修士論文　　2003年度

ユビキタスコンピューティング環境における分散コンポーネントの耐故障性に関する研究

指導教員

慶應義塾大学院 政策・メディア研究科

徳田 英幸

村井 純

高汐 一紀

慶應義塾大学院 政策・メディア研究科

柳原 正

修士論文要旨

ユビキタスコンピューティング環境における
分散コンポーネントの耐故障性に関する研究

本論文では、ユビキタスコンピューティング環境に適合した分散コンポーネント冗長化モデルの提案を第一の目的とする。また、ユビキタスコンピューティング環境への適合を実現するためのソフトウェア及びシステムの提供を第二の目的とする。

ユビキタスコンピューティング環境ではアプリケーションはコンポーネントネットワークによって構成されていると考えられる。利用者はネットワーク内のコンポーネントを集合させ、結合させることでアプリケーションを形成する。しかし、このような環境においては障害の検知と回避は困難と考えられる。例えばコンポーネント間の仕様による食い違いによってコンポーネントを組み合わせることができない場合や、ホスト障害によりコンポーネント自身が損失される場合等である。これらの障害によってアプリケーションの維持を行うことが困難であると考えられる。

本研究では、上述した問題点を解決するため、コンポーネントの複製・移送を可能にすることで障害の予防と回避するための機能を提供するフレームワークの構築を行う。本研究では、このフレームワークを実現するプロトタイプシステム FuSe を構築した。

本論文では、まずユビキタスコンピューティング環境におけるアプリケーション及びコンポーネントが抱える耐故障性における問題点を指摘し、問題点を解決するための手法を検証する。次に、コンポーネントの障害の検知・回避するための分散コンポーネント冗長化システムである FuSe を取り上げ、概要、設計及び実装方法を示す。最後に定量的及び定性的評価をもとに分散コンポーネント冗長化システムの有用性を検証する。

キーワード：
1 分散コンポーネント2 耐故障性3 オブジェクト移送
4 オブジェクト複製5 イベント配送

慶應義塾大学 政策・メディア研究科
柳原 正

Abstract of Master's Thesis

Researching Fault Tolerance Techniques for Distributed Components within a Ubiquitous Computing Environment

In this thesis, we will be focusing on two subjects; researching and developing fault tolerance techniques for distributed objects within a ubiquitous environment, and the offering of sample software implementations to realize the model.

Within a ubiquitous environment, applications are supposedly based on distributed components, where a user would combine components to create an application. However, it is expected that in such an environment would make fault detection and fault recovery difficult. For example, combination failure between components due to inconsistency of the components' specifications, or loss of an object because of host failure. Based on these factors, maintaining services with such components would be difficult.

We proposes a framework where components obtain fault tolerance by allowing component duplication and transportation.and have implemented a prototype based on this framework called "FuSe".

First, we will be looking at application and components for ubiquitous environments and pointing out their weaknesses, and verify techniques to solve such problems. Next, we will explain the summary, desigh, and implementation of Fuse. Finally, we will prove the possiblities a distributed component redundant system has by looking at the qualitative and quantative analisys of FuSe itself.

Keywords:
1 Distributed Component 2 Fault Tolerance 3 Object Transportation
4 Object Relication 5 Event Delivery

Tadashi Yanagihara

Faculty of Media and Governence
Keio University

目 次

図目次

表目次

第1章　はじめに

1.1　本研究の背景と問題意識

近年のコンピューティング環境はユビキタス化へ向かっている。ユビキタスには主にAnywhere Computing、Everywhere Computing という意味を持つが、Ubiquitous Computing [1] の場合では Anywhere Computing、Anytime Computing、Anyway Computingへと意味合いが代わる。具体的に、いかなる場所で常時コンピューティング環境が利用でき、さらに Anyway、望む通りのサービスに抽象化されたコンピューティングを受けることが可能となる。

例えば、AT&T ベル研究所で始まった Active Badge Location System [2] はこの分野の開拓的存在とも言える。いつでもどこでも、赤外線バッヂでユーザの位置を追うことができ、これにより自動コーヒーメーカーや電子会議システムと言った、その場所で有効な位置依存なサービスを受けることが可能である。

また、旧来のアプリケーションのように、開発時にアプリケーションの利用用途を想定して開発する形から、実行時に振る舞いを決定するアプリケーションの利用用途への発展が予想できる。アプリケーションをコンポーネント化し、実行時にユーザの手によって再度構築可能である。

これにより、新たな問題が新たに二つ発生する。

一つは障害が発生した際の対処は開発者ではなく、利用者が行わなければならない。特にアプリケーションを構成するコンポーネントはネットワーク上で分散されていることが考えられるため、特にコンポーネント間の仕様による食い違いから発生するエラー、実行中に OS がクラッシュする、ハードウェア自体の移動によるネットワークの切断をはじめとした障害は自動的に障害検出を行うための集中管理は困難と考えられる。

二つ目の問題として、コンポーネントの障害を検出した場合、極力利用者に障害発生を意識されること無く、デバッグ自体は開発者が行えるとしても、必ずしも利用者が行えると限らないと考えられる。しかし、アプリケーションの振る舞いを最終的に決定する際は利用者の手によって行われるため、開発者がデバッグを行うことができない。システム側で障害検出した際、極力システム側で障害回避及び障害予防を行い、最終的に利用者へ障害を意識させないことが目的である。

1.2　本研究の目的と概要

本研究において、今後増えると考えられる分散コンポーネントを基にしたコンポジットサービスの障害検出をトリガーとし、そのトリガーの内容をコンポーネントの冗長化という形でシステム全体の耐故障性を向上させる手法とそれを具体化したシステムを示すことを目標とする。

ユーザは本システムを用いることで分散コンポーネントの冗長化を計ることが可能となる。手法は複製と移送という抽象化を行うことで、他ホストへオブジェクトを委託することで障害が発生した際にコンポーネント及びトランザクションの永続化が計れる。これにより、今までユーザが特別に意識を持たさずに、障害を事前に予防でき、さらに万が一障害が発生した場合、障害を回避することが可能となる。

これにより、次のような既存システムのデメリットを解消し、提案するシステムのメリットを伸ばすことが可能である。図 1.1 を例として取り上げる。

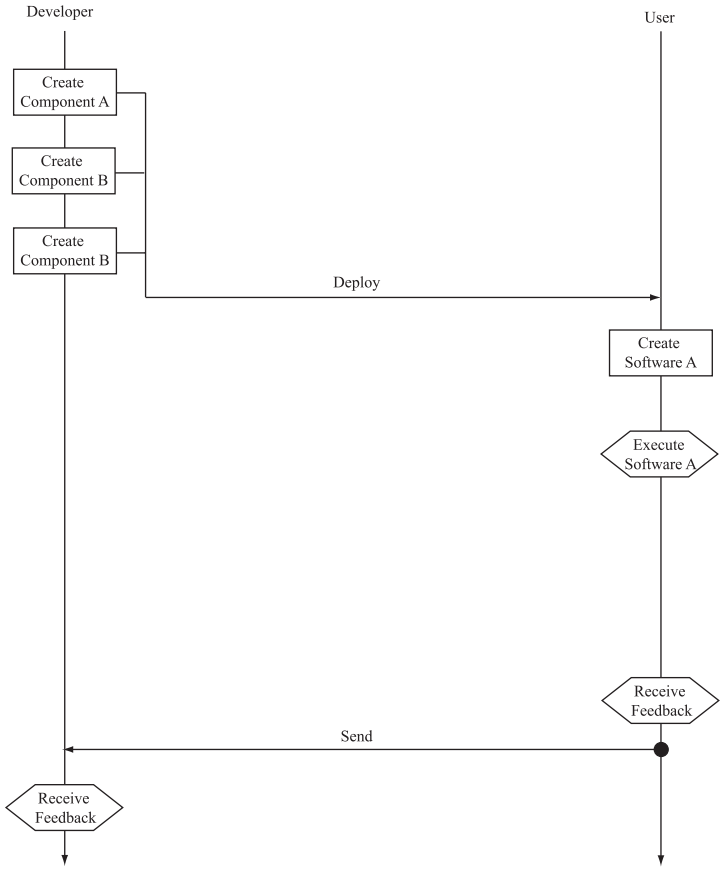

図 1.1: 既存コンポーネントベースシステム開発の流れ

既存のコンポーネントベースシステムにおける開発方式を提示する。アプリケーション A の基となるコンポーネント A、B、C を開発者が作成し、利用者へ配布する。利用者はコンポーネント A、B、C を組み合わせ、アプリケーション A を構築する。アプ

リケーションは何かしらフィードバック (正常終了、エラー、デバッグメッセージ等) を得、それ自体に問題があれば開発者へフィードバックが送られる。

このシステムは多くのソフトウェアモデルとして現在使われているが、このモデルの欠点として次の三点が挙げられる。

- コンポーネントはネットワーク上に分散して配布されることが考えられるため、あとからコンポーネントの発見を行うのが困難となる。
- フィードバックは分散して送られるため、デバッグのように集中的にエラーを管理したいときに難しい。
- フィードバックは利用者から直接開発者へ送り返されるため、何かエラー対処を行うための補助ソフトウェアを介入させられない。

そこで今回提案するシステムを間に仲介させることで今までの開発の流れは次のように変更できる。

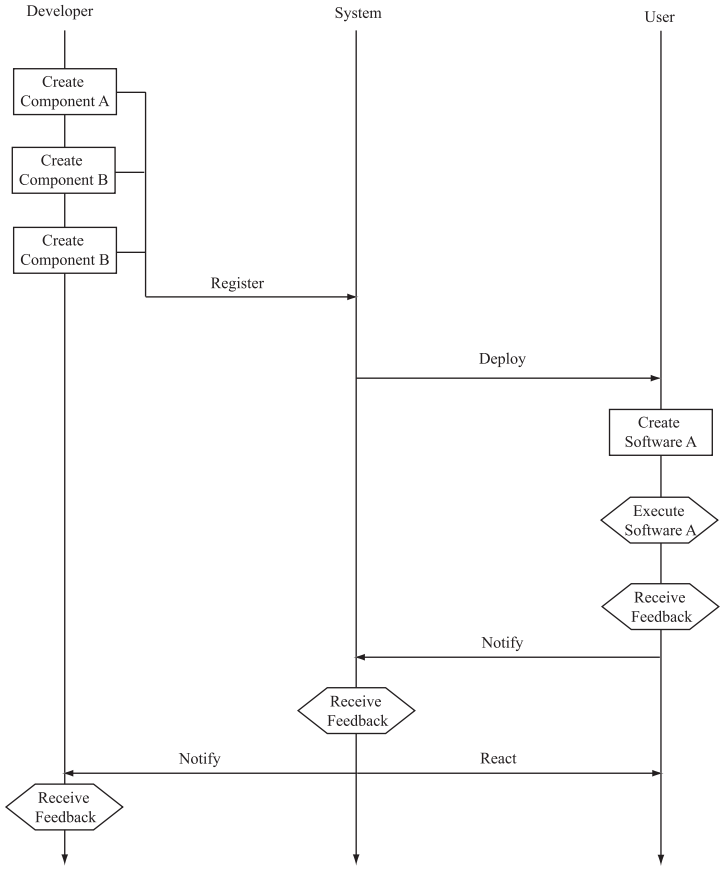

図 1.2: 提案するコンポーネント開発の流れ

ここではシステムは開発者と利用者の間に仲介を行い、分散コンポーネントを管理する。これにより、コンポーネントは仲介するシステムへ登録されるため、あとから

のコンポーネントの発見を行うのが容易となる。さらに、ソフトウェアのフィードバックも分散されたものがシステムへ送られ、集中的に管理されるため、デバッグはより容易となる。そして、フィードバックを開発者へ送り返しつつ、意図的にフィードバックに対して特定の反応を行うことができる。本システムでは冗長化、つまりコンポーネントの複製と移動が行えるようになる。

このように、ソフトウェアの配布とフィードバックを集中的に行う仲介者を投入することで、開発者はフィードバックを得つつ、特定のフィードバックへ補助的に反応しアクションを起こすことができるようになる。よって、利用者はデバッグ及びエラー処理などを行う必要性がなくなる。

1.3 本論文の構成

本論文は、以下の構成によって成る。

第 2 章では、既存の分散コンポーネントの故障に対する問題点を把握し、これを解決する分散コンポーネント冗長化モデルの考察を行い、適格なシステムモデルを提案する。

第 3 章では、第 2 章で述べた分散コンポーネント冗長化モデルの機能要件を述べ、その実装例である FuSe とその特徴について述べる。

第 4 章では、FuSe の設計方針、全体構成、各サブシステムの設計について述べる。

第 5 章では、第 2 章で述べた分散コンポーネント冗長化モデルのプロトタイプである FuSe の実装環境、各サブシステムの実装の詳細について述べる。また、FuSe を用いた応用例についても触れる。

第 6 章では、実装した FuSe のプロトタイプシステムの定量的評価と評価結果の考察、既存システムとの性能比較を行う。

第 7 章では、全体としての結論について述べ、また今後の課題を明らかにし、本論文を締めくくる。

第2章　ユビキタスコンピューティング環境における障害

2.1　用語定義

本論文において、今後いくつか想定するシステムや環境に依存した特殊な用語を使う。これらの用語は今後システムの説明を行う上で必要不可欠であるため、ここでそれらの言葉の定義を行った上で、今後の議論を進めることとする。

