卒業論文　　2004年度 (平成16年度)

既存アプリケーションも利用可能なサービスローミングフレームワーク

指導教員

慶應義塾大学環境情報学部

徳田　英幸

村井　純

楠本　博之

中村　修

南　政樹

慶應義塾大学環境情報学部

小泉 健吾

mics@sfc.wide.ad.jp

卒業論文要旨　　2004年度 (平成16年度)

既存アプリケーションも利用可能なサービスローミングフレームワーク

近年の情報科学の進歩により，様々な計算機やセンサが人々の生活空間に遍在するユビキタスコンピューティング環境が実現しつつある．このような環境では，タグやセンサの情報からユーザや空間の状態を取得できる．この情報を用いることで，ユーザに最も近い計算機へサービスを移送したり，空間のある状態に応じてサービスを再配置できる．本研究では，サービスをアプリケーションから提供される機能と定義し，サービスの移送をサービスローミングと呼ぶ．

サービスローミングで用いられるアプリケーションは，移送機能を持つ必要がある．そのため，サービスローミングに対応したアプリケーションの開発は，開発者に負担が掛かる．また，既存のアプリケーションに対し，修正を行う必要性がある．

そこで本研究では，アプリケーションに対し移送機能の自動追加を行うサービスローミングフレームワークを提案する．本フレームワークでは，単にアプリケーションを移送させるだけでなく，サービスローミングを前提として作成されていないアプリケーションに移送機能を追加する．この移送機能の追加をローミング対応化と呼び，移送機能を持ったアプリケーションをローミング型アプリケーションと呼ぶ．本機構がローミング対応化機能を持つことにより，新規にローミング型アプリケーションを作成するだけでなく，既存の非ローミング型アプリケーションも利用可能となる．

本フレームワークは，ローミング対応化を行うアプリケーション変換機構「Sararizer」と，アプリケーション移送機構「Sarari」によって構成される．ユーザがSarariを用いてアプリケーションを読み込むと，Sarari内部で動作しているSararizerがアプリケーションをローミング型アプリケーションへ変換する．Sararizerでは，この変換をアプリケーションのソースコードレベルではなく実行コードレベルで行う．

本論文では，ユビキタスコンピューティング環境を背景にサービスローミングの必要性を述べ，既存のサービスローミング技術での課題を整理する．次に，実行コード変換によるアプリケーションのローミング対応化を行う，サービスローミングフレームワークを提案する．本研究で構築したSararizerおよびSarariについてその概要と設計，プロトタイプ実装について述べる．最後に本システムを評価し，全体をまとめる．

慶應義塾大学　環境情報学部
小泉 健吾

Abstract of Bachelor's Thesis

Service Roaming Framework with Support for Legacy Applications

Recently, the progress of Informatics is leading to the realization of Ubiquitous Computing Environments. Ubiquitous Computing Environments have various sensors and computers embedded around humans. In these environments, states of users and spaces are taken from the information of sensors. By referencing the states, the environment can move the services to the nearest computer of a user or rearrange services among computers. In this research, a "*service*" is defined as a function provided from an application, and "*service roaming*" is defined as migration of a service.

Applications used in service roaming must have migration abilities. However, the implementation of migration abilities is to a cost for developers. Additional implementation is needed to use existing applications for service roaming.

In this research, we propose the service-roaming framework, which adds the migration abilities to application automatically. This framework not only moves applications but also adds the migration abilities to the applications. We define the addition of the migration abilities as "*mobilization*" and an application which has migration abilities as a "*movable application*". We can use not only movable applications built for service roaming but also non-movable legacy applications with this framework.

This framework is consists of two mechanisms. One is the application converting mechanism named "*Sararizer*", and the other is application moving mechanism named "*Sarari*". When users load an application with Sarari, Sararizer working in Sarari converts the application to the movable application. Sararizer realizes this conversion at execution code level not source code level.

In this thesis, we describe the need of service roaming and marshal the problems of existing service roaming technology. Next, we detail this service-roaming framework, which realize the application mobilization. Then, we describe the design and prototype implementation of Sararizer and Sarari. Lastly, we evaluate Sarari and Sararizer and summarize of this thesis.

Kengo Koizumi

Faculty of Environmental Information
Keio University

目 次

図 目 次

表 目 次

第1章　序論

本章では，本研究の背景である，ユビキタスコンピューティング環境について述べる．その後，本研究の目的および意義を述べ，本論文の構成について解説する．

1.1 本研究の背景

近年の情報科学の発達により，計算機やセンサの多様化が進み，日常生活の様々な場面で用いられ始めている．それに伴い，計算機やセンサが埋め込まれたユビキタスコンピューティング環境の研究や開発が進められている．

ユビキタスコンピューティング環境の例として，カーネギーメロン大学の Aura[6]，ジョージア工科大学の AwareHome[5]，マイクロソフト社の Easy Living[7] などが挙げられる．これらの研究では，日常生活空間としてユビキタスコンピューティング環境を構築し，計算機やセンサを空間内に配置している．

空間内に設置されるセンサとして，照度や温度，湿度や人の位置情報を取得できるセンサが開発されている．例えば，人の位置情報を取得するセンサとして，インターセンス社の IS-600 超音波センサ [4] が挙げられる．IS-600 は，小型超音波タグから送られる超音波を天井に設置された複数の超音波レシーバが受信し，小型超音波タグの位置情報を 3 次元で把握するシステムである．また，ケンブリッジ大学では，マット型圧力センサを用いることで，人の加重傾向から個人を特定できるシステムを構築している [9]．

図 1.1: InterSense 社 IS-600 超音波センサ

ユビキタスコンピューティング環境では，複数の計算機を協調させることによって，ユーザが利用しているサービスを別の計算機へ移送できる．これをサービスローミングと呼ぶ．サービスとは，情報機器に存在するアプリケーションからユーザへ提供される機能を指す．例えば，メールソフトであればメール作成機能やメール送受信機能を指し，電話であれば通話機能を指す．ユーザは，情報機器上で動作するアプリケーションを用いてこれらのサービスを利用する．また，サービスローミングを可能にするユビキタスコンピューティング環境をサービスローミングフレームワークと呼ぶ．

ユビキタスコンピューティング環境では，埋め込まれているセンサ情報に応じたサービスローミングが可能になることで，ユーザは計算機に捉われずにサービスを利用で

きる．従来のコンピューティング環境では，あるサービスを利用しようとした時，ユーザはそのサービスを提供するアプリケーションが設置された計算機を利用する必要があった．例えば，デスクトップにのみ動画プレイヤが設置されている場合，ユーザは動画を見るために必ずそのデスクトップを利用する必要がある．つまり，アプリケーションと計算機が強固に結合されているため，他の計算機では同様のサービスを提供するアプリケーションを利用できない．そのためユーザは，他の計算機にも同様のアプリケーションを設置するか，もしくはサービスに応じて計算機を使い分ける必要がある (図 1.2)．

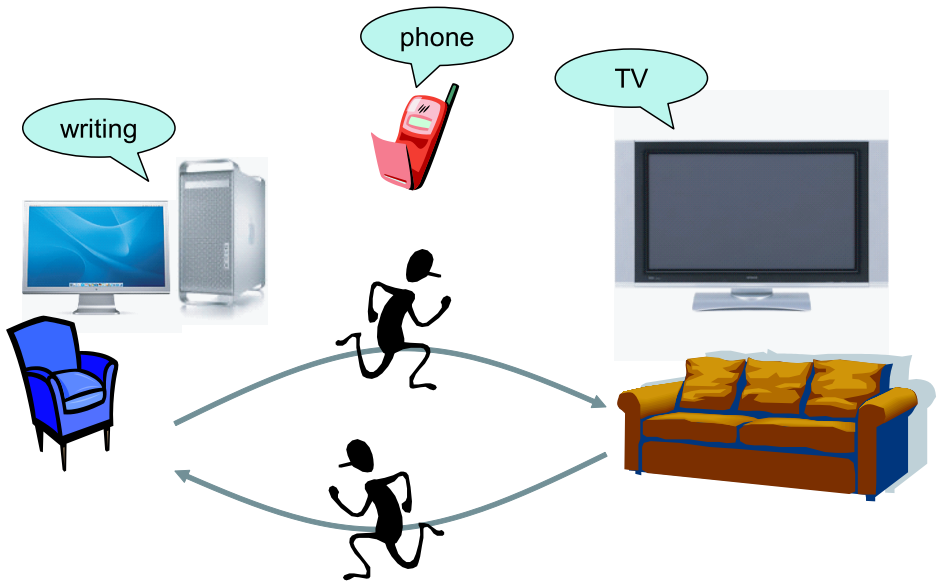

図 1.2: 従来のコンピューティング環境におけるサービス利用

これに対しサービスローミングフレームワークを利用することで，ユーザは特定の計算機に捉われずサービスを利用できる (図 1.3)．例えば，ユーザがデスクトップ PC で鑑賞していた動画を PDA で継続して鑑賞したり，リビングの大画面ディスプレイで利用していたテレビ電話を，自室の小型ディスプレイ PC へ移送したりできる．さらに，ユーザの位置情報を用いると，ユーザはユビキタスコンピューティング環境を動き回りながらサービスを継続利用できる [8]．

サービスローミングの実現手法には，アプリケーションが動作しているリモート計算機のデスクトップ画面をローカル計算機へ移送する方法と，アプリケーションそのものを移送する方法がある．前者の手法は VNC[1] で用いられている手法である．前者の手法では，リモートのデスクトップ画面をキャプチャして移送するため，既存アプリケーションをそのまま利用できる．しかし，移送できるサービスはグラフィカルなサービスに限定され，またムービープレーヤのように画面が頻繁に変わるサービスには不向きである．そのため，本研究ではアプリケーションを移送する手法に着目し，既存のアプリケーションを利用できるサービスローミングフレームワークを構築する．

