修士論文　　　　2001年度(平成13年度)

柔軟で高信頼な分散コンポーネント間接続基盤ソフトウエアの構築

慶應義塾大学　政策・メディア研究科
氏名：岩井 将行

修士論文要旨　　2001年度(平成13年度)

柔軟で高信頼な分散コンポーネント間接続基盤ソフトウエアの構築

論文要旨

今日，コンポーネント技術によって多様な分散システムが構築されている反面，それらのシステムは，高度な知識をもったシステム開発者やシステム設定者を必要としている．本論文では，ホームネットワークなどのシステム開発者の介在が困難な場面で，コンポーネント技術の利便性を生かし，一般的なユーザが即興的に分散システムの構築を可能とする分散ミドルウエア**Dragon**について述べる．

Dragonは，各コンポーネントを単純化できる複数仲介者型間接通信モデルを採用し，経路判断機能の外部化，コンテンツ適応機能の構造化，信頼性のある1対多通信機能などの特徴を持ちながら，柔軟にコンポーネント間を即興的に接続できる機構を提供している．

本論文では，基本コンポーネントサブシステム，コマンダサブシステム，ディレクトリサブシステムからなるDragonの設計を行い，Java2およびJiniを利用したDragonの実装について述べる．

Dragonは，家庭内ネットワークだけでなく工場などの時間制約を伴ったイベント配送を要求される場でも即興的なシステム構築が可能となる高信頼性をもったコンポーネント間接続ミドルウェアとして利用できる．本論文では，Dragonの優先度を区別したマルチプルユニキャストの適切な動作，仲介者の直列的な配送時間，近接コンポーネントの発見等を評価した．それによりDragonが適切な配送処理動作と妥当な動作時間で稼動が可能であることを示した．

キーワード：

1 分散コンポーネント　2 即興的システム構成　3 リアルタイムイベント配送

4 1対多通信　5 Jini

慶應義塾大学　政策・メディア研究科

岩井 将行

Abstract of Master's Thesis

A Middleware for Conposing Distributed Components with Reliability and Adaptabity

Academic Year 2001

Summary

Today, distributed systems are developed by putting various software components distributed together through network. However, such systems require developers who are familiar with the software architecture and its components.

In this research, we present a design and implementation of a new middleware, **Dragon**, which is suitable for collaboration between distributed components within networks where there are no developers, such as a home network. Dragon enables construction and modification of event-driven distributed component systems easily and extemporarily. Dragon's implementation is based on the Multiple Mediators Indirect Communication Model, which can make each component simple. One of the several features is one to many communications with reliability using multiple-unicasting. In addition, Dragon incorporates soft real-time event delivery to enhance the predictability of the behavior within a constructed system, and the composition of distributed components for event management.

We will show that Dragon has adequate performance for using in a Homenetwork area. In some systems such as sensor networks in a factory, many events occur simultaneously with a timing constraint. In these systems, Dragon can also support users to make reliable distibuted systems.

Keyworkds:

1 Distributed Components 2 Improvised System Conpotition 3 Real-Time Event Delivery
4 One to Many Communication 5 Jini

Keio University Graduate School of Media and Governance
Masayuki Iwai

目 次

図目次

表 目 次

第 1 章

序論

1.1 本研究の背景と問題意識

近年，多くのシステム開発の場面でコンポーネント技術が利用されている．コンポーネントとは，一連の動作をひとくくりにしたソフトウェア部品と言える．あるシステム向けに作成された一つの構成ソフトウェアを汎用的にコンポーネントとして構築しまとめておけば，再度他のシステムに流用可能である．このため，ショッピングモール開発などの目的を同じくするシステムの開発コストを軽減できる利点があった．さらに，これらのコンポーネントをシステム開発者間で取引することで，複雑で高度なシステム構築を容易に開発できる．このような再利用性と市場性を備えたコンポーネントは，情報のミラーリング，分散スケジューラ，組織間意思決定プロセス支援，監視アプリケーション，システムインテグレーションなどの様々なアプリケーションドメインのシステム開発で利用注目されつつある．なかでも，Javaで実装されるEJB[25]は，Webアプリケーションのサーバサイドドメインで近年急速に利用されている．しかしながら既存のコンポーネント技術は，分散するシステム開発を考えた場合，システム再構成に関する柔軟性が欠落しており，再利用性による開発コスト軽減の利点が生かされていない．さらに既存のコンポーネント技術だけでは，ユビキタス環境での構成要素の多様化に対して不適応である．

1.1.1 分散システム再構成に関する柔軟性の欠落

これらのコンポーネント技術は，現在のところシステム開発者を介して構築，運用されている場でしか利用できない．これは既存の分散システムが，システム開発者がシステム利用者の用途を想定してコンポーネントの詳細をデザインし，組み合わせるコンポーネントを決定し，それらのコンポーネント間を接続する詳細なプロトコルを決定して構築されるからである．すなわち，一般的なシステム利用者によるシステムの再構成は困難である．既存の分散システム開発では，図1.1のように，分散システムの仕様変更ごとにシステム開発者に変更と開発を要請するため，利用者にとっては経済的にもコストが生じる．しかし，高度なシステム開発の知識のない一般的な利用者

でもコンポーネント間を接続して容易に分散システムを構築できる機構は，提供されていない．

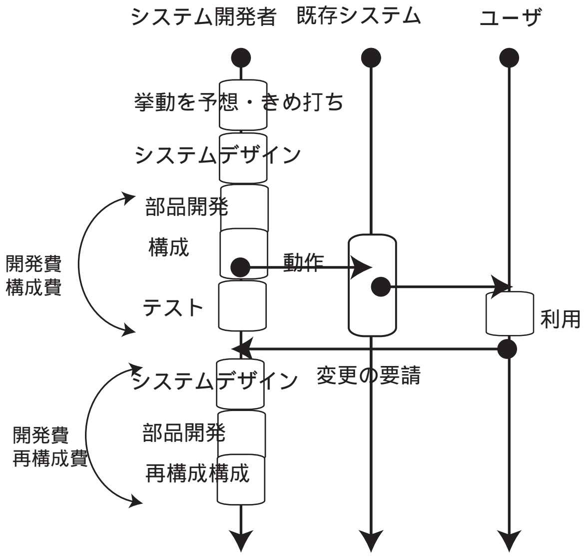


図 1.1: 既存分散システム開発の問題点

1.1.2　構成要素の多様化に対する不適応

一方，図1.2にあるような，センサや機器がユーザの周辺に偏在する環境は，近年のハードウェアの小型化と価格面でのコストダウンにより，一般的な家庭やオフィスにおいても可能になると言える[18, 8]．そのようなネットワーク環境では，独立して動作するソフトウエア，及び情報家電機器が多様に偏在し，頻繁に追加あるいは変更される．このため画一的に分散システムを構築することは難しく，即興的に個々の構成要素を発見し，異なる目的のコンポーネント間を柔軟に接続させる技術が必要である．しかし既存の機構は，システム開発者の介在を前提としているため，ユーザ周辺で即興的に分散システムが構築されるユビキタス環境には適さない．

1.2　本研究の目的と概要

本研究では，分散システム開発に要するコストを避けるため，図1.3に示すようにユーザが分散システムを利用しながらの構成変更を可能とする．図1.3の利用者主体の分散システム構成手法を即興的分散システム構成と定義し，分散コンポーネント接続ミドルウエア**Dragon**[10, 30, 31]で実装した．

一般的な利用者が，自由に選択したコンポーネントを即興的に接続し，1つの分散システムを構築するには，ミドルウエア側で近接するコンポーネントの発見，通信内容

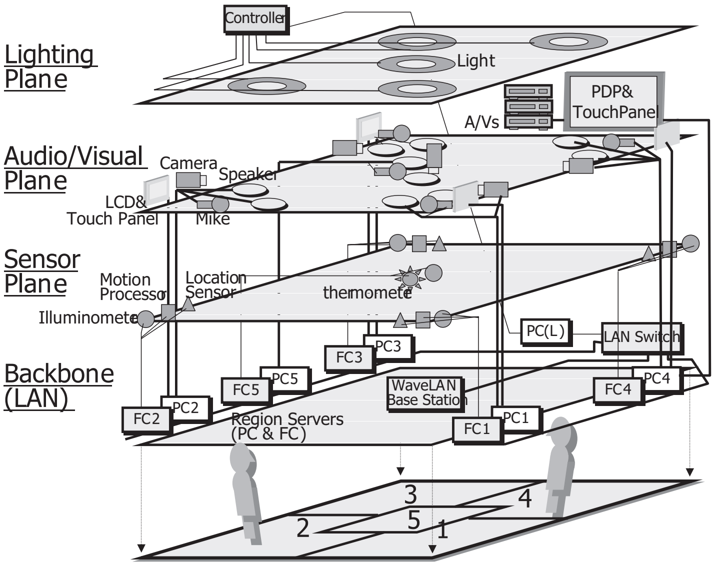


図 1.2: ユビキタス環境 :Smart Space Laboratory

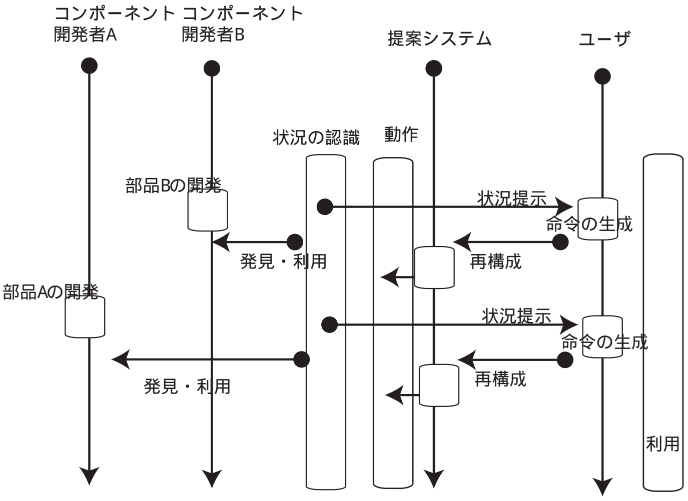


図 1.3: 提案する即興的な分散システム構築モデル

の相互適応，1対多通信における配送のオーバフローの制御などの信頼性を考慮しなければならない．またTini[4]，DUONUS[28]などのユビキタス環境における低処理能力の組み込み機器などはPCと異なり，資源が制限され，複雑なソフトウエアの挙動がコスト面で大きな問題になる．そのため各コンポーネントが単純な機能で実装されており，CPU等に負荷を与えないことが重要となる．

本研究により，複雑な前提知識がない利用者にも，分散システムの即興的な構築が可能になる．これによってコンポーネント技術の利用分野はさらに広がり，家庭内ネットワークなどをはじめとする分野での応用が期待できる．

本研究において考察していく分散コンポーネントとは，情報家電機器上などでも動作可能な目的別のソフトウェアをさす．本研究を実際に家庭内の分野で利用したシナリオとして，図1.4のスマートエンタランスシステムの例を記述する．ユーザが帰宅時を自動的に認証させ，ドアのロックを解除し，室内の情報家電を手順にしたがって制御させる例である．ユーザ認識コンポーネント，ドアロックコンポーネント，エアコンコンポーネントなどをユーザが自由に接続して，この様な例の分散システムが即興的に構築可能になる．

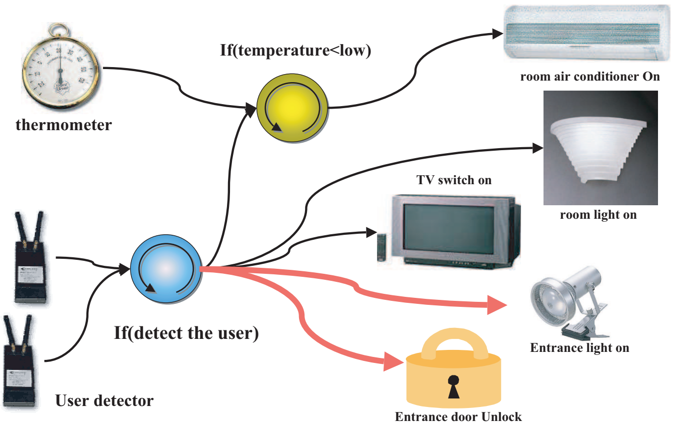


図1.4: ユーザがDragonを用いて構築できるスマートエンタランスシステムの例

1.3　本研究の構成

本論文では，まず2章で即興的分散システム構築の要件を述べる．さらに個々のコンポーネントの単純性とコンポーネント間の情報の流れに着目し，分散コンポーネント通信の分類を行う．比較した結果に基づき個々のコンポーネントが単純で，即興的な分散システムの構築が可能な複数仲介者型間接通信モデルに付いて述べる．3章では，複数仲介者型間接通信モデルを基にした，分散コンポーネント接続基盤機構Dragonの特徴と設計を述べる．

4章でDragonの実装を示し，即興的な分散システム構築を視覚的に支援するGUIツールと，ソフトリアルタイムイベント配送メカニズムの詳細を述べる．

5章では，Dragonの基本性能と1対多の配送メカニズムが正しく動作している測定し，家庭内ネットワークでの利用及び即興的分散システム構築の要件を満たしているかを評価する．6章でDragonと関連研究との比較を行いDragonの優位性を述べ，最後に7章で本論文をまとめる．

第 2 章

即興的分散コンポーネント接続機構の分類

即興的な分散システム構築に適したコンポーネント通信モデルを考察するため，本章ではコンポーネント間のイベントの流れと各コンポーネントの単純さを指標とする分類を行う．単純さを指標にする理由は，先に述べたように，各コンポーネントが単純に実装されることは組み込みシステムや既存のコンポーネントを再利用する際に重要となるからである．

2.1 用語定義

ネットワークに介在しているコンポーネント同士が協調動作を行うには，コンポーネント間でネットワークを経由した情報のやり取りが必要になる．

本節では即興的分散コンポーネント接続を議論する上で重要となる用語を定義する．まずコンポーネント間でやり取りされるメッセージ，データ，オブジェクトおよび命令をイベントとして定義する．これらイベントが，あるコンポーネントから別のコンポーネントへ伝わることを, イベント配送あるいはイベント通知と定義する．図2.1に，二つのコンポーネント間でイベント通知が行われている概念を示す．

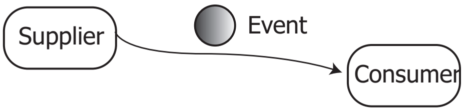


図 2.1: イベント生産者とイベント消費者の関係

イベントを発生させる送り手コンポーネントをイベント生産者，あるいは単に生産者と定義する．また，他のコンポーネントから発生したイベントを受け取り，その内容に基づいた処理を行う受け手コンポーネントをイベント消費者，あるいは単に消費者と定義する．生産者と消費者の両方を併せ持つコンポーネントの場合，受け取るイベントと生成するイベントに関連がないコンポーネントは役割としてイベント生産者とイベント消費者の二つの複合体として扱える．イベントを受け取り，その内容に基づき，新たなイベントを発生させるコンポーネントを特別にイベント仲介者，または仲介者と定義する．コンポーネントを生産者，消費者，仲介者の3つに分類して，分散システムのトポロジを考察し，即興的な分散システム構築に対する適正をそれぞれ考察する．

2.2 即興的分散システム構築の要件

システム管理者や開発者が介さない家庭内ネットワークなどの小規模ネットワークにおいて，コンポーネント間の即興的な接続を実現するために必要となる機能を示す．

コンポーネント障害検知機能

分散システムにおいては，ネットワークの状態や利用者からの外的な要因で障害が起こる可能性があり，通信相手先コンポーネントに対する信頼性は常にない．そのためにコンポーネントの障害をすばやく動的に検知する機能が必要になる．

システム障害可能性検知機能

即興的に構築される分散システムにおいては，システム開発者によるテストやチューニングが行われない．そのため現在の設定で，イベントのオーバフローなどによって障害に陥る可能性がある部分をあらかじめ検知する機構をミドルウエアが備えている必要がある．

近接コンポーネント認識機能

システム開発者がすべてのコンポーネントを認識している場合と異なり，利用者が即興的に分散システム構築を行う場合は，ミドルウエア側で近接のコンポーネントを発見する機構を提供していなければならない．特に物理的に利用者に近接する場所，あるいはマルチキャスト通信可能ネットワーク上にある頻繁に変更するコンポーネントを発見することで，ミドルウエアによる利用者に対する現状の提示や自動的なコンポーネント間接続が可能になる．近接するコンポーネントを認識する手法として，ディレクトリサービスをシステム内に設置する手法とコンポーネントが相互に自分自身の存在を通知する手法がある．

経路判断機能

経路判断機能とは，初期段階でコンポーネント間の接続が決定されていない分散システムにおいて，近接のコンポーネントの中から自らが送信あるいは受信すべきコンポーネントを選び出す機能である．経路判断機能は送り手が受け手を選んでいる場合は受け手が行い，受け手が送り手を選んでいる場合は送り手が行う．各コンポーネントで経路判断機能を持たず，この機構を外部化するコマンダを規定することも可能である．

