4P-5

AWSミドルウェアにおけるフレームワーク層の開発について

二宮良太† 安齋太一朗† 大谷真†
湘南工科大学†

1. はじめに

自律型 Web サービス（AWS；Autonomous Web Services）[1]とは，異なるビジネスプロセスモデルを持つシステムがインターネット上で遭遇した時に，互いのモデルを協調変形することで商取引を実行することを目的としている．AWS ミドルウェアは AWS を利用したシステム開発の基盤となるソフトウェア群のことであり，本論ではその中核となるフレームワークについて述べる．フレームワークはすでに先行研究でプロトタイプ実装され，スレッドプール制御[2]，デリゲーションパターンの適用[3]が検討されてきた．本論では今日行ったフレームワークの本格的な開発について述べる．また，スレッドプール制御を待ち行列ネットワークとしてモデル化し，その解析結果をシミュレーションとの比較を示し，評価結果を述べる．

2. フレームワークの開発

フレームワークの目的はビジネスプロセスの流れに従ってアプリケーションプログラム（AP）を実行することである．ビジネスプロセスの流れとは例えば，「見積もり依頼を出し，見積依頼を受信して，その結果を元に発注を行うか，取引を中断するかを指示する」といった業務処理を記述したものであり，これをビジネスプロセスモデル（BPM）と呼ぶ．この BPM は業務処理の流れを表すのに対して，AP は業務処理を構成する個々のオペレーションに対応する実際の処理を記述する．

2.1 仕様

図 1 にフレームワークの構成を示す．実線はオブジェクト間の関係を示す．これらのうち AP の開発者が作成するのは Main と AP の実装である App である．他のクラスは AWS ミドルウェアの一部である．フレームワークの処理の中心は APSController である．APSController は App オブジェクトの実行を制御する．具体的には BPM オブジェクトの状態変化が起こると，次に実行するオペレーションを決定し，そのオペレーションに対応する App 内のメソッド（APS メソッドという）を実行することでビジネスメッセージの交換を行う．なお，相手システムへビジネスメッセージを送信する場合はメッセージング機能の send()を実行し，受信する場合は receive()を実行する．AWS ミドルウェアを用いたアプリケーションが相手システムとの BPM に従って一連のビジネスメッセージ交換処理することをビジネスプロセスインスタンス（BP インスタンス）と呼ぶ．BPM と BP インスタンスの主な違いは，BPM は取引の手順そのものを定義しており，BP インスタンスは毎回発生する 1 つ 1 つの取引のことを示す．BP インスタンスが発生すると，それにしたがって図 1 の点線で囲われた部分のオブジェクトがインスタンス化され，BP インスタンスが終了するとオブジェクトも破棄される．また，BP インスタンスは複数同時に発生することが可能であり，その際には各 BP インスタンスに対して Manager が実行スレッドを割り当てることで並行実行される．

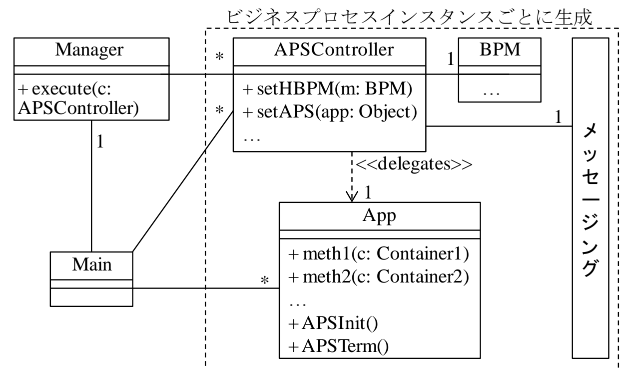

図 1 フレームワークの構成

2.2 BP インスタンスの状態管理

BP インスタンス（BPI）の状態管理を図 2 に示す．まず，BPI が生成されると，BPI の状態は Ready となり，このとき図 1 の点線で囲われたオブジェクトも生成されメモリに格納される．APSController オブジェクトに対してスレッドが割当てられると，BPI の状態が Ready から Active となり，AP が実行される．receive()に失敗すると状態が Active から Wait に遷移する．Wait 状態になると APSController オブジェクトに割当てられたスレッドが取り外される．APSController オブジェクトは Wait 状態となり一定時間経過したのちに，再び Ready 状態となり，スレッドが割当てられた後，再度 receive()を実行する．デフォルトでは receive()が失敗するごとに Wait 状態となる時間も増加される．但し，receive()が指定の回数以上失敗した場合には，オブジェクトのシリアライズを行なった後メモリから取り去り，Archive 状態すなわち長期の受信待ちとなる．

