JAIST Repository
https://dspace.jaist.ac.jp/

Title	IoTシステムの双方向データフローにおける設計と実装の複雑さを解消する手法の提案
Author(s)	栗林, 健太郎
Citation	
Issue Date	2022-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/17650
Rights	
Description	Supervisor:篠田　陽一, 先端科学技術研究科, 修士(情報科学)

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修士論文

IoT システムの双方向データフローにおける設計と実装の
複雑さを解消する手法の提案

栗林 健太郎

主指導教員 篠田 陽一

北陸先端科学技術大学院大学
先端科学技術研究科
（情報科学）

令和 4 年 3 月

Abstract

In this paper, we propose a method for describing the data flow and processing of bi-directional and diverse data flow patterns in IoT systems using a single language and communication protocol in a comprehensive manner, with the aim of reducing the complexity in the design and implementation of IoT systems.

IoT systems, which bridge devices such as sensors and actuators in physical space with computational processes in cyberspace, have been applied in various domains. Such IoT systems are designed and implemented based on an architectural model consisting of three layers: device layer, edge layer, and cloud layer. The three-layer structure allows data to be given meaning in the middle layer, ensures security by avoiding direct access by the device layer, and can take advanteges of edge computing.

On the other hand, the adoption of the three-layer architectural model poses a challenge in terms of structural complexity in system design and implementation. This is due to three factors: (1) the variety of programming languages and communication protocol options, (2) the variety of data acquisition methods and bidirectional data flow, and (3) the poor visibility of data flow throughout the IoT system. While related research can solve each of the challenges individually, this research aims to solve all the challenges mentioned above.

The proposal that addresses the first issue is a method that can design and implement the three layers in an integrated manner using the same programming language and communication protocol. We propose a method to design and implement the three layers in an integrated manner using the Elixir programming language as the core. In the proposed method, we use Elixir as a programming language and Erlang/OTP distributed network protocol as a communication protocol. In addition, we use a secure communication protocol and an authentication method for security measures among the three layers.

Our second proposal is an infrastructure that supports push, pull, and demand methods and can be used in different ways. We propose a method to solve the second problem by using Pratipad, a library developed by the author, on the distributed Erlang/OTP network infrastructure. In the proposed method, the user can specify which data acquisition method is used to acquire data from a device. Then, by defining the type of message for each mode, the proposed method can handle any data acquisition method while using the same infrastructure.

As the solution for the third problem, we propose a notation that can declaratively describe the data flow of an IoT system consisting of three layers, and then separate the data flow from the processing. The proposed notation can describe and execute the data flow as Elixir code, including the types of modes and bidirectionality described above, and the data processing in the edge layer. By using

these notations, we can grasp the whole data flow under one view.

In order to evaluate the proposed method, for each of the proposals, we evaluate (1) that the proposed method can be used to design and implement a three-layer IoT system in an integrated manner, (2) that the proposed method can easily handle any of the data acquisition methods, and (3) that the proposed notation can sufficiently represent the entire data flow. As a result, we believe that the proposed method, which simultaneously solves all of the problems that related research has attempted to solve individually, has achieved an academic contribution in that it solves the problems of IoT systems and shows a method that can be designed and implemented more effectively with concrete implementation.

目 次

第 1 章　はじめに **1**
1.1 本研究の背景 1
1.1.1 IoT システムのデータフロー 1
1.1.2 IoT システムの 3 層モデル 1
1.1.3 3 層モデルの利点 2
1.2 3 層モデルを採用する IoT システムの課題 3
1.2.1 課題 (1）各層の構成・通信の多様性 3
1.2.2 課題 (2）データ取得方式の多様性 3
1.2.3 課題 (3）データフロー・処理記述の複雑性 3
1.3 本研究の目的 3
1.3.1 統合的な設計・実装手法の提案 4
1.3.2 多様なデータ取得方式への対応 4
1.3.3 一望のもとに把握可能なデータフロー記法の創出 4
1.4 論文の構成 4

第 2 章　関連研究 **5**
2.1 課題（1）にする関連研究 5
2.1.1 単一の言語による統合的な開発環境 5
2.1.2 IoT システムのデータ通信プロトコル 5
2.2 課題（2）にする関連研究 6
2.2.1 IoT デバイスからのデータ取得方式 6
2.2.2 関連するデータ取得方式 6
2.2.3 双方向通信の必要性 6
2.3 課題（3）にする関連研究 7
2.3.1 データフローモデル 7
2.3.2 ビジュアルなデータフロー記述 7
2.4 本研究の位置づけ 7