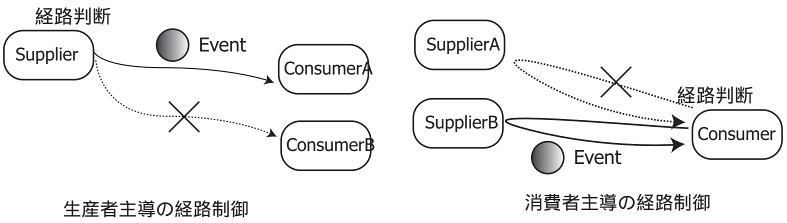


図 2.2: 経路判断機能に関する二つのモデル

コンテンツ適応機能

即興的にコンポーネントを接続する場合，生産者が生成するイベントとイベント消費者の認識可能なイベントの内容が異なる可能性がある．その差異を吸収する機能を

コンテンツ適応機能と呼ぶ．適応性があるコンポーネントとは，即興的に発見された通知先コンポーネントに合わせ，イベントのコンテンツを変換させるコンポーネントを示す．例えば，コンテンツ適応機能とは，*button.pushed()* という内部状態を示したクラスから *light.on()* という内部状態のクラスへ変換処理などをさす．(図2.3参照)

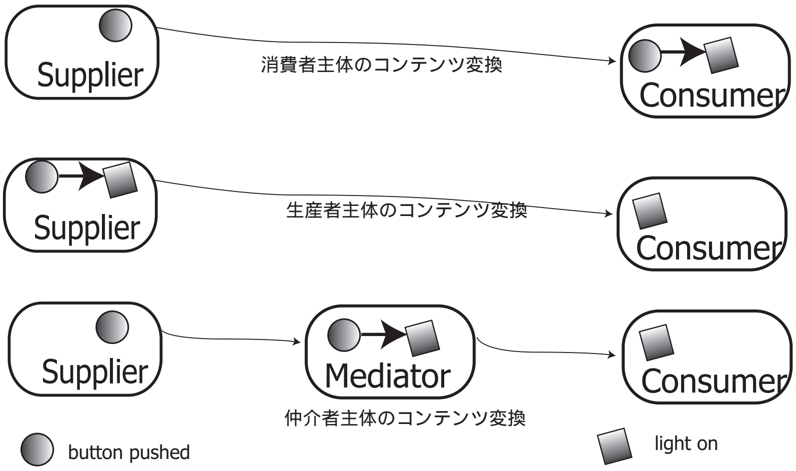


図 2.3: コンテンツ変換機能

1 対多通信機能

重要なイベントを生成する生産者が1つしかない場合には，1つの生産者から多数のイベント消費者へイベントが配送するアプリケーションを構築しなければならない．このような，1つのコンポーネントが複数のコンポーネントへイベントを一度に配送しなければならない状態 (1対多通信) は多く存在し，システムがオーバフローすることを防ぎながら，1対多通信が確実にイベント消費者に届く信頼性を保証する必要がある．

2.3 即興的分散コンポーネント通信のトポロジ分類

既存の分散ミドルウエアが提供する分散コンポーネント通信は，イベント生産者とイベント消費者の間で直接送信されているものと，間接的に送信されているものに分類できる．さらに通信の主導者がどの種のコンポーネントであるかを基準に図2.4で示すように分類し，即興的な分散システムに構成に対する適性を比較する．本章で述べきれない，同期性も含めたさらに詳しい既存の分散ミドルウエアの考察は，文献[32]で述べている．

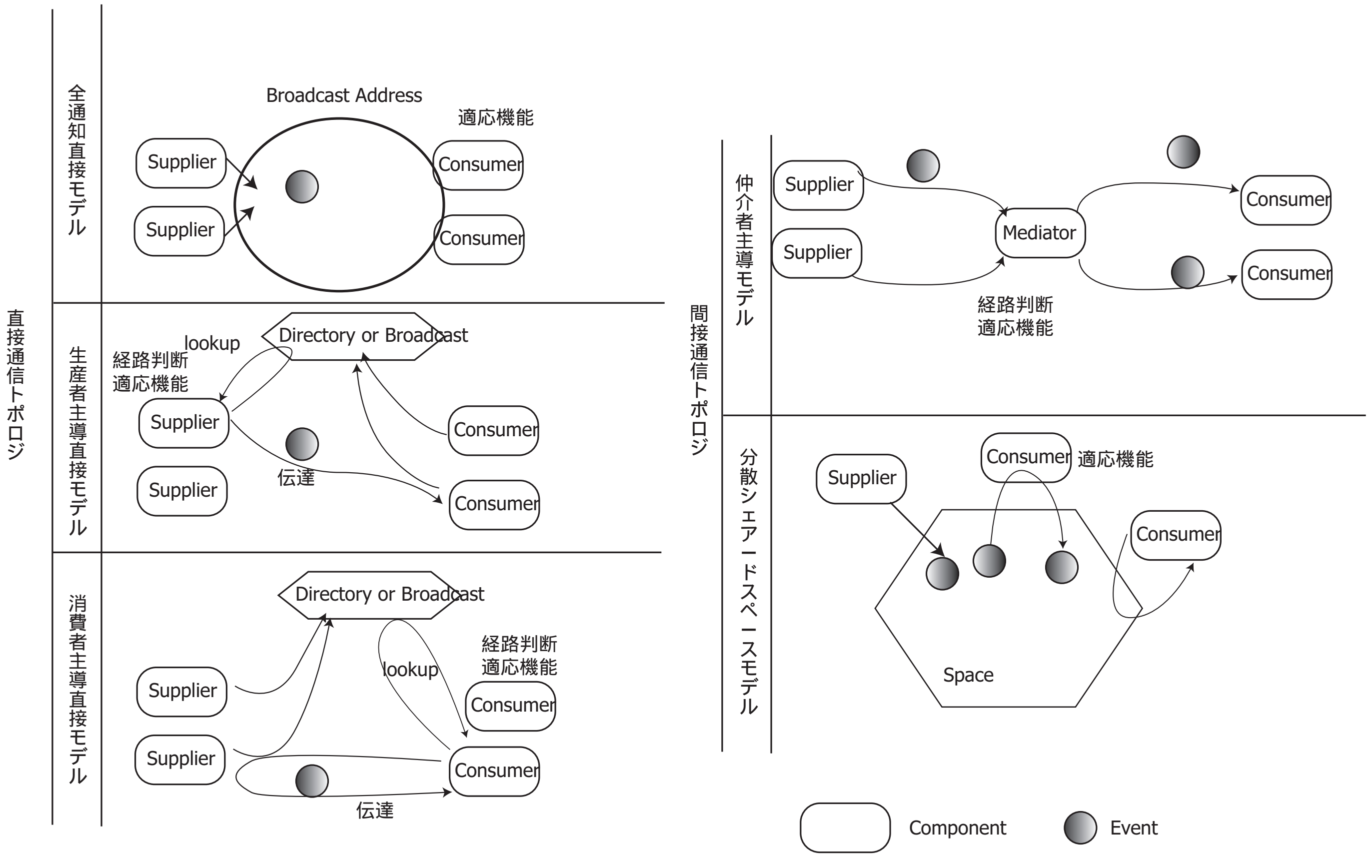


図 2.4: 分散コンポーネント通信トポロジの分類

2.3.1　直接通信トポロジ

直接通信トポロジとは，イベント生産者と消費者が直接通信を行うトポロジであり，全通知直接通信モデル，生産者主導直接モデル，消費者主導直接モデルの3つに分類できる．

全通知直接通信モデル イベント生産者が，ブロードキャストによりイベントを配送させる手法である．全通知直接通信モデルは，1対多通信としてブロードキャストを用いているため，生産者側は消費者側を考慮する必要がなく，経路判断をおこなうコンポーネントは存在しない．そのため，単一の目的コンポーネントにより構成される分散システムで信頼性が必要ない場合は，イベントの伝達が一度でよく効率がよい．しかし，消費者はすべての生産者からのイベントに対して自らに関係があるかどうかを判断する必要がある．つまり消費者側に高度な適応性がなければならない．多様な目的のコンポーネントが介在した場合，生産者の量的なスケーラビリティや，イベントの発生する周期が短い場合に消費者の処理が追いつけない．

生産者主導直接モデル 各コンポーネントが互いの存在をあらかじめ認知していない場合に，ディレクトリサービスやブロードキャストアドレスを利用して互いの存在を認知し，生産者側からダイレクトに通信を行うモデルである．JEcho[29]などがこれに当たるが，生産者はダイレクトに通信するため，必ず相手に適応したイベント通知を行うことが前提になる．経路判断とコンテンツ適応の双方を行う生産者に負荷がかかるモデルである．1対多通信としてマルチキャストグループ通信，マルチプルユニキャストの両者が可能なモデルである．

消費者主導直接モデル イベント消費者が能動的に，ディレクトリサービスやブロードキャストアドレスを利用し生産者を認識する．その中から生産者を決定し配送要求を行う．Jini[22]，VNA[14, 15]などがこのモデルに当てはまる．消費者自らイベントを能動的に獲得するため1対多通信機能は必要はない．

2.3.2　間接通信トポロジ

間接通信トポロジとは，イベント生産者と消費者が直接通信を行わず，間接的に通信を行うトポロジであり，仲介者主導通信モデル，分散シェアードスペースモデルに分類できる．

仲介者主導通信モデル システム内で唯一特別なイベント仲介者を定義し，各コンポーネントはそのイベント仲介者を認知しておく．イベント仲介者は，すべてのコンポーネントを認知している．仲介者はイベントが送られると，すぐさまコンポーネントを選択し通知する．CORBA Event Service[17]がこれにあたる．生産者と消費者は互いを認知している必要はなく，イベントの変換と引渡しは，イベント仲介者が代理する．イベント生産者とイベント消費者の構築が単純になり容易になる．しかしイベント仲介者はすべてのイベント送信者とイベント受信者の組み

合わせを想定しなければならない．この組み合わせが増えた場合，複雑なメカニズムをもつ仲介者をプログラマが実装しなければならない問題がある．また本モデルは，1 対多通信を行い多数の消費者と通信する必要もある．

分散シェアードスペース モデル ネットワーク内部にシェアードスペースを定義し，そのメモリに処理内容を通知したり，処理結果をシェアードスペースに書き込む消費者主導のモデルである．スペース側は，オブジェクト置き場としての役割を果たし，受動的に振る舞う[1]．Linda[7]，JavaSpaces[21]，TSpaces[27] などがこの例である．単純な計算処理を複数に分割し，分散処理をさせる場合などに適する．各コンポーネントは互いに認知する必要はないが共通のオブジェクトのフォーマットが必要になる．またシェアードスペースモデルでは，イベントの経路は存在せず，唯一の概念的なスペースが存在する．消費者主導なので 1 対多通信を行う必要はない．

2.4　考察と採用モデル

前章の分類を踏まえた考察結果を表 2.1 に示した．表中の複雑とは経路判断あるいは適応機能をコンポーネント内部でおこなっていることを示す．単純とはその逆であり，コンポーネントが独自のロジックのみに集中している状態を表している．

表 2.1: コンポーネント間通信機構の評価

通信モデル名	経路判断	適応機能	1 対多	送り手	仲介者	受け手	ディレクトリ
全通知型直接モデル	なし	受け手	要	単純	なし	複雑	無し
送り手主導型直接モデル	送り手	送り手	要	複雑	なし	単純	有り
受け手主導型直接モデル	受け手	受け手	不要	単純	なし	複雑	有り
仲介者主導型間接モデル	仲介者	仲介者	要	単純	複雑	単純	無し
シェアードスペースモデル	なし	受け手	不要	単純	単純	複雑	無し
複数仲介者型間接通信モデル	外部化	構造化仲介者群	要	単純	単純	単純	有り

家庭内ネットワークなどでは処理能力が期待できない組み込み機器が多く存在し，必ずしも生産者コンポーネントや消費者コンポーネントに複雑な処理を期待できない．そのために送り手と受け手の負荷が共に最小になるモデルは，仲介者主導型間接モデルである．しかし，仲介者主導型間接モデルでは，すべてのコンポーネントを認識し，適応機能も持った仲介者が必要となるため，即興的な分散システム構築の形態として適当ではない．

そこで本研究では，図 2.5 に示す複数仲介者型間接通信モデルを採用する．複数仲介者型間接通信モデルとは，先に示した仲介者主導の間接的な通信モデルの一部であるが，仲介者の負荷を分散化し構造化するモデルである．本モデルは，仲介者の構造化により，各仲介者は最小限の機能を考慮するだけでよい．

---

[1] JavaSpaces などはスペース側からオブジェクトの到着を示す notify() メソッドを持つ，このため能動的に通知することも可能であるが，通知登録を消費者が事前に行うため厳密に仲介者主導であるとはいえない．

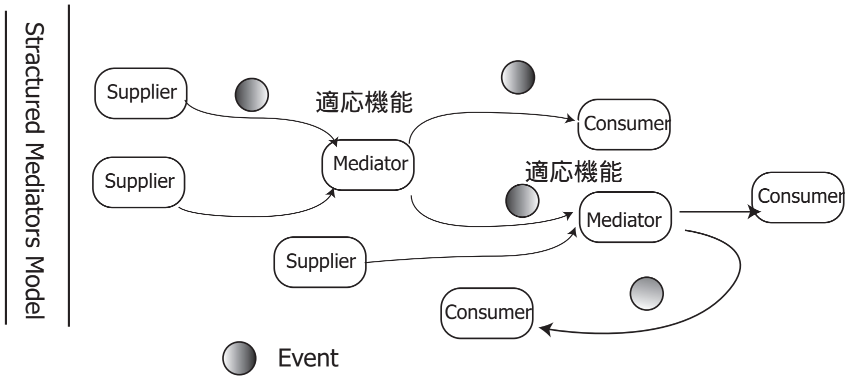


図 2.5: 複数仲介者型間接通信モデル

2.5　本章のまとめ

本章では，即興的な分散システム構築に適したコンポーネント通信モデルを考察し，複数仲介者型間接通信モデルがヘテロジニアスな環境で有用であることを述べ採用した．次章では，採用した複数仲介者型間接通信モデルを基にし，分散コンポーネント間接続基盤機構 Dragon の特徴と設計を述べる．

第 3 章

分散コンポーネント間接続基盤機構 **Dragon** の設計

本章では，前章で述べた複数仲介者型間接通信モデルを応用した分散コンポーネント間接続基盤機構Dragonの設計を述べる．Dragonは，仲介者の構造化，経路判断機能の外部化，信頼性のある1対多通信機構を特徴とする．

仲介者の構造化とは，複数の仲介者を連結して動作可能にするために単一のポートでイベントを受付け，処理を行わせる．この部分をBasicComponentサブシステムとした．また経路判断機能はコマンダサブシステムとして外部化し，4種類の接続形態を持たせた．これにより，Dragonでは多様な形態で通信が行える．さらに信頼性のある1対多通信機構を実現するために優先度をつけたユニキャスティング配送では，緊急の場合優先度のないものをスキップさせ，イベントを最優先で配送させ信頼性のある通信を保障する．

3.1 Dragonの設計方針

本節では2.2節の即興的分散システム構築を可能にする機能要件に従い，Dragonの設計方針を示していく．Dragonは，コンポーネント間の即興的な接続を可能にすることを目的としており，構成は図3.4に示すようにディレクトリサブシステム，BasicComponentサブシステム，コマンダサブシステムからなる．BasicComponentサブシステムは，分散するコンポーネントが共通で持つ機能をモジュール化したものである．コマンダサブシステムは，ディレクトリサブシステムから分散されるコンポーネントの状況を把握し，各コンポーネント間の接続する．

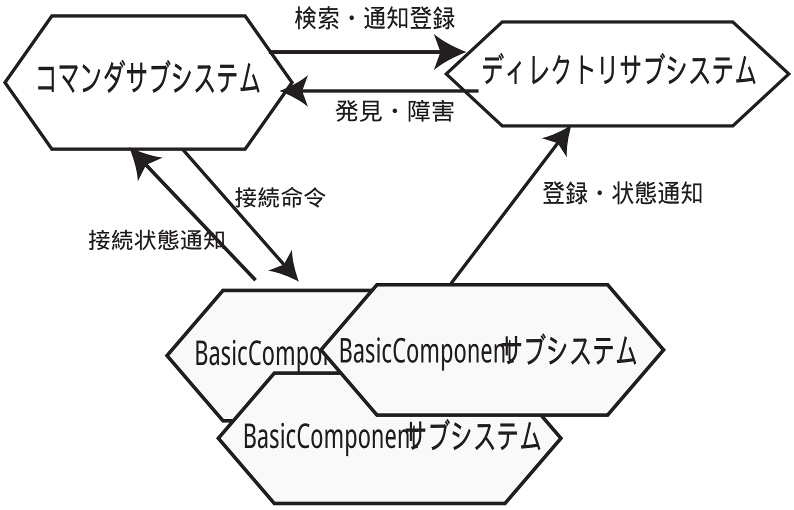


図 3.1: Dragonの3つのサブシステム構成

3.1.1 伝達されるイベントの柔軟性

イベントのコンテンツを静的に定義するとプログラムの構造や将来登場するコンポーネントを利用できない制限が生じてしまう．そのためDragonにおいて，伝達されるイベントを静的に定義しない．多様なコンテンツのイベントを配送可能にし，その経路選択は外部のコマンダサブシステムに一任する．例えばUPnP[26]で生成されたコマンドをJiniのコマンドに変換するといった接続の柔軟性が可能になる．しかし，イベントの内容が自由度が高ければ，逆に統一的なシステムを作り出すときにコンポーネント間で相互互換の問題が生じる．Dragonではそのため典型的なイベントのタイプを準備し，必要な場合はプログラマにそれを利用するように促す．

