修士論文　　2002年度 (平成14年度)

ユビキタスコンピューティング環境に適した
間接型通信ミドルウェアの設計と実装

慶應義塾大学大学院　政策・メディア研究科
松宮 健太

修士論文要旨 2002 年度 (平成 14 年度)

近年，高い計算機能を持った多様な機器が増加している．これらの多くは携帯性や，従来の計算機には見られない入出力機能といった特長を持っている．また，これらの機器が提供するサービスを協調させることで，多様な利用方法が実現できる．しかし多数の機器が遍在する環境では，通信相手を特定的に指定し，それと一対一の通信を行う現在の通信モデルは適さない．通信相手をその機能や特長で指定できない上，通信相手が利用できなくなった場合や，逆に新しい通信相手が環境に追加された場合に対応できない．

本論文では，それぞれの機器上で動作するサービスの特長を用いて記述的に通信の宛先を指定し，その記述的宛先に任意の数のサービスを対応付けることで通信相手の変化に動的に対応する記述的間接型通信モデルを提案する．また，その実現方法を，宛先の構造，宛先の名前解決，効率的な名前解決，サービスの登録，メッセージの配送，宛先とメッセージの管理の点から比較する．

記述的間接型通信を実現する通信ミドルウェアとして SBD(Service Buffer Distribution) の設計と実装を述べる．SBD を利用することで，サービスの特長を表す文字列の配列を宛先とし，メッセージを送受信できる．メッセージはミドルウェア内に一定期間保持されるため，新たに環境に追加されたサービスもそれまで送受信されたメッセージを取得できる．また，SBD はアプリケーション層で動的にネットワークを構築し，分散して記述的な宛先とそれに対して送られたメッセージを管理することで管理を容易にする．記述的な宛先とサービスのメッセージを管理する機器の対応付けは，効率性性を実現するため，分散ハッシュテーブルを用いて行う．SBD の基本的な性能の評価として，測定結果とアプリケーションの記述例を述べる．

キーワード：
1 分散システム，2 ユビキタスコンピューティング，3 アプリケーション層ネットワーク，
4 共有空間， 4 分散ハッシュテーブル

慶應義塾大学大学院 政策・メディア研究科
松宮 健太

Abstract of Master Thesis
Academic Year 2002

Recently, heterogeneous devices with high computational power are being developed. These devices often have high portability and I/O functionallity not seen in traditional computers. By coordinating these devices, we can achieve variety of applications. However, the communication model used today does not suit such environment well. The problem is that end-points of communication needs to be specified strictly using host names or addresses, and that it is a point-to-point communication. With such communication model, services on the devices cannot be specified expressively by their functionallities, and the communication system cannot cope with dynamism of the devices with which they enter and leave the network.

In this paper, a communication model to expressively specify the end-point of communication, and to handle the dynamism of devices is proposed. Different approaches to realize the communication model is discussed, from the structure of the expressive specifiers, specifiers resolving, efficient specifier resolving, service retisteration, message passing, and specifier and message management.

Design and implementation of a middleware for such communication model, called Shared Buffer Distribution(SBD) is also proposed. By using SBD, messages to services can be sent by specifying an expressive specifier, and are temporarily stored in the middleware, allowing newly joined services to acquire messages that have been passed while they were not present. SBD forms an application layer network, on which specifiers and messages are maintained in a distributed manner. To efficiently lookup these specifiers and messages, distributed hashtable is used. The result of a basic experiment conducted on the SBD prototype implementation, and an application programming example is also explained.

Key Words:
1 Distributed System, 2 Ubiquitous Computing, 3 Application Layer Network,
4 Shared Space, 5 Distributed Hashtable

Keio University Graduate School of Media and Governance
Kenta Matsumiya

目 次

図 目 次

表 目 次

第1章　序論

1.1　本研究の背景と問題意識

近年，計算機の高性能化により，計算機能を備え，ネットワーク接続性を持った多様な機器が我々の周りに遍在し始めている．市販されているものだけでも，携帯電話やPDAなどの携帯端末，ハードディスクレコーダなどの情報家電，ゲーム機，パッド型の計算機などがある．また，研究開発段階のものとしては腕時計やアクセサリなどより小型な機器から，センサや有機ELを用いたディスプレイなど，新しい入出力機能を備えた機器がある．ネットワーク基盤の普及も著しく，ホームネットワークや無線ネットワークなどにより，これまで高速ネットワークが整備されていなかった環境にもネットワークが整備され始めている．

このような環境では，機器同士の協調を利用して様々な利用方法が実現できる．例えば，センサで取得した情報をディスプレイやスピーカを用いてユーザに知覚させたり，多様な機器がそれぞれの機能を利用し，協調してユーザにサービスを提供する利用方法が挙げられる．

上に挙げた環境で機器同士の協調を実現する場合，現在一般的に利用されている通信モデルには二つの問題がある．まず，機器の名前，つまり通信の宛先を指定する方法が特定的な点がある．現在一般的に利用されている宛先の指定方法は，DNSに登録されたホスト名，もしくはIPアドレスと，ポート番号を組み合わせたものである．このような指定方法は特定の機器を指定するのには適しているが，機器が多数存在した場合，それぞれに特定的な宛先を割り当て，管理するのは困難である．次に，通信が直接的な点がある．現在一般的に利用されている通信モデルはTCP/IPを用いたユニキャスト通信で，機器から機器の二点間をつなぐ通信モデルである．小型で移動性の高い機器が多く存在すると，環境内で利用可能な機器が頻繁に変化する．例えば通信していた機器が移動により通信不可能になる場合が考えられる．ユニキャスト通信は，このような変化に対応できない．

1.2　本研究の目的

本研究の目的は，記述性の高い宛先を用いて任意の数の機器同士で通信を行う通信モデルを提案し，それを実現する通信ミドルウェアを設計，実装することである．同通信モデルでは，機器上で動作するサービスの特長を表す，複数の属性で構成される宛先を提供し，その宛先に対して送信されたメッセージを対応付けられた任意の数のサービス

に対して配送する機能を提供する．

本研究で構築する通信ミドルウェア，SBD(Service Buffer Distribution) は，文字列の配列を用いた記述的な宛先を提供し，それを機器と対応付ける．利用できる機器が動的に変化することを考慮し，機器に対して送信されたメッセージは一定期間ミドルウェア内に保持する．また，SBD では管理者の居ない環境でも動作するため，機器同士がアプリケーション層で動的にネットワークを構築し，そのネットワークを介して宛先の検索やメッセージの送受信を行う．

1.3 本論文の構成

本論文では，2 章で近年実現されつつあるユビキタスコンピューティング環境について述べ，そのような環境における既存の通信システムの問題点を示す．3 章ではそれら問題点を解決する記述的間接型通信と，その実現方法を述べる．4 章で本研究で構築する通信ミドルウェア，SBD の設計を説明し，5 章でその実装と評価について述べる．6 章では本研究と関連する研究を説明し，本研究と比較する．7 章では今後の課題とまとめを述べる．

第2章　ユビキタスコンピューティング環境

本章ではまず，本研究で構築する通信ミドルウェアの適用環境として想定する機器やネットワークとその利用方法について述べる．次に，既存の通信システムの問題である記述力の低い宛先と直接的な通信について述べる．最後に本研究の目的である記述的間接型通信について述べる．

2.1 将来の計算機環境

本研究では，近い将来，多数の計算機能を持った機器がユーザの周りに遍在し，ネットワークに接続されたそれらの機器がユーザを支援する環境が実現することを想定する．計算機能を備えた機器が遍在する環境を，本論文ではユビキタスコンピューティング環境と呼ぶ．このような想定の妥当性を示すために，計算機能を備えた機器の多様化とネットワークの整備について近年の動向を説明する．

2.1.1 機器の多様化

近年，計算機の価格性能比の低下が著しい．これに伴い，携帯電話，家電，ゲーム機などの機器が高度な計算処理能力を持ち始めている．計算機の性能は今後も向上することが予想される．また，大容量のメモリや無線ネットワーク機能を CPU と同一のチップ上に載せる技術も開発されており，今後ますます多様な機器が計算機化されることが予想できる．図 2.1 に現在高度な計算処理能力を持ち始めている多様な機器の例を示す．

図 2.1: 近年登場している多様な機器

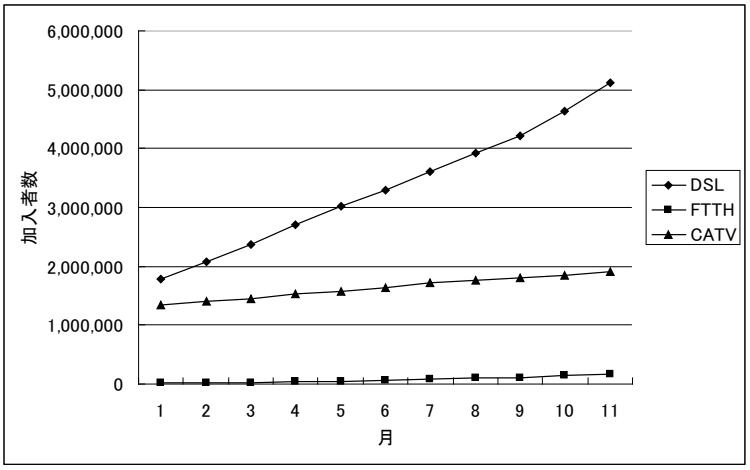


図 2.2: 平成 14 年のホームネットワーク加入者数

計算処理能力に加え，新たな入出力機能を備えた機器も登場している．入力機器で注目されているのは，各種のセンサである．特に位置情報を取得するセンサの研究開発は盛んで，GPS のように既に普及しているものから，RF や超音波を利用した室内向けのセンサや，それを統合するシステムも登場してきている [1]．また，U.C.Berkeley の Smart Dust[2] など，超小型のセンサも登場している．このようなセンサでは，壁に塗り込む設置方法も考えられている．ウェブカメラのように映像を取得する機器も多く登場している．

出力機器も，新たなものが登場している．特に注目されているのは，有機 EL を用いたディスプレイなどである．有機 EL を用いると，小型，軽量かつ軟らかなディスプレイが実現できる．有機 EL は携帯電話のディスプレイなどにも利用されている．このようなディスプレイを利用すると，服などさまざまなものを出力機器として利用できる．

また，慶應義塾大学徳田研究室では SmartFurniture と呼ぶ機器を研究開発している [3]．SmartFurniture は計算機やセンサを埋めこんで，空間を知的化するための機器だが，そのライトタイプでは，情報を光の色にエンコードして表示する．例えば室内の混雑状況を表す際には赤い色で表示する．このように，光を情報表示に利用する機器もある．

これら機器の特長は，小型かつ安価なことと，新しい入出力を備えることである．小型なことでそれぞれの機器の移動性が高くなり，特定の環境で利用できる機器は動的に変化する．また新しい入出力を利用して，多様な協調動作を用いた利用方法が実現される．

2.1.2 ネットワークの整備

近年，コンピュータネットワークの社会的な重要性が増してきている．あらゆる情報がネットワーク経由で取得できるだけでなく，ショッピングや金融取り引きなどのサービスもネットワークを介して行われている．それに伴い，ネットワークが広範に整備されてきている．特に注目されているのはホームネットワークと無線ネットワークである．

多くの家庭には，これまで電話線を利用した低速なネットワークしか整備されていなかった．しかし近年，xDSL や CATV，光ファイバなど数十 Mbps のスループットが得られるネットワークが整備されてきている．図 2.2 にこれらホームネットワークの平成 14 年における加入者の推移を示す．このため，ホームネットワークは本研究の適用環境として有力なものの一つである．ホームネットワークで考慮すべきことは，ユーザがコンピュータやネットワークの管理に不馴れな点である．研究機関や企業など，これまでネットワークが整備されてきた環境には，ネットワークの管理者が存在していた．そのためソフトウェアの管理コストに対する制約は比較的少なかったが，ホームネットワークではそのコストを大幅に低下させる必要がある．

無線ネットワークの普及も著しい．現在特に 802.11b の無線通信規格が急速に普及している．今後より速度の速い 802.11a や 802.11g なども整備されることが予想される．無線ネットワークが普及することで，携帯端末など移動性の高い機器がネットワークに接続することが増える．これによりネットワークに対する機器の出入りという観点から，ネットワークの構成はより動的になる．ただ，無線ネットワークには現状よりも強力なセキュリティ機構が必要とされており，これまでのような速度で普及が進むかはセキュリティ技術の向上によるところが大きい．

これら以外にも，Bluetooth，IrDA，電力線搬送などのネットワークが登場してきている．これらのネットワーク上で利用されるプロトコルも様々であるが，最も多く利用されているのは IP である．将来の通信プロトコルは機器の膨大な数に対応する必要があるが，IPv6 が整備されると，それらの機器に広域のアドレスが割り当てられる．このように今後も通信プロトコルとして IP が重要になることが予想されるため，本研究では下位の通信プロトコルとして IP を想定する．

2.2 想定される機器の利用方法

多数の機器がネットワークに接続された場合，それぞれの機器を単独で利用する従来の利用方法に加えて，複数の機器を協調させる新たな利用方法が可能となる．機器の協調を用いた利用方法としてまず考えられるのが，情報の共有と相互操作である．

情報の共有とは，一つの機器が保持，取得した情報を他の機器が利用することである．デジタルカメラの画像を壁掛ディスプレイに表示したり，DVD 映像の音声をスピーカに再生する場合などが考えられる．実際にテレビの映像を録画，保持するハードディスクレコーダなどの機器が普及しはじめている．また，センサが取得した情報を別の機器で利用する場合も情報共有の一つとして捉えられる．

相互操作とは，一つの機器から他の機器を操作することである．携帯端末から家電を操作したり，家電同士が互いを操作することで協調動作を実現する場合などが考えられる．実際にネットワークを介して操作可能なエアコンなどが登場している．また，HAVi[4] や UPnP[5] などのミドルウェアはこの相互操作を実現することを目的としており，それぞれの規格もまとまりつつある．家庭内ルータなどで，UPnP に対応したものも登場している．

以下に将来の計算機環境で実現することが予想できるその他の利用方法をいくつか挙げる．

情報の遍在的な視覚化

環境に遍在する機器を用いて，情報の視覚化が可能になる．災害や侵入者などの異常事態，隣の部屋に人が居るか，外で雨が降っているか，ガス洩れ，ネットワークのトラフィックなど，有用な情報をユーザに示せる．他にもニュースや天気．本やビデオの返却期限，家族の帰宅予定時間，ミーティングの開始，終了予定時間などが考えられる．

これらの情報は小型ディスプレイやSmartFurniture のような機器を用いることで，様々な形態で表示できる．例えば，天気の情報を洗濯機や玄関のドアに表示するなど，関連性の高い機器に表示することで，より利便性の高い表示方法も実現できる．また，光情報のような単純な情報として表示することで，ユーザはより直感的に情報を認知できる．例えば外で雨が降っている場合は部屋のランプが水色に光るなどの表示方法が考えられる．

生活の支援

情報家電やセンサを用いて，ユーザの生活を支援する利用方法が考えられる．例えば，電子レンジで食べ物を温めると，次はテレビや新聞を見ることを予測して，周辺のディスプレイに番組表やニュースのダイジェストを表示したり，夜にお風呂を沸かすと，次は寝室に行くことを予測して寝室の暖房や冷房が作動するといった利用方法が考えられる．