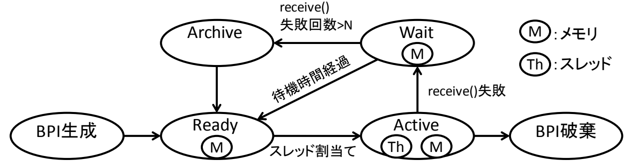

図 2BPI の状態管理

2.3 スレッドプール制御

取引は同時に並行して発生することも珍しくない．したがって，BPI は同時並行に発生し実行される．BPI を並行実行するには各 BPI に対して異なるスレッドを割当てる必要がある．しかし，同時に多数のスレッドを生成すると，スレッドが過多となってしまい，システムにオーバーヘッドが発生してしまう．そこで，BPI を実行するにはスレッドプールを使って APSController オブジェクトのスレッドの割当てを行うようにした．スレッドプール制御については[2]に述べられているため，処理の概略を次の(1)～(3)に示す．(1)Manager の execute()を実行すると，保持しているスレッドプールのスレッドに空き

Development of the Framework Layer for AWS Middleware
†Ryota Ninomiya, Taichiro Anzai, Makoto Oya,
Shonan Institute of Technology

Copyright ©2013 Information Processing Society of Japan.
All Rights Reserved.

があれば直ちに実行を行う．(2)スレッドに空きがない場合，実行待ちのキューに退避され，スレッドに空きができるまで実行待ちとなる．(3)receive()で長時間受信待ちとなった場合には APSController オブジェクトに割り当てられたスレッドが取り外されるので，そのスレッドを回収した後に，再度実行待ちのキューに APSController オブジェクトを退避する．

3. 性能評価

3.1 モデル化

AWS アプリケーションには様々な形態が考えられ，一般的な AWS アプリケーションを評価することは難しい．そこでスレッドプール制御を図 3 に示すようなモデル化を行う．実行ノードはスレッドと実行待ちのキューから構成されており，待機ノードもまた同様である．execute()を実行すると APSController オブジェクトが実行ノードに追加される． APSController オブジェクトが実行ノードに入ると AP が実行を終了した場合には AP が実行ノードから Node0 に遷移し，関連するオブジェクトも破棄される．便宜上スレッドプール制御の外側のことを Node0 としておく．APSController オブジェクトが receive()に失敗すると実行ノードから待機ノードに移る．待機ノードに遷移した後，待機時間を経過したのちに再度実行ノードに移る．モデルを単純化するため，λ時間ごとに BPI を生成し，また，AP の実行時間及び待機時間は分布関数に従うものとしサービスクラスの集合 $c=(c_1,c_2,c_3,\ldots)$のいずれかに従って決定される．また，Node1 から Node0 へ遷移確率を p とし，Node1 から Node2 に遷移確率を 1-p とした．さらに，Node1 は S 個のスレッドを保持する複数サーバ型とし，Node2 は待機処理であるため（すなわち CPU の処理を必要としないため）無限サーバ型とする．以上のようにスレッドプール制御は Node0～Node2 までの待ち行列ネットワーク[4]として構成される．システムの定常状態を求めるための積形式解は以下の通りとなる．

$$P(n_j) = C_j \Phi_j(n_j) \lambda^{n_j} \rho_j^{n_j}$$

$$\rho_j = \sum_{k=1}^{|c|} \sigma_j(k)$$

$$\sigma_j(k) = \frac{\theta_{jk}}{\mu_{jk}}$$

ここで，n_jはノード j（図では Nodej）に滞在する AP の数，C_j はノード j の正規化定数，Φ_jはノード j の容量係数，λは到着率，θj は相対訪問回数ベクトル，μ_{jk} はノード j のサービスクラス k のサービス要求時間の逆数である．この $P(n_j)$は n_j に関する周辺分布となる．θj はノード i からノード j への移動を表現する移動行列 R_{ij} からトラヒック方程式$(1,\theta_j)=(1,\theta_j)R_{ij}$ を解くことにより導出される．また，R_{ij} はノード i クラス k からノード j クラス l の推移確率 r((i,k),(j,l))を要素とする．