3.1.2 仲介者の構造化とユニポートコンポーネント

本モデルは，仲介者主導型間接モデルの1つであるが，仲介者を構造化し，複数の分散仲介者を連結している．このため各仲介者はより単純な機能を備えており，組み込み機器などの資源制約が多い環境に適しいる．例えば，仲介者Aは，各部屋の温度計からの入力の平均値を出力するコンポーネントとして実装し，仲介者Bはデータがある閾値より大きければイベントを発生させ，エアコンを起動する役割を持たせておく．平均値から最低値を変更させたい場合，仲介者Aの数値を生成する要件を変化させれば良い．この様に，仲介者を部品化することで，仲介者の再利用が可能になる．

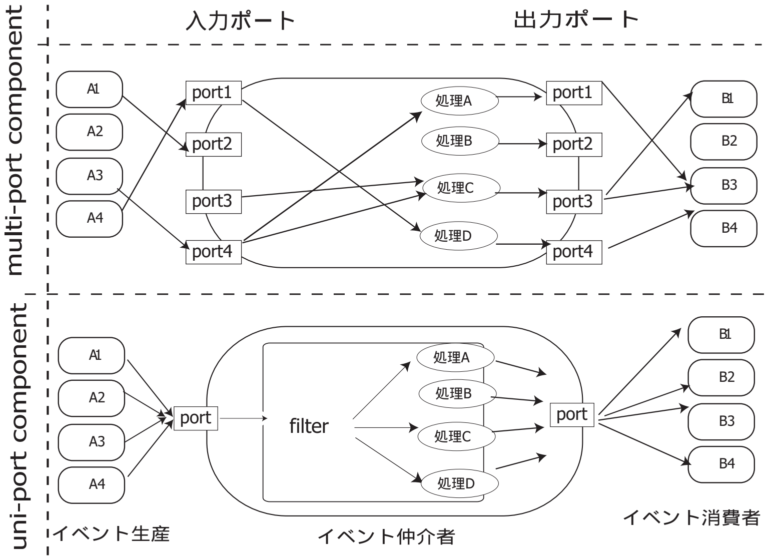


図 3.2: マルチポートのコンポーネントとユニポートのコンポーネント

この様な直列的な構成をとる仲介者は，入力ポートと出力ポートを持っている．上流のコンポーネントから到着するイベントは，入力ポートから入る．また下流のコンポーネントへは，出力ポートからイベント配送される．このポートは，コンポーネント内部のモジュールのひとつである．図3.2に示すように，上流や下流のコンポーネントごとにポートを定義できるマルチポートのコンポーネントとポートを一つしか持たないコンポーネントが存在できる．マルチポートは，出力ポートと処理内容ごとに入力のポートを規定できるが，内部の処理，入力ポート，出力ポートの依存関係が複雑になってしまう問題がある．Dragonでは，より単純なモデルであり，異なる目的のコンポーネント間を接続可能なユニポートのイベント仲介者とする．Dragonはユニポートコンポーネントモデルを基にBasicComponentサブシステムを実装する．

3.1.3 コンポーネント障害検知機能

コンポーネントがネットワーク的な障害や，物理的な故障により突如に使用不可能になった時は，他のコンポーネントやコマンダサブシステムにその障害が認知されなければならない．コンポーネント管理部は，常にディレクトリサブシステムと通信を行いディレクトリサブシステムで分散コンポーネントが，現時点で利用可能であるかを認識する．

3.1.4 近接コンポーネント認識機能

コンポーネント管理部は，コンポーネントが起動するとマルチキャスト通信可能な領域内で自動的にディレクトリサブシステムを発見し，登録を行う．これにより，自動でコンポーネントは利用可能であることが通知される．

3.1.5 経路判断機能の外部化

経路判断機能をコマンダサブシステムとして一元的に外部化することにより，各コンポーネントの負荷を減らすだけでなく，分散システム全体の挙動を認識することが容易になる．例えば，GUIにより現在のコンポーネント間の構成を即座に変更させたい場合，GUIは外部化したコマンダサブシステムとのインタラクションのみを行えばよい．本モデルでは，経路判断はコマンダサブシステムに一元化するが，経路情報は各コンポーネントが独自に保持する．このため，コマンダサブシステムが停止した場合でも，システム全体が停止することはない．また，コマンダサブシステムの外部化により，経路判断機能を自動化させたい場合は，すべてのコンポーネントと通信を行わなくてもよくなり，容易に経路判断を実装できる利点がある．

経路情報は各コンポーネントに分散してさせて保持する．コマンダサブシステムは起動されるたびに各コンポーネントから通知先のリストを取得する．

3.1.6 信頼性のある1対多通信機能

その場合，複数のイベント消費者に対して高周期でイベントが配送され，イベント生産者の配送処理が追いつかず，イベント通知の信頼性が失われる問題がある．

1対多のイベント配送手法としては，ブロードキャスト配送，マルチキャストグループ配送，マルチプルユニキャスト配送の3種類がある．ブロードキャストは，すべてのコンポーネントに一度で通知する方法であり，消費者に対するイベントの伝達を確認できないため信頼性はない．マルチキャストグループ配送は，図3.3が示すように，消費者のグループごとにマルチキャストのチャネルを定義する．マルチプルユニキャスト配送とは，個々のコンポーネントに逐次的に配送を行うモデルである．

マルチキャストグループ配送は，消費者コンポーネントごとに適切なグループを構築し一度にイベントを配送し処理することが可能となる．しかしすべてのコンポーネントにイベントが伝達が行われる有無には信頼性のないがない．そのためマルチキャ

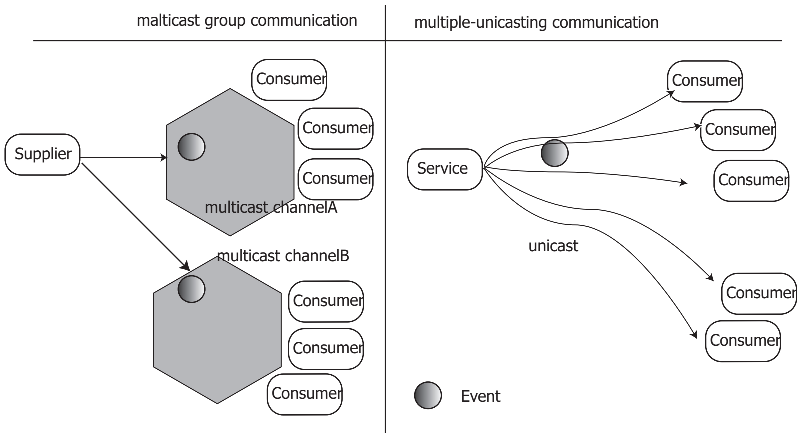


図 3.3: 1対多通信に関する二つの方法

スト通信においては，RMTP[11]やSRM[6]等のリライアブルマルチキャストを用いる必要がある．しかし現在複数のルータを介した現実的なヘテロジニアスネットワークにおいて利用できるリライアブルマルチキャストの仕組みはない．またユニキャストとマルチキャストを併用する1対多通信としてXCAST[1]がある．XCASTはIPv6拡張ヘッダとして送信時にルーティングヘッダを付け加える．XCASTが利用可能なネットワーク環境が現時点で普及していないため本研究では利用を考えない．マルチプルユニキャストは，既存のネットワーク環境で利用でき，信頼性が必要となる場面にも適しているが，個々に配送を行うため伝達時間に揺らぎが生じる．

これを回避するために本研究では優先度をつけたマルチプルユニキャスティングを新たに提案する[9]．優先度をつけたユニキャスティング配送では，優先度のないものを後回しにしてでも必ずイベントを最優先で配送させ，配送先が多数であっても一部のユニキャスティング配送は伝達時間を保障するものである．優先的に配送されるものをソフトリアルタイム配送 と呼び，ノンリアルタイム配送と呼ぶ優先度の低い配送を区別する．ソフトリアルタイム配送は，ノンリアルタイム配送の周期が非常に短い場合や配送先が多い場合でも優先され，イベントが伝達される．

3.1.7 システム障害可能性検知機能

分散システム内においてコンポーネント間のイベント通信のオーバフローを検知し，コマンダサブシステムに収集する．各コンポーネントからは，通信先のおよび配送の平均時間と最悪値等が同時に伝えられ，利用者が分散システムの状況を把握することが可能とする．

3.2 ディレクトリサブシステム

ディレクトリサブシステムは，分散されるコンポーネントの統一的な管理や検索を目的としている．本研究において既存のシステムを利用する．以下にDragonで必要とするディレクトリシステムの機能用件を述べる．

コンポーネントの検索方法が多様性 名前以外にも検索項目があること．特にインタフェースでの検索が可能であること．

マルティキャストによるディレクトリシステムの発見可能性 起動時にディレクトリを知る必要を避けるためマルティキャストをサポートしていることが望ましい．

障害検出機能 予期しないコンポーネントの障害に対応できること．障害があればすぐさまディレクトリに反映が行われることが必要である．

3.3 BasicComponent サブサブシステム

BasicComponentサブシステムは，図3.4に示すイベント入力部，イベントフィルタ部，イベント出力部，アクション部，センサ部，およびコンポーネント管理部の6つのモジュールで構成される．各モジュールは，プログラマが，開発するコンポーネントのロジックに集中できるように機能が単純化されている．

3.3.1 イベント入力部（Event Input Module）

イベント入力部は，他コンポーネントのイベント出力部からのイベント配送を受け取るモジュールである．Dragonが単純なユニポートコンポーネントモデルを採用することは3.1節で述べたが，イベント入力部では処理内容な同一だが優先度が異なる入力ポートを2種類用意する．

ソフトリアルタイム配送のイベントを受けたイベント入力部では，そのイベントに対する処理が優先的に実行されることを保証する．そのために内部で起動するフィルタの数を制限し，仮にノンリアルタイムのイベントがリアルタイムのイベントの処理中に到着した場合に待機またはスキップする．図3.5のCheck Point1では，受け付けてから配送が終了していないイベント総数を検査し，規定値を超えていなければ到着したイベントをイベントフィルタ部に渡す．Check Point2では現時点でフィルタリングを行っているスレッド数を検査し，ノンリアルタイム配送のイベントは，規定値以下になるまで待機するが，ソフトリアルタイム配送のイベントは優先される．

ノンリアルタイム配送は，リアルタイム配送に支障がない場合ベストエフォートで行われるが，すべてのコンポーネントがソフトリアルタイム配送を行えるわけではない．ソフトリアルタイム配送を保証するために以下の登録要求を決定する．新たなリアルタイム配送テーブルの登録要求は，以下の項目を満たす時に実行される．

$SRTOT$には，通知先関連情報$O_i (i = 1, 2, 3...n)$が入っている．各O_iには，通知先のスタブである$target$，デッドライン（ミリ秒），イベントを受け取ってからターゲ

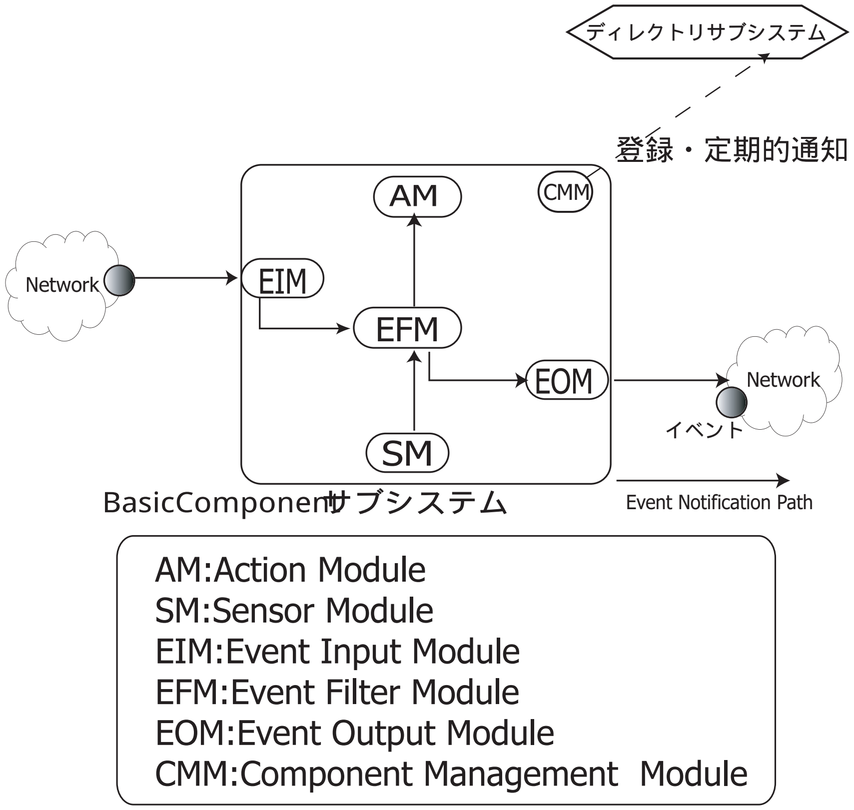


図 3.4: BasicComponent サブシステムの構成

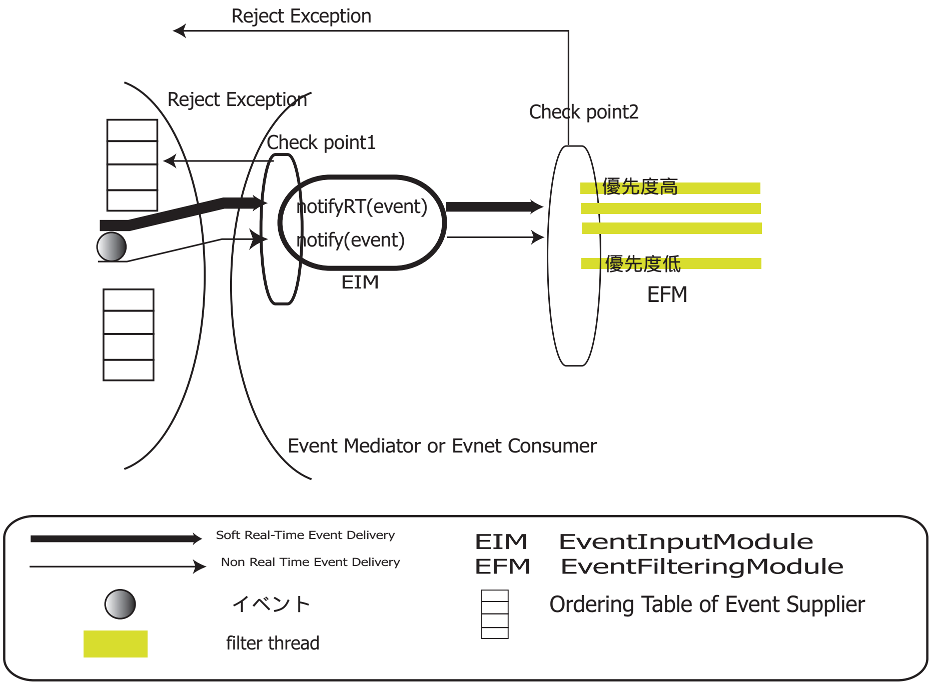


図 3.5: イベント入力部でのフィルタを行うスレッド数の制御

ットに配送を完了するまでの最悪配送時間 $worstcasetime$ が記入する．つまり O_i は，$O_i = \{target, deadline, worstcasetime\}$ となる．
ここで新たな $O_{new} = \{target, deadline, \phi\}$ なる配送先から要求があった場合

$O_{new}.deadline < O_i.deadline$ なるすべてのiに対して
$O_i.worstcasetime + durtime(O_{new}) < O_i.deadline$ を満たすならば
登録を受け付け i 番目として $SRTOT$ に挿入する．

上述の規定によりデッドラインを保証でき，既存の登録を無効化するようなソフトリアルタイム配送が追加されることはない．配送する場合は，$SRTOT$ のi番目から順に行う．

3.3.2　イベントフィルタ部 (Event Filter Module)

イベントフィルタ部は，イベント入力部より到着したイベントに対して処理を行う．このためイベントフィルタ部が複数同時に起動できるようにスレッドである必要がある．イベントをイベント出力部に通過させるかさせないかの判断およびイベントのアクション部に通過させる判断を行う．

フィルタの処理内容は以下に示すもの想定する．

- イベントを変更せず，そのまま通過させる
- イベントを通過拒否し破棄する
- イベントを変更して通過させる
- イベントを待機状態にさせ，格納しておく

3.3.3 イベント出力部（Event Output Module）

イベント出力部は，他コンポーネントのイベント入力部へイベント通知を行うモジュールである．Dragonが単純なユニポートコンポーネントモデルを採用することは3.1節で述べたが，イベント出力部でもイベント入力部と同様に部処理内容な同一だが優先度が異なる入力ポートを2種類用意する．

イベント出力部でのイベントの配送時には，図3.6の様にイベント配送先の優先度区別を行っている．ノンリアルタイム配送は，ソフトリアルタイム配送を行っていないことをCheck Point3及びCheck Point4の検査し，パスした場合に下流のコンポーネントへ配送を行う．仮にノンリアルタイムの配送がすべて終わらないうちに次のイベントが到着した場合は，ノンリアルタイム配送の登録をされている通知先はスキップされ，ソフトリアルタイムのイベント配送が優先される．一連のイベント処理の流れの詳細は，付録Cに示す．

