていく中で，常に応援し，支えてくれた家族に深く感謝いたします.

最後に，一喜一憂を共にし，常に支え続けてくれた法政大学社会学部兼任講師 堀亜砂実氏に心から感謝いたします.

参考文献

[1] 高橋登史朗：入門 Ajax，ソフトバンククリエイティブ (2005).

[2] Adobe Systems Inc.: Flex. http://www.adobe.com/jp/products/flex/.

[3] David Flanagan: *Javascript: The Definitive Guide, Third Edition*, O'REILLY (1998).

[4] Inc., A. S.: Flash. http://www.adobe.com/jp/products/flash/.

[5] Sun Microsystems Inc.: Java Web Start Technology. http://java.sun.com/products/javawebstart/index.jsp.

[6] 福田浩章，山本喜一：SmartMobile: アプリケーションの部分的なオンデマンドローディングを支援するフレームワーク，電子情報通信学会論文誌．B, Vol. J89-B, No. 10, pp. 1902–1910 (2006).

[7] 福田浩章，山本喜一：XFW: アドレス偽造に対応したオープンスペース用ネットワークアクセスサービスの実装と導入，情報処理学会論文誌， Vol. 47, No. 8, pp. 2352–2361 (2006).

[8] The World Wide Web Consortium: URIs, URLs, and URNs: Clarifications and Recommendations 1.0. http://www.w3.org/TR/uri-clarification/.

[9] 後藤英昭，安西従道，二階堂秀夫，千田栄幸，満保雅浩，静谷啓樹：ユーザ認証機構を有する安全な無線LAN・情報コンセント統合システムの構築，情報処理教育研究集会講演論文集，pp. 382–385 (2001).

[10] 石橋勇人，山井成良，安倍広多，大西克実，松浦敏雄：IPアドレス/MACアドレス偽造に対応した情報コンセント不正アクセス防止方式，情報処理学会論文誌， Vol. 40, No. 12, pp. 4353–4361 (1999).

[11] 石橋勇人，山井成良，安倍広多，阪本　晃，松浦敏雄：利用者ごとのアクセス制御を実現する情報コンセント不正利用防止方式，情報処理学会論文誌，Vol. 42, No. 1, pp. 79–88 (2001).

[12] 只木進一，江藤博文，渡辺健次，渡辺義明：利用者移動端末に対応した大規模ネットワークのOpengateによる構築と運用，情報処理学会論文誌， Vol. 46, No. 4, pp. 922–929 (2005).

[13] 広島大学情報メディアセンタ：Portguard. http://www.portguard.org/.

[14] R.Beck: Dealing with Public Ethernet Jacks-Switches, Gateways, And Authentication, *In Proceedings of the 13th USENIX conference on System administration*, pp. 149–154 (1999).

[15] IETF: TELNET PROTOCOL SPECIFICATION.
http://www.ietf.org/rfc/rfc854.txt.

[16] Nomadix Inc.: Universal Subscriber Gateway.
http://www.nomadix.com/products/usg.asp.

[17] 日商エレクトロニクス株式会社：POPCHAT. http://www.popchat.jp.

[18] Hitachi Cable, Ltd.: Apresia. http://www.apresia.jp/solution/secu.html.

[19] IETF: INTERNET CONTROL MESSAGE PROTOCOL.
http://www.ietf.org/rfc/rfc792.txt.

[20] IEEE: 802.1Q-1998 IEEE Standards for Local area Metropolitan Area Networks: Virtual Bridge Local Area Networks: Virtual Bridge Local Area Networks, *IEEE* (1998).

[21] IEEE: IEEE Draft P802.1X/D11: Standard for Port based Network Access Control, *LAN MAN Standards Committee of the IEEE Computer Society* (2001).

[22] Boca Research and Boca Raton: Citrix ICA Technology Brief, *Technical White Paper* (1999).