コンポーネント

コンポーネントとは分散アプリケーションを構成する最小粒度のソフトウェア部品のことを指す。一つの大きなアプリケーションの断片であると同時に、他コンポーネントとの連携で新たなアプリケーションを構築することを考慮して再利用性が高いソフトウェアのことを指すこともある。粒度が小さいためソフトウェアの規模は小さく、実装環境や言語に極力依存しないように記述されるため、携帯性に優れる。このため、コンポーネントはネットワーク上に分散して存在するものとも考えることができる。

また、コンポーネントには入出力を行う**ポート**という概念が存在する。ポートはコンポーネント間とメッセージをやり取りする際に利用する外部への接続口である。ポートには型があり、基本的に他コンポーネントのポートやそのポート間でやりとりされるデータの型が互換なものしか通信が行えないものと考えられる。型を持たない void 型のポートもあり、メッセージを非同期で送信し、行動を得ることができる。このようにコンポーネント間のポートへメッセージを送ることで得られる行為のことを**サービス**と呼び、これらコンポーネントを組み合わせて構築したアプリケーションが提供する機能を**コンポジットサービス**と呼ぶ。

このコンポジットサービスこそはユーザが利用するアプリケーションそのものであり、そのアプリケーションを利用することで得られるサービスはコンポジットサービス内のポート間を通してやり取りされたデータによって引き起こされる行動でもある。以下の図 2.1 に概念図を示す。

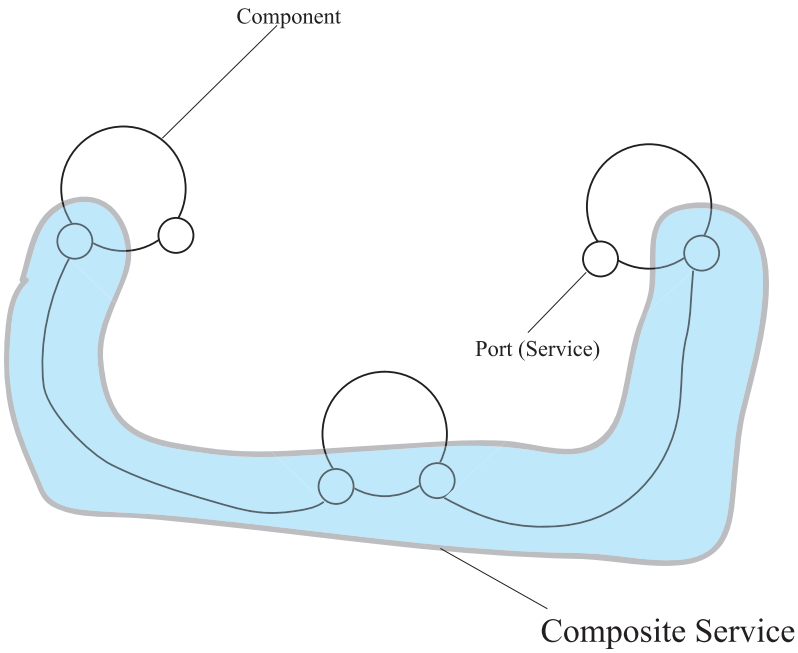

図 2.1: コンポーネントの概念

移送と複製

特定のホスト上に存在するコンポーネントを他ホストへ移す際、元々参照されていたコンポーネントをその後参照先のコンポーネントと同期を取る場合は**複製**、参照を破棄しコンポーネント間の同期をしなくなった場合は**移送**と定義する。本システムでは耐障害向上の手法として、サービスの移送と複製をシステムから提供する。これはコンポーネントの機能を他ホストへ複製及び移送することで、コンポジットサービスを構成する元のコンポーネントに障害が発生した場合、移送及び複製したコンポーネントと置き換えることでサービスの継続が保つことができる。以下の図 2.2 に概念図を示す。

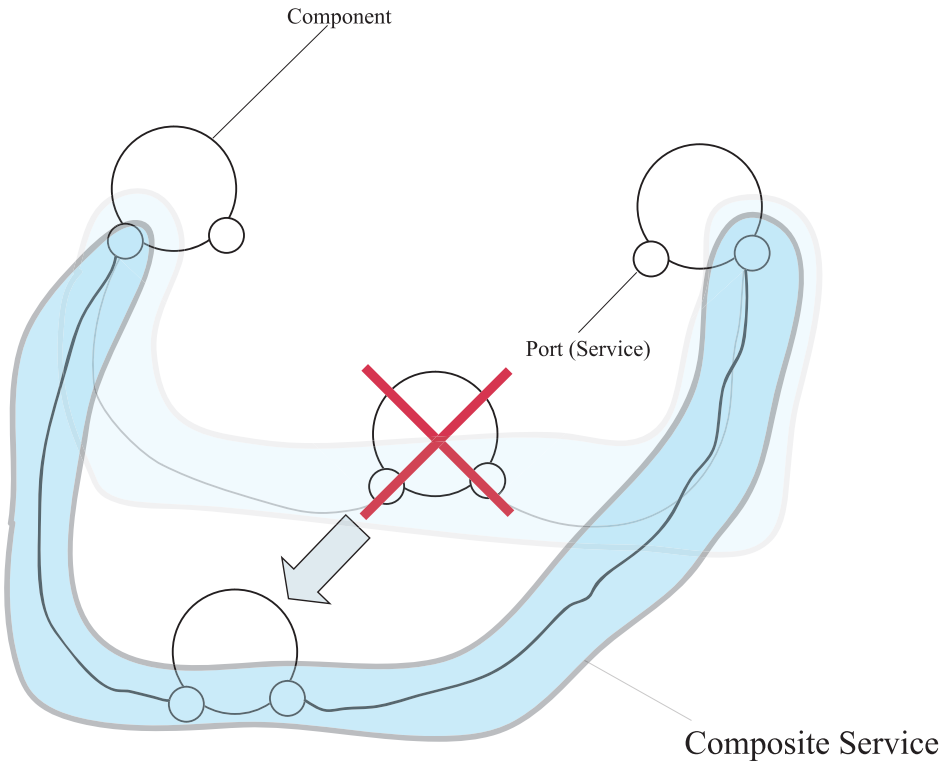

図 2.2: コンポーネント移送・複製の概念

ユビキタスコンピューティング環境

Ubiquitous には主に Anywhere、Everywhere という意味を持つが、ユビキタスコンピューティングの場合では Anywhere Computing、Anytime Computing、Anyway Computing へと意味合いが代わる。具体的に、いかなる場所で常時コンピューティング環境が利用でき、さらに Anyway、望む通りのサービスに抽象化されたコンピューティングを受けられる環境を**ユビキタスコンピューティング環境**と言う。

例えば、AT&T ベル研究所で始まった Active Badge System はこの分野の開拓的存在とも言える。いつでもどこでも、赤外線バッヂでユーザの位置を追うことができ、これにより自動コーヒーメーカーや電子会議システムと言った、その場所で有効な位置依存なサービスを受けることが可能である。

その発展系として、ハードウェアは人の目に着かないよう、壁や天井に取り付けられたり、小型化されていたりする。ソフトウェアはより小型な機械で動くように作られ、さらに実行時に振る舞いを決定するものである。ネットワークは既存の有線インフラに踏まえ、802.11g/a や bluetooth などのようなヘテロジニアスなネットワークから融合されているものが挙げられる。以下に図 2.3 を概念図として提示する。

こういったサービスを自由自在な形で受けられることがユビキタスコンピューティングの最大の特徴である。さらに細かく特徴を挙げると次のようにまとめられる。

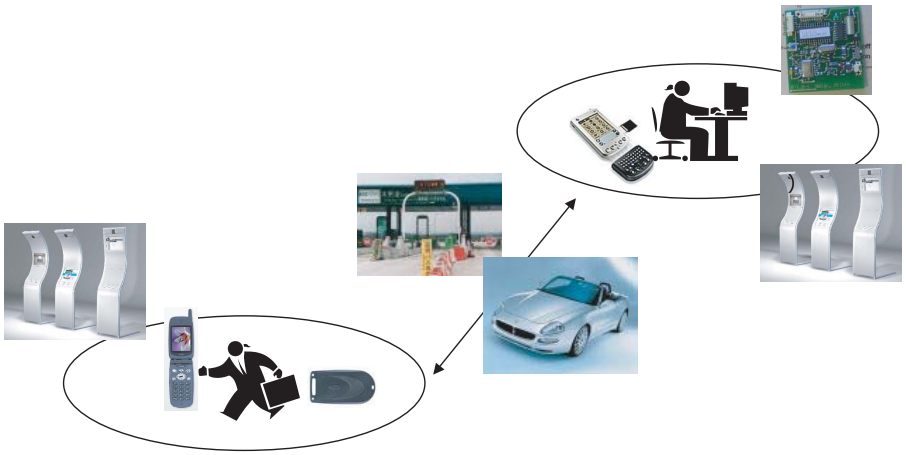

図 2.3: ユビキタスコンピューティング環境の予想図

エンドユーザプログラミング

ユビキタスコンピューティングは今までのコンピューティングと異なる点として、エンドユーザプログラミングの概念という点が挙げられる。エンドユーザプログラミングとは、プログラムは従来のように形成時に静的な振る舞いを決定するのではなく、実行時に動的な振る舞いを決定する特徴を持つ。例えば、テレビのリモコンというサービスの例を取ると、何チャンネルのボタンがついているのか、ビデオや BS との連携を想定したボタンがついているかなどの機能を出荷時に決定してしまっているが、これが動的である場合、ユーザはリモコンにどのような機能をつけるかを実際利用するときに決定し、生成されることと考えられる。

また、動的性を強調するため、サービスは生成の元となるコンポーネントを組み合わさった分散オブジェクトから成り立つコンポジットサービスであることをもう一つの特徴であることが述べられる。

このように、エンドユーザへのサービスは細かいコンポーネントを元とした分散オブジェクトの集合体から成り、利用される際にコンポジットサービスとして提供されるものである、と定義できる。

これにより、ユビキタスコンピューティングの特徴である Anyway Computing が実現される。

ネットワークの広域化

ユビキタス化におけるキー要素としてネットワークの広域化が挙げられる。近年のエンドユーザは高速な ADSL モデム [3]、さらに高速な光ファイバ [4] や無線ネットワーク [5] によって形成される。これらのネットワークの特徴として、高速であり、さらに多様であると同時に、常時接続な状態であるという点が挙げられる。常時接続であるため、これらのネットワークを利用するユーザはいかなる状況や場所にいようと

必ずネットワークに接続が可能となっている。

メディアの大容量化

近年のエンドユーザが手にするメディアは大容量化していく傾向にある。通常利用するハードディスクは 127GB の壁を越え、今なお増え続けるように見える。また、よりアドホックな用途で利用されるリムーバブルメディアであるコンパクトフラッシュやスマートメディアは 1GB クラスのものが出現し、今後は大容量化のスピードが増していくと考えら得れる。

同様に、これらのメディアをネットワークに接続するサービスも現れ始めている。例えばオンラインストレージサービスや NAS など、大容量なメディアをオンライン化し、あたかもそのメディアはローカルディスク上にあると見せかけるサービスが挙げられる。

これにより、ユビキタスコンピューティングでは上述した Anywhere Network と同様、Anywhere Media という機能も実現され、これら二つを合わせて Anywhere Computing と呼ぶことができる。

2.2 ユビキタスコンピューティング環境を構成する要素

ユビキタスコンピューティング環境でサービスを受ける際、そのサービスを提供するアプリケーションはコンポーネントベースで構成されていることが予想される。これは、アプリケーションは実行時に自由に構成を変更できるという機能用件に対し、旧来の静的に記述されたアプリケーションでは柔軟性と拡張性を持ったアプリケーションを作成することが困難であると考えられるからである。

コンポーネントは環境内に遍在している上、分散していることが考えられる。これは例えば部屋の機器で利用可能な機能を組み合わせ、最終的なアプリケーションを作成するが、このときは各機器はお互いを意識しない状態で、利用されるときに初めてお互いを意識し合うことが考えられる。これは実行されるまではコンポーネント同士は分散していると考えられる。

以上により、ユビキタスコンピューティング環境は分散されたコンポーネントを元に構成されていると考えられると言える。

2.3 ユビキタスコンピューティング環境で予想される障害

ユビキタス環境における障害は次のように分類することができる：ハードウェアにおける障害、ソフトウェアにおける障害、ネットワークにおける障害の三通りである。これより、これらの障害の特徴の分析とケーススタディを行う。

ハードウェアにおける障害

ハードウェア障害は検出が最も容易であり、ホスト上の物理的損失が発生した場合を指す。検出はユーザが行うだけでなく、OS や監視ツールを用いることで検出も可能である。このケースは例えば、身近なところではハードディスクの内部故障によるメディアの故障が挙げられる。この症状が発した場合、急遽メディア内のデータを他所へ移す必要が発生する上、メディアを交換後にアプリケーションを再度上げ直さなければならない。

ソフトウェアにおける障害

ソフトウェア障害の検出はハードウェアに比べ、問題個所を特定するのは少し困難になるが、比較的容易である。ハードウェアよりも障害検出が困難である理由として、ソフトウェアの障害はソフトウェア内の処理を監視するプロセスでのみ検出ができないためである。典型的な例として、ソフトウェアが暴走し、ゾンビプロセスになってしまったケースが挙げられる。この症状が発生した場合、多くのメモリや CPU パワーが無駄に消費されるため、ユーザサイドで対処を行う以外の方法として、プロセスを監視するソフトウェアを容易することで検出が可能となる。