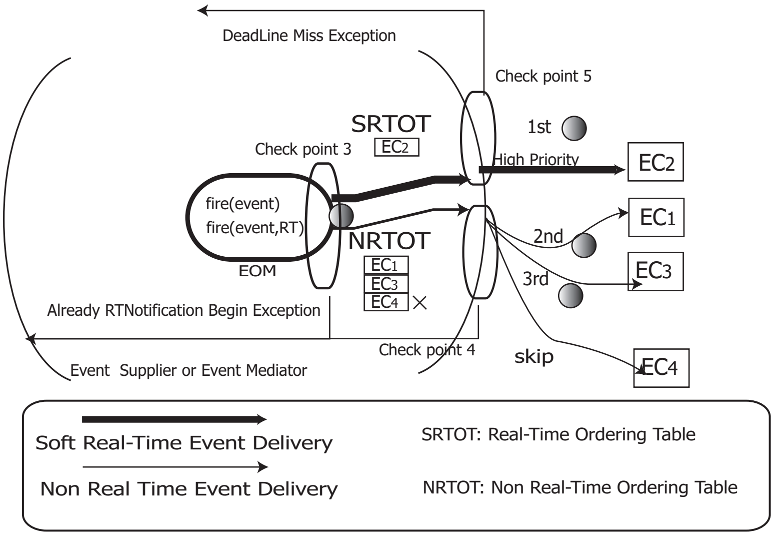


図 3.6: イベント出力部でのイベント配送時の優先度区別

3.3.4 アクション部 (Action Module)

アクション部は，イベントフィルタ部を通過したイベントを受け取ったあとの挙動を決定する部分である．Dragon側では詳細は規定せず，コンポーネント開発者に自由度を持たせる．情報家電機器上の組み込みシステムなどの場合は，情報家電機器を動作される動作を開発者が実装する．

3.3.5 センサ部 (Sensor Module)

イベントを生成する大本の部分である．Dragon側では詳細は規定せず，コンポーネント開発者に自由度を持たせる．開発者は，本モジュールで生成されるイベントが通知される先を考慮することなくイベントの発生ロジックに集中して，開発を行える．

3.3.6 コンポーネント管理部 (Component Management Module)

コンポーネント管理部は，図3.7に示す手順によりディレクトリサブシステムに対して，コンポーネントの登録を行い，自動的に一定間隔ごとに通知を行うモジュールである．複数種類のディレクトリサブシステムを用いる場合はこのモジュールでディレクトリサブシステムの差異を吸収する．

3.4 コマンダサブシステム (SettingCommander サブシステム)

Dragonでは，経路判断者からのイベント配送命令を図3.8の4種類設け，柔軟な経路判断機構を実現する．それぞれの説明を以下に示す．

Push型通信駆動 コマンダサブシステムからAへ命令を発行した瞬間のA内部に保持してあるイベントをBに対してすぐさま配送する．コマンダサブシステムはBの参照をA渡す．センサや情報家電の情報をすぐさま通知させたい場合に有効である．

Pull型通信駆動 コマンダサブシステムからBへ命令を発行するとBはAに対してpullの要求を出す．Bがイベント通知に同期を必要とする場合に有効である．

Call Back型通信要求 コマンダサブシステムはAへ「Bにイベント配送を行う」ことを登録させる．イベントが発生するごとにBに必ず通知される．コマンダサブシステムは登録をキャンセルすることもできる．イベントが継続して発生する場合に，有効であり，Dragonにおける主なイベント配送は，このCall Back通信要求の非同期通信である．

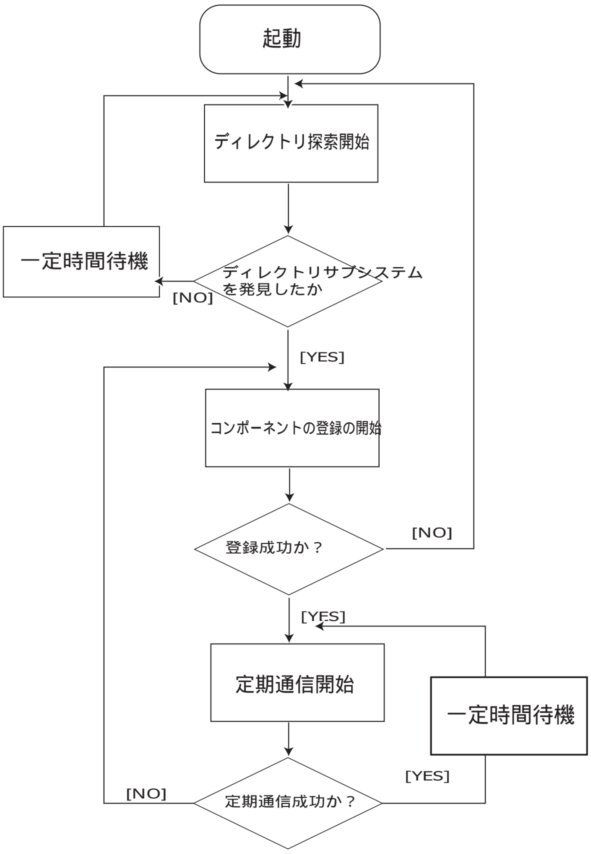


図 3.7: コンポーネント管理部の動作手順

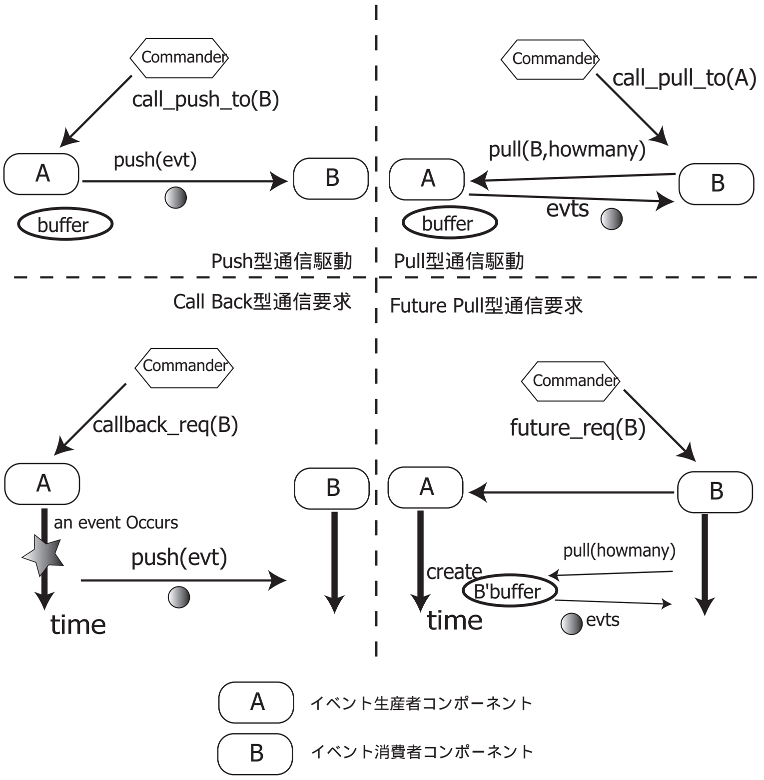


図 3.8: 経路判断者からの命令によるコンポーネント間接続の形態

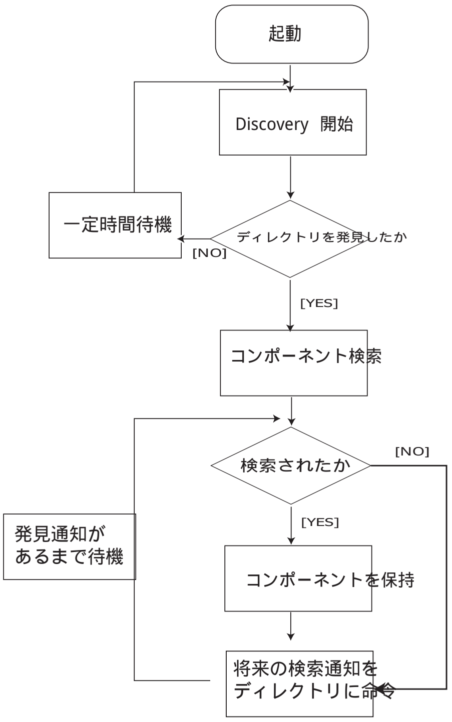


図 3.9: コンポーネント検索部の動作手順

Future Pull型通信要求 Bに対してAに将来的にPullする準備がある事を通知させる．データベース呼び出しなどAに対して特別なイベント配送を準備する場合に必要である．

3.4.1 コンポーネント検索部 (Component Searcher Module)

コンポーネント検索部は，コマンダサブシステムの中で以下の機能を図3.9に示す様にディレクトリサービスを発見し，Dragonにかかわるすべてのコンポーネントを検索する．次に将来コンポーネントが起動され，そのディレクトリサブシステムに登録される可能性がある場合は，検索通知命令をディレクトリサブシステムに発行し通知を待つ．

3.5 本章のまとめ

本章では，分散コンポーネント間接続基盤機構Dragonの特徴である仲介者の構造化，経路判断機能の外部化，信頼性のある1対多通信機構について述べた．仲介者の構造化とは，複数の仲介者が連結して動作可能にするために単一のポートでイベントを受付け，処理行わせる．また経路判断機能はコマンダサブシステムとして外部化し，4種類の接続形態を持たせた．これにより，Dragonでは多様な形態で通信が行える．さらに

信頼性のある1対多通信機構を実現するために優先度をつけたユニキャスティング配送では，緊急の場合優先度のないものをスキップさせ，イベントを最優先で配送させ信頼性のある通信を保障する．次章では，この設計の基づいたDragonの実装を述べる．

第 4 章

分散コンポーネント間接続基盤機構 Dragon の実装

本章では，前章の設計に基づいた分散コンポーネント間接続基盤機構 Dragon の実装について述べる．Dragon は，Java2 および Jini を用いて実装され，おもに BasicComponent クラスと SettingCommander クラスからなる．それぞれのフィールド，メソッド，主な実装箇所について記述し，付録として，各コンポーネントのサンプルコードと実行手順を説明する．

4.1 Dragonの実装概要

Dragonは，動作環境としてJava2の環境を想定する．Javaを用いた理由は，非常に近年多くの組み込みシステムやサーバサイドの環境で，Javaが利用されており基盤性が高く，RMIという強力で自由度の高いオブジェクト移送技術を持ち合わせているからである．Dragonは，さらにディレクトリサブシステムとしてJini LookupService[24]を用い，コンポーネント障害検知機能と近接コンポーネント認識機能の一部にJini1.1のLease機能[23]を利用する．そのためDragonは，図4.1に示すようにJava/JiniベースでTCP/IPを利用可能な情報家電機器やPC上のコンポーネントを想定する．

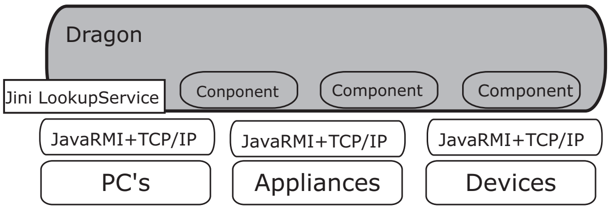


図 4.1: Dragonが想定する環境

Dragonの大まかなクラス構成を，図4.2に示す．EventSupplierクラス，EventMediatorクラス，EventConsumerクラスはイベント生産者コンポーネント，イベント仲介者コンポーネント，イベント消費者コンポーネントに相当する．EventSupplierクラスは，SensorModuleクラスおよび，EventoutputModuleクラスから成る．EventMediatorクラスは，EventInputModuleクラス，EventFilterModuleクラス，およびEventoutputModuleクラスから成る．EventConsumerクラスは，EventInputModuleクラス，ActionModuleクラスから成る．されに，コマンダサブシステムを実装したクラスとしてSettingCommanderクラスを提供する．

4.2 Jini LookupServiceの利用

近接コンポーネント認識機能とコンポーネント障害検知機能のために分散処理技術Jini1.1のLease機能を利用する．各コンポーネントは，マルチキャスト通信可能な領域内であるか，あるいはLookupServiceアドレスをあらかじめ認識している必要がある．信頼性を高めるためにLookupServiceを2個以上同時に起動することも可能である．

各コンポーネントは，定期的にLease更新を行い，自分自身が起動状態にあることをJini LookupServiceに通知する．Lease更新がない場合は，コンポーネント障害と見なし，SettingCommanderクラスに即座に通知を行う．これにより，利用者も即座に分散システム内のコンポーネントの挙動を検知できる．

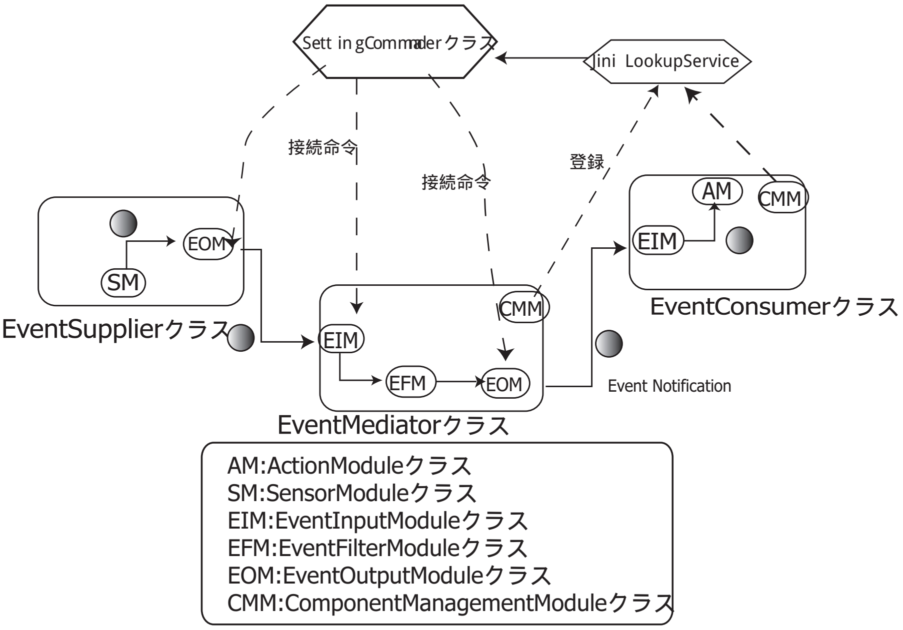


図 4.2: 3つのコンポーネントと内部モジュールの関係

4.3 DragonEventObject クラス群

Dragon のコンポーネント間で伝達される DragonEventObject としては*java.io.Serializable*を実装したあらゆるオブジェクトおよび*java.rmi.MarshalledObject*を実装したリモート呼び出し可能なオブジェクトの 2 種類が配送可能である．しかし，コンテンツ適応機能を備え，各コンポーネント間でイベントの内容を操作するためにの代表的なイベントは，Dragon でプログラマのために用意し，表 4.1のように，用途別のインタフェースで定義する．これらのイベントは *drg.event.datatype.DragonEventObject* を継承して実装される．Dragon をもちいたシステム内部で共通の認識を保持するためにこのイベントクラス群をさらに継承して独自のイベントを作成することが望ましい．実際に配送されるイベントは，さらにシーケンス番号やソースオブジェクトを付加した *RemoteEvent(java.lang.Object source, long eventID, long seqNum, java.rmi.MarshalledObject handback)* でラップされて配送される．

4.4 BasicComponent クラス

BasicComponent クラスは，自動生成されたコンポーネント固有の値である **ComponentID** を持つ．この ID は，ホストアドレス + サービス名 + 起動された時間 + ランダム定数より生成され，分散環境上でコンポーネントを一意に識別するために利用する．

表 4.1: Dragon で提供されるイベントクラス群

クラス名	型名	フィールド名
drg.event.datatype.DToken	boolean	token
drg.event.datatype.DPointer	String	url
drg.event.datatype.DApplet	Applet	applet
drg.event.datatype.DApplianceControl	String int boolean	applianceName intvolume power
drg.event.datatype.DSourceSensor	String double	sensorName volume

$$ComponentID = java.net.InetAddress.getLocalHost() + System.currentTimeMillis() + (newDouble(Math.random() * 1000)).intValue();$$

BasicComponent クラスでは，コンポーネントに様々な付加情報を添付することが可能である．表4.2にあるメソッドを用いて，位置情報，製品情報，コメント，制御アプレットを付加できる．もし面倒な場合は *initDefalut()* メソッドを用いて初期化してもよい．

表 4.2: BasicComponent クラスの主なメソッド

public void initAddress(String street, String organization, String organizationalUnit, String locality, String stateOrProvince, String postalCode, String country) ;
public void initLocation(String floor, String room, String building);
public void initComment(String comment);
public void initProductInfo(String manufacturer,String vendor,String version);
public void initApplet(Applet applet);
public void initDefalut();

以下順に BasicComponent クラスの構成要素 EventInputModule クラス，EventFilterModule クラス，EventOutputModule クラス，ActionModule クラス，SensorModule クラス，ComponentManagementModule クラスの説明を行う．

4.4.1 EventInputModule クラス

到着するイベントをハンドルするモジュールであり，表4.3のメソッドを持つ．Java RMI のインタフェース *notify()*，*notifyRT()* の引数として，イベントが渡される．受け取ったイベントに対して現在処理可能な状態であるかを判断し，EventFilterModule クラスおよび ActionModule クラスに受け渡す．

EventInputModule クラスは，表4.4で示すパラメータを保持し，入力されたイベントをフィルタ処理すべきかを判断している．図4.3および図4.4でパラメータを用いたソフトリアルタイム配送の受付とノンリアルタイム配送の受付を示した．

表 4.3: EventInputModule クラスのメソッド一覧

public String notifyRT(RemoteEvent event) throws RemoteException, RejectException;
public void notify(RemoteEvent event) throws RemoteException, RejectException;

