博 士 論 文

ユビキタスネットワークシミュレーション環境の構築に関する研究

指導教官　丹 康雄 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

中田 潤也

2009 年 2 月 4 日

要 旨

近年，ホームネットワークやセンサネットワークなどのいわゆるユビキタスネットワークの研究が盛んに進められている．これらのネットワークでは比較的小規模のノードが無数に参加する点や，ノードが物理的な環境の一部として存在し，その環境から得られる情報をノード間で交換することが重要な動作となる点などが旧来のコンピュータネットワークとは異なっている．こうした特徴を持つユビキタスネットワークシステムを検証するためのテストベッドには従来の計算機ネットワークのテストベッドとは異なる機能が求められる．そこで，本論文ではPC ベースのクラスタ環境を利用し，数段階の抽象度においてユビキタスネットワークシステムにおけるノード，ネットワークのみではなく，周囲の環境を含むシミュレーションを行なう検証環境を提供することを目的とするRUNE （Real-time Ubiquitous Network Emulation environment）の提案を行なう．

目 次

第 1章

まえがき

センサネットワークやホームネットワークを始めとするユビキタスネットワークの研究がますます盛んに行われつつある．また，ホームネットワークや RFID を利用したネットワークは一般社会における生活基盤としての普及が始まっている．ユビキタスネットワークはいくつかの面で計算機ネットワークや電話系のネットワークとは異なる特徴を持っている．こうしたネットワークの設計においては明確な方法論が確立されているとは言い難く，他のネットワークにおける方法論からの類推や設計者の勘で決められる点も多い．これらのユビキタスネットワークでは，有線のネットワークと比べ，非決定的な挙動を示す無線ネットワークが利用される点や，ノードが不要な入出力ポートやメモリを持たないため動作中に何が発生しているのかを外界に対して通知することが難しいことも方法論の確立を阻害する要因の一つである．こうしたシステムの設計や検証を行う上でシミュレーションを用いて再現可能な環境下でシステムの全ての挙動を監視しながら動作させることができれば設計や開発を行う上で有用な情報を得ることが可能となり，さらにユビキタスネットワークの発展に寄与することが可能となる．

そこで，本研究ではユビキタスネットワークシステムのシミュレーションを行うためにシミュレーション環境に求められる要件を明らかにし，その要件を満たすシミュレーション環境の検討を行う．その際には，ユビキタスネットワークシステムの開発においてどのようなフェーズでシミュレーションが利用されるかも併せて検討を行い，できるだけ多くの場面で利用することが可能なシミュレーション環境の提案を行う．

本論文では，2 章でユビキタスネットワークの特徴，そのシミュレーションに必要とされる機能について検討を行う．3 章では本論文で提案を行うユビキタスネットワーク環境が動作基盤として利用している StarBED について述べる．次に 4 章で本論文で提案を行うユビキタスネットワークシミュレーション環境 RUNE の設計と実装について述べる．5 章は RUNE を利用して行われた応用例を述べ，6 章で RUNE を用いて行われた実験から得られた結果の評価を行い，7 章ではさらに高度なシミュレーションを行うために必要となる要件を明らかにする．

第 2章

ユビキタスネットワークシミュレーション

ユビキタスネットワークは従来の計算機ネットワークとは異なる特徴を持っている．これらの特徴は主に両者の利用形態に由来している．

ユビキタスネットワーク，特にセンサネットワークでは計算機ネットワークに比べ，格段に多くの比較的小規模のノードによって構成されるシステムであることが多い．また，計算機ネットワークでは，計算機，ソフトウェア，ネットワークといった構成要素の種類はそれほど多くないのに対し，ユビキタスネットワークでは，ハードウェア，ソフトウェア，ネットワークなどの様々な面で多様性を持っている．これは，計算機ネットワークを構成する要素では性能が不足していないことが重要であるのに対し，ユビキタスネットワークの構成要素に関しては性能が不足していないことと共に，電力など資源の消費が過剰ではないことも同時に求められるためである．ユビキタスネットワークでは，このようなトレードオフを考慮して設計を行うために多様な構成要素からの選択が行われる．この結果，ユビキタスネットワークはシステム毎に異なる，またはシステム内でも用途に応じて複数のハードウェア，ソフトウェア，ネットワークが利用されるヘテロジニアスネットワークの形態を持つことになる．

もう一点，ユビキタスネットワークにおいて特徴的なのは利用者や周囲の環境に依存した動作を行う点である．センサネットワークでは周囲の環境から得た情報を送受信することが主要な機能となり，ホームネットワークでは利用者との対

話が重要となる．このことは同時に，これらのネットワークの動作においてノードの位置情報の重要性が高いことも意味している．

2.1 ユビキタスネットワークシミュレーションの目的

計算機ネットワークを模倣するためのシミュレータはすでに様々な場面で広く利用されている．こうしたシミュレータは，プロトコルやソフトウェアの設計時やシステムが完成した後の検証時に用いられることが多い．これらの場面はISO12207が定める開発モデルにおけるソフトウェア設計フェーズとシステムテストフェーズに相当し，ソフトウェア構築フェーズの単体テストやソフトウェア結合フェーズの結合テストでシミュレータが用いられることは稀である．

ユビキタスネットワークでは計算機ネットワークのシミュレーションと同様にソフトウェア設計フェーズとシステムテストフェーズでのシミュレーションは重要となるが，こうした段階以外の各開発段階でも利用できることが望ましい．

その理由として，ユビキタスネットワークで動作するノードでは，計算機ネットワークのノードとは異なり，開発環境 (ホスト環境) と実行環境 (ターゲット環境) が異なるため，開発を行った環境でそのままソフトウェアを実行することができない点や，さらにユビキタスネットワークのノードの開発過程ではハードウェアとソフトウェアが並行して開発されるコデザインの手法が用いられることも珍しくなく，ソフトウェアのテスト時にターゲットハードウェアが存在しないこともあるためである．また，利用者や周囲の環境から得られる情報が動作を行う上で重要な意味を持つユビキタスネットワークでは，ソフトウェアの単体テストの場合でさえ，周囲の環境から得られる情報を入力することが必要であることもシミュレーションの重要性を増している．

また，結合テストの段階ではユビキタスネットワークのノード間の通信に用いられるネットワークもシミュレートする必要がある．この段階では実機を用いて検証を行うことも可能であるが，期待する規模のシステムを構成する数のノードを動作させることは困難な場合が多く，またユビキタスネットワークのノードは必要最低限の機能のみを持つことが多く，期待通りの動作が得られなかった場合にその原因を追求することには困難が伴う．シミュレーションを用いた場合には

対象となるノードの動作を完全に把握することが可能であるため，実機が動作する状態であってもより詳細なシステムの挙動を探るための手段としてシミュレーションを行うことが有用となる場合もある．この段階では様々なトポロジを用いてシミュレーションを行うことが可能となるよう，構成の変更が容易であることが望ましい．

こうした目的に十分に見合ったシミュレーション環境を構築することができれば，従来はハードウェアの設計，開発，ソフトウェアの設計，開発の各フェーズを順に行う必要があったユビキタスネットワークの開発工程をシミュレーションを用いて効率良く行うことが可能となる．

2.2 ユビキタスネットワークシミュレーションに対する要求

ユビキタスネットワークはハードウェア，ソフトウェア双方の面で，用途に応じて様々なアーキテクチャが利用される膨大な数のヘテロジニアスなノードから構成されるネットワークである．

こうしたシステムの検証を行なうためには，シミュレーション環境が大規模なシミュレーションにも対応可能なスケーラビリティを持つことが第一に求められる．

また，多様なアーキテクチャを持つノードの検証を行なう必要があるため，シミュレーション環境側でできるだけ多くのハードウェア，及びソフトウェアアーキテクチャに対応していることが望ましい．しかし，ユビキタスネットワークにおけるノードで利用される無数のアーキテクチャ，様々なネットワークに予め対応した環境を用意することは現実的ではない．そこで，次善の策として，シミュレーション環境自体があらゆるアーキテクチャに対応する柔軟性を持つ構造とする方法が考えられる．こうした構造を利用してエミュレートすべきユビキタスネットワークのコンポーネントにはノードで利用されるハードウェアとソフトウェア，ノード間の対話に利用されるネットワークなどがある．

また，センサネットワークはもとより，大半のホームネットワークがそうであるように，利用者を含む周囲の環境と対話を行ないながら動作を行なうシステムの場合には，ノードが自らの動作を決定するのに十分な量の情報を得るため，周

囲の環境のシミュレーションもシミュレーション環境内で行なわれることが必要となる．

これらの多数のコンポーネントが一つのシステムとして動作するユビキタスネットワークをシミュレートするためには，単一の計算機上でシミュレーション環境を構築することは現実的ではない．従って，複数の計算機を利用したクラスタ環境での実行が望ましいが，その場合でも，シミュレーションを構成する個々のコンポーネントを実装する際にクラスタ環境を意識させないことも重要である．具体的には，各コンポーネントが遠隔でエミュレートされていることを意識させない遠隔データアクセス機構や時刻同期の機能，他ノードに渡るシミュレーションを実行する負担を軽減する自動実行機構などを提供することが考えられる．

こうしたシミュレーション環境がユビキタスネットワークシステムの開発におけるあらゆる段階で有用となるためには様々な抽象度でのシミュレーションを行えることが望ましい．この機能を持つことで，例えば，原理試作の段階ではイベントドリブンシミュレータで用いられるような，振る舞いを記述したモデルによるシミュレーションを行い，開発が進むにつれ，最終的にはプロセッサエミュレータを用いた実機と寸分違わぬ挙動を利用してシミュレーションを行うということが可能となる．また，抽象度を上げることにより一般的にシミュレーション負荷は低減されるため，利用するノード数が同一でも，より大規模のシミュレーションを実行することが可能となる．

抽象度と同様に，様々な構成でのシミュレーションを実行可能な柔軟性を有することはシミュレーションを用いた検証に対して有利に作用する．この機能によって，少数のノードから構成されるシステムから始め，徐々に規模を大きくした際のシステムの挙動を観察するといったことが可能となる．これを実現するためには，シミュレーション対象の実装が構成によって影響を受けないことが必要である．

さらに発展的な利用法として，稼働中のシステムのスケーラビリティを検証する目的で，シミュレーション内のシステムが実世界のシステムと協調して動作することが可能となれば，例えば 100 台の実機を用いて構築されたシステムと，99,900 台のノードを用いたシミュレーションを組み合わせ，十万台規模のシステムをシミュレートするといったことが可能となる．これを可能とするためにはシミュレーション内のシステムと実世界で動作するシステムとのインタフェースの提供，シ

ミュレーションの実時間実行が重要となる．

以上から，ユビキタスネットワークのシミュレーション環境に求められる機能として，

1. 多数のノードから構成されるシミュレーションにも対応できる拡張性
2. 様々なネットワークをエミュレートできる機構
3. 多様なハードウェア/ソフトウェアアーキテクチャをエミュレートできる機構
4. 周囲の環境も同時にシミュレートできる機構
5. 複数の抽象度でのシミュレーションを可能とする機構
6. クラスタ環境を意識せずにシミュレーションを実行できる機能
7. シミュレーションの構成を容易に変更可能な機構

などが重要となる．

本研究では，大規模ネットワーク実証環境として開発された StarBED 上でユビキタスネットワークシミュレーション環境 RUNE を動作させることによって上述した要求を満たすシミュレーション環境の構築を行った．

2.3 既存研究例

これまでにいくつかのユビキタスネットワークを検証するための環境が提案されている．TOSSIM[1] は仮想環境で TinyOS アプリケーションを高精度にシミュレートすることを可能とする TinyOS シミュレータである．ATEMU[2] も同様に TinyOS アプリケーションのエミュレータで，様々な環境に対応する柔軟性を持っている．MobiNet[3] はクラスタを利用し，無線ネットワークの MAC 層をエミュレートすることによってユビキタスネットワークを検証するための環境である．MobiReal[4] は，端末利用者の行動や端末間の通信をシミュレートすることでモバイルネットワークの検証を行うことを可能としている．

これらのシミュレーションソフトウェア群と本論文で提案を行う RUNE との比較は 6.4 で行う．

第 3章

大規模ネットワーク実証環境 StarBED

RUNEがその動作基盤として利用するStarBEDはインターネット上のサービスを検証することを目的として開発された実証環境である．StarBEDでは，管理系と実験系の完全に隔離された2系統のネットワークを用いて830台のIA-32アーキテクチャベースのPCを接続している．この構成に加え，物理ノード上でVMを利用することによってさらに多数の論理ノードをエミュレートすることが可能であり，数千台規模のノードを有するネットワークの実験の遂行が可能となっている．StarBEDではこれらの実験ホストを相互に接続するネットワークをVLANを用いて論理的に分割することで，あらゆるトポロジーにおけるネットワークアプリケーションの実験を行うことを可能とし，さらに完全に分離された2系統のネットワークを持つことで，実験の管理を行うためのトラフィックが実験そのもののトラフィックに干渉することを回避している．また，StarBEDでは予め実験の内容に沿ったシナリオを記述しておくことで，これらのハードウェアを利用した実験のトポロジー生成，ノードのセットアップ，実験の遂行，実験結果の収集などを自動化する実験支援環境SpringOS [5]も提供されている．

表 3.1: StarBED のノード構成

グループ		A	B	C	D	E	F	G
モデル		NEC Express 5800						Proside Amaze Blast neo920
CPU		インテル® Pentium®III 1GHz					インテル® Pentium®4 3.2GHz × 2	AMD Opteron™ 2.0GHz
メモリ		512MB					2GB	8GB/4GB
HDD		30GB (ATA)		36GB (SCSI)	30GB (ATA)		80GB × 2 (SATA)	– –
N/W I/F 実験系	ATM	–	1	1	–	–	–	–
	FE	–	1	4	1	4	–	–
	GbE	1	–	–	–	–	4	1
管理系	FE	1	1	1	1	1	–	–
	GbE	–	–	–	–	–	1	1
ノード数		208	64	32	144	64	168	150
導入時期		2002 年 4 月					2006 年 4 月	2007 年 6 月

3.1 ノード群

StarBED ではクライアント装置 A 群から G 群の計 680 台の PC がシミュレーションノードとして利用されている．各群の構成は表 3.1 に示すとおりである．後述するとおり，これらのクライアント装置は管理系と実験系の 2 系統のネットワークに接続されるため，各ノードは実験系と管理系のネットワークに対してそれぞれ 1 個以上のネットワークインタフェースを持つ．

これらのノードは KVM 装置に接続されており，遠隔からの操作を行うことができる．また，Wake on LAN を利用した電源の投入，PXE によるネットワークブートなどの機能が実装されており，後述する SpringOS ではこれらの機能を用いた実験の自動化に対応している．

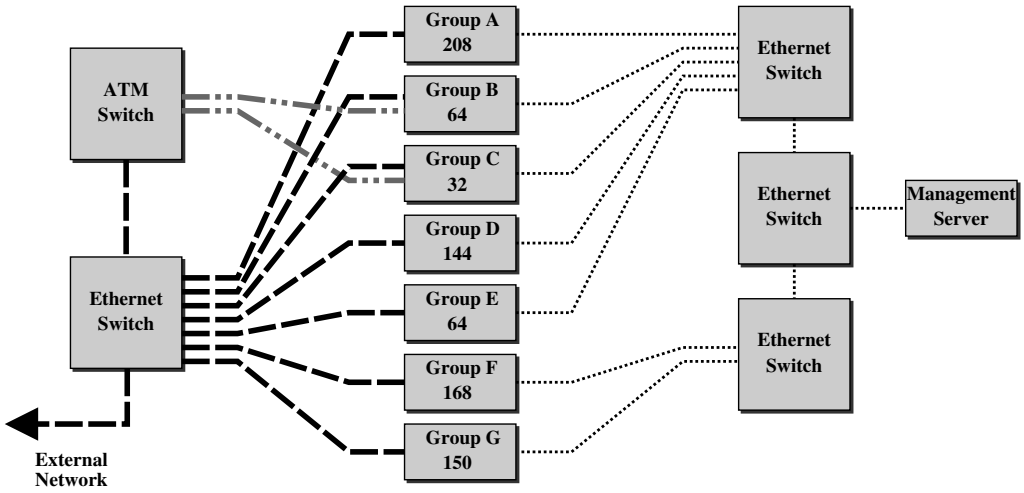

図 3.1: StarBED のネットワーク構成

3.2 ネットワーク

StarBED では ATM スイッチやイーサネットスイッチを用いてクライアント装置間の接続を行い，スイッチ間で VC や VLAN の設定を変更することによって任意のトポロジにおけるシミュレーションを可能としている．こうしたアプローチではスイッチの設定変更やスイッチの設定誤りなどによって一時的，または永続的にノードに対する到達性が失われる可能性がある．この問題を回避するため，StarBED のクライアント装置群は実験系と管理系の 2 系統の隔離されたネットワークに接続されている．管理系のネットワークインタフェースには静的 DHCP を用いてアドレスの割り当てを行い，常に到達性を確保している．また，管理系のネットワークを用いてシミュレーションの準備，制御や後処理を行うことでシミュレーション内のトラフィックが実験を制御するためのトラフィックの影響を受けることを回避することが可能となる．

StarBED のネットワーク構成を図 3.1 に示す．実験系は ATM スイッチ群とイーサネットスイッチ群から構成され，管理系ネットワークはイーサネットスイッチ群から構成されている．

3.3 実験支援ソフトウェア SpringOS

StarBED のような多数のノードからなる大規模実験施設を利用してシミュレーションを行うためには様々な機能が必要となる．最も重要な要素として，実験を予め決められた手順に従って自動で実行する機構が必要となる．人間が直接オペレーションを行って実験を行うことは不可能ではないが，その規模はせいぜい数十台程度で，それ以上の台数を用いた実験を手動で実行することは現実的ではない．また，こうした人間が全てのオペレーションを行う実験ではタイミングの制御が非常に難しく，再現性のある実験を行うことは難しい．さらに，StarBED のような大規模実験施設では単一の利用者が施設全体を占有して実験を行うことは多くの場合，非効率的であるため，必要な資源のみを利用する多数の利用者が共同で施設を利用することが望ましい．こうした利用形態では，施設側で資源管理を行い，利用者は許可された資源へのアクセスのみが可能とすることが重要である．

StarBED では SpringOS と呼ばれる支援ソフトウェアによってこれらの機能を実現している．SpringOS は単一のソフトウェアを指す呼称ではなく，StarBED における実験の実行を支援するソフトウェア群の総称である．SpringOS が提供する機能には，資源管理，ノード設定，ネットワーク設定，機器の電源管理，実験の実行，実験結果の収集などがある．

3.4 StarBED で行う実験の手順

SpringOS を用いて実行される実験の実行手順は以下の通りとなる．

- 実験シナリオの読み込み・解釈
- 資源の割り当て
- ノードの設定
- ネットワークスイッチの設定
- 実験シナリオの実行
- 実験結果の収集

SpringOS を利用することにより， StarBED では，これらの手順を自動で実行することを可能とすると同時に多数の利用者が同時に施設を利用することを可能としている．

これまでに StarBED を利用して，

- 大規模ネットワークの挙動解析シミュレーション
- P2P ネットワークのシミュレーション
- マルチキャスト通信のシミュレーション
- AS 間接続のシミュレーション
- TV 会議システムのシミュレーション
- ネットワークサービスの負荷実験
- ネットワークを介した映像配信実験
- モバイルネットワークの移動制御のシミュレーション
- 大規模ストリーミング配信のシミュレーション
- サーバ仮想化技術の検証シミュレーション

等をはじめとする様々なネットワークに関するシミュレーションが行われてきた．

第 4章

ユビキタスネットワークネットワークシミュレーション環境RUNE

RUNE は StarBED のようなクラスタ環境においてユビキタスネットワークのシミュレーションの実行を支援するプラットフォームである．

ユビキタスネットワークシステムのシミュレーションは，大別して，シミュレーション固有のロジック，シミュレーション一般に共通のロジック，シミュレーションの制御を司る部分から構成されている．RUNE はこのうち，シミュレーション一般に共通のロジックとシミュレーションの制御を司る部分を提供し，シミュレーションの実行を行う利用者が用意したシミュレーション固有のロジックと協調して動作する．RUNE では，シミュレーションの実行を司る部分を RUNE core，シミュレーションに共通するロジックであり，構成要素のシミュレーションを支援する部分を RUNE tools と呼ぶ（図 4.1）．

4.1 RUNE core

RUNE core は以下で説明を行う Space と Conduit という概念を用いて実装されたシミュレーションの対象を RUNE master と RUNE manager の協調によって実行する機能を提供している．RUNE を用いてユビキタスネットワークのシミュレーションを行う場合にはシミュレーション対象を RUNE 上の実行の単位である Space としてコンパイルする．Space 同士は RUNE が提供する機能である Conduit を利

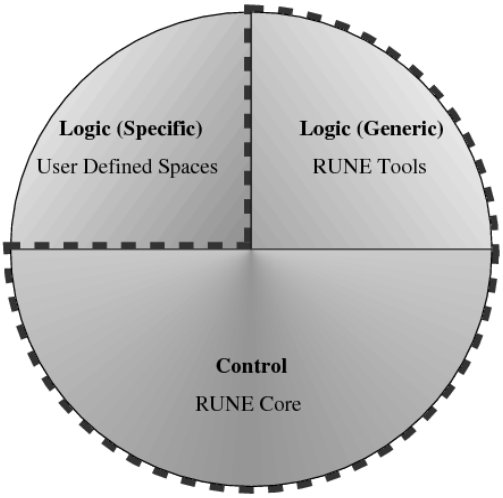

図 4.1: RUNE の構成

用して IP アドレスと独立した Space ID を宛先とした通信を行うことができ，これによってシミュレーション対象の実装がシミュレーションの構成に影響されることを防いでいる．各ノード上の RUNE manager はシミュレーション毎に唯一実行される RUNE master からの指示に従いながらシミュレーションの実行を行う．以下では，こうした RUNE の機能について説明を行う．

4.1.1 Space & Conduit アーキテクチャ

クラスタ環境において多くのコンポーネントを利用したシミュレーションを行うためには，各コンポーネントが独立して動作するための実行の単位とそれらの間で通信を行うための手段が不可欠である．これらは，通常のオペレーティングシステムにおけるプロセスとプロセス間通信に相当する．一般的なクラスタ環境でこうしたシミュレーションを行う場合，シミュレーションの実装段階でクラスタ環境を意識した実装を行う必要がある．クラスタ環境での実装に必要とされる特有の概念を適切に隠蔽する機構を提供することによって比較的容易に分散シミュレーションを実行することが可能となる．

RUNE では，こうしたシミュレーション対象を実行する単位として Space という枠組みを，Space 間の通信手段としては Conduit を用意している．利用者はシミュレーションを構成するコンポーネントを Conduit を用いて通信を行う Space

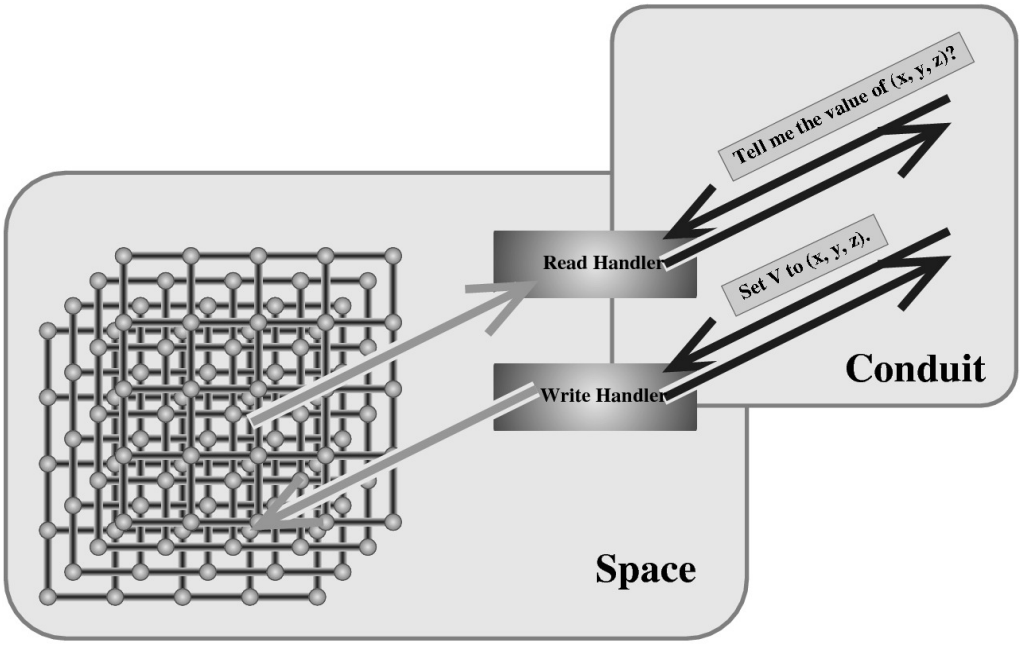

図 4.2: Space と Conduit

の形で実装する（図 4.2）．RUNE は多数の Space を並行に実行し，Conduit を介した通信の仲介を行う．Conduit はクラスタ環境における透過的なアクセスを提供しており，Space を実装する際に Conduit を通じた通信の相手先が同じノード上で動作しているのか，それとも他のノード上で実行されるのかを意識する必要はない．また，Conduit を介した通信を行う際には通信の相手先を指定するために IP アドレスを用いる必要はなく，シミュレーションを構成する Conduit に一意に割り当てられる Conduit ID を利用することができる．

Space は後述する RUNE manager のプロセス内のスレッドとして実行され，入出力イベントに結びつけられたコールバック関数は Space の実行スレッドからではなく， RUNE manager のスレッドから呼び出される．複数の Space が割り当てられたノードで動作する RUNE manager は複数のスレッドを並列して実行することになる．これらのスレッドは初期化の処理中に RUNE manager によって生成が行われる．この際，同時に 定義ファイル で定義された Conduit に対応する TCP コネクションが RUNE manager 間で確立される．

Space と Conduit を用いたシミュレーション構成の概念図を図 4.3 に示す．この

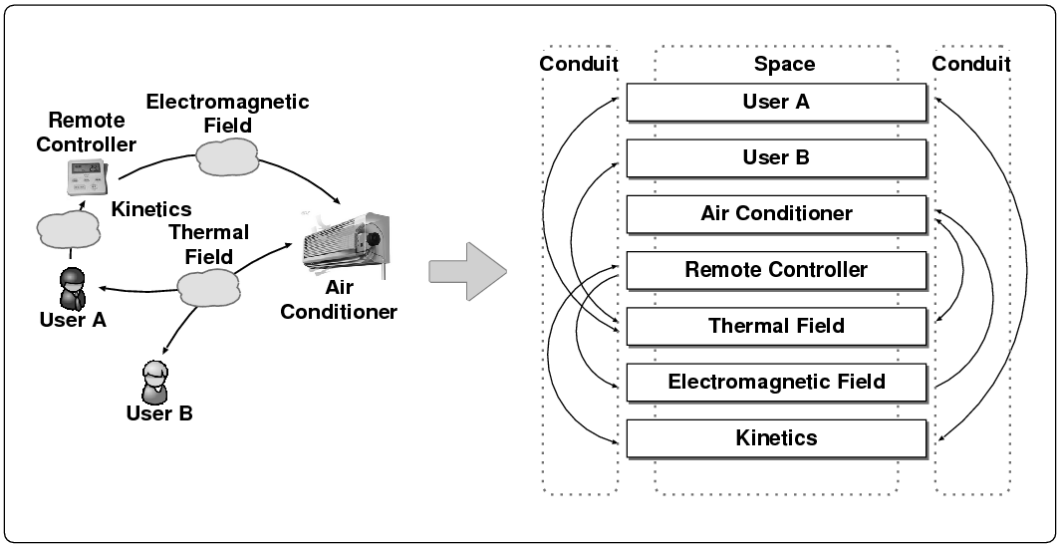

図 4.3: Space と Conduit を用いたシミュレーションの概念図

図ではユーザー A，ユーザー B，エアコンディショナ，リモートコントローラ，温度場，電磁気場，力学場がシミュレーション対象となり，それぞれ Space として実装される．また，温度場とエアコンディショナ，ユーザーなどの Space 間で情報の伝達が必要であり，Conduit が利用されることを示している．

Space をどのように組み合わせてシミュレーションを構成するかを指定するためには図 4.4 に示したサンプルのような定義ファイルを用いる．

Space の実装モデルを設計するにあたり，シミュレーションに用いられる一般的な実装形態の調査を行った．その結果，シミュレーションに用いられる実装形態には，大別して物理シミュレーションなどで一般的な周期実行型と，ネットワークアプリケーションなどに見られる事象駆動型があること，そのどちらも初期化，メインループ（もしくはイベントループとイベント処理部），通信処理，終了処理から構成されていることが分かった．これらの点を踏まえ，Space の実装規約では，図 4.5 に示すように，初期化，周期実行部（イベント処理部），他の Space からの読み出しに対するコールバック部，他の Space からの書き込みに対するコールバック部，終了処理の各処理を記述し，それぞれの処理のエントリポイントを以下の形でソースコード内に記述することとした．

```
#include "runebase.h"

BGNSPACELIST
    SPACE(UserA,                192.168.0.1,    user.so)
    SPACE(UserB,                192.168.0.1,    user.so)
    SPACE(AirConditioner,       192.168.0.2,    ac.so)
    SPACE(RemoteController,     192.168.0.2,    rc.so)
    SPACE(ThermalField,         192.168.0.4,    tf.so)
    SPACE(EMField,              192.168.0.5,    emf.so)
    SPACE(Kinetics,             192.168.0.6,    knt.so)
ENDSPACELIST

BGNCONDUITLIST
    CONDUIT(UserA,              ThermalField)
    CONDUIT(UserA,              Kinetics)
    CONDUIT(UserB,              ThermalField)
    CONDUIT(AirConditioner,     ThermalField)
    CONDUIT(RemoteController,   EMField)
    CONDUIT(RemoteController,   Kinetics)
    CONDUIT(ThermalField,       UserA)
    CONDUIT(ThermalField,       UserB)
    CONDUIT(ThermalField,       AirConditioner)
    CONDUIT(EMField,            AirConditioner)
    CONDUIT(Kinetics,           UserA)
    CONDUIT(Kinetics,           RemoteController)
ENDCONDUITLIST
```

図 4.4: 定義ファイルのサンプル

初期化処理ではコンポーネント内で必要な変数領域をヒープに確保する．周期実行部（イベント処理部）では，シミュレーションの主要な処理を記述する．周期実行型，事象同期型のいずれのモデルにも適用できる汎用性を持たせるため，ある呼び出し周期毎に呼び出しを行うよう RUNE manager に依頼し，繰り返しの内部のみを記述する動作，関数から戻らないイベントループを記述し，常にイベントの発生を待つ動作のいずれを用いることも可能である．他の Space からの読み出しに対するコールバック部では，Conduit を通じて情報の取得を要求された際に必要となる処理を記述する．書き込みに対するコールバック部では，Conduit から情報が送られてきた際に必要となる処理を記述する．終了処理では，ヒープに確保した変数領域の解放処理などを行う．

```
entryPoints ep = {
    .init = myspace_init,    .step = myspace_step,
    .fin = myspace_fin,      .read = myspace_read,
    .write = myspace_write
};

void *
myspace_init(int gsid)
{

…

}

int
myspace_step(void *p)
{

…

}

void
myspace_fin(void *p)
{

…

}

void *
myspace_read(void *p, void *a)
{

…

}

void *
myspace_write(void *p, void *a)
{

…

}
```