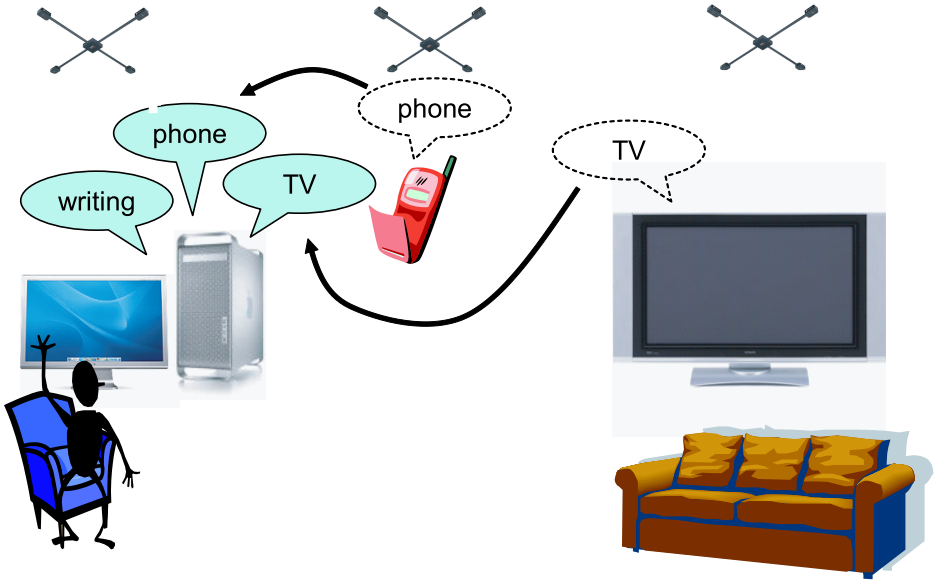

図 1.3: サービスローミングフレームワークにおけるサービス利用

1.2 本研究の目的および意義

本研究の目的は，既存のアプリケーションにおけるサービスを含めたサービスを移送することにより，複数の計算機間でサービス利用を行えるサービスローミングフレームワークを構築することである．ユーザはサービス利用の際，そのサービスを持つ計算機の前へ移動することなく，サービスが利用できる．また，移送機能を持たない既存サービスを再利用可能にすることで，サービスの開発者に対し，サービスローミングを行うための追加的な実装を要求しない．

1.3 本論文の構成

本論文の構成は，本章を含め全 7 章から成る．次の第 2 章では，本研究の基盤となるサービスローミングフレームワークおよびその構築法について論考する．第 3 章では，既存アプリケーションをサービスローミングフレームワークで用いるための方法について考察する．第 4 章では，システムの概要と設計について述べる．第 5 章では，その実装について詳述し，第 6 章でシステムの評価を行う．第 7 章にて，本論文をまとめ，今後の課題について言及する．

第2章　サービスローミングフレームワークの構築

本章ではまず、ユビキタスコンピューティング環境よって実現されるサービスローミングについて述べる．次に、サービスローミングフレームワークの構成について解説する．そして、本研究で扱う問題点を明確にする．

2.1 サービスローミング概要

本節では、本研究で焦点を当てる研究領域を明確にする．まず、本研究で利用する用語を定義し、続いてサービスローミングの概要について述べる．次に、サービスローミングフレームワークの構成について解説し、本研究で扱う問題点について述べる．

2.1.1 サービスローミングの基盤ソフトウェア技術

サービスローミングフレームワークには，ユビキタスコンピューティング環境に埋め込まれている計算機やセンサなどのハードウェア技術に加え，各計算機を連携させるソフトウェア技術が必要である．本項では，ソフトウェア技術について解説する．

サービスローミングでは，アプリケーションを様々な計算機へ移送する必要がある．そのため，ソフトウェア技術として，仮想マシンによるプラットフォーム抽象化技術と，それによって実現されるモバイルコード技術を挙げる．以下に詳細を記す．

プラットフォーム抽象化技術

プラットフォーム抽象化技術は，計算機の差異を抽象化する技術である．計算機のハードウェア的な差違を抽象化するOSと，さらにOSの差違を抽象化する仮想マシンが挙げられる．仮想マシンはOSによって抽象化されたハードウェアインタフェースを利用するため，仮想マシン上で動作するアプリケーションからはハードウェアおよびOSの差違が隠蔽される．この技術により，異なる計算機上でも同一アプリケーションの実行が可能となる．仮想マシンの代表的なものとして，Java仮想マシン [20] が挙げられる．

モバイルコード技術

モバイルコード技術は，プラットフォーム抽象化技術を利用し，アプリケーションコードを仮想マシン間で移送する技術である．仮想マシンから同一のシステムアーキテクチャが提供されるため，アプリケーションコードを同じ仮想マシンが動作している別の計算機へ移送可能となる．

2.1.2 サービス移送手法の比較

本節では，サービスの移送手法を分類する．サービスはアプリケーションによって提供されるため，以下のような移送手法が考えられる．

アプリケーションの視覚移送

アプリケーションの視覚部分のみを移送する手法である．第1章で挙げたVNCやX Window System[2]が例として挙げられる．この手法は，リモートの計算機で動作しているアプリケーションのGUI部分のみを移送する手法である．

この手法では，GUIの再描画を行う度にリモートの計算機とローカルの計算機との間でネットワークを介した通信を行わなければならない．そのため，ムービープレーヤのように再描画を頻繁に行うアプリケーションを利用する場合，ネットワークに負荷が掛かる．また，視覚部分の移送のみを行うため，サービス移送可能なアプリケーションはグラフィカルなサービスを提供するアプリケーションに限られる．

アプリケーションの全移送

アプリケーションをすべて移送する手法である．移送対象には，プロセスの状態やそのプロセスが利用していたファイル等が含まれる．この手法は主にプロセスマイグレーション [3] として実現される．この手法は，アプリケーション全てを移送するため，アプリケーションを移送元の計算機と同じ状態で移送先の計算機に移送できる．しかし，全移送を行うプロセスマイグレーションはOS内部の機能として実装されるため，プラットフォームに大きく依存する．

アプリケーションの状態移送

アプリケーションの状態を移送する手法である．移送元の計算機で動作しているアプリケーションの状態を移送し，移送先の計算機で同じ状態を持つアプリケーションを生成する．例として，モバイルオブジェクト [10] が挙げられる．この手法では，送られてきた状態に応じてアプリケーションを再生成するため，移送先の計算機に同じアプリケーションの実行コードが設置されている必要がある．

ユビキタスコンピューティング環境には様々な計算機が埋め込まれている．そのため，サービスローミングフレームワークでは，第2.1.1節で述べた仮想マシンを用いることで，アプリケーション移送を実現する．したがって，前述したアプリケーションの移送手法のうち，アプリケーションの状態移送が用いられる．

さらにアプリケーションの状態移送は，アプリケーションの実行コードを状態と共に移送することで，同じアプリケーションの実行コードが設置されていない計算機上においてもアプリケーションを再生成できる．この方法は，モバイルエージェント [11] で用いられている方法である．本研究では，アプリケーション移送手法としてこの手法に焦点を当て，アプリケーション移送とは実行コードと状態の移送である，と定義する．

2.2 サービスローミングフレームワークの構成

本節では，サービスローミングフレームワークの構成について述べる．サービスはアプリケーションによって提供されるため，サービスの移送はアプリケーションの移送を意味する．アプリケーション移送は，センサ情報から移送開始のイベントを生成し，ローミング可能なアプリケーションをその移送トリガに応じて移送することで行われる．この移送開始のイベントを移送トリガと呼ぶ．

サービスローミングフレームワークの構成図を図 2.1 に示す．移送トリガ生成機構は，センサ情報から移送トリガを生成し，移送命令発行機構へ送信する．移送命令発行機構は移送トリガを分析し，適切な移送機構へアプリケーション移送命令を送る．移送機構は移送命令に応じ，別の移送機構へアプリケーションを移送する．

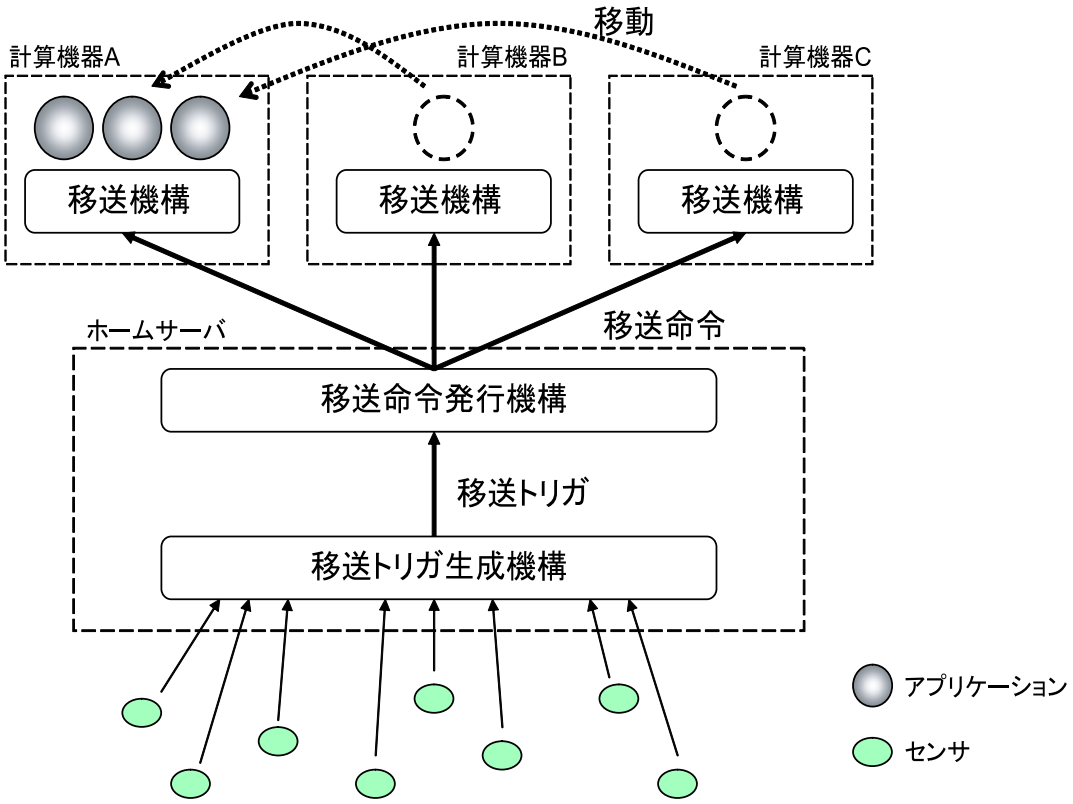

図 2.1: サービスローミングフレームワークの全体構成

サービスローミングフレームワークを構築するには，移送トリガ生成機構，移送命令発行機構，移送機構が必要となる．次項より，各機構について説明する．

2.2.1 移送トリガ生成機構

サービスローミングフレームワークにおけるサービス移送は，センサ情報より生成された移送トリガに基づいて行われる．このとき，単一のセンサからの情報だけでなく，複数の異なるセンサから得られる情報で移送トリガを生成する場合もある．そのため，移送トリガ生成機構は複数のセンサ情報を管理，解析して移送トリガを生成する．生成された移送トリガは，移送命令発行機構へ送信される．

