情報処理学会論文誌　数理モデル化と応用　Vol. 1　No. 1　160–174 (Sep. 2008)

WS-Notificationを基盤とするグリッド上のアプリケーション連携システム

下坂久司[†1,*1]　廣安知之[†2,*2]
三木光範[†2]　中尾昌広[†1]

近年，グリッド技術はWebサービス技術を基盤とした標準化が進められている．特に，Webサービスにおいて状態を持つリソースを定義できるWS-Resource Frameworkと，サービス間のメッセージ通知を実現するWS-Notificationは，グリッドに適したWebサービスの実現に必要不可欠である．本論文では科学技術計算用アプリケーションの連携に注目し，WS-Notificatitonを基盤とするグリッド上のアプリケーション連携システムを提案する．またGlobus Toolkitを用いて提案システムを実装し，これをApplication Igniting Systemと呼ぶ．構築したシステムでは，科学技術計算用のアプリケーションを容易にWebサービス化でき，サービス間のメッセージ通知によるアプリケーション実行の連鎖を実現する．さらに，エンドユーザによる柔軟なアプリケーション連携の設計を支援する複数のサービスを提供する．本論文における数値実験では，構築したシステムをバイオインフォマティクス分野のアプリケーション連携に適用し，基本性能を評価した．

Application Integration System on the Grid Based on WS-Notification

Hisashi Shimosaka,[†1,*1] Tomoyuki Hiroyasu,[†2,*2]
Mitsunori Miki[†2] and Masahiro Nakao[†1]

Recently, Grid technologies have been standardized based on Web service technologies. Of these technologies, the WS-Resource Framework and WS-Notification are indispensable to construct Grid-enabled Web services. The WS-Resource Framework enables the definition of resource with state information and the WS-Notification realizes message notification driven by state change. This paper focuses on integration in scientific applications. We propose a new application integration system on the Grid based on WS-Notification. The proposed system, called “Application Igniting System”, is implemented using the Globus Toolkit. In this system, scientific applications can be published easily as Web services and the chain of application invocation is realized using message notification. In addition, this system provides services, which supports the design of flexible application integration. In the numerical examples of this paper, we apply the Application Igniting System to application integration in the bioinformatics field and evaluate its basic performance.

1. はじめに

自動車や航空機の設計，バイオインフォマティクスなどの複合領域にわたる問題解決では，高性能な各種アプリケーションを効率良く複数つなぎ合わせて利用する必要がある．一方で，これらのアプリケーションはそれぞれ別の研究者や研究機関によって開発されていることが多い．また，科学技術計算用のアプリケーションの連携を対象とした場合，アプリケーションの実行には大規模な計算資源や専用の装置，データベースを必要とすることが多く，さらにライセンスの問題などもあるために，利用したいすべてのアプリケーションを1カ所に集めて利用することは難しい．これらから，アプリケーションの開発者や所有者は広域ネットワーク上の計算資源や情報資源上にアプリケーションを配置し，問題解決を行いたいアプリケーション利用者（エンドユーザ）が必要に応じてそれらを連携させて利用するというアプローチが有効であると考えられる．しかしながら，広域ネットワークを利用したシステムの構築にはユーザの認証や認可，シングルサインオン，通信の暗号化，資源の発見と効率的な利用，通信障害の検知やその対応といった様々な問題を解決する必要がある．

広域ネットワークを利用するシステムの構築には，分散して配置された資源を仮想的に統合して利用するための基盤技術であるグリッドの利用が，システムの実用性や開発コストを考慮した場合，必要不可欠である．最近では，Global Grid Forum（GGF）においてOpen Grid Services Architecture（OGSA）[1)]が提案されており，Webサービス技術を基盤としたグリッド技術の標準化が進められている．これにより，今後は多くの科学技術計算

†1 同志社大学大学院工学研究科
Graduate School of Engineering, Doshisha University
†2 同志社大学工学部
Department of Knowledge Engineering, Doshisha University
∗1 現在，ピーシーアシスト株式会社
Presently with PC Assist Co., Ltd.
∗2 現在，同志社大学生命医科学部
Presently with Department of Life and Medical Sciences, Doshisha University

用アプリケーションが Web サービス化され，広域ネットワーク上に点在して配置されることが期待できる．しかしながら，どのようにしてアプリケーションを Web サービス化するのか，またどのように統合利用するのかという点に関しては，深く検討する必要がある．

Web サービスを用いたアプリケーションの統合利用に関する取り組みは，ビジネス分野において数多く行われている[2]．一方で，科学技術計算用のアプリケーションに関しては，アプリケーションの実行環境やジョブの多様性，大規模なデータセットの取扱いを考慮する必要があり，標準的な枠組みはない．そのため，Web サービスを用いて科学技術計算用アプリケーションの統合利用を支援できるシステムの開発は，非常に重要であると考えられる．

本研究では，グリッド上で科学技術計算用アプリケーションを複数つなぎ合わせることにより新しいシステムを構築でき，問題解決を行える基盤システムの開発を目標とする．アプリケーションを複数つなぎ合わせて利用することを，本研究ではアプリケーション連携と呼ぶ．このようなグリッド上のアプリケーション連携を実現するために，本研究で構築するシステムは既存アプリケーションを Web サービスとして取り扱う．また Web サービスにおいてアプリケーションの実行状態を取り扱うことで，状態変化により駆動するメッセージ通知を用いたアプリケーション実行の連鎖を実現する．さらに，エンドユーザによる柔軟なアプリケーション連携の設計を可能にする複数のサービスを提供する．本研究では，構築したシステムをバイオインフォマティクス分野で標準的に用いられるアプリケーション連携に適用し，性能を評価した．

2. Open Grid Services Architecture

近年，GGF においてビジネス分野におけるグリッドの利用促進や，グリッドミドルウェア間の相互運用性の確保などを目的として，OGSA の標準化が進められている．OGSA は WS-Resource Framework（WSRF）[4] と WS-Notification（WSN）[5] の 2 つの仕様に代表される Web サービス技術を基盤としている．

2.1 WS-Resource Framework

WSRF は，Web サービスに状態を有するリソースを導入するための仕様である．リソースはリソースプロパティのセットにより構成される．WSRF におけるリソースの取扱いでは，ファクトリパターンがよく用いられる．ファクトリパターンでは，まずリソースを生成する専用の Web サービス（ファクトリサービス）を用意しておく．ファクトリサービスは，リソース生成要求ごとにリソースの生成と初期化を行い，リソースにアクセスするための情報を返信する．この情報は，エンドポイントリファレンス（Endpoint Reference: EPR）と呼ばれる．EPR には，生成したリソースとリソースにアクセスできる別の Web サービスの組合せ情報が記述されている．生成したリソースには，以後 EPR に含まれている Web サービスのポートタイプを通じてアクセスする．WSRF では，Web サービス自身は状態を持たず，バックエンドに存在するリソースにおいて状態を保持する．フロントエンドとなる Web サービスは，Web サービスクライアントから与えられるリソース情報をもとに，何らかの処理を行ってリソースを更新するための機能を提供する．一般に，フロントエンドとなる 1 つの Web サービスに対し複数のリソースがバックエンドに存在する．また，EPR で一意に識別される Web サービスとリソースの組合せが操作や処理の基本単位となる．

ここで，あるアプリケーション A を実行する Web サービス（サービス A）を想定する．WSRF において，アプリケーション A を実行したいエンドユーザなどの Web サービスクライアントは，まず専用のファクトリサービス（ファクトリサービス A）に対してリソースの生成を要求する．ファクトリサービス A はリソースを生成し，アプリケーションを実行するための初期化処理を行う．また，アプリケーションの実行状態を保持するリソースプロパティの値を待機状態（Pending）にセットする．そして，生成したリソースとサービス A を組み合わせた情報（EPR）を Web サービスクライアントに返信する．その後，Web サービスクライアントは EPR に含まれるサービス A のポートタイプを通じてアプリケーション A の実行を要求する．アプリケーションの実行要求を受け，サービス A はまず Web サービスクライアントが生成したリソースを特定する．その後，アプリケーション A を実行すると同時に，アプリケーションの実行状態を保持するリソースプロパティの値を待機状態（Pending）から実行中（Active）に更新する．最後に，サービス A はアプリケーション実行が正常に終了した場合にはリソースプロパティの値を正常終了（Finished）に，反対にアプリケーション実行に失敗した場合には異常終了（Failed）に更新する．

このように，WSRF を用いることで，Web サービス実行中の動的な状態変化を表現することが可能になる．Web サービスクライアントは，ファクトリサービスにより生成されたリソースの内容を確認することで，要求した処理が現在どのような状態にあるのかを知ることができる．

2.2 WS-Notification

状態を有するリソースを導入した Web サービスにおいて，状態変化を扱う操作は非常に重要である．WSN は，状態変化により駆動する登録型の非同期メッセージ通知の枠組みである．**図 1** に最も単純な WSN の概要を示す．WSN では，メッセージの送り手を NotificationProducer，メッセージの受け手を NotificationConsumer と呼ぶ．NotificationProducer お

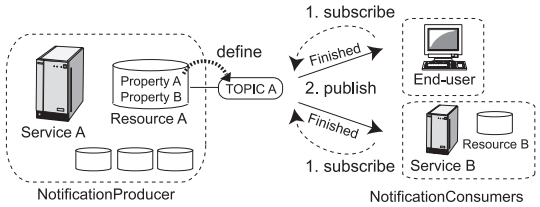


図 1　WS-Notification の概要
Fig. 1　Overview of WS-Notification framework.