図 4.5: Space の実装例

以上の規約に基づく実装の概念図は図 4.6 のようになる．図の左側が RUNE を用いない一般的なシミュレーション対象の実装である時，この実装を右側のように

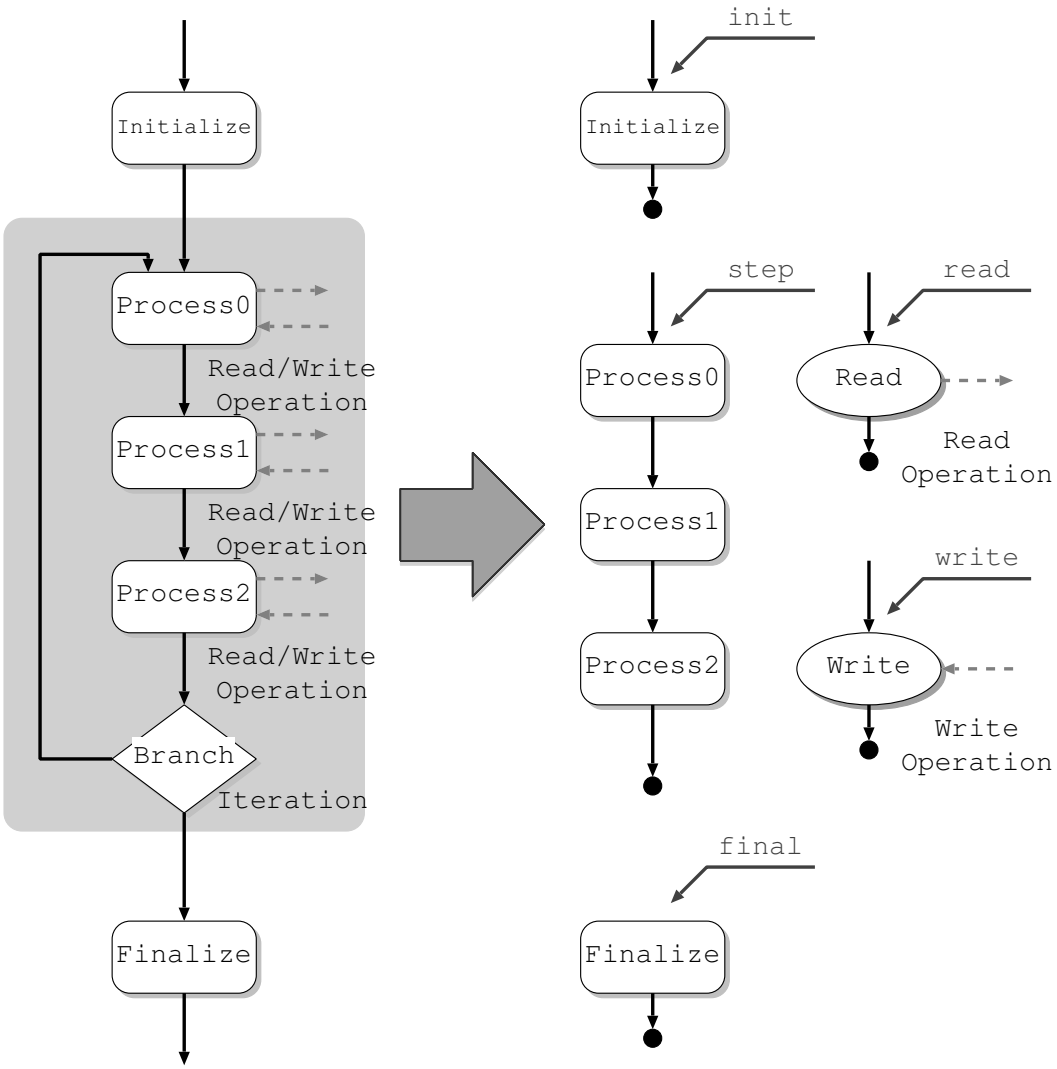

図 4.6: Space 実装規約に基づく実装

分割を行う．各エントリポイントは他の Space の実行と同期を取りながら RUNE manager から呼び出される．

このような規約に基づいて実装された Space は共有オブジェクトの形にコンパイルされ，実行時に動的リンクされ，RUNE manager のプロセス内で独立したスレッドとして実行される．この共有オブジェクトの形にコンパイルされた Space を Space オブジェクトと呼ぶ．一つのノード内で同一の Space が複数実行される場合であっても，Space オブジェクトは一度しかリンクされず，メモリ領域の節約と実行時のオーバーヘッド低減を図っている．

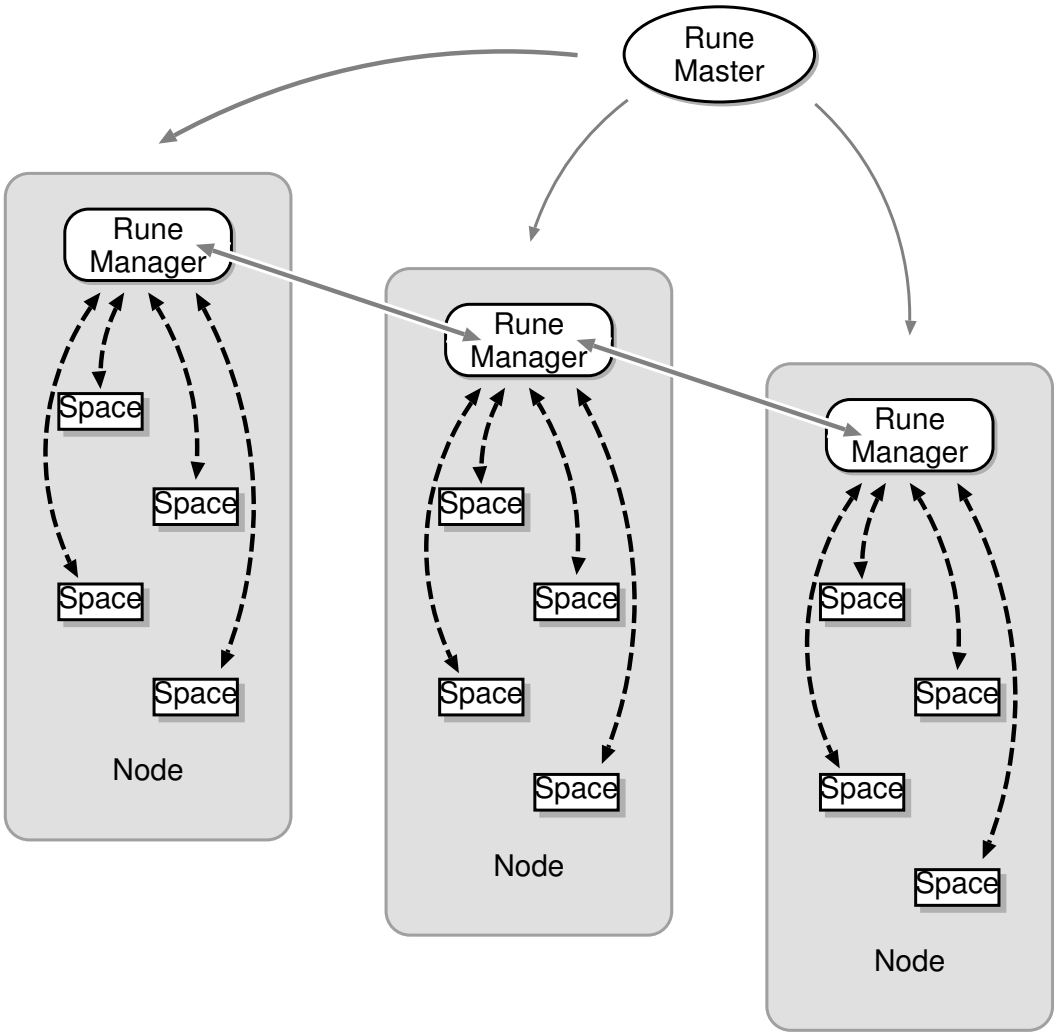

図 4.7: Space & Conduit 構造

4.1.2 RUNE master と RUNE manager による実行

RUNE では，図 4.7 のように，各ノード上で RUNE manager を実行し，その RUNE manager に対し，RUNE master がシミュレーションの構成を指示し，実行を行う．RUNE manager は，RUNE master からの指示毎に一つのプロセスを生成し，そのプロセスがノード上で実行される全ての Space オブジェクトの動的リンクを行い，呼び出しを行う．

RUNE manager にシミュレーションの構成を伝えるために RUNE master に与える定義ファイルは図 4.4 のようなフォーマットとなっている．BGNSPACELIST

と ENDSPACELIST の間でシミュレーションを構成する各Space に関して，Space 名，どのノード上で動作するか，どの Space オブジェクトによって実装されているかが指定されている．続いて，BGNCONDUITLIST と ENDCONDUITLIST の間では，どの Space 間で通信が行われるかを指定している．

この例は図 4.3 に対応した構成を 6 台のノード上で実行する場合の例である．Conduit を介した通信が IP アドレスを指定せずに Conduit ID を用いて行えることとあわせ，このようにシミュレーションの構成とシミュレーション対象の実装を切り離すことで，例えばエアコンディショナを 1 台追加するなど，シミュレーションの構成に変更があった場合であっても個々の Space の実装に影響が及ばない構造となっている．

RUNE master がこれらの情報を RUNE manager に通知すると，Space オブジェクトのリンクが行われ，各 Space の init が呼び出される．その後，RUNE master からの指示により各 Space の step の呼び出しが開始される．この呼び出しはいずれかの Space がシミュレーションの終了を宣言するまで周期的に行われ，その間，Conduit からの情報の流出入に応じて読み出し/書き込みのコールバック処理が適宜呼び出される．最後に final が呼び出され，シミュレーションの終了となる．

こうした一連の処理を行うにあたり，シミュレーションを実行する利用者が行うことは，シミュレーションを行う各ノードへの Space オブジェクトの配布，各ノードにおける RUNE manager の起動，RUNE master の実行の三点である．

4.1.3 Space と Conduit を用いたコード例

ここまでで述べた Space と Conduit を利用して簡単なコード例を以下に示す．

この例では，`write` から `reader` に対して 1 ずつ増加する 32 ビットの整数を送るというものである．

はじめに `write` と `reader` に共通のヘッダファイルを図 4.8 に示す．ここでは `write` と `reader` の間で用いられるデータ形式の定義を行っている．

```
/*
 * Copyright (c) 2007 NAKATA, Junya
 * All rights reserved.
 */

typedef struct {
    uint32_t len;
    int num;
} rwCondData;

typedef struct {
    condPacket packet;
    int num;
} rwCondPacket;
```

図 4.8: rw.h

次に reader のソースコードを図 4.9 に示す．reader は定期的な処理を何も行わず， Conduit からの書き込みがあった場合にのみ，コールバック関数内でメッセージの表示を行う．

```
/*
 * Copyright (c) 2007 NAKATA, Junya
 * All rights reserved.
 */

#include <sys/types.h>

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <runecommon.h>
#include <runeservice.h>

#include "rw.h"

void *reader_init(int gsid);
int reader_step(void *p);
void reader_fin(void *p);
void *reader_read(void *p, void *a);
void *reader_write(void *p, void *a);
```

```
typedef struct {
    int gsid;
    rwCondPacket response;
} readerstruct;

entryPoints ep = {
    .init = reader_init,
    .step = reader_step,
    .fin = reader_fin,
    .read = reader_read,
    .write = reader_write
};

void *
reader_init(int gsid)
{
    readerstruct *p;

    if((p = (readerstruct *)malloc(sizeof(*p))) == NULL)
        return NULL;
    p->gsid = gsid;
    return p;
}

int
reader_step(void *p)
{

    return 0;
}

void
reader_fin(void *p)
{
    readerstruct *s = getStorage(p);

    free(s);
}

void *
reader_read(void *p, void *a)
{

    return NULL;
}
```

```
void *
reader_write(void *p, void *a)
{
    localSpaceList *l = p;
    condPacket *packet = (condPacket *)a;
    rwCondData *data = (rwCondData *)&packet->data;
    readerstruct *s = getStorage(p);

    printf("|READER|\t %s received write request from space %d: %d\n",
      l->name, ntohl(packet->sgsid), ntohl(data->num));
    s->response.num = htonl(ntohl(data->num) * 2);
    s->response.packet.data.len = htonl(sizeof(int));
    return &s->response;
}
```

図 4.9: reader.c

図 4.10 が writer のソースコードである．writer は reader とは異なり，step() において定期的にメッセージを送信する処理のみを行い， Conduit からの受信に対するコールバック関数では何も行わない．

```
/*
 * Copyright (c) 2007 NAKATA, Junya
 * All rights reserved.
 */

#include <sys/types.h>

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <runecommon.h>
#include <runeservice.h>

#include "rw.h"

void *writer_init(int gsid);
int writer_step(void *p);
void writer_fin(void *p);
void *writer_read(void *p, void *a);
void *writer_write(void *p, void *a);
```

```
typedef struct {
    int gsid;
    int num;
} writerstruct;

entryPoints ep = {
    .init = writer_init,
    .step = writer_step,
    .fin = writer_fin,
    .read = writer_read,
    .write = writer_write
};

void *
writer_init(int gsid)
{
    writerstruct *p;

    if((p = (writerstruct *)malloc(sizeof(*p))) == NULL)
        return NULL;
    p->gsid = gsid;
    p->num = 0;
    return p;
}

int
writer_step(void *p)
{
    int i;
    localSpaceList *space = (localSpaceList *)p;
    writerstruct *s = getStorage(p);
    rwCondData *response, request;

    for(i = 0; i < space->nconds; i++) {
        request.num = htonl(s->num);
        request.len = htonl(4);
        printf("|WRITER|\t sending %d to conduit %d...\n", s->num, i);
        if((response = (rwCondData *)runeWrite(p, i, (condData *)&request))
          == NULL) {
            RUNE_INFO("failed to send conduit message\n");
            return -1;
        }
        printf("|WRITER|\t %s received write response from space %d: %d\n",
          space->name, i, ntohl(response->num));
        if(++s->num > 100)
            return -1;
    }
    releaseExec();
    return 0;
}
```

```
void
writer_fin(void *p)
{
    writerstruct *s = getStorage(p);

    free(s);
}

void *
writer_read(void *p, void *a)
{

    return NULL;
}

void *
writer_write(void *p, void *a)
{

    return NULL;
}
```

図 4.10: writer.c

3 台のノードを用いて 1 つの writer と 4 つの reader が通信を行う場合の定義ファイルは図 4.11 のようになる．シミュレーションの構成を変更する場合はこの定義ファイルの変更を行うだけで良く，実装を変更する必要はない．

```
#include "runebase.h"

BGNSPACELIST
        SPACE(writer,   192.168.0.11,   writer.so)
        SPACE(reader0,  192.168.0.11,   reader.so)
        SPACE(reader1,  192.168.0.12,   reader.so)
        SPACE(reader2,  192.168.0.12,   reader.so)
        SPACE(reader3,  192.168.0.13,   reader.so)
ENDSPACELIST

BGNCONDUITLIST
        CONDUIT(writer, reader0)
        CONDUIT(writer, reader1)
        CONDUIT(writer, reader2)
        CONDUIT(writer, reader3)
ENDCONDUITLIST
```

図 4.11: reader と reader が通信を行う場合の runedefs.h

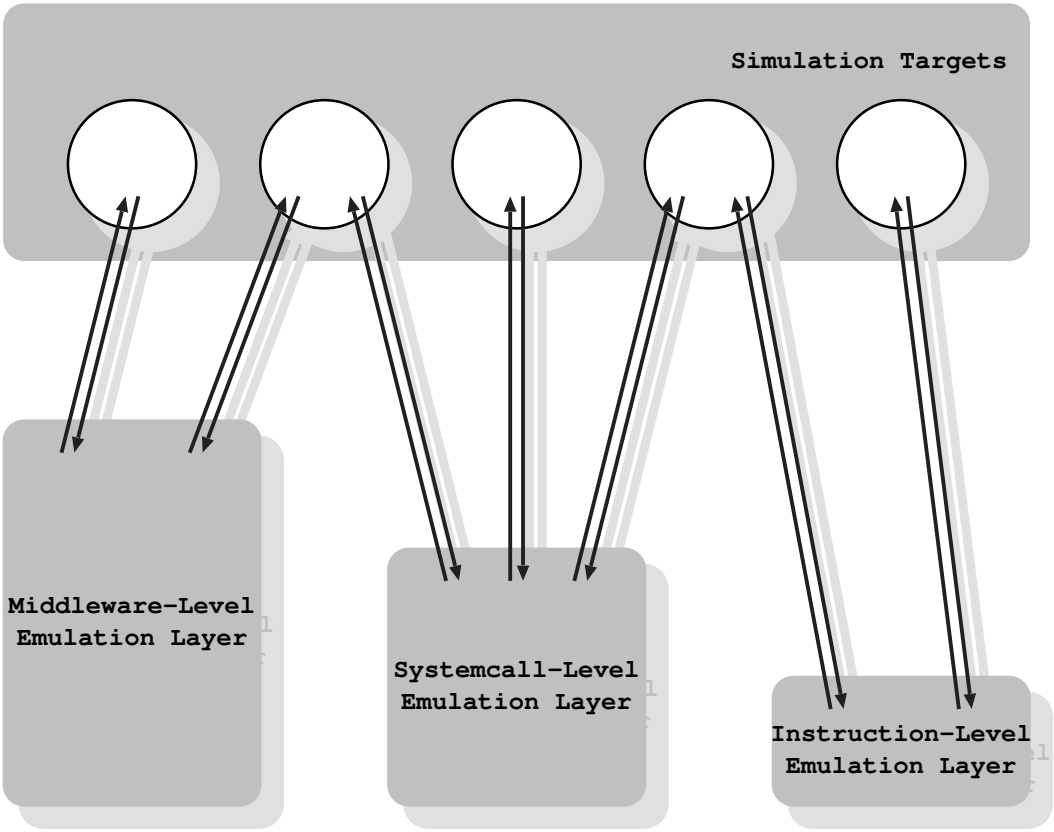

図 4.12: マルチレベルエミュレーションレイヤ

4.1.4 マルチレベルエミュレーションレイヤ

ユビキタスネットワークのシミュレーションでは，初めは要素の振る舞いのみを実装したモデルによって大まかなシミュレーションを行い，開発が進むにつれ，より精密なシミュレーションを行いたいという要求がある．こうした要求に対応するためにRUNEではシミュレーション対象を要求される抽象度で実行することを支援するマルチレベルエミュレーションレイヤを提供している (図 4.12)．マルチレベルエミュレーションレイヤでは，ミドルウェアAPI，OSのシステムコール，プロセッサのインストラクションの各レイヤを提供しており，Spaceがこれらのレイヤ上で動作することが可能となっている．

RUNE coreはこのマルチレベルエミュレーションレイヤに相当するインタフェースのみを提供しており，各レイヤの機能は後述のRUNE tools によって実現されている．

この機能を利用することにより，Space を実装する際に RUNE が提供する標準的なプロセッサやオペレーティングシステム，ミドルウェアの機能を個別に実装せずにシミュレーションを行うことが可能となる．また，独自のプロセッサやオペレーティングシステム，ミドルウェアの機能を利用したシミュレーションを行う場合には，マルチレベルエミュレーションレイヤのインタフェースを利用して実装を行うことが可能であり，これによって統一されたインタフェースによるアクセスが可能となる．

4.2 RUNE tools

RUNE では，シミュレーションの実行を制御する RUNE core の他，シミュレーションを支援するコンポーネントを提供している．代表的なものに多目的ネットワークエミュレータ QOMET [7]，各種のプロセッサエミュレータなどがある．

4.2.1 ネットワークエミュレータ

多目的ネットワークエミュレータ QOMET

QOMET（Quality Of transforMing Environments Testbed）は，イーサネット上で伝送されるパケットに対し，遅延，バンド幅制限，パケットロス率などのパラメータを適用することで様々なネットワーク上で伝送されるパケットの特性を再現するネットワークエミュレータである．QOMET は当初，IEEE 802.11 シリーズの無線 LAN を対象として開発されたが，その後 IEEE 802.3 イーサネットやアクティブタグ，IEEE 802.15.4 をベースとした Zigbee などに対象範囲を広げつつある．QOMET では，図 4.13 に示すように，ノードの位置関係やその間の環境を記述したシナリオをもとに，伝送品質がどれだけ劣化するかを示す ΔQ を求める第 1 フェーズと，ΔQ パラメータを dummynet，Netem，NIST Net，Chanel などのネットワークエミュレータが解釈可能なパラメータに変換し，リンク状態をエミュレートする第 2 フェーズの 2 フェーズの処理から構成されている．従って，QOMET が新たな無線ネットワークに対応するための作業は，メディア毎に距離

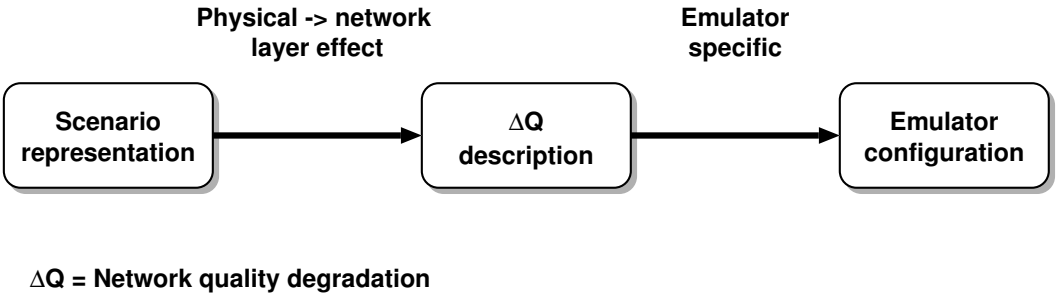

図 4.13: QOMET における 2 フェーズ処理

と減衰，受信電力と誤り率などの関係をもとに ΔQ を求める関係式を第 1 フェーズに導入するだけで完了する．

QOMET では，第 1 フェーズで求めた時間毎のパラメータを連続的にネットワークエミュレータに与えることで，シナリオで定義されたネットワーク状況の再現を行う．第 2 フェーズで利用可能なネットワークエミュレータとして，dummynet，Netem，NIST Net などの IP ネットワークエミュレータの他に，非 IP ネットワークで行われる通信に対してパラメータの適用を行うために StarBED2 プロジェクト内で開発が行われた Chanel [11] が利用可能となっている．

非 IP ネットワーク向け通信路エミュレータ Chanel

IP を用いて通信を行うネットワークにおける通信帯域，通信遅延，通信損失等の諸特性を模倣するネットワークエミュレータには dummynet [12] をはじめとして様々な実装が存在する．しかし，ユビキタスネットワークでは IP を用いないネットワークが用いられる場合も多い．そこで， RUNE では 非 IP 通信におけるネットワークの諸特性を模倣するために開発を行った Chanel (communication CHANnel Emulation Library) を利用する．この Chanel を QOMET と組み合わせて利用することにより，IP を用いない 無線ネットワークを利用した通信のエミュレーションが可能となる．さらに， Chanel を RUNE の Conduit とマッピングさせることでシミュレーション内の Space 間で非 IP 通信が行われるシステムのシミュレーションが可能となる．

Chanel は IP ネットワークにおけるネットワークエミュレータと同様に QOMET

の第 2 フェーズとして動作し，第 1 フェーズで求められた ΔQ をエミュレートすることによってネットワーク上で行われる通信の特性を模倣する．また， Chanel は必要に応じ，無線ネットワーク上におけるブロードキャスト通信のエミュレーションも行う．これは，有線ネットワークでのブロードキャストのように論理的に規定される範囲に対してパケットが配送されるのではなく，物理的な制約によって範囲が限定される無線ネットワークにおけるブロードキャストをエミュレートするため，必要に応じてブロードキャストパケットを複数のユニキャストパケットに分割することによってブロードキャスト通信の模倣を行うものである．

4.2.2 プロセッサエミュレータ

RUNE では，シミュレーション対象となるソフトウェアをバイナリのまま動作させるためにプロセッサエミュレータを利用する．ここでは，後述のアクティブタグを利用した歩行者追跡システムのシミュレーションで利用された PIC エミュレータについて述べる．

PIC エミュレータ libpic16f648

Microchip 社 [13] の PIC シリーズは広く利用されているマイクロコントローラで，制御用途に適応するよう，ソフトウェア，ハードウェアの双方の面で必要な機能のみを実装することで，低価格，低消費電力を実現している．近年，こうしたマイクロコントローラを応用したセンサネットワークノードも登場している．こうしたセンサネットワークは複雑な動作を行うため，実システムの稼働前に動作を検証したいという要望は大きい．そこで，PIC シリーズの中規模マイクロコントローラの代表格である 16F648 を対象に，実稼働前の検証に利用可能な精度で実時間動作を行うエミュレータの実装を行った．

Microchip 社の PIC 16F648A はハーバードアーキテクチャを採用した 8 ビット RISC プロセッサコア内蔵マイクロコントローラで，4,096 ワードプログラムメモリ，256 バイトデータメモリ，256 バイト E^2PROM メモリを内蔵している．内蔵オシレータの周波数は 4MHz で，外部からクロックを供給することによって最大 20MHz での動作が可能となっている．命令セットに含まれる命令のうち，2 サイク

ルを要する分岐等一部の命令を除き，大半の命令が 1 サイクルで実行される．1 サイクルは 4 クロックのため，外部オシレータ使用時には理論上秒間最大 5,000,000 命令を実行することが可能となっている．

こうした特徴を持つ PIC 16F648A を IA-32 アーキテクチャの PC 上でエミュレートするプロセッサエミュレータの開発を行った．

開発にあたっての主な要件は

- 全てのインストラクションに対応する
- Timer 0/Timer 1/Timer 2 をトリガとした割り込みに対応する
- I/O ポート A/B への入出力に対応する
- 複数のインスタンスの同時実行に対応する
- サイクルアキュレートに実行を行う
- 命令の実行レート，割り込みの頻度等の統計情報の取得機能を提供する

等である．

エミュレータコアの他に，エミュレータ内のプログラムメモリ上に Intel HEX フォーマットで記述されたコードを展開するローダの実装も行った．

このエミュレータの詳細は付録 A で述べる．

OpenRISC エミュレータ ORE

このエミュレータは OpenCores プロジェクト [14] にて開発が行われている OpenRISC 1200 [15] のエミュレーションを行う．通常メモリ空間，I/O 空間，割り込み処理を含むほぼ全てのプロセッサ機能を実装し，実プロセッサの 16MHz 相当の速度でのエミュレーションが可能となっている．また，複数インスタンスの同時実行も可能となっている．

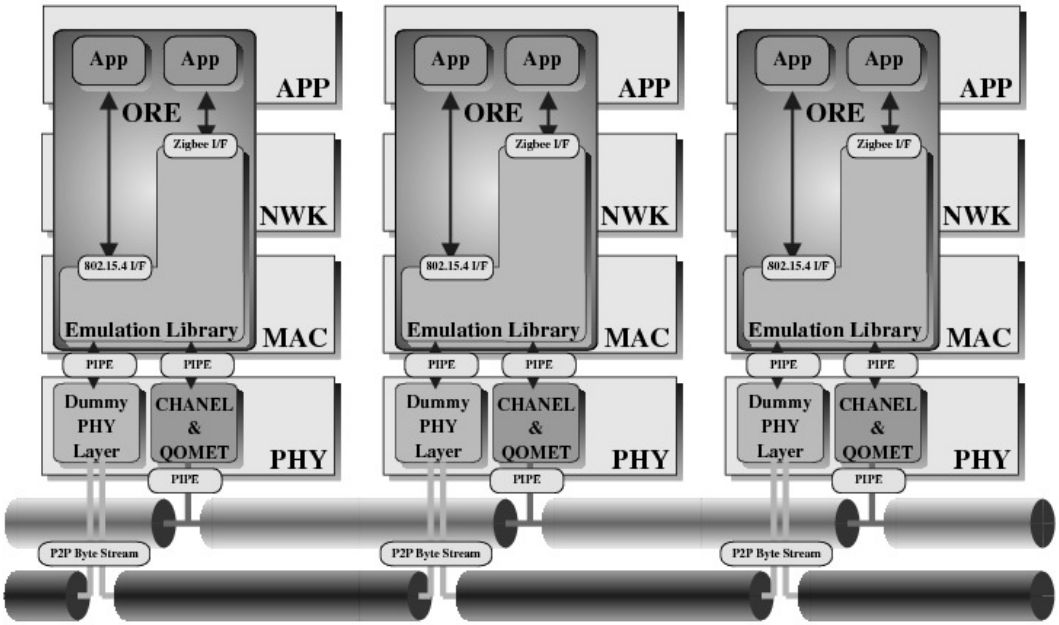

図 4.14: OpenRISC と IEEE802.15.4 のエミュレータを利用した JN-5139 エミュレーション

4.2.3 ミドルウェアエミュレータ

IEEE802.15.4 エミュレータ

このエミュレータは， Zigbee [16] が下位レイヤとして利用する IEEE802.15.4 にアクセスするためのライブラリをエミュレートする．

このエミュレータを利用することで IEEE802.15.4 の通信をエミュレートすることが可能となる．また，図 4.14 に示すように OpenRISC エミュレータ ORE と組み合わせて利用することにより，Jennic 社 JN-5139 評価ボード上で動作するアプリケーションのエミュレーションが可能となる．

Bluetooth エミュレータ

Bluetooth エミュレータは Bluetooth [17] を利用する機器を模倣する Space が利用する．本研究プロジェクトで開発を行った Bluetooth エミュレータは図 4.15 に示す Bluetooth プロトコルスタックのうち，ベースバンド層とリンクマネージャ

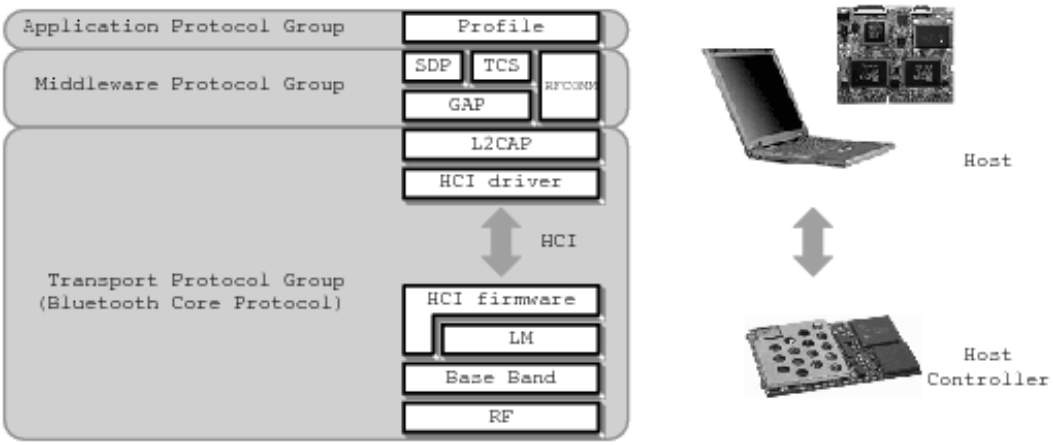

図 4.15: Bluetooth プロトコルスタック

層をエミュレートする．従って， Space からは HCI プロトコルを用いてアクセスを行うことが可能となっている．

エミュレータの機能を増強し， L2CAP 層までのエミュレーションを可能とする作業が進行中である．

その他の エミュレータ

ここまでに挙げた RUNE tools を構成するソフトウェア群の他，ホームネットワークシミュレーションのための以下のソフトウェアも開発が進められている．

- Echonet エミュレータ

  Echonet エミュレータはホームネットワークにおける共通プロトコルである Echonet [18] をエミュレートする．このレイヤを利用することにより，Echonet を利用して動作するアプリケーションの部分のみをシミュレートすることが可能となる．さほど精度を要求されないシミュレーション，単一ノード上で多くのノードをシミュレートしたい場合，アプリケーションそのものと Echonet プロトコルのいずれで問題が発生しているかの切り分けを行いたい場合などには有効である．

