IoT アプリケーションアーキテクチャに基づく アプリケーションフレームワークの設計と試作

M2023SE009 曽我康平
指導教員：野呂昌満

1 はじめに

IoT(Internet of Things) はその要素技術の進化により，大きな変容が起きている領域である．相互運用性の保証，再利用による開発コストの削減，クラウドとの連携などを目的として，SOA(Service Oriented Architecture) に基づいて IoT アプリケーションを開発するための研究が盛んに行われている [1][2].

IoT と SOA の統合について研究がなされている一方，その実用性を考慮した研究例は数少ない．これまでの研究では主に，SOA が持つ相互運用性やセキュリティ等の利点を IoT アプリケーションの領域に応用する方法が議論されている．他方，実用性の観点において，実時間性と耐故障性は IoT アーキテクチャの最重要課題である．

本研究の目的は，以下の要件を満たした，IoT と SOA を統合したソフトウェアアーキテクチャを定義することである．本研究では，リアルタイム IoT[3] を対象とする．

1. SOA におけるアプリケーション開発の簡単さの保持
2. 実時間性と耐故障性を重視した分散並行処理の実現

研究目的を具体化して定義した研究課題を以下に示す．

1. SOA に基づく，実時間性と耐故障性を保証する IoT アーキテクチャの定義
2. 提案したアーキテクチャに基づくプロトタイプの作成
3. プロトタイプを用いた，提案するアーキテクチャの妥当性検証

IoT アプリケーションは，実装前にモデルを実行し，シミュレーションすることで期待通りの動作をするかを検証する必要がある [4]．本研究では，以上の研究課題を解決することで，提案するアーキテクチャに基づくプロトタイプがテストベッドとして使用できることを示す．

2 先行研究

我々は，SOA の参照アーキテクチャと IoT の参照アーキテクチャを統合することで，SOA に基づく，IoT の参照アーキテクチャを提案した．本研究で提案するアーキテクチャは，この参照アーキテクチャに基づいて定義する．

本田ら [5] は，この参照アーキテクチャに基づき，コンテキスト指向を統合することで，IoT アプリケーションのためのアーキテクチャを提案した．マイクロサービスをマッシュアップするための協調論理をコンテキストを用いて記述することで，IoT デバイスの状況に合わせた柔軟な対応を可能にした．

3 課題解決へのアプローチ

IoT アプリケーションは，分散ネットワーク下で複数のコンポーネントが並行に動作するシステムである．IoT では，位置透過性，実時間性，耐故障性などの非機能特性が重視される．非機能特性はアスペクトとして実現するのが理想である．他方，IoT アプリケーションの本質としてコンテキスト指向が必要とされる．本研究では，これらの概念が共存するアーキテクチャを提案する．我々が提案した IoT と SOA を統合した参照アーキテクチャを詳細化することで，アプリケーションの構造，アスペクト間の関係，コンポーネントの配置の 3 つの視点から概念アーキテクチャを定義する．概念アーキテクチャを詳細化し，実時間性と耐故障性を保証するパターンを定義する．これらのパターンをアスペクトとして実現することで，SOA におけるアプリケーション開発の簡単さを損なわず，実時間性と耐故障性を保証できると考える．以上のパターンを用い，概念アーキテクチャを具体化することで，実時間性と耐故障性を保証する具象アーキテクチャを定義する．

提案したアーキテクチャに基づき，プラットフォームのプロトタイプを作成する．このプロトタイプは，提案したアーキテクチャに基づいて，プラットフォームが実現可能かを確かめるために試作する．本研究でプラットフォームに用いる ESB は，通信を実現する基本的な機能を持っていればいいので，シリアライズ，メッセージルーティング，メッセージキューイングを機能として持つように定義する．プロトタイプは使い慣れた言語である Java で実装し，リモート通信は，RMI ライブラリを用いることで簡単に実現する．