よび NotificationConsumer は，ともに EPR で一意に識別される Web サービスとリソースの組合せが基本単位となる．また，エンドユーザは NotificationConsumer として振る舞うことも可能である．

WSN において，NotificationProducer はあらかじめ通知可能なリソースプロパティをトピックとして定義する．一般に，リソースの初期化時にトピックの定義が行われる．その後，NotificationConsumer は興味のあるリソースプロパティを保持する NotificationProducer に対してそれぞれ通知予約（subscribe）を行う．これにより，NotificationProducer においてリソースプロパティに変更が加えられた際には，NotificationConsumer に対し状態の詳細（リソースプロパティの内容）が記述されたメッセージが通知（publish）される．NotificationConsumer はメッセージの内容を確認し，内容に応じた処理を連鎖的に実行することができる．たとえば，あるアプリケーション A を実行する Web サービス（サービス A）のリソース A からメッセージ Finished を受け取ることにより，エンドユーザはアプリケーション実行が正常終了したことが分かる．そして，アプリケーションの出力ファイルを取得するという処理を連鎖的に開始することができる．また別の Web サービス（サービス B）では，リソース B の情報を用いてアプリケーション B を連鎖的に実行することも可能となる．

このように，WSN は WSRF で導入されたリソース間（EPR 間）において，リソースプロパティの更新により駆動するメッセージ通知を実現する．また，NotificationConsumer はメッセージ通知を受け取ることにより任意の処理を連鎖的に開始することができる．

3. Application Igniting System

本研究では，WS-Notification を基盤としたグリッド上のアプリケーション連携システムを提案する．また提案するシステムを Globus Toolkit（Globus）[3] を用いて実装する．実装したシステムでは，ある 1 つの状態変化を起点として Web サービス間で次々とメッセージが通知され，複数のアプリケーションが様々な形で連鎖して実行される．これらはアプリケーション実行という火が，グリッド上で燃え広がるようにとらえられるため，本研究では実装したシステムを Application Igniting System と呼ぶ．本章では，まずアプリケーション連携システムの必要条件について述べ，Application Igniting System の概要について述べる．その後，提案システムの詳細について述べる．

3.1　アプリケーション連携システムの必要条件

科学技術計算用アプリケーションの連携では，大規模なデータセットの取扱いと多様なアプリケーション実行環境を考慮する必要がある．

大規模なデータセットを Web サービスで取り扱うことを考慮すると，Web サービスのポートタイプをアプリケーションの入出力と直接結び付けることは困難である．これは，一般に Web サービスは XML 文書によって様々な処理要求を受付けたり，Web サービスクライアント間との情報交換を行ったりしていることに起因する．そのため，大規模なデータセットを Web サービス間でやりとりする場合には，ファイル転送のための専用の Web サービスを用意し，ファイルの転送元や転送先などの情報のみをポートタイプを通じて提供する．そして，実際のファイル転送は FTP などのファイル転送に特化した専用のプロトコル，ツールを用いて，ファイル転送用の Web サービスの内部でその処理を実現することが現実的であると考えられる．

多様なアプリケーション実行環境の取扱いにおいても，高機能なアプリケーション実行用の Web サービスを介して，アプリケーションを起動することが望まれる．そのため，アプリケーション連携システムはアプリケーション所有者に対して，アプリケーション起動用の情報を容易に登録できる機能を提供し，さらに登録されたアプリケーション情報を用いて，アプリケーション実行用の Web サービスにアプリケーション起動を依頼できる機能が必要不可欠であるといえる．

一方で，アプリケーション利用者による任意のアプリケーション連携の設計を可能にするために，アプリケーション連携システムは登録されたアプリケーション情報を情報収集サービスにおいて一元管理し，アプリケーション利用者に提供する必要がある．また，アプリケーション実行順序とアプリケーション間の情報交換を柔軟に設計できる機能の提供が不可欠である．アプリケーション実行順序の設計では，アプリケーションの逐次実行だけでなく，並列実行や特定の連携の繰返し実行，アプリケーションの実行状態を考慮した処理の分岐などをサポートすることが望ましい．またアプリケーション間の情報交換の設計では，単

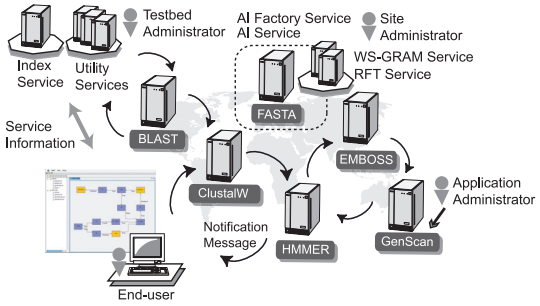


図 2　Application Igniting System の概要
Fig. 2　Overview of the Application Igniting System.

純な入出力ファイルの交換だけでなく，交換される入出力ファイルの書式が異なる場合などにおいては，ファイルの変換方法もあわせて指定できることが望まれる．

そのほかにも，負荷分散を考慮した場合，集中管理によってアプリケーション実行やファイル交換を制御するのではなく，Web サービス間の分散管理が望ましい．また障害が発生した場合には，その検知や対応を行える機能が必要不可欠である．本研究では 2 章で述べた WSRF と WSN を用いて，これらの必要条件を満たすための各機能を提供する．

3.2　Application Igniting System の概要

Application Igniting System は，WSN を基盤とするアプリケーション連携システム（ワークフロー管理システム）である．アプリケーション連携を，以後ワークフローと呼ぶ．システムの実装には，WSRF と WSN の参照実装である Globus（バージョン 4.0.1）を用いた．また，Web サービスのリソース間を流れるメッセージ通知を利用して集中管理機構を有さないシステム構成を採用した．これにより，Web サービス間の粗な結合を実現し，メッセージ通知を取り扱うユーティリティサービスを追加することで容易に機能拡張を行えることを可能にした．

Application Igniting System の概要を**図 2** に示す．このシステムでは，まずグリッドテストベッド管理者により Index サービスが用意される．Index サービスは Globus により提供される Web サービスであり，情報収集用の機能を提供する．またグリッドテストベッドの管理者は，Application Igniting System により提供される各種ユーティリティサービスを導入する．これらユーティリティサービスは，Web サービスのリソース間でやりとりされるメッセージ通知を操作する機能を提供し，エンドユーザによる柔軟なワークフロー設計を支援する．各サイトの管理者は，自身の管理するサイトにおいて Application Igniting System の提供する AI ファクトリサービス（Application Igniting Factory Service）および AI サービス（Application Igniting Service）を導入する．また Globus Toolkit の提供する WS-GRAM サービス（Grid Resource Allocation and Management Service）と RFT サービス（Reliable File Transfer Service）を用意する．WS-GRAM サービスは高度なアプリケーション実行用の機能を提供する．また，RFT サービスは信頼性の高いファイル転送用の機能を提供する．両サービスは AI サービスにより利用される．アプリケーション所有者は，WS-GRAM サービスにジョブを投入するための XML 言語である RSL（Resource Specification Language）を用いてアプリケーションの起動情報を記述し，AI ファクトリサービスに登録する．登録されたアプリケーション情報は，すべて Index サービスに集められ，エンドユーザに提供される．エンドユーザは利用したいアプリケーションを複数選択し，ユーティリティサービスを利用してワークフローを設計する．設計されたワークフローは，各サイトの AI サービスおよびユーティリティサービスのリソース間を流れるメッセージ通知により実現される．