- DLNA エミュレータ

これは，主にホームネットワークにおけるマルチメディア転送に利用される DLNA [19] を，下位の UPnP [20] も含めエミュレートするものである．このレイヤを利用することにより，Echonet エミュレーションを利用する場合と同様の利益を享受することが可能となる．

- 住環境シミュレータ

  住環境シミュレータは，宅内の情報家電を含むシミュレーションを実行する際に必要とされる温度場，湿度場，音響場，力学空間等の物理環境をシミュレートし，情報家電をシミュレートする Space に対して提供する．現在，数値流体力学の手法に従った温度，湿度，照度等の実装を進めている．

第 5章

RUNEで行われた実験例

現在までに RUNE を利用した様々なシミュレーションが行われてきた．以下で代表的な応用例の説明を行う．これらのシミュレーションはいずれも FreeBSD 5.4-RELEASE が動作する StarBED のノードを用いて実行された．

5.1　無線センシングシステムのシミュレーション

このシミュレーションでは，図 5.1 に示すように，居住者が無線 LAN を用いて通信を行う PC を利用している住居において，同じく無線 LAN を利用し，エアコンディショナが遠隔温度センサからの温度情報をもとに動作を行う状況を想定してシミュレーション [8] を行った．もともとこのシミュレーションは RUNE を利用せず，1 台の PC 上で全体のシミュレーションを行っていたが，無線 LAN による通信の伝搬のエミュレーションと居室内の温度分布のシミュレーションを行う負荷が予想以上に大きく，実時間のシミュレーションを行うことが困難だったため，RUNE を利用し，5 台のノードを利用した分散シミュレーションへの移行を行った．

このシミュレーションでは，RUNE を利用する以前から利用されていたエアコンディショナ，温度センサ，居室内の温度場などのシミュレーション要素にそれぞれ若干の変更を加え，RUNE 上で動作する Space として利用した．これらの Space 間の情報の伝達のうち，図 5.1 の Access Point，User PC，Air Conditioner，Heat Sensor 間で無線 LAN を用いて行われる通信は IP による通信が行われる．これらの

表 5.1: 無線センシングシステムのシミュレーションで用いられた Space 群

Node Number	Space Name	Destination Space
1	Heat Sensor Dummynet Control	(Air Conditioner) , (User PC) , (Access Point) , Thermal Field
2	Air Conditioner Dummynet Control	(Heat Sensor) , (User PC) , (Access Point) , Thermal Field
3	User PC Dummynet Control	(Heat Sensor) , (Air Conditioner) , (Access Point)
4	Access Point Dummynet Control	(Heat Sensor) , (Air Conditioner) , (User PC)
5	Thermal Field Dummynet Control	Air Conditioner, Heat Sensor

Space 間の通信状況は QOMET を利用して無線 LAN の伝搬状況を表すパラメータを求め，その値をもとに Dummynet Control Space が dummynet の制御を行うことによってエミュレートした．Air Conditioner，Heat Sensor，Room1 Heat，Room2 Heat，Room3 Heat，Room4 Heat 間での温度情報の伝達は Conduit を利用して行われる．表 5.1 に本シミュレーションで利用された Space と接続先 Space の一覧を示す．括弧で囲まれた接続先は IP による通信を行い，そうでない接続先とは Conduit を利用した情報の伝達が行われる．

RUNE を用いて分散シミュレーションを行うことにより，単一 PC 上では困難であった各コンポーネントが実時間で動作するシミュレーションを実行することが可能となった．また，エアコンディショナと遠隔温度センサ間のトラフィックと居住者の PC が発生させるトラフィックとの干渉によるパケット損失や，それに伴うエアコンディショナ制御の変化を観測することができた．単純化のため，熱源の温度を 120 度とし，エアコンディショナの制御を目標温度に対するオンオフ制御とした場合のシミュレーション中の室温の変化を図 5.2 に示す．

5.2 モーションプランニングロボットのシミュレーション

このシミュレーションでは，無線 LAN を利用し，互いに通信を行いながら情報を共有し，互いに衝突を回避しつつできるだけ効率の良い経路で目的地に到着することを目標とするモーションプランニングロボットの実時間シミュレーション

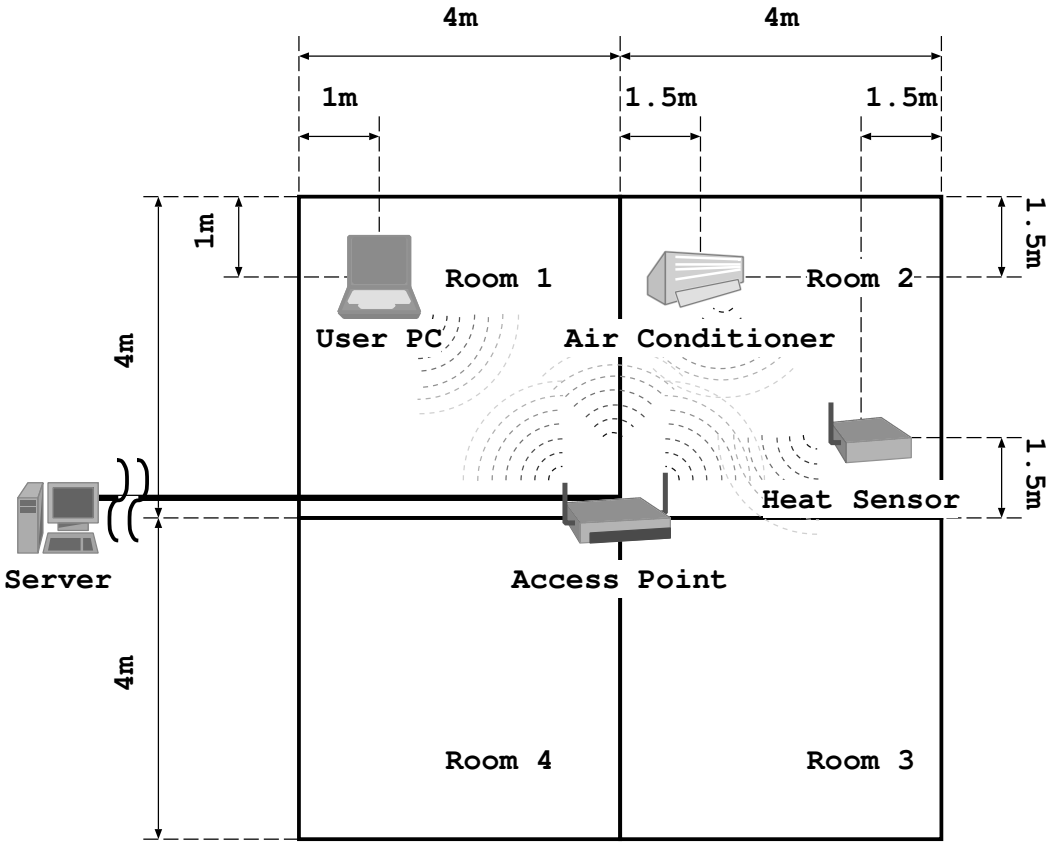

図 5.1: 無線センシングシステムのシミュレーション

[9] を行なった．このシミュレーションでは，無線 LAN インタフェース，視覚センサを持つロボットを実装した Space，QOMET を利用して求めた通信状況をもとに無線 LAN による通信をエミュレートするために dummynet の制御を行う Space，ロボットが移動を行う，障害物が配置された空間の管理を行うマップマネージャ Space を利用した．

このシミュレーションでは，10 台のロボットのシミュレーションから徐々にロボットの台数を増し，最終的に 100 台のノードを利用し，各ノード上で 4 台のロボットをシミュレートする構成で合計 400 台のモーションプランニングロボットの実時間シミュレーションを実行した．

さらに，シミュレーションの模様を実時間で可視化するビジュアライザ（図 5.3）も独立した Space として実装し，シミュレーションの過程を実時間で可視化することによって動作アルゴリズムの検証支援を行った．図 5.3 のビジュアライザはシ

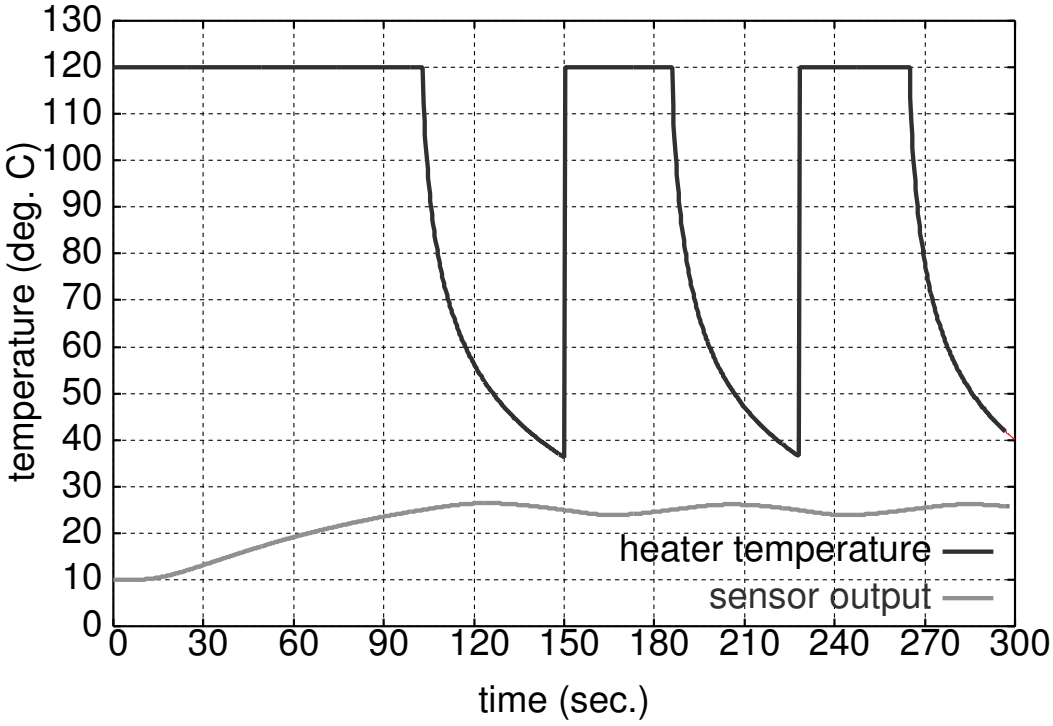

図 5.2: 熱源と室温の変化

ミュレートされたロボットのその時点での座標と移動目標，障害物の座標を示している．数字はロボットの ID を，矢印の根本がロボットの現在座標を，先が移動目標を示している．

表 5.2 にこのシミュレーションで利用した Space を示す．このシミュレーションでは，全ての Space 間の通信は IP を用いて行った．このシミュレーションでは Space 間の無線 LAN を用いた通信は，QOMET を利用して求めた伝搬状況を表すパラメータを Dummynet Control Space が dummynet に適用することによってエミュレートを行った．

5.3 アクティブタグを利用した歩行者追跡システムのシミュレーション

この例はアクティブタグを用いた歩行者追跡システムのシミュレーションである．RUNE を用いて実機では困難な大規模な実験を実現した．[10]．このシミュ

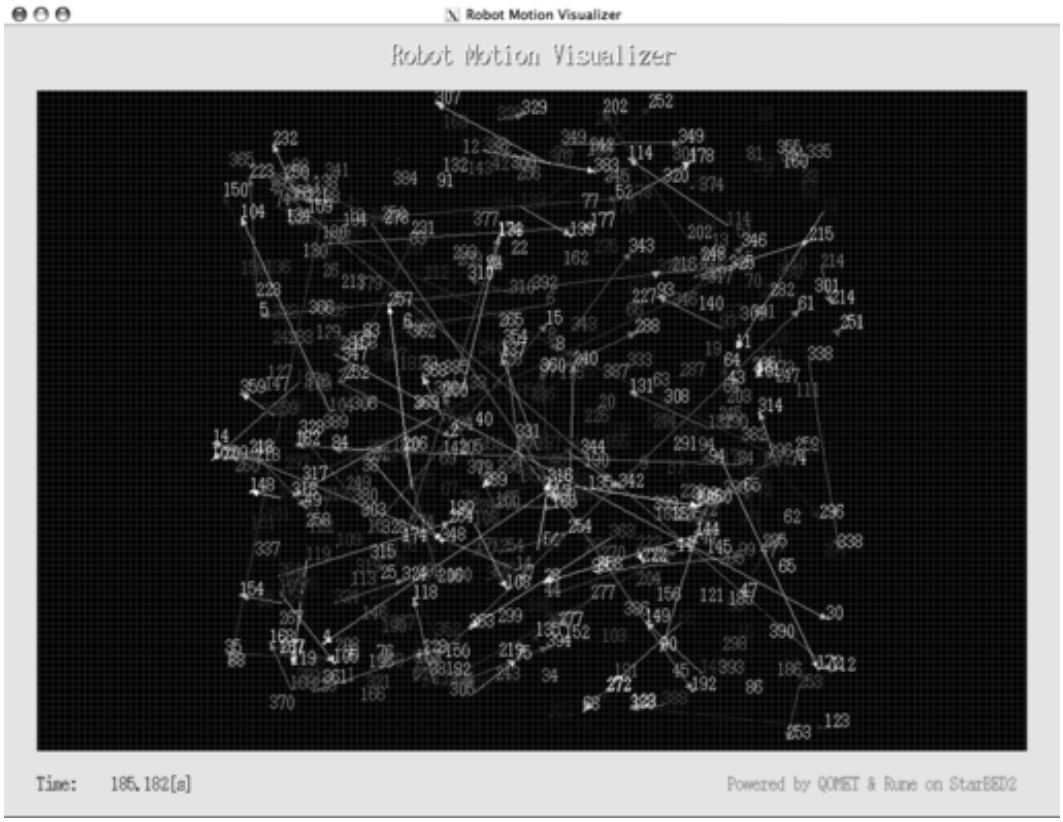

図 5.3: モーションプランニングロボットのビジュアライザ

レーションでは，タグ間の通信を実機に搭載されたトランシーバの特性に基づきエミュレートする無線エミュレータ，実機が利用するプロセッサをインストラクションだけではなく，割り込みや入出力を含め忠実にエミュレートするプロセッサエミュレーションなどを利用した．

5.3.1 歩行者位置推定システムの概要

今回シミュレーションの対象としたシステムは図 5.4 に示すように，設置位置が既知の固定タグ（図中の Fix），設置位置が既知でかつバックエンドシステムへのシリアル接続を持つゲートウェイタグ（図中の GW），歩行者が身に着けるモバイルタグの 3 種のタグを利用して歩行者が移動を行った軌跡の推測を行うシステムである．

各タグは，自発的に通信を行うアクティブ期間と，許可を受けたタグのみが通

表 5.2: モーションプランニングロボットのシミュレーションで用いられた Space

Node Number	Space Name	Destination Space
0	Map Manager	(Robot 1) , (Robot 2) , … , (Robot 400)
1	Robot 1	(Map Manager) , (Robot 2) , (Robot 3) , … , (Robot 400)
	Robot 2	(Map Manager) , (Robot 1) , (Robot 3) , … , (Robot 400)
	Robot 3	(Map Manager) , (Robot 1) , (Robot 2) , … , (Robot 400)
	Robot 4	(Map Manager) , (Robot 1) , (Robot 2) , … , (Robot 400)
2	Robot 5	(Map Manager) , (Robot 2) , (Robot 3) , … , (Robot 400)
	Robot 6	(Map Manager) , (Robot 1) , (Robot 3) , … , (Robot 400)
	Robot 7	(Map Manager) , (Robot 1) , (Robot 2) , … , (Robot 400)
	Robot 8	(Map Manager) , (Robot 1) , (Robot 2) , … , (Robot 400)
	Dummynet Control	
.		
.		
.		
100	Robot 397	(Map Manager) , (Robot 2) , (Robot 3) , … , (Robot 400)
	Robot 398	(Map Manager) , (Robot 1) , (Robot 3) , … , (Robot 400)
	Robot 399	(Map Manager) , (Robot 1) , (Robot 2) , … , (Robot 400)
	Robot 400	(Map Manager) , (Robot 1) , (Robot 2) , … , (Robot 399)
	Dummynet Control	

信を行い，通常は通信を行わないスリープ期間を繰り返している．アクティブ期間とスリープ期間の組が一周期となり，一周期には約 2.3 秒を要する．

モバイルタグと固定タグはアクティブ期間のうち，乱数で撰択を行った 1 スロットで広告パケットを送出する．ゲートウェイタグはアクティブ期間の最初と最後にスロットが予約されており，そのスロットで広告パケットの送出を行う．従って，モバイルタグと固定タグは約 2.3 秒に一度広告パケットを送出し，ゲートウェイタグは同期間内に二度広告パケットを送出することになる．

各タグは他のタグからの広告パケットを受信した記録をメモリ上に蓄積する．記録がメモリ上に蓄積されている状態では，広告パケットの特定のフィールドに蓄積されている記録の量を埋め込んだ状態で送出を行う．ゲートウェイがこの情報を受信すると，記録を蓄積しているタグに情報のアップロード許可を与える．この許可を受けると記録を蓄積しているタグはスリープ期間を利用し，ゲートウェイタグに対して記録のアップロードを行う．

ゲートウェイタグにアップロードされた情報はバックエンドシステムの位置推定エンジンで処理が行われ，歩行者の軌跡の推定が行われる．

タグの実機では，利用しているプロセッサのメモリ容量の制限から，仮想時間

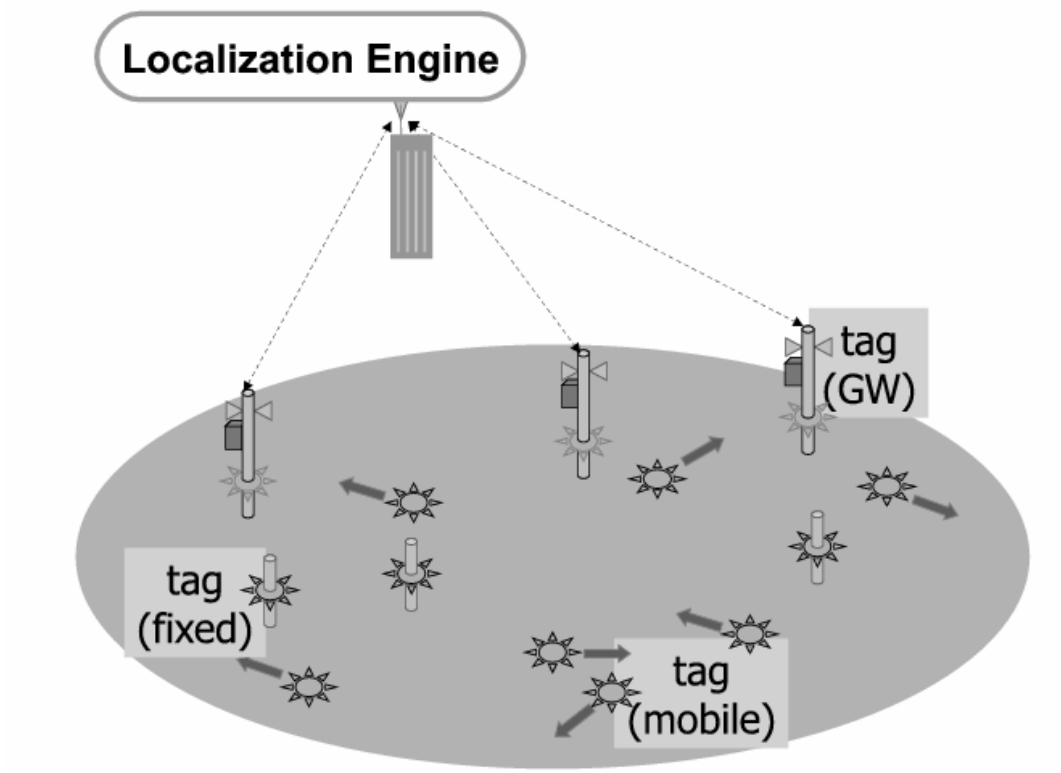

図 5.4: 歩行者位置推定システムの概要

マスクというパラメータに設定される一定の期間の間に情報を受信したタグを表すビットマップを用いて情報の蓄積を行っている．これにより，その期間における情報は一定量となる．この仮想時間マスクは短いほど情報の精度は高くなるが，その分メモリの消費が大きくなる．現在のタグの仕様ではこの期間の 16 回分の情報を蓄積することが可能となっている．メモリが一杯になる以前にゲートウェイタグにアップロードを行えなかった場合には，古い情報から消去され，新たな情報へと置き換えられていく．例えば，仮想時間マスクの長さを 32 秒とした場合，あるタグから広告パケットを受信したという記録が残っていた場合であってもそれが $n+0$ 秒であったのか $n+30$ 秒であったのかの区別はできない．仮想時間マスクの長さが 16 秒であった場合には，ある記録が $n+0$ 秒で発生したのか $n+30$ 秒で発生したのかの区別は可能となるが，仮想時間マスクの長さを 32 秒とした場合の倍の速度でメモリの消費が進むため，ゲートウェイタグと通信できる範囲に入る前に古い記録が上書きされる可能性が高くなる．

5.3.2 RUNEによるシミュレーション

RUNE を用いてこのシステムのシミュレーションを行うにあたり，StarBED2 プロジェクト内で開発が行われた QOMET，Chanel，Microchip®PIC 16LF648A エミュレータを利用し，以下のような各 Space によるシミュレーションの実行を行った．

- Chanel Space

  Chanel Space は通信回線エミュレータ Chanel を利用し，タグ間の無線通信をエミュレートしている．今回のシミュレーションでは，QOMET を利用してタグ間の無線通信の品質変化を求め，その結果を Chanel に与えることで無線通信をエミュレートしている．Chanel Space は各タグ毎に一つずつ利用され，他の全てのタグに対して Conduit を介した通信を行う．

- アクティブタグ制御 Space

  アクティブタグ制御 Space では，タグの動作を再現するために実機のファームウェアをタグの実機で利用されている Microchip®PIC 16LF648A エミュレータ上で動作させた．タグに対して送受信されるパケットを Chanel との間で転送するために Conduit が利用されている．

これらの Space を用いたシミュレーションの概念は図 5.5 のようになる．

今回のシミュレーションでは，全てのタグと歩行者の移動をシミュレートし，シミュレーション内でゲートウェイタグに対してアップロードされた情報を実際のシステムで利用されている位置推定エンジンに入力し，歩行者の位置情報を得た．従って，実機を用いた実験とシミュレーションの結果に差異が生じる場合の原因は，歩行者の移動，タグの挙動，タグ間の通信といった要素から発生していると考えられる．

5.3.3 実証実験の再現シミュレーション

この構成を用いてパナソニック株式会社によって行われた実機を用いた実証実験の再現シミュレーションを行った．実証実験は 3 台のゲートウェイタグ，4 台の

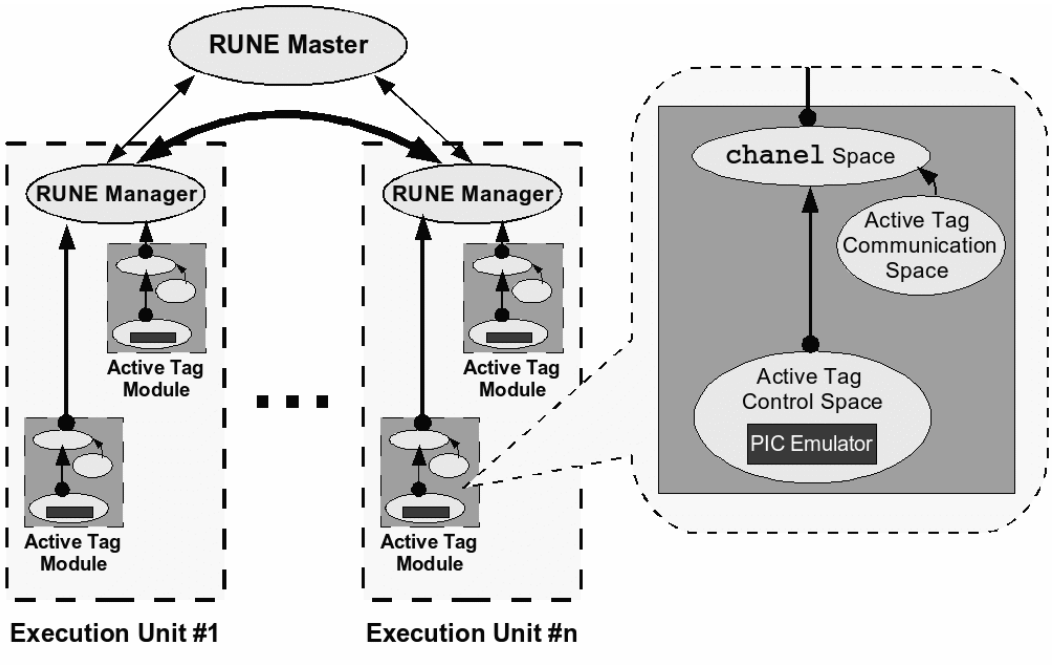

図 5.5: シミュレーションの概念図

固定タグを用いて 16 人の歩行者の位置推定を行う場面を想定して行われた．各歩行者は図 5.6 のような指示書に従い，モバイルタグを着用して歩行を行った．

この実証実験の再現シミュレーションを行うために我々はノード一台を用いてタグ一台のエミュレーションを行うこととした．従って，合計 23 台のノードを利用してシミュレーションが行われたことになる．このシミュレーションで使用した Space の一覧を表 5.3 に示す．このシミュレーションでは，全てのタグから送信されたパケットは Conduit を通じ，アクティブタグの通信をエミュレートする Chanel Space に送られる．Chanel Space でタグの位置関係に応じてパケットロスのエミュレーションが行われたパケットは再度 Conduit 経由で先のタグに対して送られる．

このシミュレーションは 2 つの段階を経て，徐々に厳密なシミュレーションへと移行を行った．最初の段階では，主にアクティブタグ間の通信を QOMET によってエミュレートする部分を中心に検証を行った．この段階では，アクティブタグの詳細な動作は必要ではなかったため,「一定の周期毎にビーコンを発する」というアクティブタグの動作を単純化した機能のみを実装した dummytag Space と，ア

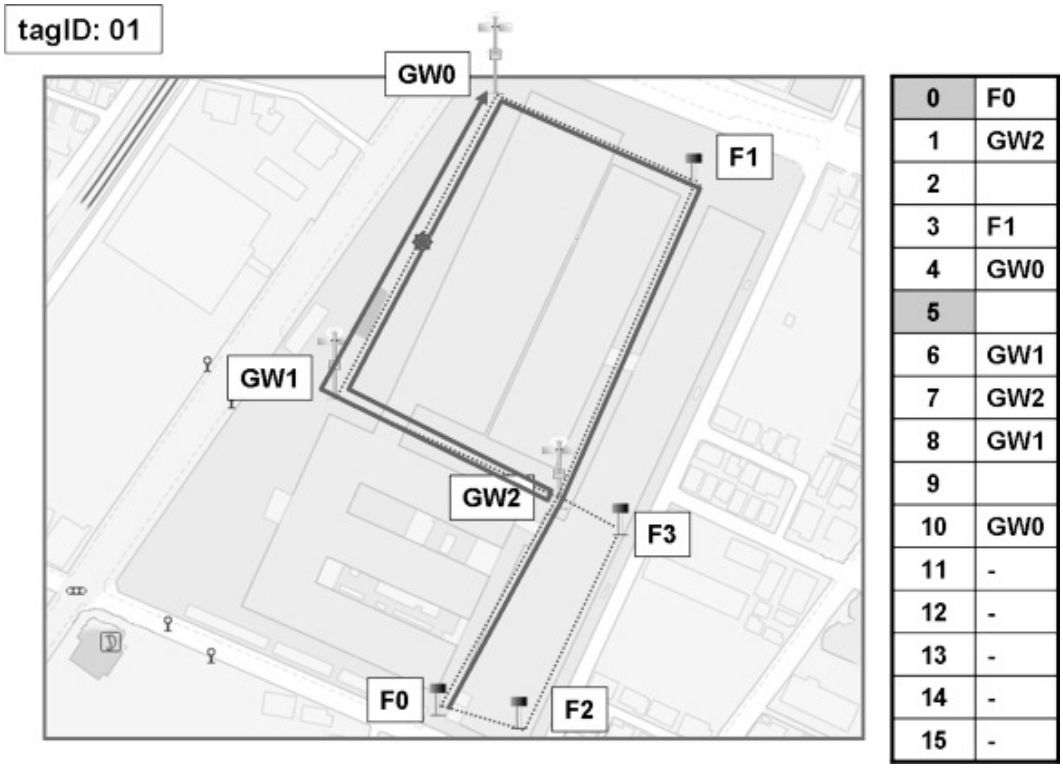

図 5.6: 実証実験の指示書

クティブタグの通信をエミュレートする Chanel Space を各ノード上で実行した．Chanel Space が通信に用いるパラメータは QOMET を用いて各タグの位置をもとに算出を行ったものを利用した．実験は 16 台の移動タグ，4 台の固定タグ，3 台のゲートウェイタグを利用して行った．このシミュレーションによって，通信部分の確認を行った後，次の段階では実際のアクティブタグで利用されているファームウェアのバイナリコードをプロセッサエミュレータ上で実行するシミュレーションを行った．

今回シミュレーションの対象としたアクティブタグのプロトコルでは，送信されたパケットのビット同期を利用してタグのクロックを補正するため，Space 間の通信のレイテンシがシミュレーションに大きな影響を与える．そこでシミュレーションでは実時間の 1/10 倍の時間で進む仮想時間を利用することによってレイテンシの影響を抑えた．この場合であっても，イベントドリブンシミュレータで行われるシミュレーションとは異なり，シミュレーションの所要時間が実行以前に把握できることはこの方式の利点の一つである．このシミュレーションの結果の

表 5.3: アクティブタグを利用した歩行者追跡システムのシミュレーションで用いられた Space 群

Node Number	Space Name	Destination Space
1	Mobile Tag 0	Chanel Space 0
	Chanel Space 0	Mobile Tag 1，Mobile Tag 2，…，Gateway Tag 2
2	Mobile Tag 1	Chanel Space 1
	Chanel Space 1	Mobile Tag 0，Mobile Tag 2，…，Gateway Tag 2
・ ・ ・		
23	Gateway Tag 2	Chanel Space 22
	Chanel Space 0	Mobile Tag 0，Mobile Tag 1，…，Gateway Tag 1