第 3 章　提案手法 **9**
3.1 提案（1）3 層を同一のプログラミング言語と通信プロトコルを用いて統合的に設計・実装できる手法 9
3.1.1 採用するプログラミング言語 9

3.1.2 採用する通信プロトコル 10
3.1.3 実装 10
3.2 提案（2）push，pull，demand 方式のいずれにも対応し，使い分けられる基盤 11
3.2.1 データ取得方式の設定方法 12
3.2.2 push モード 12
3.2.3 pull モード 13
3.2.4 demand モード 14
3.2.5 実装 16
3.3 提案（3）3 層からなるデータフローを一望のもとに把握できる記法 17
3.3.1 データフロー記法 17
3.3.2 記法の種別 17
3.3.3 データフローの記述例 18
3.3.4 プロセッサーの記述例 18
3.3.5 実装 19

第 4 章 評価 **21**
4.1 提案（1）の評価 21
4.1.1 提案手法を用いた IoT システムの構築例 21
4.1.2 評価 22
4.2 提案（2）の評価 23
4.2.1 提案手法の提供するデータ取得方式と双方向性 23
4.2.2 評価 23
4.3 提案（3）の評価 24
4.3.1 提案する記法の表現能力 24
4.3.2 評価 25

第 5 章 おわりに **26**
5.1 本研究の成果 26
5.2 本研究の社会的な意義 26
5.3 本研究の学術的な意義 26

図 目 次

1.1 IoT システムのアーキテクチャモデル 2

3.1 提案手法を用いる IoT システムのアーキテクチャ 10
3.2 TLS 通信を行うためのデバイス層の設定例 11
3.3 クライアント認証を行うためのデバイス層の設定例 11
3.4 push モードのシーケンス図 12
3.5 pull モードのシーケンス図 14
3.6 demand モードのシーケンス図 15
3.7 双方向データフローの実装 16
3.8 提案手法を用いたデータフローの記述例 18
3.9 提案手法を用いたデータ処理の記述例 19
3.10 データフロー記法のマクロを用いた実装（抜粋） 20
3.11 双方向データフローの実装 20

4.1 提案手法を用いた IoT システムの構築例 21

表 目 次

2.1 課題と関連研究に基づく本研究の位置付け 8

3.1 IoT システムのデータフローを記述するために Pratipad が提供する記法 17

4.1 IoT システムの構築例に用いた構成 22
4.2 提案手法を用いた各データ取得方式に対応するデータフローの記述 24
4.3 提案手法を用いたエッジ層におけるプロセッサーの記述 24

第1章　はじめに

本研究は，IoT システムの設計・実装における複雑さを解消することを目的として，IoT システムにおける双方向かつ多様なデータフローパターンを，単一の言語と通信プロトコルを用いつつ一望できる形でデータフローと処理を記述できる手法を提案する．本章では，本研究の背景と目的について述べる．

1.1　本研究の背景

本節では，本研究の背景である IoT システムを構成するアーキテクチャについて整理する．

1.1.1　IoT システムのデータフロー

物理空間上のセンサーやアクチュエーター等のデバイスとサイバー空間上の計算処理とを架橋する IoT システムは，様々な領域において適用事例をもたらしている [1]．IoT システムでは，デバイスによってセンシングされた物理空間上のデータが，ネットワークを通じてサイバー空間上のシステムへと送信される．また，集められたデータを用いてサイバー空間上で分析した結果に基づき，物理空間上のデバイスへのアクチュエーション指示が行われる．そのため，物理空間とサイバー空間との間における双方向のデータフローの構成が重要な課題となる．

1.1.2　IoT システムの 3 層モデル

物理空間とサイバー空間との間のデータフローを構成するために，IoT システムを階層的に構成する複数のアーキテクチャーモデルが提案されている [2]．そのうちのひとつである 3 層モデルは，IoT システムを（1）パーセプション層,（2）ネットワーク層,（3）アプリケーション層の 3 層によって構成されるものとする．また，5 層モデルは（2）ネットワーク層をさらに細分化し,（2-1）トランスポート層,（2-2）プロセッシング層,（2-3）ミドルウェア層によって構成されるものとする．本研究においては，3 層モデルと 5 層モデルを踏まえた上で，それぞれの層における計算資源の物理的な配置に基づいて（1）デバイス層,（2）エッジ層,（3）ク