提案したアーキテクチャの妥当性は，テストベッドとしての有効性の観点から定量的に検証し，再利用性，変更容易性の観点から定性的に検証する．定量的な検証では，本研究で提案するアーキテクチャに基づくプロトタイプが定量的に評価するためのテストベッドとして使用できるかを考察する．本研究では，プロトタイプを実行してシミュレーションを行い，処理速度，実時間性，耐故障性の観点からプロトタイプの実用性を定量的に評価する．再利用性の観点からは，本研究で定義した実時間性を保証するパターンと耐故障性を保証するパターンをアスペクトとして実装する際，アプリケーションロジックを記述したコードに対して，どれだけの変更を施す必要があるかを議論する．変更容易性の観点からは，ESB を変更した場合に，どのような変更が必要になるかを考察することで，本研究で提案したアーキテクチャが変更容易性を保証できているかを考察する．

4 アーキテクチャ

アスペクト間の関係，アプリケーションの構造，コンポーネントの配置はそれぞれ，Views and Beyond における Module ViewType，C&C(Component and Connector)

ViewType，Allocation ViewType で適切に表現できる．本研究では，この 3 つの ViewType でアーキテクチャを表現する．実時間性と耐故障性を保証するパターンは，概念アーキテクチャの C&C ViewType を詳細化することで定義する．

4.1 概念アーキテクチャ

概念アーキテクチャを図 1，図 2，図 3 に示す．

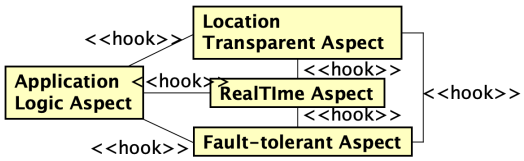

図 1 概念アーキテクチャ（Module Viewtype）

図 1 は，アスペクト間の関係を示すことを目的にして定義した．SOA に基づく IoT アプリケーションでは，アプリケーションロジックアスペクトに加え，位置透過性アスペクト，実時間性アスペクト，耐故障性アスペクトが存在する．これらのアスペクトが連携して動作するので，図 1 に示すような構造となっている．

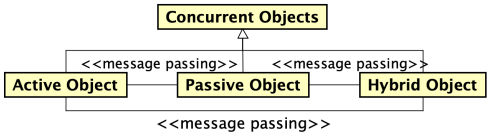

図 2 概念アーキテクチャ（C&C Viewtype）

図 2 は，並行オブジェクト間の関係を示すことを目的にして定義した．並行オブジェクトは，メソッドを起動し続けるアクティブオブジェクト，メソッドが起動されるまで待ち続けるパッシブオブジェクト，そのどちらの特性も持つハイブリッドオブジェクトに分けられる．各並行オブジェクトはメッセージパッシングにより，連携して動作するので，図 2 に示すような構造となっている．

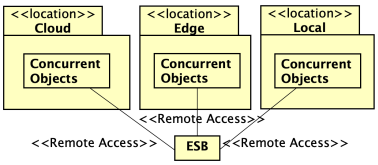

図 3 概念アーキテクチャ（Allocation Viewtype）

図 3 は，各オブジェクトが動作している物理的な配置の関係を示すことを目的にして定義した．IoT では，Cloud，Edge，Local に並行オブジェクトを配置する．各並行オブジェクトのメッセージパッシングは，ESB を介したリモートアクセスにより実現しているので，図 3 に示すような構造となっている．

4.2 実時間性と耐故障性を保証するパターン

実時間性を保証するためのパターンを図 4 に示し，耐故障性を保証するためのパターンを図 5 に示す．

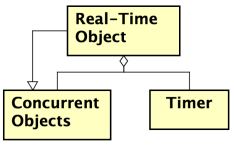

図 4 実時間性を保証するためのパターン

本研究では，タイムアウトにより実時間性を保証する．タイムアウトの高級言語での実現は，Ada における Select 構文の利用が理想である．それをオブジェクト指向でパターンとして実現するには，タイマーが必要になるので，図 4 に示すような構造になる．

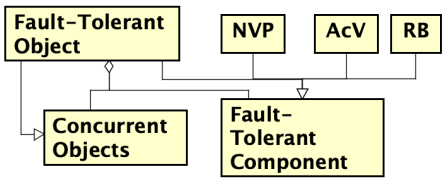

図 5 耐故障性を保証するためのパターン

本研究では，耐障害アーキテクチャである NVP(N-Version Programming)，AcV(Acceptance Voting)，RB(Recovery Block) にパターンを対応させることで耐故障性を保証する．NVP，AcV，RB はそれぞれ異なる強みを持っている．図 5 では，アプリケーションの特性に合わせて適切なパターンを選択して実行することが可能であり，柔軟に耐故障性を保証できる．