ネットワークにおける障害

ネットワーク障害はハードウェア及びソフトウェア障害に比べ特殊な性質を持っているため、検出は困難であると言える。ネットワーク障害として完全に他ホストと通信ができないならば検出は容易であるが、障害として他にパケットロスや遅延の増長などと言った、通信は不可能ではないが、利用しにくくはなっているという状態が発生すると検出は困難となる。このようなネットワーク障害が発生した際には、ソフトウェア及びハードウェア的に検出ができないので、ユーザの判断のみに障害の発生の検出を行わなければならない。

2.4　障害回避手段の分析

障害が発生した際、どのようにして回避ができるかについて、より具体的な分析をここで行う。これにより、このような障害が発生した場合の回避策についての分析を行う。

ハードウェア障害における予防策・回避策

アプリケーションを実行するホストのハードウェア自体が故障したことを指す。例えば CPU、メモリの異常消費やハードディスクプラッターの破損などが挙げられる。

検出は容易である代わりに、検出機構自体がOSや言語に深く依存してしまうため、検出自体は開発者がコンポーネントと一緒に作成することを前提とする。

また、ハードウェアの故障は物理的な破損で修復が困難であるため、現在利用しているデータを他ホストへ移動したり、複製することを予防策・回避策とする。本システムでは指定したコンポーネント(オブジェクト)とそれらを利用するデータを他ホストへ移送・複製する機構を用意する。

ソフトウェア障害における予防策・回避策

ソフトウェア障害も検出が比較的である。開発する言語に例外処理を行うためのAPIが用意されているからである。本システムではこのAPIを使って記述したアプリケーションがエラーなどのフィードバックを特定のノードへ投げる機能を予防策として提供する。また、ソフトウェア的故障への回避対策として、コンポーネント自体の複製を行い、コンポーネントがエラーなどの処理を行った際、あらかじめ複製しておいたコンポーネントと複製を行う機能を提供する。

ネットワーク障害における予防策・回避策

ネットワーク障害の検出は他二種類の障害に比べ、検出が困難である。これは完全にネットワークから切断されている場合なら検出は容易だが、例えばパケットロスや配送の遅延から障害と判断するにはユーザ自身が判断を行わなければならない。本システムではネットワークの障害により特定のコンポーネントへのアクセスが不可能になる際、複製しておいた同コンポーネントへ切り替えることを対処方法とする。

2.5 分析結果

ユビキタスコンピューティング環境におけるアプリケーションは複数のコンポーネントによって構成されている。このようなコンポーネントベースな環境における障害として、ハードウェア・ソフトウェア・ネットワークの障害が考えられる。これらの障害への対策として、コンポーネントをあらかじめ複製しておいたり、障害が発生した際に移動させたりすることで障害回避が可能となる。

以上を踏まえ、障害検知及び回避能力を備えたソフトウェアを提供することで、既存のコンポーネントに障害回避能力を得ることができると考えられる。特に課題となるのは、コンポーネントネットワークのような分散環境において障害の検知と管理が困難であるため、これらをコンポーネント上の機能として実現することが困難であると考えられる。これらの機能を踏まえ、以降採用モデルとシステム設計を行う。

2.6　本章のまとめ

本章を通して、一章で述べた問題として障害が発生した際に自動的に障害検出を行うために集中管理を行うことが困難であるという点、また、コンポーネントの障害を検出した際にその障害回避及び障害予防をコンポーネント側で実現するのが困難であるという点を踏まえ、それらの問題点を解消するための補助システムを導入することが望ましいことについて述べた。その機能用件として、コンポーネントの複製・移送を行い、障害が発生した際にこれらのコンポーネントと切り替えを行いながらコンポジットサービスを維持し続けることが挙げられる。次章ではこれらの機能用件を満たす分散コンポーネント冗長化システムの設計に付いて述べる。

第3章　分散コンポーネント冗長化システムモデル

3.1　分散コンポーネントの冗長化とは

前章ではユビキタスコンピューティング環境における障害の検証と分析を行い、障害を検出し、それらの障害に応じてアプリケーションを構成するコンポーネントを複製・移送することで障害の予防及び回避が行えることを実証した。コンポーネントの永続化を計ることで利用者はアプリケーションの形成を維持したまま、サービスを利用し続けることが可能になるため、有用であると考えられる。

本システムを構築する上で、上述したコンポーネントの複製・移送を実現により耐故障性を向上させることで冗長化を計る手法を踏まえ、開発者へ既存のコンポーネントベースアプリケーションの開発へ障害の検出と回避機能を追加し、さらにユーザへは障害の発生を意識させないことを考慮した**分**散コンポーネント冗長化システムというモデルを提唱する。分散コンポーネント冗長化システムではコンポーネントの複製及び移送、またコンポーネントの登録と検索を機能として持ち、これらによって柔軟かつ冗長なアプリケーションを構築することを可能にする。

3.2　分散コンポーネント冗長化システム：FuSe

本研究において分散コンポーネント冗長化システムのプロトタイプとしてFuSe (Fail-safe ubiquitous Service enhancement) を実装した。FuSeはコンポーネントに冗長化の機能を拡張することで耐故障性を向上させることを目的としたシステムである。

FuSeには前章で述べたハードウェア、ソフトウェア、ネットワークにおける障害検知機能、分散コンポーネントの複製及び移送の機能を特徴に持つシステムである。これらの機能を実現するにあたり、以下で述べる注意点に注意しながら設計を行う。

コンポーネントとホストとの束縛

本システムが提供するコンポーネントの複製・移送を実現するために、サービス自体はホスト非依存なものであることが望ましい。例えばビデオエンコーダのように特定のデバイスに依存したサービスはホストと切り放すことが難しいため、デバイスを利用する特定のコンポーネントを複製・移送することが困難となる。

本システムでは特定のデバイスに依存したアプリケーションへのリンクという概念

を導入することで部分的にデバイスとアプリケーションの依存性を切り放す機能を提供する。具体的に、コンポーネントを参照する仮想コンポーネントを作成し、その仮想コンポーネントによってアプリケーションを作成することで、仮想コンポーネントのホストへの依存性を切り放すことが可能となる。仮想コンポーネント自身はホストへの依存性を持たないため、障害が発生した際に他ホストへの複製・移送が可能となる。

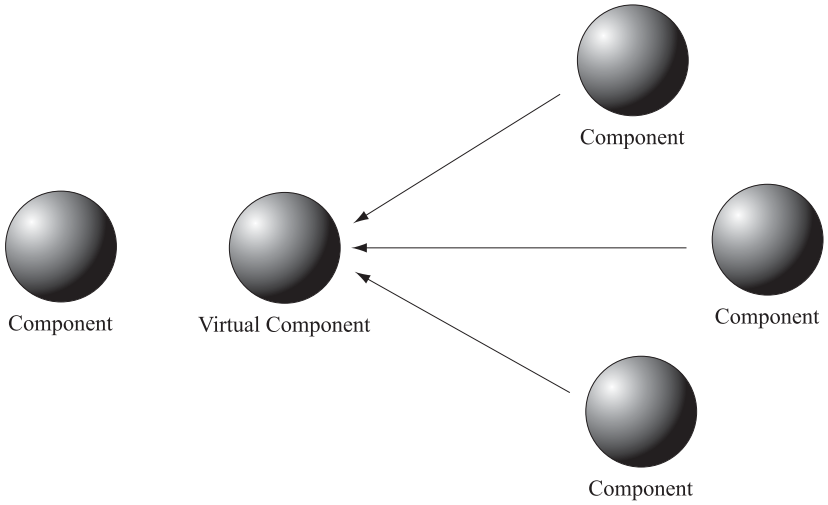

図 3.1: コンポーネントと仮想コンポーネントの関係図

分散コンポーネント参照先の特定

分散コンポーネントを複製・移送した際、その参照を保てなければならない。具体的には、特定のコンポーネントに障害が発生した際、元のコンポーネントから移送・複製した先のコンポーネントへの参照先切替を自動的に行うようにしなければならない。また、これはアプリケーションを利用しているユーザから状況に応じてインタラクティブに切り替えを行ったり、意識されずに自動的に切り替えを行ったりしなければならない。

本システムでは前述した仮想コンポーネントを利用することでこの問題を解決する。これはコンポーネントは他コンポーネントを参照する際、そのコンポーネントの仮想コンポーネントを参照するようにする。これにより、例え仮想コンポーネントが指すコンポーネントが切り替わったとしても、他コンポーネントが参照するコンポーネントは仮想コンポーネントのままであるので、結果的にコンポーネントを参照し続けることができる。一方、参照先が変更されるのがサービスの中核となる FuSe Core である。FuSe Core はサービスを構成するコンポーネントの参照先を管理し、変更があった場合に参照先を変更を行う。コンポーネントが参照を要求してきた際に現在参照している該当コンポーネントの参照先を返すような仕組みからなる。

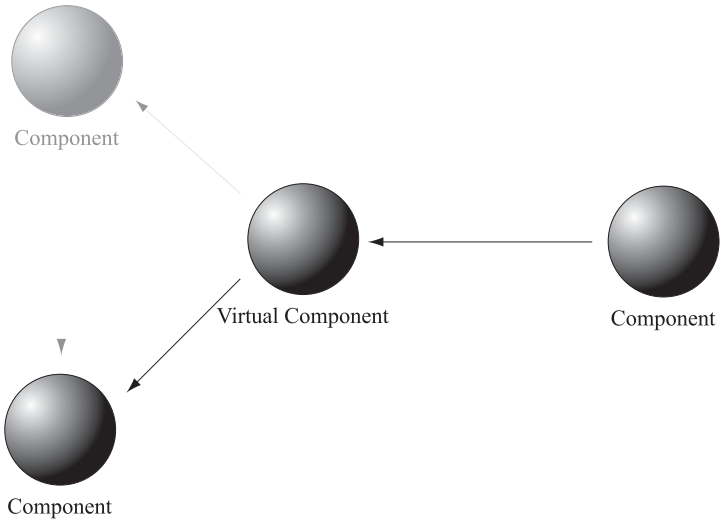

図 3.2: コンポーネントと仮想コンポーネントの参照先の関係図

分散コンポーネントの状況別行動

障害の自動検知及び自動回避を実現するために、分散コンポーネントの複製・移送による切り替えを行う際、その行動のきっかけとなるトリガ及びトリガによって引き起こされる行動をリンクしセットとして扱えるようにしなければならない。

しかし、本システムでトリガとなる障害の検知を行う際、通常にプログラムを記述する際にシステム依存的な実装となってしまう。このため、状況毎別の検知用に記述を必要とし、汎用性に欠ける。コンポーネントはあらかじめ再利用性が高いよう、言語、OS、あるいはコンポーネントとの間で汎用性を高めることで、記述したコンポーネントの再利用性が高まり、アプリケーションを再構築する必要性が減る。

汎用的な障害検知を得るため、本論文内で取り扱う障害は JavaVM によって検出されるものとする。具体的にはソフトウェアレベルで型の不一致やアプリケーションが排出するエラー、ネットワークレベルでは他コンポーネントとの通信が行えなくなった際のエラーを検知する範囲と定める。これにより、コンポーネント開発者はあらかじめエラーの定義及び行動をセットすることが可能となる。

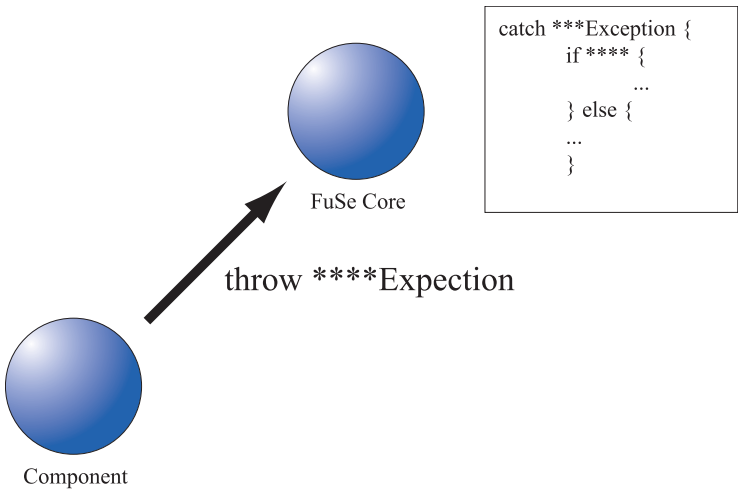

図 3.3: 障害の検知と行動のリンクの関係図

3.3 本システムで採用する概念モデル

本システムを構築する上で化学の分子構造を採用する。分子は原子が集まり形成されるもであり、自由電子を他分子間を送り合うことで安定な状態を保つ。また、状態によって不安定な状態から安定した状態へと転移しようと結合された原子を放出したり、他分子へ渡したりする。これはコンポーネントによって形成されたコンポジットサービスが障害などの要素により構造上不安定になった際、コンポーネントを他ホストへ移送する様子と非常に似る。また、自由電子をメッセージに例えるとコンポーネント間の通信と似ている。

