マルチメディア通信と分散処理　91－10
（1999．1．22）

エージェント開発環境 ADEPT
～ サービスコンポーネントモデルとその実装 ～

城島 貴弘　朝倉 敬喜　宮下 敏昭
NEC ヒューマンメディア研究所†

従来のエージェントの実装モデルでは、あるサービス(エージェントの機能)を一つのアプリケーションとして作成しているため、サービス毎にアプリケーションを用意する必要があった。また、通信処理のために他の通信エージェントを利用しなければならないといった制約により、通信経路が複雑化し、そのデバッグやテストが非常に困難になるといった問題があった。そこで本稿では、サービスをコンポーネントとして実装し、通信機能を持ったエージェントフレームワークに組み込むことで、様々なサービスを持つ単一のエージェントを実現できるサービスコンポーネントモデルを提案している。このサービスコンポーネントモデルにより、複数のサービスを一つのエージェントで実現でき、通信経路も簡素化されるため、エージェントのデバッグやテストが容易となる。

Agent Developer's Environment "ADEPT"
～ Service Component Model and its implementation ～

Takahiro Shiroshima, Takayoshi Asakura, Toshiaki Miyashita
NEC Human Media Research Laboratories

Agents used to be made as an application software at every service. According to this model, the agent's developer must make many applications with every service. Each agents, therefore, use message transfer agents, and so the route of message transfer become very complicated and hard to be debugged and to be tested each ones. So, we suggest a new agent model called Service Component Model that the agent is composed of some components which supply various services, and has the ability to communicate other agents. In this new model, the agent developer can easily debug and test the agents, because of the agents having various services as components and a simple route of message transfer mechanism.

1. はじめに

従来我々は、オフィス業務を支援するエージェントアーキテクチャとして INA/LI[1]アーキテクチャを提案してきた[1]。INA/LI アーキテクチャは、ワークフロー管理や会議調整などのグループを対象とするオフィス業務を支援するグループワーク支援層と、各個人のスケジュール管理やメール管理を支援するパーソナルワーク支援層との2つの作業階層で構成される。(図 1)

グループワーク支援層で動作するエージェントは、グループウェアエージェントと呼ばれ、ワークフロー管理や会議調整などの各機能毎にエージェントが存在する。パーソナルワーク支援層で動作するエージェントは、パーソナルエージェント、サブエージェントの二種類が存在する。パーソナルエージェントは、ある個人を代表する窓口的なエージェントで、実際のエージェント処理はスケジュール管理やメール管理などの機能を持ったサブエージェントを用いて行う。パーソナルエージェントは個人に対して一つ存在し、サブエージェントは個人業務(メール管理、

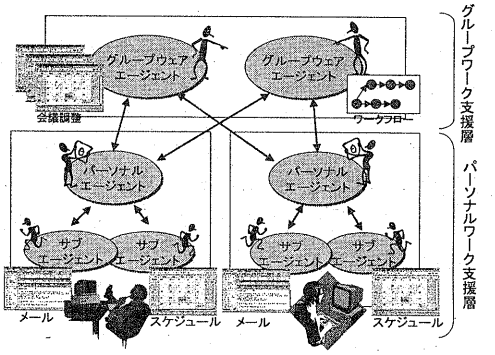


図1 INA/LI アーキテクチャ

† 連絡先: 奈良県生駒市高山町 8916-47 〒630-0101
℡:0743-72-3678 e-mail:sirosima@hml.cl.nec.co.jp
[1] Intelligent Agent/Lite の略

スケジュール管理など)毎に存在する。

この INA/LI アーキテクチャの利点として、以下が挙げられる。

1. サブエージェントの詳細はパーソナルエージェントによって隠蔽されているため、処理依頼時に個々のサブエージェントを指定せずに、パーソナルエージェントに処理の分配を任せればいい。
2. 他のエージェントの動作を考慮することなく、サブエージェントの取り外しが可能である。