ラウド層の3層において構成されるIoTシステムのアーキテクチャーモデルを検討する．各モデル間の対応を図1.1にまとめた．

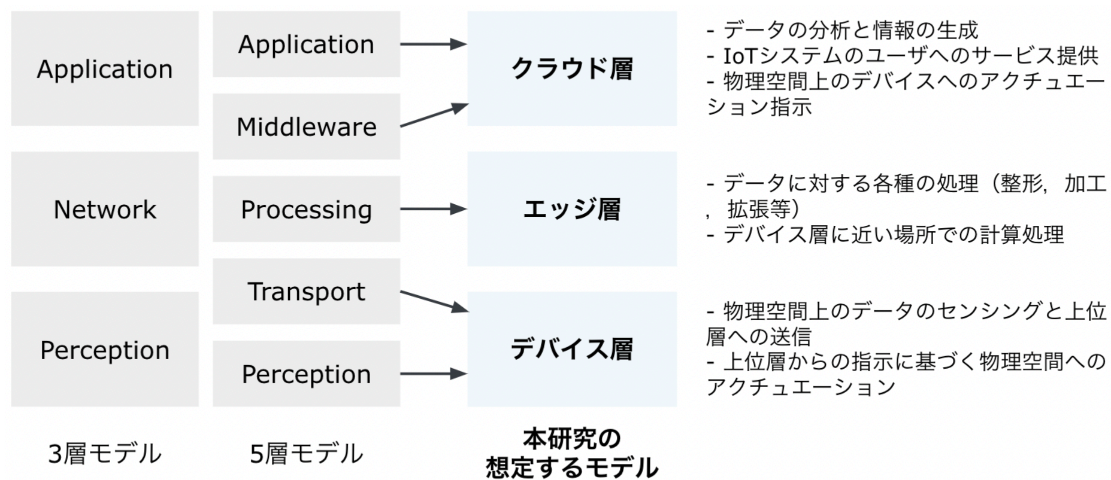

図 1.1: IoT システムのアーキテクチャモデル

1.1.3　3層モデルの利点

IoTシステムは，エッジ層を経由せずにデバイス層からクラウド層へ直接通信するモデルとしても構成し得る．一方で，エッジ層を導入することには，次の利点がある．

1. デバイス層から送信されたデータをエッジ層において一括して処理することで，データへの意味づけを行える [3].
2. エッジ層がデバイス層からの通信を中継することで，デバイス層のセキュリティを担保できる [4].
3. クラウド層が担う役割をエッジ層も担い得る構成とすることで，エッジコンピューティングと呼ばれるアーキテクチャを採用し得る [5].

さらに，エッジコンピューティングの利点について，[5] はレイテンシーの軽減，エッジ層における計算資源の活用，エッジ層からクラウド層へのトラフィックの削減，プライバシーの保護，クラウド層へのネットワーク障害時のシステム継続性の担保の5つを挙げている．

これらにより，IoTシステムをエッジ層を含む3層において構成することには十分な利点がある．

1.2　3層モデルを採用する IoT システムの課題

一方で，それら 3 層からなる IoT システムにおけるデータフローの設計と実装は，デバイス層とクラウド層とが直接通信するモデルと比較すると，複雑なものとなる [6]．その複雑さは，以下の 3 つの課題に由来する．

課題（1）プログラミング言語や通信プロトコルの選択肢が多様である
課題（2）データの取得方式が多様かつデータフローが双方向性を持つ
課題（3）IoT システムの全体を通じたデータフローの見通しが悪くなる

1.2.1　課題 (1）各層の構成・通信の多様性

課題（1）の要因は，各層を構成するソフトウェアの設計・実装に用いられるプログラミング言語に多様な選択肢があることである．また，各層間の通信プロトコルについても同様である．

1.2.2　課題 (2）データ取得方式の多様性

課題（2）の要因は，デバイス層からのデータ取得には push 方式，pull 方式があり [7]，また，必要となる分だけのデータを取得する demand 方式 [8, 9] も提案されており，データ取得の要件に応じて使い分ける必要があることである．その上，クラウド層における分析結果に基づいてデバイスへの操作指示を行うためには，双方向の通信が必要となる．

1.2.3　課題 (3）データフロー・処理記述の複雑性

課題（3）の要因は，課題（2）に加えてエッジ層におけるデータ処理の記述も加わることから，データフローの全体像の記述が複雑なものとなることである．

1.3　本研究の目的

本研究の目的は，IoT システムの設計と実装を複雑にする前述の 3 つの課題を解決することである．そのために，それぞれの課題に対応する以下 3 つの手法を提案することで，3 つの課題のすべてを解決する．