2.3　一般に利用されている通信システムの問題

情報共有，相互操作，また各利用方法を通して，協調を伴う利用方法には機器間の通信を実現する必要がある．将来の計算機環境では，ネットワーク内の機器の数が動的に増えることで新たなサービスが利用可能になる．逆に一部の機器が故障やネットワークから離れることで特定のサービスが利用できなくなる．通信システムはこのような機器の動的な変化に対応する必要がある．それには，変化があった場合でも通信システム自体が停止することなく新しい環境に適した通信を提供できることが望ましい．また，機器が多数存在する環境では，一つの機器を一意に指定して行う通信だけでなく，特定の特長を持つ機器を柔軟に指定して行う通信も必要である．以下にこれらの点における既存の通信モデルの問題を述べる．

2.3.1　記述力の低い宛先

現在一般的に利用されている宛先の指定方法は，IP アドレスもしくはホスト名とポート番号を組み合わせた宛先である．IP アドレスはインターネット上のホストを一意に指定でき，DNS によって管理されるホスト名は，IP アドレスに対応される．ポート番号は

それぞれのホスト上で動作するサービスを特定するのに用いられる．

このような宛先指定方法の問題は，宛先自体が多くの情報を持てない点である．機器が増加した場合，このように情報量の少ない名前で異なるサービスを表すのは困難になる．現在，特定のサービスがどの機器で動作しているかはホスト名で記述できる．例えば，カメラサービスがあった場合，機器には camera1.sfc.keio.ac.jp のような名前が割りふれる．カメラと全く関係の無い名前が付いている場合は，ユーザがカメラサービスを提供する機器の名前を把握している．現状のようにサービスの種類が少ない環境では，これらの方法でも管理が可能だが，サービスの種類が増えるに従って，管理は困難になる．

2.3.2 直接的な通信

現在一般的に利用されている通信はユニキャスト通信である．送信されるメッセージは送信元から一つの送信先へと送られる．ユニキャスト通信の問題は，一つの相手にしかメッセージを送れない点である．機器が多数存在し，それらが協調する環境では，複数の相手にメッセージを送る方法や，複数の相手のうちどれかにメッセージを送る方法が必要となる．前者はマルチキャスト通信，後者はエニーキャスト通信と呼ばれる．

ユニキャスト通信ではメッセージを送るべき相手が動的に増加した場合に対応できない．アプリケーション側で対応方法を定義しておく必要がある．通信システムが同じサービスを提供する全ての機器にメッセージを送る機能を備えていれば，通信システムで対応できる．

また，通信相手が故障した場合や，それがネットワークから離れた場合にそのサービスを継続して利用できない．このような場合，現在の通信システムではアプリケーション側で代替の通信相手を検索する方法を定義しておく必要がある．通信システムが類似したサービスを提供する機器のどれかにメッセージを送る機能を備えていれば，通信システムで対応できる．

既存のインターネットにもマルチキャストとエニーキャストのための機構はある．これらの機構では，特定の機器を表さない，特別なアドレスに対してメッセージを送り，それらを具体的な機器と対応付けることで複数の通信相手に対応している．しかし，広域でこれを行う際，遠隔のネットワークに存在する通信相手に効率的にメッセージを送ることが困難である．

2.4 必要となる通信モデル

上記の問題を解決するため，ユビキタスコンピューティング環境では環境の変化に動的に対応し，通信相手を柔軟に指定できる通信システムが必要となる．環境の変化に動的に対応するには，通信の宛先として特定の機器を指定しない方法がある．本論文ではそのような通信を**間接型通信**と呼ぶ．また，通信相手を柔軟に指定するには，記述力の高い宛先を用いる方法がある．本論文ではそのような宛先を用いる通信を**記述的**と呼ぶ．また，これらの特長をどちらも備えた通信を**記述的間接型通信**と呼ぶ．

特定の通信相手を一意に指定し，その相手だけにメッセージを送る場合は，その識別子にユニキャスト通信で送る方法が適している．本研究の目的は，アプリケーションが，相手や相手のグループを特定する識別子を把握していない場合でも，また相手の数を考慮せずに通信を行える通信システムの構築である．

2.5 本章のまとめ

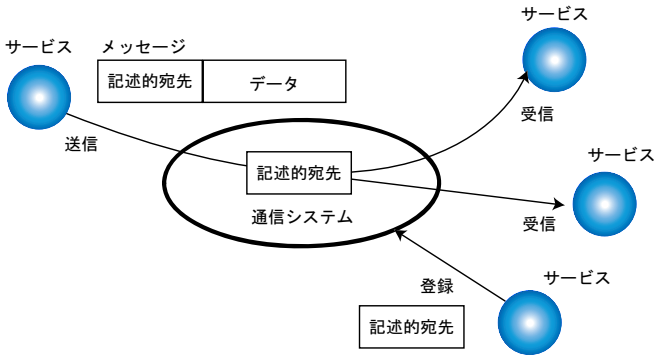
本章では，本研究で想定する将来の計算機環境において機器が多様化し，ネットワークがより広範な範囲で整備されることを述べた．次に，そのような環境では，情報の共有や機器同士の相互操作など，機器の協調を利用した利用方法が実現されることを述べた．最後に，そのような利用方法を支援する通信システムとして，現在の通信システムの問題点を指摘し，本研究の目的である記述的間接型通信の必要性を述べた．

第3章　記述的間接型通信

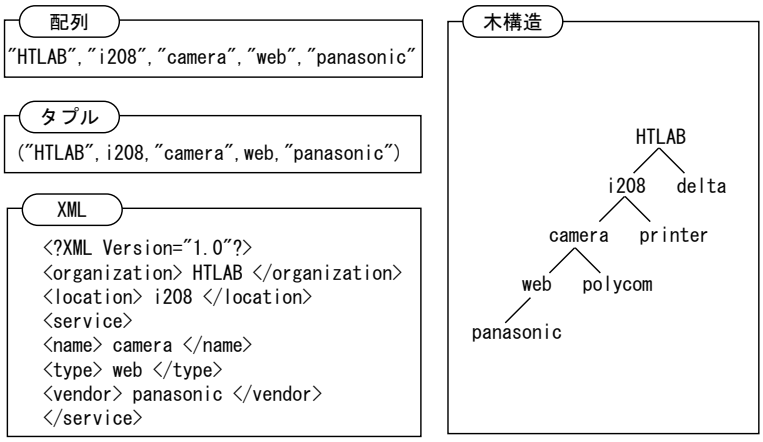
本章ではまず，本研究で実現する記述的間接型通信の概要を述べる．次に，記述的宛先と間接型通信に必要な機能を挙げ，それぞれの実現方法を比較，考察する．最後に本研究で利用する実現方法を述べる．

3.1 概要

記述的間接型通信は，一つのメッセージ送信者から，記述力の高い宛先を用いて任意の数の受信者にメッセージを送る通信モデルである．例えば，特定の部屋にある全てのライトにメッセージを送ったり，ウェブから画像が見れるウェブカメラ全てにメッセージを送ったりできる．記述的間接型通信を実現する通信システムは，アプリケーションに対して記述力の高い宛先指定方法を提供する．また，メッセージを送るべき相手の数やその可用性の動的な変化に対応するため，メッセージ受信者に間接的にメッセージを送信する機能を提供する．図 3.1 の (a) に記述的間接型通信の概観を示す．



(a) 記述的間接型通信



(b) 記述的宛先

図 3.1: 記述的間接型通信モデル

記述力の高い宛先を提供する場合，その構造を決定する必要がある．図 3.1 の (b) は記述的宛先が取り得る構造の例を示す．その構造として，単純な配列，タプル，XML，木構造などが考えられる．それぞれの構造の柔軟性や検索効率については後述する．また，通信システムが記述的宛先を利用してメッセージの配送を行うには，それを具体的な機器のアドレスに変換する名前解決機能が必要となる．広域でこのような機能を提供する場合，効率的な名前解決手法も併せて必要となる．

それぞれのサービスは，システム内で管理される記述的宛先と対応付けられる必要が

ある．これは，メッセージ送信者が記述性の高い宛先を用いて具体的なサービスにメッセージを送るためである．また，メッセージ受信者の数や可用性が変化した場合に通信システム自体が動的に対応するため，複数のサービスを同じ記述的宛先に対応付ける機能が必要である．この際，サービスが記述的宛先を登録するための機能を提供する必要がある．また通信システムは，記述的宛先に送られたメッセージを該当する機器に配送する機能も提供する必要がある．この際，宛先やメッセージを通信システムのどこで管理するか考慮する必要もある．まとめると，以下の点について異なる実現方法を比較，考察する必要がある．

- 宛先の構造
- 宛先の名前解決
- 宛先の効率的な検索
- サービスの登録方法
- メッセージの配送方法
- メッセージの管理方法

上半分の項目は記述的宛先に関係し，下半分の項目は間接型通信に関係する．以下の節では，それぞれの項目について実現手法を述べ，比較する．それぞれの点を本研究でどう実現するかは，本章の後半で述べる．

3.2 記述的宛先

3.2.1 宛先の構造

記述性の高い宛先を提供するには，その構造を考慮する必要がある．ここで記述性が高いとは，宛先が複数の属性を持つことを指す．DNS が提供するホスト名のように，ユーザが一つの属性，具体的には名前もしくは識別子しか定義できない宛先では，記述性の高い宛先を提供するのは困難である．また，それぞれの属性に型を与えるかどうか考慮する必要がある．型を与えた場合，似た内容を表す属性もその型で区別できる．従って記述がより柔軟になるが，宛先の名前解決や検索が複雑になる．宛先の構造として，配列，タプル，XML，木構造が考えられる．図 3.1 の (b) にこれらの構造を持った記述的宛先の例を示した．

配列

宛先の構造として，各属性の集合をそのまま扱う方法を配列と呼ぶ．この場合，各属性には型を与えない．この方法の利点は，単純さである．逆にこの方法の欠点は，検索効率や柔軟性が他の方法と比べて低いことである．

タプル

各属性が型と順番を持った配列をタプルと呼ぶ．この方法の利点は，それぞれの属性が型を持つため，柔軟な宛先を比較的単純に記述できることである．この方法の欠点は，属性が型を持つため，名前解決や検索が複雑になることである．Jini[6] ではサービスの記述としてタプルを利用している．

XML

XML(eXtensible Markup Language) は，データに柔軟な構造を与えるためのメタ言語である．属性の数や順番を定義したものをスキーマと呼ぶ．この方法の利点は，柔軟にデータ構造が定義でき，かつスキーマを利用することでデータの正当性を検査できることである．この方法の欠点は，宛先の構造が複雑になることと，通信者同士でデータの定義を共有する必要があることである．SDS[7] では，サービスの記述として XML を用いている．

木構造

宛先の構造として，木構造を用いる方法が考えられる．この方法の利点は，枝を降りるごとに検索範囲が絞られるため，検索効率が高いことである．逆にこの方法の欠点は，システム側が提供する制約の強い木構造を用いると，それ以外に分類できる宛先を表すのが困難になる点である．例えば管理ドメインの情報がノードとなった木構造では，どの組織が管理しているのか曖昧な機器などの宛先を表すのが困難である．DNS が割り当てる名前は木構造を用いている．

3.2.2 宛先の名前解決

通信システムは，記述的宛先を実際の機器のアドレスに変換する必要がある．そのため，サービスを表す記述的宛先と機器のアドレスを対応付けるデータ構造を管理し，それを用いてメッセージの記述的宛先を機器のアドレスに変換する必要がある．変換を行う方法として，記述的宛先に含まれる属性全てが一致する機器のアドレスに変換する完全一致と，複数の属性の一部が一致する機器のアドレスもにも返還する部分一致がある．

完全一致

記述的宛先を機器のアドレスと対応付ける際に，全ての属性が一致する機器のアドレスを対応付ける方法を完全一致と呼ぶ．この方法の利点は，宛先が明確に指定できる点である．そのため，意図しないサービスにメッセージが配送されない．また，比較方法として単純なため，実装が容易になる．この方法の欠点は，曖昧な宛先指定ができない点である．

部分一致

記述的宛先を機器のアドレスと対応付ける際に，一部の属性が一致する機器のアドレスに対応付ける方法を部分一致と呼ぶ．この方法の利点は，宛先の曖昧な指定が可能な点である．また，属性ごとに段階的に絞りこむこともできる．この方法の欠点は，実装が複雑になる点である．多くの場合，記述的宛先をその構造を保ったまま利用する必要が出てくる．

3.2.3 宛先の効率的な検索

通信システムの広域での運用や機器数の増加を考慮すると，宛先の検索を効率的に行う手法が必要となる．効率性を実現するには，まず，宛先の名前空間を複数のより小さい名前空間に分割する必要がある．個々の名前空間を小さくし，それぞれを異なるサーバに分配することでスケーラビリティが上がる．この際，複数のサーバに名前を分散する方法として，名前に情報を埋めこむ分配方法と，ハッシュ関数を利用した分配方法がある．

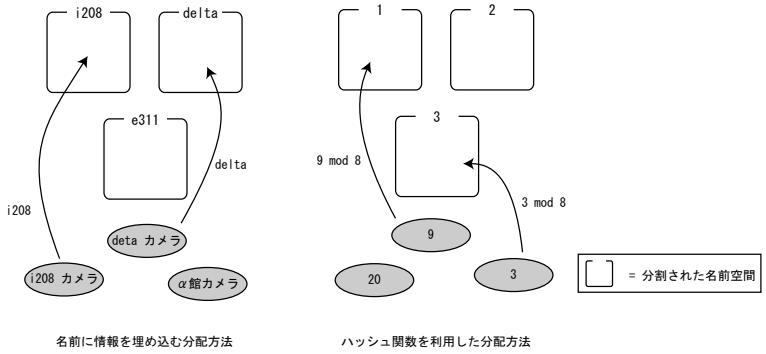


図 3.2: 宛先の効率的な検索

宛先に情報を埋めこむ分配方法

宛先に，分配を可能にするための情報を埋めこむ方法を考える．このような情報として，管理組織や位置の情報が利用できる．図 3.2 の左にこの方法を図示した．この方法の利点は，名前の構造を保持したまま分配できる点である．この方法の欠点は，名前に必須属性ができることである．最初からこのような情報を含む名前に対して有効であると言える．

ハッシュ関数を利用した分配方法

ハッシュ関数を用いて名前を分配する方法を考える．この場合も名前空間を複数のサーバで管理するが，ハッシュ関数を用いて宛先を固定長のビット列に変換し，各サーバに

分配する．図 3.2 の右にこの方法を図示した．サーバを N 個，記述的宛先を a とした場合，$Hash(a)$ mod N 番目のサーバに宛先を割り当てることで均等に分配できる．この方法の利点は，名前に新たな情報を加えること無く分配できることである．この方法の欠点は，宛先を固定長に変換するため，その記述性が失われる可能性があることである．

3.3 間接型通信

3.3.1 サービスの登録方法

メッセージを一つのサービスから別のサービスに送信する場合，サービスと記述的宛先を対応付ける必要がある．特定のサービスに対応する宛先は，通信システムに対して登録される．この際，登録されたサービスの宛先をどのように保持するか考慮する必要がある．宛先の保持方法としてハードステートで行う方法とソフトステートで行う方法が考えられる．