しかし、INA/LI アーキテクチャに基づくエージェントの実装では、グループウェアエージェント、パーソナルエージェント、サブエージェントという複数のエージェントを使用するため、サービスの実現のために様々なエージェントを作成しなければならない。また、グループワーク支援層とパーソナルワーク支援層の間で通信を中継するエージェントが必要となり、通信経路が非常に複雑である。以上の要因によって、あるサービス実行時にエラーが発生した場合、どのエージェントに原因があるのかを特定しづらいという問題が起きることがわかった。

そこで我々は、新たなエージェントモデルとして、

- INA/LI アーキテクチャの利点を継承しつつ、
- 複数のサービスの実装が容易で、
- 簡便な通信方法を持つ

エージェントモデルを提案し、エージェントの設計からデバッグまでの一連の作業を支援するエージェント開発環境 ADEPT[2]において実装を行った[2]。

以下、新たなエージェントモデルとしてサービスコンポーネントモデルについて述べ、その ADEPT での実装方式について述べる。

2. サービスコンポーネントモデル

過去、エージェントモデルとして Agent0[3]や ADIPS[4][5]などが提案されている。Agent0 は、主に自律エージェントと動作し、ADIPS は複数のエージェントを動作環境に応じて選択するシステムである。先の INA/LI やこれらのエージェントが、エージェントで行う処理(会議システムやエージェント間交渉)毎に存在するのに対し、ADEPT では、エージェントが行う処理を「**サービス**」として扱い、このサービスを組み合わせてエージェントを構成する**サービスコンポーネントモデル**というサービス指向のエージェントモデルを提案している。以下、このサービスコンポーネントモデルについて説明する。

2.1 サービス指向のエージェントモデル

サービスコンポーネントモデルでは、エージェントは、一つのエージェントフレームワークと複数のコンポーネントから構成される。コンポーネントは、一つ以上のサービスをサービス提供処理(INA/LI アーキテクチャにおけるサーブエージェントに相当)として実装し、エージェントフレームワークは、コンポーネントを組み込むことにより、エージェントとしてのサービス提供能力を得る(図 2)。

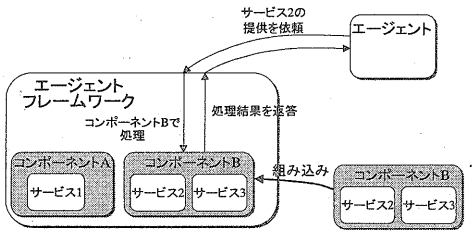


図2 サービスコンポーネントモデル

エージェントフレームワークは、通信環境の提供やコンポーネントの着脱機構、サービス提供依頼に基づいた処理の分配などの処理を行うコンポーネントマネージャとして動作する。各コンポーネントは、エージェント内部でスレッドとして動作し、エージェントフレームワークから渡されたサービス提供依頼を処理し、フレームワークに対して結果を返答する。

サービスコンポーネントモデルに基づいてエージェントを作成することにより、複数のエージェントを管理する必要がなくなり、エージェントフレームワークのみを管理するだけで、様々なサービスを実装したエージェントを利用することが可能となる。

2.2 支援階層の単一化

INA/LI アーキテクチャでは、エージェントの作業階層として、グループワーク支援層、パーソナルワーク支援層という 2 層が存在する。しかし、これがエージェント間の通信において複雑化を招いているのは、1 章で示した通りである。

よって、サービスコンポーネントモデルでは、明確にグループウェアエージェントやパーソナルエージェントといった区別をしないことにした。エージェン

[2] Agent Developer's Environment Practical Tools の略

トは全て、サービスコンポーネントモデルという同一のモデルで構成され、同一の通信手段で他のエージェントにアクセスする。

2.3 エージェントの支援対象

前節において、全てのエージェントをサービスコンポーネントモデルで同一に扱うとしたが、この場合、エージェントが支援する対象(人、組織、物、場所など)をどのように定義し、また、エージェントにどのようなサービスを持ったコンポーネントを組み込むかという問題が起こる。