4.3 具象アーキテクチャ

具象アーキテクチャを図 6，図 7，図 8 に示す．

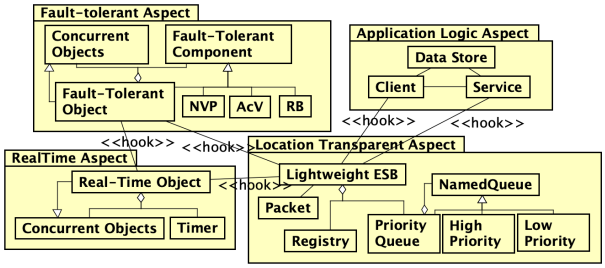

図 6 具象アーキテクチャ（Module Viewtype）

本研究では，通信の処理において実時間性アスペクトと耐故障性アスペクトを適用することを考える．図 6 では，適用するアスペクトによって，Client と Service 間の通信方式が変化することを示している．

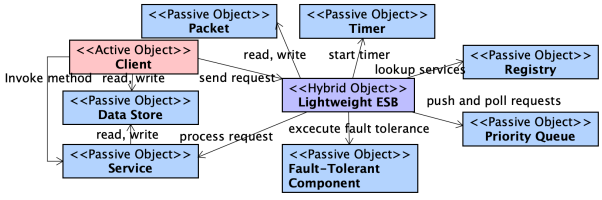

図 7 具象アーキテクチャ（C&C Viewtype）

IoT アプリケーションを構成する各コンポーネントは並行オブジェクトである．図 7 では，Client が起点となり，Service や ESB といった並行オブジェクトとメッセージパッシングにより連携して動作することを示している．

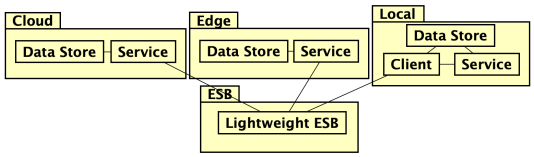

図 8 具象アーキテクチャ（Allocation Viewtype）

IoT アプリケーションでは，Local にある Client が Local，Edge，Cloud にある Service と連携して動作する．Client が Edge や Cloud に配置された Service と連携するときは，ESB を介してリモートアクセスするので，図 8 に示すような構造になる．

5 実装

提案した具象アーキテクチャに基づき，現金処理機システムを例にプロトタイプを試作する．具象アーキテクチャにおける，Client として changer を配置し，Local の Service として io_device, cash_device, cash_box を配置し，Edge の Service として context, behavior_activator を配置する．プロトタイプの動的振舞いを図 9 に示す．

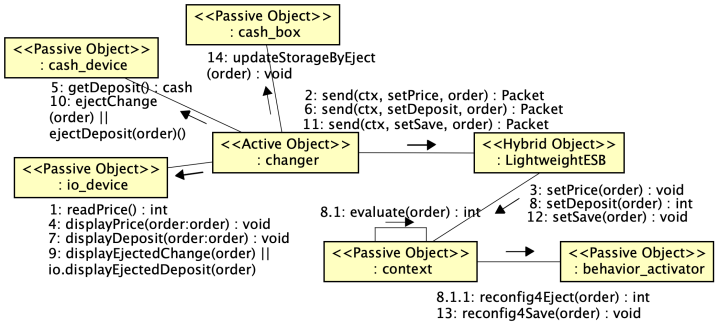

図 9 現金処理機システムの振舞い

図 9 では，決済処理におけるコンポーネント間の動的な関係が示されている．このシステムは，changer がアクティブオブジェクトとして，他のコンポーネントに命令を出すことで稼働している．

6 実験と考察

6.1 処理速度の観点からのアーキテクチャの評価

処理速度の観点からは決済処理を終えるまでにかかった時間を計測し，その処理時間が現金処理機システムとして扱うのに適切な時間であるかを検証する．加えて，決済処理を終えるまでにかかった時間の内訳を計測し，さらに通信速度をあげるためには，どのような改善が必要かを議論する．決済処理は 100 回試行し，実験で得られた処理時間の平均値を用いて議論する．各決済処理では，購入金額と投入金額がランダムに定まるようにして実験を行った．実験結果を表 1 に示す．