2.2.2 移送命令発行機構

移送命令発行機構は，移送トリガ生成機構から送信された移送トリガを元に，各計算機上で動作している移送機構へサービスの移送命令を発行する．移送命令発行機構は，移送機構と移送機構上で動作しているアプリケーションを把握している．移送トリガ生成機構から移送トリガが送られると，該当する移送機構へ移送命令を送る．移送命令には，移送対象となるアプリケーションと移送先の情報が含まれる．

2.2.3 移送機構

移送機構は，移送命令発行機構から送られてきた移送命令に基づき，アプリケーションを他の移送機構へ移送する．移送機構は，その移送機構上で動作しているアプリケーションのリストを保持している．移送命令が送られると，そのリストから移送対象となるアプリケーションを検索し，移送先へ送信する．

2.3 ローミング型アプリケーション作成時の問題点

本節では，本研究で扱う問題点について述べる．本研究では，サービスローミングフレームワークにおけるアプリケーション作成に焦点を当てる．次項にアプリケーション移送時の問題について述べる．また，アプリケーションの実装手法について解説し，本研究で構築するサービスローミングフレームワークについて述べる．

2.3.1 アプリケーション移送時の問題

サービスローミングフレームワークでは，移送トリガによって稼働状態のアプリケーションが移送される．稼動状態のアプリケーションがメモリやファイルシステムなどの計算機固有のリソースを利用している場合，利用状態を維持したままでは別の計算機に移送できない．アプリケーションが仮想マシン上で動作している場合も同じ状況が起こる．計算機固有のリソースには，メモリやファイルシステムの他に，ネットワークや外付けデバイスなどが考えられる．以下に，アプリケーションが所持し得る，計算機固有リソースを挙げる．

- スレッド
- ファイルへの入出力
- ソケットへの入出力
- デバイスへの入出力

稼動状態のアプリケーションを移送するためには，計算機固有の移送不可なリソースに対し，適切な処理を行う必要がある．例えば，スレッドが内部で動作しているアプリケーションは，スレッドを移送直前に停止させることで移送可能となる．また，ファイルやソケットの入出力に関しては，移送直前に停止させる方法の他に，スタブを用いて移送後の継続利用を可能にする方法も考えられる．

2.3.2 開発者への負担

サービスローミングフレームワークには，ローミング型アプリケーションが不可欠である．しかし，ローミング型アプリケーションの開発者は，アプリケーション本来の機能に加えて移送機能の実装も行わなければならない．そのため，アプリケーション開発における開発者の負担が大きくなる．特に，計算機固有リソースを利用している場合，それらのリソースに対する移送前後の処理を記述する必要がある．

また，非ローミング型アプリケーションに対し移送機能を追加実装する必要があるため，既存のアプリケーションをそのままでは利用できない．既存のアプリケーションを移送可能にするためには，新規に作成されるアプリケーションと同様に，開発者の手によって移送機能を追加実装する必要がある．

2.3.3 ローミング型アプリケーションの実装手法

サービスローミングフレームワークでは，移送トリガによって移送命令が発行されるため，アプリケーションは外部的な要因を移送動開始イベントとして扱う．そのため，アプリケーションには外部からの命令によって移送開始する機能を備える必要がある．また移送時には，2.3.1 項で挙げた計算機固有リソースの問題に対処する必要がある．

アプリケーションに移送機能を実装する方法として，コールバックメソッドが多く用いられる．これは，継承やインタフェースによって特定のメソッドをアプリケーションに実装し，移送時にはそれらのメソッドが呼ばれるという手法である．コールバックメソッドの中に計算機固有リソースに対処する処理を記述することで，稼動状態での移送に関してアプリケーションごとの対応が可能となる．

本研究では，移送時のコールバックメソッドを実装し，なおかつ前項に示した移送時の処理を実装したアプリケーションをローミング型アプリケーションと呼ぶ．また，非ローミング型アプリケーションをローミング型アプリケーションへ再生成することをローミング対応化と呼ぶ．

2.3.4 本研究で構築するサービスローミングフレームワーク

本研究の目的として，開発コストの低いサービスローミングフレームワークの実現を掲げる．この目的を達成するため，非ローミング型アプリケーションをローミング

対応化するアプリケーション変換機構を提案する．アプリケーション変換機構は，移送機構にてアプリケーションをロードする際にアプリケーションのローミング対応化を行う．そのため，移送機構の一部として組み込まれる．

本研究では，ローミング型アプリケーションの開発に着目するため，移送トリガ生成機構は本研究の対象外とする．本研究で構築するサービスローミングフレームワークの領域を図 2.2 に示す．

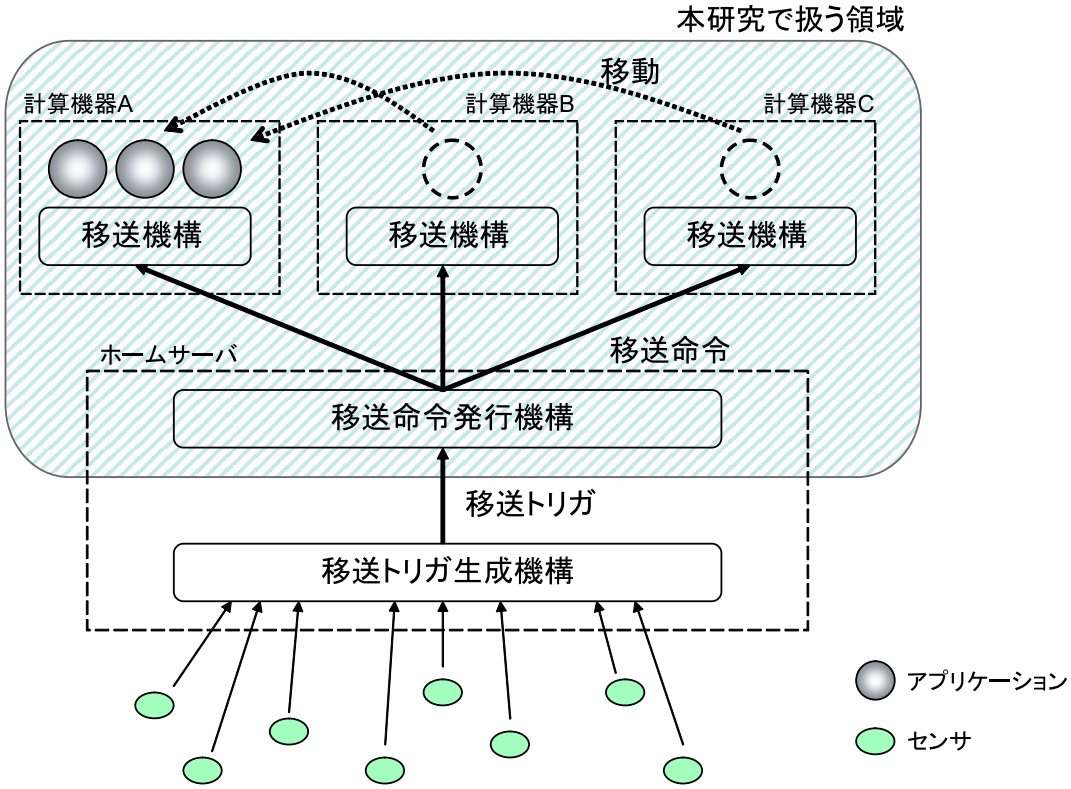

図 2.2: サービスローミングフレームワークにおける本研究の対象領域

また，アプリケーション変換機構は，新規アプリケーションではなく既存アプリケーションの移送可能化を変換対象とする．なぜなら，既存アプリケーションの移送可能化が実現することで，移送可能なアプリケーションの新規作成も実現するためである．

2.4 本章のまとめ

本章では，サービスローミングについて述べ，サービスローミングの構成について解説した．また，本研究で扱う問題点についても述べた．

サービスローミングフレームワークは，移送トリガ生成機構，移送命令発行機構，移送機構によって構成される．移送トリガ生成機構がセンサ情報を元に移送トリガを生成し，移送命令発行機構に送信する．移送命令発行機構は移送トリガを解析し，適切

な移送機構へ移送命令を送信する．移送機構は，移送命令に応じ，アプリケーションを他の移送機構へ移送する．

サービスローミングフレームワークで利用するアプリケーションには，計算機固有リソースに対応するための移送機能の実装が必要である．そのため，開発者へは負担が掛かり，また既存のアプリケーションを利用する場合に追加実装が要求される．

そこで本研究では，アプリケーションをローミング対応化するアプリケーション変換機構を提案し，この機構を含むサービスローミングフレームワークを構築する．本フレームワークを実現することにより，開発者へのコストが軽減し，また追加実装を行わずに既存アプリケーションの利用も可能となる．

第3章　アプリケーション変換機構を備えたサービスローミングフレームワーク

本章ではまず，アプリケーションを種類別に分類し，本研究が対象とするアプリケーション群を選定する．次に，アプリケーションをローミング対応化する際に求められる機能要件について考察し，本研究で用いるローミング対応化機構について解説する．

3.1 本研究の対象とするアプリケーション

本節では，アプリケーションを分類し，ローミング対応化を行うアプリケーションを明確にする．まず分類基準を定め，次にアプリケーションの分類例を示す．その後，本研究の対象とするアプリケーションについて述べる．

3.1.1 分類基準

アプリケーションの分類基準として，そのアプリケーションが自己完結型であるか，それとも連携型であるのかという点に注目する．また，状態保持型であるか，それとも状態非保持型なのかという点に注目する．この 2 点はアプリケーションの移送時処理に追加的な実装の必要性を決定する基準軸である．以下に詳細を示す．

自己完結型/連携型

アプリケーションが他のアプリケーションと連動せずに動作しているかどうかを示す．自己完結型アプリケーションとは，実行環境より提供される機能のみを用いて動作するアプリケーションである．自己完結型アプリケーション例には，エディタや電卓のような単独動作のアプリケーションが挙げられる．

逆に，連携型アプリケーションとは，他のアプリケーションと連携しながら動作するアプリケーションである．連携型アプリケーションは，Web ブラウザやチャットアプリケーションが挙げられる．

自己完結型アプリケーションは，移送先の実行環境から動作に必要なリソースが提供されれば，移送後直ちに実行を再開できる．これに対し連携型アプリケーションは，他のアプリケーションとの連携を，移送前に一度中断し移送後に再開する必要がある．そのため連携型アプリケーションは，移送時の処理が複雑になる．

状態保持型/状態非保持型

アプリケーションが実行状態を保持するかどうかを示す．状態保持型アプリケーションとは，ネットワークセッションやファイル入出力などのリソース状態に深く依存するアプリケーションである．例えば，チャットは通信相手とのセッションを保持していなければならないため，状態保持型アプリケーションであると言える．これに対し，電卓や時計アプリケーションはリソース状態にそれほど依存しないため，状態非保持型アプリケーションであると言える．

状態保持型アプリケーションは，移送前後で状態が変化した場合，移送後正常に再開できなくなる可能性がある．例えば，チャットアプリケーションにおいてセッション状態が変化すると，そのセッションによって行われていた通信は異常と判断され終了される．このため，状態保持型アプリケーションの移送には，移送前と移送後の状態に一貫性を保つ必要がある．

3.1.2 アプリケーションの分類

前項で示した分類基準を元にした既存アプリケーションの分類例を図 3.1 に示す.

