修士論文 2009 年度 (平成 21 年度)

DRIAD: インタオペラビリティを向上するデバイス依存情報解決機構

慶應義塾大学大学院 政策・メディア研究科
伊藤 友隆

tomotaka@ht.sfc.keio.ac.jp

修士論文要旨 2009 年度 (平成 21 年度)

DRIAD: インタオペラビリティを向上する デバイス依存情報解決機構

論文要旨

本論文はユビキタスコンピューティン環境において，サービスのインタオペラビリティを実現するフレームワークにデバイス発見機構の動的追加機能と，デバイスに対応するサービス記述とドライバの動的解決機能を付加する DRIAD というシステムを提案する．

ネットワーク接続可能なデバイスが増え，様々な場合でネットワークを通じて便利にサービスが利用できるようになった昨今，直感的にサービスを連携できそうな組み合わせにおいて，連携を試みようとしてもネットワークの違いや，機器が利用するプロトコルなどの技術的制約によって，サービス連携が行えないという問題がある．

様々なサービス相互運用のための技術が研究されてきた中で，本研究では，将来性に優れた中間ノードによる共通セマンティクス翻訳モデルを採用するユビキタスインタオペレーションフレームワークに焦点を当て，この機能をより実用的なものになる上記のような機能をサポートする DRIAD を追加開発した．

DRIAD を組み込むことで，中間ノード上のフレームワークが動的にサービスの発見や，その連携のためのプログラムの取得を行えるようにすることで，増え続けるネットワーク対応デバイスや，ネットワークプロトコルに対応できるようになる．

また，DRIAD は動的にサービスを発見するためのプログラムを管理する分散データベースと，デバイスに対応するサービス記述とドライバを管理する分散データベースを提案する．分散データベースは負荷分散からの面と，管理の手間，柔軟さの面で集中型のデータベースに対して優れている．

本論文では提案する DRIAD の定性的評価と，パフォーマンスの面からの定量的評価を行い，考察した．結果，ユビキタスインタオペレーションフレームワークに対して非常に低い計算コストで新しいデバイスに適応できる機能を提供できることを示した．

キーワード

ユビキタス，インタオペラビリティ，サービス記述，異種ネットワーク，分散システム

慶應義塾大学 政策・メディア研究科

伊藤 友隆

Abstract of Master's Thesis Academic Year 2009

DRIAD: Device Dependent Information Resolve System for Improving Ubiquitous Interoperability

Summary

This paper proposes system named DRIAD, which enables interoperation framework to resolve device-dependent-information and retrieve service discovery plug-in.

We usually classify devices by their functionality not by their network nor protocols. But devices can not interoperate each other on heterogeneous network environment. Therefore interoperation framework has been researched for universal service interaction.

I focused on intermediated semantics translation model framework and developed DRIAD system for imporving its interoperability. DRIAD provides the framework which includes dynamic service discovery plug-in loader and dynamic device-dependent-information resolver. It is designed to adopt to changing computing environment which is going to be more heterogeneous.

DRIAD design includes external distributed network databases for service discover program, service description for devices, and device driver. Distributed database allows developers to publish their device drivers or service descriptions of devices easily.

This paper measured its performance evaluation and analyzed the results. Based on the results, we verified that DRIAD can add adoptation ability to interoperation framework with very low cost.

Keywords

Ubiquitous, Interoperability, Service Description, Heterogeneous Network, Distributed System

Keio University Graduate School of Media and Governance

Tomotaka Ito

目次

図目次

表目次

第 1 章

序論

本章では，はじめに本研究の背景，目的について述べ，ついで，本論文の構成について述べる．

1.1 本研究の背景

近年，ユビキタスコンピューティングが盛んに研究されている [14]．ユビキタスコンピューティングは，大小を問わないコンピュータが知らず知らず生活の中に埋め込まれており，人々が意識せずにコンピューティングの恩恵を受けることができるというアイディアである．結果として現在我々は身の回りに様々な計算能力を持つオブジェクトに囲まれて生活しており，今後もますますデバイスの数やネットワークの広さはその大きさを急激に増していくものと考えられる．

そのようなユビキタスコンピューティングの環境が実現しつつあるなか，有線・無線を問わない様々なネットワーク技術や，単一のネットワークにおける標準化された相互運用のための技術の登場により，図 1.1 のような「インタオペラビリティの島」が生まれている．この図が表しているのは限られた領域ではインタオペラビリティが確保されているが，その領域の外ではインタオペラビリティがなく，互いのサービスを島を超えて利用・連携することができないということである．

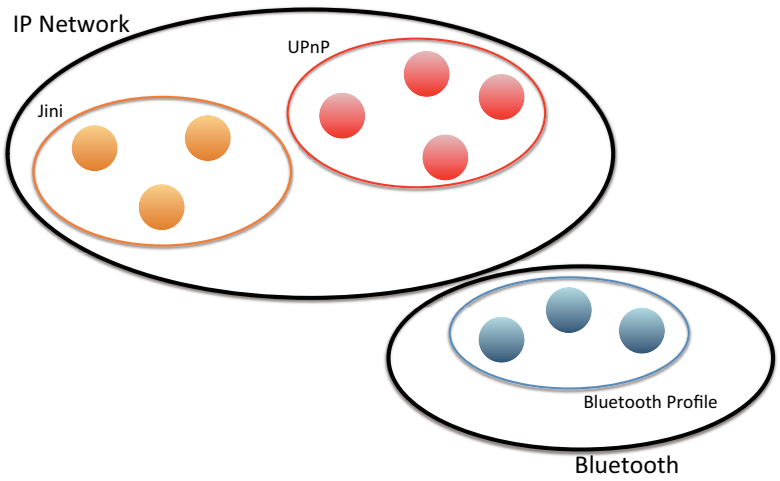

図 1.1　インタオペラビリティの島

それぞれのインタオペラビリティの島の例としては Bluetooth [4] のデバイスプロファイルであったり，UPnP [2] のデバイス型が挙げられる．これらはそれぞれ Bluetooth デバイスプロファイルや UPnP のそれぞれの型で定められたプロトコルに沿った内容で通信を行う限り，たとえ違うハードウェアであっても，「ワイヤレスヘッドセット」や，「メディア再生機器」といった概念レベルで共通のものは相互運用性が保証されている．

さまざまなネットワーク機器によるインタオペラビリティを確保するためには，これらの島をつなぎ合わせるか，大きな島にしてしまうかの二通りのアプローチが考えられる．しかしながら，「大きな島」アプローチは既存の技術を活かすことができず，また全ての問題領域をカバーするような通信方式や，プロトコルを設計することは難しい．特に無線技術では通信距離や，通信内容に応じて最適な通信規格が多くの研究者によって議論され，策定されてきた．

これらの理由から，あるインタオペラビリティの島のデバイスと，別のインタオペラビリティの島のデバイスが，島の垣根を超えてサービス連携を行えるような，島をつなぎ合わせるブリッジングの技術が研究されている．

インタオペラビリティの島をつなぎ合わせるにはいくつかの手法が考えられるが，本研究では中間ノードによるプロトコルの変換というアプローチを採用するインタオペレーションフレームワークモデルにフォーカスした．各モデルについては 2 章でより詳細に述べる．

1.2 本研究の目的

背景で述べたように，ユビキタス環境におけるインタオペレーションを実現するためのフレームワークの研究が進んできているが，既存の研究ではインタオペラビリティ実現のためのコア機能を実装するに留まっており，実際のユビキタス環境への配置の容易性などが考慮されていない．

具体的には中間ノードを利用するモデルでインタオペラビリティを確保しようとすると，中間ノードはブリッジする複数のプラットフォームのプロトコル全て解釈し，共通のセマンティクスに変換できる必要があるが，中間ノードが様々なプロトコルの実装を動的に取得・インストールできるような設計になっていない．

そこで本研究では，中間ノードを利用した異種ネットワーク上の様々なネットワーク到達性を持つ機器のサービス相互運用性確保するフレームワークに対して，機器のサービス記述およびその機器と通信するためのアプリケーションレイヤの実装を自動的に取得およびインストールする機構を提案する．

サービス記述とは，その機器がどのような機器であるかをプラットフォームに依存しない形式で記述したメタデータであり，ユビキタス環境におけるインタオペラビリティを実現するためのコア技術のひとつである．

本論文では慶應義塾大学徳田研究室で研究が進められている，USDL(Universal Service Description Language) を用いた機器のメタデータを利用する中間ノード形式のユビキタスインタオペレーションフレームワーク u-Glue を対象に，プロトタイプシステムの実装を行った．

1.3 本論文の構成

本論文の構成について述べる．本論文はこの節のある 1 章からはじまり，結論が書かれている 6 章までの全 6 章の構成である．

1 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と，その相互運用性の研究の現状について述べ，それを踏まえた上で本研究の目的についた述べた．2 章では本研究が研究の土台としている中間ノードを用いたユビキタスインタオペレーションフレームワークについて述べる．3 章では本研究の問題意識のより詳細な整理と，技術的な解となるアプローチについて述べる．4 章では本研究が提案するシステムの設計と，実装について述べる．5 章では本研究の関連研究について述べる．6 章では本論文で作成したプロトタイプシステムの評価を行った結果と，その考察につい

て述べる．7 章では本論文のまとめを行ったあと，本研究の課題と展望について述べる．

第 2 章

中間ノードを用いたインタオペレーションモデル

本章では，本研究の土台となる，中間ノードを用いる方式により，ユビキタス環境におけるインタオペラビリティを実現するフレームワークについて述べる．

2.1 サービス連携フレームワーク

2.1.1 ユビキタスコンピューティングとサービス連携

ユビキタスコンピューティングが現実になるにつれ，計算機科学における様々な課題が浮き彫りになってきている．そのひとつが，どのように機器を柔軟に連携させるか，というトピックである．古くは単一のネットワークで，そもそもアドレスを知らなければ通信できないという問題意識を解決するだけのサービスディスカバリ技術から，デバイスにどのようなサービスがあるのか，というメタ情報を表現しようとする試みまで様々な取り組みがなされてきた．

ネットワーク技術の多様化，ネットワーク接続可能なデバイスの急激な増加や，プロトコルなどの技術開発が進む中で，多種多様な通信方法が存在するようになってきている．これらの多様なデバイスやネットワークは，我々がそのサービスを享受し，生活を豊かにするために存在している．我々は直感的にデバイスを，そのデバイスのできること，提供できるサービスにおいて分類しており，ネットワークや利用している技術では分類しない．しかし現実はこのようなネットワークや，利用している通信技術の壁に阻まれ，直感的に連携可能なように思うサービスも連携することが難しい．

このように多様なネットワーク，多様なデバイスが混在するユビキタスコンピューティング環境では直感的で，ユーザフレンドリなサービス連携を行う要求がある．ユビキタスコンピューティング環境では技術やネットワークにおいて多様性を認めつつ，しかし機器や環境が提供するサービスの本質的な意味 (セマンティクス) を共通にして取り扱い，連携させる技術が必要とされている．いくつかの研究では，既存研究を組み合わせて技術の壁を越えて相互運用を行うことを実現している [3] [12] が，いくつかやり残した問題がある．本研究ではそのようなやり残された問題のうち，サービスをブリッジするノードに対する機器のサービス記述とサービス制御のためのドライバの自動提供という側面から，ユビキタス環境におけるユニバーサルインタオペラビリティの向上にコントリビューションしたいと考える．

2.1.2 中間ノードによるセマンティクス翻訳モデル

本研究では，中間ノードを用いて，異種ネットワーク間のサービスをブリッジするフレームワークを研究のベースとして焦点を当てている．中間ノードを用いる方式では，それぞれのサービス空間上のサービスに新たな実装などがなくても，中間ノードがサービスごとに適切なセマンティクスの変換を行い，中間ノード内で別のサービスへサービス連携を行う際に，適切なデータ形式に変換することができる．

理論上では，中間ノードを用いたインタオペレーションフレームワークはネットワークインタフェースが複数あり，セマンティクス翻訳のための実装や，どんなサービスが各ネットワーク上にあるかを認識する機能が十分にあれば，中間ノードが参加するネットワークの全てのサービスをインタオペラブルなサービスとして外部に提供できる．

このような性質から，サービス記述とドライバ，サービスディスカバリ機能を自動的に取得したり，解決したりする機構を組み込むことで，ユビキタスコンピューティング環境において，大幅なインタオペラビリティの向上を行うことが可能になると言える．

2.1.3 その他のインタオペレーションモデル

エッジノードによるセマンティクス翻訳

エンドポイントトランスレーションによるインタオペレーションモデルでも中間ノードを用いてインタオペラビリティを実現する．このモデルでは図 2.1 のように中間ノードではトランスポートレイヤをブリッジするのみで，プロトコルごとのセマンティクスの解釈をまったく行わない．

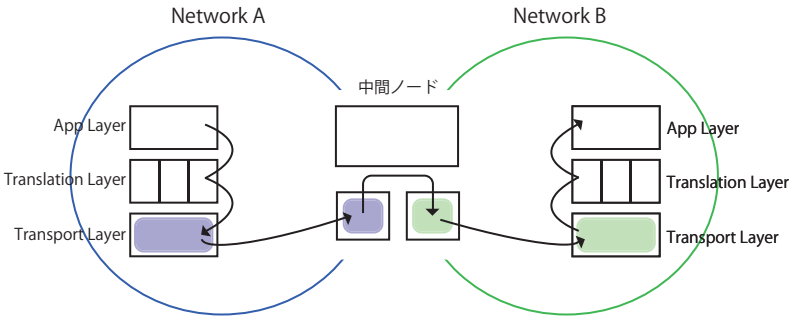

図 2.1　エンドポイントトランスレーションのフレームワークモデル

全てのエッジノードにプラットフォームごとのプロトコルスタックが備わっている前提であり，セマンティクスの解釈はエッジノードの責任というデザインになっている．このモデルでは中間ノードの責任が軽くなっている代わりに，エッジノードの責任が大きくなっており，またプロトコルスタックの拡張や追加インストールをする必要があるため，既存の機能が組み込まれているシンプルなデバイスはこのモデルではインタオペラビリティを確保することが不可能である．

また，全てのエッジノードが様々なプロトコルスタックをサポートするのはデバイスの容量や処理能力の観点から非常に非現実的であり，本研究ではこのモデルは検討外とした．

2.2 中間ノードを用いたインタオペレーションフレームワークのアーキテクチャ

2.2.1 インタオペレーションフレームワークの機能要件

本節ではユビキタス環境における，異種ネットワーク間サービス連携を実現するためのフレームワークの機能要件について整理を行う．

Uribarren らはその論文で Pervasive Application のための Middleware のデザインは 3 つの原則に基づくと主張している．

その 3 つの原則とは以下である (引用).