ハードステート

サービスの宛先が通信システム内で常に保持され，明示的に削除されるまで削除されない方法をハードステートと呼ぶ．この方法の利点は，通信システムが常にサービスの宛先を保持しているため，サービスが動作していなくてもその宛先が取得できることである．そのため，システム外部から頻繁に宛先を取得する場合に適している．しかし，サービスが利用できなくなった場合には，明示的にその宛先を削除する必要がある．そのため，動的適応性は低い．DNS はサービスと宛先の対応付けを静的なファイルで管理するため，ハードステートを用いていると言える．

ソフトステート

サービスの宛先が通信システム内で一時的に保持され，一定期間の後に削除される方法をソフトステートと呼ぶ．一般的には，サービスが動作している間は，その宛先の更新が行われる．この方法を用いると，サービスが利用できなくなった場合に自動的に情報が削除されるため，通信システムは動的に実際のサービスの状態に適応する．この方法の欠点は，更新のためのメッセージが正しく管理される必要があることと，サービスが利用できない場合にはその情報を取得できないことである．Jini[6] などではこの方法でサービスを管理している．

3.3.2 メッセージの配送方法

送信者が受信者に対してメッセージを送ったときに，通信システムはそれを受信者に配送する機能を提供する必要がある．配送の方法としてフォワーディング行う方法と共

有空間を用いる方法がある．図 3.3 にそれぞれの配送方法を示す．

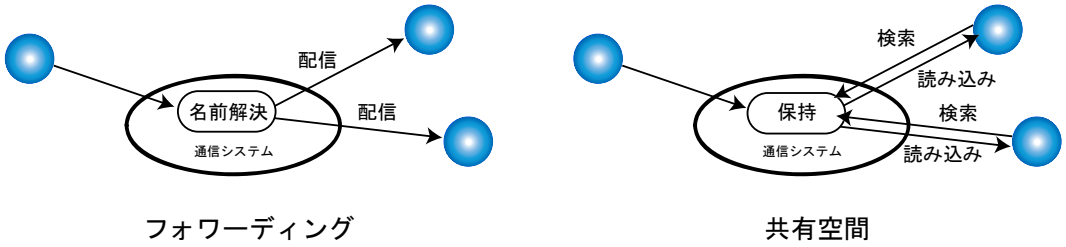


図 3.3: メッセージの配送方法

フォワーディング

IP ネットワークでのパケット配送のように，通信システムに渡されたメッセージを適切なサービスに逐次配送する方法をフォワーディングと呼ぶ．この場合，通信システムはまず宛先の名前解決を行い，配送先を決定する．配送先が決定した後，その機器に対してメッセージを送る．この方法の利点は，メッセージが逐次受信者に送られるので，遅延が少ない点である．この方法の欠点は，次に挙げる共有空間を用いた方法と異なり，メッセージを通信システム内に保持できない点である．これにより，受信者が短期間故障した場合などにメッセージが失われる．

共有空間

通信システムに渡されたメッセージが保持され，受信者が明示的にそれを読み込むことでメッセージを配送する方法を共有空間を用いた方法と呼ぶ．この方法の利点は，通信システム内でメッセージを保持するため，受信者が短期的な故障から回復した後や新たな受信者が通信に参加した際に，参加していなかった間にやりとりされたメッセージを取得できることである．これにより，故障からの復旧などが可能となる．この方法の欠点は，メッセージ受信者がメッセージがあるかどうか一定間隔で検査するため，遅延が増大する可能性があることである．しかし，実際に利用する際には，メッセージが通信システムに渡された際，そのメッセージを待っているサービスに即座に通知する機能が提供される場合も多い．

共有空間を用いて通信を行う場合，メッセージ自体をハードステートで保持するかソフトステートで保持するか決定する必要がある．メッセージをハードステートで保持した場合，送信者が送ったメッセージを受信者はいつでも取得でき，耐故障性などが向上する．ソフトステートで保持した場合，メッセージはシステムに短期間保持される．Java Spaces[8] ではソフトステートを用いている．

3.3.3　メッセージの管理方法

通信システムは，宛先もしくはメッセージ自体をシステム内で保持する必要がある．これらの情報を保持する方法として，集中管理で行う方法，IP ルータ型分散管理で行う方法，アプリケーションルータ型分散管理で行う方法が考えられる．図 3.4 にそれぞれの管理方法を示す．

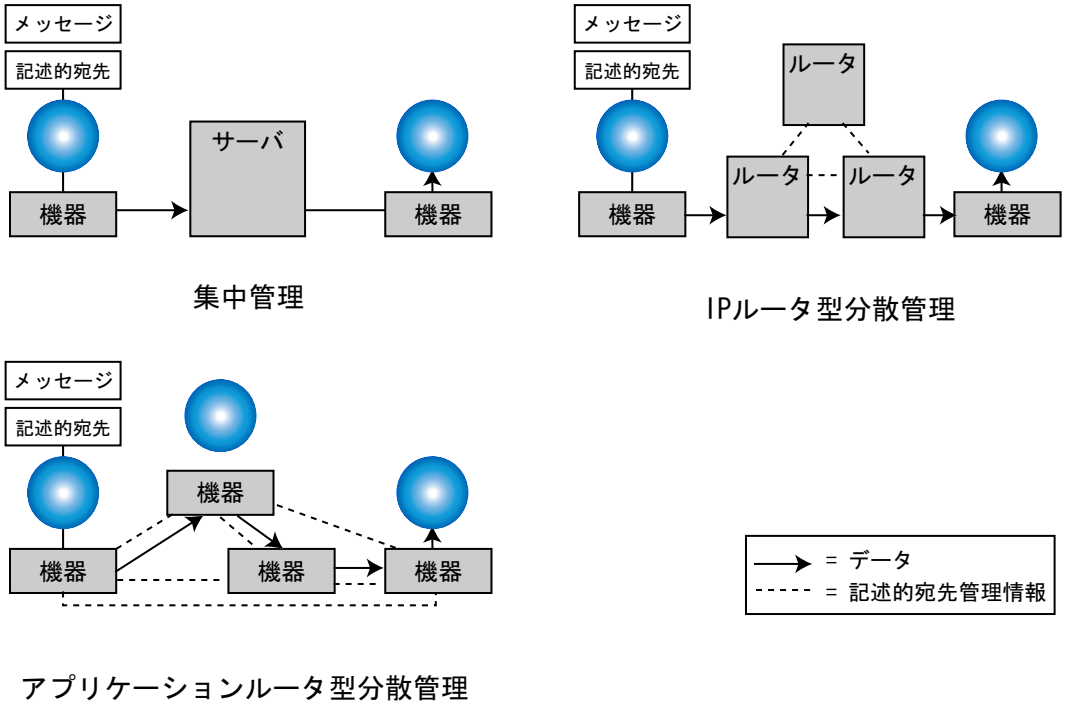


図 3.4: メッセージの管理方法

集中管理

一つのサーバで全ての宛先やメッセージを管理する方法を集中管理と呼ぶ．この方法の利点は，次に見る分散型とは異なり，複数のサーバ間で保持する情報の整合性を保つ必要が無いことである．しかし，保持する情報の増加に従ってアプリケーションからの要求処理や宛先の名前解決によるサーバの負荷が増加するため，スケーラビリティが低い．また，この方法ではサーバを他の機器とは別に保守管理する必要がある．

IP ルータ型分散管理

インターネットのように複数のルータで分散して宛先やメッセージを管理する方法を IP ルータ型分散管理と呼ぶ．この方法では，複数のサーバが宛先やメッセージを分散して管理する．これらのサーバは，アプリケーションやサービスから要求を受けつけ，名前

解決やメッセージの配送を行う．一つのサーバで宛先やメッセージを管理する場合に比べて負荷が分散され，宛先の検索効率も上がるため，スケーラビリティが向上する．しかし，宛先やメッセージの整合性を保つための機構が必要となる他，サーバを他の機器とは別に保守管理する必要がある．

アプリケーションルータ型分散管理

サービスが動作している機器自体が宛先やメッセージを管理する方法をアプリケーションルータ型分散管理と呼ぶ．この方法では，各機器が分散して宛先やメッセージを管理する．この方法の利点は，ルータが機器上で動作するため，サーバやルータを別に管理する必要が無いことである．また，スケーラビリティや耐故障性も確保できる．しかし，実装が複雑になる他，機器に高い処理能力が必要となる．

3.4 本研究での実現方法

ここまで，記述的宛先と間接型通信の実現方法を比較した．本節では，本研究での実現方法を述べる．2 章で述べたように，本研究では近い将来，高い計算能力と新しい入出力を備えた多数の機器がユーザの周辺に遍在し，それらが広範に整備されたネットワークを利用して協調動作することを想定している．また，それらの機器の一部は小型かつ安価で，動的にネットワークに出入りし，また頻繁に故障，紛失することが予想できる．そのような環境では以下の機能要件がある．

単純性 多数の機器を利用して複雑な協調動作を実現するアプリケーションは，記述的間接型通信システムが提供する機能を多用することが予想される．通信システムは，そのようなアプリケーションのプログラマを支援するため，単純なプログラミング方法を提供する必要がある．単純性を実現するため，本研究ではサービスの記述的宛先として，**配列**を用いる．従って，多数のサービスごとに XML ファイルを作成したり，木構造やタプルを用いた名前をつける必要が無い．また，同じく単純性のため，本研究では名前の解決方法として**完全一致**を用いる．

動的な環境への適応 将来の計算機環境では，機器の移動性が高いため，ユーザ周辺の環境に対して機器が動的に追加，削除される．新たに追加された機器や，短期的な故障から復帰した機器は，通信に参加していなかった間のメッセージを取得することで短期間で通信に参加できる．メッセージ送信者は，それらの機器のためにメッセージを送り直す必要が無い．本研究では，**共有空間**によってメッセージの配送を行うことでこれを実現する．また，機器は環境に参加したことを**ソフトステート**で通知する．これにより，サービスが環境に存在する間はそれにメッセージを送ることができ，機器が環境から離れると自動的にその記述的宛先は破棄されるので，どの機器やサービスが利用可能か明示的に管理する必要が無い．本研究では，メッセージを保持する際にもソフトステートを用いる．

管理の容易さ 将来の計算機環境における通信システムは，ホームネットワークなど，利用者が多様な環境でも利用するため，管理者が管理しなくても動作する必要がある．このような要件は，サーバやルータの管理が独立に必要な通信システムでは満たすのが難しい．従って，本研究では**アプリケーションルータ型分散管理**によって記述的宛先やメッセージを管理する．通信システムは，初期化作業や機器の追加と削除などを動的に行う．

効率性 多数の機器が存在する環境では，サービス間での通信量が増加することが予想できるため，効率的な名前解決方法が必要となる．本研究では，**ハッシュ関数を利用した分配**の利用によりこれを実現する．記述的宛先に効率性のための情報を埋めこまないのは，単純性を保持するためである．必須属性が生じると，単純なアプリケーションを構築する際に記述的宛先が必要以上に複雑になる．

本研究で利用する実現方法は，互いに親和性も高い．例えば，型や複雑な構造を持った記述的宛先は，ハッシュテーブルで利用する際に変換を考慮する必要があったり，構造が失われたりする．しかし単純な配列を用いた記述的宛先は，それらを考慮する必要が無く，ハッシュテーブルを利用しやすい．完全一致にも同じことが当てはまる．ハッシュテーブルを用いた部分一致の実現は困難である．また，ソフトステートを用いることで，共有空間に保持されたメッセージも動的に扱われる．

3.5 本章のまとめ

本章では，記述的間接型通信の概要を述べ，その実現方法を比較した．記述的間接型通信システムは，サービスがその記述的宛先を通信システムに登録する機能を提供し，その記述的宛先に送られたメッセージを適切な機器に配送する必要がある．記述的宛先に関しては，宛先の構造，宛先の名前解決，宛先の効率的な名前解決について複数の実現方法を比較した．また間接型通信に関しては，サービスの登録，メッセージの配送，メッセージの管理方法について複数の実現方法の利点と欠点を述べた．将来の計算機環境で利用される通信システムには，単純性，動的適応性，管理の容易さ，そして効率性が必要となるため，本研究では単純な配列を用いて記述的宛先を実現し，機器同士がアプリケーション層で動的に分散ハッシュテーブルを構築し，その宛先に対して送られたメッセージを保持する共有空間を提供する．

第4章　SBDの設計

本章ではまず，記述的間接型通信を実現する通信ミドルウェアの概要と構成を述べる．次に，それを構成するメッセージ管理モジュールと分散ハッシュテーブルモジュールの機能を挙げ，その設計を説明する．

4.1 設計概要

本研究で構築するSBD(Service Buffer Distribution)は，記述的宛先を用いたメッセージの送信やサービスの登録と，共有空間を用いたメッセージ配送を提供する通信ミドルウェアである．サービスが記述的宛先を登録すると，そのサービスに宛てられたメッセージを保持するための共有空間が作成される．メッセージ送信者がサービスに対して送信したメッセージはこの共有空間に一時的に保持され，サービスがこれを読み込むことで受信される．共有空間は本ミドルウェアを実行している機器の一つに作成される．この際，どの機器に共有空間を作成するかは，ハッシュ関数を用いて割り当てる．複数のサービスは同一の記述的宛先を登録でき，任意の数のサービスが特定の記述的宛先に宛てられたメッセージを受信できる．

記述的な宛先は単純な文字列の配列として提供する．本ミドルウェア内では，ハッシュ関数を用いてこれをバイト列に変換し，メッセージのヘッダに付加する．それぞれの機器にも同様のバイト列を用いた識別子を用意し，これらを対応付けることでどの機器がサービスのメッセージを保持するか決定する．

記述的宛先を登録したサービスは，必ず一つの機器と対応付けられる．本論文ではこの機器をサービスに対する**中間機器**と呼ぶ．サービスは一つの中間機器に対応付けられるが，中間機器は任意の数のサービスに宛てられたメッセージを保持する．耐故障性や負荷分散を向上させるため，複数の中間機器を対応付け，メッセージを複製する方法も考えられるが，本論文では最も基本的な設計としてサービスに対するメッセージは一つの中間機器で管理する設計を述べる．

個々の機器上で動作する本ミドルウェアは，アプリケーション層で分散ハッシュテーブルを提供するネットワークを構築する．サービスとその中間機器との対応付けはこの分散ハッシュテーブルを用いて行う．分散ハッシュテーブルは，検索鍵と値を複数の機器で分散して管理する構造である．それぞれの機器は全体の検索鍵と値の一部を管理し，検索アルゴリズムを用いて効率的に値を検索できる．本ミドルウェアが構築する分散ハッシュテーブルは，サービスの記述的宛先を検索鍵とし，その宛先の中間機器のアドレスを値として対応付ける．

4.2 システム構成

本ミドルウェアは二つのモジュールから構成される．アプリケーションやサービスにプログラミングインタフェースを提供し，メッセージを保持するメッセージ管理モジュールと，分散ハッシュテーブルを構築し，サービスの記述的宛先と中間機器との対応付けを行う分散ハッシュテーブルモジュールである．これらのモジュールは必ず一組で動作する．個々の機器上で動作するモジュールの組は，対照的なアプリケーション層ネットワークを構築する．つまり，検索要求の発行と検索要求の処理の両方を行う．以下にそれぞれのモジュールを説明する．