提案（1）3 層を同一のプログラミング言語と通信プロトコルを用いて統合的に設計・実装できる手法
提案（2）push，pull，demand 方式のいずれにも対応し，使い分けられる基盤
提案（3）3 層からなるデータフローを一望のもとに把握できる記法

1.3.1 統合的な設計・実装手法の提案

課題（1）で述べた各層の構成・通信の多様性に関する課題を解決するために，プログラミング言語 Elixir [10] を中核として，3 層を統合的に設計・実装できる手法を提案する．

1.3.2 多様なデータ取得方式への対応

課題（2）で述べたデータ取得方式の多様性に関する課題を解決するために，多様なデータ取得方式にそれぞれ対応しつつ双方向通信も可能にするプロトコルを，筆者の開発した Pratipad [11] によって，Elixir の実行基盤である Erlang/OTP [12] の分散ネットワーク基盤上に定めることで，課題を解決する手法を提案する．

1.3.3 一望のもとに把握可能なデータフロー記法の創出

課題（3）で述べたデータフロー・処理記述の複雑性に関する課題を解決するために，3 層からなる IoT システムのデータフローを宣言的に記述した上で，処理と分離して記述できる記法を提案する．

1.4 論文の構成

本論文の構成を述べる．2 章で，関連研究をまとめた上で本研究を位置づける．3 章で，提案手法について述べる．4 章で，提案手法について評価を行う．そして，5 章でまとめとする．

第2章　関連研究

本章では，1 章で挙げた 3 つの課題に対応する関連研究について，それぞれ 2.1 節，2.2 節，2.3 節で整理する．そして，2.4 節で本研究を位置づける．

2.1　課題（1）にする関連研究

課題（1）は，プログラミング言語や通信プロトコルの選択肢が多様であるということであった．本節では，当該課題に対する関連研究について述べる．

2.1.1　単一の言語による統合的な開発環境

[13] は，IoT システムが複数の層にまたがることによって開発効率が低下するという課題を提示している．すなわち，それぞれの層を担当する専門性の異なる開発者同志が協力しあう必要のあることが，IoT システムの開発効率を損なうとする．当該研究は課題解決のために，独自に開発した言語を用いることで各層の詳細を知る必要なくシステム全体を統合的に開発できる TinyLink 2.0 を提案している．[5] もまた，同様の課題に対する Cross-site edge framework（CEF）と呼ばれる提案を行う研究である．CEF を用いると，各層の実装をプログラミング言語 Python を用いて，単一のファイルに記述できる．

2.1.2　IoT システムのデータ通信プロトコル

[14] は，IoT システムのデータ通信に用いられるアプリケーション層のプロトコルについて，HTTP，CoAP，MQTT，AMQP，XMPP，DDS を取り上げて検討している．IoT システムの各層の間の通信プロトコルの選択については，当該研究の整理したそれぞれのプロトコルの持つ強みと弱みをよく理解し検討した上で，構築を検討している IoT システムに適したものを選ぶ必要がある．また，プロトコルの選定に際しては，セキュリティの担保も必須である．具体的には，通信内容を暗号化することで各層の間の通信経路における秘匿性を担保できること，通信を試みる主体を識別するために認証を行えることが挙げられる．

2.2 課題（2）にする関連研究

課題（2）は，データの取得方式が多様かつデータフローが双方向性を持つということであった．本節では，当該課題に対する関連研究について述べる．

2.2.1 IoT デバイスからのデータ取得方式

[7] は，IoT デバイスからのデータの取得方式について，push 方式と pull 方式とがあるとする．push 方式では，IoT デバイスがネットワークを通じてクラウド上のアプリケーションにデータを送信する．pull 方式では，IoT デバイスに対してネットワークを通じてアクセスすることで，データを取得する．[8] は，push 方式と pull 方式とを組み合わせることで，取得要求に対して必要な分に限ったデータをデバイスから取得する方式を提案している．[9] は，データの流量がデータ処理パイプラインの許容量未満である時に限ってデータ取得要求を送信するバックプレッシャーによる方式を提案している．処理が必要となるデータの流量を一定に抑えられるこの方式を，本研究では demand 方式と呼ぶこととする．