前述したようにサービスコンポーネントモデルに基づいて作成されたエージェントでは、組み込んだコンポーネントで実装しているサービスによって、提供するサービス及び支援する対象が決定する。

しかし、エージェント内のコンポーネントが別々の支援対象を持つことを許せば、エージェント管理者の混乱を招くおそれがある。

よって、エージェントは基本的に、ある特定の対象を支援するコンポーネントで構成され、エージェントの名前も支援対象に関連する名前(人ならば人名、組織ならば組織名など)を持ち、エージェントに組み込まれたコンポーネントは、エージェントの支援対象に関するサービスを提供するものとみなす。

3. サービス名によるコンポーネントの管理方式

サービスコンポーネントモデルでは、エージェント内に複数のコンポーネントが組み込まれるため、各コンポーネントを管理する方法をエージェントフレームワークに実装する必要がある。

以下、エージェント開発環境ADEPT上でのサービスコンポーネントモデルに基づくエージェントの実装において、エージェントフレームワークがどのようにコンポーネントを管理し、そのサービスを提供するかについて述べる。

3.1 木構造によるサービスの管理

ADEPT 上で作成したコンポーネントが提供するサービスは全て、そのサービスを表す「**サービス名**」を持つ。他のエージェントからなされたサービス提供依頼も、このサービス名に基づいて提供するサービス及びそのサービスを提供しているコンポーネントを決定する。

例えば、ある会議調整に関するサービスについて考える。典型的な例として、このスケジュールサービスは、以下の3つのサービスを行うとする。

1. ある期間におけるユーザのスケジュール状況を報告する。
2. ある日時にスケジュール予約を入れる。
3. ある日時のスケジュールを取り消す。

サービスコンポーネントモデルでは、このそれぞれのサービスに対して木構造を持つサービス名を割り当てる。例えば、上記の3つのサービスにそれぞれ、「Report.Schedule.NEC」、「Reserve.Schedule.NEC」、「Cancel.Schedule.NEC」というサービス名をつけた場合、図3のような木構造になる。

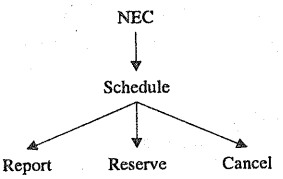


図3 サービス名の木構造

木構造中の各ノードには、サービス名の"."で区切られた文字列が対応する。木構造の最下部は、先頭の文字列が表すノードとなり、最上部は、最後尾の文字列が表すノードとなる。また、「Report.Schedule.NEC」と記述した場合には、NEC→Schedule→Reportという経路をもつReportノードを指す。

エージェントは、コンポーネントが持つ各サービス名を図3のような木構造に変換し、コンポーネント管理木という形で管理する。コンポーネント管理木の例を図4に示す。

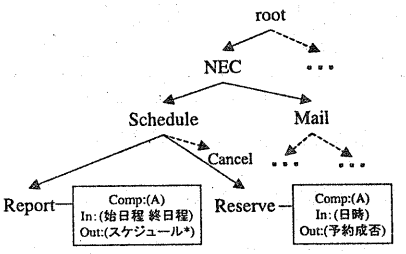
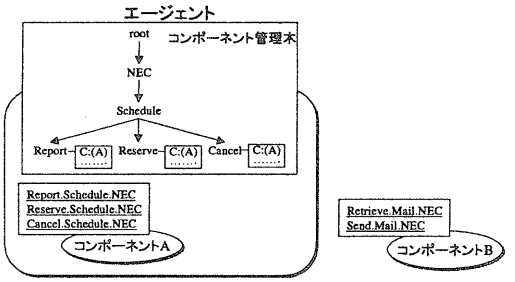