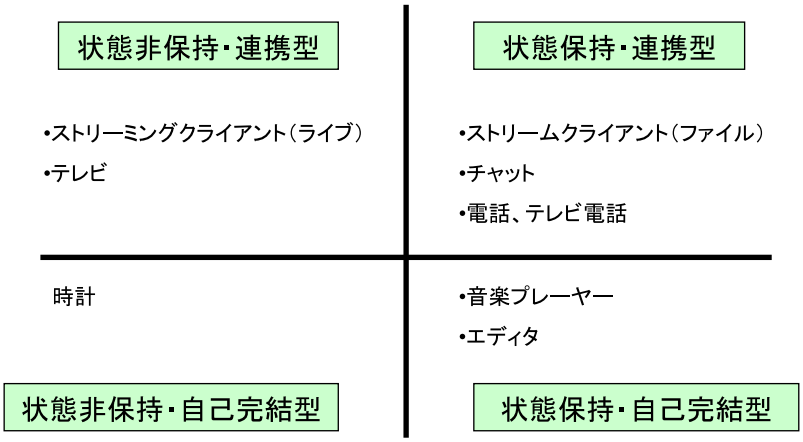

図 3.1: アプリケーションの分類例

分類によって得られた 4 つのカテゴリをアプリケーション群と呼ぶ．以下に，各アプリケーション群の具体例について述べる．

1. **状態保持・連携型**

   状態を保持し，複数のアプリケーションで連携して動作する．具体例としては，メディアファイルのストリームを受信するようなストリームクライアントや，チャットアプリケーション，テレビ電話アプリケーションなどが挙げられる．

2. **状態非保持・連携型**

   状態を保持しないが，複数のアプリケーションで連携して動作する．ライブ中継を受信するストリームクライアントやテレビなどが挙げられる．

3. **状態保持・自己完結型**

   状態を保持するが，他のアプリケーションと連携せずに動作する．ローカルホストのファイルを開く音楽プレーヤーやエディタなどが挙げられる．

4. **状態非保持・自己完結型**

   状態を保持せず，他のアプリケーションと連携せずに動作する．電卓や時計アプリケーションなどが挙げられる．

3.1.3 本研究の対象とするアプリケーション群

本研究では，4 種類すべてのアプリケーション群へのローミング対応化を行う．サービスローミング時に最も処理が複雑になるアプリケーション群は，状態保持・連携型アプリケーション群である．このアプリケーション群に属するアプリケーションをローミング対応化するには，移送機能の中に適宜処理を埋め込まなければならない．

そこで本研究では，状態保持・連携型アプリケーション群のローミング対応化を目標とし，状態非保持・自己完結型，状態非保持・連携型，状態保持・自己完結型の順に各アプリケーション群に対するローミング対応化を実現する．

3.2 機能要件

本節では，前述したアプリケーション分類を考慮し，アプリケーションのローミング対応化するための機能要件について述べる．

自動変換機能

ローミング型アプリケーションを作成する際，開発者がローミングに関する知識を持っているとは限らない．そのため，アプリケーションのローミング対応化を自動で行う必要がある．ローミング対応化を自動で行うことにより，開発者は本フレームワーク上でアプリケーションを起動するだけで移送機能を追加できる．

状態維持機能

状態保持型アプリケーションが移送される際，移送前後の状態に一貫性を保証する必要がある．状態維持機能は状態の一貫性を保証する．この機能を実現することにより，アプリケーションは，移送後も移送前と同じ状態で動作できる．

通信保証機能

連携型アプリケーションに対しては，移送前と同じ連携先アプリケーションとのコネクションを維持もしくは再開する機能が必要である．これは，TCP のような双方向型通信だけでなく，UDP のような一方向型通信に対しても同じことが言える．通信保証機能は，アプリケーションが移送後も再び通信を行えることを保証する．

3.3 アプリケーションローダとミドルウェアによるローミング機構

本節では，ローミング対応化機構の実現方法について述べる．本フレームワークは，アプリケーション変換機構とアプリケーション移送機構で構成される．2 つの機構で構成する理由は，移送機能の追加と移送時の処理を分離することで，拡張性が増した

めである．機能要件のうち，自動変換機能をアプリケーション変換機構によって実現し，状態維持機能と通信保証機能をアプリケーション移送機構によって実現する．

本節ではまず，本研究で構築するローミング対応化機構の全体構成について解説する．次に，アプリケーション変換機構とアプリケーション移送機構について考察する．

3.3.1 サービスローミングフレームワークの全体構成

本フレームワークは，アプリケーション変換機構とアプリケーション移送機構から構成される．アプリケーション変換機構は，アプリケーションにコールバックメソッドを追加し，その中に移送時処理を記述する．アプリケーション移送機構は，移送命令発行機構から命令を受信すると，アプリケーションのコールバックメソッドおよびその中に記述された移送時処理を実行する．状態保持型アプリケーションおよび連携型アプリケーションの移送時の対処は，変換機構が組み込んだ機能を移送機能が実行することにより実現される．

3.3.2 アプリケーション変換機構

本項では，移送対応化を行うアプリケーション変換機構について考察する．まず，アプリケーション変換手法を比較する．続いて，実現するアプリケーションローダについて解説する．

アプリケーション変換手法の比較

仮想マシン上で動作するアプリケーションは，コンパイラによってソースファイルから中間ファイルへ変換され，仮想マシンによって中間ファイルがネイティブコードへ解釈される．中間ファイルは仮想マシンで実行される命令が書かれた実行コードから成る．そのため，アプリケーションに機能を追加する方法は，(1) ソースコードへの追加，(2) 仮想マシンによる追加，(3) 実行コードへの追加，の 3 通りが可能となる．

1. **ソースコードによる機能追加**

   ソースコードで追加する場合は，ソースコードを解析して機能を追加する．この手法では，アプリケーション開発者がソースコードを修正する必要がある．そのため，アプリケーションの内部構造やアルゴリズムに精通している必要がある．また，ソースコードを変更するため，もう一度コンパイルを行う必要が生じる．

2. **仮想マシンによる機能追加**

   仮想マシンで追加する場合は，実行コード実行時に処理を行うよう仮想マシンを拡張する．この方法では仮想マシンを修正するため，アプリケーション単位での機能追加が困難になる．また，セキュリティ強化やバグ修正による Java 仮想マ

シンのアップデートが頻繁に行われるため，その都度仮想マシンを拡張しなければならない．

3. **実行コードによる機能追加**

   実行コードで追加する場合は，仮想マシンへの実行コードロード時に機能を追加する．実行コードに処理を加えるため，再コンパイルを行う必要がない．また，中間ファイルは実行コードによって形成されるため，対応できるアプリケーションの数も多い．さらに，実行コードで対応するため，アプリケーション単位での対応が可能である．

以上より，本研究ではアプリケーション変換手法として，実行コードへの機能追加を用いる．本フレームワークは，機能要件で述べた変換の自動化を実現するため，アプリケーションロード時に実行コードへの移送機能追加を行う．開発者者は，まず本フレームワークを用いてアプリケーションの中間ファイルをロードすることにより，そのアプリケーションは移送可能となる．

ローミング対応化アプリケーションローダ

本フレームワークのアプリケーション変換機構は，実行コードへの機能追加を行うアプリケーションローダとして組み込む．開発者がこのアプリケーションローダからアプリケーションをロードすることで，アプリケーションに移送機能が追加される．移送可能化を自動化するため，このアプリケーションローダには解析機能と変更機能を持たせる．以下に各機能の詳細を示す．

アプリケーション解析機能

アプリケーションを解析し，そのアプリケーションの所属するアプリケーション群およびその原因となる内部変数を把握する．具体的には，ロードされた実行コードを解析し，参照する変数を把握する．

アプリケーション変更機能

アプリケーション解析機能によって得られた情報を元に，アプリケーションの変更を行う．変更点は，(1) 移送時に移送機構より呼ばれるコールバックメソッドの追加，(2) 変数への処理をコールバックメソッドの中へ記述，の 2 点である．

3.3.3 アプリケーション移送機構

本項では，アプリケーション移送機構について考察する．アプリケーション移送機構は状態保持型アプリケーションへの対処を行う状態維持機能と，連携型アプリケーションへの対処を行う通信保証機能を持つ．

計算機固有リソースとして，にてスレッド，ファイルへの入出力，ソケットへの入出力，デバイスへの入出力を挙げた．状態維持機能と通信保証機能が扱うリソースを表 3.1 にまとめる．

状態維持機能	スレッド，ファイルへの入出力，デバイスへの入出力
通信保証機能	ソケットへの入出力

表 3.1: 各機能が扱う計算機リソース

次項より，各機能について解説する．まず，状態維持機能について述べる．続いて，通信保証機能について述べる．

移送前後の状態同期

状態保持機能は，移送後のアプリケーションを移送前と同じ状態にする．具体的には，アプリケーションの内部変数を移送前の状態と同じ状態にする．

この機能を実現するためには 2 つの方法が考えられる．まず，移送機構による状態のキャッシングが挙げられる．この方法ではまず，移送機構が移送前にアプリケーションのすべての状態を抽出し，アプリケーションと共に移送先へ転送する．その後，移送先でのアプリケーション再開時に状態を元に戻す．