一部を図 5.7 に示す．この図ではシミュレーションの結果を実証実験と同様に位置推定エンジンで処理した結果を点線で，ゲートウェイタグに対してアップロードされたデータに含まれている情報がやりとりされた点の位置と，それに対応する時間を X 印で表している．図の下部の点線が時間軸を表しており，点線上の X 印はゲートウェイタグに対してアップロードされたデータに含まれている点がやりとりされた時点の時間を示している．この図ではゲートウェイタグに対してアップロードされたデータに含まれている全ての点ではなく，図に表示された範囲に関する情報のみを表示した．図 5.6 と比較すると位置推定後の軌跡に関しては正しい結果が得られているといえる．しかし，実証実験時に歩行者の移動経路を測定することができておらず，歩行者が指示書通りに歩行を行ったことを仮定してシミュレーションを行ったため，この結果のみでは今回のシミュレーションが信頼に足るものであると結論付けることはできない．そこで，我々はより単純な移動経路で実機を用いた追実験を行った．

5.3.4 追実験とシミュレーションの比較

シミュレーションが実機と同様に動作していることを確認するために単純な移動経路での実機を用いた実験とそのシミュレーションを行った．この実験で用いたトポロジーは図 5.8 のように 2 人の歩行者が点 A もしくは点 B から点 C を経由し点 D に至るものである．点 A，点 B，点 C には固定タグが，点 D にはゲートウェイタグが設置されている．この移動経路において，5.3.1 で触れたタグのパラメー

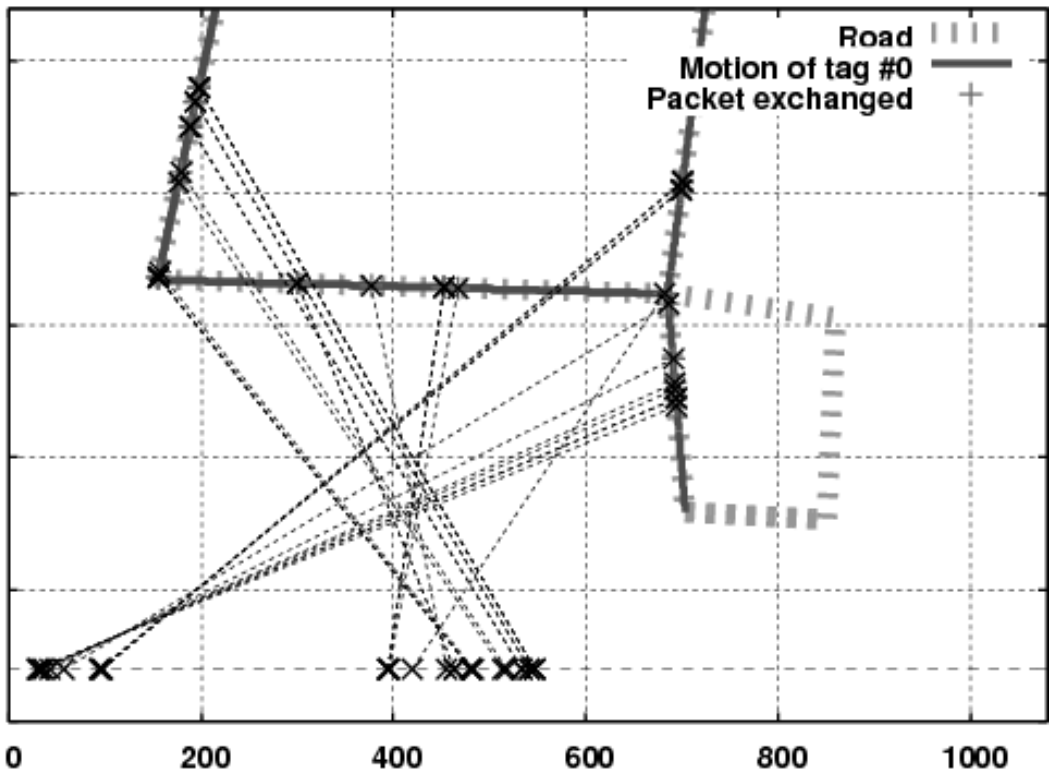

図 5.7: 再現シミュレーションの結果

タである仮想時間マスクを 0x0f，0x07，0x03，0x01 と変化させながら実験を行った．また，同様の条件でシミュレーションを行い，これらの結果を位置推定エンジンで処理した結果の比較を行った．

仮想時間マスクを 0x0f とした場合の実験とシミュレーションの比較を図 5.9 と図 5.10 に示す．

この図で分かるように，位置推定エンジンで処理を行った軌跡は完全に一致しているが，個々の推定点は一致していない．これはモバイルタグの動作は最初に固定タグかゲートウェイタグからの受信した時点でアクティブとなり，動作を始めるが，このタイミングが一定ではないためで，これはシステム自体が持つ不確定性に起因する現象である．つまり，実機を用いて実験であってもシミュレーションであってもこのタイミングは一定ではなく，何回か試行することによりたまたま一致する場合もあり得る．

次に，仮想時間マスクを 0x03 とした場合の実験とシミュレーションの比較を

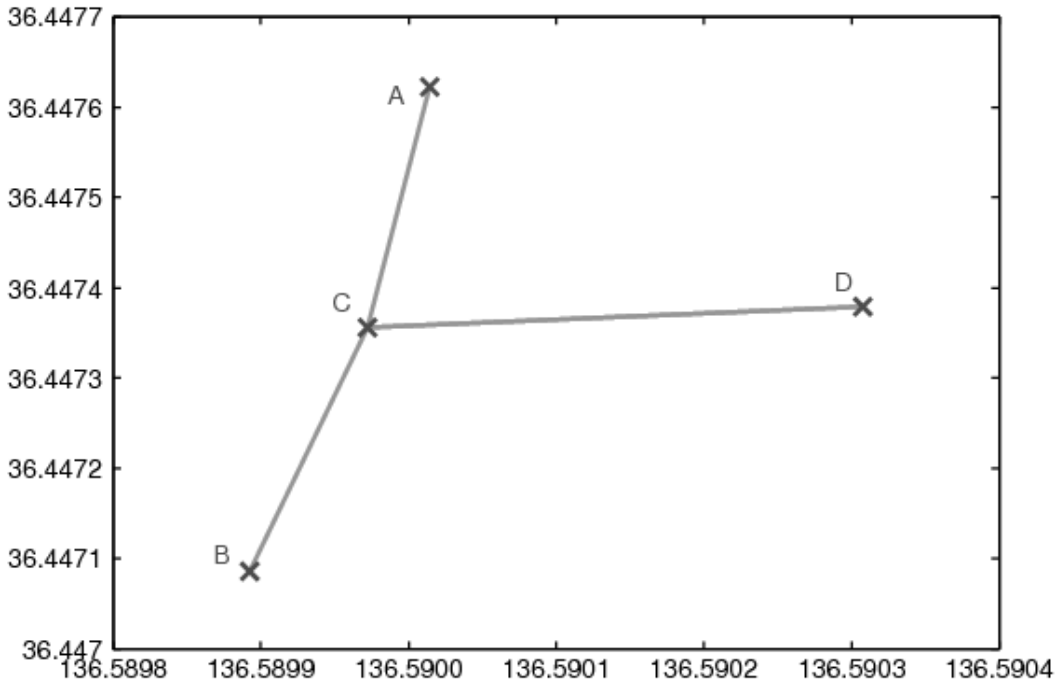

図 5.8: 追実験のトポロジー

図 5.11 と図 5.12 に示す．図から分かるように，仮想時間マスクが 0x0f の時と同様に，位置推定エンジンで処理を行った軌跡は完全に一致しているが，個々の推定点は一致していない．また，点 A や点 B から点 C に至る経路は推定されていない．これは仮想時間マスクを 0x0f から 0x03 にしたことにより，タグでの情報の蓄積が約 16 秒毎から約 4 秒毎になり，情報の精度は向上したものの，メモリがあふれて古い情報が捨てられてしまう現象が発生していることを表している．この現象は実験とシミュレーションのいずれでも発生している．

このことは，実験でもシミュレーションでも，タグのメモリに蓄積される情報の量に大幅な相違は生じていないこと，通信の可否の頻度も妥当な範囲であることを示している．

5.3.5 より大規模な実験のシミュレーション

実証実験の再現シミュレーションと追実験によってある程度の正確性が確認できたため，大規模シミュレーションプラットフォームの利点を活かし，実機を用

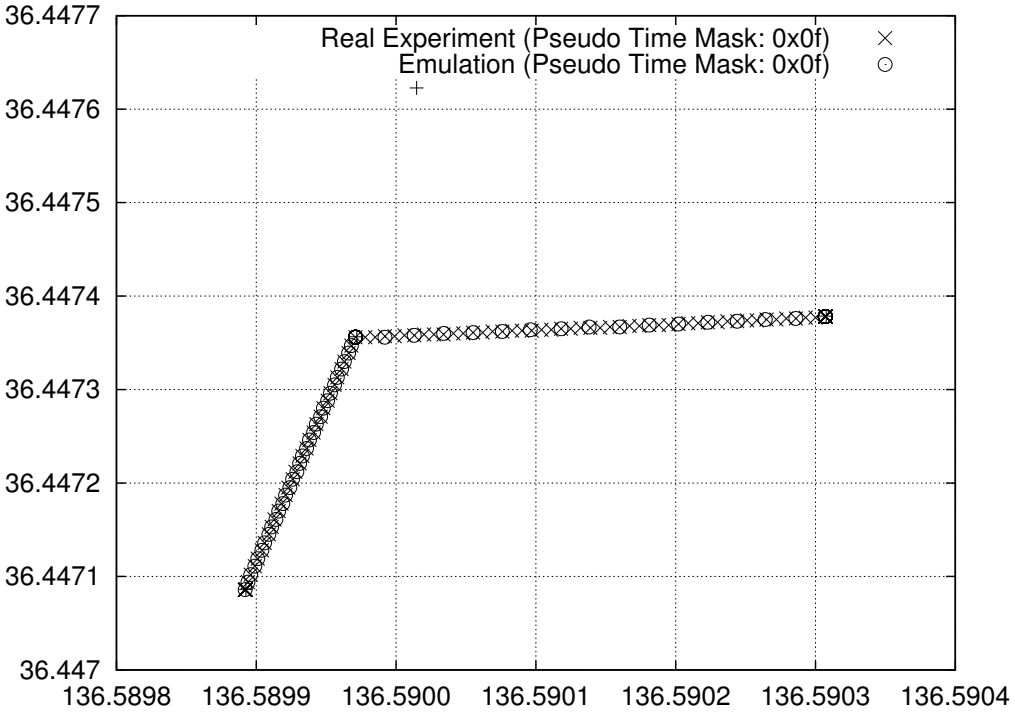

図 5.9: 実験とシミュレーションの比較 (仮想時間マスク:0x0f)

いた実験が現実的ではない規模のシミュレーションを行った．ここでは，ある地域内の 100 人の歩行者を追跡する状況を模して行ったシミュレーションの解説を行う．

このシミュレーションでは，国土地理院が提供する地理情報標準プロファイル（JPGIS; Japan Profile for Geographic Information Standards）のデータからある地域を選択し，その範囲内を移動する歩行者の移動パターンを生成するモーションジェネレータによって生成した行動パターンを利用した．図 5.13 に生成された行動パターンのスナップショットを示す．本シミュレーションでは，災害発生時を想定し，地域内の歩行者が指定避難場所に向かうパターンの生成を行った．このシミュレーションの結果，各歩行者が所持するタグから情報のアップロードが行われた回数を図 5.14 に示す．

図 5.14 から，全てのタグが情報のアップロードに成功している訳ではないことが分かる．さらに詳細な解析の結果，表 5.4 に示すように 45%のタグは全く情報のアップロードを行えていないことが判明した．また，シミュレーションが終了

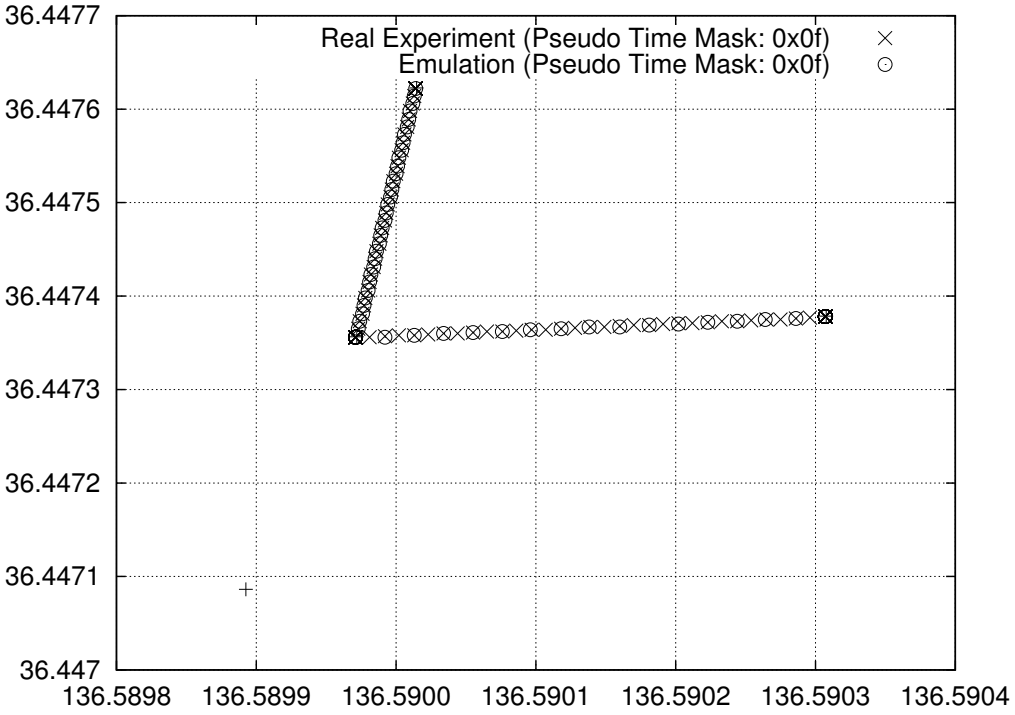

図 5.10: 実験とシミュレーションの比較 (仮想時間マスク:0x0f)

した時点でアップロードが終了していない情報をメモリ上に保有していたタグの数は表 5.5 に示すとおり，実に 91%に達していることが明らかとなった．

Number of tags that uploaded at least one record	Number of tags that uploaded no record
55	45

表 5.4: 情報のアップロードに成功したタグの数

こうした結果は，タグの通信方式に起因している．この点に関しては 5.3.6 で検討を行う．

5.3.6 シミュレーションによって発見することができた潜在的問題点

ここまでで述べたシミュレーションを行う過程でいくつかの実機のファームウェアが持つ問題を発見し，実機の開発にフィードバックすることができた．以下で

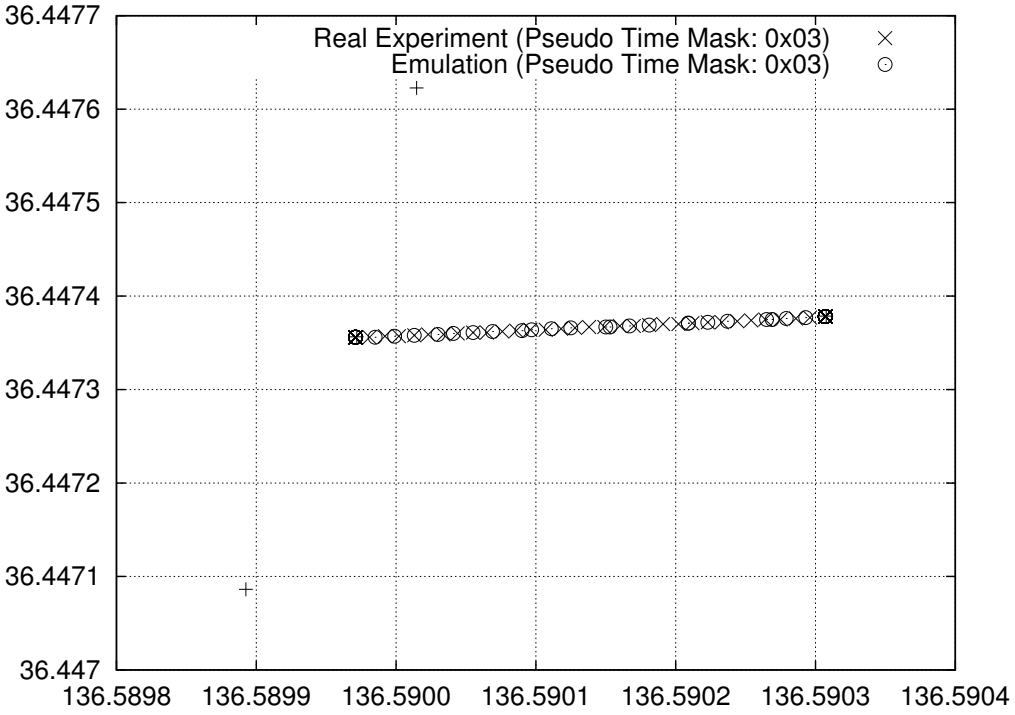

図 5.11: 実験とシミュレーションの比較 (仮想時間マスク:0x03)

Number of tags have information left in memory	Number of tags have no information left in memory
91	9

表 5.5: シミュレーション終了時にメモリ上に情報が残っていたタグの数

はシミュレーションによって判明した問題点について説明する．

時刻同期アルゴリズム

この問題は図 5.15 に示したタグの送信タイミング可視化ツールを用いて発見された．図の横軸は時間を表し，縦軸は上から順に歩行者が着用するモバイルタグ（P0 – P15），固定タグ（F0 – F3），ゲートウェイタグ（GW0 – GW2）を表している．このツールでは，タグが 2 秒間隔で繰り返すアクティブ期間とスリープ期間のうち，アクティブ期間である 11 スロットを図示している．従って，縦長の長方形が 11 個連なった横長の長方形が 2 秒間隔で図示されている．タグが送信を行っ

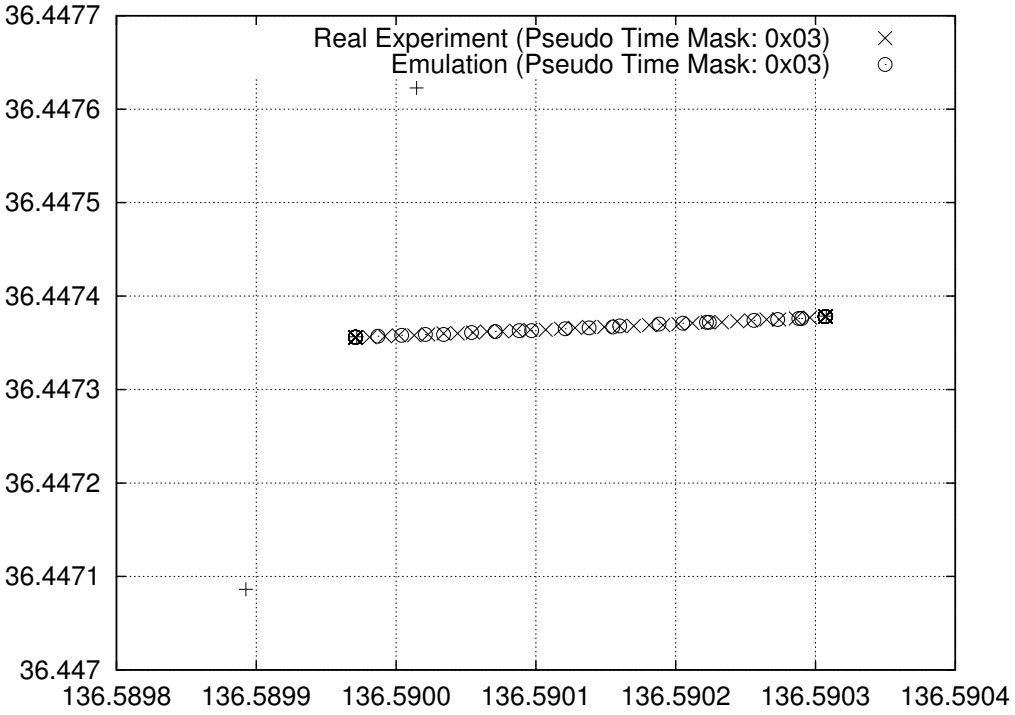

図 5.12: 実験とシミュレーションの比較 (仮想時間マスク:0x0f)

たスロットは濃い色の長方形となっている．図 5.15 では，P1，P4，P5，P7，P8 の各モバイルタグは一定時間他のタグとのパケットの送受信がなかったため非活性状態となっており，このシミュレーションで用いたファームウェアはアクティブ期間で受信された最も早いパケットに含まれる情報をもとにそのパケットの送信元に対して時刻同期を行う．アクティブ期間のうち，スロット 1 とスロット 11 はゲートウェイタグからの送信用に予約されており，ゲートウェイタグ以外のモバイルタグや固定タグはスロット 2 からスロット 9 の間からアクティブ期間毎に乱数で抽出したスロットで送信を行う．そのため，ゲートウェイタグの通信範囲内に位置するタグは常にゲートウェイタグに対して時刻を同期することになり，問題は起こらない．しかし，ゲートウェイタグの通信範囲外でモバイルタグ同士のみがパケットの交換を行う状態になると時刻同期の基準がアクティブ期間毎に異なる可能性があることになり，最悪の場合には時刻同期を行っているモバイルタグ同士の間で時刻同期を繰り返してしまい，結果的に他のゲートウェイタグとの通信範囲内にあるモバイルタグと大幅にずれた時刻を持ってしまう現象が発生し得る．

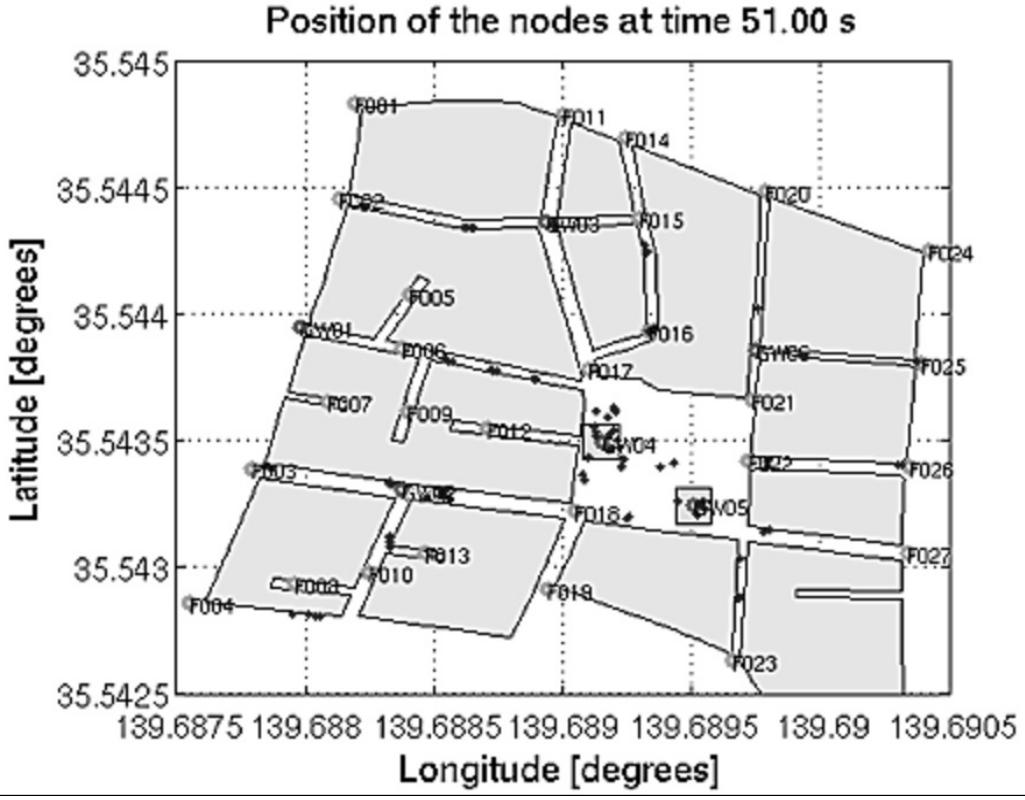

図 5.13: モーションジェネレータによって生成された歩行者の移動

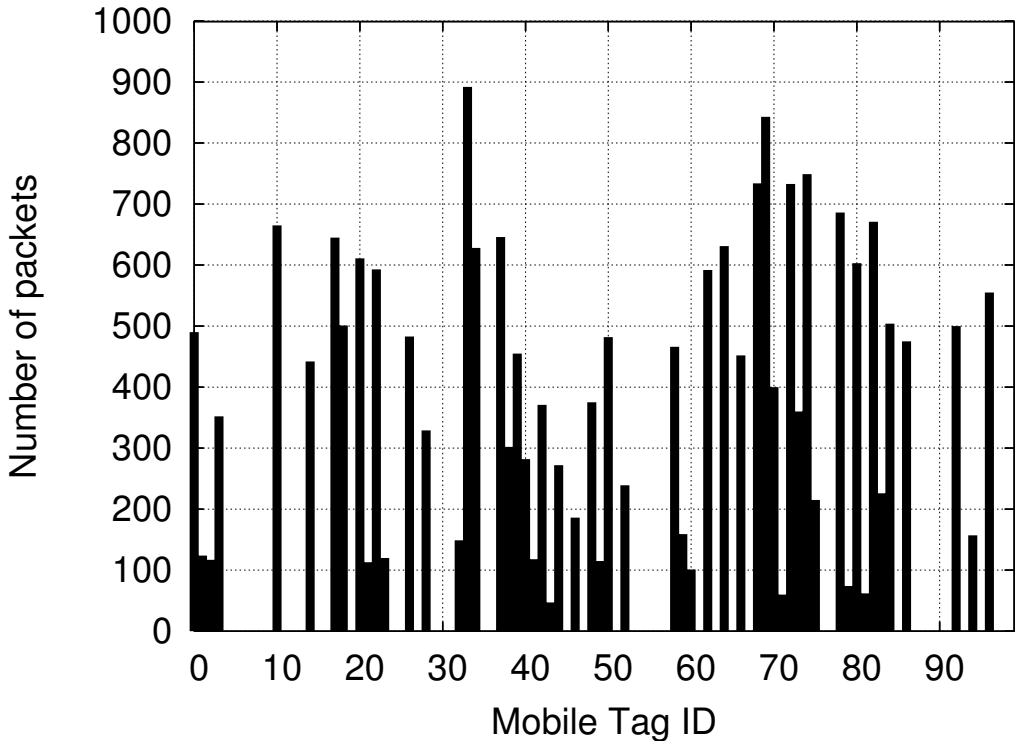

図 5.14: 各モバイルタグによってアップロードされたパケットの数

図 5.15 において，問題のクロックずれは P10 と P11 に発生している．この時間において，P10 と P11 は他のタグから孤立し，P10 と P11 の間でのみ通信が行える状態となっている．この状況で P10 と P11 の間でのみ時刻同期を繰り返す状態となっていたと考えられる．この問題はファームウェアの修正を行い，モバイルタグ同士での時刻同期を行わないようにすることで改善することができた．この変更は実機のファームウェアへのフィードバックが行われた．

乱数の品質

初期のファームウェアでは乱数の品質が十分ではなく，タグが密集した状態で複数のタグが同じスロットを撰択してしまう可能性があった．この現象が発生すると，タグが他のタグに対して自らが情報を蓄積していることを通知できる可能性が低くなってしまう可能性がある．メモリが潤沢である場合にはこの問題はそれほど問題にならないが，今回のタグのように，蓄積できる情報の量が限られている場合には深刻な問題となりうる．シミュレーションの結果からこの問題が発生していることが判明した後，プロセッサエミュレータで乱数の発生部分をバイパスし，プロセッサエミュレータが生成した乱数を利用することで改善がみられたため，後に実機のファームウェアにも対策が行われた．この現象も図 5.15 で確認することができる．例えば 122.5 秒前後のアクティブ期間で P13 から F3 までの 7 つのタグが全て第 4 スロットを選択してしまっていることが確認できるが，これは乱数の品質が十分ではないことによってによって生じる不具合である．

アクティブ期間における衝突回避アルゴリズム

今回のシミュレーションで利用したアクティブタグのファームウェアでは，アクティブ期間中の 1 スロットを乱数で選択し，そのスロットでビーコンパケットの送信を行う．このビーコンパケットの送信には 9 スロットが利用可能となっているが，5.3.5 で述べたように，規模が大きい実験ではタグからの情報のアップロードが行えない現象が発生することが分かった．そこで，タグがスロットの選択に用いる乱数が理想的なホワイトノイズに従うものとした場合のアクティブ期間のスロット数と，一つのアクティブ期間内に全くパケットの衝突が起こらない確率

との関係を求めた．この関係は，N_s をアクティブ期間を構成するスロット数，N_t をカバレッジ内のタグ数とした時，全てのタグが別々のスロットを選択する確率

$$FSR = \frac{(N_s - 1)!}{(N_s - N_t)!} \cdot \frac{1}{N_s^{n_t - 1}} \tag{5.1}$$

に等しい．この関係を図 5.16 に示す．我々が行った 100 人の歩行者を対象とするシミュレーションでは，約 45%のタグが情報のアップロードに成功している．これは，アクティブ期間を構成するスロット数が 9 であることを考慮すると，ゲートウェイタグの周辺にはシミュレーション期間を通してゲートウェイ自身を含め，平均 4 台のタグが存在していたことに相当する．

このことから，アクティブ期間を構成するスロット数を 33 にした場合には 80%程度のタグが情報のアップロードを行うことが可能となると予測される．但し，これはアクティブ期間を構成するスロット数が増えた場合であってもアクティブ期間の時間は延びない，つまりスロット数の増加に伴ってビットレートを向上させることが可能であるという仮定の下での予測となる．

5.3.7 シミュレーションによって計測が可能となった項目

タグをエミュレートすることで，実機を用いた実験では観察できないいくつかの事象が観察可能となった．以下ではこれらのうち，代表的なものを挙げる．

プロファイリング

センサネットワークのノードのように汎用の入出力のための I/O を持たない機器の場合には内部で発生している事象を観察することは難しいが，RUNE の上でシミュレーションとして実行する場合にはプロファイリング等の解析を行うことも可能である．実際にはプロファイリングを行うことで僅かながら実行時のオーバーヘッドが増すため，実行時間が正確かどうかは慎重な検討が必要となる．しかし，時間の情報が直接利用できない場合であってもプロファイリングによって関数の呼び出し頻度という情報が得られることは非常に有用である．実際にプロファイリングを行った例を表 5.6 に示す．ここで，pic16f648Op から始まる関数がプロセッサエミュレータでプロセッサの一つ一つの命令をエミュレートする部分