1. Be capable of getting available services from the heterogeneous environment
2. Extract the service information in terms of syntax description
3. Offer the available services using diverse service technologies

以降では Uribarren らが主張するこの 3 点の原則について，それぞれインタオペレーションを実現するフレームワークにとってなぜそれらが必要であるかを述べる.

サービスセマンティクスの記述

本節では，先に挙げた原則のうち 2 つ目について述べる.

インタオペレーションフレームワークでは，異種プロトコル上で交換される情報や，機器のサービスについて，意味的に同じものを同じように扱えるようにするための仕組みが必要となる.

異なるネットワーク技術の，異なるプロトコルやアプリケーションプラットフォームでは当然実際に通信で流れるデジタルデータは，同じ意味のもの，例えば画像や音楽といったデータも違うようにプロトコルのヘッダが付与され，時にはプラットフォーム固有のエンコードなどの中間処理が入る．このためユビキタスインタオペレーションフレームワークは，このようなデジタルデータのセマンティクスをなんらかの方法でプラットフォーム非依存な方法で表現しなければならない.

また，同じような機能，たとえば画像を表示したり，音声を再生したり，といった機能はプラットフォームごとに違うオペレーションの名前が与えられ，どのようなデータを取り扱うことが可能かというのはそのプラットフォームのペレーションセットの仕様に基づいており，プラットフォーム名とオペレーションセット名を記述しただけでは，サービスのセマンティクスを表現するに至らない．たとえば，「UPnP+MediaRenderer」と記述したところで，既存の技術とはメディア再生の機能として無理矢理協調動作させることは不可能ではないが，将来的に，意味的に同じデータを出力する別の新しいプラットフォーム上のデバイスと連携する際，「メディアを再生する機器」というセマンティクスが表現できていないため，インタオペラビリティがないと言える.

これらのプラットフォームごとに違う，共通のセマンティクスを持つサービスやサービスが取り扱うデータを共通に表現する方法が必要となる.

本論で実装対象としているユビキタスインタオペレーションフレームワークでは慶應義塾大学徳田研究室で研究が進められている USDL(Universal Service Description Language) をこの目的で利用している．USDL では機器が取り扱うデータを MimeType で表現しており，画像を取り扱うことのできるサービスは input や output が「`image/*`」などとなる．これによりプラットフォームや，オペレーションセットを問わず，既存技術でも新しく登場する技術でも，「画像」というデータを取り扱うというセマンティクスを表現できる.

USDL はありとあらゆる機器がコンピューティングに参加するユビキタス環境を想定して記述文法が設計されており，特定のプラットフォームに依存した表現や，特定のプラットフォームの機器やサービスしか記述できないということがないため，インタオペラビリティの研究には最適である.

USDL 以前にもデバイスやサービスが「どのようなものか」「どのようなサービスを提供しているか」をメタ情報として記述しようという試みは盛んに行われており，WSDL(Web Service Description Language) や，UPnP のデバイス記述などはまさにそういったものである．

クロスプラットフォームなサービスの提供

本研究が前提としているインタオペレーションフレームワークでは，中間ノードは異種ネットワーク間のサービス連携を行う際に，ネットワークをまたいで同じ意味のメッセージを送るために，意味を保ちつつそのメッセージをカプセル化するプロトコルの変換を行う必要がある．ここではそのプロトコル変換のメカニズムについて触れる．

もっともシンプルなパターンである，2 つの異種ネットワーク，ネットワーク A，B に所属するデバイス A，B のサービス連携をブリッジするパターンを例に，実際に処理の流れを追って説明する．中間ノードはネットワーク A，ネットワーク B にも接続されており，デバイス A，B 両方が利用するプロトコルを解釈することができる．ここではデバイス A が，デバイス B に画像データを伴ったサービス連携を行うシナリオを想定している．二つのデバイスの提供しているサービスが画像データを処理するサービスであるということは前述のサービス記述の仕組みを用いて中間ノードは解釈することが可能である．本シナリオのデバイスと処理の流れを図 hoge に示す．

まず，デバイス A は自分の参加しているネットワークを通じて，自分が使うプロトコル (以下プロトコル A) を使って中間ノードと通信する．中間ノードはプロトコル A を解析することのできる実装を持っており，プロトコル A における「画像」の表現を通信から取得する．次に，中間ノードはデバイス B の使うプロトコル (プロトコル B) における画像の表現へ，取り出した画像データを変換し，ネットワーク B を通じてプロトコル B を使ってデバイス B と通信を行い，デバイス B のサービスを呼び出す．

2.2.2 フレームワークの全景

エッジノードを含めた中間ノードによるセマンティクス翻訳を行うインタオペレーションフレームワークの全景は図 2.2 のようになる．

中間ノードは前提として，二種類以上の複数の異種ネットワークに参加しており，ある一つのネットワークからの入力を，他のネットワークのデバイスのサービスへとセマンティクスを保ったままデータの転送を行う．図の例では，中間ノードは UPnP ネットワークと，Jini [13] ネットワークに参加しており，それぞれのプロトコルを解する．

エッジノードに関しては特別な要求はなく，そのままでよい．自分が通信可能なネットワークで，自分が実装を持っているプロトコルで通信を行うだけでよい．

以下では図に登場しているエッジノード A がエッジノード B に対して画像を転送するというサービス連携を具体的な例として取り上げて説明する．

中間ノードはエッジノード A のプロトコルを解釈し，エッジノード A と対応するサービス記述からサービスのセマンティクスを解釈し，や「画像」データを得る．

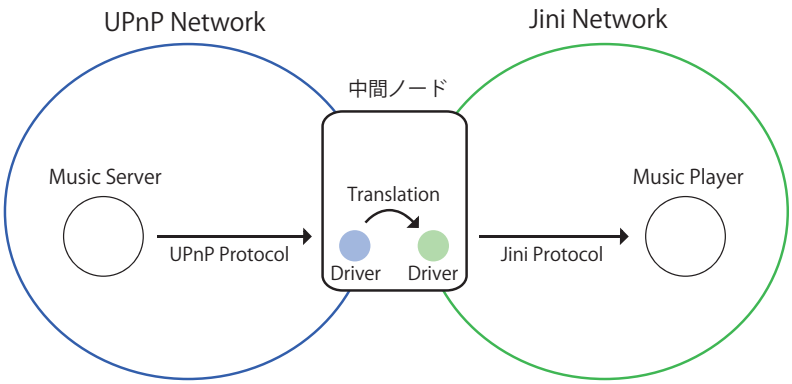

図 2.2　UPnP サービスと Jini サービスの連携の様子

これらを別のネットワークのエッジノード B に送る際に，またエッジノード B と対応するサービス記述をチェックし，これに対応するよう「画像」データをうまく組み込まれているプロトコル実装を使って変換した上でエッジノード B が参加しているネットワークを使って通信を行い，連携を実現する．

2.3　本章のまとめ

本章では，本研究の土台となる中間ノードを用いる方式によりユビキタス環境におけるインタオペラビリティを実現するフレームワークについて述べた．アーキテクチャの節では，中間ノード方式によるユビキタスインタオペレーションフレームワークの核となるサービスセマンティクスの記述と，プロトコルの変換をについて述べ，最後にフレームワーク全体のアーキテクチャについて俯瞰した．

その他のインタオペレーションフレームワークモデルの節では，本研究が土台としている中間モデルによるセマンティクスの解釈おこなうインタオペレーションフレームワーク以外のモデルについて考察し，そのメリットとデメリットについて整理した．

第 3 章

中間ノードのためのデバイス依存情報解決機構

本章では，既存研究により提案されている中間ノードを用いたインタオペレーションフレームワークの運用上の問題点を挙げ，それらの問題点を解決するためのシステムの機能要件を述べたあと，本研究が提案するシステムのアプローチについて述べる．

3.1 問題意識

本節では本研究の問題意識を整理する．まず，中間ノードを用いたユビキタスインタオペレーションフレームワークの運用面での問題を示し，中間ノードが必要とするデータであるデバイス依存情報についてまとめる．次に，現状のフレームワークはデバイス依存情報の柔軟に管理する仕組みを持たないことで起こる問題について具体的に触れる．

3.1.1 既存フレームワークの運用面での問題

既存研究で提案されている中間ノードを用いたフレームワークでは，環境に対するデバイスの出入りを想定していない．具体的には各デバイスの持つサービスを利用するためのサービス記述やドライバの，追加や更新という点を考慮していない．

このため，新しいデバイスのサポートをフレームワークで行おうとした場合，中間ノードで動いているプロセスをシャットダウンし，ドライバプログラムを追加し，ロードできるようにした状態で再起動しなければならない．

これではオペレーションの負荷が高すぎて実際にいろいろな人が前述のように多種多様なデバイスを持ち歩いて空間を出入りするような状況に対応できない．とくに公共性が高く人の出入りが活発に起こる廊下や会議室といった空間で，インタオペラブルなサービスを既存フレームワークを用いて提供しようとした場合，出入りする人が持っている全てのデバイスをあらかじめサポートするようフレームワークにサービス記述やドライバをインストールする必要があるが，現実的には全ての人の持つ全てのデバイスを予測することは極めて難しいため，実質不可能と言える．また，新しいデバイスが登場した際には新しく追加でデバイス依存情報をインストールしなければいけない．

既存研究で提案されているフレームワークでは，このようにダイナミックなサービス記述との解決やインストールという仕組みを持たないため，広く一般にインタオペラブルなサービスを提供するために利用するにはオペレーションの負荷が高すぎるという現実がある．

3.1.2 デバイス依存情報

本節では，本研究が中間ノードのために提供するデバイス依存情報という概念について述べる．

1 章および 2 章では，中間ノードによるプロトコルトランスレーションを用いたインタオペレーションモデルでは，プラットフォームに依存しない形式で記述されたデバイスごとのサービス記述と，プロトコルトランスレーションのためのドライバが必要であることを述べた．

本研究では，デバイスに依存するこの 2 点の項目をまとめたものを「デバイス依存情報」という表現で呼称することとする．本章では，現在のインタオペレーションフレームワークのインタオペラビリティの実現を大きく妨げる要因として，デバイス依存情報の配布と解決のダイナミズムの不足を問題点として挙げ，これらを解決するシステムの機能要件を導出する．

3.1.3 開発者視点でのデバイス依存情報公開手段の欠如

いろいろ技術を使って通信を行う製品や，サービスが多様に存在し，流動的に入れ替わるユビキタス環境では，そのメタ情報であるサービス記述や，制御するためのドライバを動的に手に入れる必要があることをサービスを享受するエンドユーザ側の視点でこれまでに述べた．

しかし一方で，様々なデバイスが日々新しく生まれ，世に出てきている点を考慮すると，デバイスを開発する側の視点では開発したデバイスのサービス記述とドライバ，つまりデバイス依存情報を素早く公開したいというニーズがある．

デバイスを制御するドライバプログラムも，近年デバイスの高機能化やオペレーティングシステムのアップグレードによって使って行くうちにダウンロードしてアップグレードするといった利用形態が一般的になってきている．開発者としては最新の機能を含むドライバを迅速に公開し，エンドユーザの下へ届けたいと考えるだろう．

本研究が研究対象としている中間ノードを用いるインタオペレーションフレームワークが広く実際の社会で利用されるようになれば，デバイス依存情報の有無でインタオペラビリティが確保できるか，できないかが決まるので，ユーザ側としてもベンダに対してはデバイス依存情報の迅速な公開と共有が望まれるだろうと予測できる．

ベンダごとに，迅速なデバイス依存情報の公開および共有を実現するためには，認証などのプロセスを極力少なくしなくてはならない．どこか一カ所にサービス記述やドライバを集中管理するディレクトリサービスを構築したところで，そのディレクトリサービスの運営次第では，デバイス依存情報の公開が遅れてしまったり，削除したいときに削除できない，更新したいときに更新できないといった問題が予想される．

そのため，権限を分散的に管理し，ベンダごとの裁量でデバイス依存情報を全世界に配布・更新が行えるようなシステムが必要となる．

3.1.4 エンドユーザ視点でのデバイス依存情報自動解決機構の欠如

先に述べたように，サービスをミドルウェアを通じて，インタオペラブルに利用しようと思った際に，エンドユーザはデバイス依存情報をインストールして中間ノードを再起動しなければならないという問題がある．

これは単純に再起動すればよいというだけでなく，環境の管理者以外はデバイス依存情報のインストールといったオペレーションをできない，またはそれが難しいという問題も抱えている．

最先端である異種ネットワークでバイスのインタオペラビリティ技術は，人々にとって大きいメリットをもたらすはずであるが，現状ではユーザ自身がアーキテクチャに精通しており，サービス記述やドライバを作成したり，取得したりして自分自身でミドルウェアを運用する手間をかけなければその恩恵を受けることができない．

ユビキタスコンピューティングの考え方ではコンピュータ技術に精通していないユーザでもあま

ねく恩恵を受けられるような社会をビジョンとして提示しているが，このような現状はそれに沿っているとは言い難い．

ユビキタスサービスの基盤技術となりえるインタオペラビリティ技術が，このような基本的な部分でエンドユーザのメリットを阻害するのは非常に問題がある．

エンドユーザに異種ネットワークを超えてサービスを協調させるミドルウェアの存在を意識させることなく，スムーズに未知のデバイスのデバイス依存情報のインストールを行い，ヘテロジニアスなサービス空間でのインタオペラブルなサービスを提供をするのが本来あるべきミドルウェアの姿である．

このように，ミドルウェア側で未知のデバイスを認識し，デバイス依存情報をネットワークを通じて取得するという機能がないという問題が存在するためこれを解決する必要がある．

3.2 機能要件

本節では，問題意識の節で挙げた既存のユビキタスインタオペレーションフレームワークが抱える運用面での問題を解決し，インタオペラビリティを向上するためのシステムを提案するにあたり，どのような機能要件があるかを整理し，述べる．

3.2.1 R1. デバイス依存情報の頒布および取得が行えること

問題意識の節では，開発者側の視点と，エンドユーザ側両方の視点から，デバイス依存情報の頒布および取得が必要であることを明らかにした．そのため，開発者のために広く一般的にデバイス依存情報の頒布が行えること，そしてエンドユーザのために広く一般的にデバイス依存情報の取得が行えることが問題意識の節で述べた問題を解決するシステムのための機能要件のひとつとして言える．

この機能要件を満たすことで，開発者はデバイスを作成するたびに統一的な方法でデバイス依存情報を全世界に公開することができ，エンドユーザはデバイス依存情報を統一的なリポジトリを利用して取得することができ，開発者ごとに，あるデバイスを制御するドライバの再開発をしたり，エンドユーザが必要なデバイスのサービス記述やドライバが見つけられないといったことを防ぐことができる．