[23] B. C. Cumberland, G. Carius and A. Muir: Microsoft Windows NT Server 4.0, Terminal Server Edition, *Technical Reference, Microsoft Press* (1999).

[24] T. Richardson, Q. Stafford-Fraser, K. R. Wood and A. Hopper: Virtual Network Computing, *IEEE Internet Computing*, Vol. 2, No. 1 (1999).

[25] Tarantella Inc.: Tarantella Web-Enabling Software: The Adaptive Internet Protocol, *SCO Technical White Paper* (1998).

[26] R. W. Scheifler and J. Gettys: The X Window System, *ACM Transactions on Graphics*, Vol. 5, No. 2 (1986).

[27] B. K. Schmidt, M. S. Lam and J. D. Northcutt: The Interactive Performance of SLIM: A Stateless, Thin-Client Architecture, *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (1999).

[28] Chess,D.M, Harrison, C. G and Kershenbaum,A: Mobile Agent: Are They a Good Idea?, *Lecture Notes in Computer Science*, Vol. 1219, pp. 25–47 (1997).

[29] 佐藤一郎：モバイルエージェントの動向，人工知能学会論文誌， Vol. 14, No. 4, pp. 598–605 (1999).

[30] 佐藤一郎：FAQ:モバイルエージェント，コンピュータソフトウェア， Vol. 17, No. 1, pp. 24–25 (2000).

[31] 佐藤一郎：モバイルエージェント，コンピュータソフトウェア， Vol. 17, No. 2, pp. 45–54 (2000).

[32] 岩井俊弥：Javaモバイル・エージェント，SRC (1999).

[33] White, J. E.: Telescript Technology: Mobile Agents, *Software Agents, Bradshaw, J.(ed), MIT Press* (1997).

[34] Lange D. and Oshima M.: *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley Longman Publishing Co., Inc (1998).

[35] ObjectSpace Inc.: ObjectSpace Voyager Technical Overview (1997).

[36] Satoh, I.: A Mobile Agent-Based Framework for Active Networks, *Proceedings of IEEE Systems, Man, and Cybernetics Conference (SMC'99)*, pp. 71–76 (1999).

[37] Satoh, I.: MobiDoc: A Mobile Agent-based Framework for Compound Documents, *International Journal of Computing and Informatics*, Vol. 25, No. 4, pp. 493–500 (2001).

[38] Satoh, I.: MobileSpaces: A Framework for Building Adaptive Distributed Applications using a Hierarchical Mobile Agent System, *Proc. of IEEE International Conference on Distributed Computing Systems*, pp. 161–168 (2000). IEEE Computer Society.

[39] 渡邊裕司，石田好輝：情報収集エージェントシステムにおける分散的信用度評価，電子情報通信学会論文誌 D， Vol. 85, No. 8, pp. 758–766 (2002).

[40] 宇田隆哉，岩田善行，江藤秀一， 重野寛， 松下温：多段ハッシュ方式によるモバイルエージェント認証，情報処理学会論文誌， Vol. 43, No. 8, pp. 2620–2630 (2002).

[41] 春木洋美，河口信夫，稲垣康善：耐タンパハードウェアを用いたモバイルエージェント保護手法，情報処理学会論文誌， Vol. 44, No. 6, pp. 1613–1624 (2003).

[42] 小手川祐樹，櫻井幸一：検証エージェントを用いたモバイルエージェントのためのリアルタイム改竄検出システム，情報処理学会論文誌， Vol. 45, No. 4, pp. 1144–1153 (2004).

[43] 長谷川哲夫， 長健太，粂野文洋， 中島震，大須賀昭彦，本位田真一：インカネーション・エージェントによる移動エージェントの相互運用方式，オブジェクト指向2000 シンポジウム (2000).

[44] IETF: Multipurpose Internet Mail Extensions. http://www.ietf.org/rfc/rfc1522.txt.