表 5.6: プロファイリングの結果

local		total				
sec.	%	sec.	%	calls	sec./call	name
26.261842	25.2	87.977816	84.5	1	87.977816	<PLTspaces/mtag0.so:tag_step>
13.205866	12.7	4.851277	4.7	1652395	0.000003	<PLTspaces/mtag0.so:pic16f648Wait>
12.105390	11.6	37.295617	35.8	837408	0.000045	<PLTspaces/mtag0.so:pic16f648ExecOp>
11.229424	10.8	0.459433	0.4	1652394	0.000000	<PLTspaces/mtag0.so:pic16f648ProcInt>
6.432733	6.2	10.708582	10.3	272126	0.000039	<PLTspaces/mtag0.so:pic16f648OpBtfsc>
5.215759	5.0	8.490204	8.2	269309	0.000032	<PLTspaces/mtag0.so:pic16f648OpBtfss>
4.617665	4.4	5.397146	5.2	273670	0.000020	<PLTspaces/mtag0.so:pic16f648OpGoto>
4.533895	4.4	4.533895	4.4	557980	0.000008	<PLTspaces/mtag0.so:pic16f684ReadReg>
4.116654	4.0	4.115922	4.0	814989	0.000005	<PLTspaces/mtag0.so:addWait>
0.333824	0.3	0.386500	0.4	7746	0.000050	<PLTspaces/mtag0.so:pic16f648OpBcf>
0.056427	0.1	0.056427	0.1	16562	0.000003	<PLTspaces/mtag0.so:pic16f648SetReg>
0.027595	0.0	0.047778	0.0	2709	0.000018	<PLTspaces/mtag0.so:pic16f648OpBsf>

である．pic16f648ExecOp() は命令デコードを行う関数，pic16f648ProcInt() は割り込みの処理を行う関数である．この表から，今回利用したタグのファームウェアでは BTFSC，BTFSS，GOTO など，ループを構成するインストラクションの実行頻度が高いことがわかる．

送受信や割り込みのタイミング

図 5.15 で示したような送受信のタイミングも実機の動作から得ることは難しい．また，同様にどの程度の頻度で割り込みが発生するかを確認することも難しい．こうした情報もシミュレーションとして実行している場合には比較的容易に取得可能である．図 5.17 にあるモバイルタグの割り込み周期を示す．通常，今回シミュレーションの対象としたシステムのタグは 53ms 毎にタイマ割り込みが発生し，これによりスロットの管理を行っている．ところが，他のタグに対して時刻同期を行う際にはタイマ割り込みのカウンタを操作することで同期しようとするため，割り込みの周期に不規則性が生じる．図 5.17 では，動作開始から 4 秒過ぎの部分でこうした同期が生じていることが分かる．

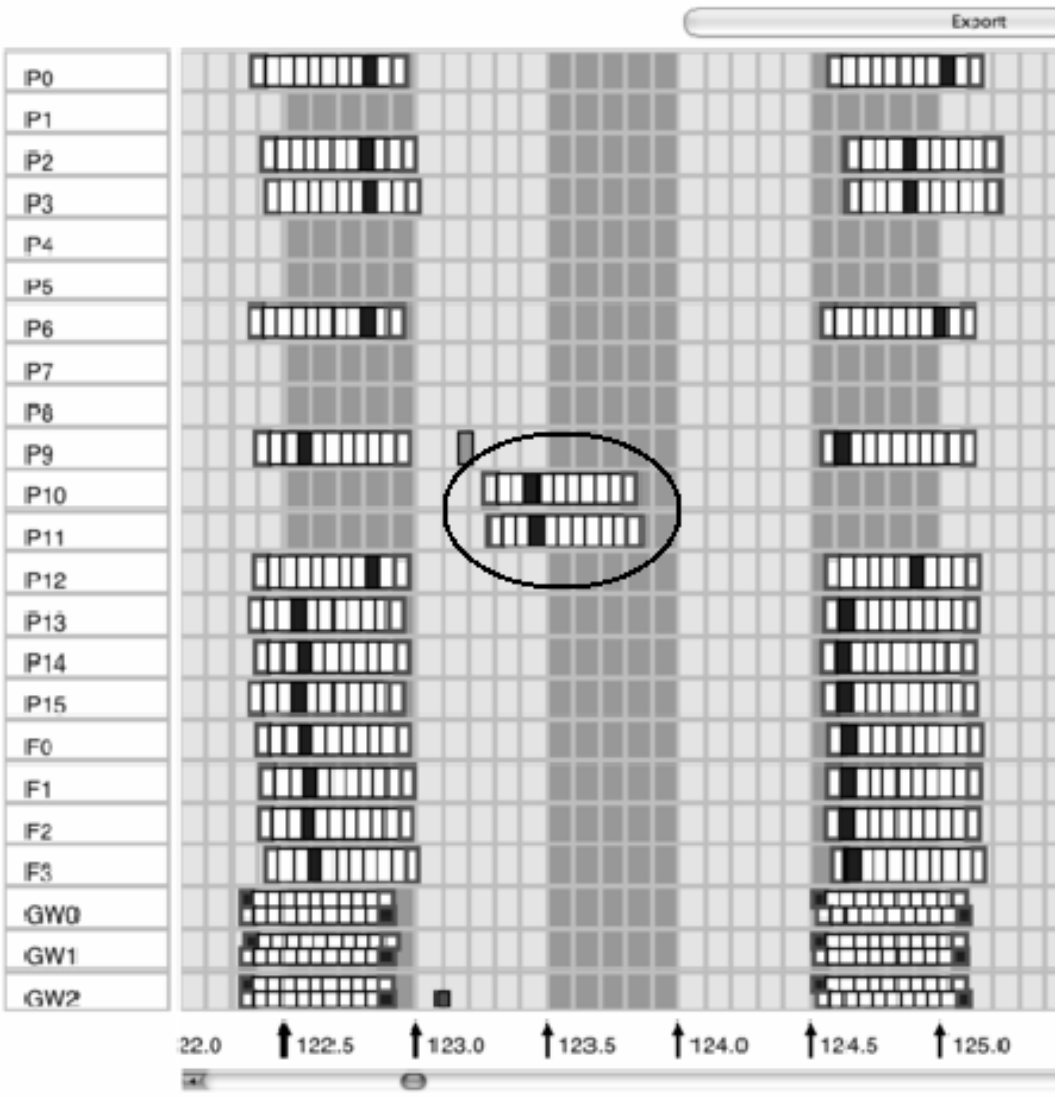

図 5.15: タイムスロットの可視化ツール

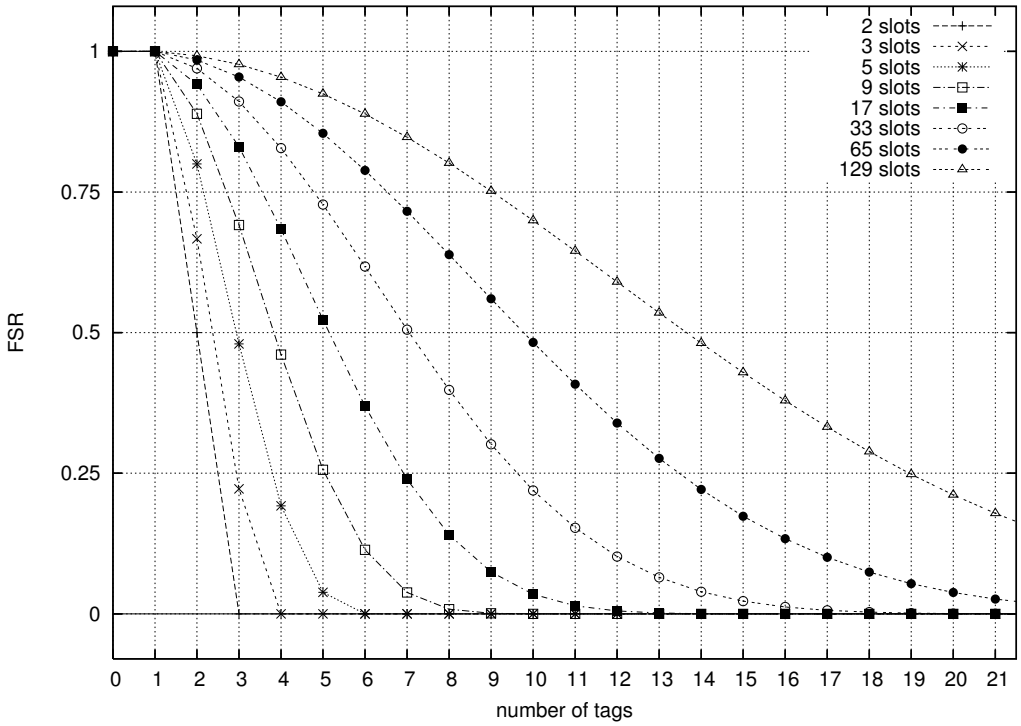

図 5.16: アクティブ期間の長さとタグ数の関係

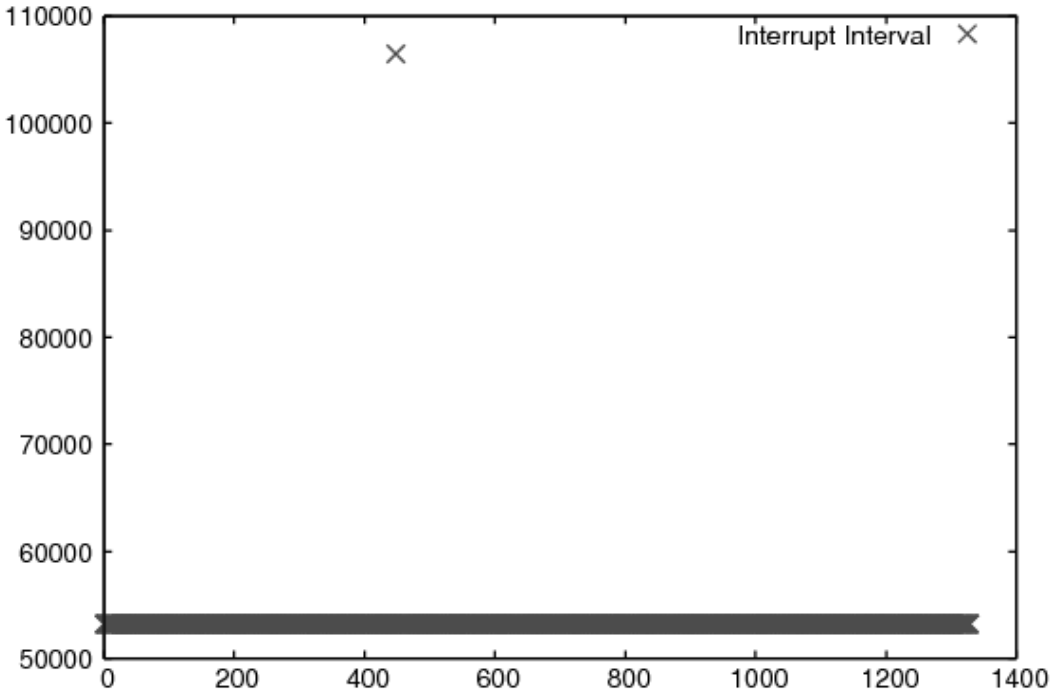

図 5.17: 割り込み周期

第 6章

結果及び評価

本論文では，ユビキタスネットワークシミュレーション環境の構築に関して述べ，ユビキタスネットワークシミュレーションプラットフォームRUNEの提案を行った．

RUNE coreでは，動作基盤としてStarBEDを利用することによって，多数のノードから構成されるシミュレーションに対応可能となっている．また，シミュレーション定義ファイルによる構成の定義とRUNE masterとRUNE managerによる自動実行によってクラスタ環境を意識しないシミュレーションの実行とシミュレーションの構成変更を容易にしている．さらに，SpaceとConduitを用いたシミュレーション対象の実装，およびマルチレベルエミュレーションレイヤの提供により，様々なネットワーク，ハードウェア，ソフトウェアアーキテクチャのエミュレーション，さらには周囲の環境のシミュレーションも可能としている．これらに加え，マルチレベルエミュレーションレイヤの提供は開発の段階に応じて複数の抽象度でのシミュレーションを可能としている．

以下では，RUNEで行った実験に関して行った計測の結果を述べる．

6.1 PICエミュレータの評価

実装を行ったPIC 16F648エミュレータを利用してパフォーマンスの評価を行った．評価は以下に示した環境で行い，評価用のバイナリコードとしては，後述するアクティブタグシステムのシミュレーションで用いられた実機のファームウェア

を利用した．

A.3 節のサンプルプログラムを用い，1MHz，2MHz，4MHz，8MHz，16MHz の各目標動作周波数で 1 個〜100 個までのプロセッサインスタンスを実行した際の実稼働周波数の測定を行った．

各目標周波数における結果を図 6.1 から図 6.5 に示す．また，全ての結果を重ねてプロットしたものを図 6.6 に示す．

図から明らかな通り，目標周波数を f[MHz] とすると，性能の落ち込みが全ての周波数においてほぼ 160/f の双曲線上に沿って生じるという結果が得られた．したがって，図 6.1 では測定結果と 160/f の線はプロセッサインスタンスが 100 個までの範囲では交わっていないが，160 個近辺で交差することが予想される．

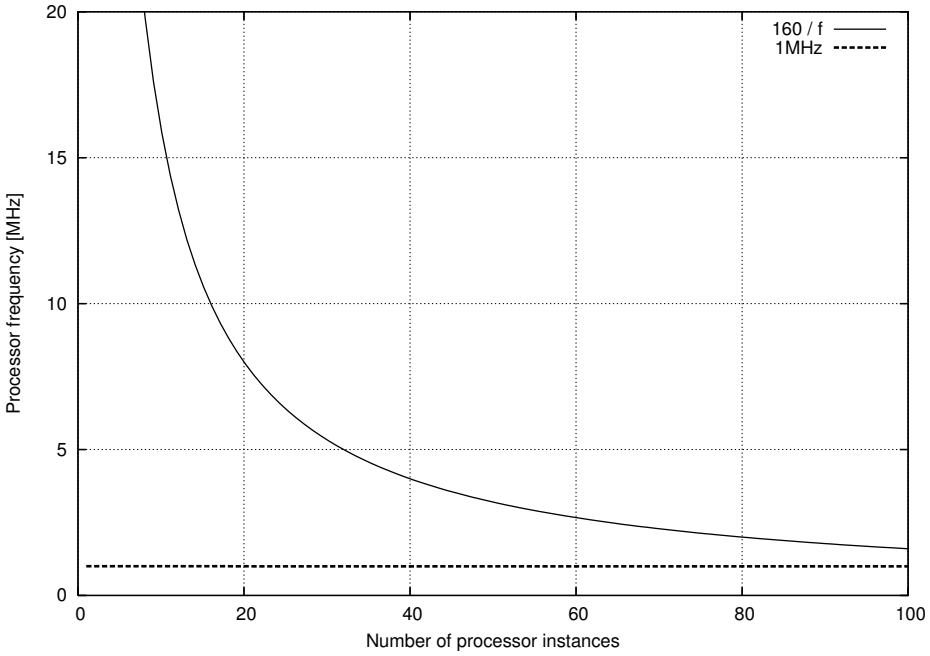

図 6.1: 1MHz 動作時の時刻精度

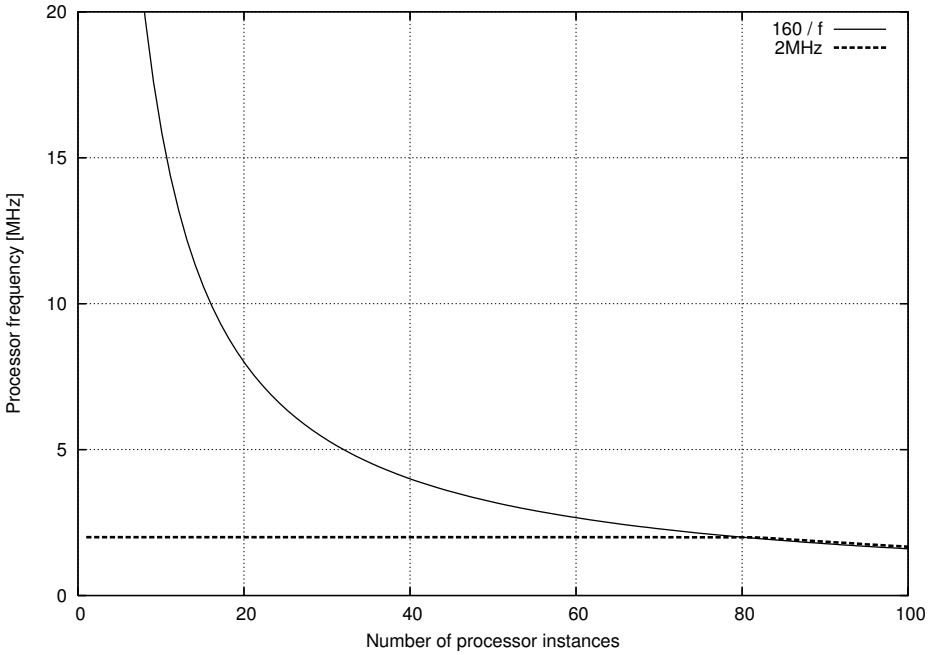

図 6.2: 2MHz 動作時の時刻精度

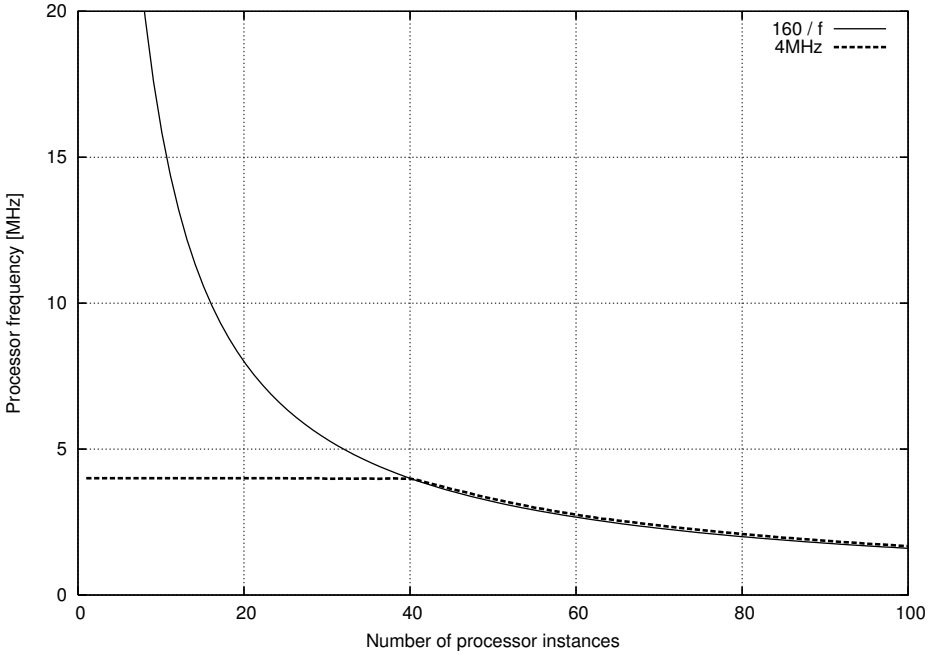

図 6.3: 4MHz 動作時の時刻精度

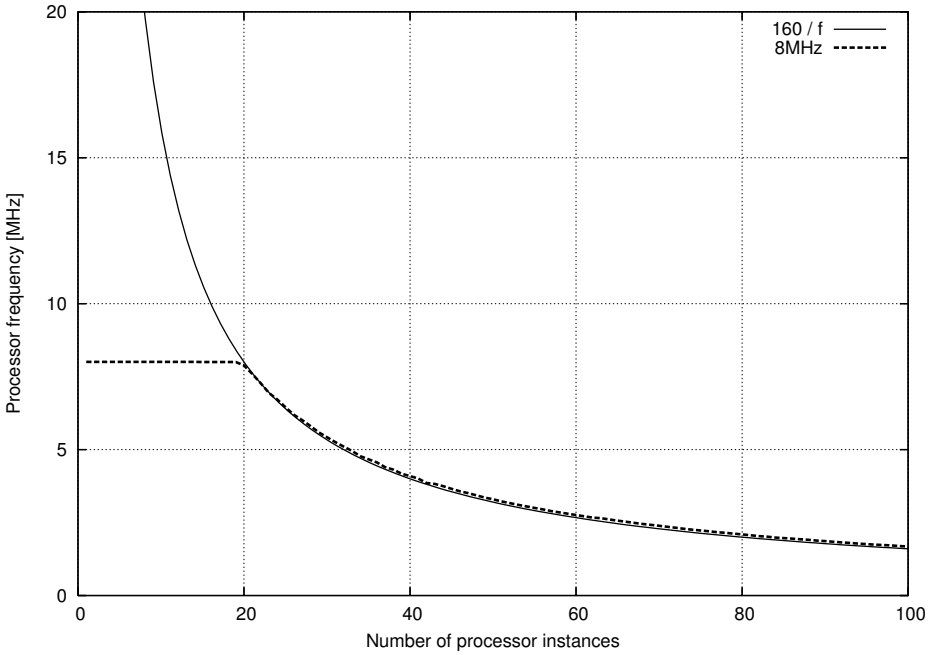

図 6.4: 8MHz 動作時の時刻精度

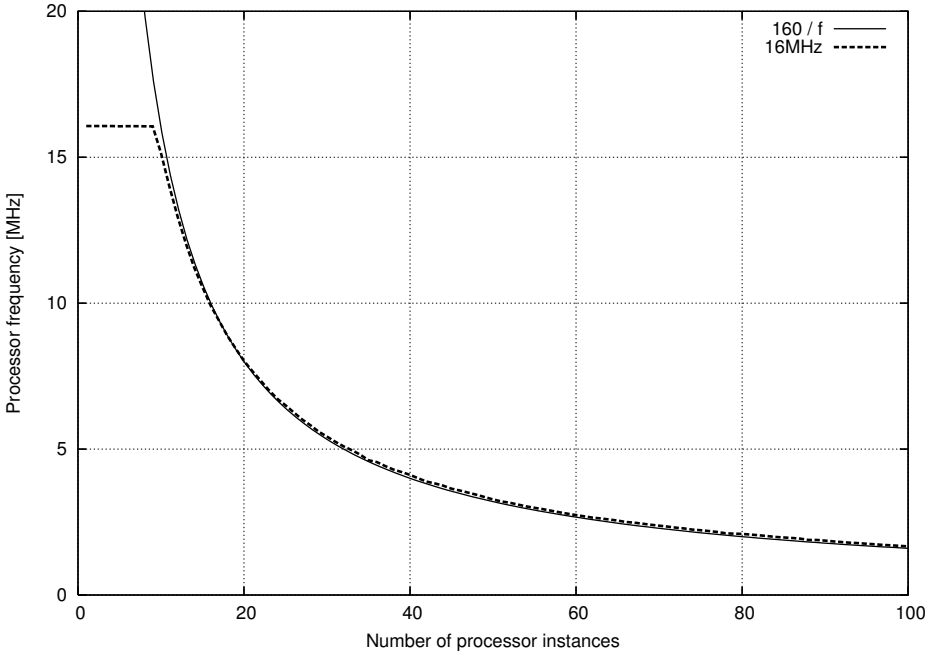

図 6.5: 16MHz 動作時の時刻精度

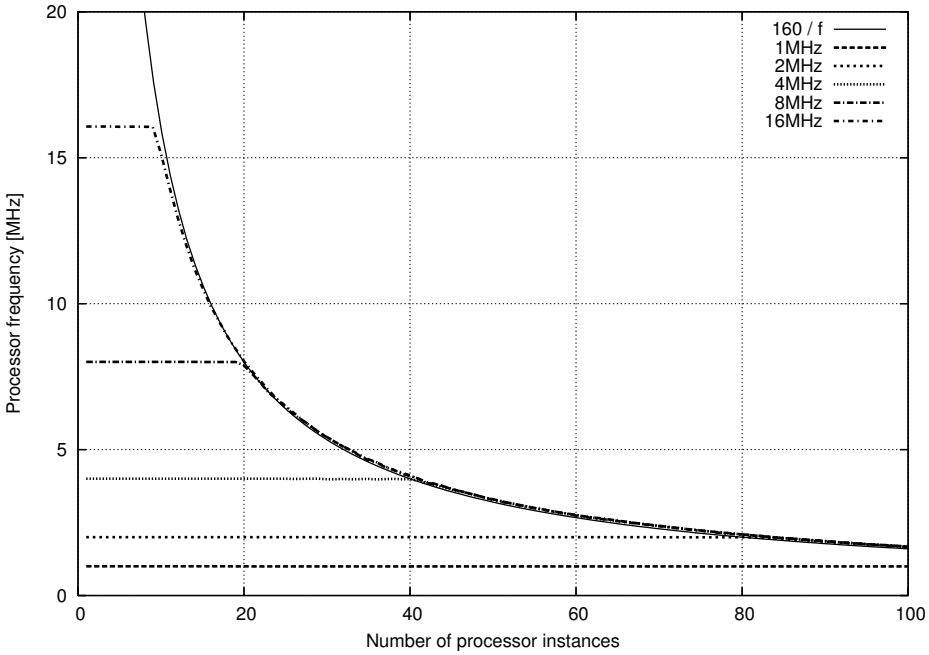

図 6.6: 各周波数動作時の時刻精度

6.2 RUNE実行時のプロファイリング

RUNE manager がシミュレーションを実行する際にどの程度の時間を実行制御に要し，どの程度の時間をシミュレーション対象自体の実行に割り当てているかを確かめるためにプロファイリングを行った．

ここでは，ネットワークによる伝送遅延の影響を抑えるため，単一ノード上で 4 つの Space を実行し，RUNE manager と 4 つの Space それぞれのプロファイルを計測した．

表 6.1 から，RUNE manager では，88%近くを spaceExecStub() で，その他の大半の時間を main() で消費していることが分かる．RUNE manager では，main() 関数で初期化と接続待ちを行い，RUNE master からの接続を受け付けると Sapce のアタッチを行う．Space の実行が開始された以降は main() 関数では特に処理を行っていない．spaceExecStub() は各 Space オブジェクトの Step() 関数を呼び出す処理を行っている．従って，RUNE manager で spaceExecStub() の実行に要して

いる時間が他の Space の処理に利用されている時間に相当する．このことは，他の Space の全関数の実行時間の合計が RUNE manager の spaceExecStub() の実行時間と同じ 88 秒弱 (84.6%) となることで確認できる．

表 6.1: RUNE core のプロファイル

local		total		calls	time	function
time[s]	%	time[s]	%		[s/call]	
87.996953	84.6	87.998882	84.6	22	3.999949	spaceExecStub
16.051536	15.4	104.061494	100.0	1	104.061494	main
0.003724	0.0	0.006349	0.0	1	0.006349	procConnection
0.003552	0.0	88.002498	84.6	1	88.002498	execSpace
0.002273	0.0	0.002273	0.0	880	0.000003	__bswap32
0.001113	0.0	0.002102	0.0	27	0.000078	protocolHandler
0.000822	0.0	0.000822	0.0	1	0.000822	attachSpaces
0.000599	0.0	87.998946	84.6	22	3.999952	spaceCommHandler
0.000523	0.0	0.000523	0.0	1	0.000523	showLocalSpaceInfo
0.000213	0.0	0.000225	0.0	1	0.000225	setupConduit
0.000064	0.0	0.000064	0.0	1	0.000064	dumpDlAddr
0.000048	0.0	0.000048	0.0	14	0.000003	timevalsub
0.000012	0.0	0.000012	0.0	6	0.000002	__bswap16

表 6.2: dist0 のプロファイル

local		total		calls	time	function
time[s]	%	time[s]	%		[s/call]	
87.993667	84.6	88.017498	84.6	1	88.017498	dist.so:dist_step
0.023582	0.0	0.023831	0.0	14	0.001702	dist.so:dist_write
0.000249	0.0	0.000249	0.0	70	0.000004	dist.so:__bswap32
0.000011	0.0	0.000011	0.0	1	0.000011	dist.so:dist_init

表 6.3: mtag0 のプロファイル (抜粋)

local		total		calls	time	function
time[s]	%	time[s]	%		[s/call]	
26.261842	25.2	87.977816	84.5	1	87.977816	mtag0.so:tag_step
13.205866	12.7	4.851277	4.7	1652395	0.000003	mtag0.so:pic16f648Wait
12.105390	11.6	37.295617	35.8	837408	0.000045	mtag0.so:pic16f648ExecOp
11.229424	10.8	0.459433	0.4	1652394	0.000000	mtag0.so:pic16f648ProcInt
6.432733	6.2	10.708582	10.3	272126	0.000039	mtag0.so:pic16f648OpBtfsc
5.215759	5.0	8.490204	8.2	269309	0.000032	mtag0.so:pic16f648OpBtfss
4.617665	4.4	5.397146	5.2	273670	0.000020	mtag0.so:pic16f648OpGoto
4.533895	4.4	4.533895	4.4	557980	0.000008	mtag0.so:pic16f684ReadReg
4.116654	4.0	4.115922	4.0	814989	0.000005	mtag0.so:addWait
0.333824	0.3	0.386500	0.4	7746	0.000050	mtag0.so:pic16f648OpBcf
0.056427	0.1	0.056427	0.1	16562	0.000003	mtag0.so:pic16f648SetReg
0.027595	0.0	0.047778	0.0	2709	0.000018	mtag0.so:pic16f648OpBsf

表 6.4: mtag1 のプロファイル (抜粋)

local		total		calls	time	function
time[s]	%	time[s]	%		[s/call]	
28.766726	27.6	87.960615	84.5	1	87.960615	mtag1.so:tag_step
11.639790	11.2	37.007512	35.6	834577	0.000044	mtag1.so:pic16f648ExecOp
11.383622	10.9	11.382184	10.9	1647814	0.000007	mtag1.so:pic16f648ProcInt
10.805442	10.4	3.243604	3.1	1647813	0.000002	mtag1.so:pic16f648Wait
6.433751	6.2	1.338605	1.3	813236	0.000002	mtag1.so:addWait
5.648357	5.4	9.747341	9.4	271491	0.000036	mtag1.so:pic16f648OpBtfsc
5.456826	5.2	10.390188	10.0	268842	0.000039	mtag1.so:pic16f648OpBtfss
4.159767	4.0	2.666711	2.6	556096	0.000005	mtag1.so:pic16f684ReadReg
3.399214	3.3	4.891159	4.7	273039	0.000018	mtag1.so:pic16f648OpGoto
0.074567	0.1	0.126449	0.1	7262	0.000017	mtag1.so:pic16f648OpBcf
0.067612	0.1	0.024652	0.0	15774	0.000002	mtag1.so:pic16f648SetReg
0.029421	0.0	0.056263	0.1	2550	0.000022	mtag1.so:pic16f648OpBsf