2.2.2 関連するデータ取得方式

[7] は，データ取得方式として publish-subscribe 方式（以下，PubSub 方式）についても言及している．この方式は，デバイス層が特定のトピックに紐づくメッセージをブローカーへ送信すると，ブローカーを通じてそのトピックの購読者（subscriber）へメッセージが発行（publish）されて伝わる．そのため，前述の整理に基づくと，データフローの開始がどの層になるのかという点においては，PubSub 方式は push 方式と同様である．また，PubSub 方式を双方向に組み合わせることで pull 方式も実現可能である．その場合は，データフローはクラウド層から開始されることとなり，本節の整理における pull 方式同様となる．

2.2.3 双方向通信の必要性

クラウド層における分析結果に基づいてデバイスへの操作指示を行うためには，デバイス層からエッジ層を経由してクラウド層へ至る順方向のデータフローだけでなく，その逆方向のデータフローを行える双方向通信も必要となる．そのため，デバイスへの操作指示が要件となる IoT システムでは，前述したデータ取得方式に加えて双方向通信をサポートするプロトコルが必要となる．その際には，WebSocket [15] のような双方向通信に対応したプロトコルを用いるか，2.1.2 節に述べたプロトコルを双方向に組み合わせて双方向性を実現できる．

2.3　課題（3）にする関連研究

課題（3）は，IoT システムの全体を通じたデータフローの見通しが悪くなるということであった．本節では，当該課題に対する関連研究について述べる．

2.3.1　データフローモデル

プログラムをプリミティブな処理を行うノードが入出力を通じて連なる有向グラフとして表現するデータフローモデル [16] は，IoT システムにおいても複数の層を通じて行われるデータ通信を表現するために活用されている．一方で，3 層にわたる IoT システムのデータフローは複雑なものとなり得る．そのため，データフローとその内部で行われるデータへの処理内容とを分離して記述することで，IoT システムの全体を通じたデータフローを見通しよく表現できる提案がなされている．

2.3.2　ビジュアルなデータフロー記述

Node-RED [17] は，IoT システムにおけるデータフローをビジュアルプログラミングによって見通しの良い形で実装するための，Web ブラウザ上で動作するアプリケーションである．GUI 上でノードを繋げていくことで，容易にデータフローを表現できる．また，[6] は Node-RED を拡張した Distributed Dataflow（DDF）および Distributed Node-RED（D-NR）を提案する．DDF および D-NR は，Node-RED を拡張することでデバイス層とクラウド層を含むデータフローを表現できる．

2.4　本研究の位置づけ

本章で検討してきた関連研究は，3 つの課題について部分的に解決策となり得ているが，その全てを解決できるものはない．

TinyLink 2.0 と CEF は，単一の言語と通信プロトコルによって IoT システムを実装できるため課題（1）の解決策として有力であるが，課題（2）に関して整理したデータ取得方式の全ては満たさない．また，データフローとデータへの処理内容とを分離して記述できないため，課題（3）を解決しない．一方で，Node-RED とそれを用いた DDF は，ビジュアルプログラミングを用いたデータフロー記述の提案により，課題（3）の解決策として有力である．しかし，Node-RED は 3 層を統合する実装が行えないため，課題（1）をクリアしない．また，DDF は 3 層を通じた実装を行えるが通信プロトコルとして PubSub 方式に基づく MQTT を用いており demand 方式には対応していないため、課題（2）を解決しない．

表 2.1: 課題と関連研究に基づく本研究の位置付け

番号	課題	関連研究による解決	本研究の位置付け
1	プログラミング言語や通信プロトコルの選択肢が多様である	TinyLink2.0 [13]，CEF [5] 課題 1 のみなら有力な解決策	3 つの課題をすべて解決する手法を提案する
2	データフローの種別が多様かつ双方向性を持つ	TinyLink2.0 [13]，CEF [5]，DDF/D-NR [6] push, pull, 双方向性を実現可能 (Websocket, Pub-Sub) ではあるが demand 方式に対応していない	
3	IoT システムの全体を通じたデータフローの見通しが悪くなる	Node-Red [17]，DDF/D-NR [6] ただし，Node-Red は 3 層を統合する実装は不可	

表 2.1 に，課題と関連研究に基づく本研究の位置づけをまとめた．本研究では，本章で検討してきた 3 つの課題の全てを解決する手法を 3 章で提案する．

第3章　提案手法

本章では，1 章で挙げた 3 つの課題に対応する提案手法について，それぞれ 3.1 節，3.2 節，3.3 節で述べる．

3.1　提案（1）3 層を同一のプログラミング言語と通信プロトコルを用いて統合的に設計・実装できる手法

課題（1）は，IoT システムを構成する 3 層の設計・実装において，プログラミング言語や通信プロトコルの選択肢が多様であるということであった．そこで，本節ではプログラミング言語 Elixir [10] を中核として，3 層を統合的に設計・実装できる手法を提案する．