[45] Sun Microsystems Inc.: Java Network Launching Protocol and API. http://java.sun.com/products/javawebstart/download-spec.html.

[46] Rigney C.: RADIUS Accounting, *RFC 2139* (1997).

[47] Y. Yeong, T.Howes and S. Kille: Lightweight directory access protocol, *RFC 1777* (1995).

[48] Microsoft Corporation: Web Application Stress Tool. http://www.microsoft.com/technet/itsolutions/intranet/download/webstress.asp.

[49] D. Wagner and B. Schneier: Analysis of the SSL 3.0 Protocol, *In Proceedings of the Second USENIX Workshop on Electronic Commerce* (1996).

[50] IETF: The TLS Protocol Version 1.0. http://www.ietf.org/rfc/rfc2246.txt.

[51] Johannes Mayer: Graphical User Interfaces Composed of Plug-ins, *Proceedings of the GCSE Young Researchers Workshop*, pp. 25–29 (2002).

[52] Johannes Mayer, Ingo Melzer and Franz Schweiggert: Lightweight Plug-in-Based Application Development, *Proceedings of the NetObjectDays*, pp. 97–111 (2002).

[53] Sun Microsystems Inc.: Trial: The Reflection API. http://java.sun.com/docs/books/tutorial/reflect/.

[54] Sun Microsystems Inc.: Using Java Reflection. http://java.sun.com/developer /technicalArticles/ALT/Reflection/.

[55] The Source for Java Technology Collaboration.: Java Compiler Compiler - The Java Parser Generator. https://javacc.dev.java.net/.

[56] 福田浩章，山本喜一：Mobicom: 部分的な移動性をもつアプリケーションを実現するためのフレームワーク，電子情報通信学会論文誌，D， Vol. J89-D, No. 12, pp. 2543–2552 (2006).

[57] 福田浩章，山本喜一：MobileStart：アプリケーションのシームレスな実行を支援するシステム，情報処理学会論文誌(2007).

[58] A. Carzaniga, A. Fuggetta, R.S. Hall, A. van derHoek, D. Heimbigner and A.L. Wolf: A Characterization Framework for SoftwareDeployment Technologies, *Technical Report CU-CS-857-98, Dept. of Computer Science,University of Colorado* (1998).

[59] Vincent Lestideau, Noureddine Belkhatir and Pierre-Yves Cunin: Towards Automated Software Component Configuration and Deployment, *8th International Conference on Information Systems Analysis and Synthesis* (2002).

[60] L. Gong, M. Mueller, H. Prafullchandra and R. Schemers: Going beyond the sandbox:an overview of the new security features in the Java Development Kit 1.2, Proceedings of the USENIX Symposium on Internet Technologies and Systems, USENIX Association (1997).

[61] L. Gong: Secure Java Class Loading, *IEEE Internet Computing*, Vol. 6, No. 2, pp. 56–61 (1998).

[62] Sourceforge.net: HTMLParser. http://htmlparser.sourceforge.net/.

[63] OBJECT REFINERY LTD: JFreeChart. http://www.jfree.org/jfreechart/.

[64] OBJECT REFINERY LTD: JCommon. http://www.jfree.org/jcommon/index.php.

[65] The Apache Software Foundation: Apache Maven. http://maven.apache.org/.

[66] P. Simon Tuffs: Deliver Your Java Application in One-JAR. http://one-jar.sourceforge.net/.

[67] Object Management Group: CORBA Component Model. http://www.omg.org/technology/documents/formal/components.htm.

[68] Sun Microsystems Inc.: Enterprise JavaBeans Technology. http://java.sun.com/products/ejb/.

[69] The World Wide Web Consortium: The World Wide Web Consortium: Web Services Architecture, *W3C Working Group Note* (2004). http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[70] Universal Remote Console Consortium: Universal Remote Console. http://www.myurc.com/.

[71] K. Finkenzeller: *RFID Handbook*, John Wiley & Sons (1999).