表 4.4: EventInputModule クラスのパラメータ

MaxNumberofFilteringThreads_SRT	SRTで同時にフィルタを行えるスレッド数の上限数．通常は3
MaxNumberofFilteringThreads_NRT	NRTで同時にフィルタを行えるスレッド数の上限数．通常は0
MaxNumberofEventsInSideAComponent	同時にコンポーネント内部で保持できるイベントの数
WaitingTimeForFilteringThread	MaxNumberofFilteringThread_SRTの閾値を超えたため待たせておくことができる時間

```
//NRTのフィルタ実行用件満たしているかを検査
if(_countsNumberofEventsInSideAComponent()<=MaxNumberofEventsInSideAComponent&&
   _countsNumberofFilteringThread_SRT()==0&&
   _countsNumberofFilteringThread_NRT()<=MaxNumberofFilteringThreads_NRT)
    EventFilterModule ft = new EventFilterModule(event,pre_catched_Time,false);
    ft.setPriority(Thread.MIN_PRIORITY);//最低優先度で起動
    ft.start();
}else{
    if(isRetryflag){
      isRetryflag=false;//一定時間待機
      try{ Thread.sleep(WaitingTimeForFilteringThread);}
      catch(InterruptedException e) { }
    }else{
      _eventAvoid(event);//2度目失敗
    }
}
```

図 4.3: EventInputModule クラス内メソッド notify() の一部

```
//SRTのフィルタ実行用件満たしているかを検査
if(_countsNumberofEventsInSideAComponent()<=MaxNumberofEventsInSideAComponent&&
   _countsNumberofFilteringThread_SRT()<=MaxNumberofFilteringThreads_SRT)
    EventFilterModule ft = new EventFilterModule(event,pre_catched_Time,true);
    ft.setPriority(Thread.MAX_PRIORITY);//最高優先度で起動
    ft.start();
}else{
    if(isRetryflag){
      isRetryflag=false;//一定時間待機
      try{ Thread.sleep(WaitingTimeForFilteringThread);}
      catch(InterruptedException e) { }
    }else{
      _eventAvoid(event);//2度目失敗
    }
}
```

図 4.4: EventInputModule クラス内メソッド notifyRT() の実装

4.4.2 EventFilterModule クラス

イベントの内部をチェックし，以下の EventOutputModule クラスへ通すか通さないかを決定する．EventFilterModule クラスを経由した場合，EventOutputModule クラスへイベントが渡される．EventFilterModule クラス，FilterObject クラスのインスタンスを保持している．その FilterObject は，表4.5に示すように *updateFilterObject(FilterObject newfilter)* メソッドによって変更可能である．

表 4.5: EventFilterModule クラスのメソッド一覧

void updateFilterObject(FilterObject filter);
void setPriority(int);

図4.5に EventFilterModule クラスの run() メソッドの内容を示す．

FilterObject クラス

本クラスは，表4.6に示すメソッドを持つ．*filterEvent(Object obj)* および *filterEventRt(Object obj)* の返り値は表4.7に示すフィールドを利用する．

4.4.3 EventOutputModule クラス

1対多でイベントを出力するクラスである．イベント配送登録を行う場合，Commnader クラスが先ず下流にあるコンポーネントの参照オブジェクトを配送依頼する上流

```
int filterState = filtermodule.filterEvent(event.getRegistrationObject());
int filterState2Act = filtermodule.filterEvent2Action(event.getRegistrationObject());

if(filterState==FilterObject.EVENT_PASS)
        outmodule.fireRemoteEvent(//EOMにイベントを伝播
                eventSerialNiumber,
                event.getRegistrationObject(),
                preevent_catched_time,
                isRTEvent
        );

if(filterState2Act==FilterObject.EVENT_PASS){
        try{
        MarshalledObject mo = (MarshalledObject)event.getRegistrationObject();
        ami.action(mo.get());//AMにイベントを伝播
        }catch(ClassNotFoundException me){}
}

if(filterState==FilterObject.EVENT_DENY){
        System.out.println("pass.deny");
}
if(filterState2Act==FilterObject.EVENT_DENY){
        System.out.println("act.deny");
}

if(filterState==FilterObject.EVENT_EXCHANGE)
        outmodule.fireRemoteEvent(//新規にイベントを取得
                eventSerialNiumber,
                new java.rmi.MarshalledObject(filtermodule.getNewEvent()),
                preevent_catched_time,
                isRTEvent
        );
if(filterState2Act==FilterObject.EVENT_EXCHANGE)
        ami.action(
                filtermodule.getNewEvent()//新規にイベントを取得
        );

if(filterState==FilterObject.EVENT_WAITING){
        waitEvent(mo.get());
}
if(filterState2Act==FilterObject.EVENT_WAITING){
        waitEvent2Act(mo.get());
}
```

図4.5: EventFilterModuleクラス内メソッドrun()の実装

表 4.6: FilterObjectのメソッド一覧

abstract public int filterEvent(Object obj);	EOMへの通過チェック
abstract public int filterEvent2Action(Object obj);	AMへの通過チェック
abstract public Object getNewEvent();	イベントの入れ替え

表 4.7: FilterObjectのフィールド一覧

public final static int EVENT_PASS=1;	イベント通過可能
public final static int EVENT_DENY=2;	イベント通過拒否
public final static int EVENT_EXCHANGE=3;	イベント変更命令
public final static int EVENT_WAITING=4;	イベント待機命令

コンポーネントのEventOutputModuleクラスに対して登録する．EventOutputModuleクラスは，配送先を記録したテーブルを保持している．配送登録には，リアルタイム性を伴う配送を要求する登録とリアルタイム性を要求しない登録の2種類がある．その登録に基づき，イベントを受け取った上流のコンポーネントは，登録されている配送先に対してイベントの配送を行う．本クラスは，表4.8に示すメソッドを持ち，表4.9のパラメータを持つ．

表 4.8: EventOutputModuleのメソッド一覧

public EventRegistration registerDragonListener(EventInPutInterface listener, MarshalledObject handback) throws RemoteException,RegistryDenyException; EventOutputModuleへノンリアルタイム属性でイベント配送登録を行う．
public EventRegistration registerDragonListenerRT(EventInPutInterface listener, MarshalledObject handback) throws RemoteException,RegistryDenyException; EventOutputModuleへソフトリアルタイム属性でイベント配送登録を行う．
public void canselDragonListener(EventInPutInterface listener) throws RemoteException; イベント配送登録をキャンセルする．
public void canselDragonListenerRT(EventInPutInterface listener) throws RemoteException;
public void updateInformation() throws RemoteException;
void fireRemoteEvent(long serialeventid,MarshalledObject mo,long pretime,boolean isRTEvent) throws RemoteException;

EventOutputModuleクラスは，登録がノンリアルタイムの場合は図4.6によって配送を行い．登録がソフトリアルタイムの場合は，図4.7により配送を行っている．イベント配送登録は，図4.8に示す．

表 4.9: EventOutputModule クラスのパラメータ

MaxNumberofAcceptableComponents_SRT	ソフトリアルタイムで配送を行える下流コンポーネントの上限数．通常は10
MaxNumberofAcceptableComponents_NRT	NRTで配送を行える下流コンポーネントの上限数．通常は200
WatingTimeforFiringEvent_SRT	すでに配送中だった場合配送待ちを行える時間

```
if(!isRTFireing()){//リアルタイム配送を行っていない場合
  Enumeration enum = core.listeners.keys();
  while(enum.hasMoreElements()){
  EventInPutInterface listener = null;
   try
    {listener = (EventInPutInterface)enum.nextElement();
    RemoteEvent event =
      new RemoteEvent(this,id,evt);

    if(!isRTFireing())//再びリアルタイム配送を行っていない
      listener.notify(event);
    else
      break;
    }
   catch(UnknownEventException e)
    { ...一部省略...}
  }
}
```

図 4.6: EOM におけるノンリアルタイムイベント配送のコード

```
//softRealTimeOrderingTableMapから配送順に
//通知先を取り出す
 Collection c = softRealTimeOrderingTableMap.values();
 Iterator it = c.iterator();
 while(it.hasNext()){
  EventInPutInterface listener = null;
   try{
    listener = (EventInPutInterface)it.next();
    RemoteEvent event =
      new RemoteEvent(this,id,time,mo);

    String componentID=listener.notifyRT(event);
    _calcDeadLine(listener,componentID);
   ...省略...
    }
   catch(RemoteException e){   }
}
```

図 4.7: EOM におけるソフトリアルタイムイベント配送のコード

```
 public synchronized EventRegistration
  registerDragonListenerRT(
    EventInPutInterface listener,
    MarshalledObject handback)
  throws RemoteException,RegistryDenyException{
  try   {
    ...一部省略...
  //登録もとのデッドラインを取り出す
   DragonEventAttachedObject  dea = (DragonEventAttachedObject)handback.get();
  //登録もとのデッドラインを検査
   if(_isAcceptableNumberofComponents()&&
      _isAcceptableDeadLine(dea.getDeadLine())){
     _insert2SoftRealTimeOrderingTableMap(dea.getDeadLine(),listener);
     ...一部省略...
    }
   return new EventRegistration(regid,this,null,seqNum);
  }
 ...省略...
}
```

図 4.8: ソフトリアルタイムイベント登録メソッドの実装

4.4.4 SensorModuleクラス

SensorModuleクラスは，主にセンサなどの入力機器からのイベントを非同期で発生させるメソッドを持つ（表4.10参照）．情報家電機器などのコンポーネントプログラマによって記述されるモジュールである．イベントが発生すると*generateEvent(evt)*メソッドを起動する．

表 4.10: SensorModuleクラスのメソッド一覧

void generateEvent(Object evt);

4.4.5 ActionModuleクラス

開発者がイベントを受け取ったあとの挙動を記述するモジュールであり，表4.11のメソッドを持つ．ホストに依存している処理内容を，ActionModuleクラスの*action()*メソッド内部にプログラマが実装する．

表 4.11: ActionModuleクラスのメソッド一覧

abstract void action(Object evt);

4.4.6 ComponentManagementModuleクラス

本クラスは，JiniLookupServiceと通信を行う．ソースコードを図4.9に示す．ComponentManagementModuleクラスには，RegisterThreadインナクラスおよびRenewThreadインナクラスがある．RegisterThreadインナクラスは，JiniLookupServiceに対してEOMおよびEIMの登録を行う．登録が完了するとすぐさま，RenewThreadインナクラスが呼び出され，Leaseが開始される．

4.4.7 イベント配送を担う3つのコンポーネントタイプ

Dragon内でのコンポーネントは，以上に示したモジュールから構築されるBasicComponentクラスが基本となる．本研究では，構築されるシステムをよりシンプルで，利用者が理解しやすい形にし，プログラマが開発を行いやすくするために，図4.10のように3つの役割をもったコンポーネントクラスを提供した．

EventSupplierクラス drg.core.component.EventSupplierクラスは，図4.2が示すようにEventOutputModuleクラスを保持する生産者コンポーネントである．本クラ

```
public class RegisterThread extends Thread {
  DiscoveryEvent evnt ;
  public RegisterThread( DiscoveryEvent evnt ){
    .......一部省略.......    }
  public void run() {
    ServiceRegistrar[] regs = evnt.getRegistrars();
      serviceitem = new ServiceItem(null, upperobj, attrs );
      for(int i=0; i<regs.length;i++){
      ServiceRegistration serviceResigtration = null ;
        try {
        serviceResigtration = regs[i].register( serviceitem, LEASE_DUR_MSEC);
        registrationVec.add(serviceResigtration);
        } catch (RemoteException e) {
        System.out.println("RemoteException while registering the "
             +"Service: "+regs[i]+"\n"+e.toString());
        lookupDiscovery.discard(regs[i]);
      return ;
        }
      System.out.println("ServiceRegistration ==> " + serviceResigtration);
      ServiceID serviceID = serviceResigtration.getServiceID();
      if(renewThread==null){
              renewThread = this.new RenewThread();
              renewThread.start();
      }
     }
  }
}
public  class RenewThread extends Thread {
  public RenewThread() {super("renewThread"); }
  public void run() {
    while (true) {
      try {
        Thread.sleep(LEASE_DUR_MSEC/2);//一定時間ごとにリースを行う.
      } catch(InterruptedException e) { }
      try {
          for (int i=0;i<registrationVec.size();i++) {
            ((Lease)
               ((ServiceRegistration)
                   registrationVec.elementAt(i)
               ) . getLease()
            ) . renew(LEASE_DUR_MSEC);
          }
      } catch (Exception e) {}
    }
  }
}
```

図 4.9: ComponentManagementModule クラスの実装

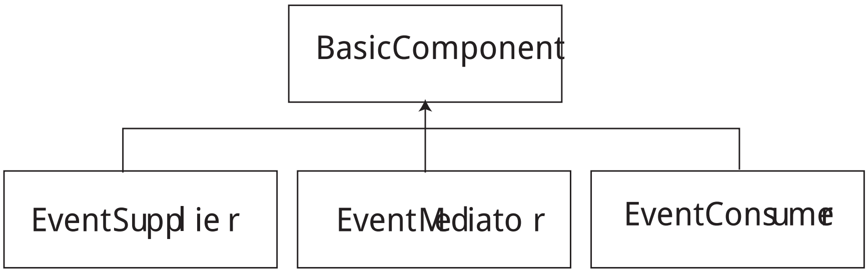


図 4.10: 3つのコンポーネントとBasicComponentの継承関係

スのインスタンスの生成には，SensorModuleクラスをコンストラクタに引き渡す(図4.12参照)．EventSupplierクラスを用いたコンポーネントの実装例を付録A.1に示す．

表 4.12: EventSupplierクラスのコンストラクタ

型	インスタンス名	説明
String	hostname	ホスト名
String	name	サービス名
String	url	アイコンのURL
SensorModule	sm	SensorModuleクラスのインスタンス

EventMediatorクラス drg.core.component.EventMediatorクラスは，複数の仲介者を連結できることが前提で実装されている．そのため，図4.2が示すようにEventOutputModuleクラス，EventInputModuleクラスを持つコンポーネントとなっている．さらにEventMediatorクラスのコンストラクタにはFilterObjectクラスが必要になる(図4.12参照)．EventMediatorクラスを用いたコンポーネントの実装例を付録A.2に示す．

表 4.13: EventMediatorクラスのコンストラクタ

型	インスタンス名	説明
String	hostname	ホスト名
String	name	サービス名
String	url	アイコンのURL
FilterObject	filter	FilterObjectクラスのインスタンス

EventConsumer クラス drg.core.component.EventConsumer クラスは，EventInputModule クラスでイベントの到着を待機し，イベントの内容に基づいた処理を行うクラスである．処理内容は ActionModule クラス内部で記述し，EventConsumer クラスのコンストラクタに引き渡す（図 4.14参照）．EventConsumer クラスを用いたコンポーネントの実装例を付録 A.3に示す．

表 4.14: EventConsumer クラスのコンストラクタ

型	インスタンス名	説明
String	hostname	ホスト名
String	name	サービス名
String	url	アイコンの URL
ActionModule	am	ActionModule クラスのインスタンス

4.5 SettingCommander クラス

コンポーネントを組み合わせて分散アプリケーションを構成するために SettingCommander クラスを提供している．

SettingCommander クラスは，設計のコマンダサブシステムの実装クラスである．システム内においてノンリアルタイム配送が大量に発生した場合に，イベントのオーバフローを検知し，ソフトリアルタイム配送の状況の変化を SettingCommander クラスに収集する，各コンポーネントからは，通信先のおよび配送の平均時間と最悪値等が同時に伝えられ，利用者がすぐさま状況を認識することが可能となっている．

経路判断は，この SettingCommander クラスに一元化される．このクラスの API を用いて，利用者がコンポーネント間を動的に決定する役割を持たす GUI を構築できる．障害で終了した場合に対処できるように経路情報は各コンポーネントに分散してさせて保持する．SettingCommander クラスは起動されるたびに各コンポーネントから通知先のリストを取得する．

同クラスは表 4.15のメソッドを持つ．各メソッドの ComponentTemplate クラスは，ComponentID とコンポーネント名，FilterObject クラス名からなる．指定し ComponentID 等を認識していない場合は，ComponentTemplate(null,”Printer01”,null) の様に記す．

4.5.1 ComponentSearcherModule クラス

ComponentSearcherModule クラスは，JiniLookupService を発見し，LookupService が管理するコンポーネントを保持する．また JiniLookupService に新たにコンポーネントが追加された場合通知するように設定しておく．図 4.11に ComponentSearcherModule クラス実装を示す．