図4 コンポーネント管理木の例

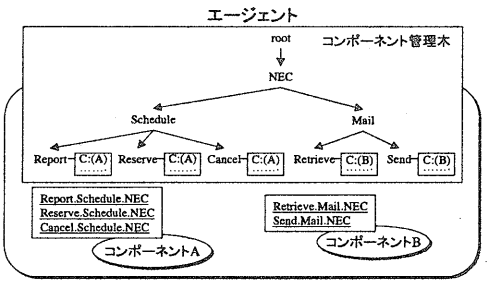
コンポーネント管理木では、全てのノードの根元としてrootノードがただ一つ存在するが、サービス名には現れない。また、ノードが示すサービスを提供

するコンポーネントがエージェントに組み込まれている場合、ノードに葉が付加され、コンポーネントへの参照(ポインタ)とサービス提供依頼を受け付けるときの入力メッセージの構成と、処理の結果送信される出力メッセージの構成とが登録される。図 4の例では、「Report. Schedule.NEC」というサービスに対して葉が存在し、その葉にはコンポーネント A への参照が設定され、入力のメッセージの構成として「始日程」と「終日程」を持ち、出力として「スケジュール」が 0 個以上返されることを示している。

このコンポーネント管理木は、各コンポーネントがエージェント内にロードされたときに接ぎ木の要領で作成される。例えば、図 5i)に示すコンポーネント管理木を保持しているエージェントに、コンポーネント B を組み込むと、コンポーネント管理木の NEC 以下のノードに Mail 以下のノードが接ぎ木され図 5ii)のようになる。



i) コンポーネントB組み込み前



ii) コンポーネントB組み込み後

図5 コンポーネントの組み込み

3.2 サービス名の競合

サービスコンポーネントモデルでは、コンポーネントの名前ではなく、サービス名を用いてコンポーネントを管理しているので、一つのサービス名に対して複数のコンポーネントがサービスを提供するという競合状態が発生する。この競合状態を解消するため

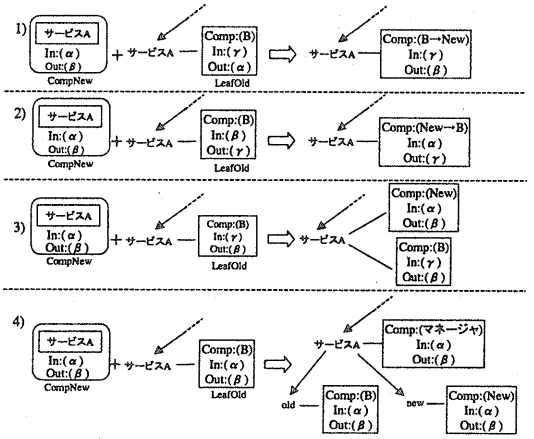


図6 サービス名の競合解消

に、ADEPT では以下にあげる4つの方法で競合の解消を行う。

1) 既存のコンポーネントの出力と新たに組み込んだコンポーネントの入力とを結合する。
(出力の加工)

2) 新たに組み込んだコンポーネントの出力と既存のコンポーネントの入力とを結合する。
(入力のプリプロセッサ)

3) 入力メッセージの構成で区別する。
(メッセージの種類による実行)

4) 競合しているコンポーネントを下位に持つマネージャコンポーネントを生成し、マネージャコンポーネントが状況に応じてコンポーネントの起動や出力メッセージの結合を行う。
(マネージャによる実行)

図 6に競合の解消方法の模式図を示す。図中では、新たに組み込んだコンポーネントを CompNew とし、競合が起こっているコンポーネント管理木中の葉を LeafOld としている。解消方法の選択は、前節のコンポーネント管理木に定義されている各ノードの葉の入力メッセージの構成及び出力メッセージの構成により判別される。

以上により、古いコンポーネントと新しいコンポーネントを同様のサービス名で使い分けたり、また、あるコンポーネントの出力を別のコンポーネントで加工し、フィルタリングを行ったり、また新たな情報を付加して返信するといった処理が可能となる。

3.3 サービスの処理依頼

ADEPT では、他のエージェントもしくは UI からの

サービス処理依頼は、通信メッセージの形で受信する。各通信メッセージは、以下のようなサービス名と通信内容のセットで構成される。

("Report.Schedule.NEC", "(from:'1998/12/17' to:'1998/12/18')")