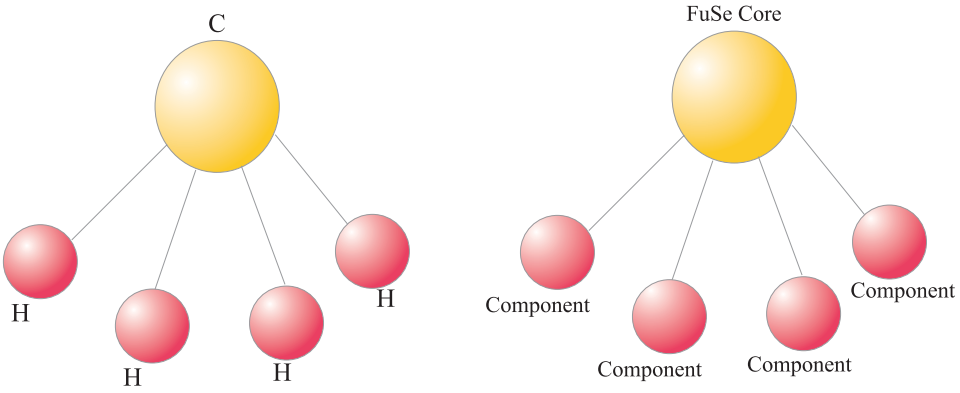

図 3.4: 分子構造とコンポジットサービスの構造の比較

よって、今後システムアーキテクチャは分子構造を参考にしながら構築する。

3.4　全体のシステム図

以上で述べた点をまとめると、システムは以下のように説明することができる。

- アプリケーションの構造
  - アプリケーションは FuSe Core 及びコンポーネントから成る。
  - FuSe Core にはコンポーネントを代表する仮想コンポーネントが生成される。
  - コンポーネントから他コンポーネントを参照する際、この仮想コンポーネントを参照する。
- 複製と移送
  - コンポーネントの複製及び移送を行う際に FuSe Core へ通知される。
  - 複製及び移送を行った際、仮想コンポーネントの参照先の切り替えを行う。
  - これにより、他コンポーネントからの参照先変更を強いることなく行える。
- 障害検知
  - エラー処理は Java の Exception を利用し、FuSe Core によって catch される。
  - catch されたエラーは FuSe Core 内で処理され、コンポーネント開発者もしくユーザによって設定された行動とリンクされる。
  - エラー検知をトリガとして、あらかじめセットされた行動が発動する。

最後にこれらを踏まえた上でのシステム動作図を図 3.5 として提示する。

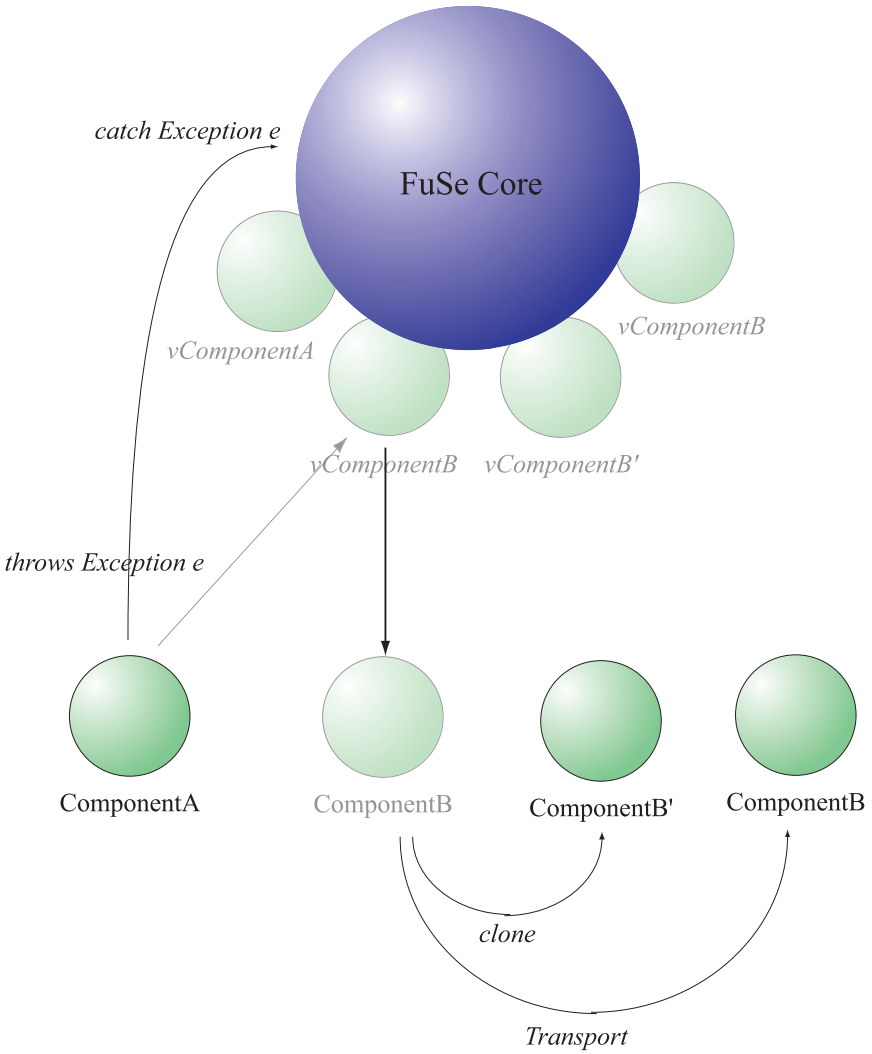

図 3.5: FuSe モデルのシステム全体図

3.5 本章のまとめ

本章では分散コンポーネント冗長化システム FuSe が提供する機能とそれらを実現するために必要とする機能要件について述べた。FuSe は様々なコンポーネントの混在を許可したシステムであるが、上述したように、他にコンポーネント自身の複製及び移送が特徴である。また、コンポーネントの障害検知と行動の設定が行え、検知と行動と結び付けることで障害回避を容易としている。

次章では FuSe を構築するための設計について述べる。

第4章　FuSeの設計

前章では分散コンポーネント冗長化システム FuSe が提供する機能とそれらを実現するために必要とする機能用件について述べた。機能用件はが必要である。本章ではこれらの機能用件のソフトウェアレベルで実現するための設計について触れる。

4.1　設計方針

本研究の目的は分散コンポーネントの複製・移送を行うことで冗長化が行えるシステムを提供することである。このため、二章で述べたように、分散コンポーネント冗長化システムの特徴であるコンポーネントの複製・移送を実現する。これらの機能をシステム内に組み込むために、以下に述べる設計方針を定める。

4.1.1　分散コンポーネント複製機構

非同期的通信

複製はコンポーネントをクローンすることで実現される。クローンを行う際、そのオブジェクトの状態を一度ロックし、コピーを行ってからロックを解除することが望ましいが、本システムの目的であるサービスの継続性が必要であるため、ロックを行うことで他オブジェクトからの参照を止めるとサービスの継続性は止められることになる。このため、本システムではスナップショットを非同期に採取し、障害が発生した際に先ほどのスナップショットと参照の切り替えを行う。

4.1.2　分散コンポーネント移送機構

同期的通信

複製と異なり、移送を行う際にファイルの読み書きをロックし、状態を保存して転送を行わなければならない。これは複製は読み取りでコピーを作成しているのに対し、移送は動作中のオブジェクトをホスト間に移送させるため、一度コンポーネントの状態を止める必要があるからである。

本システムでは Java 言語が持つシリアライゼーションを用いたオブジェクト転送機能を利用することで状態保存機能を提供する。Java 言語の場合、クラスからオブジェクトを生成し、シリアライズすることで転送などを行った際、転送先でオブジェクト

の状態が保存されたままで復元される。ファイルの同期はシリアライズ機能自身が行うため、移送を行うためのコードを記述する際、同期について考える必要性はない。

4.1.3 システム全体の設計方針

容易性

FuSe はコンポーネント開発者と専門性の高い知識を持たない一般ユーザのどちらからの利用を想定している。しかし、どちらからでも容易に扱えるようにするには困難である。例えばコンポーネント開発者は既存のコンポーネントにエラー対策用のコードを追加しないようにしなければならない上、利用者は障害を意識させることなく利用してもらう必要がある。さらにシステム自身も極力障害が発生しにくいように設計をしなければならない。このため、システム全体は構造的にシンプルにし、ユーザも開発者も全体構造を理解しやすいようにしておくことでシステムを使いやすいものにする。

4.2 本システムのソフトウェア構成

本システムは大きく分けて三つの部分から成る。コンポーネントの登録を行い、さらに Life Time を設定することで timeout による障害を検知するディレクトリ部、コンポーネントの参照と複製・移送を行うマネージメント部、そして例外処理と行動の取り扱いとリンクを行うイベント部である。さらにコンポーネントに追加する例外処理用のトリガとなるコードを入れると正式に四層から成る。以降、それぞれの層の関係について述べる。

ディレクトリ部ではサービスとしてコンポーネントの登録及び検索を行うための Directory Service とコンポーネント (及び、最終的にアプリケーション全体) の状態を監視する Life Keeper から成る。Directory Service は先ほど述べた通り、サービスに加わるコンポーネントの名前、IP 及び Port 番号が登録される。他コンポーネントは名前を渡すだけで参照先が分かる仕組みとなっている。Life Keeper はコンポーネント自身が自分の状態が管理できないことを前提に、コンポーネントの生存を確認するための機構である。具体的にコンポーネントが移送したり、ガベージコレクションによって回収され存在がなくなってしまったり、あるいは単にソフトウェア的故障を突然起こし、コンポーネントに設定された Exception Event を発行することができないことが発生したりしたときに備え、コンポーネントを一定期間事に生存を確認し、異常があった場合にコンポーネントに障害が発生したと判断する。

マネージメント部は主にコンポーネントの複製・移送を行う部分であり、また同時にコンポーネントの参照先及び仮想コンポーネントの格納場所でもある。複製・移送するイベントがイベント部から届く際、Component Replicator 及び Component Transporter、そのとき該当する機構を呼び出し、コンポーネントの複製・移送を行う。移送・複製され

たことがComponent Reference Managerに通知される。Component ManagerはDirectory Serviceによって登録されたコンポーネントの仮想オブジェクトを生成し、これはまたComponent Reference Managerに参照先を登録される。

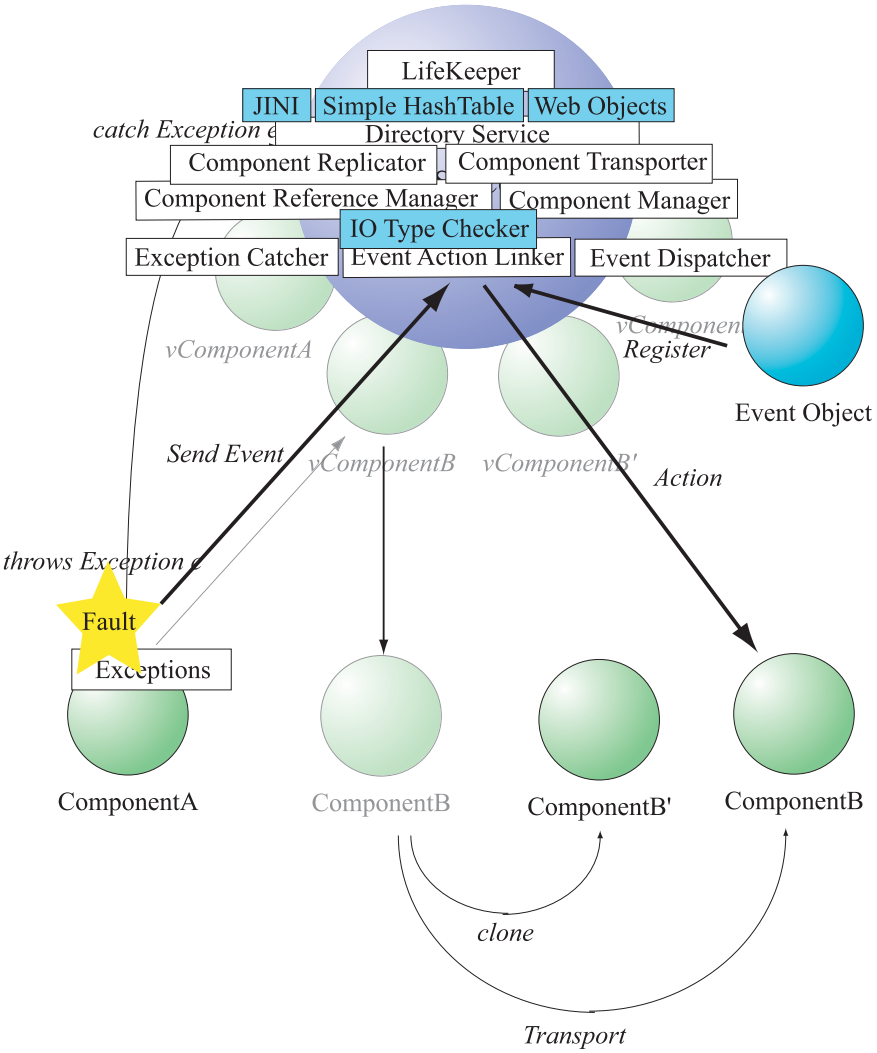

図 4.1: システムのソフトウェア構成

4.3 システム各部の詳細

全体構成について述べる。本システムは先ほど示した図のように、三つの部から成る。ディレクトリ部、マネージメント部、イベント部の4つの部である。これよりそれぞれの部におけるより細かいサブシステムの解説を行う。

ディレクトリ部

ディレクトリ部はコンポーネント自身の情報を登録し、他コンポーネントの発見を行う部分にあたる。他コンポーネントからコンポーネントの発見を行うためのリクエストが来た場合、返事を返し、コンポーネント及びユーザが入力した検索内容を他コンポーネントへ通知する役割を持つ。

Directory Service

Directory Service は上述したコンポーネントの登録及び発見を行う部分にあたる。コンポーネントは同時にオブジェクトであるため、既存の分散オブジェクト用ディレクトリサービスを利用する。本システムで想定する Directory Service として JINI や SOAP などのような分散オブジェクト用ディレクトリサービスを想定しているが、今回はプロトタイプの実装として、より簡単なハッシュテーブルを提供する。ハッシュテーブルの中身として、コンポーネントの名前、IP アドレス、ポート、ポートの型を管理する。それぞれの項目に対するリクエストを送ると、返り値としてリクエストした項目を得ることができる。