表 1 決済処理を終えるまでにかかった時間の内訳

	決済処理の合計時間	シリアライズ（デシリアライズ）	スタブの取得	通信	決済処理	ポーリング
処理時間 (ms)	602.75	106.68	256.31	127.72	21.45	1.65
合計時間に対する割合 (%)	100	18	43	21	3	0

決済処理を終えるまでにかかった時間は 602.75 ミリ秒だった．これはユーザが待ち切れないほどの時間ではないが，現金処理機では，お金の集計やお釣りの排出など，ハードウェアを制御する処理も行う必要があるので，決済処理はより高速化することが求められると考えられる．

表 1 より，シリアライズ，スタブの取得，通信にかかる時間が決済処理の合計時間の多くを占めていることが分かる．中でも，スタブを取得する処理が多くの割合を占めている．これは，RMI の lookup メソッドが，指定されたリモートオブジェクトのレジストリをネットワークを介して参照し，スタブを取得するからだと考えられる．リモートオブジェクトのスタブの取得をローカルで行えるようにするか，スタブのサイズを削減することで，処理速度を上げることができると考察できる．

6.2 実時間性の観点からのアーキテクチャの評価

実時間性の観点からは，タイムアウトが発生してから，サーバ側のキューからリクエストを削除するまでにかかる時間の計測することで，実用性を検証する．タイムアウト処理を適用したトランザクションは 100 回試行し，得られた処理時間の平均値を用いて，タイムアウトパターンの妥当性を議論する．

クライアント側でタイムアウトが発生してから，サーバ側の ESB のキューからタイムアウトが発生した処理のリクエストを削除するのにかかった時間は，平均で 151.99 ミリ秒だった．現金処理機システムの要件から考慮すると，通常の処理に加え，0.15 秒ほどの処理時間がかかるのは実用的ではないと考えられる．キューから指定したリクエストを削除するのに要した時間は 0 ミリ秒だったので，指定した要素をキューから削除する処理による遅延は少ないことが確認できた．すなわち，スタブの探索やシリアライズ等の RMI よる通信や，回線速度を向上させることで，さらなる速度向上が期待できると考えられる．

6.3 耐故障性の観点からのアーキテクチャの評価

耐故障性の観点からは，耐故障性を保証するパターンを適用した際に，4 種類の実現パターンが正常に終了するまでにかかる時間をそれぞれ計測することで，実用性を検証する．耐故障性を保証するための各パターンを適用したトランザクションをそれぞれ 100 回試行し，得られた処理時間の平均値を用いて，耐故障性を保証するためのパターンの妥当性を議論する．実験結果を表 2 に示す．

表 2 各耐故障性を保証するパターンを適用した処理時間

パターン	NVP	AcV	RB	NVP&RB
処理速度の平均 (ms)	124.93	133.78	123.67	147.17

耐故障性を保証するためのパターンを適用しなかった場合は平均 29.02ms で処理が完了した．表 2 より，耐故障性を保証するためのパターンを適用しなかった場合と比較して，NVP を適用すると約 4.3 倍，AcV を適用すると約 4.6 倍，RB を適用すると約 4.3 倍，NVP と RB を統合したパターンを適用すると約 5.1 倍の処理時間がかかることが分かる．現金処理機システムの要件から考慮すると，通常の処理の 4, 5 倍の処理時間がかかるのは実用的ではないと考えられる．

6.4 プロトタイプのテストベッドとしての評価

定量的な実験では，サービスを Cloud に配置したり，実時間性，耐故障性を保証したりすると，どの程度の遅延が発生するかを定量的に示せた．すなわち，我々が試作したプロトタイプは定量的な評価を行うためのテストベッドとして使用できることが確認できた．

我々のプロトタイプはテストベッドとして簡単に実装可能である．本研究における現金処理機システムのプロトタイプは，まず，ローカル環境で動作する現金処理機システムのアプリケーションロジックだけを実装し，そこに通信のためのプラットフォームを結合することでクラウド化した．現金処理機システムをクラウド化する際には，サーバに配置するコンポーネントのメソッドを起動するコードを，LightweightESB の send メソッドに変更するだけで簡単に実装できた．これはアーキテクチャが変更に強いことの証左でもある．