メッセージ管理モジュール アプリケーションやサービスに対してプログラミングインタ

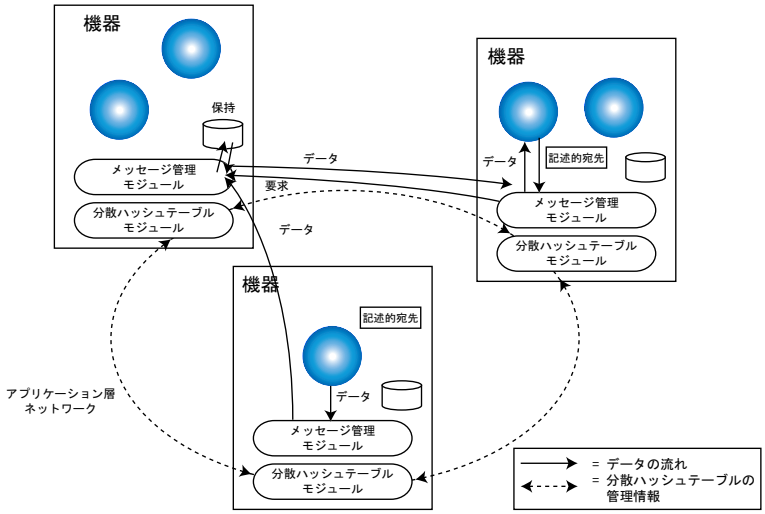


図 4.1: ミドルウェア構成図

フェースを提供し，メッセージ送信要求，メッセージ受信要求，サービス登録要求，メッセージ通知要求を処理する．また，これらの要求を中間機器で動作するメッセージ管理モジュールにフォワードする．別のメッセージ管理モジュールから要求をフォワードされると，それを処理する．

分散ハッシュテーブルモジュール アプリケーション層で動的に分散ハッシュテーブルを構築し，それを管理するメッセージの送受信と処理を行う．同モジュールは，記述的宛先と機器を対応付けるテーブルを保持する．初期化時にはこのテーブルを作成し，その後機器が追加されるたびにそれを更新する．また，検索要求を処理する際もこのテーブルを用いて行う．

図 4.1 に本ミドルウェアの構成図を示す．メッセージ管理モジュールは，サービスの記述的宛先を管理する中間機器を検索する際に分散ハッシュテーブルモジュールを利用する．分散ハッシュテーブルモジュールは，機器が追加され，テーブルが更新された際にメッセージ管理モジュールに通知する．

4.3 記述的宛先

本ミドルウェアでは，記述的宛先の構造として文字列の配列を用いる．サービスの記述的宛先は，以下のように表される．

“HTLAB” “i208” “camera” “web” “panasonic”

文字列の順序は区別しない．記述的宛先をバイト列に変換する際には，それぞれの文字列をアルファベット順に並び変え，連結した一つの文字列をハッシュ関数を用いて固定長のバイト列に変換する．ハッシュした記述的宛先は，中間機器を検索する際に用い

る．この際完全一致を用いて検索を行う．これは，ハッシュされた宛先を比較する際に部分一致を用いるのが困難だからである．

このような構造を用いることで，サービスの単純な指定が可能となる．逆にこの方法の制約として，宛先に意味を付加できないことと，名前の衝突が起きやすくなることがある．記述的宛先に構造や型を持たせれば，その意味を通信ミドルウェアが解釈し，それに従って異なる扱いができる．例えば組織を示す属性を付加できれば，中間機器としてその組織で管理されている機器を割り当てられる．また，順序や型が異なる記述的宛先同士を区別することで，サービスはより多くの記述的宛先を利用できる．本ミドルウェアでは，記述的宛先の意味はアプリケーションとサービス間で共有し，通信ミドルウェアはこれを利用しない．また，複雑な記述的宛先を用い，順序や型を持たせると，記述的宛先の僅かな違いでアプリケーションが意図したサービスにメッセージが配信されないことも考えられる．これを防ぐためには柔軟な一致を用いる必要があるが，その場合異なるサービスを区別するには注意が必要となり，ミドルウェア側の負担が増える．

4.4　メッセージ管理モジュール

メッセージ管理モジュールは，アプリケーションやサービスから本ミドルウェアを利用するためのプログラミングインタフェースを提供する．インタフェースとしてメッセージ送信要求，メッセージ受信要求，サービス登録要求，メッセージ通知要求がある．メッセージ管理モジュールはこれらの要求を該当するサービスの中間機器で動作するメッセージ管理モジュールにフォワードする．要求をフォワードされたメッセージ管理モジュールは，メッセージ送信要求の場合，該当するサービスのメッセージをメモリ領域に保持する．本論文ではこのメモリ領域を**サービスバッファ**と呼ぶ．メッセージ受信要求の場合，同サービスバッファからデータを取り出し，要求をフォワードしたメッセージ管理モジュールに返す．サービス登録要求の場合，既にそのサービスのサービスバッファが存在すれば何もせず，そうでない場合は新たにサービスバッファを作成する．サービスバッファ内のメッセージは，一定期間保持した後破棄する．バッファ自体も同様に，一定期間保持した後破棄する．まとめると，同モジュールの主な機能は，以下の通りである．

- プログラミングインタフェースの提供
- サービスの登録
- メッセージの送受信
- データの管理

4.4.1　プログラミングインタフェースの提供

アプリケーションやサービスのプログラマは，メッセージ管理モジュールのインスタンスを生成し，そのメソッドを呼び出すことで本ミドルウェアを利用する．メッセージ

管理モジュールが提供するメソッドは以下の通りである．

send メソッド

send メソッドは，アプリケーションがサービスにメッセージを送るために利用するメソッドである．引数には，サービスの記述的宛先とデータを指定する．これらを受け取ると，送信要求を中間機器に対して送信する．送信要求を受け取ったメッセージ管理モジュールは，該当するサービスバッファにメッセージを保存する．

receive メソッド

receive メソッドは，サービスがアプリケーションからのメッセージを受信するために利用するメソッドである．引数に記述的宛先を指定する．これを受け取ると，受信要求を中間機器に対して送信する．受信要求を受け取ったメッセージ管理モジュールは，メッセージを取得し，受信応答を返す．

register メソッド

register メソッドは，サービスを通信ミドルウェアに登録するためのメソッドである．引数にはサービスの記述的宛先を指定する．これを受け取ると，登録要求を中間機器に対して送信する．登録要求を受け取ったメッセージ管理モジュールは，サービスバッファが既に存在しない場合，それを新たに作成する．

requestNotify メソッド

requestNotify メソッドは，メッセージが送信された場合に即座にそれを通知することを中間機器に対して要求するためのメソッドである．引数には記述的宛先を指定する．これを受け取ると，通知要求を中間機器に対して送信する．通知要求を受け取ったメッセージ管理モジュールは，送信元を通知リストに追加する．

これらのメソッドは，実際に要求を中間機器に送信する必要がある．そのため，まず記述的宛先を一つの文字列に変換する．次にそれを分散ハッシュテーブルモジュールに渡し，中間機器を検索する．それぞれの要求は検索された中間機器に対して送信される．

4.4.2　サービスの登録

サービスは，自身が動作する機器のメッセージ管理モジュールを利用してその記述的宛先を登録する．図 4.2 にサービスの登録手順を示す．サービスが register メソッドを呼ぶと，登録要求としてその記述的宛先が分散ハッシュテーブルモジュールに渡される (1)(2)．分散ハッシュテーブルモジュールはそれをハッシュし，中間機器を検索した後に，

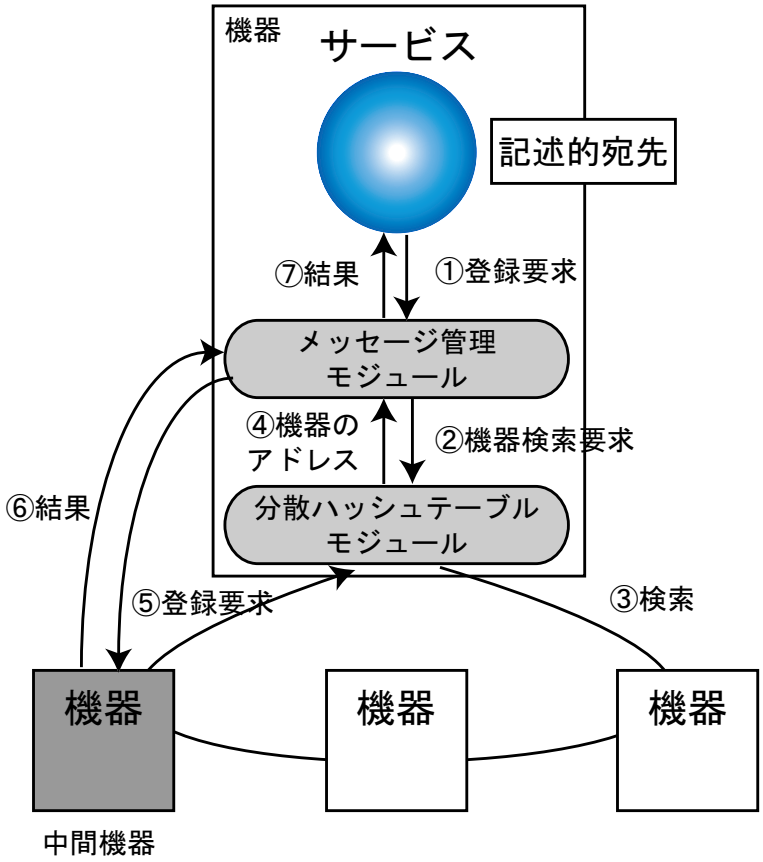


図 4.2: サービスの登録

そのアドレスを返す (3)(4)．次に，メッセージ管理モジュールはその中間機器に登録要求を送る (5)．中間機器は，まだそのサービスが登録されていない場合は新たにサービスバッファを作成する．逆に既に同じサービスが登録されていた場合は何もしない．新たに記述的宛先を割り当てたかどうかは要求元のメッセージ管理モジュールに返される (6)．メッセージ管理モジュールはこの結果をサービスに返す (7)．サービスはその結果をもとに，別の名前で登録し直すか判断できる．その後メッセージ管理モジュールは定期的に更新要求を発行する．

4.4.3　メッセージの送受信

図 4.3 にメッセージの送受信方法を示す．アプリケーションやサービスは，自身が動作する機器のメッセージ管理モジュールに，送受信要求として記述的宛先やデータを指定する．具体的なアドレスはメッセージ管理モジュールが埋める．表 4.1 にメッセージ管理モジュール同士が送受信するメッセージの形式を示す．

メッセージ送信者は，send メソッドに記述的宛先と送信するデータを指定してメッセージ管理モジュールに送信要求を渡す．メッセージ管理モジュールはサービス登録の際と同様の手順で宛先を担当する機器を検索し，該当機器のアドレスを取得した後，同機器

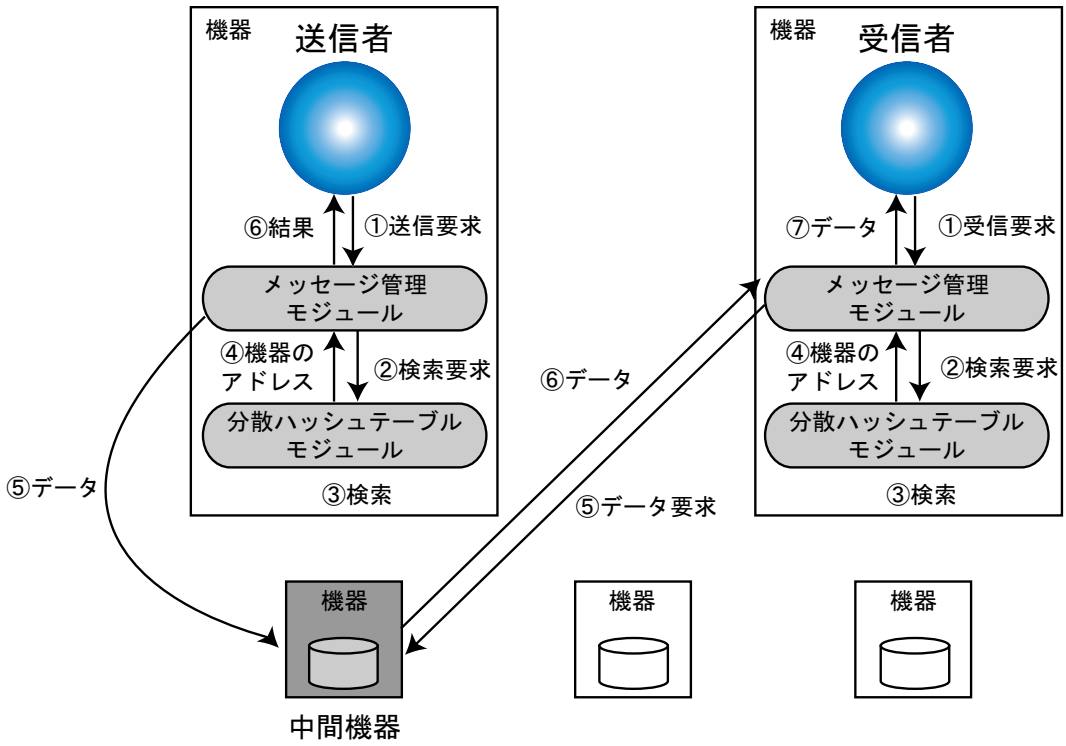


図 4.3: メッセージの送受信

に対してデータを送る．

メッセージ受信者は，receive メソッドに記述的宛先を指定してメッセージ管理モジュールに受信要求を渡す．メッセージ管理モジュールは宛先を中間機器を検索し，中間機器にデータ要求を送る．最後に，該当機器から返されたデータを受信者に渡す．

表 4.1: メッセージの形式

送信元アドレス	宛先アドレス	記述的宛先	データ

4.4.4 サービスバッファの管理

メッセージ管理モジュールは，特定のサービスに宛てられたメッセージを保持するため，サービスバッファを管理する．サービスバッファはサービスごとに用意するため，その記述的宛先と対応付けられる．一つのバッファは，そのサービスに宛てられたメッセージのデータを保持する．サービスバッファはサービス登録要求があった際に作成される．

データはリストとして保持され，メッセージ送信者からデータを渡された際に新たな要素が追加される．メッセージ受信者には，同リストからデータを返す．エントリやデータは一定期間保持した後に削除する．図 4.4 にデータ管理の概観を示す．

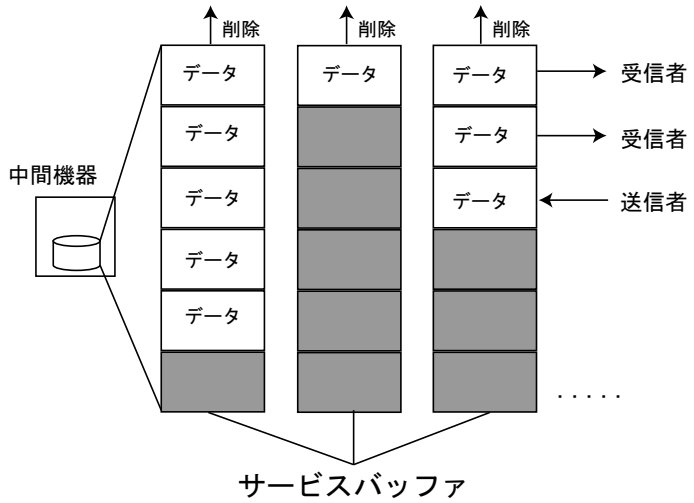


図 4.4: サービスバッファの管理

データの保持

それぞれのデータにはシーケンス番号と書き込み時間が付加される．表 4.2 にメッセージ管理モジュール内で保持されるデータの情報を示す．

表 4.2: データの形式

データ	シーケンス番号	タイムスタンプ

エントリとデータの削除

サービスバッファは，一定期間保持された後に削除される．ただし，別のメッセージ管理モジュールから更新メッセージが届いた際には，保持期間が延長される．サービスバッファが削除されると，それに含まれるデータも破棄される．