3.2.2 R2. 中間ノードがデバイス依存情報を自動で獲得できること

問題意識の節では，エンドユーザ側の視点から，中間ノードにおいて，自動的なデバイス依存情報解決の仕組みが必要であることを述べた．

ユビキタスインタオペラビリティを提供するフレームワークにおける，中間ノードに対して，中間ノードが所属するそれぞれの異種ネットワーク上の未知なデバイスを自動的に発見し，それらのデバイスに対するデバイス依存情報を自動的に取得し，インストールできることが，問題意識の節で述べた問題を解決するシステムのための機能要件のひとつとして言える．

この機能要件を満たすことで，最新のドライバに自動的に更新，インストールを行えるようになったり，管理者以外のエンドユーザがインタオペラビリティを提供するミドルウェアのデバイスサポートの不足を解消できない問題や，中間ノード管理者のオペレーション負荷を軽減することが可能になる．自動的なデバイス依存情報の解決により，多くの機器を手を煩わせることなくインタオペラブルに相互作用させることができるようになる．

3.3 アプローチ

本節では，先に述べた問題式を解決するシステムの機能要件に対して，本研究が具体的にどのようなアプローチを採っているかについて述べる．

3.3.1 A1. DNS モデルの分散管理されたデータベース

機能要件 R1 に対し，本研究では DNS モデルの分散データベースを採用することでこの機能要件を満たす．

本研究ではネットワーク上のデバイスを発見し，その ID を取得するマッパのプログラムと，デバイス依存情報そのものの 2 つの情報を分散データベースで管理するよう提案する．

中間ノードで動作するインタオペレーションミドルウェアはネットワーク上の分散されたマッパデータベースから常に最新のマッパプログラムを取得しておくことにより，自身に新しいネットワークインタフェースが追加されたり，ネットワーク上に最新の未知のプロトコルがパケットとして流通するようになった場合でもネットワーク上に流れるプロトコルをマッパプログラムにより解釈することができ，ネットワーク上のデバイスの存在を認識することができる．

DNS は，より上位のドメインがより下位のドメインの情報を責任を持って管理しており，自分の管理しているドメインの情報は全て編集が可能となっている．

デバイス依存情報は，ベンダごとに自由に更新できるリポジトリを分散データベース全体の管理者からアサインすることで，自分たちが作ったデバイスのサービス記述やドライバを簡単に頒布することができるようになり，エンドユーザはリゾルバによってそれらを取得する事で，未知のデバイスを自動的に自分たちの環境でインタオペラブルに利用することが可能になる．

ルートノードや，各ノードのリーフノードとなるノードの設計については，4 章の「DRIAD の設計」の節で詳細に述べる．

3.3.2 A2. インターネット上のデバイス依存情報ルックアップサービス

機能要件 R2 に対し，本研究ではインターネット上に前節で概要を述べた DNS モデルの分散データベースを配置することでこの機能要件を満たす．

本研究では，ネットワーク上に流れているプロトコルを解釈し，その通信からネットワーク上のデバイスの存在を認識する「マッパ」とデバイスのネットワーク上の ID 情報からデバイス依存情報を動的に解決する「リゾルバ」をコア技術としている．

ネットワークインタフェースや，ネットワーク上を流れるプロトコルが増えた場合，中間ノードはマッパを動的にネットワークから取得し，インストールできる必要がある．「Mapper-DB」はマッパをインターネット上で分散管理するデータベースである．

同様に，中間ノードはマッパにより存在が確認されたネットワーク上の未知のデバイスについて，サービス記述とドライバをネットワークから取得し，インストールできる必要がある．実際にこの働きを行うのがリゾルバであり，リゾルバはあるネットワーク上のデバイスについて，そのデバイスが実際に通信を行っているプロトコルと，そのプロトコルにおける ID をもってユニーク ID とし，インターネット上に配置された分散デバイス依存情報データベース「DDI-Repository」にデバイス依存情報取得クエリを行う．

マッパおよびリゾルバの詳細な設計については 4 章の「DRIAD の設計」の節で述べる．

3.4 本章のまとめ

本章では，本研究の問題意識と，それを解決するシステムの機能要件，そして本研究が提案するシステムのアプローチについて述べた．

問題意識の節では，中間ノードがインタオペラビリティを実現するにあたって，大きく依存しているサービス記述，およびドライバについて触れ，それらをまとめたデバイス依存情報という概念について述べた．

機能要件の節では，それに基づき，中間ノードがデバイス依存情報の扱いについて現在どのような問題点を抱えているか整理し，議論している問題を解決するためのシステムの機能要件をまとめた．

最後に，アプローチの節で，本研究が問題意識に対する解として，どのようなアプローチを採用したのかを述べた．

第 4 章

DRIAD の設計と実装

本章では，3 章で整理された問題を解決するために，本研究が提案する分散デバイス依存情報解決システム DRIAD の設計と実装について述べる．

4.1 前提環境

本研究はあくまでも中間ノードによるセマンティクストランスレーションモデルを採用するユビキタスインタオペラビリティのためのフレームワークであり，本研究のコントリビューションは単独で動くシステムではなく，これらのデザインを共通に持つフレームワークに組み込むことではじめて動作させることができる．本節では，本研究がこのようなフレームワークに対して，どのような前提条件を求めているかをまず述べる．

4.1.1 静的なサービス記述

本研究では，中間ノード上のミドルウェアは，サービスをブリッジするデバイスのサービス記述を静的に管理しているものを想定している．

このようなデザインでは，あらかじめミドルウェアが稼働する中間ノードに所定の形式で記述されたサービス記述をローカルファイルシステムや，メモリ上に配置しておき，静的にそのデータを読み込み，サービスをブリッジしてインタオペラブルなサービスとして提供するようになっている．

サービス記述の静的な管理では，新しいデバイスについて，そのサービスをインタオペラブルに提供しようと思った場合，サービス記述ファイルを新たに中間ノードにインストールする必要があり，2 章で述べたような運用上のオペレーションコストや権限の問題がある．

4.1.2 静的なドライバの配置

本研究では，中間ノード上のミドルウェアは，サービスをブリッジする際に，具体的にサービスを提供しているデバイスの，サービス記述および，通信を行うドライバを静的に管理しており，起動時にロードするようなデザインとなっているものを想定している．

このようなデザインでは，あらかじめミドルウェアが稼働する中間ノードに，ドライバをなんらかのロード可能な形としてローカルファイルシステムやメモリ上に配置しておき，ミドルウェアの起動時にその外部ライブラリをロードし，該当するデバイスとの通信が行えるようになる．

ドライバの静的な管理には，新しいデバイスをインタオペラブルにしようと思った場合，ドライバの外部ライブラリファイルを新たに中間ノードにインストールする必要があり，2 章で述べたような運用上のオペレーションコストや権限の問題がある．

4.1.3 プラガブルな API デザイン

本研究では，中間ノード上のミドルウェアがサービス記述の動的な追加や，ドライバの動的なインストールに対応するためのプラガブルな API デザインを持っていることを前提としている．

DRIAD は広く適用できるようにデザインされているが，適用するミドルウェア側にもある程度

の API の柔軟性を要求する．

DRIAD では，具体的には以下のような，デバイス発見機構の追加や，デバイス依存情報の登録のための API があるミドルウェアに組み込むことができる．以下の API 一覧ではサンプルとして，Java 言語で API の形式を記述している．

1. `public ServiceDescription getServiceDescriptionForDevice(UniqueID device)`
2. `public Driver getDeviceDriverForDevice(UniqueID device)`
3. `public void installServiceDescription(UniqueID device, ServiceDescription)`
4. `public void installDriverForDevice(UniqueID device, Driver driver)`
5. `public void installAndStartMapper(Mapper newMapper)`

もちろん完全にこの通りの API が必要なわけではなく，ミドルウェアの実装に応じてラッパを使って DRIAD の機能を利用したり，柔軟に対応することができる．しかし，DRIAD の本質的な機能を利用するためには以下の機能がミドルウェアのプラグイン側に対して公開されている必要がある．

- デバイスをネットワーク上から発見するためのモジュールを動的に追加し，動かすことができること
- デバイスに対応するドライバを保存し，自動的に有効にすること
- デバイスに対応するサービス記述を保存し，自動的に有効にすること

これら 3 点の機能を満たせば，DRIAD を組み込むことができると言える．

4.2 DRIAD の設計

本節では，分散デバイス情報解決システム DRIAD の設計について述べる．

全体の設計の節では，ユビキタスインタオペレーションフレームワーク全体に対して，DRIAD の果たす責任や，マッパやリゾルバなどの DRIAD が定義している概念および，DRIAD によって行われる処理の全体を通したシーケンスなど，DRIAD の全体的な設計について述べる．

Mapper-DB の設計の節では，インターネット上に構築されるマッパの分散管理を行うデータベースの設計について述べる．

DDI-Repository の設計の節では，インターネット上に構築される，デバイス依存情報の分散管理を行うデータベースの設計について述べる．

4.2.1 DRIAD の設計

DRIAD の責任領域

DRIAD はデバイス依存情報を動的に解決するための高度にモジュール化されたソフトウェアだが，最終的には中間ノード上で動作するミドルウェアの内部に組み込んで使われる必要がある．

そのため，本節ではデバイス依存情報の取り扱いについて，どこまでを DRIAD が処理を担当するかについて述べる．

DRIAD は，異種ネットワークをまたぐサービスのインタオペラビリティを提供するフレームワークに対して，以下の機能を提供する．

1. デバイス発見機構を中間ノードに対して，動的に取得して追加する
2. デバイスのサービス記述とドライバを動的に取得して追加する

項目 1 について，既存のインタオペレーションフレームワークでは中間ノードのサービスディスカバリ能力は動的に追加できないため，これを補完するものである．具体的にはインターネット上に構築されたデバイス発見のためのプログラム「マッパ」を定期的に調べ，自分が参加しているトランスポート層ネットワーク用の新規マッパがあればそれを取得し，それをインストール，実行する事で，サービスディスカバリ能力を動的に獲得できる機能を提供する．マッパについては「マッパ」の節で詳細に述べる．

項目 2 について，既存のインタオペレーションフレームワークでは中間ノードがプロトコルを実装しており，どのようなデバイスであるかというセマンティクスレベルの抽象的なサービス記述を持っているデバイスのサービスしか，ブリッジできないという問題があるため，これを補完するものである．具体的にはインターネット上に構築されたサービス記述とドライバからなるデバイス依存情報を解決するためのプログラム「リゾルバ」を中間ノードに提供する．リゾルバはマッパと連携し，新規に発見された中間ノードがデバイス依存情報を持たないデバイスについて，インターネット上のデータベースからデバイス依存情報を解決し，取得，インストールを行う．サービス記述とドライバを揃えることで中間ノードは新しく発見されたデバイスのサービスをインタオペラブルなサービスとして異種ネットワーク上で提供できるようになる．

マッパ

本節では DRIAD のうち，中間ノード側で動作するプログラムであるマッパについて述べる．

マッパは中間ノードに対して，デバイスを「見つける」機能を提供するプログラムである．

多くのサービス連携技術は一般的にサービスディスカバリと呼ばれるデバイスやサービスの存在をネットワーク上から見つけ出す機能を持っている．Bonjour や SLP といったサービスディスカバリプロトコルもそうであるし，UPnP の一部分として含まれる SSDP(Simple Service Discovery Protocol) もそれに該当する．

インタオペレーションフレームワークにおいて，中間ノードは単一の技術によらずに様々な技術を使うデバイスのサービスブリッジングを行えなければならない．

そのためにはまず，ネットワーク上でデバイスやサービスを発見し，特定できる必要がある．UPnP にだけ対応するのであれば SSDP だけサービスディスカバリの機能として組み込めばよいだろうが，そうではないため，デバイスやサービスを発見する部分のみを切り出してマッパとしている．

図 4.1 は静的にデバイス見つけるプログラムを管理している中間ノードが，新しいプロトコルに

対応できずに，図の下部に表記されているデバイスを検知する事ができない様子を示している．

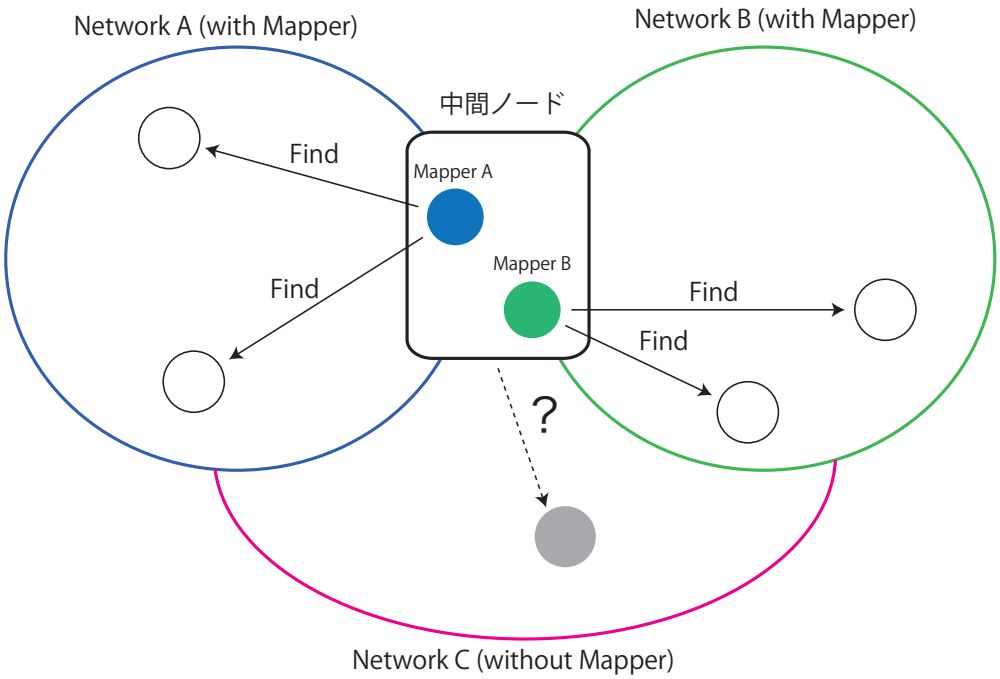

図 4.1　動的にデバイス発見機構を追加できないため，新しいプロトコルに対応できない

マッパはプラガブルに実行中の中間ノードに Mapper-DB からダウンロードされ，インストールされ，実行される．実行されたマッパはネットワークを監視したり，自分が担当するネットワークプロトコルを使って能動的にサービスディスカバリのリクエストを行ったりする．例えば UPnP のためのマッパがあれば SSDP を定期的に使用する．

このようにサービスディスカバリの機能をプラガブルなモジュールにしておくことで，未知のサービス連携技術が登場しても，Mapper-DB からその技術を実装したマッパを中間ノードに動的に追加することができるため，新しいサービス連携技術上で動作するデバイスやサービスを発見することができる．