表 4.15: SettingCommander クラスのメソッド一覧

connectPushOnce(ComponentTemplate tmp1, ComponentTemplate tmp2, boolean isRT)
3.1.5項の Push 型通信駆動で二つのコンポーネントを接続するメソッド． tmp1 から tmp2 へ現在コンポーネントに保持されているイベントを非同期に配送する．
connectPullOnce(ComponentTemplate tmp1, ComponentTemplate tmp2, boolean isRT)
3.1.5項の Pull 型通信駆動で二つのコンポーネントを接続するメソッド． tmp2 から tmp1 へ現在コンポーネントに保持されているイベント取得する．
connectPushForever(ComponentTemplate tmp1, ComponentTemplate tmp2, boolean isRT)
3.1.5項の CallBack 型通信要求で二つのコンポーネントを接続するメソッド． tmp1 に tmp2 の参照を保持させ，イベントが発生するたびに tmp2 への配送を行わせる．
connectPullForever(ComponentTemplate tmp1, ComponentTemplate tmp2, boolean isRT)
3.1.5項の FuturePull 型通信要求で二つのコンポーネントを接続するメソッド． tmp2 に tmp1 の参照を保持させ一定周期で tmp1 へイベントを取得させる．
disConnect(ComponentTemplate tmp1, ComponentTemplate tmp2)
tmp1 と tmp2 間のイベントの配送をキャンセルする
getSupplierConponents()
現在起動されているイベント生産者を取得
getConsumerConponents()
現在起動されているイベント消費者を取得
getMediatorConponents()
現在起動されているイベント仲介者を取得

```
public class ComponentSearcherModule implements Runnable{
  ...省略...
  //↓ コンストラクタ
  public ComponentSearcherModule(String[] groups,DragonListener dragonEventListener)
    throws Exception{
  this.dragonEventListener=dragonEventListener;
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }
    LookupDiscovery discovery =
      new LookupDiscovery(groups);
    discovery.addDiscoveryListener(new Discoverer());
  }

  class Discoverer implements DiscoveryListener {
    public void discovered(DiscoveryEvent ev) {
      newregs = ev.getRegistrars();
      for (int i=0 ; i<newregs.length ; i++) {
          registrars.put(newregs[i].getServiceID(),newregs[i]);
          try{//コンポーネントに変化があった場合の通知要請を行う．
          System.out.println("transition to lus");
           EventRegistration reg = newregs[i].notify(
                         new ServiceTemplate(null,null,null),
                         transitions,
                         dragonEventListener,
                         null,
                         Lease.ANY);
          leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, null);
          }catch(RemoteException re){
          }
      }
    }
  ...省略...
  }
...省略...
}
```

図 4.11: ComponentSearcherModule クラスの実装コード

4.5.2 SettingCommanderクラスの実装例

SettingCommanderクラスのAPIを用いて実装されたComponentsBindingEditorを図4.12に示す．ComponentsBindingEditorクラスは，高度な知識をもった開発者が存在しない即興的システム構築を可能にするためのツールである．図4.12はネットワーク上で3つの生産者と2つの仲介者と5つのイベント消費者を起動しGUIにより接続命令をおこなったものである．このGUIの基盤となるSettingCommanderクラスは新たに起動されたコンポーネントや障害が起きたコンポーネントを認識でき, 高度な知識のない利用者が容易に分散システムを構築可能になる．ソフトリアルタイム配送を行った場合は，図4.13に示すように各線の上に過去のイベント配送に関する最短時間，平均時間，最悪時間が表示されイベントのオーバフローを認識することが可能である．

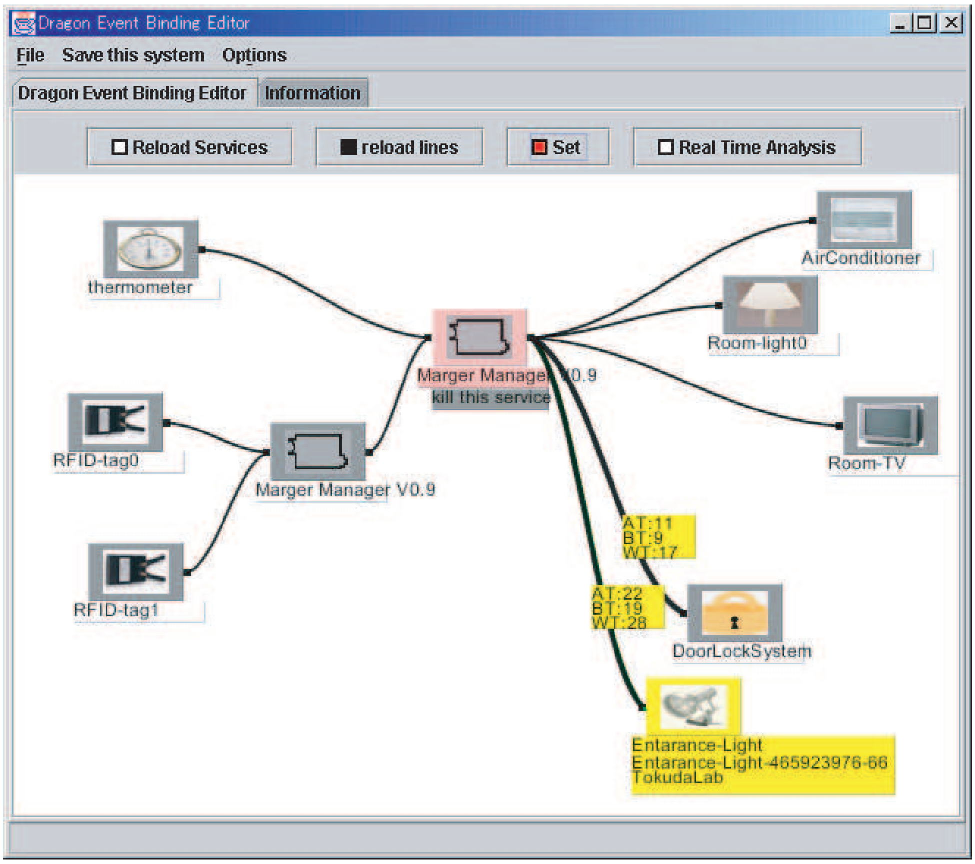


図 4.12: SettingCommanderクラスを実装したGUI

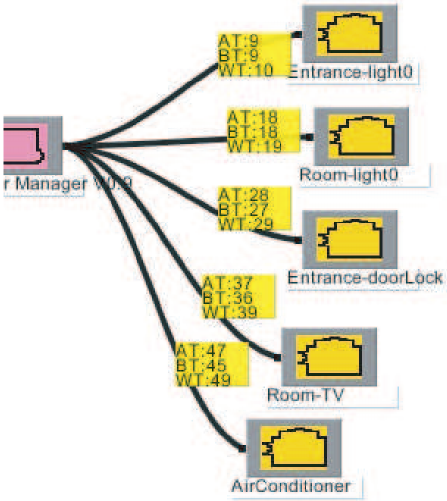


図 4.13: 最短時間 BT，最悪時間 WT，平均時間 AT の表示

4.6 DragonAPI のパッケージ構成

DragonAPI のパッケージ構成を表 4.16に示す．*drg.event.flowtype*, *drg.filter*, *drg.components* は，現時点でクラス数が少ない．今後実装を増やコンポーネント開発者のために十分に用意する．本 API を利用した Dragon の起動手順は付録 Bに示す．

表 4.16: DragonAPI のパッケージの構成

パッケージ名	説明	主なクラス
drg.core.component	各コンポーネントを構成するコア部分 25 クラス 1602 行	BasicComponent ComponentManagement
drg.core.ui	コマンダ関連クラス 13 クラス 1591 行	SettingCommander ComponentSearcher
drg.gui.editor	UI 関連クラス 14 クラス 2677 行	ComponentsBindingEditor
drg.event.datatype	Dragon が提供している EventObjec 群	DToken
drg.event.flowtype	Dragon が提供している制御型イベント群	FilterUpdatableEvent
drg.filter	Dragon が提供している FilterObject 群	DTokenPassFilter
drg.components	Dragon が提供しているサンプル Component 群	SampleSupplier

4.7 本章のまとめ

Dragon は，Java2 および Jini を基盤として実装した．BasicCommander サブシステムとして BasicCommander クラス，コマンダクラスとして SettingCommander クラスの実装を述べた．各コンポーネント開発者が知っておくべきクラスのメソッドとフィー

ルドを示した．本章の最後で，各コンポーネントのサンプルコードと実行手順を記述した．次章では，実装されたDragonが分散システムを即興的に構成できる機能用件をそれぞれ満たしているかを検証する．

第 5 章

Dragonの基本性能の評価

本章では3.1節で述べた近接コンポーネント認識機能，経路判断機能，コンテンツ適応機能，信頼性のある1対多通信機能，システム障害可能性検知機能の実装方針が正しく動作しているかDragonの基本性能を含めて以下で説明する．

5.1 測定環境

測定環境には，100Base-T で相互接続された，表5.1に示す機器を利用した．測定用コンポーネントには，1回ごとに8バイトの付加情報を保持させたイベントの伝搬を行う．

表 5.1: 測定環境

項目	マシン A(mugen)	マシン B(dali)	マシン C(vivi)
CPU	Duron 800MHz	UltraSPARC-II 248MHz×2	PentiumIII 750MHz
主記憶	128MB	512MB	256MB
OS	FreeBSD4.4-RELEASE	Solaris7	Windows2000
JDK	JDK1.3.1-patch4	HotSpot Client VM 1.3.1	HotSpot Client VM 1.3.0

5.2 SettingCommander クラスのコンポーネント発見完了と障害検知時間

経路判断機能をもつSettingCommander クラスは，近接コンポーネントの発見の障害も把握も行わなければならない．そのために本実験では，コンポーネントが起動されてからSettingCommander クラスに検出されGUI に反映されるまでの時間と，コンポーネントに障害が起こりダウンしてしまった場合の障害検知時間を測定した．図5.1の横軸は各コンポーネントがLease を行う周期を表し，縦軸にSettingCommander クラスに検出されるまでの経過時間を表している．それぞれのLease 時間でコンポーネントを動作させ100 回の試行をし，その平均値を示している．

コンポーネント障害検知機能は，Lease 周期の設定を短くすることで障害から10 秒以下で反映されることが可能になり家庭内などでは実用に耐えるといえる．但し，Lease 周期を短くしすぎるとLease 更新にかかる通信コストおよびLookupService の負荷が増加してしまうため，Lease は一定値より低くならない方がよい．実際には，図4.9のソースコードようにLease 周期が切れる十分前にLease 更新を行わなければネットワークの状況によっては間に合わない場合があるためLease の周期を短く設定しすぎることは付加が高くなる．発見完了時間は，ほとんどLease に影響はなく，100 ミリ秒以内にSettingCommander クラスに伝達され近接コンポーネント認識から経路判断が行える．

5.3 EventMediator 群の直列な配送時間

Dragon では，コンテンツ適応機能を複数の構造的なEventMediator 群で行っている．このためEventMediator が直列に接続された場合のオーバヘッドを測定し，コンテンツ適応機能の妨げにならない値であるかを調べる．マシンA とマシンB の間で起動しているEventMediator 間を交互に接続し，EventMediator の間を直列的に配送するため

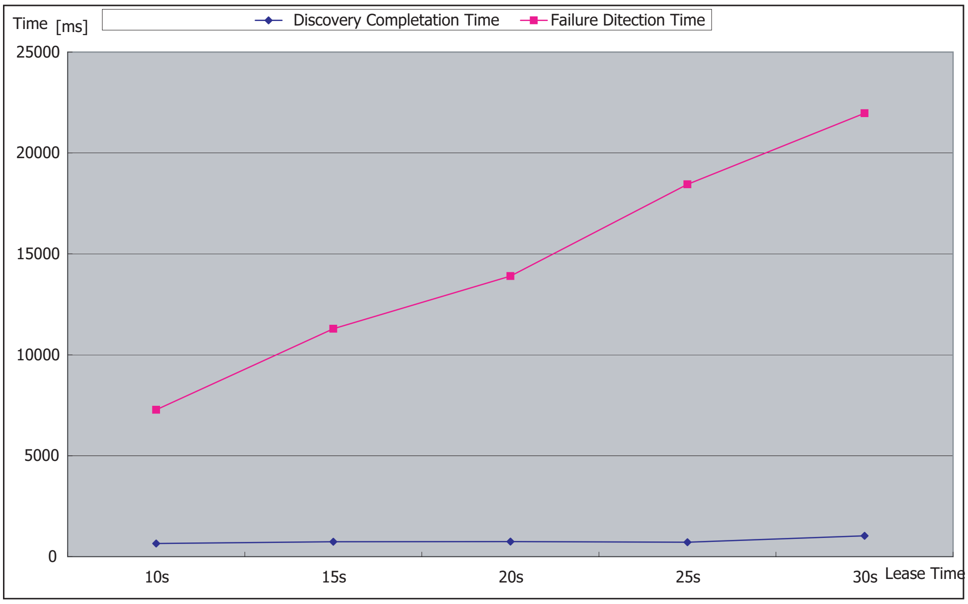


図 5.1: 発見完了時間及び障害検知時間

に要する時間を測定する．EventMediator 内部の処理は行わない．コンポーネントが配送を終えるまでの時間を正確に測定するために生産者と消費者は同一のホスト上で行い測定した．その結果，図 5.2に示す結果が得られた．全配送時間を y，直列につながっているコンポーネントの数を x とすると，式 $y = 11.083x - 0.46$ で近似できた．理論上望ましいのは，直列 1 の場合の計測値 10.222 に比例する $y = 10.222x$ であるが，与えられた式は CPU のオーバヘッドを考慮すると妥当な式と言え，直列に並ぶコンポーネントの数が多くなっても影響を受ける時間は，予測可能である．

このためコンテンツ適応機能のオーバヘッドを 1 秒と許容すると約 90 個の EventMediator を直列に配置可能であり，家庭ネットワークや小規模ネットワークでの実用に十分である．

5.4 イベント入力時及びフィルタ処理時におけるソフトリアルタイム性の評価

Dragon で信頼性のある 1 対多通信機能が満たされていること評価するために，まずソフトリアルタイムイベント配送は，優先的にフィルタ処理が行われていることを示す．本測定には以下のポアソン分布に基づく記号を利用する．$NonRTN(d,p)$ は，分散 d ミリ秒，周期 p ミリ秒で上流のコンポーネントからイベントが到着している周期を表す．同様に $RTN(d,p)$ は，分散 d ミリ秒，周期 p ミリ秒で上流のコンポーネントからソフトリアルタイム配送でイベントが到着している周期を示している．

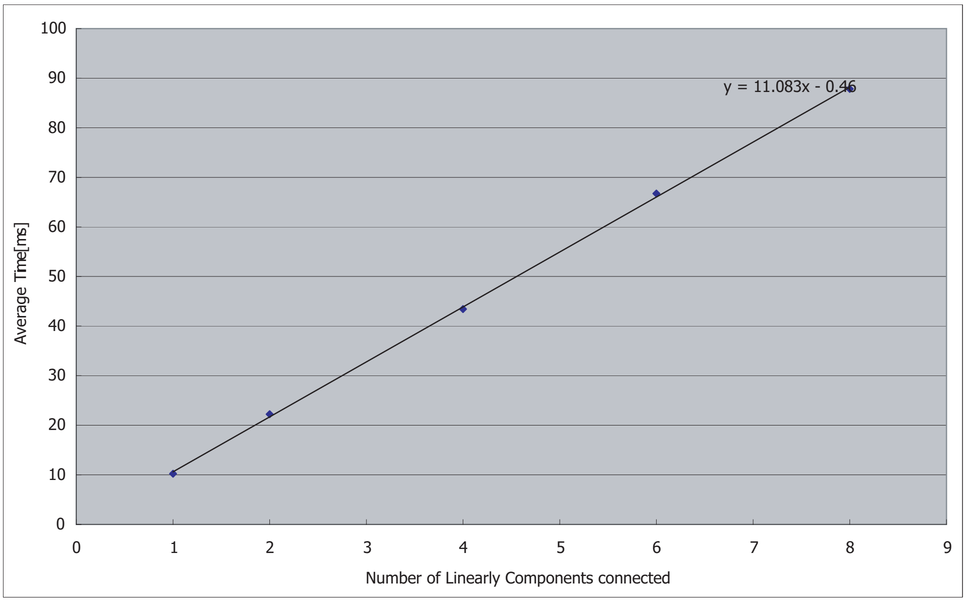


図 5.2: 仲介者の直列配送の時間測定

図5.3では，マシンCにおいてあるメディエータにおいて$NonRTN(5,80)$と$RTN(5,80)$のイベントが到着している状態を示している．そのなかで，それぞれのイベントが入力からフィルタで処理終了されるまでの時間を横軸にとり，縦軸にそのイベントのフィルタ累積成功率を示す．フィルタの負荷として0から15000の間から素数の数をカウントするダミー処理を行う．図5.4では，ノンリアルタイムの周期を$NonRTN(5,20)$にし，負荷を高くした時点の状態を表している．図5.3と図5.4を比較すると，ノンリアルタイムの配送の成功率が低下しているが，ソフトリアルタイム配送の処理には影響を与えていないことが累積成功確率曲線から分かる．このことからイベント入力時およびイベントフィルタ処理で，信頼性のあるソフトリアルタイム配送が行われていることが分かる．これにより，信頼性のある時間見積りが可能であれば，最悪時間を測定した結果を基に行うシステム障害可能性検知機能が有効になる．

5.5 イベント配送時のソフトリアルタイム性の評価

イベント配送時にソフトリアルタイム配送機構がある場合とない場合での評価をおこなった．図5.5では横軸に1秒間のイベント数を取り，縦軸にそのイベントがマシンB上の5つの消費者コンポーネント対するに配送成功率を検証した．ソフトリアルタイム配送では5つの消費者のうち2つをソフトリアルタイム配送に登録し，その他のものをノンリアルタイム配送として登録した．アドミッションコントロールがない場合は，5つの消費者にイベントを配送する要求を制御しないまま1対多通信を行っている．通常，1つのイベントの到着に対して5つの消費者へイベントが配送されるが，処理しき