以降では，まず最初にアプリケーション所有者によるアプリケーションの登録方法について述べる．その後，エンドユーザによるワークフロー設計について述べ，最後に設計されたワークフローを Application Igniting System がどのように実行するかについて述べる．

3.3　RSL を用いたアプリケーションの登録

Application Igniting System の概要で述べたように，アプリケーション所有者は WS-GRAM サービスにジョブをリクエストするための RSL を用いてアプリケーションの起動情報を記述し，AI ファクトリサービスに登録する．**図 3** に，アプリケーション所有者が記述する RSL ファイルの例を示す．紙面の都合上，一部の情報を省略している．この例で記述されたアプリケーションは，AI ファクトリサービスと同じホスト上に配置されており，同ホスト上の WS-GRAM サービスによって起動される．一方で，RSL の factoryEndpoint 要素を用いてアプリケーション起動用の WS-GRAM サービスを指定することで，AI ファクトリサービスとは別サイトにあるアプリケーションを起動したり，ジョブスケジューラを経由したアプリケーションを起動したりすることも可能である．

一般に，RSL ファイルはアプリケーションの起動に必要な情報を用意したエンドユーザにより記述される．Application Igniting System では，入力ファイルのパス情報などのアプリケーション起動に必要な情報の一部をシステム自身が扱うために，RSL ファイルを記述するアプリケーション所有者に対して必要な情報を提供する必要がある．図 3 に記述さ

```
<?xml version="1.0" encoding="UTF-8"?>
<job>
  <executable>/opt/blast/bin/blastall</executable>
  <directory>${AIS_JOB_HOME}</directory>
  <argument>-p</argument>
  <argument>blastp</argument>
  <argument>-d</argument>
  <argument>nr</argument>
  <argument>-i</argument>
  <argument>${AIS_INPUT_FILE_PATH#0}</argument>
  <argument>-o</argument>
  <argument>${AIS_OUTPUT_FILE_PATH#0}</argument>
  <stdout>stdout</stdout>
  <stderr>stderr</stderr>

  <extensions>
    <aisExtension> Application Description ... </aisExtension>
  </extensions>
</job>
```

図 3　RSL ファイルの例
Fig. 3　RSL file example.

表 1　代表的な AIS 変数
Table 1　AIS variable descriptions.

Variable	Description
AIS_JOB_HOME	リソース固有のディレクトリ
AIS_INPUT_FILE_PATH	入力ファイルパス (#0-99)
AIS_OUTPUT_FILE_PATH	出力ファイルパス (#0-99)
AIS_INPUT_DIRECTORY	入力ファイル用ディレクトリ
AIS_OUTPUT_DIRECTORY	出力ファイル用ディレクトリ
AIS_HOST	ホスト名
AIS_GSIFTP_PORT	GridFTP のポート番号
AIS_RESOURCE_KEY	リソース固有の番号

れている「AIS_ 」で始まる各変数は，システムが提供する情報を表現しており，約 20 個程度用意されている．**表 1** に代表的な変数の説明を示す．アプリケーション所有者は，これらの変数を用いて入力ファイルのパス情報などをシステムから取得する．また，アプリケーションの出力ファイルをシステムが定めるパス上に配置することで，他のアプリケーションとの入出力ファイル交換を実現する．

さらに，RSL の extensions 要素内においてシステムが独自に定義した aisExtension 要素を用い，アプリケーションの名前や説明といった追加的な情報を記述することができる．

```
<?xml version="1.0" encoding="UTF-8"?>
<ais>
  <services>
    <application>
      <factory>https://.../ApplicationIgnitingFactoryService</factory>
      <name>blastall</name>
      <id>ais.blastall</id>
    </application>
    ...
  </services>
  <subscriptions>
    <subscriptionRequest>
      <producerId>ais.if.start</producerId>
      <consumerId>ais.blastall</consumerId>
    </subscriptionRequest>
    ...
  <subscriptions>
  <transfers>
    <stageIn>...</stageIn>
    <stageOut>...</stageOut>
    <stageInRequest>
      <sourceId>ais.blastall</sourceId>
      <sourceFileNumber>0</sourceFileNumber>
      <destinationId>ais.editor</destinationId>...
      <destinationFileNumber>0</destinationFileNumber>
    </stageInRequest>
    ...
  </transfers>
  <invocation>...</invocation>
  <termination>...</termination>
</ais>
```

図 4　AIDL ファイルの例
Fig. 4　AIDL file example.

3.4　ワークフロー設計とユーザインタフェース

Application Igniting System において，エンドユーザはシステムが独自に定義したアプリケーション連携記述言語（Application Integration Description Language: AIDL）を用いてワークフローを設計する．エンドユーザが記述する AIDL ファイルの例を**図 4** に示す．紙面の都合上，一部の情報を省略している．AIDL は ais 要素をルートとする XML 言語であり，services 要素，subscriptions 要素，transfers 要素などで構成される．

AIDL ファイルの記述は，一般のエンドユーザにとって負担の大きいものと考えられる．そのため，現在まだ試作品ではあるが，**図 5** に示すクライアントツールを実装した．このツールでは，まず左側のペインにエンドユーザの用意したファイル群のリストが表示される．ファイル群のリストには，ワークフローを構成するアプリケーション群の中の一部のアプリケーションの入力ファイル，後述するユーティリティサービス（Editor サービス）の

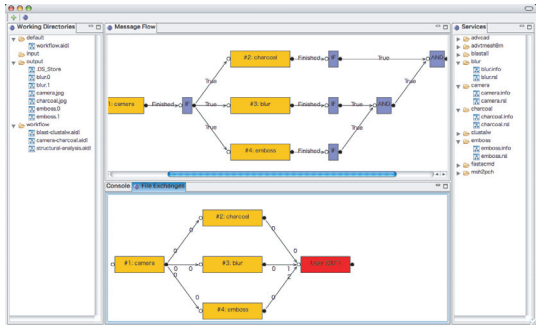


図 5　Application Igniting System のリッチクライアント
Fig. 5　Rich client for the Application Igniting System.

入力ファイルなどが含まれる．また，右側のペインに Index サービスより提供されたアプリケーション群の情報が表示される．エンドユーザは，まず最初に利用したいアプリケーションを複数選択する．これは AIDL における services 要素に対応する．その後，中央の 2 つのペインにおいて GUI 上の操作によりワークフローを設計する．ワークフロー設計では，まず中央上部のペインにおいてユーティリティサービスを利用し，選択したアプリケーション間のメッセージ通知の流れを設計する．これは AIDL における subscriptions 要素に対応し，アプリケーションの実行順序を決定する．次に，エンドユーザは中央下部のペインにおいてエンドユーザおよびアプリケーション間の入出力ファイル交換を定義する．入出力ファイル交換の定義は，AIDL における transfers 要素に対応する．この際，ユーティリティサービスにより交換されるファイルの変換もサポートされる．

エンドユーザにより設計されたワークフローは，ワークフローエンジンにより解釈され，実行される．ワークフローエンジンは，まず最初に AI サービスおよびユーティリティサービスのリソースを生成する．その後，エンドユーザにより設計されたメッセージ通知の流れに従い，生成したリソース間の通知予約を指示する．これにより，アプリケーションの実行順序が決定される．また，エンドユーザの保持するアプリケーションの入力ファイルの転送とアプリケーション間の入出力ファイル交換を指示する．最後に，ワークフロー実行を開始するために，ワークフローの先頭のリソースに対し，リソースを保持する Web サービスのポートタイプを通じて最初のメッセージ通知の送信を指示する．

以降では，まず最初にアプリケーション所有者により登録されたアプリケーションを起動する AI サービスの詳細について述べる．その後，ユーティリティサービスの詳細とメッセージ通知を基盤としたワークフロー実行について述べる．