デバイスを発見することで，一般的にデバイスを一意に特定することができる．サービス連携やサービスディスカバリのためのプロトコルはデバイスのユニークな ID を知ることができるので，プロトコルと組み合わせることで，まさにそのデバイスのユニークな ID を生成することができる．

本研究ではこのプロトコルと，そのプロトコルにおける ID を組み合わせたものをユニーク ID と呼称する．ユニーク ID を用いることでリゾルバは DDI-Repository からデバイス依存情報を解決することができる．

リゾルバ

本節では DRIAD のうち，中間ノード側で動作するプログラムであるリゾルバについて述べる．

リゾルバは中間ノードに対して，未知のデバイスを制御し，インタオペラブルに外部提供するた

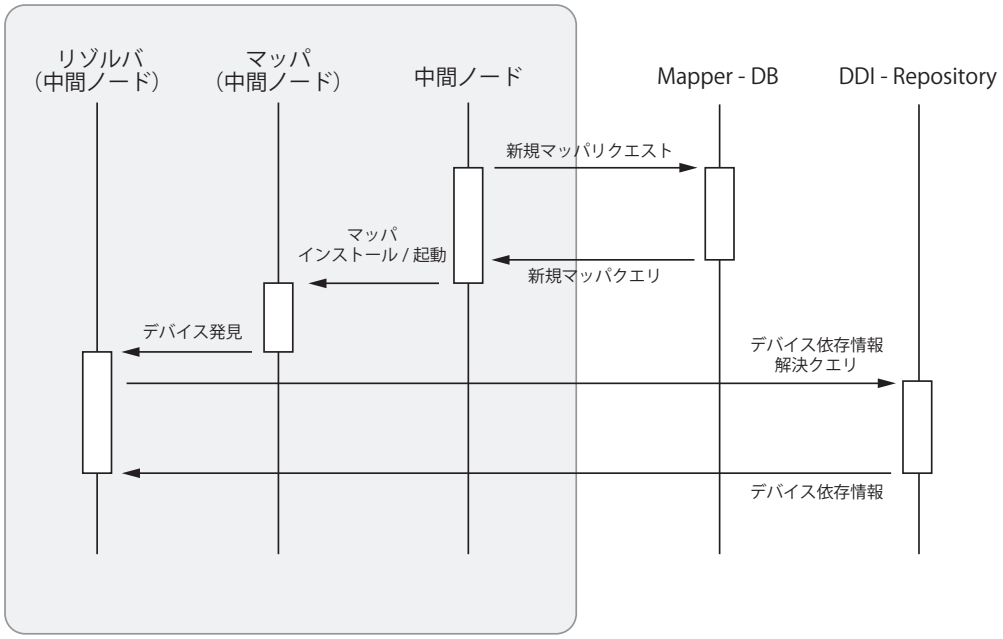

図 4.2　マッパ取得からデバイス依存情報解決までのシーケンス

めのサービス記述とドライバを獲得する機能を提供するプログラムである．

マッパにより，ネットワーク上の存在が確認されたデバイスが，中間ノードにとって未知なデバイスであった場合，そのデバイスがどのようなサービスを提供しているかを記述しているサービス記述と，デバイスのサービスを実際に制御するためドライバをインストールすればフレームワークの一部として利用できるようになる．

リゾルバはマッパがネットワーク上に存在を確認したデバイスから生成したユニーク ID を基に，DDI-Repository に対してクエリを発行し，デバイス依存情報をダウンロードし，中間ノードにインストールする．

DDI-Repository には中間ノードが動作するプラットフォームに応じてドライバを複数格納することができるので，リゾルバは中間ノードの動作しているプラットフォームに応じたドライバを取得することができる．

デバイス依存情報解決処理のシーケンス

本節では，中間ノード側の DRIAD システムの動きを追って中間ノードが未知のデバイスを発見し，そのデバイスのデバイス依存情報を取得してインストールするまでの流れをシーケンス図として図 4.2 に示す．

4.2.2 Mapper-DB の設計

分散データベース

Mapper-DB は一カ所にデータを集約しない，分散形式のデータベースとなっている．データが一カ所に集約しないことで，クエリが集中しない，更新や管理が容易という利点がある．Mppaer-DB は分散配置されたノードへのアクセス手法を DNS をモデルに設計されており，上位のノードに問い合わせ，その次の層のアドレスを得るという形式になっている．このアクセス方式ではルートノードにアクセスが集中しやすいのが欠点だが，クライアントや，分散ノードごとのキャッシュを利用することで大幅に負荷を分散し，軽減させることができる．その代わり，ノードのデータベースの内容に更新があった際の反映が遅れることがキャッシュ機構のデメリットとなっている．

データベースの構造

Mapper-DB は 3 層からなるツリー構造になっている．

Mapper-DB の全体像を図 4.3 に示す．

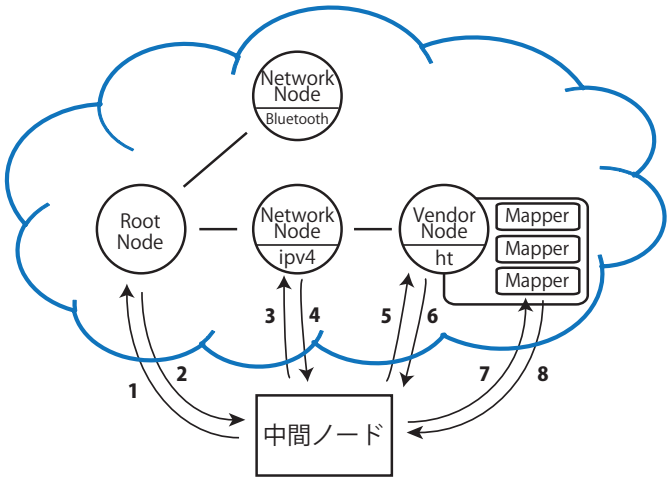

図 4.3　Mapper-DB と Mapper 取得の様子

1 層目はルートノードであり，2 層目のネットワークノード全てへの参照を持っている．クライアントはまず最初にルートノードに，自分が持っているネットワークの種類に応じてクエリを行う．たとえば IP のインタフェースしか持っていない中間ノードはルートノードに IP のネットワークを担当している 2 層目のノードのアドレスを教えてくれるようクエリを行う．複数のネットワークインタフェースを持っている場合は複数回，クエリを行う．

2 層目はネットワークノードである．ネットワークノードは，そのネットワークで流通する可能性のある全てのマッパを，ベンダーノードを通じて包括的に管理している．クライアントはネット

ワークノードから，そのネットワークノードに紐づいている全てのベンダのリストを得る．クライアントはベンダリストに従って順番にベンダノードにアクセスし，自分が持っていないマッパプログラムがないかどうか調べる．

3 層目はベンダノードである．各ベンダは自分のノードを持つことができ，自由にマッパを公開，更新することができる．クライアントはネットワークノードから教わったベンダリストに従い，ベンダノードにアクセスしてくる．最初のアクセスではクライアントは各ベンダノードが持つマッパのリストを得る．クライアントは全てのベンダノードからリストをもらった後，自分の中間ノードにインストールされていないマッパがないか確認する．まだインストールされていないマッパがあれば，提供されているベンダノードにリクエストを行い，マッパプログラムをダウンロードし，インストールする．

中間ノードはこれらの複数層のノードを渡り歩いてマッパプログラムを集めて，インストールする．

マッパ提供のシーケンス

本節では，ベンダや，技術標準化団体がデバイスをネットワーク上から発見するためのマッパを，Mapper-DB を通じて提供するための具体的な手順について述べる．

Mapper-DB では，ベンダや技術標準化団体が対象としているトランスポートレイヤのネットワークに対応した Mapper-DB 上のネットワークノード下に，ベンダノードを設けて管理しているという想定である．ベンダノード以下に，公開したいマッパプログラムをファイルとして配置し，ベンダノードの設定ファイルを書き換えて，再起動することで Mppaer-DB を通じてマッパを公開することができる．

ベンダノードプログラムの設定ファイルはシンプルな XML ファイルとなっており，以下のリストのような形式となっている．

ソースコード 4.1　Mapper-DB ベンダノードプログラムの設定ファイル例

```
1  <?xml version="1.0" ?>
2  <vendorNodeConfiguration type="mapper">
3    <identity>
4      <network>ipv4</network>
5      <vendor>ht</vendor>
6    </identity>
7
8    <mappers>
9      <mapper>
10       <id>ipv4-ht-UPnPMusicServer-java</id>
11       <binary type="file" platform="java">resource/fs-mapper/
             FileSystemUPnPMusicServerMapper.class</binary>
```

```
12      </mapper>
13
14      <mapper>
15        <id>ipv4-ht-UPnPMusicPlayer-java</id>
16        <binary type="file" paltform="java">resource/fs-mapper/
              FileSystemUPnPMusicPlayerMapper.class</binary>
17      </mapper>
18
19      <mapper>
20        <id>ipv4-ht-JiniMusicPlayer-java</id>
21        <binary type="file" platform="java">resource/fs-mapper/
              FileSystemFakeMusicPlayerMapper.class</binary>
22      </mapper>
23    </mappers>
24  </vendorNodeConfiguration>
```

identity 要素では，自身が Mapper-DB の，どのネットワークノード下にあるのか，どのベンダ名で登録するのかを記述する．ここで挙げた例では，IPv4 ネットワークの，ベンダ名 ht というノードとして，Mapper-DB データベースネットワークに参加している．

mappers 要素では，自身が管理しているマッパプログラムを列挙する．各マッパプログラムは mappers 要素中の mapper 要素で表現される．mapper 要素はマッパ ID を記述する id 要素と，実際に動作するマッパプログラムのファイルのロケーションを記述する binary 要素から構成される．binary 要素では，中間ノードが自分のプラットフォームにあったマッパを取得できるように，提供側も複数のプラットフォームの実装を提供できるように複数の列挙を許可している．例えばここで挙げた例では platform 属性に java が指定されているが，i386-darwin などとして，OS やアーキテクチャを指定したプラットフォームを指定することも可能となっている．

マッパ取得のシーケンス

本節では，DRIAD のマッパローダモジュールが具体的にどのように Mapper-DB から中間ノードがまだ持たない，新しいマッパを発見し，取得するのかの流れを示す．図 4.4 は実際の処理のシーケンス図である．

ユニーク ID 解決のシーケンス

本節では，マッパがネットワーク上からデバイスを発見し，それを DRIAD で利用するユニーク ID に変換する手順について述べる．

ユニーク ID は，ネットワーク上のデバイスに一意に付けられる ID であり，ネットワークプロトコルを含めて，一意に識別できるよう，以下の情報を含んだ ID である．

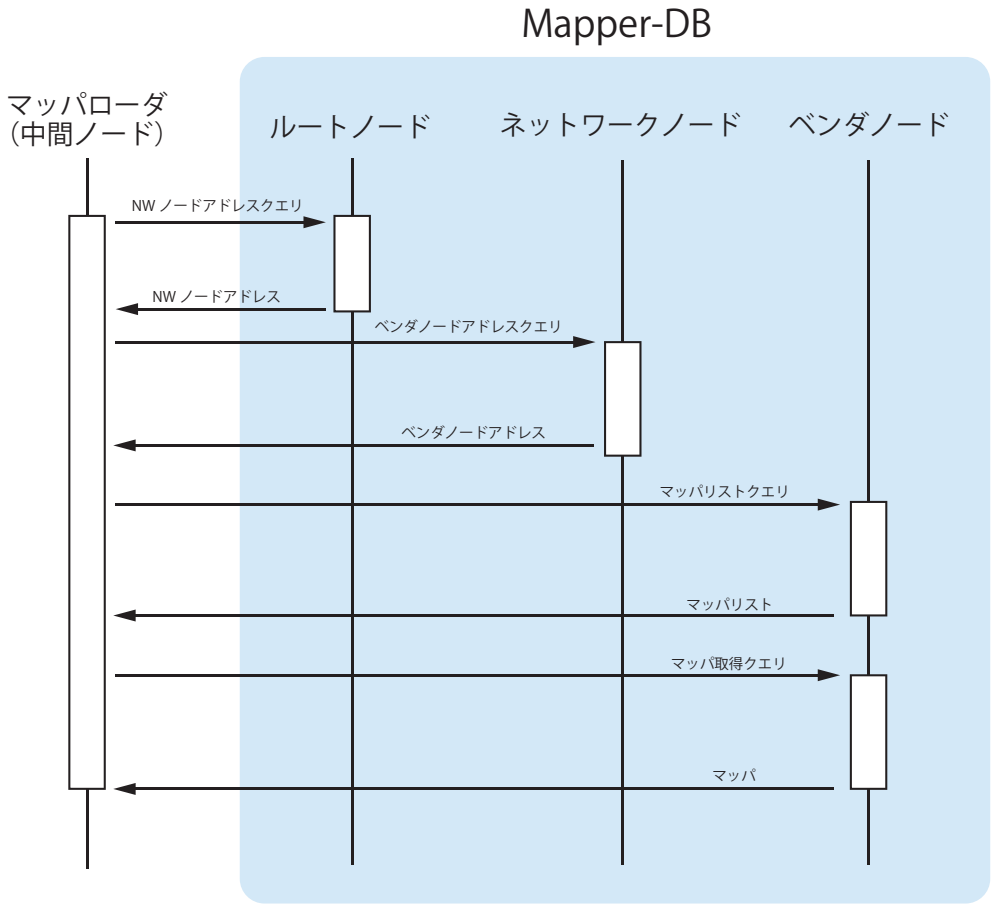

図 4.4　マッパローダが Mapper-DB から新しいマッパを探して取得する処理のシーケンス

- トランスポート層ネットワーク識別子 (例: `ipv4`, `bluetooth`)
- サービスインタラクションプロトコル識別子 (例: `upnp`, `jini`)
- 製品ベンダ識別子 (例: `ht`)
- 製品識別子 (例: 製品型番)
- 製品製造番号 (例: `1`, シリアルナンバ)

ユニーク ID はこれらの要素をハイフンで連結したものとしている．例えば `ipv4-upnp-ht-htTV-1` などがユニーク ID の例である．

マッパはそれぞれのトランスポート層ネットワークで，自分が理解できるサービスインタラクションプロトコルを使って製品を見つけることができる．そのプロトコルのパケットを解釈することで，これらの情報を得ることができる．

4.2.3 DDI-Repository の設計

分散データベース

DDI-Repository は Mapper-DB とほぼ同じ構造になっており，同様の分散データベースである．おおよそ，同じメリットやデメリットを持つのでここでは割愛する．