Life Keeper

Life Keeper はコンポーネントの参照先が保つことができているかを管理する部分である。これは、コンポーネント自身が障害に陥った際、その状態変化を他コンポーネントへ通知することができないために必要な機構である。例えばコンポーネントが存在するホストの電源が落ちてしまった場合、コンポーネントは通知する時間もなく損失されてしまう。このため、コンポーネント自身が障害の通知を行うことは望ましくないと言える。

本システムではこれらを考慮し、コンポーネントと連携する FuSe Core 内に Life Keeper という機構を用意する。コンポーネントの参照を一定期間ごとにチェックし、参照が無くなっている (オブジェクトが無くなっている) 場合、障害が発生したと判断し、行動のトリガをイベント部へ発行する。

なお、Directory Service が JINI であった場合、JINI が提供する Lease Time を利用することができるため、この場合において Life Keeper は動作する必要はない。また、参照先を管理するのはマネージメント部の Component Reference Manager であるので、参照関係はそちらを参考にして頂きたい。

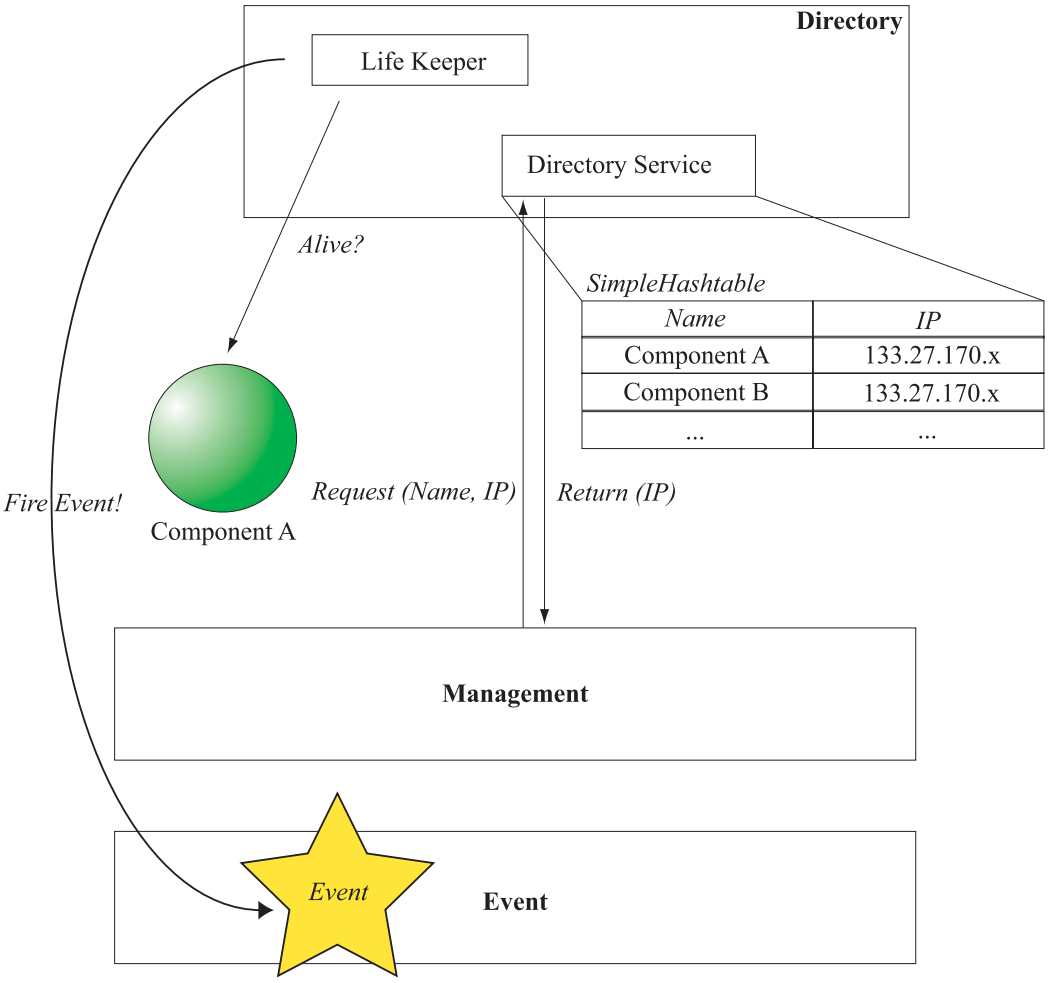

図 4.2: ディレクトリ部のシステム構成

マネージメント部

マネージメント部ではコンポーネントの全般的な管理を行う。コンポーネントの複製・移送、この他にコンポーネントの参照を保たれているかの確認及びコンポーネントの仮想コンポーネントを格納するための場所でもある。以降、それぞれの機能に対応した四つの機構について取り上げ、説明を行う。

Component Replicator

Component Replicator は呼ばれると参照しているコンポーネントの複製を行う。また、特定のコンポーネントから仮想コンポーネントを作成する場合にも呼び出される。複製されたコンポーネント及び仮想コンポーネントは作成後、Component Manager に渡され、そのとき参照されているコンポーネントの情報を Component Reference Manager へ登録する。

Component Transporter

Component Transporter はコンポーネントの移送を行う際に呼び出される機構である。具体的には、コンポーネントをロックし、シリアライズしてから転送先のホストへ転送を行う。この機構と Component Replicator との差として、Transporter は同期を行う必要があるのに対し、Replicator は非同期にスナップショットを作成するところに差異がある。具体的に、コンポーネントのロックを行うか行わないかによって呼び出す機構が異なる。

Component Reference Manager

Component Reference Manager は現在 FuSe Core が認識しているコンポーネントがどの他コンポーネントを参照しているかの情報を格納する。複製・移送によって特定コンポーネントの参照先が変更された場合、この Component Reference Manager へその情報が通知され、参照が変更される。また、本来参照を行っていなければならないコンポーネントの生存がディレクトリ部の Life Keeper によって確認された場合、障害が発生したと判断され、行動を開始するトリガが発行される。

Component Manager

Component Manager は仮想コンポーネントが生成された場合、その仮想コンポーネント自身を格納するための機構である。実際では生成された仮想コンポーネントは他ホスト上に保存されることもあるが、基本的に同じホスト上に登録されることが必要である。また、現在の仕様では複製されたコンポーネントもこの Component Manager によってコンポーネントが格納されるものとなっている。

(IO Type Checker)

正確にはマネージメントの部ではないが、Component Reference Manager と深く関わりを持つため、本節において型の照合を行う IO Type Checker について取り上げる。

コンポーネント間で所在 (IP) 及び通信インタフェース (port) を指定することで通信が開始できるが、この他に型情報が必要である。例えば、void のように返り値のないような簡単なアプリケーションの場合ではスイッチを入れる感覚でメソッドを呼ぶだけで実行できてしまうが、より高級なアプリケーションでは異なる型の情報をやり取りする際に型情報が必要である。型照合をしなければ通信を開始しようとした際、型の一致がないため通信自体は失敗してしまう。よりシンプルなシステムでは通信はできるものの、通信される情報は正しく解釈されない可能性がある。本システムの場合、コンポーネント間通信が確立されない場合は障害と見なされ、例外処理が行われてしまう。

プリンタを例に取り上げる。プリンタはプレーンテキストやポストスクリプト言語で係れたデータを受け取るサービスとも言える。しかし、ここに型の異なるもの、例えばプリンタへオーディオファイルを送信すると、意味のない情報が印刷されてしまう。

このため、高級なアプリケーションを構築するために型照合は必要であると言える。本システムではコンポーネント間通信を確実に行うため、通信を確立する際に型照合を行う機構として IO Type Checker を用意する。

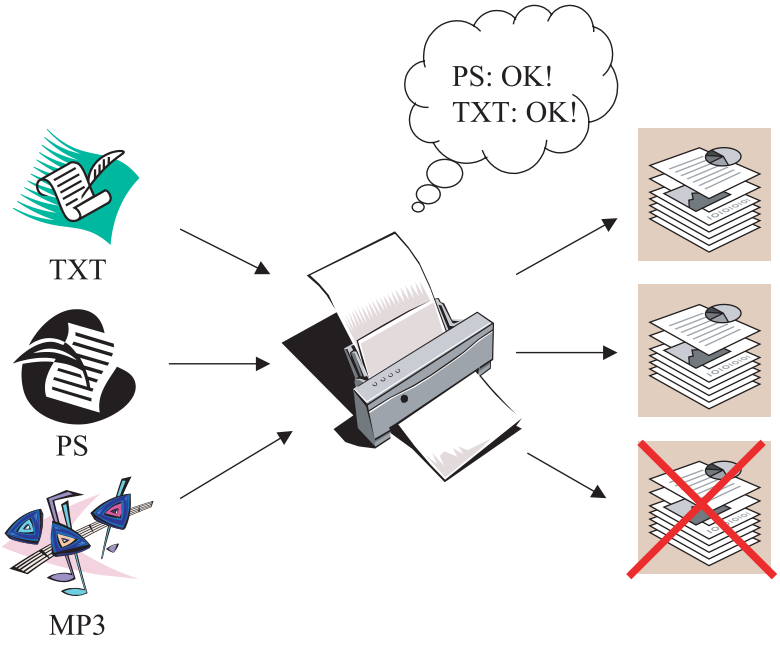

図 4.3: プリンタの型を扱った例

最後にマネージメント部の全体ソフトウェアシステム図 4.4 を提示する。

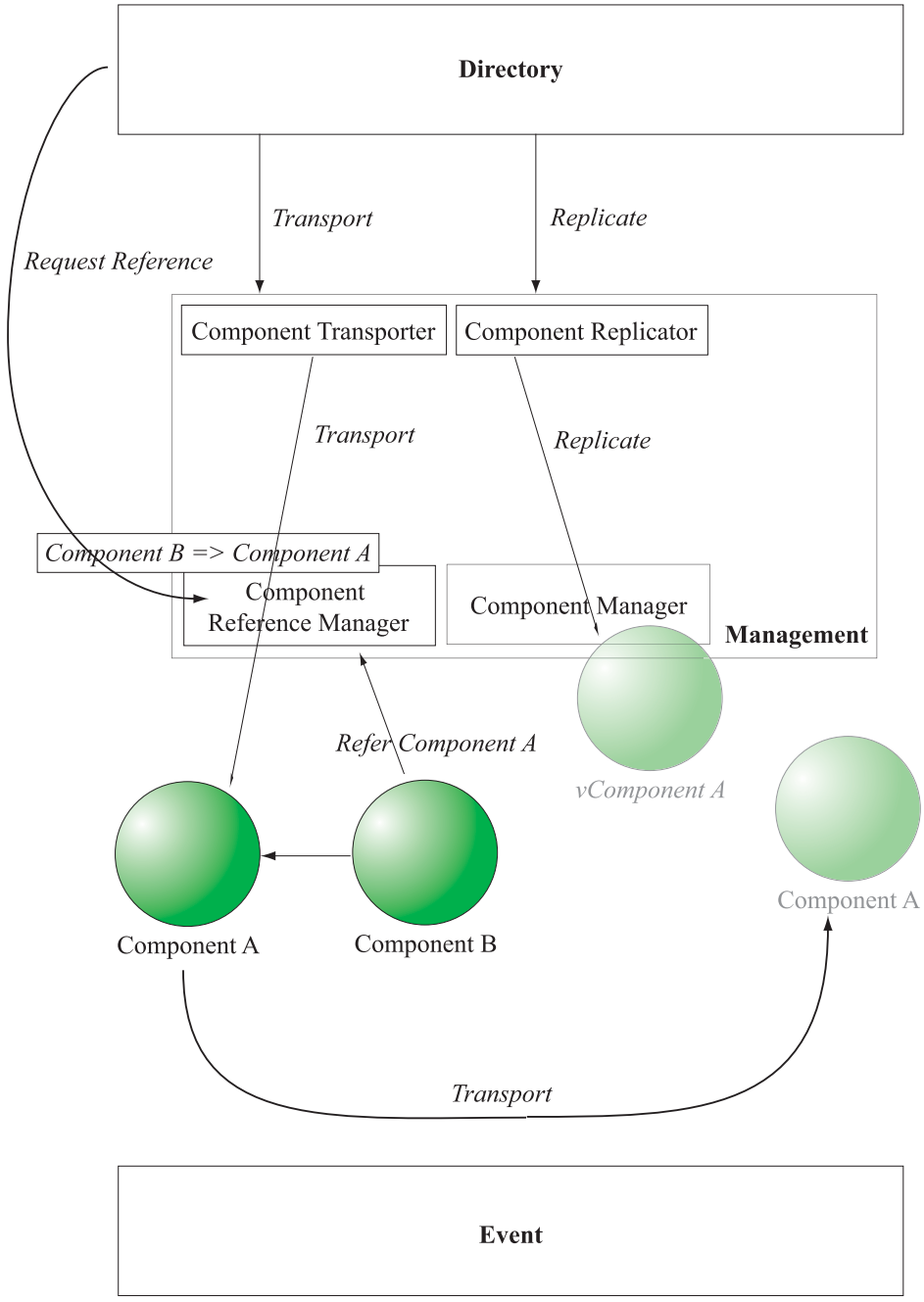

図 4.4: マネージメント部のシステム構成

イベント部

Exception Catcher

本システムにおいて、障害発生の通知を受け取るのが Exception Catcher である。システム内で発生する障害の検知を行うのがこの機構であり、これにより障害発生時の行動を発動させるためのトリガを生成する。具体的に、コンポーネントによって宣言された例外処理によって発生した例外を受取り、Event Action Linker へ渡す役割を持つ。その後、トリガと行動がリンクされ、最終的に行動が起動される。