また，アプリケーションの複製による移送後の同期が挙げられる．この方法ではまず，移送機構は移送時にアプリケーションを終了するのではなく，アプリケーション複製を生成する．そして，移送を行い，移送先で複製されたアプリケーションが稼働し始める．ここで移送元で動作しているオリジナルのアプリケーションと，移送先で動作している複製されたアプリケーションを同期する．

前者の方法は，アプリケーションの移送時間が長くなると，サービス価値の低下が生じる可能性がある．例えば，状態と実時間が連携しているアプリケーションでは，移送時間が移送後の状態と本来あるべき状態との差違となる．逆に後者の方法は，移送した後にオリジナルと複製の間で同期を行うため，この差違は最小限になる．本研究では，後者の方法を用い，状態維持機構を作成する．

移送中の通信保証

通信保証機能を実現する方法としては，MobileIP[12] のような通信の転送が挙げられる．この方法では移送時，移送機構がアプリケーションのセッションを仲介し，移送中もセッションを保持する．移送後は移送先へ通信データを転送する．

この機構では，移送機構によって提供されるスタブによってこの機能を実現する．アプリケーションは，通信時にこのスタブを生成して呼び出すことで，同セッションによる移送後の通信を可能にする．

3.4 関連研究

MobileSpaces

MobileSpaces[11] は，国立情報学研究所の佐藤氏によって開発されたモバイルエージェントシステムである．MobileSpaces では RFID を用いて，ユーザの位置に応じたエージェント移送を行っている．アプリケーション例としては，RFID リーダが接続された照明機器のコントローラを，RFID タグのついた PDA へ移送されるアプリケーションが挙げられる．また，デスクトップを別計算機へ移送するアプリケーションも挙げられる．

MobileSpaces で利用するアプリケーションは，MobileSpaces より提供される API を実装することで行われる．しかし，開発者はこの API について把握していなければならない．

AOP(アスペクト指向プログラミング)

AOP[13] は，アプリケーションの挙動の中に処理を埋め込み，既存のアプリケーションを拡張するプログラミングスタイルである．AOP では，あるアプリケーションの状態 (Pointcut) に，特定の処理 (Advice) を織り込むことによって機能を追加する．

Java で AOP を実現したライブラリとして AspectJ[14] がある．AspectJ では，Java バイトコードに AOP の処理を埋め込むことで，Advice を織り込んでいる．そのため，開発者は既存のアプリケーションコードを変更することなく処理を追加できる．

しかし，AOP はアプリケーションのあるアスペクトに注目するという性質上，アプリケーションの挙動とまったく関係なく発生する処理を埋め込めない．また，AOP は独自の言語を用いるため，開発者は AOP の言語を取得する必要がある．そのため，本研究の目指すアプリケーション変更の自動化は行えない．

Thread Migration

Thread Migration[15] は，Java 仮想マシンを用いたスレッドの計算機間移送を実現している．移送するスレッドに関するスタック情報を Java 仮想マシンから取り出し，移送先の Java 仮想マシンにてスレッドを同じ状態で再開させている．この機能は，アプリケーションのバイトコード変換によって実現されるため，ソースコードへの変更を要求しない．

3.5 本章のまとめ

本章では既存のアプリケーションを分類した．また，アプリケーション作成に求められる機能要件について考察し，本研究で用いるアプリケーション変換手法について

解説した．

本研究では，アプリケーション変換機構とアプリケーション移送機構で構成される．アプリケーション変換機構は，ローミング対応化を実行コードレベルのアプリケーション変換により行う．また，アプリケーション移送機構は，移送時における計算機固有リソースへの対応を行う．

第4章　アプリケーション変換機構を備えたサービスローミングフレームワークの設計

本章では，本研究で実現するサービスローミングフレームワークの設計について述べる．

本フレームワークはアプリケーションのローミングを行うミドルウェアと，アプリケーションのローミング対応化を行うアプリケーションローダとで構成される．

4.1 設計方針

本研究で実現するサービスローミングフレームワークを，ローミング対応化を行うアプリケーションローダと，実際の移送を行うミドルウェアとに分けて設計する．アプリケーションローダを Sararizer と呼び，移送を行うミドルウェアを Sarari と呼ぶ．Sararizer は Sarari の一部に含まれる．Sararizer および Sarari の全体構成を図 4.1 に示す．

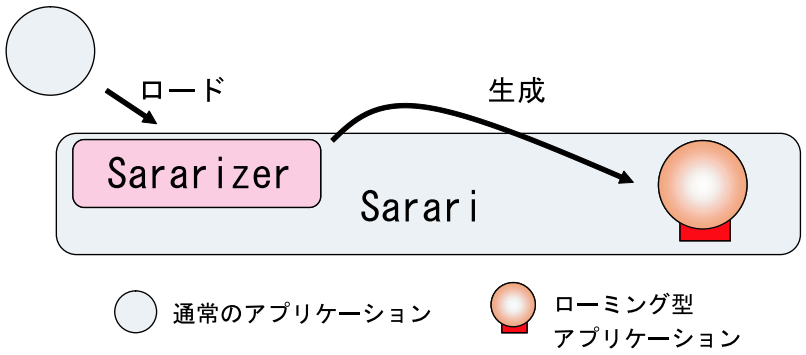

図 4.1: Sararizer および Sarari の全体像

次項より，Sararizer と Sarari について概要を述べる．

4.1.1 Sararizer

アプリケーションを移送可能化させるローダである．Java では，アプリケーションはクラスファイルより構成されるため，変換対象はクラスファイルとなる．クラスファイルがロードされると，Java 仮想マシンでは実行コードとして扱われる．Sararizer はその実行コードを解析し，変更を加える．これを実現するため，アプリケーション解析機能とアプリケーション変換機能を備える．

4.1.2 Sarari

Sararizer によって移送可能化されたアプリケーションを，実際に移送するミドルウェアである．Java では，アプリケーションが実行されると，クラスファイルからオブジェクトが生成される．Sarari は外部からの移送命令を受信し，その命令に応じてオブジェクトを別計算機で動作する Sarari へと移送する．また，アプリケーションが状態保持型もしくは連携型であった場合，オブジェクト移送時にアプリケーションに応じた処理を実行する．

4.2 SarariInterfaceの設計

本節では，Sarari によって呼び出されるインタフェースである SarariInterface の設計について解説する．

アプリケーション移送時，Sarari によって呼び出されるオブジェクトの処理は，移送開始時の処理と移送完了時の処理である．これらにアプリケーションの終了を加えた 3 種類のコールバックメソッドを SarariInterface メソッドとして定義する．

表 4.1 に処理とメソッドの対応を示す．メソッド名は，アプリケーションに元来備わっている名前との衝突を回避するため，dispatch，arrive，finish のような単純な名前ではなく，beforeSarariMigration，afterSarariMigration および sarariEndOfLife と名付ける．Sarari はこの各メソッドを呼び出すことにより，アプリケーション移送時の処理および終了処理を行う．

移送前の処理	*public void* beforeSarariMigration()
移送後の処理	*public void* afterSarariMigration()
終了処理	*public void* sarariEndOfLife()

表 4.1: SarariInterface のメソッド

また，異なる Sarari 上で複数の同一オブジェクトが動作する可能性もあるため，オブジェクト識別子を SarariInterface 内で定義する．オブジェクト識別子は，そのオブジェクトを生成した計算機の IP アドレス，そのオブジェクトの元となるクラスファイル名および生成された時間を用いて定義する (図 4.2)．Sarari がオブジェクト識別子の二重定義を防ぐことで，ネットワーク上での名前衝突を回避できる．

計算機の IP アドレス ： クラスファイル名 ： 生成された時間

図 4.2: オブジェクト識別子

4.3 Sararizerの設計

本節では，Sararizer の設計について解説する．Sararizer は，変換定義リスト，解析部，変更部からなる (図 4.3)．

解析部がロードされた実行コードを変換定義リストを元に解析する．その結果を元に変更部が実行コードを変更する．次項より詳細を述べる．

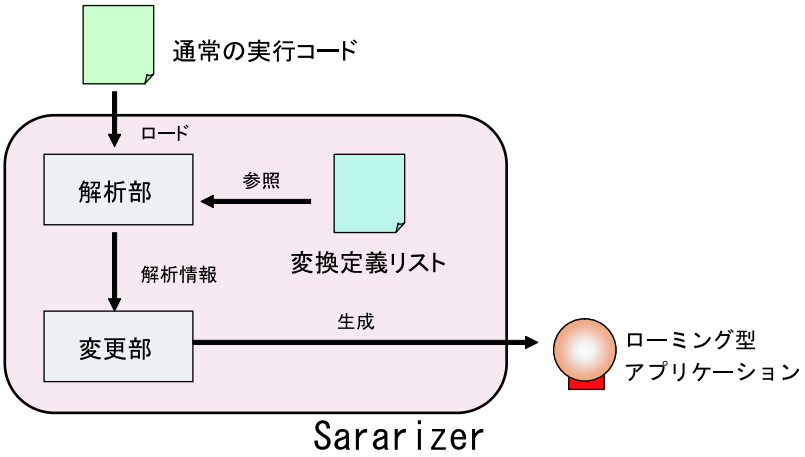

図 4.3: Sararizer の構成

4.3.1 変換定義リスト

Sararizer は，計算機固有リソースの変換定義リストを持つ．このリストには，変換対象となる内部変数名およびその変更方法を記述する．

変換対象となる内部変数は，移送不可能な変数とする．Java では Serialize インタフェイスを実装していないクラスを示す．これには，入出力ストリーム，スレッド，ソケットなどの計算機固有リソースを利用した変数が含まれる．具体例を以下に示す．

スレッド

実行中のスレッドは，計算機のメモリにアクセスしているため移送できない．そのため，オブジェクトがスレッドを所持していた場合，移送前にスレッドを終了し，移送後に新たなスレッドを生成する必要がある．該当する Java クラスには，Thread クラスがある．

入出力

入出力ストリームは，仮想マシンから OS のネイティブ API を呼び出すため移送できない．入出力ストリームがアクセスする対象には，ファイル，ネットワークおよび標準入出力が挙げられる．該当する Java クラスには，OutputStream クラスや InputStream クラスなどが挙げられる．

ネットワーク

ネットワークへのアクセスは，OS 内部からネットワークインタフェースを通して行われる．そのため，アプリケーションに対する仮想的なネットワークインタフェースの役割を果たすソケットは，移送できない．該当する Java クラスには，Socket クラスや ServerSocket クラスがある．

以上を表 4.2 に整理する．

移送不可能な要素	該当する Java クラス例
IO	Output(Input)Stream クラス，Writer(Reader) クラス　等
スレッド	Thread クラス
ネットワーク	Socket クラス，ServerSocket クラス　等
OS ネイティブな GUI	JFrame クラス，JPanel クラス　等