ツリー構造

DDI-Repository は 3 層からなるツリー構造になっている．

DDI-Repository の全体像を図 4.5 に示す．

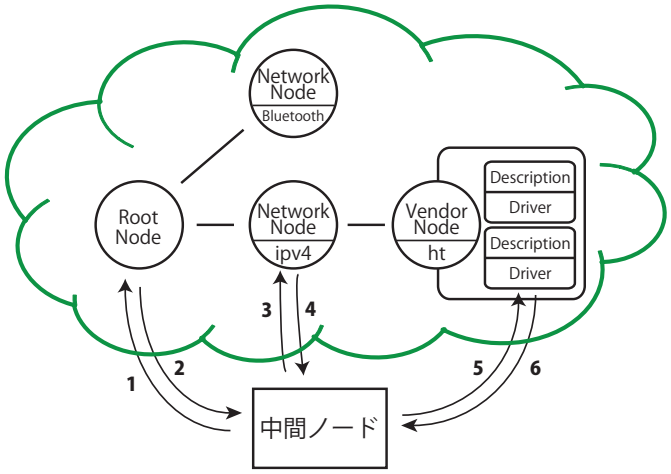

図 4.5　DDI-Repository を利用したデバイス依存情報解決の様子

Mapper-DB へのクエリと違い，DDI-Repository へのクエリは明確に「このデバイスのデバイス依存情報が欲しい」という要求がある．「このデバイス」というアイデンティティはユニーク ID で示され，各データベースの中でユニーク ID をもとにマッチングされる．デバイス依存情報は最終的にベンダノードに格納されている．

1 層目はルートノードである．ルートノードは Mapper-DB と同じく，ネットワークノードへのポインタを示している．クエリのユニーク ID のネットワーク部分を見て，該当するネットワークノードへのアドレスを返信する．

2 層目はネットワークノードである．ネットワークノードはベンダノードへのポインタを示している．Mapper-DB との違いは DDI-Repository のネットワークノードは単一のアドレスしか返さないことである．デバイス依存情報を探しているデバイスのユニーク ID から，ベンダ情報を抜き出し，自分の管理下の中で該当するベンダノードのアドレスを返信する．

3 層目はベンダノードである．各ベンダは自分のノードを持つ事ができ，デバイス依存情報を自由に公開／更新することができる．ベンダノードにはユニーク ID と共に，関連付けられてデバイス依存情報が格納されており，最終的にデバイス依存情報解決のクエリはこのノードに到達するこ

とでデバイス依存情報を解決することができる．ベンダノードはデバイスのサービスを利用するプログラムであるドライバに関して，プラットフォームごとに違うバイナリファイルを格納することができ，中間ノードの実行環境にあったバイナリをダウンロードすることができる．

リゾルバはこれら複数層のノードを渡り歩いてデバイス依存情報を取得し，インストールする．

デバイス依存情報提供シーケンス

本節では，デバイスのベンダがサービス記述およびドライバを，DDI-Repository を通じて提供を行う具体的な手順について述べる．

ベンダは，デバイスが実際に通信を行うネットワークに対応した DDI-Repository 上のネットワークノード下にある，自分のベンダノードを管理している想定であり，そこでベンダノードにサービス記述とドライバをファイルとして配置し，設定ファイルを書き換え，ベンダノードプログラムを再起動することでデバイス依存情報を公開することができる．

ベンダノードプログラムの設定ファイルはシンプルな XML ファイルになっており，以下のような形式となっている．

ソースコード 4.2　DDI-Repository ベンダノードプログラムの設定ファイル例

```
1  <?xml version="1.0" ?>
2  <vendorNodeConfiguration type="device-dependent-information">
3    <identity>
4      <network>ipv4</network>
5      <vendor>ht</vendor>
6    </identity>
7
8    <deviceDependentInformations>
9      <deviceDependentInformation>
10       <id>ipv4-upnp-ht-UPnPMusicServer-1</id>
11       <serviceDescription type="file">resource/
             servicedescription/ipv4-upnp-ht-UPnPMusicServer-1.usdl
             </serviceDescription>
12       <drivers>
13         <driver platform="java" type="file">resource/fs-driver/
               FileSystemUPnPMusicServerDriver.class</driver>
14       </drivers>
15     </deviceDependentInformation>
16
17     <deviceDependentInformation>
18       <id>ipv4-upnp-ht-UPnPMusicPlayer-1</id>
```

```
19        <serviceDescription type="file">resource/
              servicedescription/ipv4-upnp-ht-UPnPMusicPlayer-1.usdl
              </serviceDescription>
20        <drivers>
21          <driver platform="java" type="file">resource/fs-driver/
                FileSystemUPnPMusicPlayerDriver.class</driver>
22        </drivers>
23      </deviceDependentInformation>
24
25      <deviceDependentInformation>
26        <id>ipv4-upnp-ht-JiniMusicPlayer-1</id>
27        <serviceDescription type="file">resource/
              servicedescription/ipv4-jini-ht-JiniMusicPlayer-1.usdl
              </serviceDescription>
28        <drivers>
29          <driver platform="java" type="file">resource/fs-driver/
                FileSystemFakeMusicPlayerDriver.class</driver>
30        </drivers>
31      </deviceDependentInformation>
32    </deviceDependentInformations>
33  </vendorNodeConfiguration>
```

identity 要素では，自身が DDI-Repository のどのネットワークノード下にあるのか，どのベンダ名で登録するのかを記述する．deviceDependentInformations 要素では，デバイス依存情報を列挙する．ここで挙げた例では，IPv4 ネットワークの，ベンダ名 ht というノードとして，DDI-Repository データベースネットワークに参加している．

デバイス依存情報はユニーク ID を記述する id 要素，サービス記述ファイルのロケーションを記述する serviceDescription 要素，各プラットフォームごとのドライバファイルのロケーションを記述する drivers 要素で構成される．drivers 要素で，ドライバを複数記述できることからもわかるように，DRIAD では Java だけに依存しないオープンな設計を目指している．さまざまなプラットフォームごとのドライバが提供されれば，さまざまな中間ノード上でドライバを動作させることができ，DDI-Repository の価値が向上し，よりインタオペラビリティの向上に役に立つと言える．

デバイス依存情報解決シーケンス

本節では，中間ノードがマッパによってネットワーク上のデバイスの存在を検知したあと，DDI-Repository を用いてデバイス依存情報を解決する具体的な方法について述べる．処理のシー

ケンス図を図 4.6

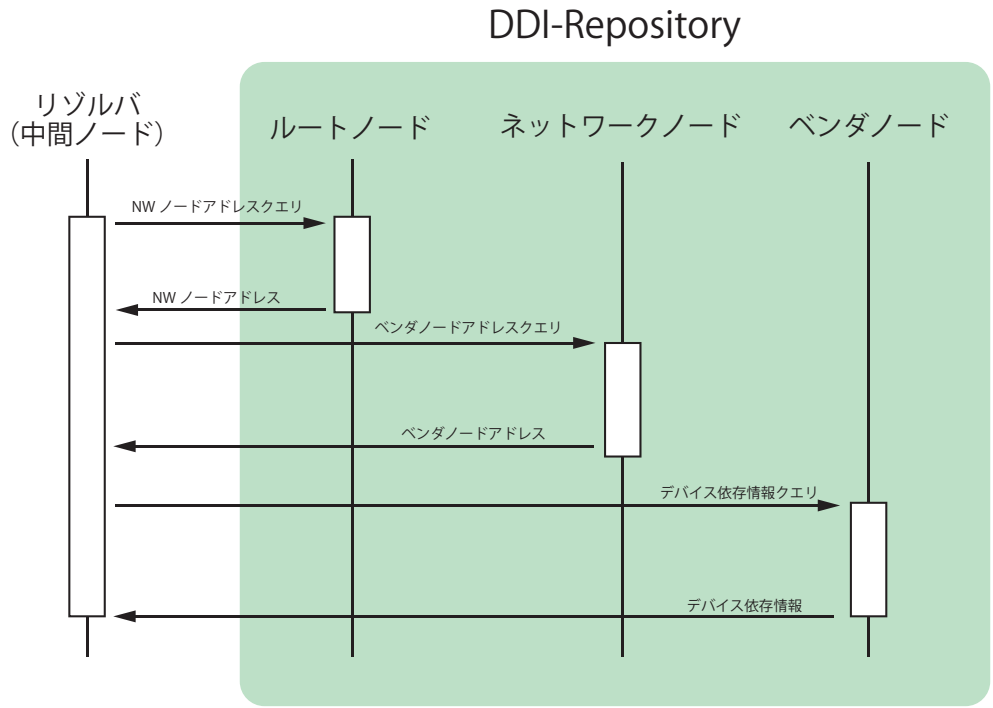

図 4.6　リゾルバによる DDI-Repository からのデバイス依存情報解決処理シーケンス

リゾルバはまず，DDI-Repository のルートノードに対して，ユニーク ID 中のトランスポートネットワークに対するノードのアドレスを問い合わせるクエリを行う．この際，ルートノードのアドレスは DNS のようにあらかじめ定義されているものと想定している．

4.3　DRIAD の実装

4.3.1　実装環境

実装マシン

DRIAD および，DRIAD の実装プラットフォームとして利用したインタオペラビリティのためのミドルウェア u-Glue の開発は Apple 社のオペレーティングシステム Mac OS X を用いたパーソナルコンピュータ Mac Pro 上で行った．

Mac Pro は表 4.1 のようなスペックを持っている．

また，DRIAD および u-Glue の開発には記述されたプログラムの動作が環境に依存しないプログラミング言語 Java を利用した．

Java は仮想マシンの概念を用いることで，実際に動作するハードウェアを抽象化し，一度記述されたプログラムをプラットフォームを問わずに同じように動作させることができる．

このような性質から，環境を問わず広く利用される可能性のあるソフトウェアの開発には Java

CPU	2.8 GHz Quad Core Intel Xeon 2 基
メモリ	16GB
HDD	320GB SATA 1 基
JDK	JavaSE 1.5.0

表 4.1　実装環境 MacPro のスペック

言語は非常に適しているため，本研究でも DRIAD の実装に Java 言語を採用した.

実装デバイス

パフォーマンス評価実験を行うため，UPnP と Jini を使った仮想デバイスを Java 言語で実装した.

UPnP 言語では，独自のアクションで楽曲リストと，楽曲のデータを提供するネットワーク音楽サーバと，楽曲データをネットワーク経由で入力することで音楽を鳴らすネットワーク音楽プレイヤを実装した.

Jini 言語では先の UPnP で実装されたネットワーク音楽プレイヤと USDL の operation 定義が同じになるような，ネットワーク音楽プレイヤを実装した.

UPnP のネットワーク音楽サーバのデバイス記述は以下のようになっている.

ソースコード 4.3　UPnP ネットワーク音楽サーバのデバイス記述 (UPnP)

```
1  <?xml version="1.0" ?>
2  <root xmlns="urn:chemas-upnp-org:device-1-0">
3    <specVersion>
4      <major>1</major>
5      <minor>0</minor>
6    </specVersion>
7
8    <device>
9      <deviceType>urn:schemas-upnp-org:device:tomotakaMusicPlayer
           :1</deviceType>
10     <friendlyName></friendlyName>
11     <manufacturer>tomotaka ito</manufacturer>
12     <manufacturerURL>http://www.ht.sfc.keio.ac.jp/~tomotaka/</
           manufacturerURL>
13     <modelDescription>Tomotaka's Master Thesis Sample
           Application</modelDescription>
14     <modelName>MusicPlayer</modelName>
```

```
15      <modelNumber>1.0</modelNumber>
16      <modelURL>http://www.ht.sfc.keio.ac.jp/~tomotaka/</modelURL>
17      <serialNumber>1234567890</serialNumber>
18      <UDN>uuid:tomotakaMusicPlayer</UDN>
19      <UPC>123456789012</UPC>
20      <iconList>
21        <icon>
22          <mimetype>image/gif</mimetype>
23          <width>48></width>
24          <height>32</height>
25          <depth>8</depth>
26          <url>icon.fig</url>
27        </icon>
28      </iconList>
29      <serviceList>
30        <service>
31          <serviceType>urn:schemas-upnp-org:service:player:1</
                serviceType>
32          <serviceId>urn:schemas-upnp-org:serviceId:player:1</
                serviceId>
33          <SCPDURL>/service/player/description.xml</SCPDURL>
34          <controlURL>/service/player/control</controlURL>
35          <eventSubURL>/service/player/eventSub</eventSubURL>
36        </service>
37
38
39      </serviceList>
40      <presentationURL>http://www.ht.sfc.keio.ac.jp/~tomotaka/</
            presentationURL>
41    </device>
42  </root>
```

Jini で記述されたネットワーク音楽プレイヤの Java インタフェースは以下のようになっている.

ソースコード 4.4　Jini ネットワーク音楽プレイヤの定義

```
1         public interface MusicPlayer {
2           public void playSong(byte[] data) throws
```

```
                RemoteException;
        }
```

playSong メソッドはリモートから受け取った音楽データのバイト列を受け取り，音楽を再生する．Jini の音楽プレイヤは MP3 を想定しているため，このサービスのドライバはこのサービスを実行する際に与える音楽データのセマンティクスを解釈し，適宜 MP3 に変換を行う．

これらのデバイスをインタオペラブルにサービスを行うために，さらに USDL でサービスを記述している．以下は UPnP ネットワーク音楽プレイヤの例である．

ソースコード 4.5　UPnP ネットワーク音楽プレイヤの USDL

```
<?xml version="1.0" ?>
<entity>
  <properties>
    <id>ipv4-upnp-ht-UpnPMusicPlayer</id>
    <inherit></inherit>
    <name lang="ja">UPnP Network Player</name>
    <date>2010/01/10</date>
    <version>1.0.0</version>
    <provider>Hide. Tokuda Lab.</provider>
    <description lang="en">Network Music Player using UPnP
        Technology</description>
    <description lang="ja">
        UPnP技術を用いたネットワーク音楽プレイヤ </description>
  </properties>

  <interface>
    <operation name="play" sequence="in">
      <description lang="ja">楽曲を再生する</description>
      <description lang="en">play music</description>
      <input>
        <data type="mime" connectionType="packet">audio/mp3</
            data>
        <description lang="ja">オーディオデータ</description>
        <description lang="en">Audio Data</description>
      </input>
    </operation>
  </interface>
```

```
25 </entity>
```

4.3.2 ミドルウェア u-Glue

本節では，DRIAD が研究対象としているユビキタスインタオペラビリティのためのフレームワークの実験的な実装として，筆者が用意したフレームワーク u-Glue について説明を行う．

USDL: 抽象サービス記述言語

u-Glue では，サービスを抽象的に記述するためのフォーマットとして，慶應義塾大学徳田研究室で研究が進められている USDL(Universal Service Description Language) を用いる．

USDL は XML をベースとした抽象サービス記述言語であり，サービスのメタ情報と，デバイスがどのようなサービスを持つかという情報をプラットフォームに依存しない形式で表現することが可能である．