アプリケーションが削除されたサービスバッファに対してメッセージを送ると，サービスが存在しない場合と同様，メッセージは破棄される．同様に，サービスが削除された記述的宛先からメッセージの読み込みを行うと失敗する．

データも同様に一定期間保持された後に削除される．データが挿入された際，その時の時刻を表すタイムスタンプが付加される．メッセージ管理モジュールは定期的にデータを検査し，ミドルウェアで定義された保持期限とタイムスタンプの時刻の和が現在の時刻を越えていた場合にデータを削除する．

データにはシーケンス番号を割り振る．これは，メッセージ受信者側でメッセージの順序を管理するために利用する．受信者は，一度受信した後に再び受信する際には，最後に受信したデータの次のシーケンス番号を持つデータを要求する．

4.5　分散ハッシュテーブルモジュール

分散ハッシュテーブルモジュールは，分散ハッシュテーブルの構築と管理，及び中間機器の検索を行う．分散ハッシュテーブルの構築は，モジュールが初期化される際に行う．分散ハッシュテーブルモジュールは，アプリケーションやサービスのプログラマがメッセージ管理モジュールのインスタンスを生成した際に初期化される．新たな機器が分散ハッシュテーブルに参加すると，それが更新される．この際，分散ハッシュテーブルモジュールはメッセージ管理モジュールにそれを通知する．中間機器の検索はメッセージ管理モジュールに要求された際に行い，結果として中間機器のアドレスを返す．分散ハッシュテーブルモジュールは以下の機能を提供する．

- 分散ハッシュテーブルの構築
- 分散ハッシュテーブルの管理
- 中間機器の検索

4.5.1　アルゴリズム

分散ハッシュテーブルのアルゴリズムについて説明する．分散ハッシュテーブルは，複数のノードで構成されるハッシュテーブルであり検索鍵と値をノードに割り当てる．そのため，検索鍵を指定するとそれを割り当てられたノードを特定できる．また，検索を効率的に行うためのアルゴリズムとして Chord[9]，CAN[10]，Tapestry[11]，Pastry[12] などが提案されている．これらのアルゴリズムは検索を高速にし，ノードの動的な追加と削除に対応する．それぞれ共通するのは，複数のノードで仮想的な識別子空間を構築し，各ノードに均等に検索鍵を割り当てることである．本ミドルウェアでは Chord を利用している．

Chord

Chord アルゴリズムは，m ビットにハッシュされた検索鍵と，同じく m ビットにハッシュされたノード ID を，同一の仮想的な環状の名前空間に割り当てる．ノードとノードの間にある検索鍵は，時計回りに見て一つ先にあるノードに割り当てられる．このノードを検索鍵の**担当ノード**と呼ぶ．また，各ノードは自身の次のノードを把握する．これにより，全ての検索鍵について，担当ノードを検索できる．また，ノードの追加と削除を容易にするため，各ノードは自身の一つ前のノードも把握する．自身の次のノードを**後継ノード**，自身の前のノードを**前任ノード**と呼ぶ．検索をより効率的にするため，各ノードは検索鍵とその担当ノードの ID を対応付けるテーブルを保持する．各ノードが保持するテーブルは，m 個のエントリで構成され，自身の ID を n とした場合，それぞれのエントリは $n + 2^{k-1} \bmod 2^m (1 \leq k \leq m)$ の検索鍵に対する担当ノードの情報を含む．

図 4.5 に Chord アルゴリズムの概観を示す．Chord の環に参加する場合，自身のテーブルを作成してから他のノードに対してテーブルの更新要求を送る．既存のノードが指定

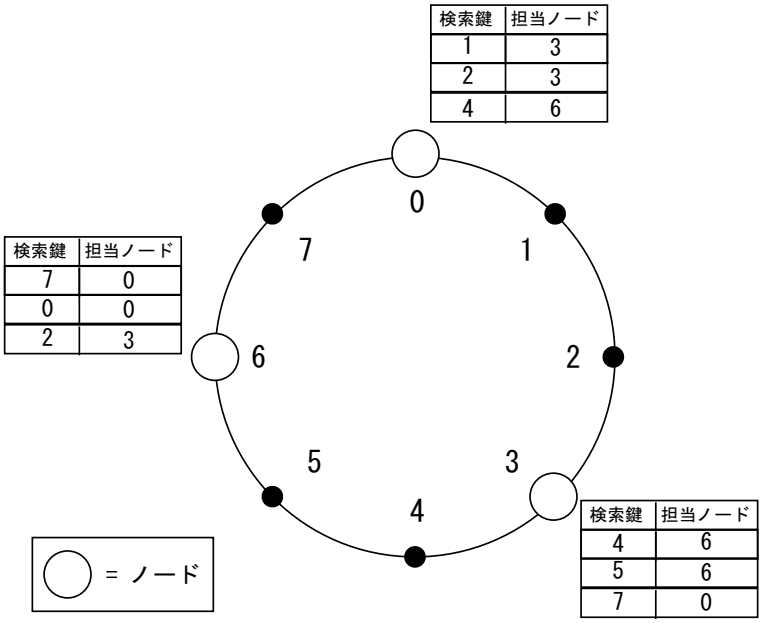


図 4.5: Chord アルゴリズム

されている場合，まずそのノードに問い合わせて自身の次のノードを検索し，そのノードに，前任ノードを自身に変更するよう要求する．その後，各エントリの検索鍵に対して，担当ノードの情報を埋める．次のノードより前の検索鍵については次のノードを登録し，それ以外の担当ノードについては，既存ノードを利用して検索する，

自身が新たに参加されたことを他のノードに通知する際は，$n - 2^{k-1} \bmod 2^m (1 \leq k \leq m)$ の検索鍵の前のノード ID を持つノードに，k 番目の検索鍵に対する担当ノードを自身に設定するように要求する．

特定の検索鍵の担当ノードを検索するには，自身のルーティングテーブルを利用する．ルーティングテーブルに指定された検索鍵のエントリが存在する場合はその担当ノードが返される．そうで無い場合は，検索鍵の直前のノードに検索要求がフォワードされる．これは，各ノードが自身の ID に近い情報程多く保持しているためで，効率的に検索を行える．

4.5.2 分散ハッシュテーブルの構築

分散ハッシュテーブルモジュールは，検索鍵とノードを対応付けるテーブルを保持する．本論文ではこれを**ルーティングテーブル**と呼ぶ．ルーティングテーブルは，記述的宛先をハッシュした検索鍵と機器のアドレスを対応付ける．新しい機器がネットワークに参加した際，既に他の機器間が分散ハッシュテーブルを構築している場合，それらの機器の一つを利用することで分散ハッシュテーブルに参加する．分散ハッシュテーブルが構築されていない場合，自身を唯一のノードとして初期化を行う．本ミドルウェアでは，メッセージ管理モジュールのインスタンスを生成する際にホスト名を指定し，かつ

そのホストが実際に本ミドルウェアを実行している場合は，既に分散ハッシュテーブルが構築されているものとして初期化を行う．そうでない場合はまだ分散ハッシュテーブルが構築されていないものとして初期化を行う．

4.5.3 分散ハッシュテーブルの管理

ルーティングテーブル

分散ハッシュテーブルに新たな機器が参加した場合，ルーティングテーブルが更新される．ルーティングテーブルには，項目番号，検索鍵，機器の ID，機器のアドレスが保持される．項目番号は，別の分散ハッシュテーブルモジュールから番号を指定したルーティングテーブルの更新要求を受け取る際に利用する．機器の ID は，機器のアドレスを記述的宛先と同じ方法でハッシュしたもので，Chord アルゴリズムにおいて機器を記述的宛先と同じ識別子空間に対応付けるために用いる．表 4.3 にルーティングテーブルの例を示す．

表 4.3: ルーティングテーブルの例

項目番号	検索鍵	機器の識別子	機器のアドレス
1	67492142	67492142	133.27.170.2
2	67492143	74596704	133.27.170.79
3	67492145	74596704	133.27.170.79

特定の検索鍵に対応付けられる機器は，環状の識別子空間を時計回りに探索した際に，検索鍵と同じかそれより先の識別子を持った税所の機器である．検索鍵の先の識別子は，0 をまたがない限り検索鍵より大きい識別子ということになる．特殊な項目として，ルーティングテーブルの項目 1 に対応付けられた機器は，環状の識別子空間で自身の次のノードなので，後継ノードを表す．各ノードがその後継ノードを正しく管理する限り，ルーティングテーブルの情報が正しく無くても，検索は行える．また，機器の参加を効率的に処理するため，各ノードはルーティングテーブルとは別に環状の識別子空間で自身の前のノードを表す前任ノードのアドレスも管理する．分散ハッシュテーブルに新たな機器が参加すると，後継ノードと前任ノードが更新した後，新しい機器をルーティングテーブルに登録する必要がある．

ルーティングメッセージ

分散ハッシュテーブルモジュールは，ルーティングテーブルを管理するためのメッセージを他の分散ハッシュテーブルモジュールと送受信する．このメッセージの形式を表 4.4 に示す．種類はこのメッセージの種類を表し，データにはメッセージの種類ごとに異な

る情報が付加される．メッセージの種類とその種類のメッセージに付加されるデータを表 4.5 に示す．

表 4.4: ルーティングメッセージの形式

送信元アドレス	宛先アドレス	種類	データ

表 4.5: ルーティングメッセージの種類

種類	データの種類
後継ノード検索要求	検索鍵
後継ノード検索応答	機器の識別子，機器のアドレス
前任ノード検索要求	検索鍵
前任ノード検索応答	機器の識別子，機器のアドレス
ルーティングテーブル更新要求	項目番号，機器の識別子，機器のアドレス

4.5.4 中間機器の検索

メッセージ管理モジュールから中間機器の検索を要求されると，分散ハッシュテーブルモジュールはまず自身のルーティングテーブルを検索する．そこに該当する記述的宛先の検索鍵が無い場合，ルーティングテーブルから適切なノードを選び，要求を別の機器で動作する分散ハッシュテーブルモジュールに要求をフォワードする．それぞれの分散ハッシュテーブルモジュールはその後任ノードを正しく管理しているので，自身と自身の後任ノードの識別子の間に検索鍵がある分散ハッシュテーブルモジュールにたどり着くまで要求はフォワードされる．

4.6 本章のまとめ

本研究で構築する通信ミドルウェア，SBD は，複数の機器上で動的にアプリケーション層ネットワークを構築し，サービスに宛てられたメッセージを保持する役割を機器に割り当てる．本研究ではこの機器をサービスに対する中間機器と呼び，メッセージを保持する領域をサービスバッファと呼ぶ．SBD はメッセージ管理モジュールと分散ハッシュテーブルモジュールから構成される．メッセージ管理モジュールはアプリケーションやサービスにプログラミングインタフェースを提供し，メッセージを送受信するとともにそれを保持，管理する．分散ハッシュテーブルモジュールは記述的宛先を特定の機器に割り当て，それを検索するための分散ハッシュテーブルを構築し，管理する．

第5章　SBDの実装と評価

本章ではまず，第4章で述べた設計に基づいて行ったSBDのプロトタイプ実装を説明する．次に，同プロトタイプ実装の基本性能の評価と，それを用いたアプリケーションの記述例を示す．

5.1 実装環境

Java 言語を用いて SBD のプロトタイプ実装を行った．各モジュールは，UDP を用いて別の機器で動作する SBD と通信する．SBD は，一つの既存ノードを指定されると，それを利用してアプリケーション層で動的にネットワークを構築する．各機器上で動作する本ミドルウェアのプロセスは，要求の送信と処理の両方を行う．表 5.1 にプロトタイプの実装環境を示す．以下の節では，実装したクラスのソースコードの一部を示し，それぞれの機能を説明する．ソースコード内で「...」と表記されている部分は，具体的な記述を省略した部分である．

表 5.1: 実装環境

項目	実装環境
ハードウェア	AT 互換機 AMD 600MHz
OS	Linux2.4.18
言語	JDK 1.4.0

5.2 メッセージ管理モジュールの実装

メッセージ管理モジュールは，アプリケーションやサービスにプログラミングインタフェースを提供する MessageManager クラス，アプリケーションやサービスが送受信するメッセージを表す Message クラス，このサーバに送られたメッセージを管理する DataManager クラス，そして保持されたデータを表す DataEntry クラスから構成される．

5.2.1 MessageManager クラス

図 5.1 と図 5.2 に MessageManager クラスの一部を示す．MessageManager クラスは，アプリケーションやサービスが getInstance メソッドを呼び出すことで初期化される．その際，ホスト名を指定するとすでに分散ハッシュテーブルが構築されているものとして初期化を行う．ホスト名が指定されないと，新たな分散ハッシュテーブルを構築する．本プロトタイプ実装では，一つの機器に対して一つの MessageManager インスタンスを生成する．アプリケーションやサービスへのプログラミングインタフェースとして，send，receive，register，requestNotify のメソッドを提供する．これらのメソッドは，まず中間機器を検索し，メッセージを作成してそれを中間機器に送信する．中間機器の検索は，Router インスタンスに記述的宛先を渡すことで行われる．この際，Router インスタンスは中間機器のアドレスを返す．図 5.1 には send メソッドの内容だけ示したが，残りのメソッドもメッセージの種類が異なるだけで同様の処理を行う．実際の送信では，メッセージがバイト列に変換され，UDP ソケットを用いて送信される．

```
public class MessageManager{
    public static final int DEFAULT_PORT;
    MessageManager instance = null;

    Router router;
    DataManager dmanager;
    DatagramSocket sock;
    InetAddress myAddress;

    MessageManager(String knownHost){
      router = new Router(knownHost);
      dmanager = new DataManager();
      try{
          myAddress = InetAddress.getLocalHost();
          sock = new DatagramSocket(DEFAULT_PORT);
      }catch(Exception e){ e.printStackTrace(); }

      new ListenerThread(sock).start();
    }
    // インスタンスの初期化
    MessageManager getInstance(String knownHost){
      if(instance == null)
        return new MessageManager(knownHost);
      else return instance;
    }

    MessageManager getInstance(){
      if(instance == null)
        return new MessageManager();
      else return instance;
    }
    // アプリケーションやサービスに提供するインタフェース
    public void send(String[] desc, String data){
      InetAddress dst = search(desc);
      Message msg = new Message(myAddress, dst, Message.SEND_REQUEST,
                                desc, data);
      sendMessage(msg);
    }
    public String receive(String[] desc){ ... }
    public void register(String[] desc){ ... }
    public void requestNotify(String[] desc){ ... }
    // メッセージの送受信
    void sendMessage(Message msg){
      try{
        byte[] data = msg.toByteArray();
        DatagramPacket packet = new DatagramPacket( ... );
        sock.send(packet);
      }catch(Exception e){ e.printStackTrace(); }
    }
    Message receiveMessage(int number){ ... }
```