表 6.5: gtag0 のプロファイル (抜粋)

local		total		calls	time	function
time[s]	%	time[s]	%		[s/call]	
28.690911	27.6	87.967716	84.5	1	87.967716	gtag.so:tag_step
12.439581	12.0	0.590366	0.6	1690081	0.000000	gtag.so:pic16f648ProcInt
11.441663	11.0	8.416385	8.1	1690083	0.000005	gtag.so:pic16f648Wait
11.127512	10.7	13.298082	12.8	853142	0.000016	gtag.so:pic16f648ExecOp
8.182470	7.9	14.643640	14.1	412601	0.000035	gtag.so:pic16f648OpBtfsc
6.608960	6.4	9.329999	9.0	423438	0.000022	gtag.so:pic16f648OpGoto
6.056274	5.8	6.056274	5.8	836942	0.000007	gtag.so:addWait
3.183431	3.1	3.182709	3.1	427532	0.000007	gtag.so:pic16f684ReadReg
0.128513	0.1	0.211524	0.2	11512	0.000018	gtag.so:pic16f648OpDecfsz
0.052611	0.1	0.052611	0.1	15015	0.000004	gtag.so:pic16f648SetReg
0.018511	0.0	0.036937	0.0	1	0.036937	gtag.so:pic16f648LoadHex
0.017506	0.0	0.017506	0.0	5103	0.000003	gtag.so:hexStrToByte

6.3 RUNEを用いたシミュレーション実行開始処理に要する時間

クラスタ環境でシミュレーションを行う際には実行開始の同期という問題が生じる可能性がある．そこで，23 ノードで実行したシミュレーションの実行開始時刻のばらつきの測定を行った．以下に示すように，最初のノードの実行開始から最後のノードの実行開始までの時間差は約 760[μs] となった．これは実用上十分に小さい時間と考えられる．

```
librune: Execution spaces on remote host took 0.142781 seconds in total.
librune:  launching spaces on 172.16.3.42 took 0.142022
librune:  launching spaces on 172.16.3.43 took 0.142070
librune:  launching spaces on 172.16.3.44 took 0.142076
librune:  launching spaces on 172.16.3.45 took 0.142082
librune:  launching spaces on 172.16.3.46 took 0.142089
librune:  launching spaces on 172.16.3.47 took 0.142096
librune:  launching spaces on 172.16.3.48 took 0.142102
librune:  launching spaces on 172.16.3.49 took 0.142108
librune:  launching spaces on 172.16.3.50 took 0.142114
librune:  launching spaces on 172.16.3.51 took 0.142273
librune:  launching spaces on 172.16.3.52 took 0.142281
librune:  launching spaces on 172.16.3.53 took 0.142288
librune:  launching spaces on 172.16.3.54 took 0.142294
```

```
librune:  launching spaces on 172.16.3.55 took 0.142301
librune:  launching spaces on 172.16.3.56 took 0.142307
librune:  launching spaces on 172.16.3.57 took 0.142312
librune:  launching spaces on 172.16.3.58 took 0.142319
librune:  launching spaces on 172.16.3.59 took 0.142325
librune:  launching spaces on 172.16.3.60 took 0.142377
librune:  launching spaces on 172.16.3.61 took 0.142444
librune:  launching spaces on 172.16.3.62 took 0.142452
librune:  launching spaces on 172.16.3.63 took 0.142483
librune:  launching spaces on 172.16.3.64 took 0.142516
librune:  starting time difference between 172.16.3.42 and 172.16.3.43 is 0.000062
librune:  starting time difference between 172.16.3.43 and 172.16.3.44 is 0.000019
librune:  starting time difference between 172.16.3.44 and 172.16.3.45 is 0.000019
librune:  starting time difference between 172.16.3.45 and 172.16.3.46 is 0.000019
librune:  starting time difference between 172.16.3.46 and 172.16.3.47 is 0.000019
librune:  starting time difference between 172.16.3.47 and 172.16.3.48 is 0.000019
librune:  starting time difference between 172.16.3.48 and 172.16.3.49 is 0.000017
librune:  starting time difference between 172.16.3.49 and 172.16.3.50 is 0.000018
librune:  starting time difference between 172.16.3.50 and 172.16.3.51 is 0.000172
librune:  starting time difference between 172.16.3.51 and 172.16.3.52 is 0.000020
librune:  starting time difference between 172.16.3.52 and 172.16.3.53 is 0.000019
librune:  starting time difference between 172.16.3.53 and 172.16.3.54 is 0.000018
librune:  starting time difference between 172.16.3.54 and 172.16.3.55 is 0.000019
librune:  starting time difference between 172.16.3.55 and 172.16.3.56 is 0.000018
librune:  starting time difference between 172.16.3.56 and 172.16.3.57 is 0.000017
librune:  starting time difference between 172.16.3.57 and 172.16.3.58 is 0.000018
librune:  starting time difference between 172.16.3.58 and 172.16.3.59 is 0.000018
librune:  starting time difference between 172.16.3.59 and 172.16.3.60 is 0.000064
librune:  starting time difference between 172.16.3.60 and 172.16.3.61 is 0.000078
librune:  starting time difference between 172.16.3.61 and 172.16.3.62 is 0.000019
librune:  starting time difference between 172.16.3.62 and 172.16.3.63 is 0.000043
librune:  starting time difference between 172.16.3.63 and 172.16.3.64 is 0.000044
```

6.4 先行技術との比較

ここではいくつかの基準に沿ってRUNEと先行技術の比較を行う．比較の指標として，拡張性，実時間性，実コードの利用の可否，シミュレーション構成変更の容易性，移動モデルと通信モデルの提供の有無を用いた．拡張性の評価は，シミュレーションの負荷が大きくなった場合にシミュレーションの実行者がシミュレーションに利用するノードを増すことによって克服できる場合がある場合に拡張性が有るとした．実時間性に関してはシミュレーションの実行を実時間で行うための機能を提供している，もしくは有している場合に実時間性が有るとした．シミュレーション構成変更の容易性はシミュレーションの構成を変更する場合のコストを抑える機能を提供している場合に有るとした．実コードの利用の可否はシミュ

レーション対象が利用するコードを変更することなくシミュレーションに利用することが可能な場合に可能であるとした．移動モデルと通信モデルの提供は，シミュレーション対象の移動や通信をシミュレートする際に利用可能なモデルを提供している場合に有るとした．

拡張性の面では，MobiNet や RUNE ではシミュレーションの実行者がシミュレーションに利用するノード数を決定することが可能である．実時間性では，MobiNet はネイティブなアプリケーションをエミュレーションやシミュレーションを経ずに実行するため，その実行は実時間であると考えることが可能である．RUNE では RUNE manager がノード間の同期を行いながら Space の周期的な呼び出しを行うことで実時間実行の支援を行う．シミュレーション構成変更の容易性では全てのシミュレータで何らかの対応が行われている．実コードの利用は TOSSIM，ATEMU，MobiNet，RUNE で可能となっている．通信モデルは RUNE 以外の全てシミュレータで提供されている．移動モデルに関しては MobiNet，MobiReal で提供されており，TOSSIM，ATEMU では言及されていない．RUNE ではこれらのいずれも直接対応してはいないが，多目的ネットワークエミュレータ QOMET がこれらの機能を提供している．

第 7章

より高度なシミュレーション環境の構築にむけて

この章では現在までに RUNE を用いてシミュレーションを行うことによって得られた知見から，ユビキタスネットワークシミュレーションプラットフォームが今後備えていくべき機能に関する議論を行う．

ユビキタスネットワークに限らず，シミュレーションを行う際には，同じシミュレーションを複数回行った場合に必要であれば同じ結果が何度でも得られることが必要である．そのためにはシミュレーションの再現性を保証する機構が必要となる．

また，シミュレーションの結果が現実の事象やシステムの挙動に対して十分な忠実性を有しているかを判断する機構が提供されていればシミュレーションプラットフォームの利用者によって大変有用である．

RUNE のように，既存のシミュレーション要素をできるだけ活用しつつ大規模シミュレーションを行おうとするプラットフォーム上では時間に対して様々な概念をもつシミュレーションターゲットが混在することになる．このような状況で個々のシミュレーションターゲットが有効に働くためには何が必要かも検討が必要となる．

以下では，こうしたより高度なシミュレーション環境を実現する機能に関する検討を行う．

7.1 再現性の保証

シミュレーションの再現性を得るためには，再現性を損なう要素，つまりシミュレーションの結果に乱数性をもたらす要素を制御する必要がある．シミュレーションの結果に乱数性をもたらす要素には，大別して

- 情報に含まれる乱数性
- シミュレーションの実行における時間的乱数性

の二種類がある．前者は，主に統計的な性質に従うシミュレーションから生成される情報が持つ乱数性である．後者は RUNE のように多数のモジュールが集合して全体のシミュレーションを構成する場合に問題となる．これは例えば複数のモジュールが通信を行うシミュレーションを行う場合に，実行の順序が入れ替わってしまうとパケットの到着順序も影響を受けるといった要因から発生する．実際にはこれらの情報の乱数性と時間の乱数性を独立したものではなく，通信メディアの特性を情報的乱数性を用いて統計的にシミュレートすることによってパケットの再送率が変化し時間的乱数性を招く，時間的乱数性によってモジュールの実行順序が前後すると取得される乱数値が変化し情報的乱数性が生じる，というように互いに影響を及ぼす場合がある．

こうした不必要な乱数性がシミュレーションの結果に与える影響を排除するためにシミュレーションプラットフォームにおいて実行可能な対策としては，以下のような方法が有効であると考えられる．

- Space 毎に毎回同一の出力系列を生成可能な乱数生成機能の提供
  これは例えば以下のコードのように RUNE 内部で Space 毎のコンテキスト値を管理し，この値を用いて Space 毎に決まった系列の乱数を生成する rand() や srand() 互換の関数を提供することで，Space 間での実行順序が乱数の生成に影響を及ぼさなくするもので，少なくとも Space 内での乱数の生成順序を一定とする効果が得られる．

```
static ctx[nspcs];

void
runeSrand(int gsid, unsigned seed)
```

```
{
    ctx[gsid] = seed;
}

int
runeRand(int gsid)
{
    return(rand_r(ctx[gsid]));
}
```

- 静的 (オフライン) スケジューリングのサポート
  静的スケジューリングのサポートを行うためには，例えば現在 RUNE における Space のエントリポイントの定義,

```
typedef struct {
    void *(*init)(int gsid);
    int (*step)(void *elem);
    void (*fin)(void *elem);
    void *(*read)(void *p, void *a);
    void *(*write)(void *p, void *a);
} entryPoints;
```

  を

```
typedef struct {
    void *(*init)(int gsid);
    int (*step)(void *elem);
    void (*fin)(void *elem);
    void *(*read)(void *p, void *a);
    void *(*write)(void *p, void *a);
    unsigned int interval;
} entryPoints;
```

  のようにし，各 Space のエントリポイント定義時に宣言した呼び出し間隔をもとに予めスケジューリングを行うことで実現可能である．これにより，動的な呼び出し間隔の変更は不可能となるが，時間的な変動による不必要な乱数性の発生を抑えることが可能となる．C 言語の文法では,

```
typedef struct {
    void *(*init)(int gsid);
    int (*step)(void *elem);
    void (*fin)(void *elem);
    void *(*read)(void *p, void *a);
    void *(*write)(void *p, void *a);
    unsigned int interval;
} entryPoints;
```

として宣言された型に対して，

```
entryPoints ep = {
.init = myspace_init,
.step = myspace_step,
.fin = myspace_fin,
.read = myspace_read,
.write = myspace_write
};
```

という定義を行うことは問題がなく，またこの ep は Space オブジェクトの .data セクションに置かれるため，メンバの初期値は 0 となる．そこで，RUNE では ep.interval の値を確認し，0 であれば通常のスケジューリングを，非 0 であれば静的スケジューリングを行うということが可能である．

こうした対策でさえ完全には抑えられないような微小の時間的乱数性が問題になる場合には，問題の原因に応じ

- シミュレーションの実行時間を実時間に対して n 倍とする
- パケットに付けたタイムスタンプに基づく受信時キューイングを行う

などの対策が考えられる．

7.2 結果の正確さの判定

シミュレーションから得られた結果がどの程度現実の事象やシステムの挙動を模倣しているかといった正確さを判定することは大変難しい．特に，RUNE のように，異なる目標精度を持つシミュレーションターゲットが混在してシミュレーションを構成することを許すプラットフォーム上で実行されたシミュレーションは，単純に最も精度の劣るターゲットと同程度の精度で結果が得られるという訳ではない．従って，シミュレーション全体の精度を求めることは容易ではない．しかし，シミュレーションが局所的に破綻を来したことを検知し，その時刻と発生箇所を利用者にフィードバックすることができれば大変有用である．こうした機能を実現するためにシミュレーションプラットフォーム側で実現可能な機能としては，

- シミュレーションターゲット実行時のデッドラインミスの検出と通知

  RUNE のスケジューラで Space の次回実行時間の要求と現時刻との比較を行うことにより検出する

- シミュレーションターゲット間での異常値の通信の検出

  RUNE が Conduit を介して行われる値の送受信を監視し，例えば IEEE754 で定義される非数などが含まれていないかを監視する

などが考えられる．こうした異常を検出した場合にシミュレーションを中止するべきなのか，それとも他の処理を行ってからシミュレーションを続行するのかはシミュレーション依存である．

7.3 様々な時間の概念を持つシミュレーションの混在

シミュレーションを分類する場合に，一つの分類として周期性を持つ (時間起動) シミュレーションとイベント駆動 (イベント起動) シミュレーションという尺度が考えられる．時間起動はさらに，デッドラインミスを絶対に許容しないハードリアルタイムシミュレーションとある程度のデッドラインミスを許容するソフトリアルタイムシミュレーションに分類することができる．これらの時間起動シミュレーションは対象とするシミュレーションの種類により，計算周期はまちまちである．特殊な場合では，計算周期が変動するシミュレーションも存在する．

RUNE では，こうした時間起動とイベント起動のどちらもシミュレーションを構成するモジュールとして利用することが可能であり，両者の混在も可能である．時間起動のシミュレーションの場合には RUNE に周期的なモジュールの呼び出しを依頼する．イベント起動のシミュレーションの場合には RUNE のスケジューラに頼らず，モジュール内部でイベント待ちをすることが可能である．

イベント起動，時間起動 （ハードリアルタイム，ソフトリアルタイム）とそれぞれ異なる種類のシミュレーションを動作させるためには各シミュレーション間でのスケジューリングが重要である．時間起動のシミュレーションのみを考えた場合，スケジューリングアルゴリズムとして一般的なものには以下のようなものがある．

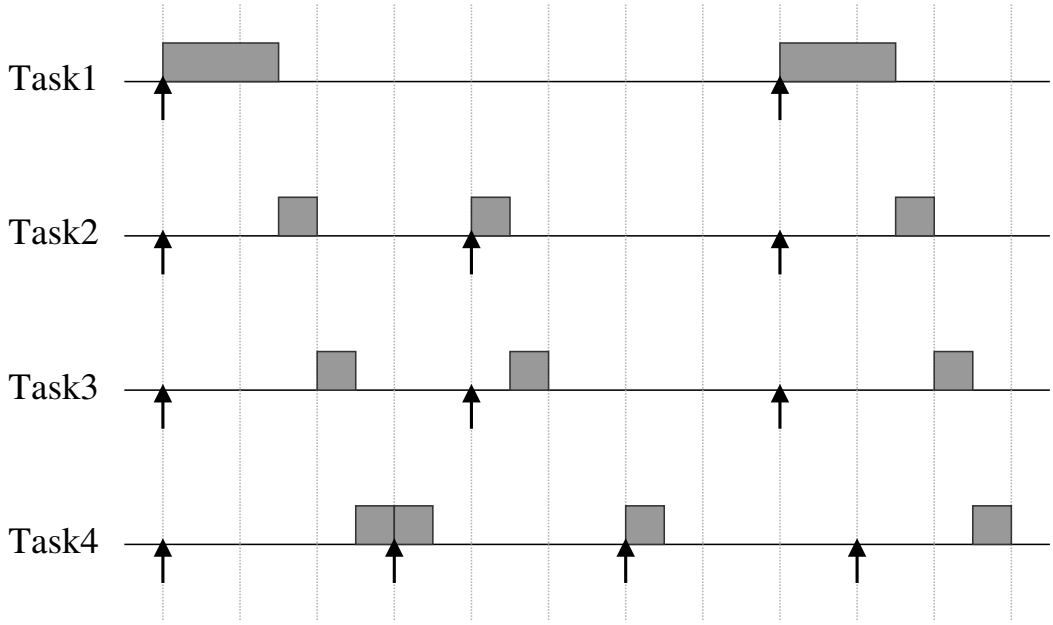

図 7.1: First In First Out Algorithm

- First In First Out （FIFO）

  First In First Out アルゴリズムは図 7.1 で示すように，処理が発生した時点の早いものから実行を行うアルゴリズムである．このアルゴリズムでは，タスクの要求次第では単一のタスクが連続して資源を独占してしまうおそれがあり，公平性は確保できない．また，周期タスクの周期性は保証されない．

- Round Robin （RR）

  Round Robin アルゴリズムは図 7.2 で示すように，各タスクに一定時間を割り当て，順に処理を行うアルゴリズムである．このアルゴリズムでは，各タスクを機械的に実行するため，タスクのデッドラインに対する余裕は考慮されない．

- Earliest Deadline First （EDF）

  Earliest Deadline First アルゴリズムは図 7.3 で示すように，デッドラインが早いタスクから順に処理を行うアルゴリズムである．このアルゴリズムでは，タスクがデッドラインミスまでに実行を終えられない処理の量を最小と

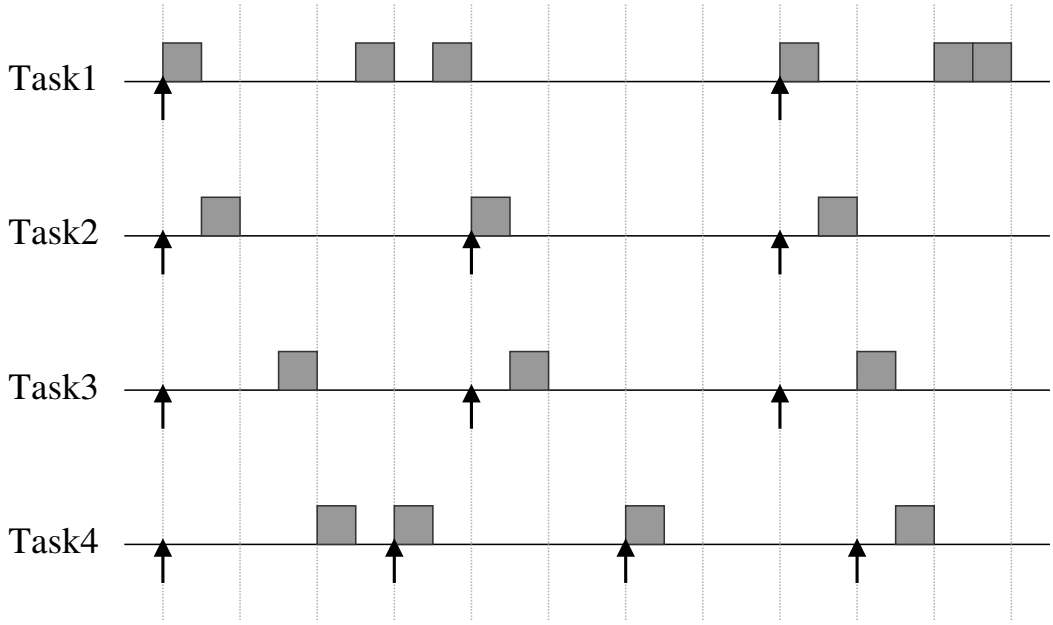

図 7.2: Round Robin Algorithm

することができる．

- Rate Monotonic （RM）

  Rate Monotonic アルゴリズムは図7.4 で示すように，起動周期の短いタスクを優先的にスケジュールするアルゴリズムである．このアルゴリズムでは，短時間の周期性を持つタスクの周期性を確保することができるが，動的に起動周期が変動するタスクには対応できない．

シミュレーションのサイクルでは，ある時間 t の状態を時間 t までに計算するという動作の繰り返しであり，デッドライン t を基準にしてスケジューリングを行うと必要な状態が時系列に沿った状態で得られる．この点がシミュレーションの実行に有利に働くため，現状の RUNE では，次回タスク実行時の時刻を基準としてスケジューリングを行うアルゴリズムを利用している．

シミュレーションプラットフォームのスケジューリングを考えた場合，通常のスケジューリングアルゴリズムが動作の指標とする資源の公平な利用や有効利用といった側面よりも，デッドラインミスの管理がより強く求められる．これは，デッドラインミスがシミュレーションから得られる結果に大きな影響を及ぼす恐れが

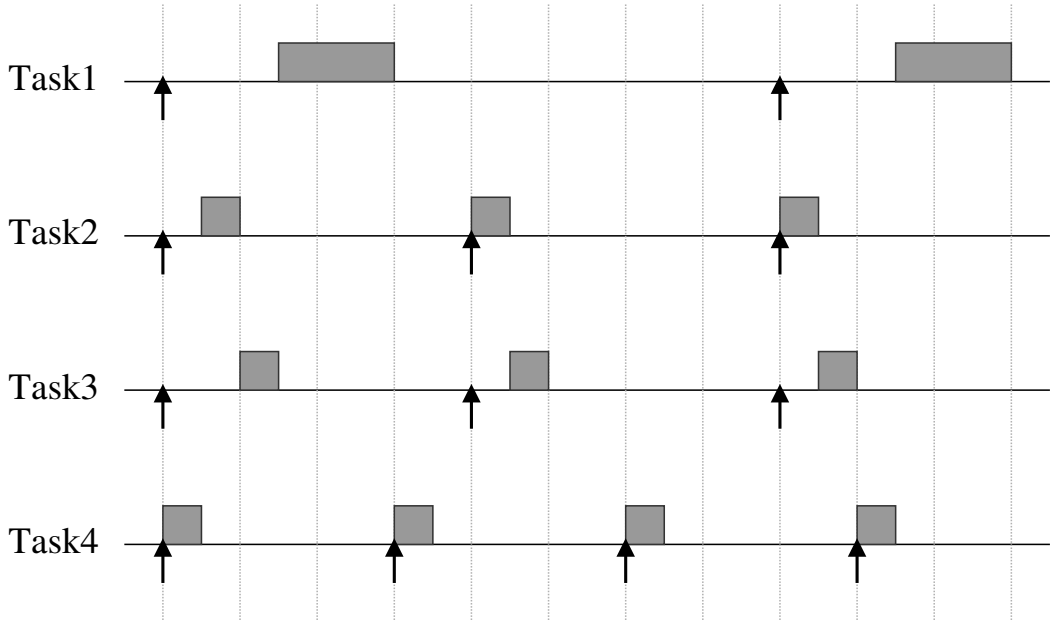

図 7.3: Earliest Deadline First Algorithm

あるためである．また，デッドラインミスを発生させないようにすることももちろん重要ではあるが，デッドラインミスが発生した時に適切に通知することが重要である．RUNE のようにハードデッドラインシミュレーションとソフトデッドラインシミュレーションが混在するシミュレーションを実行する可能性のあるプラットフォームでは，プラットフォーム側でデッドラインミスに対する処置を行うことはできないため，シミュレーションターゲット側にデッドラインミスの通知を行う．この通知はシミュレーションターゲットのデッドラインミスハンドラを呼び出すことで行う．シミュレーションを即座に中止するのかしばらく経過を観察するかは通知を受けたシミュレーションターゲット側で判断を行う．動的に精度を変化させるシミュレーションターゲットの場合にはこのデッドラインミスハンドラの呼び出しを契機として精度を変化させることも可能である．また，デッドラインミスを起こさない場合であっても，シミュレーションターゲットが RUNE が提供するシミュレーション時刻を取得する機能を呼び出すことでデッドラインに対してどれだけ余裕を持って処理を終えたかを常に把握することが可能である．

実際にデッドラインミスや 7.2 節で述べた Conduit における異常値の送受信といったフォールトハンドルの機構を RUNE に持たせるための方法として，現在の

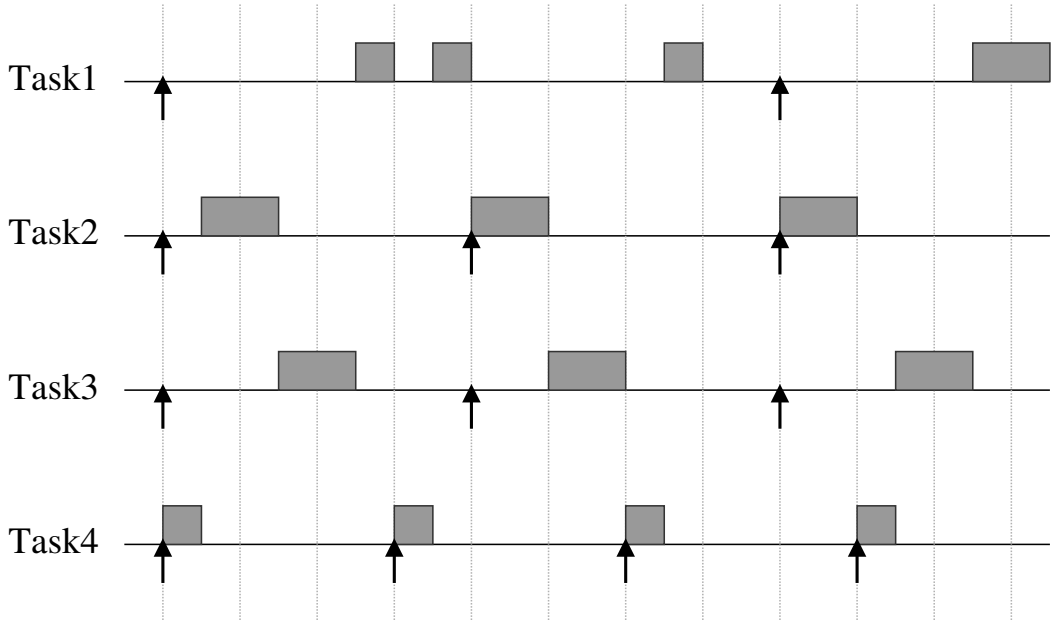

図 7.4: Rate Monotonic Algorithm

RUNE における Space のエントリポイントの定義

```
typedef struct {
    void *(*init)(int gsid);
    int (*step)(void *elem);
    void (*fin)(void *elem);
    void *(*read)(void *p, void *a);
    void *(*write)(void *p, void *a);
} entryPoints;
```

を

```
typedef struct {
    void *(*init)(int gsid);
    int (*step)(void *elem);
    void (*fin)(void *elem);
    void *(*read)(void *p, void *a);
    void *(*write)(void *p, void *a);
    int (*faulthandler)(void *elem, int reason);
} entryPoints;
```

のようにし，フォールトハンドラを定義するようにする．RUNE は実際にフォールトが発生すると， ep.faulthandler の値を確認し， NULL でなければその原因

を引数としてフォールトハンドラを呼び出す．フォールトハンドラでは，シミュレーションを中止する，フォールトを記録する，フォールトの発生頻度などをもとにシミュレーションを中止するなどの処理を行うことができる．この変更は 7.1 節での変更と同様に現行の Space の実装との互換性を損なうことはない．

以下にフォールトハンドラの実装例を示す．

- 以下の例では，デッドラインミスが発生すると即座にシミュレーションを中止し，異常値の送受信に対しては何も行わず，実行を継続する．

```
int
faulthandler(void *elem, int reason)
{
    switch(reason) {
    case RUNEFAULTDEADLINEMISS:
        return -1;
    case RUNEFAULTABNORMALVALUE:
        myspacestep(elem);
        return 0;
    }
}
```

- 以下の例では，異常値の送受信の発生回数を記録する．

```
int
faulthandler(void *elem, int reason)
{
    myspacestruct *s = elem;

    switch(reason) {
    case RUNEFAULTDEADLINEMISS:
        break;
    case RUNEFAULTABNORMALVALUE:
        s.fault++;
        break;
    }
    myspacestep(elem);
    return 0;
}
```

- 以下の例では，デッドラインミスの発生回数が 100 回を越えるとシミュレーションを中止する．

```
int
```

```
faulthandler(void *elem, int reason)
{
    myspacestruct *s = elem;

    switch(reason) {
    case RUNEFAULTDEADLINEMISS:
        if(++s.fault > 100)
            return -1;
        break;
    case RUNEFAULTABNORMALVALUE:
        break;
    }
    myspacestep(elem);
    return 0;
}
```

- 以下の例では，デッドラインミスが10回連続するとシミュレーションを中止する．

```
int
myspacestep(void *elem)
{
    myspacestruct *s = elem;

        ...

    s.fault = (s.fault << 1) & 0x000003ff;
    return 0;
}

int
faulthandler(void *elem, int reason)
{
    unsigned int nofb;
    myspacestruct *s = elem;

    switch(reason) {
    case RUNEFAULTDEADLINEMISS:
        s.fault = (s.fault << 1) & 0x000003ff + 1;
        if(s.fault == 0x000003ff);
            return -1;
        break;
    case RUNEFAULTABNORMALVALUE:
        break;
    }
    myspacestep(elem);
    return 0;
}
```

- 以下の例では，直近10回の呼び出しにおいて，5回以上のデッドラインミスが発生するとシミュレーションを中止する．

```
int
myspacestep(void *elem)
{
    myspacestruct *s = elem;

        ...

    s.fault = (s.fault << 1) & 0x000003ff;
    return 0;
}

int
faulthandler(void *elem, int reason)
{
    unsigned int nofb;
    myspacestruct *s = elem;

    switch(reason) {
    case RUNEFAULTDEADLINEMISS:
        s.fault = (s.fault << 1) & 0x000003ff + 1;
        nofb = ((s.fault & 0xaaaaaaaa) >> 1) + (s.fault & 0x55555555);
        nofb = ((nofb & 0xcccccccc) >>  2) + (nofb & 0x33333333);
        nofb = ((nofb & 0xf0f0f0f0) >>  4) + (nofb & 0x0f0f0f0f);
        nofb = ((nofb & 0xff00ff00) >>  8) + (nofb & 0x00ff00ff);
        nofb = ((nofb & 0xffff0000) >> 16) + (nofb & 0x0000ffff);
        if(nofb > 4);
            return -1;
        break;
    case RUNEFAULTABNORMALVALUE:
        break;
    }
    myspacestep(elem);
    return 0;
}
```

こうしたフォールトハンドラを利用することでシミュレーション毎に異なるフォールトへの対応を適切に扱うことが可能となる．

第 8章

まとめ

本論文では，ユビキタスネットワークシステムのシミュレーションを行うためのプラットフォームである RUNE について述べた．RUNE core では，動作基盤として StarBED を利用することによって，多数のノードから構成されるシミュレーションに対応可能となっている．また，シミュレーション定義ファイルによる構成の定義と RUNE master と RUNE manager による自動実行によってクラスタ環境を意識しないシミュレーションの実行とシミュレーションの構成変更を容易にしている．さらに，Space と Conduit を用いたシミュレーション対象の実装，およびマルチレベルエミュレーションレイヤの提供により，様々なネットワーク，ハードウェア，ソフトウェアアーキテクチャのエミュレーション，さらには周囲の環境のシミュレーションも可能としている．これらに加え，マルチレベルエミュレーションレイヤの提供は開発の段階に応じて複数の抽象度でのシミュレーションを可能としている．

このような機能を持つ RUNE を用いて現在までに様々なシミュレーションが実行された．本論文ではこれらのうち，代表的なものについて説明を行い，これらのシミュレーションから得られた知見について述べた．

また，より高度なシミュレーションにも対応し，実用性の高いプラットフォームとするためには何が必要となるかについて議論を行った．

謝辞

本研究を遂行するにあたり終始御指導を賜わりました指導教官の北陸先端科学技術大学院大学情報科学研究科丹康雄教授に深く感謝致します．また，本研究を行う過程で示唆に富んだ多くの助言を与えて下さいました北陸先端科学技術大学院大学情報科学センター篠田陽一教授，北陸先端科学技術大学院大学日比野靖副学長，北陸先端科学技術大学院大学情報科学研究科 Defago Xavier 准教授，東京工業大学権藤克彦准教授に深い謝意を表します．

研究者としての活躍の場を与えて下さいました北陸先端科学技術大学院大学，ソニーケミカル&インフォメーションデバイス株式会社，北陸日本電気ソフトウェア株式会社，独立行政法人情報通信研究機構の関係者各位に感謝致します．

国際会議における優秀論文賞の受賞という栄誉に浴する原動力となる研究を共同で遂行して下さいました船井電機株式会社鈴木良宏様，パナソニック株式会社川上哲也様，独立行政法人情報通信研究機構北陸リサーチセンター Razvan Beuran 博士，北陸先端科学技術大学院大学情報科学研究科助教知念賢一博士，北陸先端科学技術大学院大学情報科学研究科岡田崇氏，北陸先端科学技術大学院大学情報科学研究科芳炭将氏，北陸先端科学技術大学院大学情報科学研究科 Khin Thida Latt 氏，北陸先端科学技術大学院大学インターネット研究センター向千昌氏に深謝致します．

研究以外の活躍の場を与えてくださいました株式会社カットシステム，有限会社スペースソフト，JAF 加盟クラブアールエイト石川，獅子吼高原パラグライダースクール，North Mon Language Institute の関係者各位に御礼申し上げます．

本研究は StarBED なくしては成し得ませんでした．研究のあらゆる過程で多大なる御助言，御支援を頂きました The StarBED Team の皆様，篠田陽一プロジェクトリーダー，丹康雄プロジェクトサブリーダー，知念賢一研究員，Razvan Beuran 研究員，宮地利幸研究員，三輪信介研究員，太田悟史技術員，宇多仁研究

員，小原泰弘研究員，岡田崇研究員，芳炭将研究員，堀雅和研究員，増井健司研究員，佐野正行技術員，竹中ゆかり氏，向千昌氏，福地美郎氏に感謝致します．

最後に，研究生活を様々な面で支援してくれた家族に感謝します．

参考文献

[1] P. Levis, N. Lee, M. Welsh, D. Culler, “TOSSIM: Accurate and Scalable Simulation of Entire TinyOS App lications”, the First ACM Conference on Embedded Networked Sensor Systems (SenSys2003), Los Angeles, California, U.S.A., 2003

[2] J. Polley, D. Blazakis, J. McGee, D. Rusk, J. S. Baras, “ATEMU: A Fine-grained Sensor Network Simulator”, the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON 2004), Santa Clara, California, U.S.A., 2004

[3] P. Mahadevan, A. Rodriguezand D. Becker, A. Vahdat, “MobiNet: A Scalable Emulation Infrastructure for Ad Hoc and Wireless Networks”, International Workshop on Wireless Traffic Measurements and Modeling (WiTMeMo) in conjunction with MobiSys, Seattle, WA, June 2005.