3.5　リソースの生成とポートタイプ

エンドユーザによるワークフロー設計で述べたように，エンドユーザはまず最初に Index サービスより提供されたアプリケーション群の情報を参照し，利用したいアプリケーションを複数選択する．エンドユーザのアプリケーション選択に従い，ワークフローエンジンは各アプリケーション情報を保持する AI ファクトリサービスに対してリソースの生成を要求する．AI ファクトリサービスは，登録されたアプリケーション情報に基づくリソースを生成した後に，リソースにアクセスするための情報をワークフローエンジンに返信する．リソースにアクセスするための情報には，生成したリソースと AI サービスを組み合わせた情報（EPR）が含まれている．生成されたリソースには，EPR に含まれる AI サービスのポートタイプを通じてアクセスできる．ワークフローエンジンは，返信された情報をもとに，AI サービスのポートタイプを利用してリソース間（EPR 間）のメッセージ通知と入出力ファイル交換を設計する．AI サービスが提供する代表的なポートタイプとその機能は以下のとおりである．すべての機能と付随する操作は，EPR で一意に識別される Web サービスとリソースの組合せを対象とする．また，AI サービスの各リソースはアプリケーションの実行状態を取り扱うリソースプロパティを保持しており，「APPLICATION INVOCATION TOPIC（トピック AIT）」として定義している．アプリケーションの実行状態には正常終了を表す Finished，異常終了を表す Failed などの値が代入される．さらに，同様の仕組みを用いて，ワークフローエンジンはユーティリティサービスのリソースも生成する．EPR には，生成されたリソースとリソースにアクセスできるユーティリティサービスを組み合わせた情報が含まれる．いくつかのユーティリティサービスの各リソースは，条件の真偽値を取り扱うリソースプロパティを保持しており，「CONDITIONAL BRANCH TOPIC（トピック CBT）」として定義している．条件の真偽値には True もしくは False の値が代入される．

- setSubscription：メッセージ通知元（NotificationProducer）のトピック CBT に通知予約する．引数ではユーティリティサービスの EPR を指定し，これがメッセージ通知元に対応する．
- createTransferResource：入出力ファイルを送受信するための RFT サービスのリソースを生成し，生成したリソースと RFT サービスの組合せ情報（EPR）を返信する．呼び出し側は返信された EPR を利用し，EPR に含まれる RFT サービスのポートタイプを通じてファイル転送を実現する．引数には，送受信したい入出力ファイルの組合せを指定する．

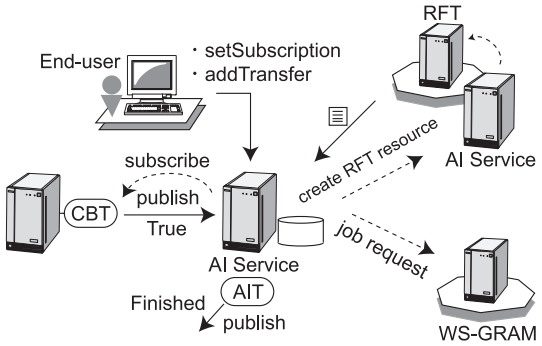


図 6　アプリケーションの起動手順
Fig. 6　Application invocation sequence.

- addTransfer：他のアプリケーションの出力ファイルを自身のアプリケーション実行前に取得するファイルのリストに追加する．取得したファイルはアプリケーションの入力ファイルとして利用される．引数には，出力ファイルを生成する AI サービスの EPR と入出力ファイルの組合せを指定する．

3.6　メッセージ通知によるアプリケーション実行

AI サービスにおけるアプリケーション起動手順を**図 6** に示す．ワークフローエンジンは，あらかじめエンドユーザにより設計されたワークフローに従い，AI サービスの setSubscription ポートタイプを利用して AI サービスの各リソースにメッセージ通知元（ユーティリティサービスの EPR）への通知予約を指示しておく．また AI サービスの addTransfer ポートタイプを利用し，各リソースに含まれるアプリケーション情報ごとにアプリケーションの入力ファイルとなる他のアプリケーションの出力ファイルを指定する．その後，AI サービスはメッセージ通知元からメッセージ True を受け取ることで，対象となるリソースを特定し，リソースに含まれるアプリケーション情報を利用してアプリケーション起動を開始する．アプリケーションの起動は次の手順により構成される．まず，AI サービスは起動対象となるアプリケーションのために，ワークフローエンジンにより指定された他のアプリケーションの出力ファイルを取得する．これは，addTransfer ポートタイプの引数で指定された EPR と入出力ファイルの組合せ情報を利用し，EPR に含まれる AI サービスの createTransferResource ポートタイプを利用して実現される．すべてのファイル転送終了後，アプリケーション情報に含まれている RSL ファイルの変数を表 1 で述べた値に置換し，WS-GRAM サービスにアプリケーション実行をリクエストする．アプリケーション実行が正常に終了した場合，アプリケーションの実行状態を取り扱うリソースプロパティの値を Finished に，異常終了した場合は Failed に更新する．この状態変化により，新たなメッセージ通知が別のリソースに送信される．

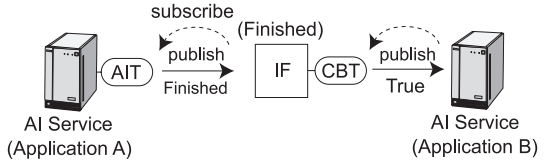


図 7　ワークフローの逐次実行
Fig. 7　Sequential invocation.

3.7　ユーティリティサービスとワークフロー実行

Application Igniting System は，Web サービスのリソース間（EPR 間）を流れるメッセージ通知を操作し，柔軟で粗結合なワークフロー制御を実現する複数のユーティリティサービスを提供する．現在，ユーティリティサービスとして，IF サービス，AND サービス，OR サービス，LOOP サービス，Editor サービスの 5 つを提供しており，ワークフローの逐次実行，並列実行，繰返し実行，および，アプリケーション間で交換される入出力ファイルの変換機能を提供する．

3.7.1　ワークフローの逐次実行と条件分岐

ワークフローの設計と実行を支援する最も重要なサービスは IF サービスである．IF サービスは，AI サービスと同様にメッセージ通知元のトピック CBT に通知予約を行えるポートタイプを持つ．一方で，AI サービスと異なり，トピック AIT に通知予約を行えるポートタイプも保持する．引数ではメッセージ通知元の EPR を指定する．さらに，この引数にはメッセージ通知の条件値を含めることができる．IF サービスのリソースには，条件の真偽値を取り扱うリソースプロパティが含まれており，自身のリソースに通知されたメッセージが条件値と一致する場合に，リソースプロパティの値を True に更新する．また一致しない場合には False に更新する．この状態変化により新たなメッセージ通知が送信される．条件の真偽値を取り扱うリソースプロパティは，トピック CBT として定義されている．

IF サービスを用いたワークフローの逐次実行例を**図 7** に示す．この例では，アプリケーション A，B が連続して実行される．図 7 において，ワークフローエンジンはあらかじめ 3 つのリソース（IF サービスとアプリケーション A，B の情報を保持する AI サービスの

リソース）を生成する．その後，IF サービスの EPR とポートタイプを用いて，IF サービスのリソースからアプリケーション A の情報を保持する AI サービスのリソースに対し通知予約を行うよう指示する．その際，条件値として Finished を指定しておく．また，アプリケーション B に対応する AI サービスの EPR と setSubscription ポートタイプを用いて，アプリケーション B の情報を保持する AI サービスのリソースから IF サービスのリソースへ通知予約を行うよう指示する．これにより，アプリケーション A の実行が正常終了して IF サービスのリソースにメッセージ Finished が通知されると，連鎖的に IF サービスのリソースからアプリケーション B の情報を保持する AI サービスのリソースに対してメッセージ True が通知される．メッセージ True の通知により，アプリケーション B が実行される．一方で，アプリケーション A の実行が異常終了した場合には，IF サービスのリソースにメッセージ Failed が，アプリケーション B の情報を保持する AI サービスのリソースに対してメッセージ False が通知される．そのため，アプリケーション B は実行されない．ここで，IF サービスのリソースに対して指定する条件値を Failed とすることで，アプリケーション A の実行が異常終了した場合にアプリケーション B を実行することもできる．このように，IF サービスはメッセージ内容の変換を図る機能を提供しており，ワークフロー設計において重要な役割を担う．

3.7.2　ワークフローの並列実行と同期処理

メッセージ通知は通知予約を行った NotificationConsumer すべてに送信されるために，AI サービスの複数のリソースに同時にメッセージ True を通知することで，アプリケーションの並列実行，ならびにワークフローの並列実行を実現できる．一方で，並列に実行しているアプリケーション，もしくはワークフローの同期をとって，次のアプリケーションを実行できる機能が不可欠である．このような同期処理を実現するために，AND サービスが提供される．AND サービスは，メッセージ通知元のトピック CBT に通知予約を行えるポートタイプを持つ．引数にはメッセージ通知元の EPR を指定する．AI サービスや IF サービスと異なり，AND サービスは 2 つのメッセージ通知元（EPR）に通知予約を行える特徴を持つ．AND サービスのリソースには，条件の真偽値を取り扱うリソースプロパティが含まれており，トピック CBT として定義されている．AND サービスは，自身のリソースに通知されたメッセージの値が True の場合に，メッセージ送信元の EPR を記録する．そして，2 つの EPR の両方からメッセージ True を受け取った時点で，リソースプロパティの値を True に更新する．この状態変化により新たなメッセージ通知が送信される．