Event Dispatcher

イベント部における役割であるリンク及び行動のうち、行動を発動させる部分に当たるのが Event Dispatcher である。Event Dispatcher は Event Action Linker より開始命令を受けると、Event Action Linker によって指定されたコンポーネントに対し Event Object へそのコンポーネントを引数として渡す。これにより、Event Object によって定義された行動は該当コンポーネントに適応されることとなる。

Event Action Linker

上述した Exception Catcher と Event Dispatcher をつなげる役目を持つのが Event Action Listener である。あらかじめユーザが定義された、特定コンポーネントに障害が発生した場合、それに対する行動を記述した Event Object が登録される。Exception Catcher によって渡された Exception を受取り、その Exception が発行したコンポーネントが Event Object として登録されているかを確認する。もし登録されていた場合、その Event Object は Event Dispatcher へ渡され、実行される。

最後にイベント部のシステム全体図を提示する。

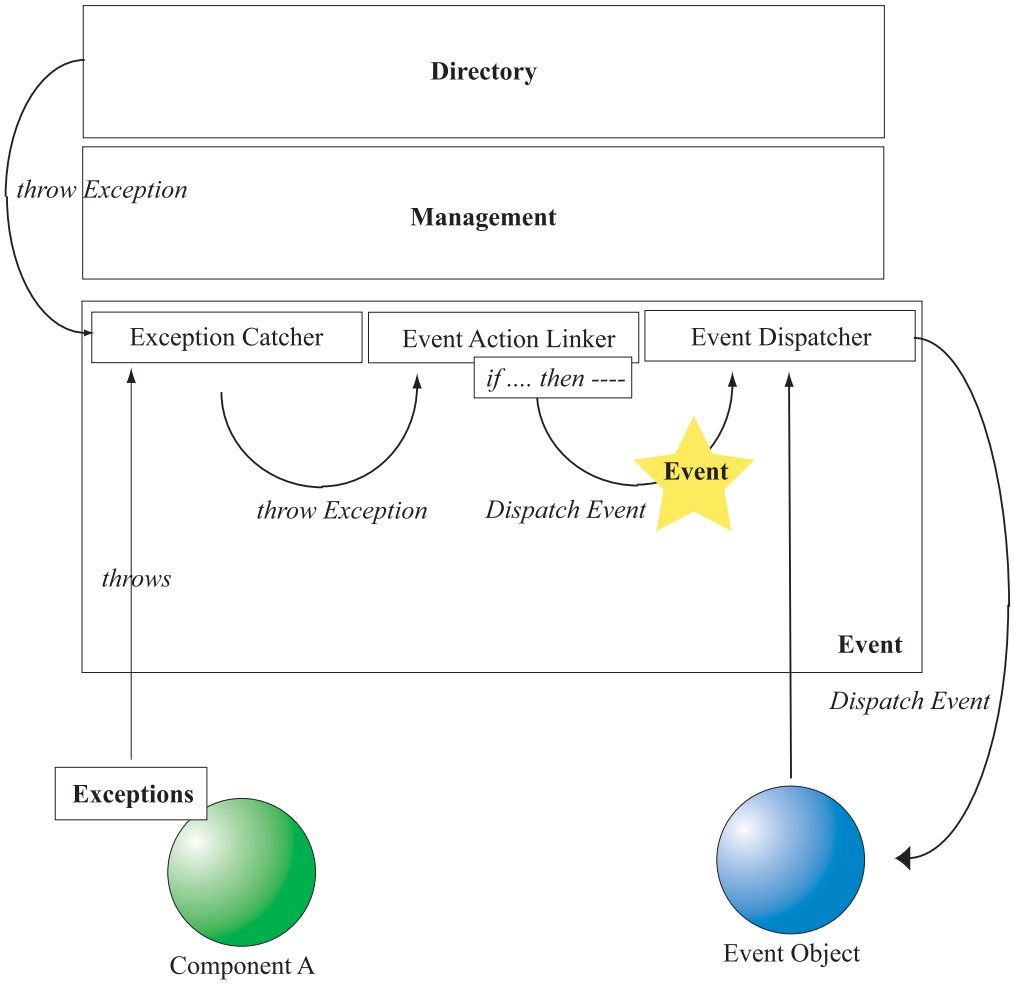

図 4.5: イベント部のシステム構成

その他

ここまで取り上げたシステムの詳細以外に、ユーザが直接定義するもの、コンポーネント開発者が直接定義するものがある。本節ではシステムと関連するこれらのサブシステムについて取り上げる。

(Event Object)

イベント部の正式的な一部ではないが、Event Action Linker と深く関わりを持つため、本節ではイベントの行動を記述する Event Object について取り上げる。Event Object はユーザによって定義された行動と条件文である。これは特定のトリガに対し、どういった行動を定義するかを設定したオブジェクトであり、記述されたものは Event Dispatcher に登録される。Event Dispatcher によって登録された Event Object は保持される。あとは Event Action Linker に対しトリガと行動による条件文が渡され、該当する例外処理が渡されたときに始めて Event Object が呼び出され、実行される。

(Exceptions)

上述した Event Object 以外に、本システムとは別に、コンポーネント開発者が記述する Exceptions について取り上げる。Exceptions は具体的に Java API で言う throws に当たる。コンポーネント開発者がコンポーネントを処理する際に予想される Exceptions を API として記述することで、後に利用者が設定する行動とリンクされるようになる。

4.4　本章のまとめ

本章では FuSe の各部の設計方針として同期的通信、非同期的通信、容易性を重視することを目的として定める。これらの機構を実現するために、FuSe のサブシステムであるディレクトリ部、マネージメント部、イベント部、さらにそれらを構成する詳細ソフトウェアである 11 種類のソフトウェア構成について述べた。これにより、システムソフトウェア全体の構成を具体化した。次章では FuSe のシステムソフトウェアの全体設計を元に、プロトタイプの構築方法似ついて述べる。

第5章　FuSeの実装

5.1　実装環境

本システムの実装はJava言語 [6] を用いて行った。実装言語としてJava言語を選んだ理由は以下の通りである。

マルチプラットホーム性

Java言語で書かれたプログラムは一度中間バイトコードを生成し、その後Java Virtual Machineと呼ばれるインタプリタを用いて実行される。このため、一度コンパイルされたプログラムはマシンアーキテクチャが異なるJava Virtual Machineであっても再コンパイルせずに実行ができる。本システムはネットワークに接続されたマシンアーキテクチャが同じとは限らないホスト上で利用されることを前提としているため、この特性は有益である。

ネットワーク指向型

本システムにおける作業時間の多くはネットワークを利用したアプリケーション及びデータの転送である。このため、ネットワークを考慮したプログラミング言語を用いて開発を行うことが前提となる。Java言語はネットワークに関するライブラリが豊富であり、ネットワーク通信を実装したアプリケーションの構築が容易である。

コンポーネント指向

Java言語では実行時にクラスをロードし、インスタンスを形成する特徴を持つ。これは実行時にクラスの振る舞いが決定されることを意味する。このため、アプリケーションをコンポーネント指向する上で適していると言える。

5.2　プロトタイプの実装範囲

本システムの設計を基に実装を行うが、今回の目的はモデルの正当性を確かめることであるため、極力シンプルな構造を持つプロトタイプによるテストを行い評価を取

らなければならない。このため、設計において示したシステム図のうち、最低限のサブシステムを抽出し、プロトタイプとして実装する。

本論文におけるプロトタイプを実装する上で、これより述べるサブシステムを実装したものをプロトタイプとして実装する。

5.3 FuSe の実装概要

5.4 ディレクトリ部の実装

ディレクトリ部は二つの部である LifeKeeper と Directory から成り、さらに三つの細かいクラスから成る。本節ではこれらのクラスの実装及び上部の二つの部との関連について取り上げる。

Life Keeper における実装

Life Keeper はコンポーネントの参照が保てられているかを確認するための機構である。基本的に unicast でオブジェクトを問い合わせ、返事がなかった場合に NullPointerException を EventNotifier に対して送信する。これらの機能を提供するクラスとして、LeaseTime、ReferenceChecker、EventNotifier を実装した。次にこれらのクラスの細かい仕様について取り上げる。

LeaseTime の実装

LeaseTime は先ほど述べた文章のように、一定期間ごとにオブジェクト参照機構である ReferenceChecker を呼び出す機構である。具体的に、time として設定された期間ごとに wait し、time 分の時間が経過すると ReferenceChecker に対し、boolean を返り値として持つ isObjectAlive メソッドを呼び出す。返り値として true が返ってくるようであればコンポーネントは生きていることになり、再度 time 分の経過を待つ。もし Exception が返ってきた場合は EventNotifier に対し Exception の内容と Object を値として渡す。

public final void wait(long time);
public LeaseTime(long time);
public boolean isObjectAlive(Object object);
public ReferenceChecker(long time, Object object);

表 5.1: LeaseTime クラスのメソッド一覧

ReferenceChecker の実装

ReferenceChecker は LifeKeeper から渡された time と object の情報を基に、Thread として抽象化されたコンポーネントを notify し、該当 Thread を wake する。ここでもし Thread が無い場合は Exception が発生する。結果として、LifeKeeper へは boolean である true、及び Exception の内容を含んだメッセージが返されることになる。

public boolean isObjectAlive(Object object);
public final void wait(long time);
public EventNotifier(Exception e);

表 5.2: ReferenceChecker クラスのメソッド一覧

EventNotifier の実装

EventNotifier はシステム内のところどころにあるが、基本的な役割はソフトウェア内の Exception を Exception Catcher へ集約させることである。このため、機能としては基本的に Exception を引数に EventNotifier は生成され、EventNotifier 自身は受け取った Exception を特定プログラム、今回の場合は Exception Catcher へ向けて投げるようになっている。

public void throwsException(Exception e);

表 5.3: EventNotifier クラスのメソッド一覧

Directory Service における実装

Directory Service はコンポーネント自身の情報を登録及び削除するための機構である。基本的な役割として他コンポーネントから登録されたコンポーネントと通信するための情報を保持する。今回の実装では最も管理が楽な SimpleHashtable によるコンポーネント管理を用いた。

SimpleHashtable の実装

SimpleHashtable は極めてシンプルな構造を持つ。他コンポーネントからは自分自身を登録する場合は addComponent を呼び、情報の登録を解除する場合は removeComponent を呼ぶだけで Object のディレクトリサービスへの登録・解除を行うことが可能である。登録されたコンポーネントの情報を引き出す際にはクエリメソッドに該当する引数を

渡すことで、その他のフィールドの情報を得ることができる。例えば引数に Object を渡すことでそのオブジェクトが持つ IP アドレス、ポート番号、またその port の型に関する情報が引き出される。

public String query(Object);
public String query(String);
public String query(Port);
public String query(Class);
public void addComponent(Object);
public void removeComponent(Object);

表 5.4: SimpleHashtable クラスのメソッド一覧

なお、SimpleHashtable クラスが所持するメソッド一覧の他に、ハッシュテーブルの仕様も掲載しておく。

Object object	String IP	Port port	Class class
Component A	133.27.170.97	80	String
Component B	133.27.170.98	22	IOStream
Component C	133.27.170.99	25	String

表 5.5: SimpleHashtable が所持するテーブルの仕様図

5.5 マネージメント部の実装

マネージメント部は四つの部である Component Replicator、Component Transporter、Component Reference Manager、Component Manager から成り、それぞれのサブシステムを構成する四つの細かいクラスから成る。本節ではこれらのクラスの実装及び上部である四つの部との関連について取り上げる。

Component Replicator における実装

Component Replicator の役割は単純且つ明確である。特定コンポーネントを受け取ると、そのクローンを生成し、他ホストへ送信される。その後、複製コンポーネントの存在はディレクトリサービスへ登録される。これにより、複製前のコンポーネントと同じように利用することができる。

ComponentClone の実装

ComponentClone は Component Replicator の性質である複製を担当するクラスである。用意されているメソッドは二つである：clone() と send() である。本システム内に存在するオブジェクトは基本的に cloneable を継承しており、clone することで同じオブジェクトを生成することができる。生成されたコンポーネントはその後 Java のシリアライズ機能を使うことで、特定の他ホストへ送信され、復元される。もちろん、このホスト上に FuSe Core が走っているので、そちらのディレクトリサービスに登録されることで他 FuSe Core から利用可能となる。

protected Object clone();
public void send(InetAddress);

表 5.6: ComponentClone クラスのメソッド一覧

Component Transporter における実装

Component Transporter の役割は Component Replicator と同様、単純且つ明確である。Component Replicator が持つ役割の片方である、コンポーネントの送信を専門とするのがこのサブシステムである。今回の実装においては、ComponentClone の送信機能のみを所持したものとなっている。

ComponentSend の実装

ComponentSend は Component Transporter が専門とするコンポーネントの送信機能を提供する。メソッドも send() と一つしか所持しない。これは現在の実装では ComponentClone と同じ機構である Object Serialization しか行っていないが、将来的に排他制御が行えるように仕様を拡張したものを提供する。

public void send(InetAddress);

表 5.7: ComponentSend クラスのメソッド一覧

Component Reference Manager における実装

Component Reference Manager は同ランタイム内でどのコンポーネントがどのコンポーネントを参照しているかの一括管理を行っている。これにより、先ほど述べた Component Transport によって転送されたコンポーネントを参照先を保ったまま、コンポーネント自身の転送が行えるようになる。

SimpleReferenceManager の実装

SimpleReferenceManager はハッシュテーブルであり、Directory Service においてクエリを発行したコンポーネントとそのクエリと一致したコンポーネントは互いに参照されていると判断され、それらの Object 情報はハッシュテーブル内のエントリーとして追加される。具体的に、addObject() を呼ぶことでハッシュテーブルに参照されているオブジェクトに対し、参照していると思われるオブジェクトがフィールドとして追加される。後に、getField() に引数として該当 Object を渡すことで参照している・されているオブジェクトの情報を得ることが可能となる。

public void getField(Object)
public void addObject();
public void removeObject();

表 5.8: SimpleReferenceManager クラスのメソッド一覧

Object	Object
Component A	Component B
...	...