[4] Kazuki Konishi, Kumiko Maeda, Kazuki Sato, Akiko Yamasaki, Hirozumi Yamaguchi, Keiichi Yasumoto, Teruo Higashino, “MobiREAL Simulator – Evaluating MANET Applications in Real Environments”, Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS2005) pp. 499–502, 2005

[5] T. Miyachi , K. Chinen , Y. Shinoda, “Automatic Configuration and Execution of Internet Experiments On An Actual Node-based Testbed”, 1st International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities(Tridentcom) (2005) 274–282, 2005

[6] Toshiyuki Miyachi, Ken-ichi Chinen and Yoichi Shinoda, "Automatic Configuration and Execution of Internet Experiments on an Actual Node-based Testbed", International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom2005), pp.274–282 (2005).

[7] R. Beuran, L. T. Nguyen, K. T. Latt, J. Nakata, Y. Shinoda, "QOMET: A Versatile WLAN Emulator", IEEE International Conference on Advanced Information Networking and Applications (AINA2007), pp.348–353, Niagara Falls, Canada, May 2007.

[8] J. Nakata, R. Beuran, T. Miyachi, K. Chinen, S. Uda, K. Masui, Y. Tan, and Y. Shinoda, "StarBED2: Testbed for Networked Sensing Systems", International Conference on Networked Sensing Systems (INSS2007), pp.142–145, Braunschweig, Germany, Jun. 2007.

[9] T. Okada, R. Beuran, J. Nakata, Y. Tan, and Y. Shinoda, "Collaborative Motion Planning of Autonomous Robots", 3rd International Conference on Collaborative Computing (CollaborateCom 2007), White Plains, NY, USA, Nov. 2007.

[10] J. Nakata, R. Beuran, T. Kawakami, K. Chinen, Y. Tan, Y. Shinoda, "Distributed Emulator for a Pedestrian Tracking System Using Active Tags", 2nd International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM2008), Valencia, Spain, September 29-October 4, 2008, pp. 219-224.

[11] R. Beuran, J. Nakata, T. Okada, Y. Suzuki, T. Kawakami, K. Chinen, Y. Tan, Y. Shinoda, "Active Tag Based Pedestrian Localization Emulation System", demonstration, 5th International Conference on Networked Sensing Systems (INSS2008), Kanazawa, Ishikawa, Japan, June 17-19, 2008, pp. 258.

[12] http://www.iet.unipi.it/ luigi/ip_dummynet/

[13] http://www.microchip.com/

[14] http://www.opencores.org/

[15] http://www.opencores.org/projects.cgi/web/or1k/openrisc_1200/

[16] http://www.zigbee.org/

[17] http://www.bluetooth.com/

[18] http://www.echonet.gr.jp/

[19] http://www.dlna.org/

[20] http://www.upnp.org/

本研究に関する発表論文

[1] J. Nakata, R. Beuran, T. Kawakami, K. Chinen, Y. Tan, Y. Shinoda: "Distributed Emulator for a Pedestrian Tracking System Using Active Tags", International Journal on Advances in Intelligent Systems 投稿中

[2] 中田 潤也, R. Beuran, 向 千昌, 川上 哲也, 岡田 崇, 知念 賢一, 丹 康雄, 篠田 陽一: "アクティブタグを利用した歩行者位置推定システムの分散シミュレーション", インターネットコンファレンス 2008 (IC20008), ポスターセッション, 沖縄, 2008 年 10 月 23 日〜10 月 24 日, pp. 126

[3] J. Nakata, R. Beuran, T. Kawakami, K. Chinen, Y. Tan, Y. Shinoda: "Distributed Emulator for a Pedestrian Tracking System Using Active Tags", 2nd International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM2008), **Awarded as the best paper**, Valencia, Spain, September 29-October 4, 2008, pp. 219-224.

[4] J. Nakata, R. Beuran, T. Miyachi, K. Chinen, S. Uda, K. Masui, Y. Tan, Y. Shinoda: "StarBED2: Testbed for Networked Sensing Systems", 4th International Conference on Networked Sensing Systems (INSS2007), BoF, Braunschweig, Germany, June 6-8, 2007, pp. 142-145.

[5] J. Nakata, T. Miyachi, R. Beuran, K. Chinen, S. Uda, K. Masui, Y. Tan, Y. Shinoda: "StarBED2: Large-scale, Realistic and Real-time Testbed for Ubiquitous Networks", TridentCom 2007, Orlando, Florida, U.S.A., May 21-23, 2007.

[6] Junya Nakata, Toshiyuki Miyachi, Ken-ichi Chinen, Yasuo Tan and Yoichi Shinoda: "StarBED2: Real-time Testbed for Ubiquitous Networks", 3rd

International Conference on Networked Sensing Systems (INSS2006), Poster, ISBN: 0-9743611-3-5, p.145, Illinois, USA, Jun. 2006.

[7] J. Nakata, Y. Tan: "The Design and Implementation of Large Scale Ubiquitous Network Testbed", Workshop on Smart Object Systems in Conjunction with the Seventh International Conference on Ubiquitous Computing (UbiComp 2005), Tokyo, Japan, Sep 11, 2005.

[8] R. Beuran, J. Nakata, T. Okada, T. Kawakami, K. Chinen, Y. Tan, Y. Shinoda: "Emulation System for Active Tag Applications", 4th International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP 2008), Sydney, Australia, December 15-18, 2008.

[9] T. Okada, J. Nakata, R. Beuran, Y. Tan, Y. Shinoda: "Large-scale Simulation Method of Mobile Robots", 2nd International Symposium for Universal Communication (ISUC2008), Osaka, Japan, December 15-16, 2008.

[10] R. Beuran, J. Nakata, T. Okada, T. Kawakami, K. Chinen, Y. Tan, Y. Shinoda: "Emulation of an Active Tag Location Tracking System", Ambient Intelligence Forum (AMIF2008), Hradec Kralove, Czech Republic, October 15-16, 2008, pp. 53-60.

[11] R. Beuran, J. Nakata, T. Okada, Y. Suzuki, T. Kawakami, K. Chinen, Y. Tan, Y. Shinoda: "Active Tag Based Pedestrian Localization Emulation System", demonstration, 5th International Conference on Networked Sensing Systems (INSS2008), Kanazawa, Ishikawa, Japan, June 17-19, 2008, pp. 258.

[12] R. Beuran, J. Nakata, Y. Suzuki, T. Kawakami, K. Chinen, Y. Tan, Y. Shinoda: "Active Tag Emulation for Pedestrian Localization Applications", 5th International Conference on Networked Sensing Systems (INSS2008), Kanazawa, Ishikawa, Japan, June 17-19, 2008, pp. 55-58.

[13] R. Beuran, J. Nakata, T. Okada, L. T. Nguyen, Y. Tan, Y. Shinoda: "A Multi-purpose Wireless Network Emulator: QOMET", 22nd IEEE Inter-

national Conference on Advanced Information Networking and Applications (AINA2008) Workshops, FINA2008 symposium, Okinawa, Japan, March 25-28, 2008, pp. 223-228.

[14] R. Beuran, J. Nakata, T. Okada, Y. Tan, Y. Shinoda: "Real-time emulation of networked robot systems", 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2008), Marseille, France, March 3-7, 2008.

[15] 宮地 利幸, 中田 潤也, 知念 賢一, Razvan Beuran, 三輪 信介, 岡田 崇, 三角 真, 宇多 仁, 芳炭 将, 丹 康雄, 中川 晋一, 篠田 陽一, "StarBED：大規模ネットワーク実証環境 ", 情報処理学会学会誌, Vol. 49, No. 1, 2008 年 1 月

[16] T. Okada, J. Nakata, R. Beuran, Y. Tan, Y. Shinoda: "Motion Planning of Autonomous Robots", demonstration, Adjunct Proceedings of 4th International Symposium on Ubiquitous Computing Systems (UCS 2007), Tokyo, Japan, November 25-28, 2007, pp. 57-58.

[17] R. Beuran, J. Nakata, T. Okada, T. Miyachi, K. Chinen, Y. Tan, Y. Shinoda: "Performance Assessment of Ubiquitous Networked Systems", 5th International Conference on Smart Homes and Health Telematics (ICOST2007), Nara, Japan, June 21-23, 2007, pp. 19-26.

[18] 福田 隆弘, 中田 潤也, 岡田 崇, 丹 康雄, "ホームネットワークを用いたプッシュ型情報のユーザ提示システム", 平成 19 年度電気関係学会北陸支部連合大会, 2007.

[19] T. Miyachi, J. Nakata, R. Beuran, K. Chinen, K. Masui, S. Uda, Y. Tan, Y. Shinoda: "Realistic Simulation of Internet", Asian Simulation Conference 2006 (ASC2006), Tokyo, Japan, October 30-November 1, 2006, pp. 386-390.

[20] Yoshiki Makino, Junya Nakata, Yasuo Tan,"Integrated Visual Communication Network Architecture", International Symposium on Towards Peta-Bit Ultra-Networks (PB2003), Ishikawa, Japan, Sep 8 – 9, 2003

[21] 中村 太一, 中田 潤也, 牧野 義樹, 丹 康雄, “ホームネットワークにおけるUI（ユーザインタフェース）の統一的なAPI提供に向けたUI要素の分類”, 電子情報通信学会2006ソサイエティ大会講演論文集, 2006年9月

[22] 川上 哲也, 鈴木 良宏, 中田 潤也, Razvan Beuran, 丹 康雄, 篠田 陽一, “階層化アクティブタグと歩行者位置推定”, 信学技報, vol. 107, no. 524, NS2007-152, pp. 123-128, 2008年3月.

[23] T. Okada, R. Beuran, J. Nakata, Y. Tan, Y. Shinoda: “Collaborative Motion Planning of Autonomous Robots”, 3rd International Conference on Collaborative Computing (CollaborateCom 2007), White Plains, New York, U.S.A., November 12-15, 2007.

[24] R. Beuran, T. Okada, J. Nakata, T. Miyachi, K. Chinen, Y. Tan, Y. Shinoda: “Network-enabled Sensing Robot Emulation”, demonstration, 4th International Conference on Networked Sensing Systems (INSS2007), Braunschweig, Germany, June 6-8, 2007, pp. 308.

[25] 今井 智大, 岡田 崇, 中田 潤也, 丹 康雄, “ホームネットワークにおけるリモート管理を考慮したサービスインタフェースに関する提案”, 情報処理学会ユビキタスコンピューティングシステム研究会研究報告, Vol. 2008, No. 66, 2008年7月

[26] 磯貝 彰則, 牧野 義樹, 中田 潤也, 丹 康雄, “近距離無線通信規格ZigBeeにおけるノードグルーピング方式”, 電子情報通信学会2006ソサイエティ大会講演論文集, 2006年9月

[27] 相川 恵, 牧野 義樹, 中田 潤也, 丹 康雄, “ホームネットワークの障害診断に関する研究”, 電子情報通信学会2006ソサイエティ大会講演論文集, 2006年9月

[28] 増田 耕一, 牧野 義樹, 中田 潤也, 丹 康雄, “家電製品使用における異常状態のモデル化とその検知手法の提案”, 情報処理学会ユビキタスコンピューティングシステム研究会研究報告, Vol. 2006, No. 54, 2008年5月

[29] R. Beuran, L. T. Nguyen, K. T. Latt, J. Nakata, Y. Shinoda: "QOMET: A Versatile WLAN Emulator", 21st IEEE International Conference on Advanced Information Networking and Applications (AINA2007), Niagara Falls, Ontario, Canada, May 21-23, 2007, pp. 348-353.

[30] R. Beuran, Y. Shinoda, S. Nakagawa, J. Nakata, T. Miyachi, K. Chinen, Y. Tan: "Performance Analysis of VoIP over WLAN", Multimedia, Distributed, Cooperative and Mobile Symposium (DICOMO) 2006, Kagawa, Japan, July 5-7, 2006, pp. 849-852.

[31] R. Beuran, K. Chinen, K. T. Latt, T. Miyachi, J. Nakata, L. T. Nguyen, Y. Shinoda, Y. Tan: "Application Performance Assessment on Wireless Ad Hoc Networks", Asian Internet Engineering Conference (AINTEC) 2006, Springer-Verlag LNCS 4311, Bangkok, Thailand, November 28-30, 2006, pp. 128-138.

[32] R. Beuran, K. Chinen, K. T. Latt, T. Miyachi, J. Nakata, L. T. Nguyen, Y. Shinoda, Y. Tan, S. Uda, S. Zrelli: "WLAN Emulation on StarBED", IET International Conference on Wireless, Mobile & Multimedia Networks (ICWMMN) 2006, Hangzhou, China, November 6-9, 2006, pp. 856-859.

第 A章

PICエミュレータ

A.1　PIC 16F648Aのアーキテクチャ

A.1.1　PIC 16F648Aのインストラクションセット

PIC 16F648Aの命令セットはバイト指向レジスタ操作，ビット指向レジスタ操作，定数/制御操作等の命令群からなる計35命令 (表A.1) で構成されている．演算命令にはそれぞれアキュムレータとレジスタ，または定数間の移動，加減算，アキュムレータやレジスタに対する0クリア，増減，論理和/論理積/排他的論理和/否定，回転，交換，ビットセット/ビットクリアが含まれる．制御命令にはジャンプ，コール，ウォッチドッグタイマカウンタのリセット，リターン，割り込みからのリターン，スリープステートへの移行等が含まれる．これらの他にテスト命令として，加減算，ビットテストがあり，これらの命令では，条件が成立すると次の命令をスキップするという動作を行う．

表 A.1: PIC 16F648A の命令セット

Mnemonic	Operands	Description	Cycles	Opcode				Status Affected
BYTE-ORIENTED FILE REGISTER OPERATIONS								
ADDWF	f,d	Add W and f	1	00	0111	dfff	ffff	C,DC,Z
ANDWF	f,d	And W with f	1	00	0101	dfff	ffff	Z
CLRF	f	Clear f	1	00	0001	1fff	ffff	Z
CLRW	–	Clear W	1	00	0001	0xxx	xxxx	Z
COMF	f,d	Complement f	1	00	1001	dfff	ffff	Z
DECF	f,d	Decrement f	1	00	0011	dfff	ffff	Z
DECFSZ	f,d	Decrement f, Skip if 0	1(2)	00	1011	dfff	ffff	
INCF	f,d	Increment f	1	00	1010	dfff	ffff	Z
INCFSZ	f,d	Increment f, Skip if 0	1(2)	00	1111	dfff	ffff	
IORWF	f,d	Inclusive Or W with f	1	00	0100	dfff	ffff	Z
MOVF	f,d	Move f	1	00	1000	dfff	ffff	Z
MOVWF	f	Move W to f	1	00	0000	1fff	ffff	
NOP	–	No Operation	1	00	0000	0xx0	0000	
RLF	f,d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C
RRF	f,d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C
SUBWF	f,d	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z
SWAPF	f,d	Swap nibbles in f	1	00	1110	dfff	ffff	
XORWF	f,d	Exclusive Or W with f	1	00	0110	dfff	ffff	Z
BIT-ORIENTED FILE REGISTER OPERATIONS								
BCF	f,b	Bit Clear f	1	01	00bb	bfff	ffff	
BSF	f,b	Bit Set f	1	01	01bb	bfff	ffff	
BTFSC	f,b	Bit Test f, Skip if Clear	1(2)	01	10bb	bfff	ffff	
BTFSS	f,b	Bit Test f, Skip if Set	1(2)	01	11bb	bfff	ffff	
LITERAL AND CONTROL OPERATIONS								
ADDLW	k	Add literal and W	1	11	111x	kkkk	kkkk	C,DC,Z
ANDLW	k	And literal with W	1	11	1001	kkkk	kkkk	Z
CALL	k	Call Subroutine	2	10	0kkk	kkkk	kkkk	
CLRWDT	–	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}$,$\overline{PD}$
GOTO	k	Go to Address	2	10	1kkk	kkkk	kkkk	
IORLW	k	Inclusive Or literal with W	1	11	1000	kkkk	kkkk	Z
MOVLW	k	Move literal to W	1	11	00xx	kkkk	kkkk	
RETFIE	–	Return from Interrupt	2	00	0000	0000	1001	
RETLW	k	Return with literal in W	1	11	00xx	kkkk	kkkk	
RETURN	–	Return from Subroutine	2	00	0000	0000	1000	
SLEEP	–	Go into Standby mode	1	00	0000	01110	0011	$\overline{TO}$,$\overline{PD}$
SUBLW	k	Subtract W from literal	1	11	110x	kkkk	kkkk	C,DC,Z
XORLW	k	Exclusive Or literal with W	1	11	1010	kkkk	kkkk	Z

A.1.2 PIC 16F648A の割り込み

PIC 16F648A では I/O ピンの状態変化，コンパレータの入力，E^2PROM への操作終了，USART(Universal Synchronous Asynchronous Receiver Transmitter) への操作終了等をトリガとした割り込みが利用可能となっている他，タイマ 0，タイマ 1，タイマ 2 の 3 つのタイマ割り込みが利用可能となっている．割り込みが発生すると GIE(Global Interrupt Enable) ビットがリセットされ，割り込みが禁止された状態で割り込みベクタである 0004H に制御が移る．

タイマ 0 はタイマモードとカウンタモードで動作が可能となっている．タイマモードで動作する場合，プリスケーラを用いない場合には命令サイクル毎に内部の 8 ビットカウンタが増加され，カウンタがオーバーフローした時点で T0IF(Timer 0 Interrupt Flag) がセットされ，割り込みが生じる．タイマ 0 には 1:2〜1:256 倍で動作するプリスケーラを割り当てることが可能となっており，その場合にはプリスケーラの設定サイクル毎にカウンタが増加される．したがって，タイマ 0 は動作周波数を f とすると，$1 \times 256 \times 4/f = 1,024/f$[s]〜$256 \times 256 \times 4/f = 262,144/f$[s] 周期の範囲で割り込みを発生させることが可能となっている．これは，内蔵の 4MHz オシレータ使用時には 0.256[ms]〜65.536[ms] に相当する．

タイマ 1 はタイマモードとカウンタモードで動作が可能となっている．タイマモードで動作する場合，1:1〜1:8 で動作するプリスケーラに設定されたサイクル毎に内部の 16 ビットカウンタが増加され，カウンタがオーバーフローした時点で T1IF(Timer 1 Interrupt Flag) がセットされ，割り込みが生じる．したがって，タイマ 1 は動作周波数を f とすると，$1 \times 65,536 \times 4/f = 16,384/f$[s]〜$8 \times 65,536 \times 4/f = 131,072/f$[s] 周期の範囲で割り込みを発生させることが可能となっている．これは，内蔵の 4MHz オシレータ使用時には 4.096[ms]〜32.768[ms] に相当する．

タイマ 2 はタイマモードと PWM(Pulse Width Modulation) モードで動作が可能となっている．タイマモードで動作する場合，1:1，1:4，1:16 倍で動作するプリスケーラに設定された命令サイクル毎に内部の 8 ビットカウンタが増加され，1:1〜1:16 倍で動作するポストスケーラに設定された回数だけカウンタがオーバーフローを生じた時点で T2IF(Timer 2 Interrupt Flag) がセットされ，割り込みが生じる．したがって，タイマ 1 は動作周波数を f とすると，$1 \times 1 \times 256 \times 4/f = 1,024/f$[s]〜$16 \times 16 \times 256 \times 4/f = 262,144/f$[s] 周期の範囲で割り込みを発生させることが可能

表 A.2: PIC 16F648A のプログラムメモリマップ

0000H	Reset Vector
0001H	
0002H	
0003H	
0004H	Interrupt Vector
0005H	On-chip Program Memory
0fffH	

となっている．これは，内蔵の 4MHz オシレータ使用時には 0.256[ms]〜65.536[ms] に相当する．

A.1.3 PIC 16F648A の外部 I/O

PIC 16F648A は 16 ピンの PIO を持っている．これらのポートは設定によりアナログコンパレータ，PWM，USART 等の I/O として利用可能となっている．

A.1.4 PIC 16F648A のプログラムメモリマップ

PIC 16F648A は 4,096 ワードのプログラムメモリを持つ．このうち，0000H はリセットベクタ，0004H は割り込みベクタとなっている．

A.1.5 PIC 16F648A のデータメモリマップ

PIC 16F648A のデータメモリは表 A.3 に示した様に 4 つのメモリバンクにマップされた 256 バイト (うち 16 バイトはどのバンクからもアクセス可能) の汎用レジスタと特殊レジスタから構成されている．

表 A.3: PIC 16F648A のデータメモリマップ

0000H	Indirect Addr.	0080H	Indirect Addr.	0100H	Indirect Addr.	0180H	Indirect Addr.
0001H	TMR0	0081H	OPTION	0101H	TMR0	0181H	OPTION
0002H	PCL	0082H	PCL	0102H	PCL	0182H	PCL
0003H	STATUS	0083H	STATUS	0103H	STATUS	0183H	STATUS
0004H	FSR	0084H	FSR	0104H	FSR	0184H	FSR
0005H	PORTA	0085H	TRISA	0105H		0185H	
0006H	PORTB	0086H	TRISB	0106H	PORTB	0186H	TRISB
0007H		0087H		0107H		0187H	
0008H		0088H		0108H		0188H	
0009H		0089H		0109H		0189H	
000aH	PCLATH	008aH	PCLATH	010aH	PCLATH	018aH	PCLATH
000bH	INTCON	008bH	INTCON	010bH	INTCON	018bH	INTCON
000cH	PIR1	008cH	PIE1	010cH		018cH	
000dH		008dH		010dH		018dH	
000eH	TMR1L	008eH	PCON	010eH		018eH	
000fH	TMR1H	008fH		010fH		018fH	
0010H	T1CON	0090H		0110H		0190H	
0011H	TMR2	0091H		0111H		0191H	
0012H	T2CON	0092H	PR2	0112H		0192H	
0013H		0093H		0113H		0193H	
0014H		0094H		0114H		0194H	
0015H	CCPR1L	0095H		0115H		0195H	
0016H	CCPR1H	0096H		0116H		0196H	
0017H	CCP1CON	0097H		0117H		0197H	
0018H	RCSTA	0098H	TXSTA	0118H		0198H	
0019H	TXREG	0099H	SPBRG	0119H		0199H	
001aH	RCREG	009aH	EEDATA	011aH		019aH	
001bH		009bH	EEADR	011bH		019bH	
001cH		009cH	EECON1	011cH		019cH	
001dH		009dH	EECON2	011dH		019dH	
001eH		009eH		011eH		019eH	
001fH	CMCON	009fH	VRCON	011fH		019fH	
0020H	General Purpose Register 80 Bytes	00a0H	General Purpose Register 80 Bytes	0120H	General Purpose Register 80 Bytes	01a0H	
006fH		00efH		016fH		01efH	
0070H	Shared Memory 16 Bytes	00f0H	Shared Memory 16 Bytes	0170H	Shared Memory 16 Bytes	01f0H	Shared Memory 16 Bytes
007fH		00ffH		017fH		01ffH	

A.2 PIC 16F648A のエミュレータの実装

A.2.1 PIC 16F648A のエミュレータの機能制限

現状では以下の機能には対応していない.

- ウォッチドッグタイマ
- スリープステート
- I/O ポートの状態変化をトリガとした割り込み

A.2.2 pic16f648_t 構造体

プロセッサインスタンスを保持するデータ構造として pic16f648_t 構造体を定義した.

```
typedef struct {
        union {
                uint8_t m[0x0200];
                struct {
                        uint8_t bank0[0x80];
                        uint8_t bank1[0x80];
                        uint8_t bank2[0x80];
                        uint8_t bank3[0x80];
                };
        } reg;
        uint16_t memory[0x1000];
        uint8_t eeprom[0x0100];
        uint16_t pc;
        uint16_t stack[8];
        uint32_t sp;
        uint16_t config;
        uint8_t accumulator;
        uint8_t WDT;
        uint8_t presc0;
        uint8_t presc1;
        uint8_t presc2;
        uint8_t pstsc2;
        uint32_t tick;
        uint64_t next;
        int extracycle;
```

```
        int loaded;
        int running;
        int intr;
        pthread_mutex_t lock;
        pthread_t thread;
        uint64_t m_freq;
        uint64_t m_sttm;
        uint64_t m_crtm;
        uint64_t m_step;
        uint64_t m_int0;
        uint64_t m_int1;
        uint64_t m_int2;
        FILE *logfd;
} pic16f648_t;
```

この構造体内には以下の挙げた通り，全てのプロセッサステートを保持するための情報が定義されている．

- union reg
  データメモリ領域

  - uint8_t m[0x0200]
    フラットアクセス用領域

  - struct{
    バンクアクセス用無名構造体

    * uint8_t bank0[0x0080]
      バンクアクセス用領域 (Bank0)

    * uint8_t bank1[0x0080]
      バンクアクセス用領域 (Bank1)

    * uint8_t bank2[0x0080]
      バンクアクセス用領域 (Bank2)

        * uint8_t bank3[0x0080]
          バンクアクセス用領域 (Bank3)

  - };

- uint16_t memory
  プログラムメモリ領域 (14 ビットワード ×4,096)

- uint8_t eeprom
  E^2PROM 領域 (8 ビットワード ×256)

- uint16_t pc
  プログラムカウンタ

- uint16_t stack[8]
  スタック (14 ビットワード ×8 の FILO)

- uint32_t sp
  スタックポインタ

- uint16_t config
  コンフィグレーションワード

- uint8_t accumulator
  ワーキングレジスタ W

- uint8_t WDT
  ウォッチドッグカウンタ

- uint8_t presc0
  タイマ0プリスケーラ

- uint8_t presc1
  タイマ1プリスケーラ

- uint8_t presc2
  タイマ2プリスケーラ

- uint8_t pstsc2
  タイマ2ポストスケーラ

また，これらのプロセッサ内部の状態の他，以下の実行制御用の変数も定義されている．

- uint32_t tick
  エミュレートされたプロセッサの1サイクルに相当するホストPCのクロック数を保持

- uint64_t next
  エミュレートされたプロセッサの次のサイクルが開始される時刻におけるホストPCのクロックカウンタ値を保持

- int extracycle
  現在エミュレートされている命令の実行サイクル数を保持

- int loaded
  該当するプロセッサインスタンスにバイナリコードがロードされていることを示すフラグ

- int running
  該当するプロセッサインスタンスのエミュレーションが実行中であることを示すフラグ

- int intr
  該当するプロセッサインスタンスのエミュレーションが割り込みコンテキストを実行中であることを示すフラグ

- pthread_mutex_t lock
  プロセッサインスタンスの内部データへのアクセスに対する排他制御を行うためのミューテックス

- pthread_t thread
  プロセッサインスタンスの実行スレッドを保持するスレッド構造体

- uint64_t m_freq
  エミュレーションを行っているホストPCのプロセッサの動作周波数を保持

- uint64_t m_sttm
  エミュレートされたプロセッサの実行開始時刻におけるホストPCのクロックカウンタ値を保持

- uint64_t m_crtm
  エミュレートされたプロセッサの現在時刻におけるホストPCのクロックカウンタ値を保持

- uint64_t m_step
  エミュレートされたプロセッサの実行開始時からの実行サイクル数

- uint64_t m_int0
  実行開始時からタイマ0割り込み発生回数

- `uint64_t m_int1`

  実行開始時からタイマ 1 割り込み発生回数

- `uint64_t m_int2`

  実行開始時からタイマ 2 割り込み発生回数

- `FILE *logfd`

  デバッグ用ログファイルのファイルポインタを保持

A.2.3 libpic16f648 の API

- 統計情報の表示

```
void
pic16f648ShowInfo(pic16f648_t *p)
```

  実行開始時からの実行ステップ数，割り込み発生回数，平均クロックレート等を表示.