AND サービスを用いたワークフローの並列実行例を**図 8** に示す．この例では，アプリケーション A，B が並列に実行され，両アプリケーションの正常終了後にアプリケーション C が実行される．図 8 において，ワークフローエンジンはあらかじめ 7 つのリソース（AND サービスと 3 つの IF サービスのリソース，ならびにアプリケーション A，B，C の情報を保持する AI サービスのリソース）を生成する．その後，図 8 に示す矢印の逆向きにリソース間（EPR 間）の通知予約を指示する．IF サービスのリソース B，C に対しては，条件値 Finished をそれぞれ指定しておく．これにより，IF サービスのリソース A（IF* に対応する EPR）においてリソースプロパティの値が True に更新されることで，アプリケーション A，B の情報を保持する AI サービスの 2 つのリソースに同時にメッセージ True が通知される．メッセージ True の通知により，アプリケーション A，B が並列に実行される．また，両アプリケーションの正常終了時に AND サービスのリソースに 2 つのメッセージ True が通知される．AND サービスはメッセージ通知の同期をとってリソースプロパティの値を True に更新する．これにより，アプリケーション C の情報を保持する AI サービスのリソースにメッセージ True が通知され，アプリケーション C の実行が開始される．

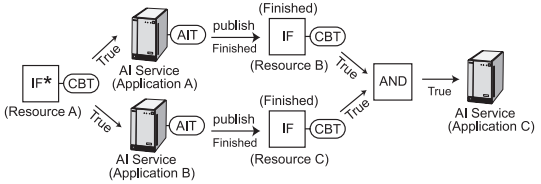


図 8　ワークフローの並列実行
Fig. 8　Parallel invocation.

3.7.3　ワークフローの繰返し実行

ワークフローの繰返し実行をサポートするために，OR サービスおよび LOOP サービスが提供される．両サービスのリソースには，条件の真偽値を取り扱うリソースプロパティが含まれており，トピック CBT として定義されている．OR サービスは，AND サービスと同様に 2 つのメッセージ通知元に通知予約を行えるポートタイプを持つ．しかしながら，OR サービスはメッセージ通知の同期をとらず，いずれかからメッセージ True を受け取ることでリソースプロパティの値を True に更新する．LOOP サービスは，OR サービスを補助的に用いてワークフローの繰返し実行を実現する．LOOP サービスは，1 つのメッセージ通知元のトピック CBT に通知予約を行えるポートタイプを保持し，メッセージ True を待つ．LOOP サービスは，メッセージ True を受け取った回数をカウントすることができ，

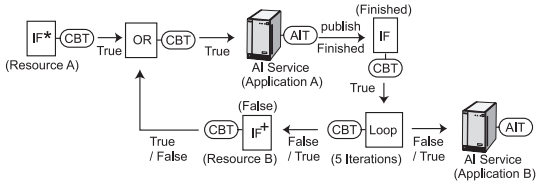


図 9　ワークフローの繰返し実行
Fig. 9　Loop invocation.

エンドユーザによりあらかじめ指定された回数に満たない場合にはリソースプロパティの値を False に，同数の場合には True に更新する．

OR サービスおよび LOOP サービスを用いたワークフローの繰返し実行例を**図 9** に示す．この例では，アプリケーション A が 5 回繰り返して実行され，その後アプリケーション B が実行される．図 9 において，ワークフローエンジンはあらかじめ 7 つのリソース（OR，LOOP サービスと 3 つの IF サービスのリソース，ならびにアプリケーション A，B の情報を保持する AI サービスのリソース）を生成する．その後，図 9 に示す矢印の逆向きにリソース間（EPR 間）の通知予約を指示する．IF サービスのリソースに対しては，条件値として Finished もしくは False を指定しておく．また，LOOP サービスのリソースに対しては繰返し回数 5 を指示しておく．ここで，OR サービスのリソースは IF サービスのリソース A（IF^* に対応する EPR）とリソース B（IF^+ 対応する EPR）に通知予約を行っている．そのため，いずれかのリソースからメッセージ True を受け取ることで，OR サービスのリソースにおいてリソースプロパティの値が True に更新される．これにより，アプリケーション A の情報を保持する AI サービスのリソースにメッセージ True が通知され，アプリケーション A が実行される．アプリケーション A の実行が正常に終了した場合には，LOOP サービスのリソースにメッセージ True が通知される．LOOP サービスは，メッセージ True を受け取った回数をカウントしており，1 から 4 回目のメッセージ通知ではリソースプロパティの値を False に更新する．この更新により，アプリケーション B の情報を保持する AI サービスのリソースおよび IF サービスのリソース B（IF^+ 対応する EPR）にメッセージ False が通知される．AI サービスは，メッセージが False のためにアプリケーション B を起動しないが，IF サービスはリソースの条件値 False が通知されたメッセージと一致しているために，メッセージを True に変換する．変換されたメッセージは，OR サービスのリソースを経由してアプリケーション A の情報を保持する AI サービスのリソースに届けられ，アプリケーション A が繰り返して実行される．一方で，5 回目のメッセージ通知では LOOP サービスはリソースプロパティの値を True に更新する．これにより，アプリケーション B の情報を保持する AI サービスのリソースにメッセージ True が通知され，アプリケーション B の実行が開始される．反対に，リソース B を保持する IF サービス（IF^+ 対応する EPR）はメッセージを False に変換するために，OR サービスにおけるリソースプロパティの更新が行われず，アプリケーション A は実行されない．

3.7.4　アプリケーション間の入出力ファイル交換

エンドユーザおよびアプリケーション間の入出力ファイル交換は，AI サービスのポートタイプを利用して行われる．エンドユーザがアプリケーションの入力ファイルを用意した場合，ワークフローエンジンは，該当するアプリケーション情報を保持する AI サービスのリソースに対し，createTransferResource ポートタイプを利用して入力ファイルを送信するための RFT サービスのリソースを生成してもらう．その後，RFT サービスのリソース情報（EPR）を用いてファイル転送を実現する．エンドユーザがアプリケーションの出力ファイルを受信する場合も同様である．一方で，あるアプリケーションの出力ファイルを別のアプリケーションの入力ファイルとする場合，後者のアプリケーション情報を保持する AI サービスのリソースに対し，addTransfer ポートタイプを利用して入出力ファイルの組合せを指定する．AI サービスはリソースに対するメッセージ True を受け取ることで，指定された他のアプリケーションの出力ファイルを起動対象となるアプリケーションの入力ファイルとして取得した後に，アプリケーション実行を開始する．

3.7.5　交換される入出力ファイルの変換

アプリケーション間で交換される入出力ファイルの書式が異なる場合や出力ファイルの一部を抜き出して別のアプリケーションの入力ファイルとしたい場合，交換される入出力ファイルの変換機能が必要不可欠である．このようなファイルの変換をサポートするために，Editor サービスが提供される．Editor サービスのポートタイプや機能，リソースやトピックは，アプリケーションの取扱い方法を除いて AI サービスとほぼ同等である．AI サービスは，アプリケーション所有者が配置したアプリケーションを AI ファクトリサービスに登録された RSL ファイルを用いて起動する．一方で，Editor サービスはエンドユーザが用意したファイル変換用のアプリケーションを，同じくエンドユーザが用意した RSL ファイルを用いて起動する．エンドユーザが用意したアプリケーションは，入力ファイルと同様に Editor サービスが提供する createTransferResource ポートタイプを用いて転送する．また RSL ファイルを登録するために，新たに setEditor ポートタイプを用意しており，RSL

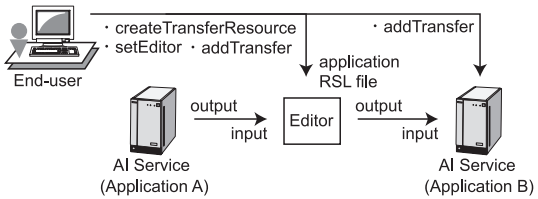


図 10　交換される入出力ファイルの変換
Fig. 10　File translation.