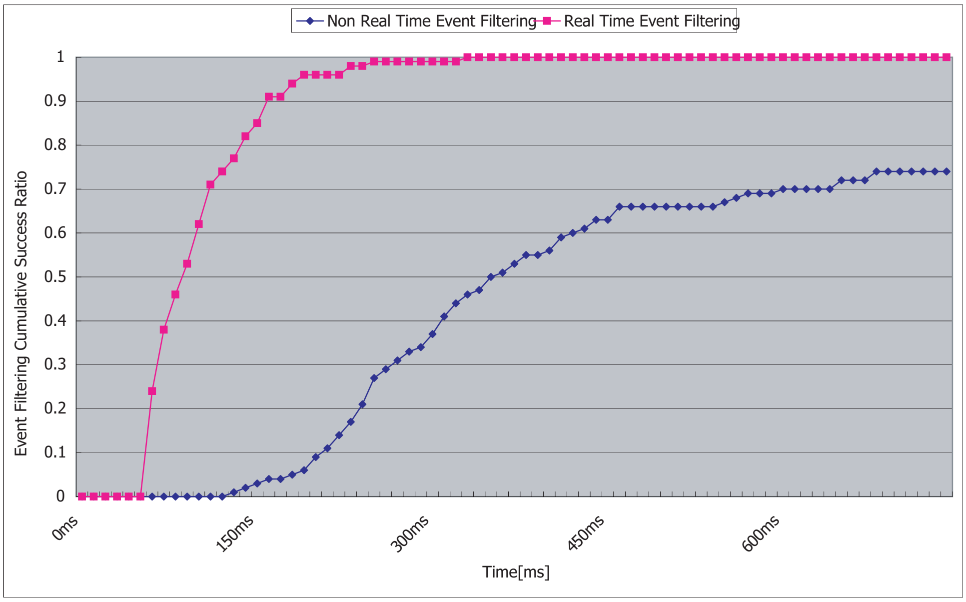


図 5.3: $NonRTN(5, 80)$, $RTN(5, 80)$ システムにおけるフィルタの成功率

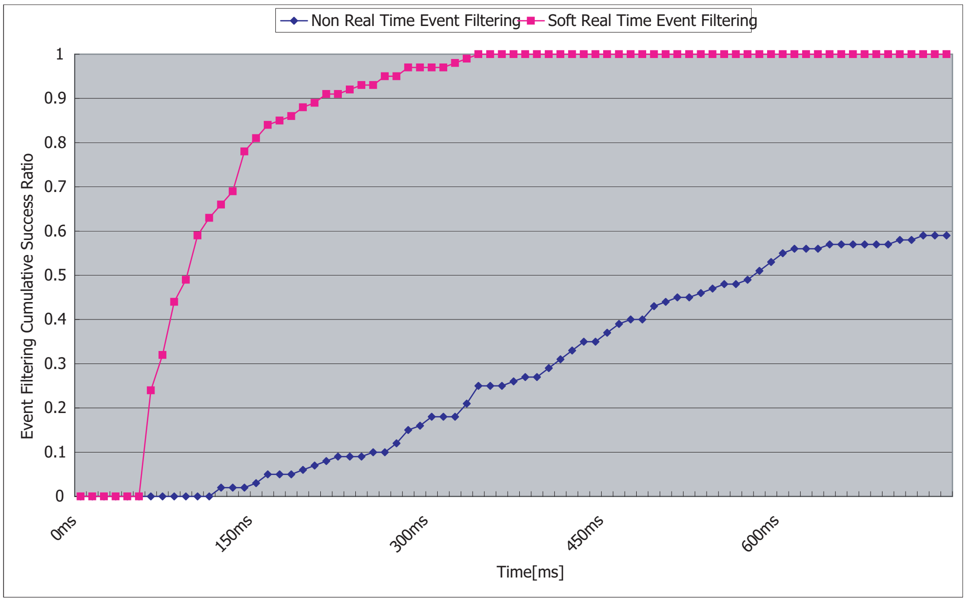


図 5.4: $NonRTN(5, 20)$, $RTN(5, 80)$ システムにおけるフィルタの成功率

れないイベントは破棄されていく．図5.4では，通常の配送ではイベント配送の1秒あたりのイベント数（1/配送周期）が高まると成功率が下がっているが，ソフトリアルタイム配送は高い配送成功率をたもっているため，信頼性のある1対多通信が実現できていることが分かる．特に周期15ミリ秒での比較は，デフォルトのマルチプルユニキャストの50%以上の成功率を示し，優先度をつけたマルチプルユニキャストが正しく動作していることを示している．

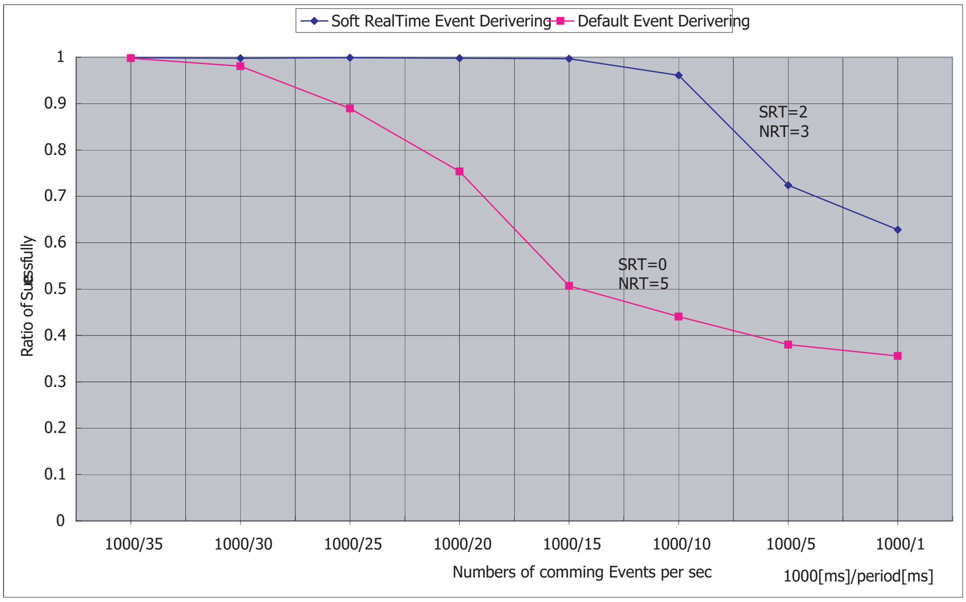


図 5.5: 1対多通信の成功率

5.6 本章のまとめ

本章では，Dragonの評価を行った．Dragonは20ミリ秒の周期という負荷の高いノンリアルタイムイベント配送がある中で，ソフトリアルタイム性をもたせたイベント処理は負荷が80ミリ秒の時とほぼ同一の累積成功確率曲線を示し，負荷の高い配送においても優先度処理が保たれていることを示した．またDragonは，周期15ミリ秒の1対多通信において，優先度を区別したマルチプルユニキャストがデフォルトの50%以上も成功率が高い結果を得られた．そのためDragonは家庭内ネットワークだけでなく時間制約のある工場等のシステムに対する応用も想定できる．また仲介者コンポーネントであるEventMediatorを連結した配送のオーバヘッドを測定した結果，オーバヘッドが1秒以上となるには約90個の連結まで許容できることが示せたため，Dragonは小規模ネットワークで稼動可能といえる．

第 6 章

関連研究との比較

本章では，分散コンポーネント間接続を可能とするミドルウエアとしてCORBA Event Service，JavaSpaces，JECho，SIENA，Carp@を挙げ，それぞれDragonと比較しながら議論する．

6.1 CORBA Event Service

CORBA Event Service は，イベント消費者に対して，push メソッドのみを用いれば，仲介者主導型間接モデルの典型であるが，pull メソッドを備えているため一時的なオブジェクトの格納を行う役割を持たせることも可能である．またイベント生産者からの push メソッドや，イベント生産者に対する pull メソッドインタフェースを規定したもので，開発者はこのインタフェースに基づいて仲介者を実装できる．しかし複数のコンポーネントがある中で，即興的に分散システムを構築したい場合，すべての生産者コンポーネントと消費者コンポーネントに対す変換を想定し，中心的な仲介者である Event Service をつくり込む必要がある．そのため様々なコンポーネントの種類と低機能を前提とする即興的な分散システム構築には向かない．

6.2 JavaSpaces

JavaSpaces は，Jini を基盤としたシェアードスペースプログラムを可能にする．*read()*，*write()*，*take()* というシンプルなオペレーションをスペースに対して行うことが可能である．引き渡されるオブジェクトはすべて net.jini.core.entry.Entry インタフェースを実装したオブジェクトであり，値渡しでなければならない．クライアントが *take()* を行い，一度オブジェクトに対して作業を完了すると，*write()* によって再びスペースに戻す作業を行う．クライアントがお互いを認知することがない即興的な分散処理システム構築には有効であるが，複数のコンポーネントからなるヘテロジニアスな環境でクライアントを相互に理解しなければならない場合には不向きである．Dragon は構造的な仲介者群を用いているため，イベントのコンテンツの変換を行う EventMediator が介在できる．相互性のないコンポーネント間の異なるイベントコンテンツによるプリケーションが構築できる．

6.3 JECho

Java Event Channel は，配送されるクライアントグループごとにチャネルを定義し，そのチャネルに基づいて高速にイベント通知を通気を行う．独自の名前空間を持ち生産者とチャネルを結び付けている．しかし直接的な通信トポロジのため，プログラマは互いのインタフェースを熟知していなければならない．一方 Dragon で採用した配送モデルは，複数の仲介者を介した間接的な通信モデルであり，すべてのコンポーネントのインタフェースを知らずにイベント仲介者が変換を行える．また JECho のモデルでは，生産者が増加した場合，消費者がそのイベントを扱い切れない問題が起こる．

6.4 SIENA

SIENA[3] は，仲介者を構造化でき，本研究でも採用している複数仲介者間接通信モデルに当てはまる．SIENA は厳密に型付けされたイベントと pattern と呼ばれるイベントのマッチングルールにより，複数の仲介者に渡ってイベントのルーティングを動的に行うこと可能である．一方我々の Dragon モデルは，RMI によるオブジェクト渡しであればどのようなイベントでも配送可能であり，ルーティング自体に外部のアルゴリズムが反映できるように SettingCommander クラスを提供している．従って Dragon は自律的なルーティングに不向きであるが，個々のコンポーネントの機能を軽量化できる．Dragon で配送するイベントによって個々のコンポーネントは，より自由度が高いアプリケーションに適応でき，既存のソフトウエアに対して最小限の変更を行えば，Dragon として利用可能になる．

6.5 Carp@

Carp@[5] は，Jini のサービスに対する設定用 GUI ツールを提供している．バイトコードジェネレータを用いて，動的に Jini のサービスをラッパーさせるオブジェクトを生成する．このラッパーオブジェクトにより，GUI から操作可能な Carp@Bean と呼ばれるサービスとなる．*port* と *connector* を用いて Jini の詳細なメカニズムを認識することなくコンポーネント間のつながりを即興的に設定できる．しかし既存の Jini のサービスを基盤としているため，イベントの仲介を行うサービスは生成されず，やはり既知の Jini サービス間同士の通信になってしまう．そのためヘテロジニアスな分散アプリケーションを構築することは難しい．Dragon では，内部を通るイベントオブジェクトを自由に変更可能であるので，Jini を基盤としないコンポーネントを含めた分散アプリケーションが構築できる．また Carp@では提供されていないイベント配送の時間制約を提示と設定が可能であり，より時間や配送の信頼性に厳密さを要するアプリケーションにも応用できる．

第 7 章

結論

7.1　今後の課題

本ミドルウェアは，今後オープンソースとして広く配布し，そのフィードバックを元にさらに多くの分散システム構築の場面で利用されるとこを期待している．Dragonは，**http://www.ht.sfc.keio.ac.jp/˜tailor/dragon/**からソースコード，バイナリ，API，マニュアル，評価等が閲覧可能である．今後は，RMI以外のイベント配送手法への適応，Jini以外の分散サービス管理機構への適応，インターネットスケールでの自律的イベントルーティング応用範囲の広い分散アプリケーション構成機構への発展を予定している．さらによりリアルタイム性の高いJavaVM[2, 12]を用いることでハードリアルタイムを実現し，Dragon以外のソフトウェアが同時に使用されてる環境でも信頼性保つことを検討していく．以下でそれぞれの課題の詳細を述べる．

RMI以外のイベント配送手法への適応

Dragonは，すべてのイベント配送の入力ポートと出力ポートとしてJavaRMIを利用してきた．RMIがJini-Basedのサービスとの親和性があり，オブジェクト移送手段として広く使われている手法だからである．しかしRMIは，ファイヤウォールの制約や既存のJava以外のコンポーネントへの適応が不可能である．そのためDragonは，将来，イベントの通信手法としてHTTPやSMTPを利用したSOAP[16]との連携を考慮する．SOAPで伝達できうるイベントはXML形式のため表現力は，RMIより劣る．しかし今後UDDI[19]等のビジネス基盤プロトコルを採用する方針である．

Jini以外の分散コンポーネント管理機構への適応

本研究で，Dragonは，JiniのLookupServiceとLease機構を用い分散するコンポーネントの管理を行った．しかし今後の更なる偏在環境の変化を考えた場合Jini以外の分散サービス管理技術も透過的に扱える拡張を行う．JXTA[20]やVNAを考慮し，異なる分散サービス管理機構上でコンポーネント間を接続できるシステムを目指し，よりDragonの利用分野を広げていく予定である[13]．

インターネットスケールでの自律的イベントルーティング

SIENAのような自律的なアプリケーションレベルルーティングは，Dragonの想定しているような比較的小規模のネットワークでは必要ない．しかし将来的には，Dragonをインターネットスケールに拡張させる研究に合わせて課題としていく．その際に経路判断者として外部化したSettingCommanderクラスが利用できる．この課題の解決にあわせて複数のコンポーネント間で配送されるイベントがデッドロックやループに陥ることなく配送することを可能とする研究も行う．

7.2 本論文のまとめ

コンポーネントソフトウエアによる分散システムには，高度な知識をもった開発者が必要とされてきた．近年の家庭内ネットワークなどのシステム開発者の介在が困難な場面で，コンポーネント技術は有効利用されていなかった．本論文では高度な知識を持ったシステム開発者が介さずとも，一般的なユーザが，分散システム構成要素の入れ替わりや分散システム構成変更の要求が頻繁に起こる場合に，即興的に分散システムを構成可能なコンポーネント接続ミドルウエアDragonを構築した．Dragonは複数仲介者間接通信モデルに基づき，経路判断機能の外部化，コンテンツ適応機能の構造化，信頼性のある1対多通信機能などの特徴を持ち，構成要素の各コンポーネントが単純な機能をもつモデルである．ユーザが即興的に分散システムを構築可能とするDragonの各モジュールを利用したGUIツールを示した．このツールにより，即興的に分散システムを構築できる．

評価においてDragonは，20ミリ秒の周期という負荷の高いノンリアルタイムイベント配送がある中で，ソフトリアルタイム性をもたせたイベント処理は負荷が80ミリ秒の時とほぼ同一の累積成功確率曲線を示し，負荷の高い配送においても優先度処理が保たれていることを示した．またDragonは，周期15ミリ秒の1対多通信において，優先度を区別したマルチプルユニキャストがデフォルトの50%以上も成功率が高い結果を得られた．そのためDragonは家庭内ネットワークだけでなく時間制約のある工場等のシステムに対する応用も想定できる．また仲介者コンポーネントであるEventMediatorを連結した配送のオーバヘッドを測定した結果，オーバヘッドが1秒以上となるには約90個の連結まで許容できることが示せたため，Dragonは小規模ネットワークで稼動可能であり，時間的制約を伴う工場などに用いることが可能な即興的アプリケーション構築機構といえる．

謝辞

本研究の機会を与えてくださり，絶えず懇切丁寧なご指導を賜りました，慶應義塾大学環境情報学部教授徳田英幸博士に深く感謝致します．また，本論文の副査として貴重な助言を頂いた，慶應義塾大学環境情報学部教授萩野達也博士，同大学同学部助教授楠本博之博士に深く感謝致します．

慶應義塾大学政策・メディア研究科ITプログラムの指導教員の方，諸先輩方には，折りに触れ貴重な示唆や助言，ご指導を頂きました．特に慶應義塾大学政策・メディア研究科博士課程2年の中澤仁氏には本論文執筆にあたって絶えざる励ましと御助言を賜りました．また，KMSF研究グループの方々には大変お世話になりました．ここに，深い感謝の意を表します．

平成14年 2月 27日
岩井 将行

参考文献

[1] R. Boivie, N. Feldman, Y. Imai, W. Livens, D. Ooms, and O. Paridaens. Explicit Multicast (Xcast) Basic Specification, October 2001. draft-ooms-xcast-basic-spec-02.txt.

[2] Greg Bollella, Ben Brosgol, Peter Dibble, Steve Furr, James Gosling, David Hardin, and Mark Turnbull. *Real-time Specification for Java http://java.sun.com/aboutJava /communityprocess/first/jsr001/.*

[3] A. Carzaniga, D.S.Rosenblum, and A.L.Wolf. Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service. In *Nineteenth ACM Symposium on Principles of Distributed Computing*, July 2000.

[4] Dallas Semiconductor Corp. Tiny InterNet Interface. http://www.ibutton.com/TINI/index.html.

[5] Michael Fahrmair, Chris Salzmann, and Maurice Schoenmakers. A Reflection Based Tool for Observing Jini Services. In *Reflection and Software Engineering*, pp. 209–227, June 2000.

[6] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, Vol. 5, No. 6, pp. 784–803, 1997.