表 4.2: 移送不可能な要素

```
<?xml version="1.0">
<TARGETCLASS name="対象クラス名">
        <FINISH>終了メソッド名</FINISH>
<RECREATE>再生成メソッド名</RECREATE>
</TARGETCLASS>
```

図 4.4: 変換定義リストの記述法

移送不可能な要素に対しては，移送前に終了し，移送後に再生成する．変換定義リストへは，各要素の終了メソッドと再生成メソッドを定義する．定義には XML を用いる．図 4.4 に定義手法を示す．終了メソッドは <FINISH> </FINISH> タグで定義し，再生成メソッドは <RECREATE></RECREATE> タグで定義する．メソッドに渡される引数は，解析部と変更部によって渡される．

4.3.2　解析部

解析部はロードされたクラスファイルの実行コードを解析する．解析部は予め，クラスファイルがロードされる前に変更定義リストを読み込み，変更対象となるクラスを把握する．クラスファイルがロードされると，そのクラスファイルに書かれた実行コードを解析し，その結果を変更部に渡す．解析は以下の 2 点に焦点を絞り行われる．

- 変換定義リストに記されたクラスの変数が存在するか
- SarariInterface メソッドと名前が衝突するメソッドをもっているか

変換定義リストに記されたクラスの変数が存在した場合，そのクラス名および変数名を変更部へ伝達する．また，メソッド名の解析中に衝突が起きた場合はロードを中止する．

4.3.3 変更部

変更部は，実行コードに SarariInterface によって定義されたメソッドを加え，解析部より得られた変数情報を元に処理を追加する．以下に手順を記す．

1. SarariInterface クラスファイルの実行コードからメソッドを抽出する
2. 変換対象の実行コードに SarariInterface を実装し，メソッドを加える
3. 解析部から得られた変数およびメソッドを，該当する SarariInterface のメソッド内に追加する

4.4 Sarari の設計

本節では，Sarari の設計について解説する．Sarari の構成図を図 4.5 に示す．

Sarari は大きく分けて，オブジェクトの送受信を行う送受信部，外部の移送命令発行機構との受け口である移送命令受信部から構成される．また，オブジェクトの管理を行う管理部を持つ．管理部がオブジェクト生成時に Sararizer と連携することで，アプリケーションのローミング対応化が行われる．

さらに Sarari は，状態保持型アプリケーションを移送する際に状態維持部を生成し，連携型アプリケーションを移送する際に通信保持部を生成する．状態維持部と通信保持部はアプリケーションごとに生成される．次項より各部設計の詳細について解説する．

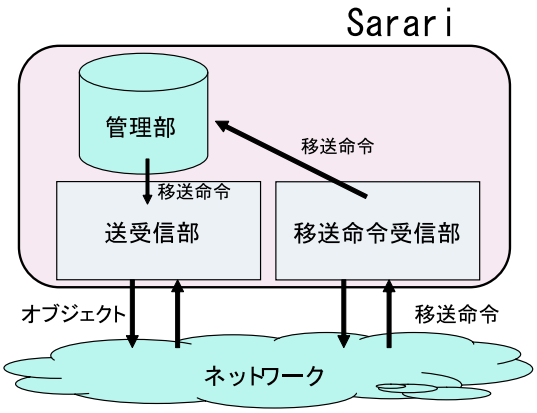

図 4.5: Sarari の構成図

4.4.1 送受信部

SarariInterfaceが実装されたオブジェクトの移送を行う．送受信部は，オブジェクト送信を行うSarariObjectSenderと，受信を行うSarariObjectReceiverによって構成される．SarariObjectSenderは送信メソッドを，SarariObjectReceiverは受信メソッドを持つ．これらの対応を表4.3にまとめる．

送信	SarariObjectSenderクラス	*public void* sarariObjectSend(*SarariInterface s*)
受信	SarariObjectReceiverクラス	*public SarariInterfaces*arariObjectReceiver()

表 4.3: Sarari送受信部のクラスおよび送受信メソッド

送受信部はオブジェクトを移送する際，Sararizerによってオブジェクトへ追加されたメソッドを実行する．移送手順を以下に示す．1，2，3は送信側Sarariにて行われ，4，5は受信側Sarariにて行われる．

1. オブジェクトがSarariInterfaceを実装しているかを検証する
2. オブジェクトのbeforeSarariMigrationメソッドを実行する
3. sarariObjectSenderにてオブジェクトを送信する
4. sarariObjectReceiverでオブジェクトを受信する
5. オブジェクトのafterSarariMigrationを実行する

4.4.2 移送命令受信部

移送命令受信部は，移送命令発行機構より送られてくる命令を解析し，オブジェクトの移送を行う．移送命令には，オブジェクトの名前と送信先のSarariのアドレスが含まれる．Sarari上に対象となるオブジェクトが存在しない，もしくは移送に失敗した場合，移送命令発行機構に対し，エラーを返す．また，複数の移送命令を受け取った場合は移送を排他的に行い，命令の衝突を回避する．命令はキューに入れられ，順に実行される．

4.4.3 管理部

管理部はオブジェクトの管理を行う．移送命令受信部から渡される移送命令に応じ，オブジェクトを送受信部へ渡す．また，オブジェクト生成時にSararizerを呼び出すことにより，そのオブジェクトのローミング対応化を行う．

4.4.4　状態維持部

状態維持部は，オブジェクトの移送時に動的に生成される．移送元 Sarari の状態保持部は，移送したオブジェクトが実行を再開すると，移送元 Sarari で動作しているオブジェクトの状態を取り出し，移送先 Sarari の状態維持部へ渡す．移送先 Sarari の状態維持部は，受け取った状態と実行を再開したオブジェクトの状態を同期させる．

4.4.5　通信保証部

通信保証部は，オブジェクトが通信を開始した時に生成される．通信保証部は，通信のプロキシを行う．移送時は，オブジェクト側の接続を切断するが，通信相手との接続は切断しないため，移送されたオブジェクトが移送元 Sarari の通信保証部にアクセスすれば再び接続を張れる．移送中の通信データはバッファリングされ，接続が再開されるとオブジェクトに送られる．

通信保証部は，Sarari の API として提供され，Sararizer によるローミング対応化の再，通常用いられるクラスおよびメソッドと置き換えられる．例えば，Socket であれば SarariSocket に置き換えられ，オブジェクトは SarariSocket を用いて通信を開始する．

4.5　本章のまとめ

本章では，ローミング対応化機能を備えたサービスローミングフレームワークを構成する Sararizer および Sarari の設計について述べた．

Sararizer は，実行コードを解析し，Sarari 上で動作するアプリケーションの共通インタフェースである SarariInterface インタフェースおよびメソッドを実行コードに加える．Sarari は，移送時の固有リソースへの処理を SarariAPI を通して行う．Sararizer はアプリケーションの内部変数を解析し，SarariAPI を通常の内部クラスと置き換える．

第5章　サービスローミングフレームワーク構築機構の実装

本章では，アプリケーションのローミング対応化を備えたサービスローミングフレームワークのプロトタイプ実装について述べる．

5.1 実装環境

本節では，実装環境について述べる．本研究では，開発環境として Java を用いた．また，Java の実行コードであるバイトコードを修正するライブラリとして BCEL(Byte Code Engineering Library)[16] を用いた．表 5.1 に詳細を示す．

実装言語	Java1.4.2
バイトコード処理ライブラリ	BCEL5.0

表 5.1: 実装環境

5.1.1 Java のバイトコード

本研究では，Java のバイトコード [21] に修正を加え，ローミング対応化を行う．バイトコードは主に以下のデータで構成される．

コンスタントプール Java プログラム内で用いられる定数，クラスへの参照，フィールドへの参照，メソッドへの参照などがテーブル形式で格納されている．

アクセスフラグ 当該クラスのアクセス権限をフラグで指定している．

クラス名 当該クラスのクラス名を指定している．

親クラス名 当該クラスの親クラス名を指定している．

インタフェース 当該クラスが実装しているインタフェースの個数と各インタフェースの型を指定している．

フィールド 当該クラスが持つフィールドの個数と，各フィールドのアクセスフラグ，フィールドの型，フィールド名，属性リストを指定している．

メソッド 当該クラスが持つメソッドの個数と，各メソッドのアクセスフラグ，メソッド名，メソッドの型，実行内容が記述されたインストラクションセット (例外ハンドラも含める) などを指定している．

5.2 実装概要

本節では，プロトタイプとして実装したサービスローミングフレームワークについて解説する．本論文では，プロトタイプ実装として，状態非保持・自己完結型アプリケーションをローミング対応化させるサービスローミングフレームワークを構築した．次項より，今回実装した Sararizer および Sarari についての概要を述べる．

5.2.1 Sararizer の実装概要

本論文で実装した Sararizer は，非ローミング型アプリケーションのクラスファイルからローミング対応のオブジェクトを生成する．まず，クラスファイルを読み込み，そのクラスファイルを変更定義ファイルを参考に解析する．次に，クラスファイルに SarariInterface および変更定義ファイルに書かれたメソッドを書き込み，オブジェクトを生成する．

Sararizer のクラス構成を表 5.2 に示す．これらのクラスは，sararizer パッケージに含まれる．

Sararizer	Sararizer のメインクラス
SararizreLoader	クラスロードを行うクラス
SararizerParser	バイトコード解析クラス
SararizerModifier	バイトコード変更クラス

表 5.2: Sararizer のクラス構成

次に，各クラスの概要について述べる．Sararizer クラスは，Sararizer パッケージのメインクラスである．SararizerLoader クラスは，主にクラスファイルの読み込みや Class オブジェクトの生成を行う．SararizerParser クラスは，バイトコードを解析するクラスであり，設計で述べた解析部に相当する．SarariModifier クラスは，バイトコードを変更するクラスであり，変更部に相当する．

5.2.2 Sarari の実装概要

本論文で実装する Sarari は，Sararizer によってローミング対応化されたオブジェクトの移送機能を持つ．Sararizer によって生成されたオブジェクトを管理し，移送命令に応じて別の計算機で動作する Sarari へオブジェクトを移送する．

Sarari のクラスはすべて sarari パッケージに含まれる．表 5.3 に主なクラスを示す．

Sarari	Sarari のメインクラス
SarariManager	オブジェクトの管理を行うクラス
AppliSender	オブジェクトの送信を行うクラス
AppliReceiver	オブジェクトの受信を行うクラス