3.1.1　採用するプログラミング言語

プログラミング言語としては Elixir を，通信プロトコルとしては Elixir の基盤となる Erlang/OTP [12] の TCP による分散ネットワークプロトコル [18] を用いる．デバイス層の実装には，Elixir による IoT デバイス開発基盤である Nerves [19] を用いる．エッジ層の実装には，筆者が Elixir を用いて開発したデータフロー基盤である Pratipad [11] を用いる．クラウド層には，Elixir で開発したサーバーアプリケーションを用いる．

Elixir は動的型付けの関数型プログラミング言語であり，Erlang/OTP の動作する仮想機械（Erlang VM）上で動作するよう設計されている．また，Erlang/OTP の提供する分散ネットワーク基盤も，Elixir から利用可能である．Nerves は，Elixir によって IoT デバイスを開発するためのプラットフォームである．Elixir が Erlang VM 上で動作するのに必要十分かつ最小限の機能を持つ Linux によるファームウェアを提供し，Raspberry Pi [20] や BeagleBone [21] 等を用いた IoT デバイスの開発を迅速に行える仕組みを提供している．Elixir と Nerves を用いることで，デバイス層，エッジ層，クラウド層のいずれをも同一の言語で開発することが可能となる．

3.1.2　採用する通信プロトコル

また，プログラミング言語として Elixir を採用することで，各層の間の通信プロトコルも，Erlang/OTP の提供する分散ネットワークプロトコルに統一できる．Erlang/OTP は大規模な分散システムの構築に利用されてきた実績があるため，IoT システムを構成する言語・通信基盤として十分な堅牢性を有する [22]．

各層の間の通信プロトコルの選定においては，セキュリティを担保できることも必須の要件である．Erlang/OTP は，ノードと呼ばれる Erlang ランタイム間における通信によって分散ネットワークを実現する．提案手法では，デバイス層，エッジ層，クラウド層の各層においてノードを起動し，それぞれのノード間で通信することで IoT システムのデータフローを実現する．ノード間通信においては，TLS（Transport Layer Security）を用いて通信内容を暗号化することで，各層の間の通信経路における秘匿性を担保できる．また，クライアント証明書による認証を用いることで，不正なノードによる通信への介入を防ぐことができる．

3.1.3　実装

前 2 項に述べた，本研究が採用するプログラミング言語と通信プロトコルを用いた IoT システムの実装の概要を図 3.1 の通りまとめた．

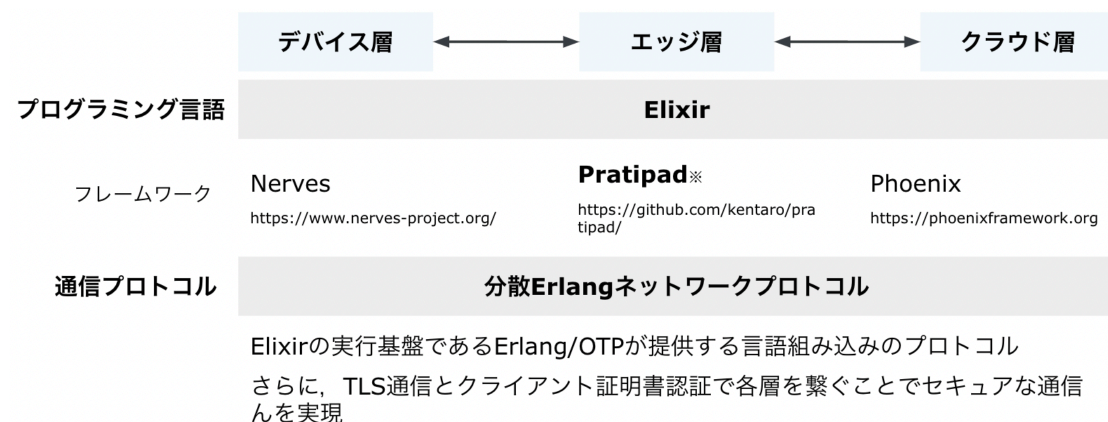

図 3.1: 提案手法を用いる IoT システムのアーキテクチャ

前述の通り，デバイス層およびクラウド層については既存のフレームワークをそれぞれ用いる一方で，エッジ層については筆者が開発したフレームワークである Pratipad [11] を用いる．Pratipad そのものの実装については，3.2.5 項，3.3.5 項でそれぞれ後述する．ここでは，採用した技術を用いた安全な通信の実現について説明する．