USDL は `properties` 要素と，`interface` 要素から構成される．`properties` 要素では，デバイスのメタデータを表現しており，デバイスを作成したベンダの情報や，デバイス固有の ID などが主に記述される．

`interface` 要素では，外部へ公開されるサービスのリストが定義されており，どのような入力や出力があるかといった項目を記述することができる．それぞれのサービスは `operation` 要素で表現される．

デバイス依存環境管理 API の設計

u-Glue は図 4.7 に示すような API を用いて，サービスディスカバリを行うマッパを持続的に読み込む機構，およびそれらのマッパによりデバイスをネットワーク上で発見した際に対応するデバイス依存情報とドライバを解決する機構を提供している．

`MapperLoader` は文字通りマッパを継続的に読込み，随時新しいスレッドとしてスタートさせ，ネットワークを監視させるプログラムの実装のための API で，

`DescriptionResolver` と `DriverLoader` はそれぞれユニーク ID にサービス記述とドライバを読み込むのプログラムの実装のための API である．

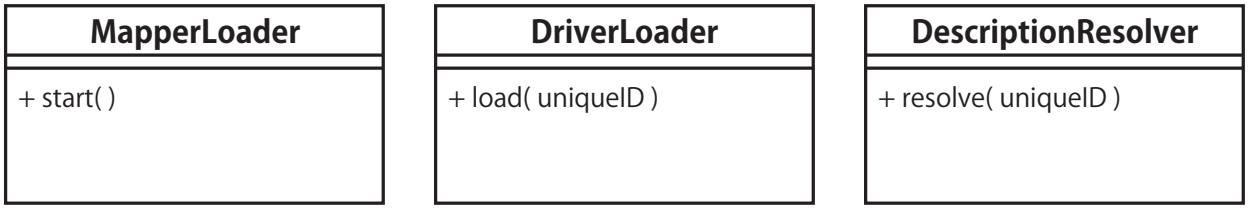

図 4.7　u-Glue のマッパ/デバイス依存情報管理 API

デフォルトでは u-Glue はこれらのクラスを拡張したファイルシステムからマッパやサービス記

述，ドライバを読み込む仕様になっているが，これらの API を用いて DRIAD を組み込むことができる．

本研究の前提環境に対して

u-Glue は，先に述べた本研究が前提としているミドルウェアの要件を満たすように設計されている．

u-Glue は前述のように USDL をサービス記述として利用している．USDL は単なる XML ファイルであり，DRIAD を組み込む前の状態では u-Glue は XML ファイルをローカルファイルシステム上で静的に管理しており，あらかじめ記述したデバイスのユニーク ID との対応にしたがってロードされ，メモリ上にロードされる．

u-Glue は各デバイスを実際に制御するためのドライバを Java のコンパイル済み class ファイルとしてローカルファイルシステム上で管理しており，USDL の type 型に基づいてデバイスに対応するドライバプログラムのクラス名を決定し，Java の組込み機能である ClassLoader を使って実行時にロードするようになっている．

前節で述べたように，サービス記述やドライバをデバイスのユニーク ID に基づいて解決するための API と，デバイスのユニーク ID に対応するサービス記述やドライバをインストールするための API を備えており，これに準ずる設計を行えば容易に DRIAD のコア機能を u-Glue に組み込むことが可能となっている．

前提環境の節で挙げた，以上の 3 つの前提条件を満たすため，u-Glue は DRIAD のプロトタイプ開発環境として機能する．

具体的なサービス連携手法

u-Glue では，制御コンソールで選択した二つのサービスを連携させることができる．

二つのサービスの入出力の互換性は，USDL で定義されている input 要素，output 要素の data 要素で指定される mime タイプによって決定される．`audio/*`タイプであればいったん連携元サービスからの出力をニュートラルな wav 形式に変換を行い，連携先のサービスへの出力時に連携先サービスの output 側に応じて変換を行い，各サービスを駆動するドライバに引数として渡される．

ドライバはファイルシステムから読み込まれるものと，DRIAD により動的に解決されるものがあり，両方とも Java 言語の interface 機能による多態性によって u-Glue からは同等に扱うことができる．

4.3.3 Mapper-DB の実装

本節では，Mapper-DB の実装について述べる．Mapper-DB 設計の節でも述べたように，3 層からなる階層型分散データベースであり，各ノードによって実行されるプログラムが違う．ノードの種類は全部でルートノード，ネットワークノード，ベンダノードである．クライアントと各ノードは XML によりメッセージを交換する．UDP パケットによる通信を使用する場合があるため，

通信パケットのヘッダに MD5 チェックサムを入れており，通信内容を確かめている．

ルートノード

Mapper-DB のルートノードは，全てのネットワークノードのアドレスを，そのネットワークノードが対応しているトランスポート層ネットワークのシンボル文字列と対応させて保持している．

ルートノードは UDP/IP でクライアントと通信を行う．UDP10512 番ポートで接続を待ち受けており，クライアントからのリクエストを受信した際には，リクエストに記載されているネットワークを示す文字列と，自身のメモリ内のルックアップテーブルを照らし合わせ，該当するネットワークノードのアドレスを返す．クライアントに返信する際も UDP/IP により XML メッセージを送出する．

ネットワークを示す文字列というのは具体的には `ipv4` や `bluetooth` といった文字列を想定している．UPnP や Jini というサービスレベルのネットワークではなく，トランスポートレベルのネットワークを想定している．

ネットワークノード

Mapper-DB のネットワークノードは，自身が管理するトランスポート層ネットワークで動作する Mapper を持つ，全てのベンダノードのアドレスをベンダ名の文字列と対応させて保持している．

クライアントはネットワークノードにアクセスする前に，ルートノードに該当するネットワークの Mapper を管理しているネットワークノードのアドレスを訪ねる．アドレスを取得したのち，ネットワークノードにアクセスを行う．

クライアントからネットワークノードへの通信も UDP/IP を用いて行われる．ルートノードと同様にレスンポスも UDP パケットを用いる．ネットワークノードはクライアントから自身が管理するベンダノードリストのリクエストがあった場合，クライアントがリクエストしているネットワーク表現文字列が，自分の担当するネットワークと一致すれば，ベンダノードリストを XML として UDP パケットでクライアントへ送信する．UDP パケットを待ち受けるポート番号は 10513 である．

ベンダノード

Mapepr-DB のベンダノードは，自身が管理するマッパプログラムを，そのマッパ ID と対応させて保持している．

クライアントはベンダノードにアクセスする前に，ルートノードにネットワークノードのアドレスをたずね，レスポンスされたアドレスにあるネットワークノードに対して，アドレスを含むベンダノードのリストを参照している．

クライアントとベンダノードのやり取りは 2 ステップに分かれており，最初のステップはそのベンダノードが保持している全てのマッパのリストを得る通信，次のステップは必要に応じてクラ

イアントが指定したマッパを取得するための通信である．これらの，特に後者の通信においてそれなりに大きなファイルを正確に送信しなければならないという通信の性格上，MapperDB のベンダノードは，ルートノードとネットワークが UDP を通信に用いるのに対し，TCP を用いている．TCP 接続をリッスンしているポート番号は 10514 である．

ベンダノードとの通信が 2 ステップに分かれている理由は，中間ノードはベンダノードが持つマッパのうち，自分が持っていないもののみを取得すればよいからである．最初の手順で中間ノードは自分が持つマッパのリストと，ベンダノードが持つマッパのリストを比較し，持っていないもののみを 2 つ目の手順を使ってリクエストする．このプロトコルは Mapper-DB との通信のうちでも比較的重要な部分なのでクライアントがどのようなフォーマットの XML でマッパの取得を行うかをリスト 4.6 に，ベンダノードがマッパプログラムをどのようなフォーマットでクライアントにレスンポンスするかをリスト 4.7 に示す．

ソースコード 4.6　マッパプログラム取得のリクエスト XML

```
1 <mapperRequest>
2   <requestMapperId>ipv4-ht-MyServiceInteractionProtocol-java</
      requestMapperId>
3 </mapperResponse>
```

ソースコード 4.7　マッパプログラム取得のレスポンス XML

```
1 <mapperResponse>
2   <mapper>
3     <length>12345</length>
4     <encoding>base64</encoding>
5     <id>ipv4-ht-MyServiceInteractionProtocol-java</id>
6     <checksum type="md5">18BA37ED13F4A1B9887FEE576387FE2D</
        checksum>
7     <binary>...</binary>
8   </mapper>
9 </mapperResponse>
```

mapper 要素には，マッパを構成する様々な要素が格納されている．length 要素はバイナリデータの長さ (バイト長) を示している．id は，マッパの ID を示している．checksum 要素は，type 属性で指定されたチェックサム計算アルゴリズムで作成されたバイナリデータのチェックサムが格納されている．例では，`md5` が指定されているため，MD5 アルゴリズムによるチェックサムが格納されている．binary 要素には実際に動作するマッパプログラムがのバイナリデータが encoding 要素で指定された符号形式で符号化されて格納される．例では `base64` が指定されているため，電子メールなどの通信でも利用される Base64 エンコーディングを用いて，符号化され，格納される．

Base64 エンコーディングは古典的なエンコーディング方式であるが，3 バイトのバイナリデータを 4 バイトのアスキー文字列で表現できるため，単純に 16 進数をアスキー文字でダンプするより効率がいいことから採用した．

現在の実装では，ベンダノードが持つマッパのうち，複数のマッパを取得したい場合は複数回 TCP セッションを張らなければいけない実装になっている．

各ノードの通信仕様のまとめ

表 4.2 に Mapper-DB の通信方式をまとめた．ベンダノードは TCP による通信なのでストリーム通信を行うため，送信ポート番号を記述していない．

ノード	通信方式	受信ポート番号	送信ポート番号
ルートノード	UDP	10512	30512
ネットワークノード	UDP	10513	30513
ベンダノード	TCP	10514	-

表 4.2　Mapper-DB 各ノードの通信仕様のまとめ

4.3.4　DDI-Repository の実装

本節では，DDI-Repository の実装について述べる．DDI-Repository も Mapper-DB と同じように，3 層からなる階層型分散データベースであり，各ノードによって実行されるプログラムが違う．ノードの種類は全部で，ルートノード，ネットワークノード，ベンダノードである．Mapper-DB と同じように，XML をメッセージングに用いており，MD5 チェックサムを使うなどの詳細なプロトコルの仕様も一緒である．

ルートノード

DDI-Repository のルートノードは，全てのネットワークノードのアドレスを，そのネットワークノードが対応しているトランスポート層ネットワークのシンボル文字列と対応させて保持している．

ルートノードは UDP/IP でクライアントと通信を行う．待ち受けているポート番号は 20512 番である．クライアントからリクエストされたネットワークを示す文字列と，自身のメモリ内にあるルックアップテーブルを照らし合わせ，該当するネットワークノードのアドレスを返す．

ネットワークノード

DDI-Repository のネットワークノードも，Mapper-DB のそれとほぼ同じ働きをする．異なる点は Mapper-DB のネットワークノードはクエリに対し，自身が管理する全ての Mapper-DB ベンダノードのアドレスを返すことに対して，DDI-Repository のネットワークノードは，ベンダを

指定したクエリに対し，そのベンダ名に該当するベンダノードのアドレスのみを返す点である．

ネットワークノードは UDP ポート 20513 番でリクエストを待ち受け，リクエストで指定されたベンダのアドレスを返信する．

ベンダノード

DDI-Repository のベンダノードは，Mapper-DB のそれと格納している情報が異なる点がまず大きく違う．DDI-Repository のベンダノードは，デバイス依存情報をユニーク ID に関連づけて保持しており，リクエストで指定されたユニーク ID に対応するデバイス依存情報をクライアントへ提供する．Mapper-DB と同様に，クライアントとベンダノードの通信が始まる前には，ルートノードへのクエリ，ネットワークノードへのクエリが必要になる．

ベンダノードはサービス記述ファイルと，任意のプラットフォームのドライバをデバイス依存情報として，保持している．ベンダノードと，クライアント間で利用されるメッセージ XML のフォーマットをリスト 4.8, リスト 4.9 に示す．

ソースコード 4.8　DDI-Repository ベンダノードへのデバイス依存情報取得クエリ

```
<deviceDependentInformationRequest>
  <id>ipv4-upnp-htlab-htTV-12345</id>
</deviceDependentInformationResponse>
```

リクエストのパケットではシンプルに，デバイス依存情報を手に入れたいデバイスのユニーク ID を id 要素に書き込んでいる．

ソースコード 4.9　DDI-Repository ベンダノードからのデバイス依存情報レスポンス

```
<deviceDependentInformationResponse>
  <id>ipv4-upnp-htlab-htTV-12345</id>
  <serviceDescription>....</serviceDescription>
  <drivers>
    <driver platform="java" encoding="base64">....</driver>
  </drivers>
</deviceDependentInformationResponse>
```

レスポンスのパケットでは，マッパ取得のレスポンスパケットと同じように，実際に動作するプログラムを含むメッセージとなっている．順番に見て行くと，id 要素はレスンポンスメッセージに含まれるデバイス依存情報が，どのユニーク ID を持つデバイスのためのものかを示している．この要素はリクエストで送信された ID と完全に一致する．serviceDescription 要素は現在はサービス記述言語の USDL を想定している．サービス記述は実際にプログラムを動作させる環境を選ぶことはないので，ここでは設計思想として複数のサービス記述言語の格納は許容せず，USDL のみを扱えるようにベンダノードを設計した．drivers 要素は，プラットフォームを示す

文字列と，それに対応する，デバイスのサービスを駆動するためのドライバプログラムをプラットフォームごとに driver 要素にまとめて列挙していいる．driver 要素は単純プラットフォーム情報を格納する platform 属性と，本体部分のエンコーディング方式を記述している encoding 属性，そしてバイナリデータを示すテキストノード部分から構成されている．この例では Java のクラスファイルをバイナリデータとして，Base64 エンコーディングで符号化した文字列をテキストノードとして保持している．

各ノードの通信仕様のまとめ

表 4.3 に DDI-Repository の通信方式をまとめた．ベンダノードは TCP による通信なのでストリーム通信を行うため，送信ポート番号を記述していない．

ノード	通信方式	受信ポート番号	送信ポート番号
ルートノード	UDP	20512	40512
ネットワークノード	UDP	20513	40513
ベンダノード	TCP	20514	-

表 4.3　DDI-Repository 各ノードの通信仕様のまとめ

4.3.5　u-Glue への組み込み

u-Glue は，以下の 3 つのインタフェースを用いて，ファイルシステムからのマッパ，デバイス依存情報の読込みと，DRIAD からの読込みの差異を意識せずにアプリケーション全体を構築できるよう設計されている．