表 5.3: Sarari のクラス構成

次に，各クラスの概要について述べる．Sarari クラスは，本フレームワークのメインクラスである．SarariManager クラスは，オブジェクトの生成および管理を行う．ま

た，SarariManager クラスが Sararizer クラスを呼び出すことによって，ローミング対応化されたオブジェクトが生成される．AppliSender および AppliReceiver クラスは，オブジェクトの送受信を行う．

5.3 Sararizer

本章では，Sararizer の実装について述べる．Sararizer の機能要件は，クラスファイルをロードし，ローミング対応化を行うことである．ローミング対応化とは具体的に，SarariInterface を実装し，メソッドの中に移送前処理，移送後処理，終了処理を埋め込むことであった．

まず，Sararizer クラスについて述べる．次に SararizerParser が行う解析について述べ，最後に SararizerModifier が行う変更について述べる．

5.3.1 Sararizer クラス

Sararizer クラスは，アプリケーションのクラスファイルを読み込み，SarariParser クラスおよび SarariModifier クラスを通してローミング対応化されたオブジェクトを生成する．この作業は sararize メソッドを用いて行われる．図 5.1 に sararize() メソッドの詳細を示す．クラス変数である parser は SarariParser のインスタンスであり，modifier は SarariModifier のインスタンスである．

```
public SarariInterface sararize(Class target){
    Class newClass = null;
    if(this.parser.checkHaveSarariInterface(target)==false){
        SararizerChangeList list = this.parser.parse(target);
        JavaClass newJavaClass = this.modifier.addSarariInterface(target);
        newJavaClass = this.modifier.addInstructionIntoMethod(
            newJavaClass,list);
        newClass = this.loader.changeJavaClassToClass(
            newJavaClass, newJavaClass.getClassName());
    }else{
        newClass = target;
    }
    return this.makeSarariObject(newClass);
}

public SarariInterface sararize(String className){
    Class target = this.loader.getClass(className);
    return this.sararize(target);
}
```

図 5.1: Sararizer クラスの sararize メソッド

5.3.2 SararizerParser クラス

SarariParser は，クラスファイルを解析するクラスである．SarariParser が保持するメソッドを表 5.4 に示す．

public boolean checkHaveSarariInterface(Class classObj)	インタフェースの判別
public boolean checkHaveSarariInterface(String className)	インタフェースの判別
public Field getField(JavaClass jc)	フィールドの取得
public Field getFieldClassName(Field field)	フィールドのクラス名の取得
public JavaClass getJavaClass(Class classObj)	JavaClass クラスの取得
public Class getSuperClass(Class classObj)	親クラスの取得
public Class getSuperClassName(JavaClass jc)	親クラスの取得
public SarariChangeList getSuperClassName(Class classObj)	変更リストの取得

表 5.4: SararizerParser のメソッド

5.3.3 SararizerModifier クラス

SararizerModifier クラスは，クラスファイルを変更するクラスである．

SarariModifier クラスではまず，ローミング対応化の対象となるクラスへ SarariInterface を実装する．実装は SarariModifier の addSarariInterface メソッドで行われる．詳細を図 5.2 に示す．

```
public JavaClass addSarariInterface(Class targetClass) {
    try {
        JavaClass jc_target = Repository.lookupClass(targetClass);

        ClassGen cg_target = new ClassGen(jc_target);
        ConstantPoolGen cpg_target = new ConstantPoolGen(jc_target
                .getConstantPool());

        JavaClass jc_sarari = Repository.lookupClass(this.templateClass);
        ConstantPoolGen cpg_sarari = new ConstantPoolGen(jc_sarari
                .getConstantPool());

        Method methods[] = jc_sarari.getMethods();
        for (int i = 1; i < methods.length; i++) {
            MethodGen mg = new MethodGen(methods[i],
                                         jc_sarari.getClassName(),
                                         cpg_sarari);
            MethodGen newMethod = mg.copy(targetClass.getName(),
                                          cpg_target);
            cg_target.addMethod(newMethod.getMethod());
        }

        cg_target.setConstantPool(cpg_target);

        JavaClass iface = Repository.lookupClass(
                                Class.forName(this.ifaceName));
        cg_target.addInterface(iface.getClassName());

        JavaClass jc2 = cg_target.getJavaClass();
        jc2.setFileName(targetClass.getName());
        jc2.setConstantPool(cpg_target.getFinalConstantPool());

        return jc2;

    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

図 5.2: SarariModifier クラスの addSarariInterface メソッド

本論文では，クラスファイルへの SarariInterface および SarariInterface メソッドの実装を容易にするため，SarariInterface を実装したテンプレートクラスである SarariObjectTemplate クラスを作成した．図 5.3 に詳細を示す．SarariObjectTemplate クラスは，SarariInterface を実装し，SarariInterface のメソッドを備えたクラスである．

```
public class SarariObjectTemplate implements SarariInterface {

    public void beforeSarariMigration() {}

    public void sarariEndOfLife() {}

    public void afterSarariMigration() {}
}
```

図 5.3: SarariObjectTemplate

SarariModifier は，まずこのクラスからメソッドを抜き出し，ローミング対応化の対象となるクラスファイルに追加する．

5.4 Sarari

本節では Sarari の実装について述べる．Sarari は主に Sarari クラス，SarariManager クラス，SarariManager クラス，AppliSender(Receiver) クラスによって構成される．次項より各クラスについて解説する．

5.4.1 Sarari クラス

Sarari クラスは，本フレームワークのメインクラスである．Sarari クラスは，SarariManager クラスおよび ServerManager クラスを保持する．

また，他の Sarari との相互発見を行うため，Sarari に multicastDNS[17] を実装し，サービスディスカバリー機能を持たせた．multicastDNS の実装には JmDNS ライブラリ [18] を利用した．JmDNS は Apple の Rendezvous[19] との互換性を持つ．

この multicastDNS 機能を統括するため，RendezvousManager クラスを作成した．RendezvousManager クラスは Sarari が起動されると設定ファイルを読み込み，他の Sarari の発見を行う．図 5.4 に設定例を示す．

5.4.2 SarariManager クラス

SarariManager クラスは，オブジェクトの管理を行う．管理用クラスには SarariObjectHolder クラスが用いられる．オブジェクトの生成は，Sararizer によって行われる．

```
rendezvousType=_srr._tcp.local.
rendezvousName=sarari
```

図 5.4: Rendezvous(multicastDNS の設定例

Sararizer によって生成されたオブジェクトは，SarariObjectHolder に格納される．移送命令が来るとオブジェクトが取り出され，AppliSender クラスへ渡される．

また，SarariManager は外部からオブジェクトが送られてきた場合に備え，AppliReceiver クラスと連携を取るための SarariObjectHolderHandler インタフェースを実装している．SarariObjectHolderHandler インタフェースの詳細を図 5.5 に示す．AppliReceiver クラスは SarariObjectHolderHandler インタフェースのメソッドを用いてオブジェクトの格納を行う．尚，オブジェクトを取り出せるクラスを SarariManager に限定するため，このインタフェースはオブジェクトの取り出しメソッドを所持していない．