ファイルを読み込んだデータを引数にして利用する．

図 10 に，アプリケーション A の出力ファイルを変換してアプリケーション B の入力ファイルとする手順について示す．図 10 において，エンドユーザはあらかじめ Index サービスに収集されたアプリケーション情報からファイル変換用のアプリケーションを作成する．また，作成したアプリケーションを起動するための RSL ファイルを用意する．その後，アプリケーション A，B の情報を保持する AI サービスのリソースおよび Editor サービスのリソースを生成する．そして，Editor サービスの createTransferResource ポートタイプを利用し，Editor サービスのリソースに対してファイル変換用のアプリケーションを送信する．また，setEditor ポートタイプを利用して，用意した RSL ファイルを登録する．さらに，addTransfer ポートタイプを利用して，アプリケーション A の出力ファイルをファイル変換用のアプリケーションの入力ファイルとするよう指示する．一方で，アプリケーション B の情報を保持する AI サービスのリソースに対して，ファイル変換用のアプリケーションの出力ファイルをアプリケーション B の入力ファイルとするよう addTransfer ポートタイプを利用して指示する．最後に，アプリケーション A，ファイル変換用アプリケーション，アプリケーション B の順にアプリケーションが起動するようリソース間のメッセージ通知を設計する．これにより，アプリケーション B は変換されたアプリケーション A の出力ファイルを自身の入力ファイルとして利用することができる．

3.8　モニタリングとハートビート

AI サービスやユーティリティサービスのすべてのリソースは，アプリケーションの実行状態を取り扱うリソースプロパティ，もしくは条件の真偽値を保持するリソースプロパティのいずれかを含んでいる．また，アプリケーションの実行状態を取り扱うリソースプロパティはトピック AIT として，条件の真偽値を保持するリソースプロパティはトピック CBT として定義されている．そのため，ワークフローエンジンは生成したこれらすべてのリソースに対して通知予約を行うことで，リソース間を流れるメッセージ通知を自身にも派生させることができる．ワークフローエンジンに派生して通知されたメッセージは，図 5 に示したクライアントツール上に表示される．これにより，エンドユーザは現在どのリソース間でメッセージのやりとりが行われているのかを把握することができる．また，リソース間のメッセージ通知の状況を追うことでワークフローの実行状況のモニタリングが可能となる．

さらに，AI サービスやユーティリティサービスのすべてのリソースは，ハートビート用のリソースプロパティを保持しており，APPLICATION HEARTBEAT TOPIC（トピック AHT）として公開している．このリソースプロパティは一定時間ごとに更新される．そのため，設計したワークフロー内に実行時間の長いアプリケーションが含まれる場合には，エンドユーザはワークフローエンジンを通じてこのトピック AHT にも通知予約を行うことで，Web サービスおよびリソースの生存を確認することができる．

これらの機能を応用することで，エンドユーザは簡易的にではあるが設計したワークフローのデバッグやプロファイリングを行うことができる．つまり，設計したワークフローにおいてどのアプリケーションの実行に失敗したのか，また並列に実行されるワークフローにおいてどのパスの実行時間が最も長いのかなどを知ることができる．デバッグやプロファイリングの機能の充実はシステムの実用的な利用に不可欠であるため，今後のさらなる改善を検討している．

4. 性 能 評 価

本章では，BLAST [6] および ClustalW [7] を用いてバイオインフォマティクス分野の代表的なワークフローを設計し，Application Igniting System の性能評価を行う．本性能評価は，ワークフローの実行においてアプリケーション実行に要した時間を除く，システムのオーバヘッド時間を測定することで行う．また，3.7.2 項で述べたように，Application Igniting System はワークフローの並列実行をサポートしている．科学技術計算では，異なる入力で同じワークフローを複数回実行したいという要望が多々ある．そのため，同じワークフローを複数用意し，これらを並列に（並行して）実行した際のオーバヘッド時間も測定する．

4.1　BLAST と ClustalW の連携

BLAST はホモロジ検索のためのコマンド群を含むパッケージである．BLAST に含まれる blastall コマンドは，未知の DNA 配列を入力とし，配列データベース中のすべての配列とのペアワイズアライメントにより類似度の高い配列 ID のリストを出力する．fastacmd

コマンドは，配列データベースを検索するコマンドであり，配列 ID のリストを入力として配列の実データのリストを出力する．一方で，ClustalW はマルチプルアライメントのためのコマンド群を含むパッケージである．ClustalW に含まれる clustalw コマンドは，配列の実データのリストを入力としてマルチプルアライメントを実行し，入力配列の共通の機能や特徴を出力する．これらのコマンドを用いたアプリケーション連携として次のようなシナリオが考えられる．まず，未知の DNA 配列 1 つをエンドユーザが用意し，blastall コマンドを用いてホモロジ検索を行う．その後，ホモロジ検索で得られた配列 ID のリストから興味のある組合せを複数抜き出し，それぞれを入力として fastacmd および clustalw コマンドのペアを連続して実行する．連続して実行する両コマンドのペアは，抜き出した組合せごとに用意し，それぞれを並列に実行する．

本性能評価では，上記のアプリケーション連携を実現するワークフローを設計して Application Igniting System の基本性能を評価する．基本性能の評価という観点から，上記のアプリケーション連携を 5 回繰り返して実行するが，それぞれのホモロジ検索の入力ファイルには同じ DNA 配列を記述する．また，連続して実行する fastacmd および clustalw コマンドのペアを複数用意し，それぞれを並列に実行する．これにより，ワークフローを並列に実行した際の基本性能も評価する．並列実行数を 1，2，4，8 とした 4 つのパターンの基本性能を測定するが，基本性能の評価という観点から各 fastacmd コマンドの入力ファイルは同一であり，類似度の高い順に 250 本の配列 ID を抽出したものを用いる．一方で，異なる入力ファイルで同じワークフローで実行することも可能である．

4.2　アプリケーションの配置と実験環境

本性能評価では，**図 11** に示すグリッド環境を用いる．各アプリケーションは，アプリケーション所有者により各クラスタ上に配置され，それぞれの AI ファクトリサービスにアプリケーション情報が登録されていることを想定している．また，各クラスタ上の WS-GRAM サービスを用いてアプリケーションは起動され，ジョブスケジューラによりクラスタ内部の計算機で実行される．さらに，グローバル IP アドレスを割り当てた 3 つの計算機に，エンドユーザ，Editor サービス，4 つのユーティリティサービス（IF，AND，OR，LOOP サービス）をそれぞれ割り当てた．同一ネットワーク上に 3 つの計算機は接続されている．Netperf ベンチマーク[8] を用い，1 KB のメッセージを送受信しあう TCP Stream 性能を計測したところ，Galley クラスタと Xenia クラスタ間で 79.9 Mbps，Galley クラスタとエンドユーザ間で 78.6 Mbps，Xenia クラスタとエンドユーザ間で 93.6 Mbps の通信性能が得られた．

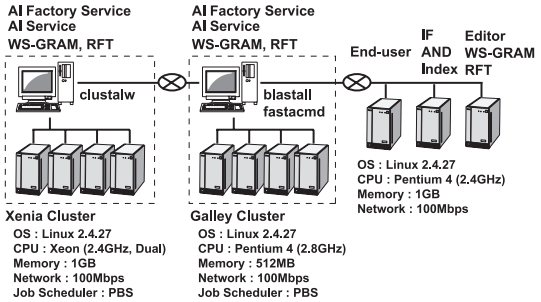


図 11　性能評価に用いたグリッド環境
Fig. 11　Specification of the Grid environment.

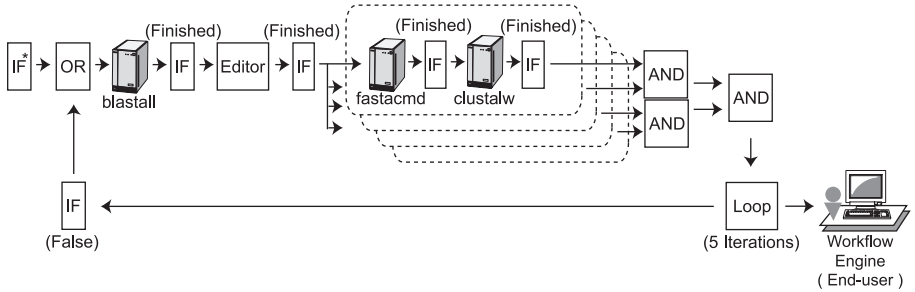


図 12　メッセージ通知の流れ
Fig. 12　Notification message flow.