ノード 1 は複数サーバ型なので，

$$\Phi_1(n_1) = \begin{cases} \{n_1!\,\nu_1(1)^{n_1}\}^{-1}, & n_1 \le S \\ \{S!\,S^{n_1-S}\,\nu_1(1)^{n_1}\}^{-1}, & S < n_1 \end{cases}$$

$$\nu_1(n_1) = \begin{cases} n_1\,\nu_1(1), & n_1 \le S \\ S\,\nu_1(1), & S < n_1 \end{cases}$$

$$C_1 = \sum_{n_1=0}^{S} \frac{\lambda h_1^{n_1}}{n_1!} + \frac{\lambda h_1^{S+1}}{S!\,(S-\lambda h_1)}$$

となり，ノード 2 は無限サーバ型なので，

$$\Phi_2(n_2) = \{n_2!\,\nu_2(1)^{n_2}\}^{-1}, \quad n_2 = 1,2,\ldots$$

$$\nu_2(n_2) = n_2\,\nu_2(1), \quad n_2 = 0,1,2,\ldots$$

$$C_2 = e^{-\lambda h_2}$$

$$h_j = \rho_j \nu_j(1)^{-1}$$

となる．ここで$\Phi_2(0) = 1$とする．

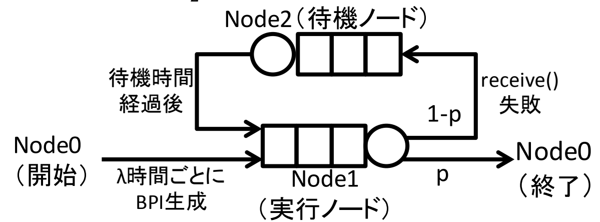

図 3 スレッドプール制御のモデル化

3.2 実験

3.1 に示した積形式解及びシミュレーションの結果を表 1 に示す．パラメータは$\lambda=1/15$,$S=4$,$\nu_1=1$,$\nu_2=1$,
r(0,(1,1))=1,r((1,1),0)=0.9,r((1,2),0)=0.9,r((1,3),0)=0.9,
r((1,1),(2,2))=0.1,r((1,2),(2,3))=0.1,r((1,3),(2,3))=0.1,
r((2,1),(1,1))=1,r((2,2),(1,2))=1,r((2,3),(1,3))=1,
μ=((10,6,4),(10,6,4))とした．シミュレーションと積形式解の値を比較すると近似できていることが分かる．また，図 4 に積形式解による解析結果を示す．解析結果から実行ノード，待機ノードともにキューの長さが定常状態では 4 以下になる確率が高いこと分かった．

表 1 解析結果

	積形式解		シミュレーション解	
n_j	Node1	Node2	Node1	Node2
0	0.163	0.747	0.163	0.759
1	0.292	0.218	0.303	0.209
2	0.262	0.032	0.264	0.029

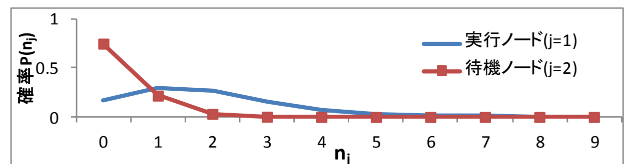

図 4 積形式解による解析結果

4. おわりに

本論では，AWS ミドルウェアのフレームワークの開発について述べた．また，実際のシステムに対して積形式解での解析方法を示した．今回の実験した範囲ではシステムのオーバーヘッドは問題にならないことが分かった．今後は効率的に並行取引を実行するスレッドプール制御の開発が課題として挙げられる．

参考文献

1) 大谷, 自律型 Web サービス：原理と実装, 情報処理学会論文誌, 54 巻 2 号, 2013

2) 二宮, 他, AWS(自律型 WEB サービス)ミドルウェアフレームワーク制御, 情報処理学会第 73 回全国大会, pp.1-707-708, 2011

3) 二宮, 他, 委譲型 AWS フレームワークの実現方法, 情報処理学会第 74 回全国大会, pp.1-83-84, 2012

4) 紀, 待ち行列ネットワーク, 朝倉書店,2002

Copyright ©2013 Information Processing Society of Japan.
All Rights Reserved.