図 5.1: MessageManager クラスの一部

```
    // 中間機器の検索
    InetAddress search(String[] desc){
      InetAddress ia = router.findSuccessor(desc);
      return ia;
    }
    // 配列から文字列への変換
    String marshal(String[] buf){ ... }
    // 別の MessageManager からの要求，応答の処理
    void processSRQ(Message msg){ ... }
    void processRRQ(Message msg){ ... }
    void processRRSP(Message msg){ ... }
    void processREGRQ(Message msg){ ... }
    void processAN(Message msg){ ... }
    void processN(Message msg){ ... }
    // 別の MessageManager からのメッセージを受信する内部クラス
    class ListenerThread extends Thread{
      DatagramSocket sock;
      DatagramPacket packet;
      byte[] buffer;

      ListenerThread(DatagramSocket sock){ this.sock = sock; }

      public void run(){
          while(true){
            try{
                packet = new DatagramPacket(buffer, buffer.length);
                sock.receive(packet);
                ...
                Message msg = new Message(data);
                new MessageHandler(msg).start();
            }catch(Exception e){ e.printStackTrace(); }
          }
      }
    }
    // 別の MessageManager からのメッセージを処理する内部クラス
    class MessageHandler extends Thread{
      Message msg;
      MessageHandler(Message msg){
          this.msg = msg;
      }
      public void run(){
          byte cmd = msg.cmd;

          switch(cmd){
            // 種類ごとの処理
          }
      }
    }
}
```

図 5.2: MessageManager クラスの一部 (続き)

requestNotify メソッドを呼び出し，MessageManager からメッセージを通知されるアプリケーションやサービスは，以下に示す MessageListener インタフェースが定義するメソッドを実装する．

```
public interface MessageListener{
    public void messageNotified(Message msg);
}
```

MessageManager クラスは，上で述べたサービスやアプリケーションからの要求以外に，他の機器で動作する MessageManager の要求も処理する．そのため，逐次 UDP パケットを受け付ける．受信した UDP パケットからメッセージを復元し，その種類に応じて処理を行う．図 5.2 に示した processXX メソッドの内，接尾に RQ と名前のついたメソッドは MessageManager の要求を処理する．接尾に RSP と名前のついたメソッドは，自身が発行した要求に対する応答を処理する．

```
public class Message{
    static final byte SEND_REQUEST = 0x00;
    static final byte SEND_RESPONSE = 0x01;
    static final byte RECEIVE_REQUEST = 0x02;
    static final byte RECEIVE_RESPONSE = 0x03;
    static final byte REGISTER_REQUEST = 0x04;
    static final byte REGISTER_RESPONSE = 0x05;
    static final byte NOTIFIFY_REQUEST = 0x06;
    static final byte NOTIFIFY = 0x07;

    InetAddress src, dst;
    byte cmd;
    int number;
    String desc;
    String data;

    Message(InetAddress src, InetAddress dst,
            byte cmd, String desc, String data)
    {
      this.src = src;
      this.dst = dst;
      this.cmd = cmd;
      this.desc = desc;
      this.data = data;
    }

    void build(byte[] bytes){ ... }
    byte[] toByteArray(){ ... }
    String toString(){ ... }
}
```

図 5.3: Message クラスの一部

5.2.2 Message クラス

図 5.3 に Message クラスの一部を示す．Message クラスは，MessageManager 同士が送受信するメッセージを表す．内部には，送信元と送信先の IP アドレス，メッセージの種類，シーケンス番号，記述的宛先，およびデータを保持する．シーケンス番号は，MessageManager がブロックする受信，つまり応答を期待する要求を発行する際に利用される．また，本プロトタイプ実装ではデータとして文字列を送受信する．従ってデータは String 型で保持する．

5.2.3 DataManager クラス

図 5.4 と図 5.5 に DataManager クラスの一部を示す．DataManager クラスは，サービスバッファとその中のデータを管理する．サービスバッファは Hashtable を用いて管理される．同 Hashtable は，サービスの記述的宛先とそのデータを対応付ける．データは次に述べる DataEntry インスタンスのリストとして管理される．

サービスの登録を処理する際には，まず Hashtable 内にその記述的宛先があるか検査し，それが無い場合は処理を行わない．データの追加や取得を処理する場合にも同様の検査を行う．サービスの登録は空の DataEntry を挿入することで行われる．検査の結果は MessageManager に返す．

サービスバッファにデータを追加する際には，最後に追加されたデータのシーケンス番号を一つ増加させたシーケンス番号と，データを追加する時間を指定して新しい DataEntry を作成し，サービスバッファに追加する．逆にサービスバッファからデータを取得する際には，サービスバッファの Vector から DataEntry を取りだし，そのデータが返される．現在の実装では，最後に追加されたデータを返している．本研究の目的は保持されてから一定期間内のデータを受信者が取得できることなので，この実装には問題がある．どの範囲のデータを返すか，またどのようにその範囲を指定できるか考慮し，新たな機能を追加する必要がある．

図 5.5 に示した 2 つの内部クラスは，一定間隔でサービスバッファとそれに含まれるデータを検査し，保持期限の過ぎているものを削除する．

5.2.4 DataEntry クラス

図 5.6 に DataEntry クラスの一部を示す．DataEntry クラスはサービスバッファに保持されるデータを表す．内部には，データ自体，シーケンス番号，タイムスタンプを保持する．既に述べたように，本プロトタイプ実装ではデータを文字列として保持する．また，シーケンス番号は受信者側で受信するデータの範囲を指定するのに利用される．

```
public class DataManager{
    public static long SERVICE_EXPIRATION;
    public static long DATA_EXPIRATION;

    Hashtable dataTable;

    DataManager(){ ... }

    // サービスの登録
    boolean registerService(byte[] desc){
      if(dataTable.containsKey(desc))
          return false;

      Vector v = new Vector();
      DataEntry entry = new DataEntry();
      v.add(entry);
      dataTable.put(desc, v);

      return true;
    }
    // データの付加
    void appendData(byte[] desc, String data){
      Vector v;
      Enumeration keys = dataTable.keys();

      if((v = (Vector)dataTable.get(desc)) == null)
          return;

      int sequence = ((DataEntry)(v.lastElement())).getSequence() + 1;
      long timeStamp = System.currentTimeMillis();

      DataEntry de = new DataEntry(data, sequence, timeStamp);

      v.add(de);

      dataTable.put(desc, v);
    }
    // データの取得
    Vector getData(byte[] desc){
      Vector v;
      Enumeration keys = dataTable.keys();

      if((v = (Vector)dataTable.get(desc)) == null)
          return null;

      DataEntry entry = (DataEntry)v.lastElement();

      return (String)de.data;
    }
```

図 5.4: DataManager クラスの一部

```
    // データの破棄を行う内部クラス
    class ServiceChecker extends Thread{
      Enumeration services;

      public void run(){
          try{
            services = dataTable.keys();
            while(services.hasMoreElements()){ ... }
            Thread.sleep(SERVICE_EXPIRATION);
          }catch(Exception e){ e.printStackTrace(); }
      }
    }

    // データの破棄を行う内部クラス
    class DataChecker extends Thread{
      Enumeration services;
      Vector data;
      DataEntry entry;
      long currentTime;

      public void run(){
          try{
            services = dataTable.keys();
            currentTime = System.currentTimeMillis();

            while(services.hasMoreElements()){
                data = (Vector)dataTable.get(services.nextElement());
                for(int i=0; i<data.size(); i++){
                  entry = (DataEntry)data.elementAt(i);
                  if((entry.timeStamp + DATA_EXPIRATION) > currentTime)
                    data.removeElementAt(i);
                }
            }
            Thread.sleep(DATA_EXPIRATION);
          }catch(Exception e){ e.printStackTrace(); }
      }
   }
}
```

図 5.5: DataManager クラスの一部 (続き)

5.3 分散ハッシュテーブルモジュールの実装

分散ハッシュテーブルモジュールは，ルーティングメッセージの送受信を行い，MessageManager に分散ハッシュテーブルの機能を提供する Router クラス，ルーティングテーブルを表す RouteTable クラス，ルーティングメッセージを表す RouteMessage クラス，各機器の情報を表す Node クラスから構成される．

```
public class DataEntry{
    String data;
    int sequence;
    long timeStamp;

    DataEntry(String data, int sequenceNumber, long timeStamp){
      this.data = data;
      this.sequence = sequence;
      this.timeStamp = timeStamp;
    }

    int getSequence(){ return sequence; }

    public String toString(){ ... }
}
```

図 5.6: DataEntry クラスの一部

5.3.1 Router クラス

図 5.7 から 5.9 に Router クラスの一部を示した．Router という名前だが，メッセージの経路を撰択するのでは無く，特定の検索鍵が割り当てられた機器を検索するのがその主な機能である．Router クラスは大きいクラスなので，おおまかな処理に区切って説明する．また，本ミドルウェアでは一つの機器を Chord アルゴリズムにおける一つのノードとして扱うため，以下ではこれらの用語を同義として扱う．

初期化と分散ハッシュテーブルへの参加

まずインスタンス自体の初期化と MessageManager に提供する機能，分散ハッシュテーブルへの参加の部分を説明する．図 5.7 と図 5.8 にこの部分を示した．コンストラクタの内容は図示しなかったが，自身の情報，RouteTable クラス，UDP ソケット，内部クラスの初期化が行われる．自身の情報には ID と Node インスタンスが含まれる．これらは自身が動作する機器の IP アドレスを用いて初期化される．ID は createID メソッドを用いて初期化する．同メソッドは，任意のバイト列を引数として取得し，SHA アルゴリズムを用いて 160bit の識別子を生成する．本プロトタイプ実装で MessageManager に提供している機能は，記述的宛先を指定することで中間機器を検索する機能である．検索には MessageHandler 内部クラスが利用される．検索処理は，他の機器で動作する Router クラスの機能を呼び出し，ブロックする可能性があるので，処理の際にインスタンスを生成する．この部分はスレッドプールなどを用いてメモリをより効率的に利用する実装を行う必要がある．

分散ハッシュテーブルに参加する join メソッドは，既に分散ハッシュテーブルを構築している機器のホスト名が指定されているかどうかで処理が異なる．以下ではこのノー

```
class Router{
    public static final int DEFAULT_PORT;
    public static final int ADDRESS_SPACE = 160;

    private InetAddress myAddress;
    private byte[] myID;
    private Node myNode;

    Node knownNode;

    private DatagramSocket sock;
    private RouteTable table;
    private ListenerThread lThread;

    ...

    public Router(String name){ ... }

    // 識別子の生成
    static byte[] createID(byte[] bytes){
      byte[] id = null;
      try{
          MessageDigest md = MessageDigest.getInstance("SHA");
          md.update(bytes);
          byte[] digest = md.digest();
          id = new BigInteger(digest).abs().toByteArray();
      }catch(Exception e){ e.printStackTrace(); }
      return id;
    }

    // MessageManager に提供する中間機器検索メソッド
    public InetAddress findSuccessor(String s){
      byte[] key = Router.createID(s.getBytes());
      Node n = new MessageHandler().findSuccessor(key);
      return n.ia;
    }

    // 分散ハッシュテーブルへの参加
    void join(){
      if(knownNodeName != null){
          initNodeTable();
          updateOthers();
      }
      else{
          for(int i=1; i<=ADDRESS_SPACE; i++)
            table.putNode(i, myNode);
          myNode.successor = myNode.predecessor = myNode;
      }
    }
```

図 5.7: Router クラスの一部

```
// ルーティングテーブルの初期化
void initNodeTable(){
  InetAddress addr;
  Node currentNode;
  Node n;

  Node successor = new MessageHandler().
                     findSuccessor(knownNode, myID);
  myNode.successor = successor;
  myNode.predecessor = successor.predecessor;
  new MessageHandler().updatePredecessor(successor, myNode);

  for(int i=1; i<=(ADDRESS_SPACE-1); i++){
      currentNode = table.getNode(i);

      if(table.isEqual(ns, myID) ||
         table.inBetween(ns, myID, cni)){
        // 既存ノードをルーティングテーブルに登録
        table.putNode(i+1, currentNode);
      }
      else{
        // ノードを検索してルーティングテーブルに登録
        n = new MessageHandler().
              findSuccessor(knownNode, ns);
        table.putNode(i+1, n);
      }
  }
}

// 他の分散ハッシュテーブルモジュールのルーティングテーブルの更新
void updateOthers(){
  RouteMessage msg;
  BigInteger offset;
  byte[] id;
  Node n;

  for(int i=1; i<=ADDRESS_SPACE; i++){
      offset = new BigInteger("2").pow(i-1);
      id = new BigInteger(myID).subtract(offset).toByteArray();
      n = new MessageHandler().findPredecessor(id);

      new MessageHandler().updateFingerTable(n, myNode, i);
  }
}
```

図 5.8: Router クラスの一部 (ルーティングテーブルの初期化)

ドを既存ノードと呼ぶ既存ノードが指定されていない場合には，自身を唯一の機器としてルーティングテーブルを初期化する．そうでない場合は既存ノードに一連の要求を送ることでルーティングテーブルの初期化を行う．図 5.8 にホスト名が指定されている場

合の処理を示した．

initNodeTable メソッドは，ルーティングテーブルの初期化に必要な作業を行う．その処理は以下の手順で行われる．

1. 後継ノードの検索と取得
2. 前任ノードの取得
3. 後継ノードの前任ノードを自身に更新
4. その他のノードをルーティングテーブルに登録

後継ノードの検索と取得は既存ノードに通常の検索要求を渡すことで行われる．結果として自身の後継ノードと，後継ノードの前任ノードが返される．自身はそれらの間に入るので，自身の後継ノードと前任ノードにそれらを登録する．また，後継ノードの前任ノードを自身に更新する要求を送る．その他のルーティングテーブルの初期化では，ルーティングテーブルの検索鍵ごとに機器の情報を割り当てる．自身と後継ノードの間の識別子を持った検索鍵は，全て後継ノードに対応付ける．それ以外の検索鍵については既存ノードに検索要求を渡すことでその情報を取得する．

自身のルーティングテーブルを初期化した後，他の機器のルーティングテーブルを更新する．これは updateOthers メソッドが行う．同メソッドでは，自身の情報をルーティングテーブルに保持すべき，つまり自身の ID を n，ルーティングテーブルの項目数を m としたときに $n - 2^{k-1} \bmod 2^m (1 \leq k \leq m)$ の検索鍵の情報をルーティングテーブルに持つノードに対してルーティングテーブルの更新要求を送る．

ルーティングメッセージの処理

図 5.9 に，Router クラスでルーティングメッセージの処理を行う部分のソースコードを示した．Router クラスは，ルーティングメッセージを UDP ソケットを用いて逐次受信する．この受信は ListenerThread 内部クラスを用いて行う．データグラムを受信すると，そこからルーティングメッセージを復元し，その処理を行う．メッセージの処理は MessageHandler 内部クラスが行う．MessageHandler は受信したルーティングメッセージの処理以外に，分散ハッシュテーブルに関する検索要求の全てを行う．受信したルーティングメッセージの処理は run メソッドで行う．ここでは，ルーティングメッセージ種類に応じて異なる処理を行う．本プロトタイプ実装では，ルーティングメッセージは他の機器で動作する Router クラスに順次フォワードされ，処理される．検索要求の一部は，処理を続けるために，フォワードしたルーティングメッセージの応答を待つ必要がある．それにより他の処理が影響されることを無くすため，分散ハッシュテーブルに関する検索要求を行う度に MessageHandler のインスタンスが生成される．

```
// ルーティングメッセージの受信を行う内部クラス
class ListenerThread extends Thread{
  DatagramSocket sock;
  DatagramPacket packet;
  byte[] buffer = new byte[65535-40];

  ListenerThread(DatagramSocket sock){ this.sock = sock; }

  public void run(){
      try{
        while(true){
            packet = new DatagramPacket(buffer, buffer.length);
            sock.receive(packet);

            ...

            RouteMessage msg = new RouteMessage(data);

            new MessageHandler(msg).start();
        }
      }catch(Exception e){ e.printStackTrace(); }
   }
}

// ルーティングメッセージの処理を行う内部クラス
class MessageHandler extends Thread{
  RouteMessage msg;

  MessageHandler(RouteMessage msg){ this.msg = msg; }

// ルーティングメッセージの送受信
void sendMessage(RouteMessage msg){ ... }
RouteMessage sendReceive(RouteMessage msg){ ... }
RouteMessage receiveMessage(int number){ ... }
```