4.3　ワークフローの設計

エンドユーザは，Index サービスに収集されたアプリケーション情報を利用してワークフローを設計する．

エンドユーザにより設計されたリソース間のメッセージ通知の流れを**図 12** に示す．図 12 は，連続して実行する fastacmd および clustalw コマンドのペアが 4（並列実行数 4）の場合のメッセージ通知の流れである．並列実行数 1，2，8 の場合は，点線で囲まれた部分に相当する 4 つのリソースからなるメッセージ通知を並列実行数分用意する．そして，並列数 2 の場合は 1 つの AND サービスのリソース，並列実行数 8 の場合は 7 つの AND サービスのリソースを用いて同期処理を実現する．並列実行数 1 の場合は同期処理を必要としないため，AND サービスを用いず，LOOP サービスのリソースに直接メッセージを通知する．

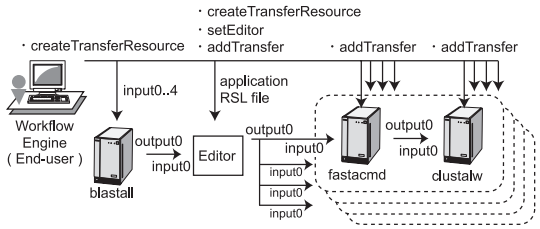


図 13　アプリケーション間のファイル交換
Fig. 13　File transfers and file format translation.

そのため，生成されるリソース数は並列実行数 1 の場合で 12，2 の場合で 17，4 の場合で 27，8 の場合で 47 となる．ワークフローエンジンは各リソースを生成後に，図 12 の矢印の逆向きに通知予約を指示する．指示する通知予約数（矢印の数と同数．ただし，ワークフローエンジンからの通知予約は除く）は，並列実行数 1 の場合で 12，2 の場合で 18，4 の場合で 30，8 の場合で 54 である．また，IF サービスのリソースに対しては条件値として Finished または False を，LOOP サービスに対しては繰返し回数を 5 と指示しておく．ワークフローエンジンは，IF* に該当するリソースに対し，IF サービスのポートタイプを通じて最初のメッセージ True の通知を指示することでワークフローの実行を開始する．また LOOP サービスのリソースからメッセージ True を受け取ることで，ワークフローの実行終了を判断する．

次に，エンドユーザにより定義されたアプリケーション間のファイル交換を**図 13** に示す．図 13 は，連続して実行する fastacmd および clustalw コマンドのペアが 4（並列実行数 4）の場合のファイル交換を定義している．並列実行数 1，2，8 の場合は，点線で囲まれた部分を並列実行数分用意して同様のファイル交換を定義する．図 13 において，ワークフローエンジンはあらかじめ blastall コマンドに対し，AI サービスの createTransferResource ポートタイプを利用してエンドユーザの用意した同じ DNA 配列が記述された入力ファイルをアプリケーション連携の繰返し回数（5 回）分送信する．Editor サービスでは，blastall コマンドのホモロジ検索結果から類似度の高い 250 本の配列 ID を抜き出す処理により出力ファイルの変換を実現する．そのため，エンドユーザはファイル変換用のアプリケーションを作成し，その起動方法を示した RSL ファイルとともにワークフローエンジンを通じて Editor サービスのリソースに提供する．本性能評価では，Ruby 言語を用いた 7 行のスクリプトと 9 行の RSL ファイルを作成した．さらに，ワークフローエンジンは AI サービス

表 2　実行時間の内訳
Table 2　Breakdown of the elapsed time.

項目	項目詳細	時間 (s)
前処理	12 のリソース生成	28.5
	12 回の通知予約の指示	4.9
	5 つの入力ファイルの送信（各 0.3 KB）	25.7
	1 つのアプリケーションの送信（0.1 KB）	6.0
	1 つの RSL ファイルの登録	0.1
	3 回の入出力ファイル交換の指示	0.4
ワークフロー実行	アプリケーション実行（blastall）	1,047.7
	入出力ファイル交換（Editor，341 KB）	37.5
	Editor サービスの実行（ファイル変換）	19.1
	入出力ファイル交換（fastacmd，3 KB）	20.9
	アプリケーション実行（fastacmd）	63.2
	入出力ファイル交換（clustalw，107 KB）	22.4
	アプリケーション実行（clustalw）	678.4
	その他メッセージ通知など	12.8
後処理	5 つの出力ファイルの受信（各 146 KB）	17.2
	12 のリソース破棄	4.9

および Editor サービスの addTransfer ポートタイプを利用して，アプリケーション間の入出力ファイル交換を指示する．指示する入出力ファイル交換数は，並列実行数 1 の場合で 3，2 の場合で 5，4 の場合で 9，8 の場合で 17 である．ワークフロー実行の終了後，ワークフローエンジンは clustalw コマンドの出力ファイルを受信する．出力ファイルはすべて同一なため，いずれの並列実行数においても受信するファイル数を 5 に統一した．

4.4　ワークフローの実行結果

並列実行数を 1 とした際の，前処理および後処理を含むワークフローの総実行時間は 1,989.7 秒であった．実行時間の内訳を**表 2** に示す．表 2 より，前処理および後処理ではリソースの生成に要する時間が最も長いことが分かる．リソースの生成時間に関して詳細を調査したところ，最初のリソース生成に 12.7 秒必要なことが分かり，Globus のセキュリティ関連の処理に原因があると考えられる．その後のリソースは最大でも 2.6 秒，最小の場合は 0.5 秒で生成できており，リソース生成に関するオーバヘッド時間は許容範囲内であるといえる．次に，入出力ファイルおよびアプリケーションの送受信に多くの時間を要していることが分かる．ファイルの送受信では，1 つのファイル転送に平均 4.4 秒を要していることからオーバヘッド時間は短くない．

前処理および後処理を除くワークフローの実行時間の内訳に着目すると，WS-GRAM サー

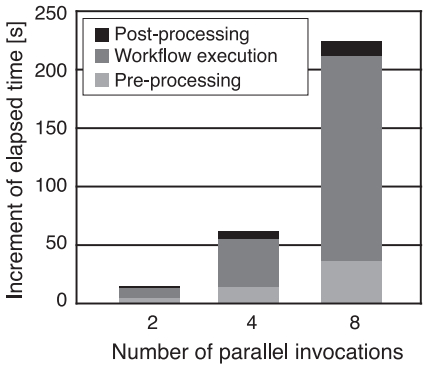


図 14　アプリケーション連携実行時間の増加量
Fig. 14　Increment of elapsed time.

ビスを通じた 5 回のアプリケーション実行に 3 つのアプリケーションで 1,789.3 秒を要しており，残りの 112.6 秒が実質的なオーバヘッド時間といえる．このうち，Editor サービスを含むアプリケーション間の 15 回の入出力ファイル交換に合計で 80.8 秒要しており，平均 5.4 秒必要なことが分かる．前処理や後処理と同様に，ファイル転送にかかるオーバヘッド時間は短くないが，RFT サービスを用いることで障害に対する信頼性や高いセキュリティを確保できるという利点がある．1 回のファイル変換を含む 3 つのアプリケーションの連携において，その他のオーバヘッド時間は平均 6.4 秒である．1 回のファイル変換に平均 3.8 秒を要することを考慮すると，メッセージ通知に関連するオーバヘッド時間は平均 2.6 秒であり，全体に占める割合は非常に小さいといえる．

並列実行数を 2，4，8 とした際の，ワークフローの総実行時間の増加量およびその内訳を**図 14** に示す．生成するリソース数はそれぞれ 17，27，47 である．また通知予約をそれぞれ 18 回，30 回，54 回指示し，入出力ファイル交換をそれぞれ 5 回，9 回，17 回指示する．図 14 より，生成・破棄するリソース数の増加などから前処理および後処理に要する時間は単調に増加していることが分かる．また，前処理および後処理を除くワークフローの実行時間の増加量が，全体の増加量の大部分を占めていることが分かる．

ここで並列実行数が 2 の場合，fastacmd および clustalw コマンドのペアが 2 つ並列に（並行して）実行される．表 2 より，両コマンドの実行および関連する入出力ファイル交換には 784.9 秒を要していることから，並列実行数が 1 のときと比べ，最大で 784.9 秒程度ワークフローの実行時間が増加する可能性があった．同様に，並列実行数 4，8 においても，並列実行数が 1 増加するごとにワークフローの実行時間が 784.9 秒ずつ増加する可能性があったことを考慮すると，図 14 の増加量は非常に小さいことが分かる．