Elixir の基盤となる Erlang/OTP が提供する分散ネットワークプロトコルは，元来は信頼できる内部ネットワークでの利用を前提として開発されたため，デフォ

ルト設定のままでは平文で通信が行われること，認証の仕組みがブルートフォース耐性のない方式が用いられている等，広域ネットワークにおいて用いるには問題がある [23, 24].

そのため，4.1 節でのべる提案手法を用いた IoT システムの構築例では，図 3.2 および図 3.3 に例示する設定 [25, 26] を 3 層のそれぞれに施すことで，Erlang/OTP の分散ネットワークプロトコルを用いつつ，TLS 認証による通信経路の暗号化と，クライアント証明書による認証を行うことで，前述の問題を解決している.

```
-proto_dist        inet_tls
-ssl_dist_optfile /etc/<%= System.get_env("PRATIPAD_DEVICE") %>.
   tls.conf
-start_epmd        false
-erl_epmd_port     44300
```

図 3.2: TLS 通信を行うためのデバイス層の設定例

```
[
  {server,
    [{cacertfile, "/etc/ca.crt"},
     {certfile, "/etc/device001.pratipad.local.crt"},
     {keyfile, "/etc/device001.pratipad.local.key"},
     {secure_renegotiate, true},
     {fail_if_no_peer_cert, true},
     {verify, verify_peer}
    ]},
 {client,
    [{cacertfile, "/etc/ca.crt"},
     {certfile, "/etc/device001.pratipad.local.crt"},
     {keyfile, "/etc/device001.pratipad.local.key"},
     {secure_renegotiate, true},
     {fail_if_no_peer_cert, true},
     {verify, verify_peer}
    ]}
].
```

図 3.3: クライアント認証を行うためのデバイス層の設定例

3.2 提案（2）push，pull，demand 方式のいずれにも対応し，使い分けられる基盤

課題（2）は，各層におけるデータの取得方式が多様かつデータフローが双方向性を持つということであった．そこで，本節ではいずれの方式にも対応しつつ双方向通信も可能にするプロトコルを，前述の Pratipad によって Erlang/OTP の分散ネットワーク基盤上に定めることで，課題を解決する手法を提案する.

3.2.1 データ取得方式の設定方法

Elixir のノード間通信においては，相互に接続されたノード同士が Elixir において識別子として用いられるデータ構造である Atom によってメッセージの種別を指定することで，複数の種別のメッセージをやりとりできる．提案手法では，デバイスからのデータ取得においてどのデータ取得方式を用いるか（以下，これをモードと呼ぶ）を設定できるようにする．そして，モードごとのメッセージの種別を定義することによって，同じ基盤を用いながらいずれのデータ取得方式をも扱えるようにする．