```
public interface SarariObjectHolderHandler {
    public void addSarariObjectIntoHolder(Object obj);
    public int getSize();
    public SarariInterface findSarariObjectFromHolder(int index);
}
```

図 5.5: SarariObjectHolderHandler インタフェース

5.4.3 AppliSender クラスおよび AppliReceiver クラス

AppliSender クラスおよび AppliReceiver クラスは，アプリケーションの送受信を行う．AppliSender クラスは，移送時に SarariManager によって生成される．AppliReceiver クラスは，Sarari 起動時に生成され，外部からの受信を待つ．

本フレームワークでは，アプリケーションの移送を，アプリケーション実行コードと状態の移送と定義した．この機能を実現するため，AppliSender クラスおよび AppliReceiver クラスは，FileInput(Output)Stream クラスおよび ObjectInput(Output)Stream クラスを用いてアプリケーションの移送を行う．オブジェクトの状態 (Object クラス) は，Serialize によってバイト配列に変換され，Deserialize によって再びオブジェクトの状態 (Object クラス) に戻される．以下にアプリケーションの移送手順を示す．

1. (送信側) 移送するオブジェクトのクラスを読み込み，送信する．
2. (受信側) オブジェクトのクラスを受信し，ファイルへ書き込む
3. (送信側) オブジェクトを Serialize し，送信する

4. (受信側) 受信したオブジェクトを Deserialize し，クラスファイルを参考にインスタンスを生成する

5.5 本章のまとめ

本章では，アプリケーションのローミング対応化を備えたサービスローミング機構のプロトタイプ実装について述べた．

第6章　評価

本章では，アプリケーション変換機構およびアプリケーション移送機構のプロトタイプとして実装されたSararizerおよびSarariについての評価を行う．

6.1 Sarariの基本性能

本節では，Sarari の基本性能の評価について述べる．本論文では特に状態維持機能に注目し，評価を行った．

状態維持機能は，アプリケーションの複製を送信し，移送元の計算機上で動作するアプリケーションとの同期を行う．そのため，移送元のアプリケーションは，複製されたアプリケーションが実行状態になるまで動作し続けている必要がある．すなわち，わずかな時間だけオリジナルのアプリケーションと複製されたアプリケーションが同時に動作していることになる．

本節では，この同時に動作している時間の計測結果を述べる．この時間が 0 より小さければ，Sarari の状態維持機能は機能要件を満たさないことになる．次項から計測環境および計測結果を述べる．

6.1.1 計測環境

計測用アプリケーションとして，簡単なストリーミングアプリケーションを用いた．このアプリケーションは，サーバから配信される動画を受信するアプリケーションである．受信された動画を各計算機の画面に表示している状態を実行状態とし，オリジナルのアプリケーションと複製されたアプリケーションが同時に動作している時間を計測した．

計測用計算機として，移送元の計算機と移送先の計算機の 2 台およびストリーミングサーバ 1 台を用意した．それぞれ端末 A，端末 B，サーバ端末とする．図 6.1 に各計算機の詳細を示す．

	端末 A	端末 B	サーバ端末
PC	ThinkPad X30	ThinkPad X30	ThinkPad X30
CPU	1.2GHz	1.2GHz	1.2GHz
メモリ	760MB	256MB	760MB
OS	Windows XP	Windows XP	Windows XP

表 6.1: 評価用端末の性能表

6.1.2 計測結果

端末 A および端末 B 上で，サーバ端末から送られてきた RTP パケットのシーケンス番号を，約 120 ミリ秒間隔で計測した．得られた値の中からアプリケーションが移送された部分を抜き出し，Sarari の状態維持機能を用いない場合の RTP シーケンス番号グラフを図 6.1 に，状態維持機能を用いた場合を図 6.2 に示す．

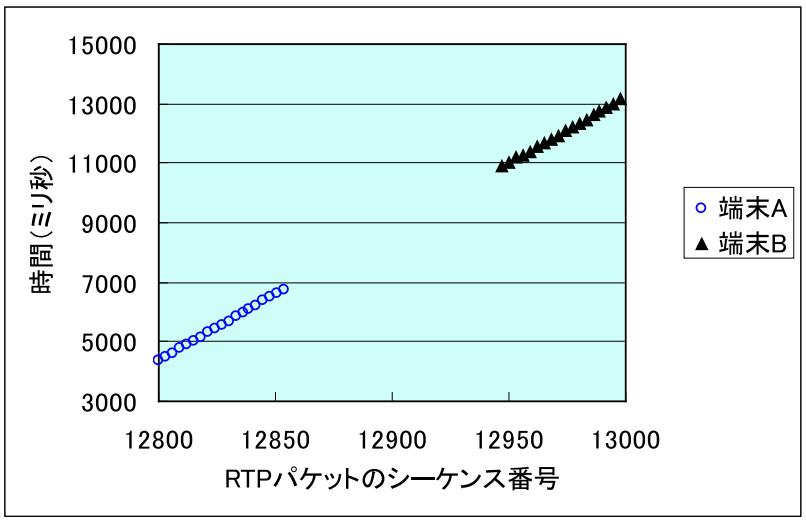

図 6.1: RTP のシーケンス番号の推移 (状態維持機能無し)

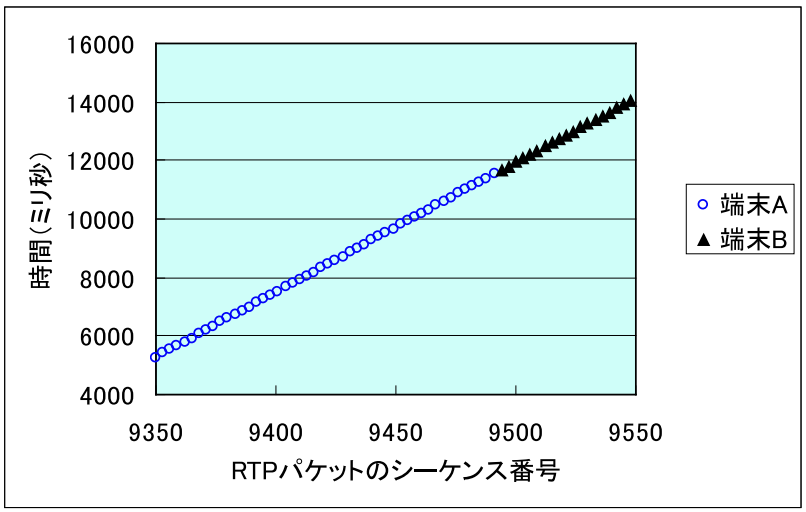

図 6.2: RTP のシーケンス番号の推移 (状態維持機能有り)

図 6.1 では，アプリケーションを端末 A から端末 B へ移送させている間，端末 A および端末 B で約 4.1 秒間 RTP パケットが得られていない．このことから移送には約 4.1 秒必要であることがわかる．これに対し，Sarari を用いた図 6.2 では，端末 A で計測された最後の RTP パケットのシーケンス番号と，端末 B で計測された最初の RTP パケットのシーケンス番号が同じであった．したがって，端末 A および端末 B で同じアプリケーションが 120 ミリ秒以上動作していると言える．

6.2 本章のまとめ

本章では，Sarari の状態維持機能について評価を行った．その結果，Sarari は状態維持機能を実現するための基本性能を満たしていた．

第7章　結論

本章では，今後の課題について述べ，最後に本論文をまとめる．

7.1 今後の課題

本節では，今後の課題について述べる．以下に課題を列挙する．

他のアプリケーション群への対応

Sarari の状態維持機能および通信保証機能を実装し，本プロトタイプで実装した状態非保持・自己完結型アプリケーション群以外のアプリケーション群に対してもローミング対応化を行う．

異なるネットワークへの移送

本プロトタイプでは，同一のサービスローミングフレームワークに属する移送機構同士でのアプリケーション移送を想定した．今後は，ホームネットワークやオフィスネットワークなどの異なるネットワーク間サービスローミングを行う．

セキュリティ・プライバシの考慮

本フレームワークでは，アプリケーションの持つ情報に対し，セキュリティやプライバシを考慮していない．もし移送するアプリケーションがクレジットカード情報や住所・氏名などの個人情報を保持していたとしても，本フレームワークではそれらの情報に対し特別な対処を行わない．したがって，移送するアプリケーションの持つ情報を保護し，セキュリティやプライバシを考慮したサービスローミグを行う必要がある．

7.2 まとめ

本論文では，はじめにサービスローミングフレームワークについて解説し，開発者へのサービスローミングを考慮したアプリケーション実装の負担について述べた．開発者への負担は，アプリケーションに移送機能を追加実装する際，そのアプリケーションが利用している計算機固有リソースへの対処を行わなければならないため生じる．

そこで本研究では，アプリケーションのローミング対応化機構を備えたサービスローミングフレームワークを提案した．本フレームワークの実現に向け，アプリケーションを分類し，各アプリケーション群への対処法について述べた．具体的には，アプリケーション変換機構で移送時に呼ばれるコールバックメソッドを実装し，移送時の各アプリケーション群への対処はアプリケーション移送機構で行うという対処法であった．

本研究では，アプリケーション変換機構を Sararizer，アプリケーション移送機構を Sarari として，設計およびプロトタイプ実装を行った．本論文の対象とするアプリケーション群には，状態非保持・自己完結型のアプリケーション群を選択し，簡単なアプリケーションを実装した．このアプリケーションを Sararizer にてローミング対応化し，Sarari 上で移送を行った．

謝辞

本研究を進めるにあたって貴重な御指導を賜りました，慶應義塾大学環境情報学部教授徳田英幸博士に深く感謝致します．また，的確かつ重要な御助言を頂きました，慶應義塾大学政策・メディア研究科助教授高汐一紀博士の御指導に感謝します．

慶應義塾大学徳田・村井・楠本・中村・南研究室の皆様には，折りに触れ貴重な示唆や御助言，御指導を頂きました．特に，徳田研究室の先生方や諸先輩方，及び，move! 研究グループの皆様には，深く感謝の意を表します．由良淳一氏， 守分滋氏，出内将夫氏， 丸山大佑氏には，本論文の執筆に当たって絶えざる御指導や励ましを頂きました．さらに，お忙しい中 SFC まで出向いて御指導頂いた NTT DoCoMo ネットワーク研究所永田智大博士には，感謝の言葉が絶えません．

また，研究の日々を共に過ごした，榊原寛氏，米澤拓郎氏をはじめとする多くの友人に深く感謝します．最後に，研究会の活動を通じて数多くの経験や刺激を受ける機会を頂きましたことに心からの謝意を表し，謝辞と致します．

2004 年 12 月 29 日
小泉 健吾

参考文献

[1] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, Andy Hopper, “Virtual Network Computing”, IEEE Internet Computing, Vol. , No. 1, 1-2/1998.

[2] X.Org, http://www.x.org/

[3] F. Douglis and J. Ousterhout. “Transparent process migration: Design alternatives and the Sprite implementation”. Software: Practice and Experience, 21(8):757–785, August 1991.

[4] InterSense, Inc. “IS-600 Mark 2 Plus Precision Motion Tracker”, http://www.isense.com/products/prec/is600/mk2plus.htm.

[5] Kidd, C, et al. “The Aware Home: A living laboratory for ubiquitous computing research”. Proceedings of CoBuild’99 (1999)

[6] Garlan, D., Siewiorek, D., Smailagic, A., Steenkiste, P.: “Project aura: Toward distraction-free pervasive computing”. IEEE Pervasive computing (2002) 22-31 4

[7] Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven Shafer. “EasyLiving: Technologies for intelligent environments”. In Proceedings of Second International Symposium on Handheld and Ubiquitous Computing, HUC 2000, pages 12-29, Bristol, UK, September 2000.

[8] A. Harter, A. Hopper, P. Steggles, A. Ward and P. Webster. “The Anatomy of a Context-Aware Application”. In Proc. of the ACM/IEEE MobiCom, August 1999

[9] R. Headon. “Movement awareness for a sentient environment”. In First IEEE International Conference on Pervasive Computing and Communications (PerCom’03), pages 99–106. IEEE Computer Society Press, March 2003.

[10] A. Fuggetta, G. P. Picco, and G. Vigna. “Understanding Code Mobility”. IEEE Trans. on Software Engineering, May 1998.

[11] Ichiro Satoh, “Linking Physical Worlds to Logical Worlds with Mobile Agents”, In Proceedings of IEEE International Conference on Mobile Data Management (MDM’04), pp. 332-343, IEEE Computer Society, January 2004.

[12] Mobile IP, IETF
http://www.ietf.org/html.charters/mobileip-charter.html

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. “Aspect-oriented programming”. In ECOOP’97—Object-Oriented Programming, 11th European Conference, LNCS 1241, pages 220–242, 1997.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. “An overview of AspectJ. In Proceedings of the European Conference on Object-Oriented Programming”, Budapest, Hungary, 18–22 June 2001.

[15] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. “Bytecode Transformation for Portable Thread Migration in Java”. In Proceedings of the Joint Sym- posium on Agent Systems and Applications / Mobile Agents (ASA/MA), pages 16–28, September 2000.

[16] Markus Dahm. “Byte Code Engineering with the BCEL API”. Technical Report B-17-98, Freie Universit at Berlin, Institut f ur Informatik, April 2001.

[17] Zero Configuration Networking(Zeroconf), http://www.zeroconf.org/

[18] JmDNS, http://jmdns.sourceforge.net/

[19] Apple Computer, Inc., http://developer.apple.com/macosx/rendezvous/

[20] ティム・リンドホルム著, フランク・イェリン著，村上 雅章訳，“Java 仮想マシン仕様”，ピアソンエデュケーション，2001

[21] ジョン・メイヤー著, トロイ・ダウニング著, 鷲見 豊訳，“JAVA バーチャルマシン”，オライリー・ジャパン，1998.