ワークフローを並列に実行する際，実行されるワークフローによっては同一の計算機上で同じアプリケーションが複数同時に実行されることが多々ある．そのような状況では，計算機上のアプリケーション実行がボトルネックとなり，ワークフローを並列に実行する利点が生かされず，並行して実行されるワークフロー（アプリケーション）をそれぞれ順番に，逐次で実行した場合のワークフローとほぼ同等の実行時間となる．本実験では，2 つの PC クラスタ上で fastacmd および clustalw コマンドそれぞれが実行されるため，並列実行数が増えた場合，各 PC クラスタ上で両コマンドを複数同時に実行させる必要があった．一方で，Application Igniting System では PC クラスタ上の WS-GRAM サービスを通じたアプリケーション起動をサポートしている．WS-GRAM サービスを通じてアプリケーションを起動させることにより，PC クラスタ内の各計算機で独立して複数のアプリケーションを並列に実行させることができる．そのため，各コマンドを複数同時に起動させることによる多少のオーバヘッド時間は必要であるものの，コマンドの実行は PC クラスタ内の各計算機で独立して実行できたことにより，並列に実行されるワークフローの実行時間を大幅に短縮できたものと考えられる．これらから，並列に実行可能な部分を多く含むワークフローにおいては，Application Igniting System はワークフローの実行時間を大幅に短縮可能であるといえる．

5. 関 連 研 究

グリッド上のアプリケーション連携に関する研究の多くは，ワークフローに関連する取り組みとして多くなされている．科学技術計算分野では，取り扱うジョブやデータセットの多様性から標準的な枠組みはなく，様々なプロジェクトで多様なワークフロー管理システムが開発されている[9]．そのなかでも文献 9) では，代表的な 12 のワークフロー管理システムをスケジューラとの連携方法やデータセットの取扱いにより分類している．本研究で構築した Application Igniting System は，アプリケーションの起動制御およびアプリケーション間の入出力ファイル交換の両方で集中管理機構を有さず，Triana[10] と類似した分類に属する．Triana は P2P をベースとしたワークフロー管理システムである．一方で，Web サービスの連携を基盤とし，リソース間のメッセージ通知によるワークフローの実行制御は，他のワークフロー管理システムにはない非常に特徴的な機能である．OGSA の標準化が進められるにつれ，今後はリソースを有しメッセージ通知を扱える Web サービスが数多く利用

可能になることが予想される．Application Igniting System は，そのような Web サービスを取り込んで容易に機能拡張を行える可能性があり，今後の発展が期待できる．たとえば，Globus の提供する Trigger サービスは，リソース間を流れるメッセージ通知を収集し，メッセージに特定の内容が含まれている場合にあらかじめ設定したアクションを実行させることができる．Trigger サービスを利用することで，ワークフロー実行中になんらかの異常を検知した場合に，エンドユーザにその旨を通知したり，異常を取り除く処理を実行したりするといったことを Application Igniting System において容易に実現できると考えられる．ワークフロー設計では，ワークフローの逐次実行，並列実行，繰返し実行といった non-DAG をサポートし，アプリケーション間で交換される入出力ファイルの変換機能など，システムが備えるべき必要最小限の機能も満たしていると考えられる．

6. まとめと今後の課題

本研究では，グリッド上で科学技術計算用アプリケーションを複数つなぎ合わせることのできる基盤システムの開発を目標として，WS-Notification を基盤とするグリッド上のアプリケーション連携システム（ワークフロー管理システム）を提案した．また提案するシステムを Globus を用いて実装し，Application Igniting System を構築した．構築したシステムは，Web サービスのリソース間を流れるメッセージ通知を利用して，集中管理機構を有さないシステム構成を採用する．これにより，メッセージ通知を取り扱うユーティリティサービスを追加することで，容易に機能拡張を行うことを可能にした．アプリケーション所有者に対しては，RSL を用いたアプリケーション登録方法を提供する．登録されたアプリケーションは，AI サービスがメッセージ通知を受け取ることによって起動される．また，エンドユーザに対しては，多様なユーティリティサービスを提供することで，柔軟なワークフロー設計を可能にした．設計されたワークフローは，AI サービスおよびユーティリティサービスのリソース間を流れるメッセージ通知により実現される．構築したシステムをバイオインフォマティックス分野における代表的なワークフローに適用し，基本性能を評価した結果，メッセージ通知に関連するオーバヘッドは比較的小さいことが分かった．

今後の課題として，ワークフロー実行における障害の検知およびその対処方法の検討があげられる．また，実用的な利用においては，課金の仕組みや利用可能なアプリケーションの検索機能などが必要不可欠であると考えられる．現在，課金などに代表される高度な Web サービス仕様の標準化が活発に議論されており，今後は洗練された機能を持つ Web サービスが多数利用可能になることが期待される．そのため，それら高度な Web サービスを多数取り入れることによって，構築したシステムの実用性を大きく向上させることができると考えられる．

参 考 文 献

1) Foster, I., et al.: The Open Grid Services Architecture, Version 1.0, Global Grid Forum OGSA-WG, GFD-I.030 (2005).
2) Business Process Execution Language for Web Services version 1.1 (2003). http://www-128.ibm.com/developerworks/library/specification/ws-bpel/
3) Foster, I. and Kesselman, C.: Globus: A Metacomputing Infrastructure Toolkit, *International Journal of Supercomputer Applications*, Vol.11, No.2, pp.115–128 (1997).
4) Czajkowski, K., et al.: The WS-Resource Framework, Version 1.0 (2004). http://www.oasis-open.org/committees/download.php/6796/ws-wsrf.pdf
5) Graham, S., et al.: Publish-Subscribe Notification for Web services, Version 1.0 (2004). http://www.oasis-open.org/committees/download.php/6661/WSNpubsub-1-0.pdf
6) Altschul, S.F., Gish, W., Miller, W., Myers, E.W. and Lipman, D.J.: Basic local alignment search tool, *Journal of Molecular Biology*, Vol.215, No.3, pp.403–410 (1990).
7) Thompson, J.D., Higgins, D.G. and Gibson, T.J.: CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice, *Nucleic Acids Reseach*, Vol.22, No.22, pp.4673–4680 (1994).
8) Netperf benchmark. http://www.netperf.org/
9) Yu, J. and Buyya, R.: A Taxonomy of Scientific Workflow Systems for Grid computing, *Special Issue on Scientific Workflows*, *SIGMOD Record*, Vol.34, No.3, pp.44–49, ACM Press (2005).
10) Taylor, I., Shields, M. and Wang, I.: *Resource Management of Triana P2P Services*, *Grid Resource Management*, pp.451–462, Kluwer Academic Press (2004).

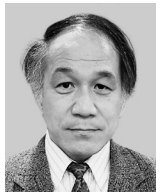
(平成 18 年 4 月 28 日受付)
(平成 19 年 12 月 17 日再受付)
(平成 20 年 1 月 30 日再受付 (2))
(平成 20 年 1 月 30 日採録)

下坂　久司

1979 年生．2007 年同志社大学大学院工学研究科博士後期課程修了．博士（工学）．同年ピーシーアシスト株式会社入社．グリッド技術を応用したアプリケーション開発，ヒューリスティック最適化手法の 1 つである遺伝的アルゴリズムに興味を持つ．人工知能学会，超並列計算研究会各会員．

廣安　知之（正会員）

1966 年生．1997 年早稲田大学大学院理工学研究科後期博士課程修了．早稲田大学理工学部助手，同志社大学工学部助手．知識工学科専任講師，インテリジェント情報工学科准教授を経て 2008 年から生命医科学部教授．進化的計算，最適設計，並列処理，設計工学，医療画像工学等の研究に従事．IEEE，電子情報通信学会，計測自動制御学会，日本機械学会，超並列計算研究会，日本計算工学会各会員．



三木　光範（正会員）

1950 年生．1978 年大阪市立大学大学院工学研究科博士課程修了，工学博士．大阪市立工業研究所研究員，金沢工業大学助教授を経て，1987 年大阪府立大学工学部航空宇宙工学科助教授，1994 年同志社大学工学部教授．進化的計算手法とその並列化，および知的なシステムの設計に関する研究に従事．著書は『工学問題を解決する適応化・知能化・最適化法』（技法堂出版）等多数．IEEE，米国航空宇宙学会，人工知能学会，システム制御情報学会，日本機械学会，計算工学会，日本航空宇宙学会各会員．超並列計算研究会代表．

中尾　昌広（学生会員）

1980 年生．2005 年同志社大学大学院工学研究科博士前期課程修了．同年 NTT アドバンステクノロジ株式会社入社．2007 年同志社大学大学院工学研究科博士後期課程入学．ハイパフォーマンスコンピューティング，進化的計算に興味を持つ．超並列計算研究会会員．