- プロセッサインスタンスの確保

```
pic16f648_t *
pic16f648Alloc(uint32_t Hz)
```

  pic16f648 構造体を確保し構造体へのポインタを返すと共にエミュレーションスレッドを生成

- プロセッサインスタンスの解放

```
void
pic16f648Release(pic16f648_t *p)
```

  エミュレーションスレッドの終了と pic16f648 構造体の領域の解放

- プロセッサインスタンスへのバイナリコードの読み込み

```
int
pic16f648LoadHex(pic16f648_t *p, FILE *fd)
```

  Intel HEX フォーマットで記述されたバイナリコードをプログラムメモリ領域にロード

- エミュレーションスレッドの実行開始

```
int
pic16f648Start(pic16f648_t *p)
```

プロセッサインスタンスの実行を開始

A.2.4 libpic16f648 の主要関数

- ホスト PC のクロックレートを取得

```
static uint64_t
getCpuFreq(void)
```

sysctl(2) を利用してホスト PC のクロックレートを取得

- 次のサイクルまでのウェイト

```
static int
pic16f648Wait(pic16f648_t *p)
```

エミュレートされるプロセッサの次のサイクルまで待機

- レジスタアクセス

```
inline static uint8_t
pic16f684ReadReg(pic16f648_t *p, uint16_t a)

inline static void
pic16f648SetReg(pic16f648_t *p, uint16_t a, uint8_t v)
```

データメモリに対する直接アクセス/間接アクセスを利用した値の読み出し/書き込み

- メインループを構成する関数

```
static int
pic16f648ExecOp(pic16f648_t *p)
```

メモリからのオペコードのフェッチ・デコード・命令のディスパッチ

```
static int
pic16f648ProcInt(pic16f648_t *p)
```

割り込みの処理

```
static void *
pic16f648Emul(void *arg)
```

pic16f648ExecOp() / pic16f648ProcInt() / pic16f648Wait() の呼び出し

- 個々の命令に相当する処理群

```
static int
pic16f648OpAddwf(pic16f648_t *p, uint16_t op)

static int
pic16f648OpAndwf(pic16f648_t *p, uint16_t op)

static int
pic16f648OpClrf(pic16f648_t *p, uint16_t op)

static int
pic16f648OpClrw(pic16f648_t *p, uint16_t op)

static int
pic16f648OpComf(pic16f648_t *p, uint16_t op)

static int
pic16f648OpDecf(pic16f648_t *p, uint16_t op)

static int
pic16f648OpDecfsz(pic16f648_t *p, uint16_t op)

static int
pic16f648OpIncf(pic16f648_t *p, uint16_t op)

static int
pic16f648OpIncfsz(pic16f648_t *p, uint16_t op)

static int
pic16f648OpIorwf(pic16f648_t *p, uint16_t op)

static int
pic16f648OpMovf(pic16f648_t *p, uint16_t op)

static int
pic16f648OpMovwf(pic16f648_t *p, uint16_t op)

static int
pic16f648OpNop(pic16f648_t *p, uint16_t op)

static int
pic16f648OpRlf(pic16f648_t *p, uint16_t op)

static int
pic16f648OpRrf(pic16f648_t *p, uint16_t op)

static int
```

```
pic16f648OpSubwf(pic16f648_t *p, uint16_t op)

static int
pic16f648OpSwapf(pic16f648_t *p, uint16_t op)

static int
pic16f648OpXorwf(pic16f648_t *p, uint16_t op)

static int
pic16f648OpBcf(pic16f648_t *p, uint16_t op)

static int
pic16f648OpBsf(pic16f648_t *p, uint16_t op)

static int
pic16f648OpBtfsc(pic16f648_t *p, uint16_t op)

static int
pic16f648OpBtfss(pic16f648_t *p, uint16_t op)

static int
pic16f648OpAddlw(pic16f648_t *p, uint16_t op)

static int
pic16f648OpAndlw(pic16f648_t *p, uint16_t op)

static int
pic16f648OpCall(pic16f648_t *p, uint16_t op)

static int
pic16f648OpClrwdt(pic16f648_t *p, uint16_t op)

static int
pic16f648OpGoto(pic16f648_t *p, uint16_t op)

static int
pic16f648OpIorlw(pic16f648_t *p, uint16_t op)

static int
pic16f648OpMovlw(pic16f648_t *p, uint16_t op)

static int
pic16f648OpRetfie(pic16f648_t *p, uint16_t op)

static int
pic16f648OpRetlw(pic16f648_t *p, uint16_t op)

static int
pic16f648OpReturn(pic16f648_t *p, uint16_t op)

static int
pic16f648OpSleep(pic16f648_t *p, uint16_t op)

static int
pic16f648OpSublw(pic16f648_t *p, uint16_t op)

static int
pic16f648OpXorlw(pic16f648_t *p, uint16_t op)
```

A.2.5 libpic16f648 の内部構造

以下では libpic16f648 の主要な内部処理を説明する.

- pic16f648_t *pic16f648Alloc(uint32_t Hz)

プロセッサインスタンスの確保とエミュレーションスレッドの生成を行う.

```
pic16f648_t *
pic16f648Alloc(uint32_t Hz)
{
    uint64_t freq;
    pic16f648_t *p;

    if((p = (pic16f648_t *)malloc(sizeof(pic16f648_t)))
      == NULL) {
        perror("malloc()");
        return NULL;
    }
    bzero(p, sizeof(pic16f648_t));
    p->reg.bank0[STATUS] = p->reg.bank1[STATUS] =
      p->reg.bank2[STATUS] = p->reg.bank3[STATUS]
      = 0x18;
    p->reg.bank1[OPTREG] = p->reg.bank3[OPTREG]
      = 0xff;
    p->reg.m[TRISA] = 0xff;
    p->reg.m[TRISB] = 0xff;
    p->reg.m[PCON] = 0x08;
    p->reg.m[PR2] = 0xff;
    p->reg.m[TXSTA] = 0x02;
    p->logfd = stderr;
    DEBUGMSG(p);
    SETPC(p, RSTVEC);
    if((freq = getCpuFreq()) == 0) {
        WARNINGMSG(p, "cannot obtain processor clock");
        return NULL;
    }
    p->tick = freq / Hz * 4;
    DEBUGMSG(p, "tick = %d", p->tick);
    if(pthread_create(&p->thread, NULL, pic16f648Emul,
      p) != 0)
        return NULL;
    return p;
}
```

この関数では，プロセッサインスタンスの領域を確保した後，必要なデータメモリに初期設定を行い，プログラムカウンタにリセットベクタのアドレスをセットする．次に，ホストPCのクロックレートを取得し，引数として渡されたエミュレートされるプロセッサのクロックレートを基に，エミュレートされるプロセッサの1サイクルがホストPCの何クロックに相当するかを計算する．最後に実際のエミュレーションを行うスレッドの生成を行う．

- static void *pic16f648Emul(void *arg)

この関数がエミュレーションを実行するスレッドのメインの処理となる．

```
static void *
pic16f648Emul(void *arg)
{
    pic16f648_t *p = arg;
    uint64_t current;

#ifdef MEASUREMENT
    p->m_freq = getCpuFreq();
    rdtsc(p->m_sttm);
    p->m_int0 = 0;
    p->m_int2 = 0;
#endif /* MEASUREMENT */
    DEBUGMSG(p);
    while(p->running == 0)
        sched_yield();
#ifndef BESTEFFORT
    rdtsc(current);
    p->next = current + p->tick;
#endif
    for(;;) {
        if(pic16f648ExecOp(p) != 0) {
            ERROR(p, "fatal error\nabort\n");
     return NULL;
 }
 p->extracycle++;
 do {
     if(pic16f648ProcInt(p) != 0) {
 ERROR(p, "fatal error\nabort\n");
 return NULL;
     }
     if(pic16f648Wait(p) != 0) {
 WARNINGMSG(p, "deadline missed!\n");
     }
```

```
        p->extracycle--;
    } while(p->extracycle > 0);
    }
    return NULL;
}
```

この関数では，pic16f648Start() によってエミュレーションの開始を指示されるまで待機を行う．エミュレーションが開始されると pic16f648ExecOp() pic16f648ProcInt() pic16f648Wait() の各関数を周期的に実行する．

- static int pic16f648ExecOp(pic16f648_t *p)

この関数ではプログラムメモリからのオペコードのフェッチ，デコードを行う．

```
static int
pic16f648ExecOp(pic16f648_t *p)
{
    uint16_t op = p->memory[p->pc];

#if 0
    DEBUGMSG(p, "%04x: %04x", p->pc, op);
#endif
    switch((op & 0x3000) >> 12) {
    case 0x00:
        switch((op & 0x0c00) >> 10) {
        case 0x00:
            switch((op & 0x0380) >> 7) {
            case 0x00:
                switch(op & 0x007f) {
                case 0x00:
                case 0x20:
                case 0x40:
                case 0x60:
                    return pic16f648OpNop(p, op);
                case 0x08:
                    return pic16f648OpReturn(p, op);
                case 0x09:
                    return pic16f648OpRetfie(p, op);
                case 0x63:
                    return pic16f648OpSleep(p, op);
                case 0x64:
                    return pic16f648OpClrwdt(p, op);
                default:
                    return -1;
                }
                break;
            case 0x01:
```

```
            return pic16f648OpMovwf(p, op);
        case 0x02:
            return pic16f648OpClrw(p, op);
        case 0x03:
            return pic16f648OpClrf(p, op);
        case 0x04:
        case 0x05:
            return pic16f648OpSubwf(p, op);
        case 0x06:
        case 0x07:
            return pic16f648OpDecf(p, op);
        default:
            return -1;
        }
        break;
    case 0x01:
        switch((op & 0x0300) >> 8) {
        case 0x00:
            return pic16f648OpIorwf(p, op);
        case 0x01:
            return pic16f648OpAndwf(p, op);
        case 0x02:
            return pic16f648OpXorwf(p, op);
        case 0x03:
            return pic16f648OpAddwf(p, op);
        default:
            return -1;
        }
        break;
    case 0x02:
        switch((op & 0x0300) >> 8) {
        case 0x00:
            return pic16f648OpMovf(p, op);
        case 0x01:
            return pic16f648OpComf(p, op);
        case 0x02:
            return pic16f648OpIncf(p, op);
        case 0x03:
            return pic16f648OpDecfsz(p, op);
        default:
            return -1;
        }
        break;
    case 0x03:
        switch((op & 0x0300) >> 8) {
        case 0x00:
            return pic16f648OpRrf(p, op);
        case 0x01:
            return pic16f648OpRlf(p, op);
```

```
        case 0x02:
            return pic16f648OpSwapf(p, op);
        case 0x03:
            return pic16f648OpIncfsz(p, op);
        default:
            return -1;
        }
        break;
    default:
        return -1;
    }
    break;
case 0x01:
    switch((op & 0x0c00) >> 10) {
    case 0x00:
        return pic16f648OpBcf(p, op);
    case 0x01:
        return pic16f648OpBsf(p, op);
    case 0x02:
        return pic16f648OpBtfsc(p, op);
    case 0x03:
        return pic16f648OpBtfss(p, op);
    default:
        return -1;
    }
    break;
case 0x02:
    switch((op & 0x0800) >> 11) {
    case 0x00:
        return pic16f648OpCall(p, op);
    case 0x01:
        return pic16f648OpGoto(p, op);
    default:
        return -1;
    }
    break;
case 0x03:
    switch((op & 0x0c00) >> 10) {
    case 0x00:
        return pic16f648OpMovlw(p, op);
    case 0x01:
        return pic16f648OpRetlw(p, op);
    case 0x02:
        switch((op & 0x0300) >> 8) {
        case 0x00:
            return pic16f648OpIorlw(p, op);
        case 0x01:
            return pic16f648OpAndlw(p, op);
```

```
            case 0x02:
                return pic16f648OpXorlw(p, op);
            default:
                return -1;
            }
            break;
        case 0x03:
            switch((op & 0x0200) >> 9) {
            case 0x00:
                return pic16f648OpSublw(p, op);
            case 0x01:
                return pic16f648OpAddlw(p, op);
            default:
                return -1;
            }
            break;
        default:
            return -1;
        }
        break;
    default:
        return -1;
    }
    return -1;
}
```

- static int pic16f648ProcInt(pic16f648_t *p)

この関数では割り込み条件の判定，タイマカウンタの処理等を行う．

```
static int
pic16f648ProcInt(pic16f648_t *p)
{

/*
 *      Timer0
 */
    if((p->reg.m[OPTREG] & T0CS) == 1)
        goto timer1;
    p->presc0++;
    if((p->reg.m[OPTREG] & PSA) == 0) {
        if(p->presc0
          == (1 << ((p->reg.m[OPTREG] & 0x07) + 1))) {
            p->presc0 = 0;
            if((p->reg.bank0[TMR0] = p->reg.bank2[TMR0]++)
              == 0xff) {
                p->reg.bank0[INTCON]
                  = p->reg.bank1[INTCON]
```

```
                  = p->reg.bank2[INTCON]
                  = p->reg.bank3[INTCON] |= T0IF;
                if((p->reg.bank0[INTCON] & GIE)
                  && (p->reg.m[INTCON] & T0IE)) {
#ifdef MEASUREMENT
                    p->m_int0++;
#endif /* MEASUREMENT */
                    if(p->intr == 0) {
                        p->intr = 1;
                        PUSH(p, p->pc);
                        SETPC(p, INTVEC);
                        CLRGIE(p);
                    }
                }
            }
        }
    } else {        /* prescaler not in use */
        if((p->reg.bank0[TMR0] = p->reg.bank2[TMR0]++)
          == 0xff) {
            p->reg.bank0[INTCON] = p->reg.bank1[INTCON]
              = p->reg.bank2[INTCON]
              = p->reg.bank3[INTCON] |= T0IF;
            if((p->reg.m[INTCON] & GIE)
              && (p->reg.m[INTCON] & T0IE)) {
#ifdef MEASUREMENT
                p->m_int0++;
#endif /* MEASUREMENT */
                if(p->intr == 0) {
                    p->intr = 1;
                    PUSH(p, p->pc);
                    SETPC(p, INTVEC);
                    CLRGIE(p);
                }
            }
        }
    }
/*
 *      Timer1
 */
timer1:
    if((p->reg.m[T1CON] & TMR1ON) == 0)
        goto timer2;
    if((p->reg.m[T1CON] & TMR1CS) == 1)
        goto timer2;
    p->presc1++;
```

```
    if(p->presc1
      == (1 << ((p->reg.m[T1CON] & 0x30) >> 4))) {
        p->presc1 = 0;
        if(p->reg.m[TMR1L]++ == 0xff) {
            if(p->reg.m[TMR1H]++ == 0xff) {
                p->reg.m[PIR1] |= TMR1IF;
                if((p->reg.m[INTCON] & GIE)
                  && (p->reg.m[PIE1] & TMR1IE)) {
#ifdef MEASUREMENT
                    p->m_int1++;
#endif /* MEASUREMENT */
                    if(p->intr == 0) {
                        p->intr = 1;
                        PUSH(p, p->pc);
                        SETPC(p, INTVEC);
                        CLRGIE(p);
                    }
                }
            }
        }
    }
/*
 *      Timer2
 */
timer2:
    if((p->reg.m[T2CON] & TMR2ON) == 0)
        goto skip;
    p->presc2++;
    if(((p->reg.m[T2CON] & T2CKPS1) != 0)
      && (p->presc2 == 16)) {
    /* x16 prescaler */
        p->presc2 = 0;
        if(p->reg.m[TMR2]++ == p->reg.m[PR2]) {
            p->reg.m[TMR2] = 0;
            if(p->pstsc2++
              == ((p->reg.m[T2CON] & 0x78) >> 3)) {
                p->pstsc2 = 0;
                p->reg.m[PIR1] |= TMR2IF;
                if((p->reg.m[INTCON] & GIE)
                  && (p->reg.m[PIE1] & TMR2IE)) {
#ifdef MEASUREMENT
                    p->m_int2++;
#endif /* MEASUREMENT */
                    if(p->intr == 0) {
                        p->intr = 1;
```

```
                        PUSH(p, p->pc);
                        SETPC(p, INTVEC);
                        CLRGIE(p);
                    }
                }
            }
        }
    } else if(p->presc2
      == (1 << ((p->reg.m[T2CON] & T2CKPS0) * 2))
      - 1) {
    /* x1/x4 prescaler */
        p->presc2 = 0;
        if(p->reg.m[TMR2]++ == p->reg.m[PR2]) {
            p->reg.m[TMR2] = 0;
            if(p->pstsc2++
              == ((p->reg.m[T2CON] & 0x78)
              >> 3)) {
                p->pstsc2 = 0;
                p->reg.m[PIR1] |= TMR2IF;
                if((p->reg.m[INTCON] & GIE)
                  && (p->reg.m[PIE1] & TMR2IE)) {
#ifdef MEASUREMENT
                    p->m_int2++;
#endif /* MEASUREMENT */
                    if(p->intr == 0) {
                        p->intr = 1;
                        PUSH(p, p->pc);
                        SETPC(p, INTVEC);
                        CLRGIE(p);
                    }
                }
            }
        }
    }
skip:
    return 0;
}
```

- static int pic16f648ProcInt(pic16f648_t *p)

この関数ではエミュレートされるプロセッサの次のサイクルまでの待ちを行う.

```
static int
pic16f648Wait(pic16f648_t *p)
{
```

```
    uint64_t current;

    COUNTSTEP(p);
    rdtsc(current);
    while(current < p->next - p->tick / 2) {
        sched_yield();
        rdtsc(current);
    }
    p->next += p->tick;
    return 0;
}
```

ホスト PC の TSC カウンタの値を取得し，エミュレートされるプロセッサの次のサイクルが開始される TSC カウンタ値が格納されている tick と比較を行い，半サイクル以上の余裕がある場合には他のスレッドに実行権を譲る．

この時刻管理の方法は，gettimeofday() を利用した方法等に比べ，

- システムコールの呼び出しを行わないため処理が軽量
- 精度が高い

という利点がある反面，

- マルチプロセッサのシステムでは一定のプロセッサからカウンタを読み出すことができない可能性もある
- プロセッサの駆動周波数を動的に変化させるシステムでは必ずしも正しいカウンタ値を読み取ることができる訳ではない

という欠点がある．

A.2.6 libpic16f648 の利用

以下では複数のプロセッサインスタンスに同一のバイナリコードをロードし実行するサンプル (A.3 節参照) を例に libpic16f648 の利用について述べる．

```
for(i = 0; i < ncpus; i++)
    if((cpu[i] = pic16f648Alloc(atoi(argv[2])))
      == NULL) {
        fprintf(stderr,
```

```
            "failed to allocate instance\n");
        exit(1);
    }
```

プロセッサインスタンスの確保を行う．pic16f648Alloc() は引数としてエミュレートされるプロセッサのクロックレートを取る．インスタンスの確保に失敗した場合には NULL を，成功した場合にはプロセッサインスタンスの領域を指すポインタを返す．

```
for(i = 0; i < ncpus; i++) {
    if((fd = fopen(argv[1], "r")) == NULL) {
        perror("fopen()");
        exit(1);
    }
    if(pic16f648LoadHex(cpu[i], fd) < 0) {
        fprintf(stderr, "failed to load binary\n");
        exit(1);
    }
    fclose(fd);
}
```

ロードするバイナリコードを含むファイルをオープンし，各インスタンスのプログラムメモリにロードを行う．pic16f648LoadHex() はプロセッサインスタンスへのポインタとバイナリコードを含むファイルのファイルポインタを引数に取り，ロードが成功した場合には 0 を，それ以外には-1 を返す．

```
signal(SIGINFO, showinfo);
for(i = 0; i < ncpus; i++)
    if(pic16f648Start(cpu[i]) < 0) {
        fprintf(stderr, "failed to load binary\n");
        exit(1);
    }
for(;;);
```

ここまででプロセッサインスタンスは実行可能状態になっている．次に pic16f648Start() を呼び出すことによってエミュレーションが開始される．pic16f648Start() はプロセッサインスタンスへのポインタを引数に取り，エミュレーションの実行開始に成功した場合には 0 を，それ以外には-1 を返す．pic16f648Start() の実行が成功した後はエミュレーションは別スレッドで行われるため，メインスレッドは無限ループを実行している．

このサンプルでは，実行を開始する前にSIGINFOに対するシグナルハンドラとしてshowinfo()を設定している．

```
void
showinfo(int sig)
{
        int i;

        for(i = 0; i < ncpus; i++)
                pic16f648ShowInfo(cpu[i]);
}
```

showinfo()はlibpic16f648内のpic16f648ShowInfo()を各インスタンス毎に呼び出している．これにより，プロセスに対してINFOシグナルを送ることでインスタンス毎の統計情報を得ることが可能となる．INFOシグナルを送ることで得られる情報は以下のようなものになる．

```
> bin/memultest648 test.hex 20000000 1
pc: 0137        operation: 0bd4 step: 63827666
int0:        0  int1:        0  int2:        0
 29767152954 cycl      12.775602 sec
    4.996059 step/usec  0.200158 usec/step
    0.000000 int0/msec       inf msec/int0
    0.000000 int1/msec       inf msec/int1
    0.000000 int2/msec       inf msec/int2
   19.984237 MHz
```

A.2.7 libpic16f648の定義済みマクロ

- 実行制御

```
#define rdtsc(t) asm volatile("rdtsc" : "=A" (t))
```

インラインアセンブラを用いてIA-32アーキテクチャのプロセッサのTSCレジスタから値を読み出すマクロ

- プログラムメモリアドレス

```
#define RSTVEC  0x0000
#define INTVEC  0x0004
```

リセットベクタと割り込みベクタアドレス

- データメモリアドレス

```
#define TMR0    0x0001
#define PCL     0x0002
#define STATUS  0x0003
#define FSR     0x0004
#define PORTA   0x0005
#define PORTB   0x0006
#define PCLATH  0x000a
#define INTCON  0x000B
#define PIR1    0x000c
#define TMR1L   0x000e
#define TMR1H   0x000f
#define T1CON   0x0010
#define TMR2    0x0011
#define T2CON   0x0012
#define CCPR1L  0x0015
#define CCPR1H  0x0016
#define CCP1CON 0x0017
#define RCSTA   0x0018
#define TXREG   0x0019
#define RCREG   0x001a
#define CMCON   0x001f
#define OPTREG  0x0081
#define TRISA   0x0085
#define TRISB   0x0086
#define PIE1    0x008c
#define PCON    0x008e
#define PR2     0x0092
#define TXSTA   0x0098
#define SPBRG   0x0099
#define EEDATA  0x009a
#define EEADR   0x009b
#define EECON1  0x009c
#define EECON2  0x009d
#define VRCON   0x009f
```

データメモリ上の特殊レジスタ類のアドレス

- 特殊レジスタのビットマップ

```
#define IRP     0x80
#define GIE     0x80
#define PSA     0x08
#define T0CS    0x20
#define T0IE    0x20
#define T0IF    0x04
#define TMR1CS  0x02
#define TMR1IE  0x01
```

```
#define TMR1IF  0x01
#define TMR1ON  0x01
#define TMR2IE  0x02
#define TMR2IF  0x02
#define TMR2ON  0x04
#define T2CKPS1 0x02
#define T2CKPS0 0x01
#define RD      0x01
```

特定の用途が割り当てられた特殊レジスタのビットに対するマスク

- レジスタアクセス

```
#define REG(p, a)                                \
/* access register on current bank */            \
((p)->reg.m[(((p)->reg.m[STATUS] & 0x60) << 2) \
+ ((a) & 0x7f)])
```

現在選択されているバンクのデータメモリへのアクセス

```
#define REGD(p, a)
/* access register on all banks directly*/       \
((p)->reg.m[(a)])
```

あらゆるバンクのデータメモリへのアクセス

- プログラムカウンタ関連

```
#define PADDR(p)                                 \
((p)->pc)
```

プログラムカウンタへのアクセスを行うためのマクロ

```
#define SETPC(p, a) do {                         \
(p)->pc = (a);                                   \
(p)->reg.bank0[PCL] = (p)->reg.bank1[PCL] =      \
(p)->reg.bank2[PCL] = (p)->reg.bank3[PCL] =      \
(p)->pc & 0xff;                                  \
} while(0)
```

プログラムカウンタに値を設定するためのマクロ

- アキュムレータ (ワーキングレジスタ) 操作

```
#define SETACC(p, a)                            \
(p)->accumulator = (a) & 0x00ff
```

アキュムレータに値を設定

- プログラムカウンタ操作

```
#define INCPC(p) do {                           \
(p)->pc++;                                      \
(p)->reg.bank0[PCL] = (p)->reg.bank1[PCL] =     \
(p)->reg.bank2[PCL] = (p)->reg.bank3[PCL] =     \
(p)->pc & 0xff;                                 \
} while(0)
```

プログラムカウンタを 1 増加

- フラグ操作

```
#define GETCF(p) ((p)->reg.bank0[STATUS] & 0x01)
#define SETCF(p) ((p)->reg.bank0[STATUS]        \
= (p)->reg.bank1[STATUS]                        \
= (p)->reg.bank2[STATUS]                        \
= (p)->reg.bank3[STATUS] |= 0x01)
#define CLRCF(p) ((p)->reg.bank0[STATUS]        \
= (p)->reg.bank1[STATUS]                        \
= (p)->reg.bank2[STATUS]                        \
= (p)->reg.bank3[STATUS] &= 0xfe)
#define GETDC(p)                                \
(((p)->reg.bank0[STATUS] & 0x02) >> 1)
#define SETDC(p) ((p)->reg.bank0[STATUS]        \
= (p)->reg.bank1[STATUS]                        \
= (p)->reg.bank2[STATUS]                        \
= (p)->reg.bank3[STATUS] |= 0x02)
#define CLRDC(p) ((p)->reg.bank0[STATUS]        \
= (p)->reg.bank1[STATUS]                        \
= (p)->reg.bank2[STATUS]                        \
= (p)->reg.bank3[STATUS] &= 0xfd)
#define GETZF(p)                                \
(((p)->reg.bank0[STATUS] & 0x04) >> 2)
#define SETZF(p) ((p)->reg.bank0[STATUS]        \
= (p)->reg.bank1[STATUS]                        \
= (p)->reg.bank2[STATUS]                        \
= (p)->reg.bank3[STATUS] |= 0x04)
```

```
#define CLRZF(p) ((p)->reg.bank0[STATUS]       \
= (p)->reg.bank1[STATUS]                       \
= (p)->reg.bank2[STATUS]                       \
= (p)->reg.bank3[STATUS] &= 0xfb)
#define SETGIE(p) ((p)->reg.bank0[INTCON]      \
= (p)->reg.bank1[INTCON]                       \
= (p)->reg.bank2[INTCON]                       \
= (p)->reg.bank3[INTCON] |= GIE)
#define CLRGIE(p) ((p)->reg.bank0[INTCON]      \
= (p)->reg.bank1[INTCON]                       \
= (p)->reg.bank2[INTCON]                       \
= (p)->reg.bank3[INTCON] &= ~GIE)
#define SETTO(p) ((p)->reg.bank0[STATUS]       \
= (p)->reg.bank1[STATUS]                       \
= (p)->reg.bank2[STATUS]                       \
= (p)->reg.bank3[STATUS] |= 0x10)
#define SETPD(p) ((p)->reg.bank0[STATUS]       \
= (p)->reg.bank1[STATUS]                       \
= (p)->reg.bank2[STATUS]                       \
= (p)->reg.bank3[STATUS] |= 0x08)
#define CLRTO(p) ((p)->reg.bank0[STATUS]       \
= (p)->reg.bank1[STATUS]                       \
= (p)->reg.bank2[STATUS]                       \
= (p)->reg.bank3[STATUS] &= 0xef)
#define CLRPD(p) ((p)->reg.bank0[STATUS]       \
= (p)->reg.bank1[STATUS]                       \
= (p)->reg.bank2[STATUS]                       \
= (p)->reg.bank3[STATUS] &= 0xf7)
#define CLRWDTPS(p) ((p)->reg.bank1[OPTREG]    \
= (p)->reg.bank3[OPTREG] &= 0xf8)
```

演算命令によって変化するフラグのビットをセット/クリア

- 命令デコード支援

```
#define DST(p) (((p) & 0x0080) ? 'F' : 'W')
#define BIT(p) (((p) & 0x0380) >> 7)
```

オペコードからディスティネーションレジスタ指定子，ビット指定子を抽出

- スタック操作

```
#define PUSH(p, v) do {                         \
(p)->stack[(p)->sp] = v;                        \
(p)->sp = ((p)->sp + 1) & 0x07;                 \
} while(0)
#define POP(p, v) do {                          \
(p)->sp = ((p)->sp - 1) & 0x07;                 \
v = (p)->stack[(p)->sp];                        \
} while(0)
```

スタックとの間の push/pop 操作

- 統計処理支援

```
#ifdef MEASUREMENT
#define COUNTSTEP(p) ((p)->m_step++)
#else /* !MEASUREMENT */
#define COUNTSTEP(p) /* */
#endif /* MEASUREMENT */
```

統計処理が有効になっている場合に実行ステップ数をカウント

- メッセージ出力・ロギング関連

```
#define SHOWMSG(p, …) do {                      \
fprintf((p)->logfd,                             \
"[%s:%6d] ", __FILE__, __LINE__);               \
fprintf((p)->logfd, "%s(): \t", __func__);      \
fprintf((p)->logfd, __VA_ARGS__);               \
fprintf((p)->logfd, "\n");                      \
} while(0)

#ifdef VERBOSE
#define DEBUGMSG(p, …)                          \
SHOWMSG((p), "debug \t" __VA_ARGS__)
#define WARNINGMSG(p, …)                        \
SHOWMSG((p), "warning \t" __VA_ARGS__)
#else /* VERBOSE */
#define DEBUGMSG(p, …) /* __VA_ARGS__ */
#define WARNINGMSG(p, …) /* __VA_ARGS__ */
#endif /* !VERBOSE */

#define ERROR(p, …)                             \
SHOWMSG((p), "error " __VA_ARGS__)
```

コンパイル時のオプションに応じてメッセージを出力

A.3 サンプルプログラム

```
/*
 * Copyright (c) 2007 NAKATA, Junya
 * All rights reserved.
 *
 */

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <pic16f648.h>

int ncpus;
pic16f648_t **cpu;

void
showinfo(int sig)
{
        int i;

        for(i = 0; i < ncpus; i++)
                pic16f648ShowInfo(cpu[i]);
}

void
usage(char *argv0)
{

        fprintf(stderr,
            "usage: %s [hex file] [frequency] [number of instances]\n", argv0);
}

int
main(int argc, char *argv[])
{
        int i;
        FILE *fd;

        if(argc != 4) {
                usage(argv[0]);
                exit(1);
        }
        ncpus = atoi(argv[3]);
        if((cpu = (pic16f648_t **)malloc(sizeof(pic16f648_t *) * ncpus))
            == NULL) {
                fprintf(stderr, "failed to allocate memory\n");
                exit(1);
        }
        for(i = 0; i < ncpus; i++)
                if((cpu[i] = pic16f648Alloc(atoi(argv[2]))) == NULL) {
                        fprintf(stderr,
                            "failed to allocate instance\n");
```

```
                exit(1);
        }
    for(i = 0; i < ncpus; i++) {
        if((fd = fopen(argv[1], "r")) == NULL) {
                perror("fopen()");
                exit(1);
        }
        if(pic16f648LoadHex(cpu[i], fd) < 0) {
                fprintf(stderr, "failed to load binary\n");
                exit(1);
        }
        fclose(fd);
    }
    signal(SIGINFO, showinfo);
    for(i = 0; i < ncpus; i++)
        if(pic16f648Start(cpu[i]) < 0) {
                fprintf(stderr, "failed to load binary\n");
                exit(1);
        }
    for(;;);
    return 0;
}
```