図 5.9: Router クラスの一部 (ルーティングメッセージの処理)

5.3.2 RouteMessage クラス

図 5.10 に RouteMessage クラスの一部を示す．RouteMessage クラスは Router 同士が送受信するルーティングメッセージを表す．内部には，送信元と送信先のノード情報，ルーティングメッセージの種類，シーケンス番号，任意の識別子，および任意のノード情報を保持する．ノード情報は後述する Node クラスで管理される．シーケンス番号は，Router が応答を期待する送信を行う際に利用する．これらは必須フィールドとなる．それ以外に，任意の識別子とノード情報を保持できる．これらは，検索要求やその応答で頻繁に利用される．

```
public class RouteMessage{
    public static final byte SUCCESSOR_REQUEST = 0x00;
    public static final byte SUCCESSOR_RESPONSE = 0x01;
    public static final byte PREDECESSOR_REQUEST = 0x02;
    public static final byte PREDECESSOR_RESPONSE = 0x03;
    public static final byte CPF_REQUEST = 0x04;
    public static final byte CPF_RESPONSE = 0x05;
    public static final byte UPDATE_TABLE = 0x06;

    Node src;
    Node dst;
    byte command;
    int number;
    byte[] data;
    byte[] id;

    public RouteMessage(byte[] buf){ build(buf); }
    public RouteMessage(Node src, Node dst,
                        byte command, Node node, byte[] data){ ... }

    public RouteMessage(Node src, Node dst,
                        byte command, byte[] id){ ... }

    public RouteMessage(Node src, Node dst,
                        byte command){ ... }

    void build(byte[] bytes){ ... }
    byte[] toByteArray(){ ... }
    String toString(){ ... }
}
```

図 5.10: RouteMessage クラスの一部

5.3.3 RouteTable クラス

図 5.11 に RouteTable クラスの一部を示す．RouteTable クラスはルーティングテーブルを表す．ルーティングテーブルは項目番号と検索鍵を対応付ける keyTable と，検索鍵とノード情報を対応付ける nodeTable に分割して管理される．テーブルを分割したのは，それぞれの検索を効率的に行うためである．RouteTable は，これらルーティングテーブルにアクセスするメソッド以外に，keyTable を初期化する initKeyTable メソッドを提供する．keyTable に保存される検索鍵は，自身の機器の識別子 (n) と識別子空間の大きさ (m) が得られれば，$n + 2^{k-1} \bmod 2^m, 1 \leq k \leq m$ を計算することで初期化できる．これに対してノード情報は既存ノードに問い合わせなければ取得できないため，RouteTable は nodeTable の初期化を行うメソッドを持たない．

```
class RouteTable{
    Hashtable keyTable; // 検索鍵と機器の ID を対応付けるテーブル
    Hashtable nodeTable; // 機器の ID と機器の情報を対応付けるテーブル
    Node myNode;
    int addressSpace;

    RouteTable(Node myNode, int addressSpace){ ... }

    // 検索鍵テーブルの初期化
    void initKeyTable(){
      BigInteger base ,offset;
      BigInteger two = new BigInteger("2");

      base = two.pow(addressSpace);

      if(myID != null){
          for(int i=1; i<=addressSpace; ++i){
            offset = two.pow(i-1);
            putStart(i, new BigInteger(myID).add(offset).
                 mod(base).toByteArray());
          }
       }
    }

    // ルーティングテーブルへのアクセス
    byte[] getStart(int index){ ... }
    void putStart(int index, byte[] value){ ... }
    Node getNode(int index){ ... }
    Node getNode(byte[] key){ ... }
    void putNode(byte[] id, Node n){ ... }
    void putNode(int index, Node n){ ... }
}
```

図 5.11: RouteTable クラスの一部

5.3.4 Node クラス

図 5.12 に Node クラスの一部を示す．Node クラスはノードの情報として，その ID と IP アドレスを保持する．また，その後継ノードと前任ノードの情報も保持する．この二のノード情報を保持するのは必須ではなく，可能な限り保持され，機器検索に利用される．Node 自体は Message の送信元や宛先，またルーティングテーブルのノード情報として利用される．

5.4 基本性能評価

本節では，本プロトタイプ実装の基本性能を評価した結果を示す．メッセージに含まれるデータ量に対する応答時間の変化と SBD のサーバ数に対する応答時間の変化を測定

```
class Node{
    static int DEFAULT_SIZE = 24;

    byte[] id = null;
    InetAddress ia;
    Node successor, predecessor;

    Node(byte[] id, InetAddress ia){ ... }
    Node(byte[] data){ build(data); }

    void setSuccessor(Node n){ this.successor = n; }
    void setPredeccessor(Node n){ this.predecessor = n; }

    public void build(byte[] data){ ... }
    public byte[] toByteArray(){ ... }
    public String toString(){ ... }
}
```

図 5.12: Node クラスの一部

する 2 つの実検を行った．どちらも SBD の実行には同一の計算機を用いた．表 5.2 にその測定環境を示す.

表 5.2: 測定環境

項目	測定環境
ハードウェア	Compaq EVO D500SF
CPU	Pentium4 1.6GHz
OS	FreeBSD 4.0
メモリ	256MB
ネットワーク	100Mbps Fast Ethernet

測定 1

本測定では，機器 2 台で構成された SBD でのメッセージの配送時間を測定した．数値を記録する側の機器は，サービスの登録とメッセージ通知要求を送った後，自身の記述的宛先を指定したメッセージを送り，その通知が届くまでの時間を記録した．メッセージ自体はもう一方の機器に保持された．機器が 2 台だけなので，ルーティングのオーバヘッドは無く，測定値は純粋なデータ配送にかかる時間を示す．図 5.13 に本測定の結果を示す.

グラフから見て取れるように，データサイズの増加に対して応答時間が増加している．これは予想された結果と言える．データサイズが 0 バイトでも 100 ミリ秒かかっており，

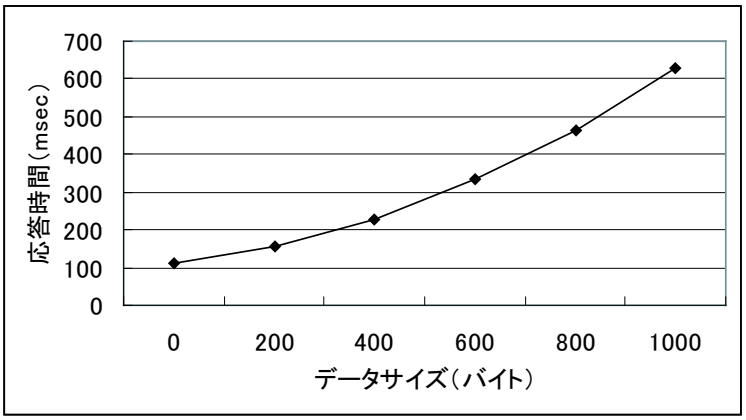


図 5.13: データ量に対する応答時間の変化

配送自体の遅延が大きいと言える．今後ミドルウェアのメッセージ配送部分の改善が必要だと言える．

測定 2

本測定では，データ量が 0 バイトのメッセージの配送時間を，異なる数の SBD のサーバを用いて測定した．宛先はランダムに生成し，メッセージ送信者がメッセージを送信してからそれがサービスバッファに保持されるまでの時間を記録した．図 5.14 に本測定の結果を示す．

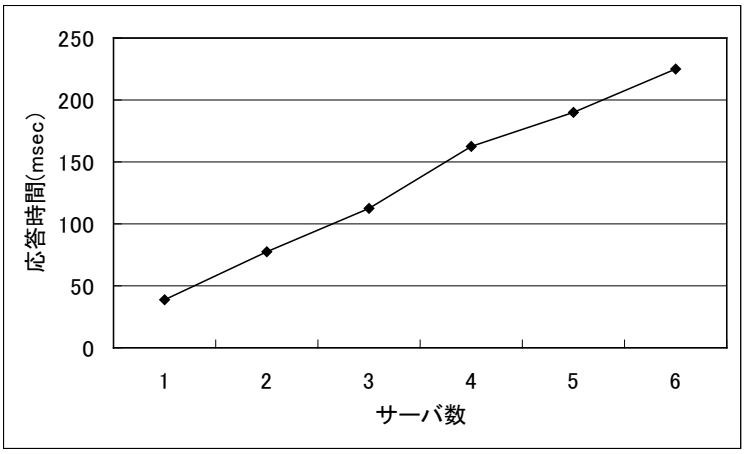


図 5.14: サーバ数に対する応答時間の変化

グラフでは，サーバ数に対して応答時間が増加している．これはルーティングに要する時間の増加を示している．サーバ数が少ないため，応答時間は単調増加している．今回の測定に用いたサーバ数でもメッセージの送信に最大で 200 ミリ秒以上かかっており，

ここでもメッセージ配送自体の速度面での改善が必要であると言える．

5.5 アプリケーション記述例

```
import jp.ac.keio.sfc.ht.kmsf.sbd.*;

public class LightController{
    MessageManager manager;
    String[] lightDescription = {"light", "i208", "color"};

    LightController(){
      manager = MessageManager.getInstance();
      turnLightBlue();
    }

    void turnLightBlue(){
      manager.send(lightDescription, "BLUE");
    }
}
```

```
import jp.ac.keio.sfc.ht.kmsf.sbd.*;

public class LightService implements MessageListener{
    MessageManager manager;
    String[] myDescription = {"i208", "light", "color"};

    LightService(){
      manager = MessageManager.getInstance();
    }

    void register(){
      manager.register(myDescription);
      manager.requestNotify(myDescription);
    }

    void messageNotified(Message msg){
      processMessage(msg.data)
    }

    ...
}
```

本節では，SBD を利用したアプリケーションの記述例を示す．サービスとして，i208 という名前の部屋に設置された，異なる色を表示できるライトの機能を提供する LightService を考える．また，そのライトを制御する LightController アプリケーションを考える．どちらも，getInstance メソッドを用いて，まず MessageManager のインスタンスを取得する．LightService は同インスタンスに対してサービス登録要求とメッセージ通知要求を渡す．また，メッセージが通知された際に実行する messageNotified メソッドを実装する．

LightController は，MessageManager のインスタンスに対して “BLUE” という文字列の送信要求を渡す．同文字列は LightService に対して通知され，LightService はこれを処理する．

メッセージの記述的宛先として文字列の配列を利用するため，単純な方法でサービスを指定できる．また，配列の各要素の順序が異なっていても同一の記述的宛先として扱われるため，アプリケーションプログラマはサービスの記述的宛先に含まれる要素だけ把握すればよい．

5.6 本章のまとめ

本章では，SBD のプロトタイプ実装を説明した．SBD のプロトタイプ実装は Java 言語を用いて行われ，SBD の実行プロセス同士は UDP を用いてメッセージとルーティングメッセージの両方を送受信する．基本性能の測定からは，メッセージの配送部分で改善が必要なことが言える．また，アプリケーション記述例を用いて SBD が簡潔なアプリケーションプログラミングインタフェースを提供することを示した．

第6章　関連研究

本章では，SBD を関連するシステムと比較する．遍在する機器に対して記述的間接型通信を提供するシステムとして，JavaSpaces，Intentional Naming System，Internet Indirection Infrastructure を挙げる．

6.1 Java Spaces

Sun Microsystems で開発されている Java Spaces[8] は，複数のサービス同士がデータを交換するための共有空間である．サービスは，Entry と呼ばれるオブジェクトの集合をこの共有空間を介して交換する．この共有空間に対しては write，read，take の操作が行える．write は共有空間に対して Entry を書き込む．書き込まれた Entry は Template と呼ばれるオブジェクトの集合を用いて検索される．Template を指定して read を行うと，オブジェクト同士が一致する Entry のコピーが返される．この際オブジェクトに NULL が指定されていると，ワイルドカードとして利用できる．take は read と同様に検索を行うが，Entry のコピーを返す際に Entry 自体が削除される．図 6.1 に Java Spaces の概観を示す．

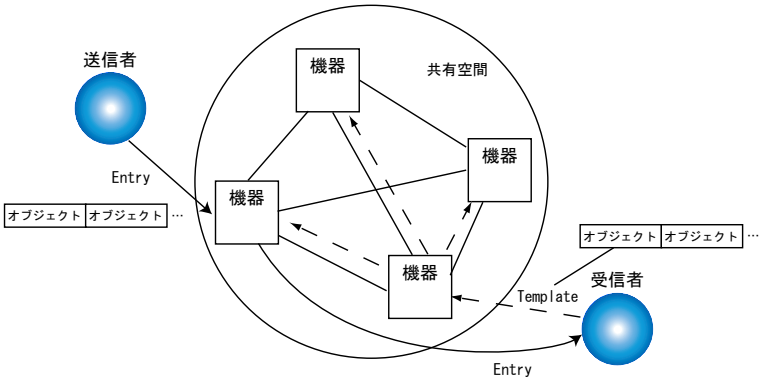


図 6.1: Java Spaces の概要

JavaSpaces はタプルスペース [13] の概念を利用している．タプルスペースは当初，並列コンピューティング環境で異なる計算機上で動作するプロセス同士が協調するために考案されたものである．異なるプロセスは共有空間を用いてタプルと呼ばれる型を持った変数の集合を読み書きすることでデータの交換を行う．write，read，take の操作もタプルスペースの概念である．タプルは永続的にタプルスペースに保持され，送信者がメッセージを送った直後に停止した後でもメッセージ自体は残り続け，その後に実行された受信者がそのメッセージを受け取れる．

Java Spaces がタプルスペースと異なる主な点は，変数の集合の代わりにオブジェクトの集合を用いていることと，Entry に有効期間を設けていることである．write 操作を行うときにこの有効期間を指定すると，Entry はその期間の間は共有空間で保持される．

Java Spaces を分散環境で効率的に実装するには，Entry を分散して保持する方法を考慮する必要がある．検索空間を縮小するため，Entry に含まれるオブジェクトごとに異なる下位の共有空間を構築できる．また，Entry の一部をハッシュし，その値ごとに異なる空間を構築することで下位の共有空間をさらに分割できる．この際，複数の機器への Entry の分配も可能となる．分散環境での Entry の書き込みと読み込みについては，Entry をマルチキャストする方法と Taemplate をマルチキャストする方法が考えられる．しかし，広域でのスケーラビリティを実現するには，効率的なマルチキャスト手法が必

要である，

多くの場合 Entry には Java RMI によるリモートオブジェクトへの参照が含まれる．Entry を読み込んだプロセスはそれを利用して直接リモートオブジェクトの機能を利用できる．