表 5.9: SimpleReferenceManager が所持するテーブルの仕様図

Component Manager における実装

Component Manager は複製された仮想コンポーネントを格納する機能を提供する。仮想コンポーネントは複製元のコンポーネントとリンクされており、仮想コンポーネントに対して呼ばれたメソッドや渡された引数はそのまま大元のコンポーネントへ渡すようになっている。これにより、他コンポーネントは仮想コンポーネントを通して本来通信したいコンポーネントを呼ぶことが可能となる。

SimpleComponentManager の実装

SimpleComponentManager はハッシュテーブルである。ハッシュテーブルはコンポーネントに対する仮想コンポーネントを対応させるために利用する。利用方法は他のハッシュテーブルと同様、addObject()/removeObject() によって getObject() で該当するオブジェクトに対応したオブジェクトを取り出すことができる。

public void getObject(Object);
public void addObject();
public void removeObject();

表 5.10: SimpleComponentManager クラスのメソッド一覧

Object object	Object object
Component A	vComponent A
...	...

表 5.11: SimpleComponentManager が所持するテーブルの仕様図

イベント部の実装

イベント部は三つの部である Exception Catcher、Event Action Linker、Event Dispatcher から成り、さらにつの細かいクラスから成る。本節ではこれらのクラスの実装及び上部である三つの部との関連について取り上げる。

Exception Catcher における実装

Exception Catcher はシステム内の各サブシステムが throw する Exception を受け取る役割を持つ。Exceptions にはクラス自身が投げる Exception と JavaVM 自身が投げる Exception がある。後者は投げた時点で JavaVM が停止してしまうため、JavaVM 内にロードされている Exception Catcher によって catch することができないため、今回は前者の Exception を取り扱うこととする。

ExceptionCatcherImpl の実装

先ほど書いたように、このクラスは他クラスから排出する Exception をキャッチすることを目的とする。これは具体的に、各クラスをそれぞれ ThreadGroup として束ね、その ThreadGroup 内で Exception が検出された場合、uncaughtExceptions が throw される。これは引数として Thread t, Exception e が渡される。これらの情報を Event Action Linker へ渡すことで Exception の通知を行う。

Event Action Linker における実装

Event Action Linker は行動のトリガである Exception と行動自身である Event Object の連携を決定する機構である。これは具体的に、特定の Exception に対する Event Object がセットで登録され、Exception が catch されたときに照合を行い、Exception が照合し

public Thread(ThreadGroup group, String name);
public void addThread(ThreadGroup group, String name);
public void removeThread(ThreadGroup group, String name);
public void getThreadName(String name);
public void list();

表 5.12: ExceptionCatcherImpl クラスのメソッド一覧

た場合に登録された Event Object は Event Dispatcher へ渡される。

EventActionLinkImpl の実装

Event Action Linker を実装したものが EventActionLinkImpl である。これはハッシュテーブルであり、登録されたコンポーネントに対し、Event Object がセットで登録される。Exception Catcher から Exception e を渡されると、ハッシュテーブル内で Exception を登録しているものを抽出する。一致したものがあった場合、それらと対になっている Event Object が呼び出され、Event Dispatcher へ渡される。

public getException(Exception)
public getObject(Object object)
public void addObject(Exception, Object);
public void removeObject(Exception, Object);
public void list();

表 5.13: EventActionLinkImpl クラスのメソッド一覧

Exception	Object
Exception A	Event Object A
Exception B	Event Object B
Exception C	Event Object C

表 5.14: EventActionLinkImpl が所持するテーブルの仕様図

Event Dispatcher における実装

Event Dispatcher の役割は単純であり、EventActionLinker から Event Object を渡され、それを実行する役割を持つ。Event Object 自身は実行体であるため、受け取るとクラスをロードし実行することを機能として持つ。

EventDispatcherImpl の実装

先ほど述べたように、ただ Object を取り込んで実行するだけで役割を果たすため、今回実装したクラスの中のメソッドは一つしかない。より正確には、EventActionLinkImpl からそれに引数として Event Object が渡され、その値を exec() メソッドに呼びこみ実行される。

public void exec(Object);

表 5.15: EventDispatcherImpl クラスのメソッド一覧

5.6 本章のまとめ

本章では分散コンポーネント冗長化システムの実装例である FuSe ランタイムのプロトタイプについて述べた。まず FuSe の実装環境について述べ、次にディレクトリ部、マネージメント部、イベント部について述べた。そしてさらに、それらのサブシステム内の具体的なソフトウェアの実装仕様について述べた。次章ではこのシステムの評価を行い、その有用性について検証する。

第6章　システムの評価

6.1　定量的評価

ここでは3章で示した原理を基に構築した分散コンポーネント冗長化システムであるMuGeNが正しく動作しているかを検証するため、MuGeNの基本性能を定量的に評価を行いながら以下に説明する。

測定環境

測定環境として本研究室のネットワークを介して測定を行った。環境は図6.1のような構成となっている。

なお、図6.1中の各マシンの仕様は表6.1に示す。

項目	speed	nismo	mugen	trd
ハードウェア	AMD Athlon XP 1.53GHz	Pentium III 800MHz	AMD Duron 800MHz	Pentium III 750MHz
オペレーティングシステム	WindowsXP 2002 SP1	Windows2000 SP4	FreeBSD 5.1R	Windows2000 SP4
実行環境	JDK1.4.2	JDK 1.3.1 (FreeBSD-native)	JDK 1.4.2	JDK 1.4.2

表 6.1: 測定で利用したマシンの仕様表

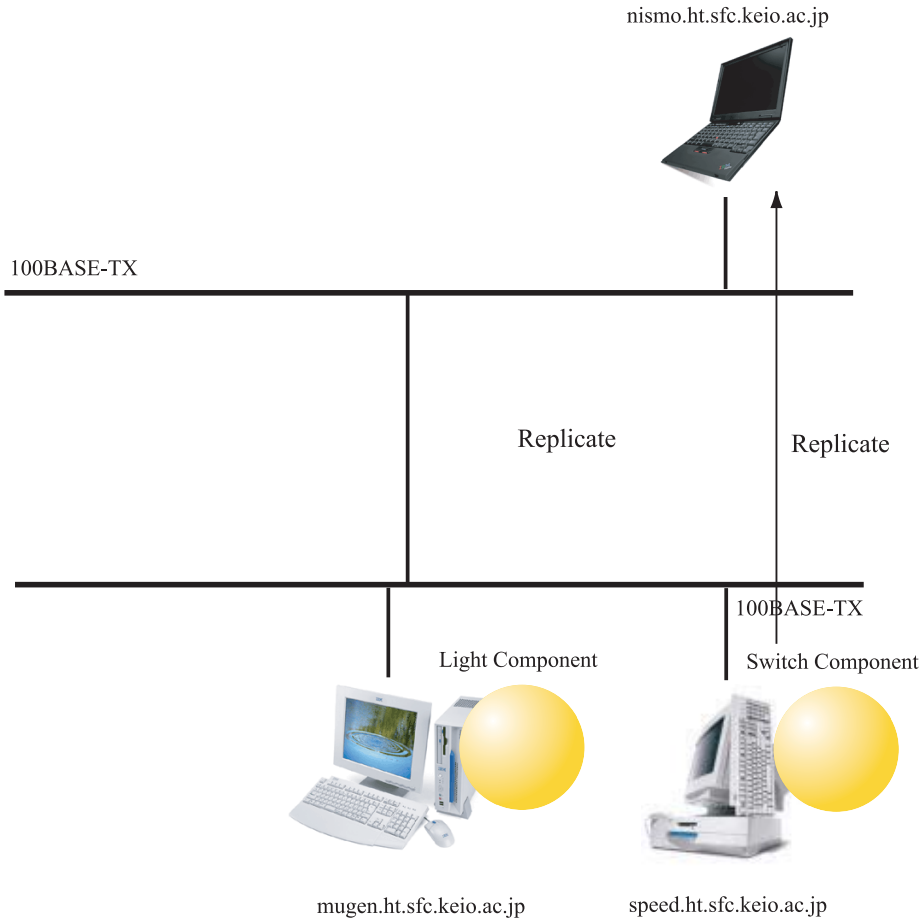

図 6.1: 測定環境

測定方法

各ホスト上で FuSe Core を起動し、それぞれをコンポーネントを保持できるようにプラットホームとして機能してもらうようにした。また、speed と nismo 上にそれぞれ Light コンポーネントと Switch コンポーネントを設置した。このネットワークは有線によって接続されている。ここで Light コンポーネントと Switch コンポーネントを連携し、スイッチを捻ると電気が付くアプリケーションを構築するまでの時間 (ブートストラップ時間)、コンポーネントの複製にかかる時間、また擬似的に障害を発生させ、他ホスト上のクローンと参照を切り替えるまでの時間を検証する。

ブートストラップ時間の検証

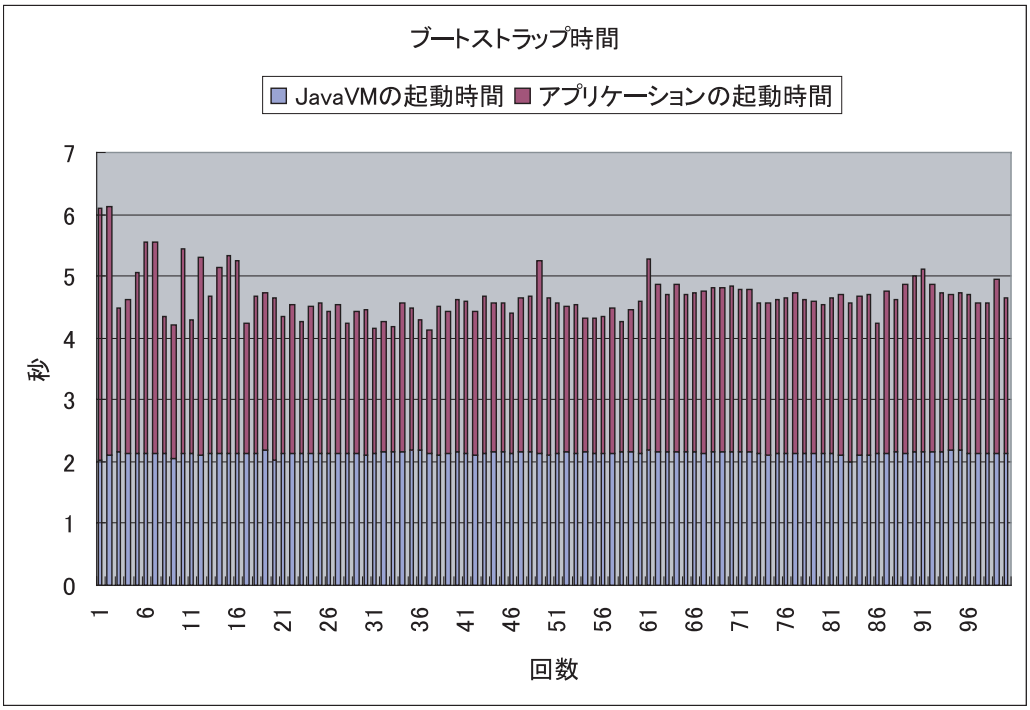

図 6.2: ブートストラップにかかる時間

起動するまでの時間を 100 回起動してみて時間を計った結果、平均して 4.61 秒の起動時間を要することが分かった。しかし、このうちの 2.13 秒は JavaVM 自身の起動時間であるため、実質ランタイムのロード時間は 2.54 秒ほどである。これは十分実用に耐えうる時間であると考える。また、仮に時間がかかったとしても、起動は基本的に一回のみしか行わないため、使用上においては問題にならないと考える。

コンポーネントの複製にかかる時間

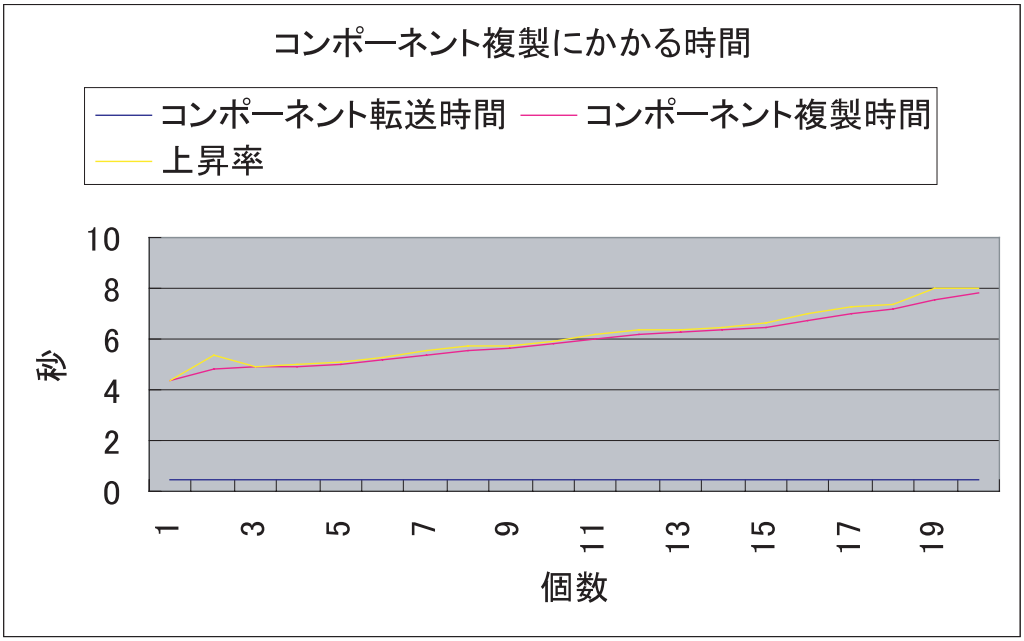

図 6.3: コンポーネントの複製にかかる時間

次にコンポーネントの複製の検証を行った。speed 上のコンポーネントを複製し、nismo 上に転送するという手順で行った。この検証では複製の個数が増えていくことでどれほど動作が重くなることを検証する。