- MapperLoader: マッパを随時ロードし，実行開始させるプログラムのための API
- DescriptionResolver: ユニーク ID に対応するサービス記述を読み込むためのプログラムの API
- DriverLoader: ユニーク ID に対応するドライバを読込み，インストールするためのプログラムの API

MapperLoader が Mapper-DB からマッパを，DescriptionResolver と DriverLoader が DDI-Repository からデバイス依存情報を取得するようになっている．

マッパを見つけ次第随時ロードし，サービスディスカバリを開始させる MapperLoader とその子クラスは，図 4.8 のようなクラス図になっており，FSMapperLoader はローカルのファイルシステム上から読み込むだけのシンプルなマッパローダであり，DRIADMapperLoader は DRIAD の Mapper-DB から随時新規マッパを検索する実装になっている．

デバイス依存情報を解決する二つのクラスと，その子クラスは，それぞれ図 4.9，図 4.10 のようなクラス構成になっている．

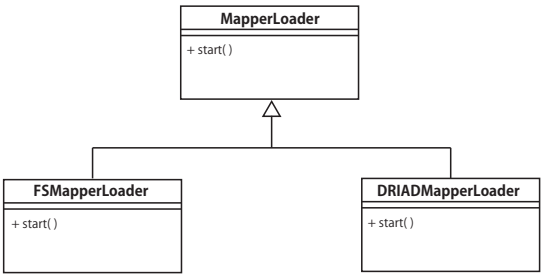

図 4.8　MapperLoader とその子クラス

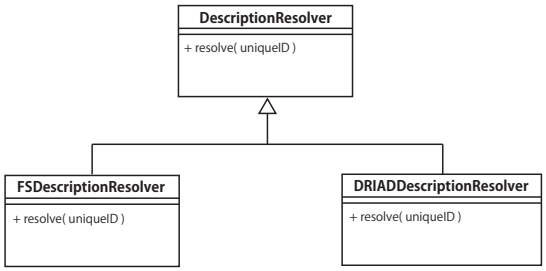

図 4.9　DescriptionResolver とその子クラス

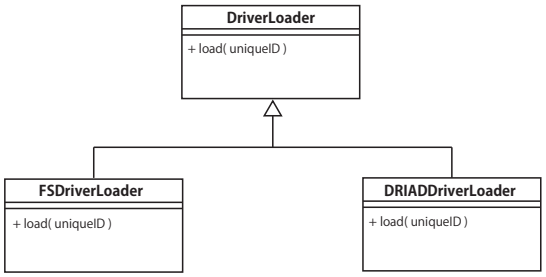

図 4.10　DriverLoader とその子クラス

u-Glue が提供するこれらの API を通じて，DRIAD のモジュールがリモートの Mapper-DB と DDI-Repository の二つのデータベースから取得してくるマッパとデバイス依存情報を，インタオペレーションフレームワークに組み込むことができる．

4.4　本章のまとめ

本章では，デバイス依存情報を動的な解決を行うためのシステム「DRIAD」の設計と実装について述べた．

前提環境の節では，具体的に DRIAD を組み込むのに必要なフレームワーク側の制約について述べた．

DRIAD の設計の節では，Mapper-DB や DDI-Repository などの分散データベースシステムのモデルや，システムが行う一連の処理のシーケンスなどについて述べた．

DRIAD の実装の節では，Mapper-DB や DDI-Repository の実装の詳細などについて述べた．

第 5 章

関連研究

本章では，本研究の関連研究についてそれぞれその立ち位置について述べ，本研究と関連する部分について述べる．また目的意識を同じくしている研究およびプロジェクトについては本研究との差異について考察する．

5.1 サービスディスカバリ技術

5.1.1 Bonjour

Bonjour [7] は Apple 社によって策定された，IP ネットワークにおけるマルチキャスト通信を用いた，サービスディスカバリのためのプロトコルである．のちに IETF によって技術が吸い上げられ，Zeroconf という名前で標準化された．厳密にはサービスディスカバリだけではなく，自動的にアドレス付与を行うなどの規格も含む．

あるサービスに対して，それを利用する実装が既に存在する場合は，Bonjour のようなシンプルなサービスディスカバリは非常に便利に利用できる．機器の存在を確認し，その機器に対して既にある実装を使ってアクチュエートすればよいためである．

SLP も含め，サービスディスカバリのみのプロトコルでは，本研究の問題意識のうち，未知のデバイスがどのようなデバイスかを自動的に認識し，それを使えるようにする，という課題がクリアできない．

5.1.2 INDISS

Yerom らによる INDISS [5] は，単一のプロトコルではなく，サービスディスカバリの技術をインタオペラビリティのために統一的に扱えるようにするフレームワークである．本研究では INDISS をマッパ部分に用いなかったが，中間ノードを含めた実装を行う際はこの技術を使うことで，サービスディスカバリからの処理を単純化できる可能性がある．

5.2 サービス相互運用技術

5.2.1 UPnP

UPnP(Universal Plug and Play) は Microsoft 社によって策定された，IP ネットワークにおけるサービスディスカバリと，サービス相互運用のためのプロトコルである．

UPnP はマルチキャストでサービス発見を行ったり，メディア再生装置や，メディアサーバなどの基本的なデバイス型を定義することによって現在でも多くの家電機器がサポートするようになった現在では最も身近なサービス相互運用のための技術である．ネットワーク TV やゲーム機などのデジタル情報家電がサポートする DLNA も UPnP/AV がベースとなっている技術である．

デバイスディスカバリ，サービス記述，サービス相互運用のためのプロトコルと，IP ネットワークに閉じた環境では非常に強力な機能を持つ UPnP だが，異種ネットワーク間でのサービス連携が行えないことなどから，ユビキタス環境で期待されるインタオペラブビリティが実現できない．

5.2.2 Jini

Jini は Sun Microsystems 社によって策定された，IP ネットワークにおけるサービスディスカバリと，サービス相互運用のための技術である．

Java の RMI(Remote Method Invocation) 技術をベースとした Jini では，サービスを利用する側で抽象的な機能を定めた interface だけ保持しておればよく，サービスを使う際にシリアライズされた Java のバイトコードをネットワーク越しに取得し，実行するという設計になっている．

本研究の問題意識であるデバイス依存情報のうちドライバを自動的に解決する機能に相当するが，実際にサービスの利用を行うためには，抽象的な機能を定めた interface を保持しておく必要があるため，デバイス依存情報のうちサービス記述を自動的に解決できない点が問題となる．

また Java 言語で IP ネットワークという閉じた範囲のインタオペラビリティであるため，多種多様なデバイスとネットワークが混在するユビキタス環境では実際の利用は難しい．

5.2.3 WSDL

WSDL [6] は，現在 W3C によって勧告されている Web サービスのサービス記述言語である．WSDL を用いることによって，Web サービスがどのような入力および出力を行うのかを記述することができ，入力と出力のフォーマットを調べることで Web サービスのプログラムからの利用が行えるようになる．

WSDL により，世界中のどこからでもアクセスできるサービスのメタ記述を表現できるようになったため，様々な研究が継続的に行われている．セマンティック Web と呼ばれる分野の研究では，WSDL も利用した Web のオートメーション化が進められている． [10] しかし，セマンティクスの粒度が不十分だったため，それを補うような研究も出現した． [9]

Web サービスはインターネット接続性さえあればどこからでも，いつでも利用でき非常にユビキタスコンピューティングと相性のよい技術であるため，WSDL を使ってうまく Web サービスを異種ネットワーク間でインタオペラブルなサービスとしてブリッジするための仕組みが提供されることが望ましい．

本研究で用いた，ユビキタス環境を想定したデザインである USDL にうまく変換することができれば実空間上での Web サービスの役割などもより正確に表現できるようになり，応用の幅が広がると考えられる．

5.2.4 OSGi

OSGi(Open Service Gateway Initiative) [1] は，Java 言語上で動作する比較的新しいプラットフォームであり，OSGi が動作するゲートウェイから，コンポーネントを遠隔で更新したり，インストールしたり，停止したりといった制御から，連携させることも可能なフレームワークである．OSGi を用いて家庭内のデバイスをコントロールしてスマートスペースを構築する研究も行われて

いる．[8]

OSGi のアイディアは非常に本研究の問題意識をカバーする部分があるが，Java 環境に特化した設計のため，ネットワークやプラットフォームの多様性を意識したインタオペラビリティの提供を行うことができない．

5.3 インタオペレーション技術

5.3.1 uMiddle

uMiddle [11] は慶應義塾大学の中澤らによる研究で，中間ノードによるセマンティクス翻訳モデルを用いたインタオペラビリティのためのフレームワークである．

uMiddle は異種ネットワーク間のインタオペラビリティを実現するためのほぼ完全なフレームワークを提供しており，デバイスを発見する Mapper，サービス同士のセマンティクス表現を翻訳する Translator，サービス記述のための USDL といったコンポーネントで構成されており，本論文で試作したプロトタイプ u-Glue も uMiddle を非常に参考にして実装された．

uMiddle はサービス同士を連携させるための機能に不足はなく，異種ネットワーク間でサービスをブリッジすることが可能であるが，Mapper や，Translator の動的な追加などには明示的に触れられていない．このことから，本研究で問題意識として挙げている，デバイス依存情報の自動的解決機構の欠如が依然として解決されていない．

uMiddle はその設計から，DRIAD をシステムに組み込めるミドルウェアであると推測されるので，DRIAD を組み合わせることでインタオペラビリティを向上できるものと考えられる．

5.4 本章のまとめ

本章では，本研究の関連研究として，サービスディスカバリ技術，サービス相互運用技術，インタオペレーション技術の 3 つに既存研究を分類したあと，それぞれがインタオペラビリティを実現した理想的な状態に対して，どの程度実現できているか，また本研究が挙げている問題意識に対してどの程度の解決策を持っているのかについて比較しながら議論を行った．

第 6 章

評価

本章では，DRIAD の評価について述べる．まず評価方針について述べ，次いで行った評価実験について述べ，最後に得られた実験結果からシステムの評価について考察を行う．

6.1 定性的評価

DRIAD を用いた場合と，用いない場合について定性的評価を行う．また関連研究との比較による定性的評価についても行う．

表 6.1 は関連研究と，サービス連携のための機能や，インタオペラビリティを向上するための機能の有無を比較した表である．比較対象の関連研究として，IP ネットワークにおけるサービス相互運用を実現している技術である UPnP と，Jini を，また中間ノードを用いたインタオペレーションフレームワークとしての関連研究として uMiddle を機能評価の対象とした．本研究で開発したシステム DRIAD は単体では，インタオペラビリティの機能は成さないため，本論文で作成したミドルウェア uGlue と組み合わせた際の機能を，他の研究対象と比較している．uGlue 単体の機能は uMiddle を参考に実装されているため，uMiddle とほぼ同じと考えてよい．

-	UPnP	Jini	uMiddle	u-Glue+DRIAD
サービスディスカバリ	○	○	○	○
サービス相互運用	○	○	○	○
異種ネットワーク間サービス相互運用	×	×	○	○
サービス発見機構の自動追加	×	×	×	○
デバイス依存情報の自動解決	×	×	×	○

表 6.1　機能比較表

表 6.1 の比較によって，本研究を用いた「u-Glue+DRIAD」の項目が関連研究と比較して，最も多くのインタオペラビリティのための機能を実現することに成功していることが明らかになった．

6.2 パフォーマンス評価実験

6.2.1 実験概要

本節では DRIAD のパフォーマンス評価実験の概要について述べる．

評価実験は Mapper-DB からのマッパの取得と，DDI-Repository からのデバイス依存情報の取得のパフォーマンスを調べる目的で行った．

それぞれ，分散データベースのノードアドレスをクライアントサイド，つまり中間ノード上でメモリにキャッシュしておく場合と，まったくキャッシュ機構を使わない場合の 2 つのパターンで計測を行った．

純粋に通信部分のみを計測するため，中間ノードで用いられている，MapperLoader というマッパをロードするための API と，デバイスに応じたドライバをロードする DriverLoader API，デバイスに応じたサービス記述をロードする DescriptionResolver API を用いて計測を行った．

図 6.1　評価実験で用いた分散データベースをシミュレートするためのコンピュータ

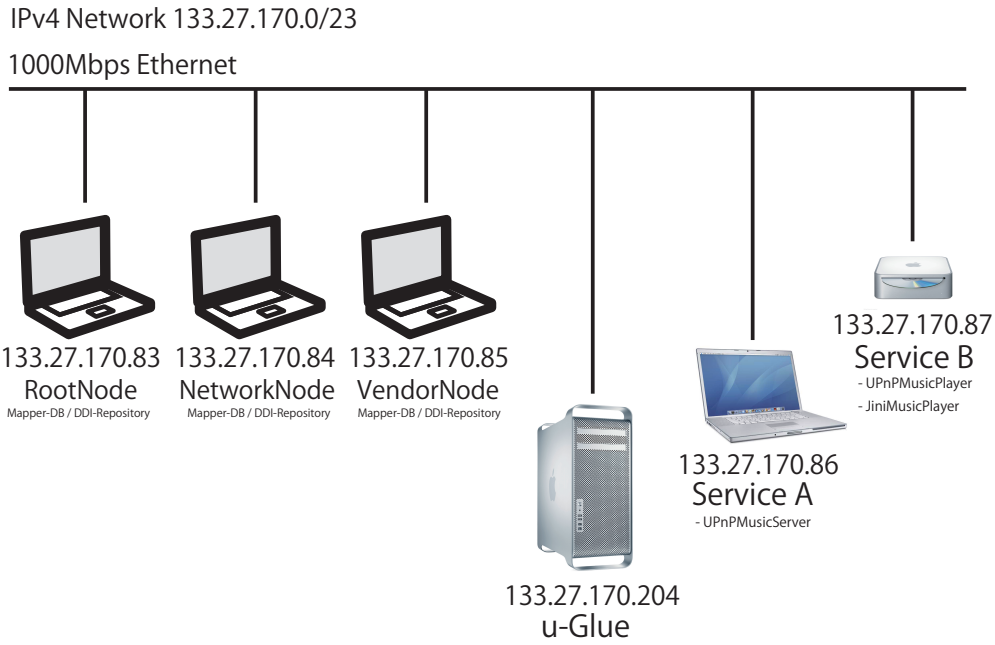

図 6.2　実験で用いたネットワークのトポロジ図

今回の実験では，Mapper-DB と DDI-Repository それぞれについて，キャッシュを用いる場合と用いない場合それぞれで，Mapper-DB からは 1 つのマッパを，DDI-Repository からは 1 つのデバイス依存情報をロードするクエリを，50 回ずつ行い，毎回のクエリ完了時間の計測を行った．