図7 通信メッセージの例

メッセージの配信先は、受信したサービス名によって決定する。この時の配信先コンポーネントの決定アルゴリズムを図 8に示す。

```
procedure Search-Component(ct, st, mc)
 inputs: ct, コンポーネント管理木
         st, サービス名で示される木
         mc, メッセージの構成
 local variables: nodes , ノードの集合
                  leaf, ノードの葉

 nodes ← GET-NODE(ct, st)
 while nodesが空でない
    for each node in nodes do
       if nodeが葉を持つ then
          for each leaf in nodeの葉 do
             if leafの入力の構成とmcが一致 then
                SEND-MESSAGE(leaf)
          end
    end
    if 一回以上メッセージを配信した then
       return
    nodes ← GET-LOWER-NODES(nodes, 1)
 end
 REPLY-NO-SERVICE
```

図8 配信コンポーネント決定アルゴリズム

GET-NODE(*ct, st*)は、サービス名で示される経路を持つノードと同じ経路を持つコンポーネント管理木中のノードの集合(要素数は必ず 1 または 0)を返す。SEND-MESSAGE(*leaf*)は *leaf* に登録されているコンポーネントに対して、受信したメッセージを配信する。GET-LOWER-NODES(*nodes*, 1) は、*nodes* 中の各ノードの 1 階層下のノードの集合を返す。REPLY-NO-SERVICE は、受信したメッセージを処理するコンポーネントがないことを、メッセージの送信元に返信する。

このアルゴリズムは、配信コンポーネントの検索を横優先探索を行い、ある階層でコンポーネントにメッセージが送信されると、それ以上深くは検索しない。よってこのアルゴリズムは、木の深さを優先度としたメッセージ配信ポリシーを持つ。

このアルゴリズムでは、サービス名で一意に特定されるノード及び葉が存在する場合には、通常の 1 対 1 の通信になるが、それ以外に以下の配信方法が可能となる。

1) 曖昧なサービス名によるサービスの指定
   完全なサービス名を指定せずに、大まかなサービス名によりサービスを提供できる (3.4節にて後述)。
2) サービス競合時のマネージャによるコンポーネントの管理実行(図 6)
   深さを優先度としたメッセージ配信を行うため、マネージャより下位層のコンポーネントにはメッセージを配信せず、マネージャにそのサービスの処理を委託することができる。

3.4 サービス名の利用例

前節で述べた曖昧なサービス名によるサービス指定の利用例として、コンポーネント開発段階における、サービス名を用いたDebugモードとReleaseモードの使い分けについて述べる。

開発中のコンポーネントで仮に「B.A」というサービスを提供したいとする。この時、そのサービス名にRelease ノードを付加したサービス名(Release.B.A)と、Debug ノードを付加したサービス名(Debug.B.A)の二つをエージェントに登録するようにし、Debugサービスの入力メッセージの構成にデバッグ情報などの構成を付け加えておく。

このコンポーネントを普通に利用したいエージェントがコンポーネントを使用する際には、通常のメッセージの構成でメッセージを送信し、デバッグを行いたいエージェントがコンポーネントを利用する場合には、デバッグ情報を付加したメッセージを送信する。この時、指定するサービス名は、どちらも「B.A」を指定する。受信側のエージェントでは、受信したメッセージの構成に応じて、「Release.B.A」サービスもしくは「Debug.B.A」サービスの区別がされ、各モードでコンポーネントが呼び出される。

開発が終わった段階で、サービス名「B.A」のみを登録するように変更すれば、Debug に関するメッセージには、「指定されたサービスは使用できない」という旨を記したメッセージが自動で返信される。