6.5 再利用性の観点からのアーキテクチャの評価

再利用性の観点では，実時間性，耐故障性を保証するパターンをアスペクトとして実装する際，現金処理機システムのアプリケーションロジックを記述したコードにどのような変更を加える必要があるかを示すことで，再利用性を保証できたかを考察する．

実時間性，耐故障性を保証するためのパターンをアスペクトとして実装することを考える．実時間性を保証するパターンを AspectJ で実装することを考えると，ポイントカットの定義，run メソッドの before アドバイスとしての実装，cancel メソッドの after アドバイスとしての実装は，すべてアスペクトコンポーネントである Timer クラス内で完結する．耐故障性を保証するためのパターンを AspectJ で実装することを考えると，ポイントカットの定義，run メソッドの around アドバイスとしての実装は，すべてアスペクトコンポーネントである Delegator クラス内で完結する．

以上より，アスペクトを適用する先のコンポーネントの変更なしに，実時間性と耐故障性を保証することが可能なので，本研究で提案するアーキテクチャでは，再利用性を保証することができたと考えられる．

6.6 ESB の変更容易性の観点からのアーキテクチャの評価

本研究では，ESB の機能は考慮せず，実時間性と耐故障性は ESB の機能と独立して保証できるようにアーキテクチャを定義した．実時間性と耐故障性の保証をより効果的にするために，ESB はより軽量化した機能を有する必要があるので，提案したアーキテクチャでは，ESB を変更することが想定される．図 6 より，ESB の変更によって影響が出るコンポーネントは，クライアント，サーバの役割を担うコンポーネントと実時間性アスペクト，耐故障性アスペクトだけであることが分かる．クライアントの役割を担うコンポーネントにおいて，ESB と関連を持っている箇所は，サーバ側のメソッドを起動するための ESB の通信用メソッドだけである．サーバの役割を担うコンポーネントは，ESB からメソッドが起動されるだけであり，サーバ側で ESB の機能を意識する必要はないので，変更は必要ない．アスペクトを変更後の ESB に適用するには，アスペクトファイルで変更後の ESB の通信用メソッドをポイントカットとして追加するだけでいい．

以上より，ESB の変更に伴って必要になる変更は少なく，本研究で提案したアーキテクチャは変更容易性を保証していると考えられる．本研究の主題はプロトタイプを用いて定量的に性能評価をすることだったので，IoT 用の軽量な ESB の設計は今後の課題とする．

7 おわりに

本研究では，SOA におけるアプリケーション開発の簡単さを損なうことなく，実時間性と耐故障性を保証できる IoT のためのアーキテクチャを定義した．提案したアーキテクチャの妥当性を検証するために，テストベッドとしての有効性の観点からの定量的な議論と，再利用性，変更容易性の観点からの定性的な議論を行った．定量的な議論では，本研究で提案するアーキテクチャが，IoT アプリケーションを定量的に評価するためのプラットフォームとして使用できることを示した．定性的な議論では，本研究で提案するアーキテクチャが，再利用性と変更容易性が保障されていることを示した．

今後の課題は，IoT のための ESB の設計と本アーキテクチャのクリーンアーキテクチャとしての再定義である．

参考文献

[1] R Kohar, “IoT systems based on SOA services: Methodologies, Challenges and Future directions,” *2020 Fourth ICCMC*, 2020.

[2] ZD Patel, “A review on service oriented architectures for internet of things (IoT).” *2018 2nd ICOEI*, 2018.

[3] CY Chen, et al,. “Securing real-time internet-of-things,” *Sensors*, 2018.

[4] B Costa, et al,. “Towards the adoption of OMG standards in the development of SOA-based IoT systems,” *JSS*, 2020.

[5] 本田一輝, 野呂昌満, 他., “IoT アプリケーションのためのコンテキスト指向ソフトウェアアーキテクチャ,” FOSE2022, 2023.

[6] P Clements, F Bachmann, et al,. “Documenting Software Architectures: Views and Beyond,” Second Edition, Addison Wesley, 2010.