図 6.1 は実験で用いた 3 台のラップトップコンピュータである．

図 6.2 に実験で用いたネットワークのトポロジ図を示す．

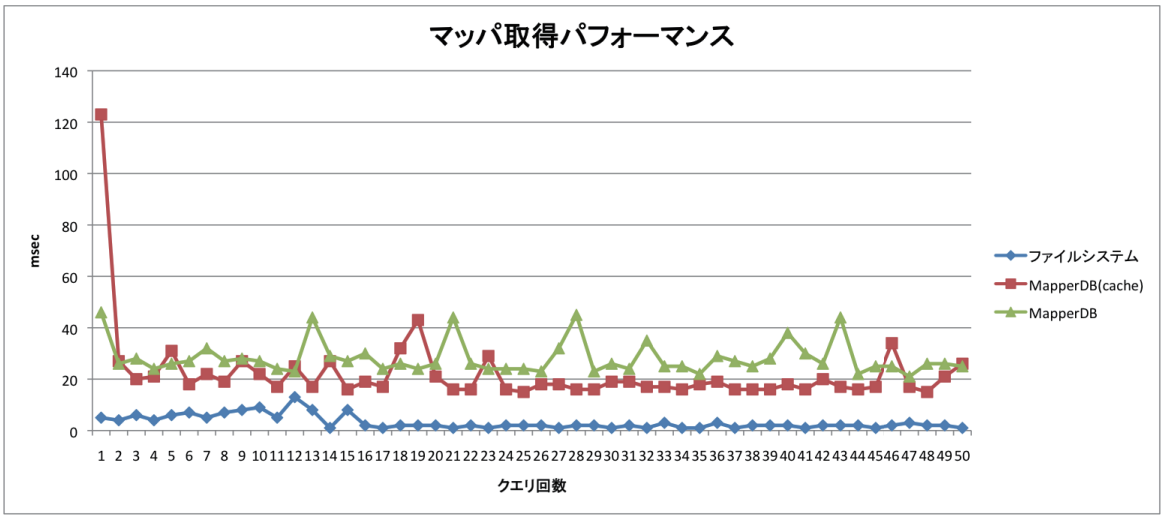

図 6.3　マッパ取得パフォーマンス

6.2.2　実験結果の考察

DRIAD のパフォーマンス評価実験結果を考察する．

Mapper-DB からのマッパ取得

図 6.3 は，50 回のクエリそれぞれでかかった時間 (単位ミリ秒) をグラフにしたものである．

グラフには，ファイルシステムからの読み込んだ場合，Mapper-DB から読み込んだ場合，クライアントサイドで Mapper-DB の各ノードのアドレスキャッシングを用いて読み込んだ場合の 3 パターンがある．

表 6.2 にクエリ実行にかかった時間の平均を示す．

ファイルシステムからの読込み	3.14msec
Mapper-DB からの読込み	28.14msec
Mapper-DB からの読込み (キャッシュあり)	22.28msec

表 6.2　Mapper 取得にかかった平均時間

最も早いのはファイルシステムからの読込みで，平均で 3 ミリ秒程度しかかかっていない．次いで早いのが，各ノードのアドレスをメモリ上にキャッシュするようにして，ルートノードとネットワークノードとの通信を減らした Mapper-DB からの取得で，最も時間がかかったのはキャッシュを使わず，毎回ルートノードとネットワークにベンダノードのアドレスを問い合わせる Mapper-DB からの取得であった．

キャッシュありの Mapper-DB からの取得は平均して約 7 倍の時間がかかり，キャッシュを用い

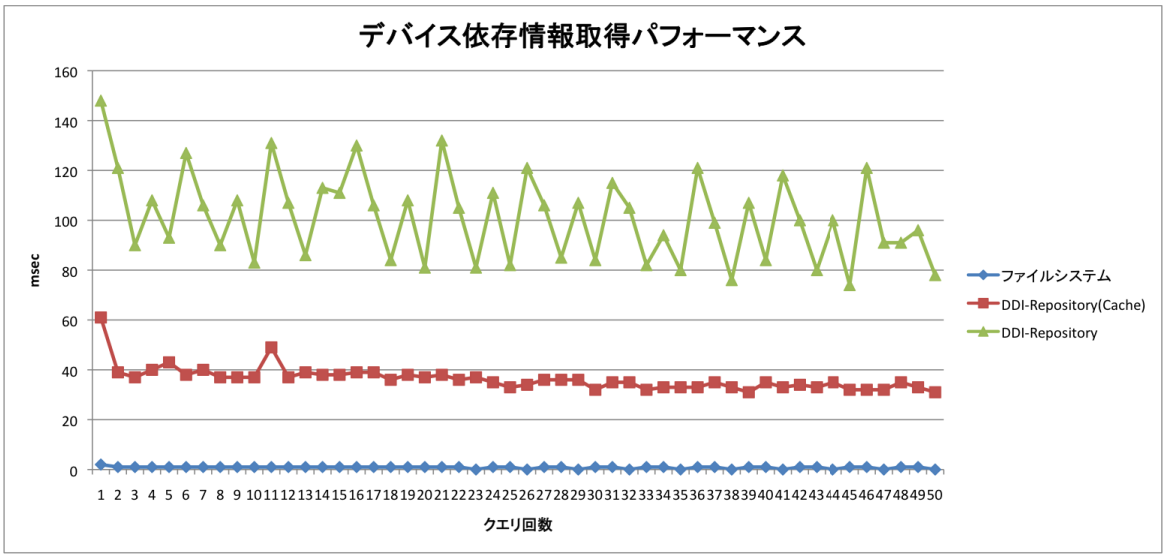

図 6.4　デバイス依存情報取得パフォーマンス

ない場合，約 9 倍の時間がかかることが評価実験の結果からわかった．この実験用ネットワークはローカルなものであり，ネットワーク通信を行うには理想的な状況なので，ルートノードやネットワークノードまでの遅延は実際はもっと大きくなるはずであり，そのような状況でキャッシュ機構は，使わない場合に比べさらなる高速化のパフォーマンスを発揮するものと思われる．

後述の DDI-Repository に比べ，キャッシュ機構の高速化の恩恵はベンダノードに対する通信が 1 ステップ大きいためだと考えられる．また，キャッシュを用いた場合，最初の処理に長い時間がかかるのは，正確に原因が特定できたわけではないが，キャッシュ機構のためのデータ構造を初回実行時に作成しているためではないかと考えられる．

DDI-Repository からのデバイス依存

図 6.4 は，50 回のクエリそれぞれでかかった時間 (単位ミリ秒) をグラフにしたものである．

グラフには，ファイルシステムからデバイス依存情報を読み込んだ場合，DDI-Repository を読み込んだ場合，クライアントサイドで DDI-Repository の各ノードのアドレスキャッシングを用いて読み来んだ場合の 3 パターンがある．

表 6.3 にクエリ実行にかかった時間の平均を示す．

ファイルシステムからの読込み	0.82msec
DDI-Repository からの読込み	101.14msec
DDI-Repository からの読込み (キャッシュあり)	36.34msec

表 6.3　デバイス依存情報取得にかかった平均時間

最も早いのはマッパ取得の実験と同じで，ファイルシステムからの読込みで，平均で 1 ミリ秒以下でクエリが完了している．次いで早いのが，各ノードのアドレスをメモリ上にキャッシュするようにしてルートノードとネットワークノードとの通信を減らした DDI-Repository からの取得で，最も時間がかかったのはキャッシュを使わないで DDI-Repository から，毎回ルートノードからアドレスを辿ってデバイス依存情報を取得するクエリであった．

マッパの評価実験に比べ，大幅にキャッシュによる高速化の効果が現れており，約 2.8 倍の高速化に成功している．

考察まとめ

ファイルシステムから直接読み込む方が圧倒的に早いが，キャッシュを使ってできるだけノードのアドレスを問い合わせないで，往復で行う通信を減らすことで大幅な高速化が行えることが本評価実験で確認された．キャッシュを使った場合，Mapper-DB からのマッパの取得と，DDI-Repository からのデバイス依存情報の解決はどちらも 40 ミリ秒以内に完了しており，中間ノードがデバイスの存在をネットワーク上に確認してから，サービスをブリッジ可能になるまでの時間としてはさして問題が起きない値ではないかと考えられる．広く分散したノードの配置によって追加の評価実験を行う必要があると思うが，アーキテクチャ的に致命的な遅さになることは考えにくい．

第 7 章

結論

本章では，評価を踏まえた上で本研究の今後の課題について述べたあと，本論文の主張のまとめを行う．

7.1 本論文のまとめ

本論文では，ユビキタス環境における，異種ネットワーク間サービス連携のためのフレームワークに対して，動的にサービス発見機構 (マッパ) をインストールする機能と，動的にネットワーク上で発見されたデバイスに対するデバイス依存情報を解決する機能 (リゾルバ) を提供するシステム DRIAD を提案した．

今後より多様化するネットワーク技術や，ネットワークプロトコル，ネットワーク到達性を持つデバイスの登場などにより異種ネットワーク間連携の技術の重要性が高まって行く中で，DRIAD は分散データベースを用いてマッパやデバイス依存情報を管理することで，開発者とエンドユーザ双方に大きなメリットをもたらすと同時に，ユビキタス環境のインタオペラビリティを向上させることができた．

7.2 課題と展望

DRIAD の今後の課題と，本研究を用いた応用研究の展望について述べる．

7.2.1 課題

本研究では，今後以下のような研究課題に取り組んで行く．

負荷分散

DRIAD の Mapper-DB および DDI-Repository は，分散データベースとして設計されているが，DNS のように辿る階層が 4 層や 5 層といったように深くならない上，2 層ノードの数が少ないため，クエリあたりの負荷の分散率が DNS に比べて劣っていると考えられる．

本論文で作成したプロトタイプでは，中間ノード用のプログラムとしてマッパ取得プログラムとリゾルバがあったが，これらに両方ともクライアントサイドで各層のノードのアドレスをキャッシュする機構を備えたところ，評価の章でも考察されたようにインターネット経由の通信が減り，よりパフォーマンスが出ることが確認された．

DNS では，クライアントサイドのキャッシュ以外にも DNS サーバ単位のキャッシングや，ネットワークの種類や，ベンダ名といったキーに対し，複数のレコードを保持する DNS ラウンドロビンのような負荷分散機構を実装することで負荷が大きいノードの負荷分散を行う仕組みを導入する必要があると考えられる．

より詳細かつ大規模な評価

DRIAD の Mapper-DB および DDI-Repository は，広く一般的に使われるようになった場合を想定すると，その分散データベースという性格上大規模でトロポジの違う場所に点在するノードに

より構成されるはずである．

本論文ではそのような大規模でトポロジの違う複数の地点にノードを配置して評価実験を行うことができなかったため，実運用に向けて今後より大規模で広く分散した環境で Mapper-DB と DDI-Repository へのクエリ速度や，ネットワークに対する負荷を測定する必要がある．

Java 言語以外の API 開発

本論文で実装された DRIAD のプロトタイプでは，クライアントサイドの API として，Java 言語の API しか実装されていない．しかし実際は DRIAD はプラットフォームに依存せずドライバやマッパなどのプログラムを中間ノード向けに提供できるように設計されているため，今後 Java 言語以外にインタオペレーションフレームワークが DRIAD を利用できるように Java 言語以外の DRIAD の API を充実させる必要がある．

7.2.2 応用研究の展望

Mapper-DB および DDI-Repository への接続性の向上

本論文で試作された DRIAD のプロトタイプは中間ノードに IPv4 によるインターネットへの接続性が備わっていることを前提としていた．

しかし，一般的に運用する中でインターネットへの接続性が確保できない中間ノードがある可能性があり，その場合 DRIAD で提案されている一切の機能は利用できなくなる．

そのような中間ノードであっても，IP 以外のネットワークインタフェースを持っているはずなので，それらのネットワークで中継できるノードを探して，Mapper-DB や，DDI-Repository への接続をブリッジしてくれる仕組みがあれば IP への接続性を持たない中間ノードであってもマッパのダウンロードや，デバイス依存情報の解決が行えると考えられる．

謝辞

本研究の機会を与えてくださり，絶えず丁寧なご指導を賜りました，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，貴重なご助言を頂きました慶應義塾大学環境情報学部教授清木康博士，慶應義塾大学環境情報学部准教授高汐一紀博士に深く感謝致します．

また慶應義塾大学徳田研究室の諸先輩方には，折に触れ貴重な御助言を頂き，また多くの議論の時間を割いて頂きました．特に，環境情報学部専任講師中澤仁博士，環境情報学部特別助教由良淳一博士，政策・メディア研究科特別研究助教伊藤昌毅博士には本研究を進めるにあたって多くのご指導と励ましを頂きました．ここに深い感謝の意を表します．

最後に，遠く離れた札幌の地からいつも応援してくれた両親，学部 2 年生の頃から長きに渡り研究生活を共に過ごした河田恭兵氏，橋爪克弥氏，生天目直哉氏，そして多くの励ましを頂いた KMSF 研究グループ，HORN 研究グループ，研究室の外で支えてくれた多くの友人に心から感謝し，謝辞と致します．

2010 年 2 月 11 日

伊藤 友隆

参考文献

[1] Osgi alliance.

[2] Upnp forum.

[3] J. Allard, V. Chinta, S. Gundala, and G.G. Richard III. Jini meets UPnP: an architecture for Jini/UPnP interoperability. In *Proceedings of the 2003 International Symposium on Applications and the Internet (SAINT 2003)*. Citeseer.

[4] SIG Bluetooth. Specification of the Bluetooth system. *Core, version*, Vol. 1, No. 2, 2004.

[5] Y.D. Bromberg and V. Issarny. INDISS: Interoperable discovery system for networked services. *Lecture notes in computer science*, Vol. 3790, p. 164, 2005.

[6] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1, 2001.

[7] Apple Inc. Bonjour.

[8] C. Lee, D. Nordstedt, and S. Helal. Enabling smart spaces with OSGi. *IEEE Pervasive Computing*, Vol. 2, No. 3, pp. 89–94, 2003.

[9] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, et al. Bringing semantics to web services: The OWL-S approach. *Lecture Notes in Computer Science*, Vol. 3387, pp. 26–42, 2005.

[10] S. McIlraith, T.C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, Vol. 16, No. 2, pp. 46–53, 2001.

[11] J. Nakazawa, H. Tokuda, W.K. Edwards, and U. Ramachandran. A bridging framework for universal interoperability in pervasive systems. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, p. 3. Citeseer, 2006.

[12] A. Sameh and R. El-Kharboutly. Modeling Jini-UPnP Bridge using Rapide ADL. In *Proceedings of the IEEE/ACS International Conference on Pervasive Services (ICPS'04), Beirut, Lebanon*, p. 237, 2004.

[13] Inc. Sun Microsystems. Jini.org.

[14] M. Weiser. The computer for the 21st century. *Scientific American*, Vol. 272, No. 3, pp. 78–89, 1995.