[7] D. Gelernter. Generative Communication in Linda, 1985. ACM Transactions on Programming Languages and Systems, 7(1):80–112, 1985.

[8] Todd D. Hodes and Randy H. Kats. Enabling smart spaces: Entity description and user interface generation for a heterogeneous component-based distributed system. In *DARPA/NIST Smart Smaces Workshop*, July 1998.

[9] Masayuki Iwai, Jin Nakazawa, and Hideyuki Tokuda. Dragon: Soft Real-Time Event Delivering Architecture for Networked Sensors and Appliances. In *The 7th International Conference on Real-Time Computing System and Applications(RTCSA2000)*, pp. 425–432, December 2000.

[10] Masayuki Iwai, Jin Nakazawa, and Hideyuki Tokuda. Flexible Distributed Event-Driven Programming Framework for Networked Appliances and Sensors. In *3rd International Symposium on Distributed Objects and Applications (DOA'01) Short Papers Proceedings*, pp. 61–68, September 2001.

[11] John C.-H. Lin and Sanjoy Paul. RMTP: A Reliable Multicast Transport Protocol. In *INFOCOM (3)*, pp. 1414–1424, 1996.

[12] Akihiko Miyoshi and Hideyuki Tokuda. Real-Time Java Server for Real-Time Mach. In *The 5th International Workshop on Parallel and Distributed Real-Time Systems*, April 1997.

[13] Jin Nakazawa, Masayuki Iwai, and Hideyuki Tokuda. Dynamic message-path generation for improvised composition of networked appliances services. In *International Workshop on Networked Appliance (IWNA4)*, January 2002.

[14] Jin Nakazawa, Tadashi Okoshi, Masahiro Mochizuki, Yoshito Tobe, and Hide Tokuda. VNA: An Object Model for Virtual Network Appliances. In *IEEE International Conference on Consumer Electronics*, June 2000.

[15] Jin Nakazawa, Yoshito Tobe, and Hideyuki Tokuda. On dynamic service integration in vna architecture. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. 7, No. E84-A, pp. 1610–1623, July 2001.

[16] W3C Note. Simple Object Access Protocol (SOAP) 1.1, May 2000. http://www.w3.org/TR/SOAP/.

[17] Object Management Group. The Common Object Request Broker Architecture and Specification 2.2ed CORBA Event Service, February 1998.

[18] Tadashi Okoshi, Shirou Wakayama, Yousuke Sugita, Soko Aoki, Takeshi Iwamoto, Jin Nakazawa, Tomohiro Nagata, Daichi Furusaka, Masayuki Iwai, Akihiko Kusumoto, Noriyuki Harashima, Junichi Yura, Nobuhiko Nishio, Yoshito Tobe, Yasushi Ikeda, and Hideyuki Tokuda. Smart space laboratoty project: Toward the next generation computing environment. In *IWNA2001*, February 2001.

[19] UDDI org. Uddi version 2.0 api specification, http://www.uddi.org/specification.html, June 2001.

[20] Sun Microsystems, Inc. JXTA v1.0 Protocols Specification. http://www.jxta.org/project/www/docs/ProtocolSpec.pdf.

[21] Sun Microsystems Inc. JavaSpaces Service Specification, version 1.1, October 2000. http://www.sun.com/jini/specs/js1_1.pdf.

[22] Sun Microsystems Inc. *Jini Architecture Specification*, October 2000. http://www.sun.com/jini/specs/jini1_1.pdf.

[23] Sun Microsystems Inc. *Jini Distributed Leasing Specification*, October 2000. http://www.sun.com/jini/specs/core1_1.pdf.

[24] Sun Microsystems Inc. *Jini LookupService Specification*, October 2000. http://www.sun.com/jini/specs/core1_1.pdf.

[25] Sun Microsystems, Inc. Enterprise Java Beans Technology, 2001. http://java.sun.com/products/ejb/.

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

[27] P. Wycko, S. McLaughry, T. Lehman, and D. Ford. Ibm systems journal, 1998. P. Wycko, S. W. McLaughry, T. J. Lehman, and D. A. Ford. TSpaces. IBM Systems Journal, 37(3), 1998.

[28] Yokogawa Electric Corporation. Duonus, 2000. http://www.yokogawa.co.jp/DUONUS.

[29] Dong Zhou, Karsten Schwan, Greg Eisenhauer, and Yuan Chen. JECho–Interactive High Performance Computing with Java Event Channels. In *International Parallel and Distributed Processing Symposium*, April 2001.

[30] 岩井将行, 楠本晶彦, 中澤仁, 徳田英幸. 情報家電機器間の動的なイベントバインディング機構の構築. 情報処理学会 システムソフトウェアとオペレーティングシステム研究会, May 2000.

[31] 岩井将行, 大越匡, 中澤仁, 徳田英幸. 家電ネットワークにおけるサービスの動的提供機構. 第59回情報処理学会全国大会, pp. 109–110. 情報処理学会, September 1999.

[32] 岩井将行, 中澤仁, 徳田英幸. 広域および小規模分散コンポーネント間通信機構の評価. マルチメディア通信と分散処理第105回研究会, pp. 19–24. 情報処理学会, November 2001.

付録 A

サンプルソースコード

A.1 EventSupplier サンプルコード

```
package drg.component;
import drg.core.component.EventSupplier;
import drg.core.component.SensorModule;
import drg.event.datatype.DToken;
public class SampleSupplierComponent{
  static int POLLING_TIME=100;//周期100ミリ秒でイベントを配信
  static int WAITING_TIME=20000;//起動時に20秒待機

   public static void main(String[] args) throws Exception{

      SensorModuleInnerImple sm = new SensorModuleInnerImple();
      new Thread(sm).start();//ダミーセンサを新たなThreadで起動

      String hostname= args[0];
      String name= args[0];
      String url="http://"+args[1];

      EventSupplier es = new EventSupplier(hostname,name,url,sm);
      es.initDefault();
      /*
      位置情報を記入する場合は
      initLocation("2F","s213","DeltaBuild")を利用．
      コンポーネントの詳しい説明，Version情報コピーライト等は
      initComment(String comment)を利用可能．
      */
      es.exec();//Supplierを起動し，LUSへの登録開始
   }

  static class SensorModuleInnerImple //センサモジュールの実装を行う．
    extends SensorModule implements Runnable{
      SensorModuleInnerImple(){super();}

    public void run() {
      DToken tokenEvent =new DToken();
      try {Thread.sleep(WAITING_TIME); }
      catch (InterruptedException ex) {}
      while (true) {
        try {
            Thread.sleep(POLLING_TIME);//一定周期待機
            tokenEvent.reverseToken();
            generateEvent(tokenEvent);
          }
        catch (InterruptedException ex) {System.exit(0);}
      }
    }
  }
}
```

A.2 EventMediator サンプルコード

```
package drg.component;

import drg.core.component.FilterObject;
import drg.core.component.EventMediator;

public class SampleMediatorComponent{
   public static void main(String[] args) throws Exception{
      FilterObjectInnerImple filter = new FilterObjectInnerImple();
      String hostname= args[0];
      String name= args[0];
      String url="http://"+args[1];
      EventMediator em = new EventMediator(hostname,"V0.9",url,filter);
      em.initDefault();
      em.exec();//ComponentManagementModule を起動
   }

  static  class FilterObjectInnerImple extends FilterObject{
    FilterObjectInnerImple(){super();}
    public int filterEvent(Object obj){
      //通常この場に何らかの処理を記述．
      return FilterObject.EVENT_PASS;//イベントを通過させる．
    }

    public int filterEvent2Action(Object obj){
      return FilterObject.EVENT_DENY;
    }

    public Object getEvent(){
      return "this is new string event";
    }

  }
}
```

A.3 EventConsumer サンプルコード

```
package drg.component;
import drg.core.component.ActionModule;
import drg.core.component.EventConsumer;
import drg.event.datatype.DToken;

public class SampleConsumerComponent{

   public static void main(String[] args) throws Exception{
      ActionModuleInnerImple am= new ActionModuleInnerImple();

      String hostname= args[0];
      String name= args[0];
      String url="http://"+args[1];

      EventConsumer ec = new EventConsumer(hostname,name,url,am);
      ec.initDefault();
      /*
      位置情報を記入する場合は
      initLocation("2F","s213","DeltaBuild")を利用．
      コンポーネントの詳しい説明，Version情報コピーライト等は
      initComment(String comment)を利用可能．
      */
      ec.exec();//ComponentManagementModuleを起動し，LUSへの登録開始
   }


  static  class ActionModuleInnerImple extends ActionModule{
    ActionModuleInnerImple(){super();}

    public void action(Object o){
      if(o instanceof DToken){//DTokenを継承したイベントかどうかのチェック
        DToken tokenevent =(DToken)o;//キャストを行う
        System.out.println(tokenevent.getToken());
      }
    }
  }
}
```

付録 B

Dragonの起動手順

DragonはJiniのLookupServiceの起動を前提としている．起動は以下の用に行いLookupServiceのオプションであるグループ名を*dragon*に指定する．

```
%rmid -log rmid/log &
%java -jar -Djava.security.policy=dragonRMI.policy \\
        reggie.jar  \\
        http://www.ht.sfc.keio.ac.jp/~tailor/reggie-dl.jar  \\
        dagonLUSPolicy.policy  \\
        reggie_log \\
        dragon  \\
```

以下に各コンポーネントの起動手順をしめす．生産者コンポーネントの起動．引数のyoshiwo/jini/bs/SS.gifは，GUI上に表示されるアイコン画像のURLの一部である．

```
%java -classpath lib/jini-core.jar: \\
                lib/sun-util.jar:  \\
                lib/jini-ext.jar:  \\
                dragonAll.jar  \\
     -Djava.security.policy=dragonPermission.policy \\
      -Djava.rmi.server.codebase=http://www.ht.sfc.keio.ac.jp/~tailor/core.jar  \\
      drg.component.SampleSupplierComponent \\
      SampleSupplierComponent \\
      yoshiwo/jini/bs/SS.gif \\

%This component is lunched at :RFID-tag0-822762982
%------ LUS discovery phase------
%Now I am looking for LUSs : group name=dragon
%LUS discovered...net.jini.discovery.LookupDiscovery@31aa65
%ServiceRegistration ==> com.sun.jini.reggie.Registration@4da8f4
```

仲介者コンポーネントの起動手順．仲介者コンポーネントがEOMおよびEIMがサービスとしてJiniに登録されるそのため，ServiceRegistrationが2回行わている．

```
%java -classpath lib/jini-core.jar: \\
                lib/sun-util.jar:  \\
                lib/jini-ext.jar:  \\
                dragonAll.jar  \\
     -Djava.security.policy=dragonPermission.policy \\
      -Djava.rmi.server.codebase=http://www.ht.sfc.keio.ac.jp/~tailor/core.jar  \\
      drg.component.SampleMediatorComponent \\
      SampleMediatorComponent \\
      yoshiwo/jini/bs/MS.gif \\

%------ LUS discovery phase------
%Now I am looking for LUSs : group name=dragon
%------ LUS discovery phase------
%Now I am looking for LUSs : group name=dragon
%LUS discovered...net.jini.discovery.LookupDiscovery@29f3b5
%LUS discovered...net.jini.discovery.LookupDiscovery@3f3045
%ServiceRegistration ==> com.sun.jini.reggie.Registration@7c4c51
%ServiceRegistration ==> com.sun.jini.reggie.Registration@680842
```

消費者コンポーネントの起動手順．

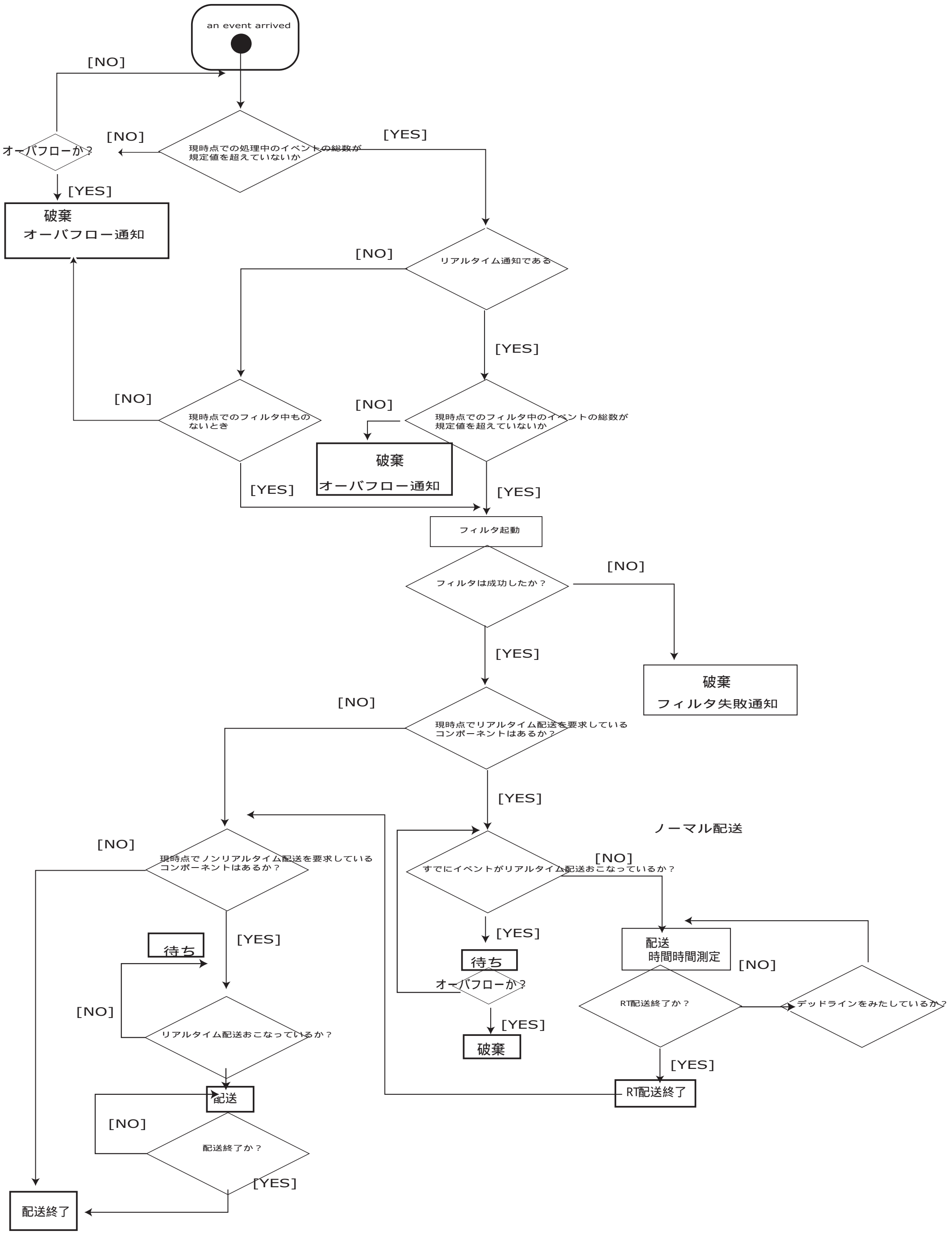
```
%java -classpath lib/jini-core.jar: \\
                lib/sun-util.jar:  \\
                lib/jini-ext.jar:  \\
                dragonAll.jar  \\
     -Djava.security.policy=dragonPermission.policy \\
      -Djava.rmi.server.codebase=http://www.ht.sfc.keio.ac.jp/~tailor/core.jar  \\
      drg.component.SampleConsumerComponent \\
      SampleConsumerComponent \\
      yoshiwo/jini/bs/CS.gif \\

%------ LUS discovery phase------
%Now I am looking for LUSs : group name=dragon
%LUS discovered...net.jini.discovery.LookupDiscovery@19f91c
%ServiceRegistration ==> com.sun.jini.reggie.Registration@4282a1
```

ComponentsBindingEditorの起動手順．これらのコンポーネントおよびComponentsBindingEditorの起動は，順不同でもよい．

```
%java -classpath lib/jini-core.jar: \\
                lib/sun-util.jar:  \\
                lib/jini-ext.jar:  \\
                dragonAll.jar  \\
     -Djava.security.policy=dragonPermission.policy \\
      -Djava.rmi.server.codebase=http://www.ht.sfc.keio.ac.jp/~tailor/core.jar  \\
      drg.gui.editor.ComponentsBindingEditor
```

付録 C

イベント入力から配送までのフローチャート


an event arrived
現時点での処理中のイベントの総数が規定値を超えていないか
[NO]
オーバフローか？
[NO]
[YES]
破棄
オーバフロー通知
[YES]
リアルタイム通知である
[NO]
[YES]
現時点でのフィルタ中もの ないとき
[NO]
[YES]
現時点でのフィルタ中のイベントの総数が規定値を超えていないか
[NO]
破棄
オーバフロー通知
[YES]
フィルタ起動
フィルタは成功したか？
[NO]
破棄
フィルタ失敗通知
[YES]
現時点でリアルタイム配送を要求しているコンポーネントはあるか？
[NO]
[YES]
ノーマル配送
現時点でノンリアルタイム配送を要求しているコンポーネントはあるか？
[NO]
[YES]
待ち
[NO]
リアルタイム配送おこなっているか？
配送
[NO]
配送終了か？
[YES]
配送終了
すでにイベントがリアルタイム配送おこなっているか？
[NO]
[YES]
待ち
オーバフローか？
[YES]
破棄
配送
時間時間測定
[NO]
RT配送終了か？
デッドラインをみたしているか？
[YES]
RT配送終了