以上の方法を用いることで、コンポーネントの機能を提供しながら、デバッグ、テストを行うといったことが可能となる。

4. ADEPT でのエージェントの実装

サービスコンポーネントモデルでは、エージェントはエージェントフレームワークとサービスコンポーネ

ントにより構成される。以下、ADEPT におけるエージェントフレームワークの実装について述べ、サービスコンポーネントの実装方法について説明する。

4.1 エージェントフレームワークの実装

ADEPT におけるエージェントフレームワークは大きく分けて、コンポーネントのロードやエージェント全体の構成を管理する Agent Composition Manager と、エージェントが提供するサービスを管理する Service Manager、エージェント間通信に関する処理を行う Agent Socket から成る。

エージェントが起動されると、Agent Composition Manager が、起動環境に応じて通信手段である Agent Socket を初期化する。次に、プロファイルに指定されたコンポーネントを Service Manager を通じてロードする。この時、Service Manager は、コンポーネントに記述されているサービス名を読み込み、図4に示したコンポーネント管理木に接ぎ木していく。

外部エージェントからのサービス提供依頼は、Agent Socket で受信し、Service Manager によってサービスを提供しているコンポーネントに渡す。

4.2 コンポーネントの実装

ADEPT におけるコンポーネントは、Agent Service Object(以下 ASO)という Java のオブジェクトで実装される。これらのオブジェクトは、AgentServiceObject クラスを継承したサブクラスのインスタンスである。

ASO で処理するサービスは、MessageListener クラスを継承したクラスで実装する。MessageListener クラスでは、そのクラスで提供するサービスのサービス名と実際に処理する処理ルーチンを実装する。

図 9に、簡単な ASO の例をあげる。

例では、ASO として Sample クラスを定義し、その中で"Hello.Sample"というサービス名を持つサービスを実装している HelloService クラスを定義している。この HelloService クラスでは、"Hello.Sample"というサービス依頼に関し、そのメッセージをコンソールに表示し、返信として"Hello! Thank you!"というメッセージを返すというサービスを提供する。

現在まだ入力および出力メッセージの構成をエージェントに受け渡す機能は実装できていない。

```
public class Sample extends AgentServiceObject{

 public class HelloService implements MessageListener{
  public String getServiceName(){
   return "Hello.Sample";
  }
  public Answer handleMessage(MessageEvent mes){
   System.out.println(mes.getMessage());
   return new Answer("Hello! Thank you!");
  }
 }

 public Sample(){
  addMessageListener(new HelloService());
 }
}
```

図9 ASO の実装例

5. おわりに

本稿では、エージェントモデルとしてサービスコンポーネントモデルを提案し、エージェント開発環境である ADEPT での実装方式について説明した。このサービスコンポーネントモデルを利用することにより、エージェントによるサービスの実装が簡素化され、エージェントのデバッグやテストが容易となる。

また、エージェント利用者にとっても、様々なサービスを提供するエージェントを容易に導入できるといった利点がある。

現在の ADEPT におけるエージェントの実装では、Java を使用している。また、今後の課題としては、以下があげられる。

- サービス実行権限や組み込み権限などのサービスアクセスポリシーの実装。
- コンポーネントの入出力メッセージの構成に基づく、サービスが競合したコンポーネント間の競合解決の自動化。
- 実行環境に適応した、コンポーネントの動的な組み込み。

今後これらの課題の検討、実装を行う。

参考文献

[1] Ishiguro,Y., Tarumi,H., Asakura,T., Kida,K., Kusui, D., Yoshifu,K., :「An Agent Architecture for Personal and Group Work Support」, ICMAS96, p134-141 (1996)

[2] 城島, 朝倉:「エージェント開発環境 ADEPT における通信基盤の試作」,情報処理学会第 56 回全国大会, 3J-05 (1998)

[3]Shovam,Y.:「Agent-oriented programming」, Artificial Intelligence 60, p51-92 (1993)

[4]藤田,菅原,木下,白鳥:「分散処理システムのエージェントアーキテクチャ」,情報処理学会論文誌, Vol37, No5

[5]菅原, 藤田, 木下, 白鳥:「ADIPS フレームワークとその応用(1)～(4)」, 情処第 56 回全国大会, 4J-6～9, (1998)