実際の通信はこのように多くの場合 RMI を利用して行うため，Java Spaces は共有空間を用いてデータを交換するよりも，サービスの検索を行うことを重視していると言える．それに対して本研究では，共有空間を介してデータ通信を行うことを目的としている．Entry やタプルは，型を持つため，本研究で用いる記述的宛先よりも柔軟である．本研究の記述的宛先は柔軟性の代わりに単純性を向上している．Entry は宛先とデータ両方にその内容を利用するが，本研究の記述的宛先は宛先だけを表す．また，本研究では分散ハッシュテーブルを用いることにより記述的宛先の効率的な分配を実現している．

6.2 Intentional Naming System

米 MIT で開発されている INS(Intentional Naming System)[14] は，記述的宛先を用いてサービスの検索とメッセージの配送を同時に行う，ネーミングシステム兼メッセージ配送システムである．記述的宛先として，木構造を持った名前を利用する．同宛先は，要素と値の組が木構造を成すもので，値が次の要素を子として持つ．要素と値は文字列で表される．このため，要素が同じでも異なる値に対して別々の枝を作成でき，記述力が高い．また，それぞれの枝はサービスが定義できるため，DNS のように要素に対する制約が高いシステムよりも柔軟である．図 6.2 の (a) に宛先の例を示す．記述的宛先の名前解決は INR(Intentional Naming Resolver) と呼ばれるリゾルバが行う．INR はアプリケーション層で動的にネットワークを構築し，複数の INR は分散して記述的宛先を管理する．図 6.2 の (b) に INS の概要を示す．

INS ではサービスがその記述的宛先を INR に登録し，アプリケーションやユーザは INR に対して記述的宛先を指定してサービス検索要求を発行する．INS は，既存のネーミングシステムと同様にサービスの名前解決だけのために利用できる他，メッセージをサービス検索に付加することでメッセージ配送にも利用できる．この際，通信方法の指定によりマルチキャストとエニーキャストが行える．

INS では，効率的な宛先の検索を行うために二つの方法を提案している．vspace を用いる方法 [15] と，INS/Twine[16] と呼ばれる拡張システムである．

最初の方法では，名前空間を仮想的な名前空間に分割する．この仮想的な名前空間は vspace と呼ばれる．各 vspace は任意の数の INR で管理され，また各 INR は任意の数の vspace を管理できる．vspace を管理する INR はその更新情報を新たに処理する．一つの管理ドメインに存在する vspace の情報は，DSR(Domain Space Resolver) が管理する．各 INR は，自身が管理する vspace をソフトステートで DSR に対して通知する．また，アプリケーションから自身が管理していない vspace に属するサービスに対する要求があった場合，DSR に問い合わせる．複数の vspace は統合可能で,．各 INR が全体の名前空間の一部を管理するため，それぞれの INR が保持する名前情報は分散される．また，vspace

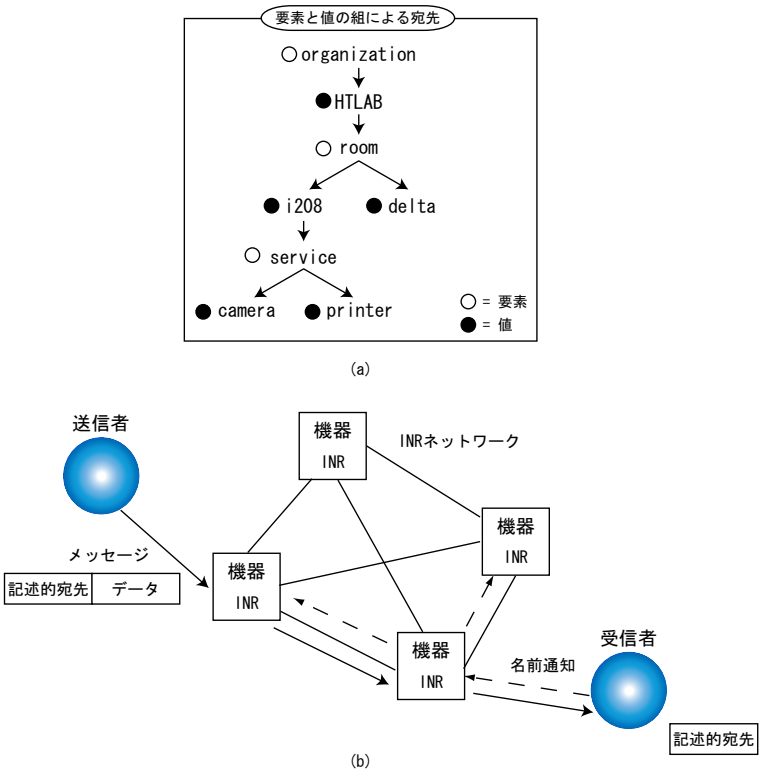


図 6.2: Intentional Naming System の概要

を管理する INR の数は必要に応じて変えられるため，検索要求が増えすぎたり，更新要求が増えすぎたりした場合に，INR を増やす，または減らすことで負荷を分散できる．

INS を拡張した INS/Twine は，記述的宛先をより効率的に管理するために分散ハッシュテーブルを利用している．記述的宛先に含まれる木は，下位の枝ごとに分割される．単純に要素と値の組で分割しないのは記述的宛先の記述性を保持するためである．分割した名前はハッシュされ，複数の INR に分配される．システムは分割された枝と元の名前の対応表を新たに保持する．サービス検索要求を渡された場合，この対応表に合わせて適切な INR から枝を取得する．

INS では，単純で記述的な宛先を提供している．また，INR が動的にネットワークを構築するため，管理コストも低く，効率性を実現する機構も提案している．本研究との相異点は，INS がメッセージ配送にフォワーディングを用いているのに対して，本研究では共有空間を用いている点である．これにより，サービスの追加と削除により動的に対応できる．

6.3 Internet Indirection Infrastructure

米 U.C. Berkeley で開発されている I3(Internet Indirection Infrastructure) は，サービスと識別子を対応付け，その識別子を介して通信を行うことで間接型通信を実現するメッ

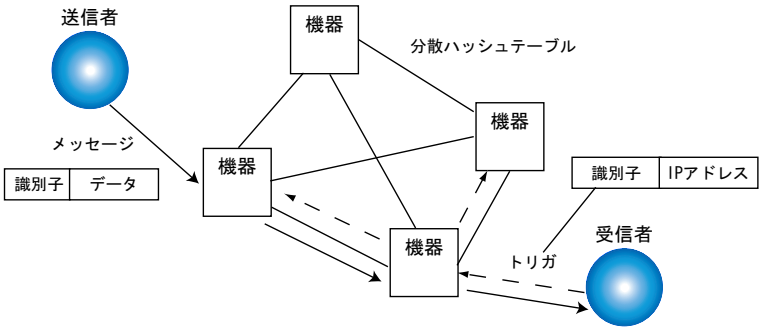


図 6.3: Internet Indirection Infrastructure の概要

セージ配送システムである．サービスの識別子は，本研究と同様分散ハッシュテーブルを用いて，複数の機器に分配される．分散ハッシュテーブルは複数の機器が動的に構築する．

メッセージ受信者は，まずトリガと呼ばれる識別子と IP アドレスの組を I3 に挿入する．メッセージ自体は識別子とデータの組で構成され，送信のため I3 に挿入されるとトリガが検索され，トリガと一致した場合にそこに含まれる IP アドレスへと送信される．マルチキャスト通信を実現するには，複数のサービスが同一の識別子を持ったトリガを I3 に挿入する．エニーキャスト通信を実現するには，複数のサービスが先頭の k ビットが等しい識別子を持ったトリガを I3 に挿入する．残りのビット列を利用して一つの機器が撰択される．また，トリガを入れ子にすることで，メッセージのルーティングを指定できる．これにより，複数のサービスの組み合わせを指定できる．図 6.3 に I3 の概要を示す．

識別子の生成方法は指定されていないが，上位のシステムで本研究同様サービスの記述をハッシュすることでも生成できる．また，分散ハッシュテーブルの利用により管理の容易さと効率も実現される．INS 同様，本研究との相異点は，メッセージの配送にフォワーディングを用いるか共有空間を用いるかである．

6.4 本章のまとめ

分散コンピューティング環境で共有空間を提供する JavaSpaces は，主にサービスの検索を共有空間を用いて行う．メッセージの送受信は Java RMI を用いて行われる．また，サービスの記述的宛先としてタプルを用いるが，本研究ではより単純性の高い配列を用いる．INS は，木構造を用いた記述的宛先を分散したサーバで管理する．本研究とは異なり，メッセージをサービスにフォワーディングするため，一定期間前に送信されたメッセージを取得できない．I3 は分散ハッシュテーブルを用いて記述的宛先を分配するが，INS 同様，共有空間を提供しない．

第7章　結論

7.1　今後の課題

7.1.1　セキュリティ

今回の設計，実装では，SBD の基本的な機能を重視したため，セキュリティについては考慮しなかった．しかし，任意のサービス同士の通信を考慮する場合，メッセージ送信者の認証とメッセージの暗号化が重要になる．ユニキャスト通信でメッセージ送信者の認証を行う場合は，メッセージの送信者と受信者が鍵を共有する．その鍵で暗号化した認証情報をメッセージに付加することで，通信者同士の認証が可能となる．しかしメッセージの受信者が複数存在する通信で同様の手法を用いると，受信者が独自にメッセージを作成し，別の受信者に送信者と偽ってメッセージを送信できる．送信者の秘密鍵を使ってメッセージに認証情報を付加する非対称鍵方式もあるが，認証情報の暗号化と復号化に時間がかかる．従って，このような場合は独自の認証方法が必要となる．

7.1.2　より柔軟な検索

今回の設計と実装では，記述的宛先の名前解決方法として完全一致を用いた．しかし部分一致を用いることでより柔軟な検索が可能となる．これを実現する際の問題は，ハッシュされた値を用いての部分一致が困難な点である．一つの方法として，記述的宛先を構成する配列の各要素を異なる組み合わせごとにハッシュし，もとの記述的宛先の中間機器への参照と対応付ける方法が考えられる．しかしこのような方法は通信ミドルウェアをより複雑にし，効率性が失われる可能性があるため，注意深く設計する必要がある．

7.2　まとめ

本論文では，ユビキタスコンピューティング環境において現在の通信システムで問題となる記述性の低い宛先と直接的な通信を解決するため，記述的間接型通信を提案した．同通信モデルは，複数の属性から成る記述性の高い宛先に対して，任意の数のサービスを対応付けることで，機器数の増加やその可用性の動的な変化に対応する．

記述的間接型通信を実現する際には宛先の構造，その名前解決方法，および効率的な名前解決方法を考慮する必要がある．また，サービスの登録，メッセージの配送，そして記述的宛先およびメッセージの管理方法を考慮する必要がある．ユビキタスコンピューティ

ング環境における通信ミドルウェアはプログラマに単純なプログラミングインタフェースを提供し，動的な環境に適応する必要がある．また，管理の容易さや効率性も実現しなくてはならない．

本研究で設計，実装したSBD通信ミドルウェアは，文字列の配列で表される単純な記述的宛先を提供した．また，複数のサービスをこの記述的宛先に対応付けることで，機器の動的な変化に対応できる．それぞれの機器上で動作する本ミドルウェアは，アプリケーション層で動的に分散ハッシュテーブルを構築することで，複雑な管理を必要とせず，記述的宛先の効率的な検索を実現した．

Java言語を利用してSBDのプロトタイプ実装を行った．各機器には一つのSBDが割り当てられ，機器の識別子としてそのIPアドレスをハッシュしたバイト列を用いる．各モジュールはUDPを用いてメッセージを交換する．プロトタイプの基本的な性能評価としてメッセージに含まれるデータ量とサーバ数に対する応答時間の変化を測定した．その結果，メッセージ配送の性能面で改善が必要なことが分かった．同プロトタイプ実装では簡潔なアプリケーションプログラミングインタフェースを提供している．

謝辞

本研究を進めるにあたり，御指導頂き，貴重な助言を下さっ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，本論文の副査として貴重な助言を頂いた慶應義塾大学大学院政策・メディア研究科助教授高汐一紀博士ならびに慶應義塾大学環境情報学部助教授楠本博之博士に深く感謝致します．

折りにふれ適切な助言を頂き，本論文の執筆を支援して頂いた慶應義塾大学大学院政策・メディア研究科助教授西尾信彦博士，慶應義塾大学徳田研究室の諸先輩方，後輩の皆様に対し，感謝の意を表します．

平成 15 年 2 月 22 日

松宮権太

参考文献

[1] Harter, A., Hopper, A., Steggles, P., Ward, A. and Webster, P.: The Anatomy of a Context-Aware Application, *International Conference on Mobile Computing and Networking (Mobicom '99)* (1999).

[2] Kahn, J., Katz, R. and Pister, K.: Next Century Challenges:Mobile Networking for 'Smart Dust', *Mobicom* (1999).

[3] Takashio, K., Aoki, S., Murase, M., Matsumiya, K., Nishio, N. and Tokuda, H.: Smart Hot-spot: Taking out Ubiquitous Smart Computing Environment Anywhere, *International Conference on Pervasive Computing (Pervasive2002) Demo Presentations* (2002).

[4] Sony, Matsushita, Philips, Thomson, Hitachi, Toshiba, Sharp and Grundig: Specification of the Home Audio/Video Interoperability (HAVi) Architecture (1998). http://www.havi.org/home.html.

[5] Universal Plug and Play Forum: Universal Plug and Play (UPnP) (1999). http://www.upnp.org.

[6] Sun Microsystems, Inc.: Jini Architecture Specification (1998). http://www.javasoft.com/products/jini/specs/jini-spec.pdf.

[7] Czerwinski, S., Zhao, B. Y., Hodes, T. D., Joseph, A. D. and Katz, R. H.: An Architecture for a Secure Service Discovery Service, *MobiCom'99* (1999).

[8] Sun Microsystems, Inc.: JavaSpace Specification (1998). http://java.sun.com/products/jini/specs.

[9] Stoica, I., Morris, R., Karger, D., Kaashoek, F. and Balakrishnan, H.: Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications, *SIGCOMM'01 Conference*, San Diego, California, ACM, pp. 149–160 (2001).

[10] Ratnasamy, S., Francis, P., Handley, M., Karp, R. and Shenker, S.: A Scalable Content-Addressable Network, *SIGCOMM '01 Conference*, San Diego, California, ACM (2001).

[11] Zhao, B. Y., Kubiatowicz, J. D. and Joseph, A. D.: Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing, Technical Report UCB/CSD-01-1141, UC Berkeley (2001).

[12] Rowstron, A. and Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems, *Lecture Notes in Computer Science*, Vol. 2218, pp. 329–?? (2001).

[13] Carriero, N. and Gelernter, D.: The S/Net's Linda Kernel, *Transactions on Computer Systems*, Vol. 4, No. 2, pp. 110–129 (1986).

[14] Adjie-Winoto, W., Schwartz, E., Balakrishnan, H. and Lilley, J.: The design and implementation of an intentional naming system, *Symposium on Operating Systems Principles*, Kiawah Island, SC, ACM (1999).

[15] Lilley, J.: Scalability in an Intentional Naming System, Master's thesis, Massachusetts Institute of Technology (2000).

[16] Balazinska, M., Balakrishnan, H. and Karger, D.: INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery, *Pervasive 2002 - International Conference on Pervasive Computing*, Zurich, Switzerland (2002).