結果的にコンポーネント 1 つ複製をし、転送するのに必要な時間は 4.9 秒、個数 1 個につき 0.17 秒ずつ増えていくことが分かった。(転送時間はおよそ 0.4 秒ほどかかっている) また、100 個までコンポーネントを増やすにつれ、比例して 0.17 秒ずつ時間が増えていくことが分かった。これにより、ユーザがコンポーネントの個数を増やすに連れ十分にスケールすることが考えられる。

コンポーネントを切り替えにかかる時間の検証

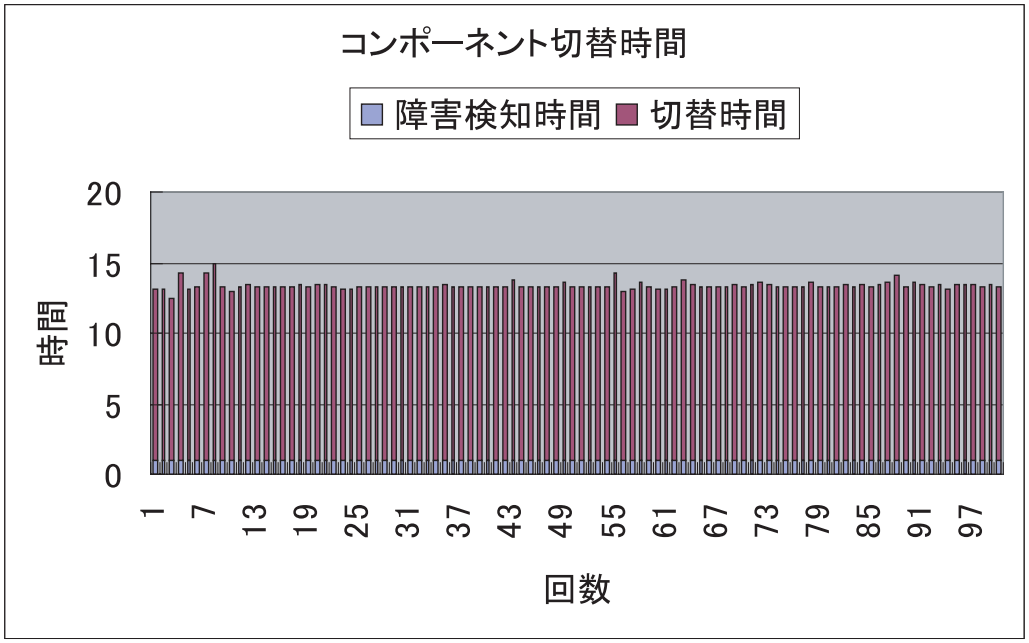

図 6.4: コンポーネントの切り替えにかかる時間

最後に、nismo 上に複製コンポーネントを設置した状態で意図的に speed 上で障害を発生させることでコンポーネントの切り替えを行わせた。障害の内容は speed 上のコンポーネントを意図的に削除すると言った内容である。この実験を 100 回繰り返し、グラフに表してみた。

結果として、切り替えに必要とする時間は平均して 12.36 秒かかった。これは Lease Time が 1 秒で設定されているため、実質切り替えを行うまでの時間は 11 秒ほどと考えられる。ボトルネックは切り替える際に複製コンポーネントを発見し、接続しに行くところと考えられ、今後改善の余地はあると考えられる。

6.2 関連研究との比較

ここでは本研究と関連する冗長化に対する研究として Network File System(NFS)、Aglets、CORBA、RIO を取り上げ、分散コンポーネント冗長化システムである FuSe と比較しながら議論を進める。

Network File System との比較

複製を作成するという観点で Network File System [7] を取り上げる。Network File System は広く使われている仮想ファイルシステムの一つであり、既存のファイルシステム上からもう一段皮を被せることで OS 非依存な分散ファイルシステムを提供している。

しかし、Network File System はデータを設置するはできるものの、クラスから実体化されたオブジェクトを格納することはできない。このため、コンポーネントの保存はできないと考えられる。また、ファイルの I/O を正確に排他制御を行うための機構は正式的にないため、ファイルへのアクセス中にデータが壊れることがあるため、本システムが想定する環境での利用には適しないと考えられる。

Aglets との比較

Aglets [8] は IBM 研究所によって開発されたモバイルエージェントプラットホームである。アプリケーションはモバイルエージェントであり、複数のホスト間を飛び回り実行していく性質を持つ。アプリケーション自身がシステムとは自律的に行動を行うのがこのシステムの最大の特徴である。

Aglets を取り上げた理由として、アプリケーションの移送が行えるプラットホームであり、且つ複製も簡単に行える点を考慮したためである。しかし、Aglets ではモバイルエージェント自身がアプリケーションであり、さらに一体化された単独アプリケーションでもある。これは分散コンポーネントから構成されたアプリケーションは扱えないことを意味する。このため、本システムが想定する分散コンポーネントベースのアプリケーションを構築する上で不適切であると考えられる。

CORBA との比較

CORBA[9] は OMG 団体によって提唱されているクロスプラットホームなコンポーネントを構築するための仕様である。この仕様に基づいて CORBA ベースのコンポーネントを作成することで、異なる言語であっても同じ CORBA ベースのコンポーネントと相互接続は可能となる。

また、サードパーティライブラリの多さも特徴の一つであり、これらを利用することで様々な機能拡張を行うことができる。現時点では、本システムが提供するコンポー

ネントの移送と複製、またコンポーネント間の型一致や分散例外処理を提供するサードパーティライブラリの確認が行えた。

しかし、本システムのように例外処理を障害として捕らえ、それに対する行動とリンクする機能を提供するライブラリは現時点では見当たらない。このため、本システムはCORBAに提供されるライブラリより機能性の面でより充実している面があるため、分散コンポーネントの冗長化という観点において適していると言える。

RIOとの比較

RIO [10] はJiniベースの分散サービス指向のシステムであり、特徴としてネットワーク上のサービスを探し出して、そのサービスを受取る性質を持ち合わせています。作者はこの性質を”Dynamic Service Delivery Architecture” と呼んでおり、この性質を利用することでサービスは固定的に特定のホストに結び付けられずに、ネットワーク上の利用可能なコンピュータ資源に動的に割り当てられる。これは本システムで言うコンポーネントの移送・複製機能によって副作用的に産み出されるサービスの移動という性質と近似している。しかし、耐故障性用の対策として、RIOの場合ではJiniのLease Timeを利用することで障害検出を行っている。FuSeではこれと似たLifeKeeperという機能があるが、この他にExceptionCatcherによる例外処理をキャッチするというもう一手の障害検出機構を備えている。このため、耐故障性の面においてはFuSeの方が目的により適していると言える。

6.3 本章のまとめ

本章ではFuSeプロトタイプの評価を行った。本章の前半ではFuSeプロトタイプの特徴であるコンポーネントの複製と切り替えに関する定量的評価を行い、さらに起動時に係る時間と合わせ、一般利用には適切であることを実証した。また、本章の後半では、FuSeを関連研究のシステムであるNetwork File System, Aglets, CORBA, RIOとの機能面における比較を行い、分散コンポーネント冗長化システムを構築する上で適格なモデルであることを示した。

第7章　おわりに

7.1　今後の課題

FuSe システムの今後の課題として示以下の 4 点を取り上げる。

移送機能の完全実装

移送機構は現時点ではオブジェクトをシリアライズ化し、転送するという手法を用いているが、本来ならばローカル上のオブジェクトをロックし、さらにそのロックした状態で転送したあと、転送先のホストでロックを解除する必要がある。これは転送中の間ファイルが書き変わることで転送先におけるファイル損失を防ぐために必要であり、同時にローカルオブジェクトと遠隔オブジェクトの同期を実現する上で必要な機能である。

現時点として、Java RMI の起用について検討を行っている。この手法を最初から適応しなかった理由として、以前の経験から転送時におけるオーバーヘッドが非常に大きいため、転送効率が悪いからである。この他の理由として、極力特定の言語に依存しないようなシステムを設計することを考慮して、普通の InetSocket を改造したものを採用した。しかし、実現がより簡単になるのであれば、RMI の採用は有力であると考え、今後採用について検討を計る。

仮想コンポーネントの分散ホスト対応化

今回実装した FuSe Core においては、仮想コンポーネントは遠隔ホスト上に置くことができない問題があった。これは以前にも述べたように、仮想コンポーネントとコンポーネントの参照を共有する上で同ホスト上であれば同メモリ上にオブジェクトが設置されているため容易であるが、遠隔ホスト上にオブジェクトを置くことで同メモリ上にオブジェクトを参照し続けることができなくなるため、一方のオブジェクトへの反映はもう一方のオブジェクトに反映させることができない。今後研究調査を進めて、適切な回を導き足すこととする。

ディレクトリサーバのメタ情報への対応

ディレクトリサービスでは今回実装したSimpleHashtable以外に、JiniやWebObjectsも利用することができる。しかし、どれも現時点ではメタデータが扱えるものはない。

現時点でオブジェクトを指すときはその名前であるIDで呼び出すようになっているが、メタ情報を付加することでメタ情報を元にオブジェクトの抽出が行えるようになる。今後、メタデータを付加したディレクトリサービスを導入することでコンポーネントの呼び出しをより容易にしたいと考えている。

コンポーネントの充実

本システムが最も肝心である、コンポーネントからアプリケーションを構築することが前提であるにも関わらず、現時点システムが提供するコンポーネントはswitchコンポーネントとlightコンポーネントの二つのみしかない。これではアプリケーションを組む上で組み合わせが限られてしまうため、今後コンポーネントを増やし、より多くの組み合わせによって作られるアプリケーションの種類を増やすようにしたいと考えている。

7.2 まとめ

本論文においては、ユビキタスコンピューティング環境におけるアプリケーションの構築方法について取り上げ、さらに構築後に発生しうる障害の分析を行った。このような障害を回避するにはシステム自体を冗長化しておくことで耐故障性を向上させることができると考え、コンポーネント自身の移送・複製を提供することで実現ができた。このようなシステムのことを分散コンポーネント冗長化システムと呼び、そのプロトタイプであるFuSeシステムの特徴であるコンポーネントの移送と複製を取り上げ、それらを具体化するためのシステムの設計と実装を行った。

その後に定量的評価及び定性的評価を行い、本システムは分散コンポーネント冗長化システムとして適切なモデルであることを実証した。今後は移送機能の完全実装、仮想コンポーネントの分散ホスト対応化、ディレクトリサーバのメタ情報への対応、コンポーネントの充実を計ることでより完成したシステムへの発展へとつなげることとする。

謝辞

本研究を進めるにあたり、御指導を賜わりました、慶應義塾大学 政策・メディア政策 徳田英幸教授に深く感謝致します。

慶應義塾大学徳田・村井・楠本・中村・南研究会の先輩方には、折りにふれ貴重な助言及び御指導を頂きました。特に慶應義塾大学政策・メディア研究科 訪問研究員の中澤仁氏と博士課程の岩井将行氏には本論文執筆を終えるまでにあたり、絶えざる励ましとアドバイスを賜わりました。また、Keio Media Space Family(KMSF) 研究グループのメンバー、特に高橋元氏には絶えざる励ましと本研究に関する様々な議論をして頂きました。

この場を借りて深く感謝の意を表したいと思います。

平成 16 年 2 月 10 日
柳原 正

References

[1] Mark Weiser. “Ubiquitous Computing”. In *IEEE Computer Hot Topics*, 1993.

[2] Roy Want, Andy Hopper, Veronica Falcao, and Jonathan Gibbons. “The Active Badge Location S”. Technical Report 92.1, Olivetti Research Ltd. (ORL), 24a Trumpington Street, Cambridge CB2 1QA, 1992.

[3] “Asymmetric Digital Subscriber Line”. ANSI Standard T1.413-1998.

[4] “ANSI X32T12 (FDDI)”. http://www.nswc.navy.mil/ITT/x3t12/FDDIFAQ.html.

[5] “The Working Group for Wireless Lans”. http://grouper.ieee.org/802/11/.

[6] James Gosling, Bill Joy, and Guy Steele. “The JAVA Language Overview”, 1998.

[7] Sun Microsystems. “NFS: Network File System Protocol Specification”, 1989. RFC 1049.

[8] D. Lange and M. Oshima. “Mobile Agents with java: The Aglets API”. In *World Wide Web*, Vol. 1(3), 1998.

[9] “Object Management Group”. http://www.omg.org.

[10] “Project Rio”. http://rio.jini.org.