3.2.2 push モード

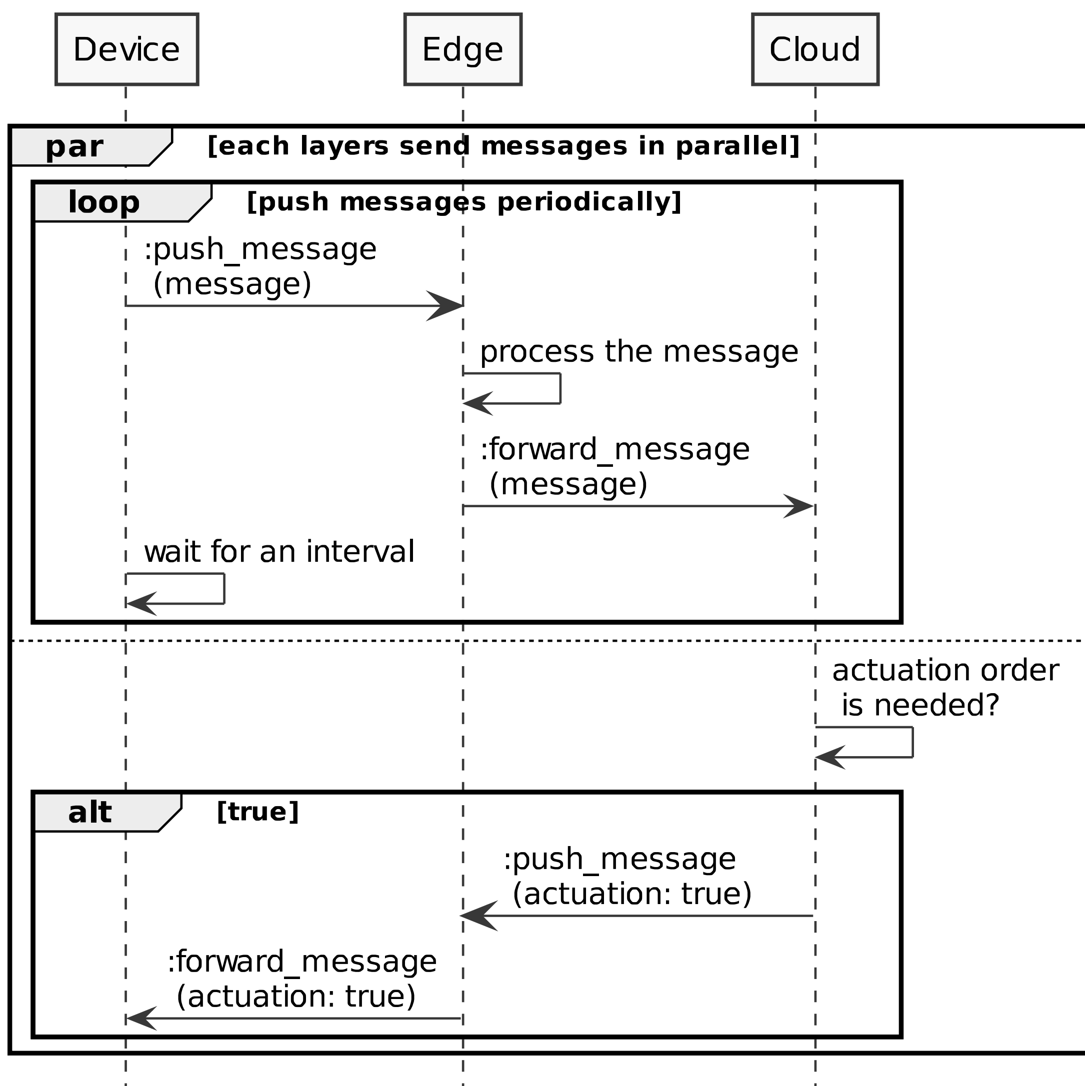

図 3.4: push モードのシーケンス図

push モードのシーケンス図を図 3.4 に示す.

各層のプログラムはそれぞれ独立に動作している．push 方式において，データフローに送り込まれるメッセージ数を決定するのはデバイス層である.

まずは順方向（デバイス層からクラウド層への方向）のデータフローから説明する．デバイス層からメッセージが`:push_message` [1] によって送信されることで，データフローが開始される．エッジ層は，受け取ったメッセージに対して変換・情報の加除・集約等を施した上で，`:forward_message` によりクラウド層へ送信する.

次に逆方向（クラウド層からデバイス層への方向）のデータフローについて説明する．クラウド層からデバイス層へのアクチュエーション指示のようなメッセージを送るためには，まずクラウド層が`:push_message` に引数を渡した上でエッジ層へメッセージを送る．エッジ層は，受け取ったメッセージを`:forward_message` によりエッジ層へ転送する．本研究の想定する IoT システムは階層的なアーキテクチャーであり，順方向と逆方向のデータフローは非対称的である．そのため，本手法では逆方向のデータフローについては，エッジ層における処理を行わない.

3.2.3 pull モード

pull モードのシーケンス図を図 3.5 に示す.

各層のプログラムがそれぞれ独立に動作しているのは push 方式同様である．pull 方式において，データフローに送り込まれるメッセージ数を決定するのはクラウド層である．順方向のデータフローでは，クラウド層からデバイス層へのメッセージ送信を指示するメッセージが`:push_message` に引数を渡した上で送信されることで，データフローが開始される．エッジ層は，受け取ったメッセージを`:forward_messge` によりデバイス層へ転送する．デバイス層は，メッセージを`:push_message` により送信する．エッジ層は，受け取ったメッセージに対して変換・情報の加除・集約等を施した上で，`:forward_message` によりクラウド層へ送信する．逆方向のデータフローについては，push 方式同様である.

[1] この識別子は，前述した Elixir の Atom によって表現されている．以下，同様である.

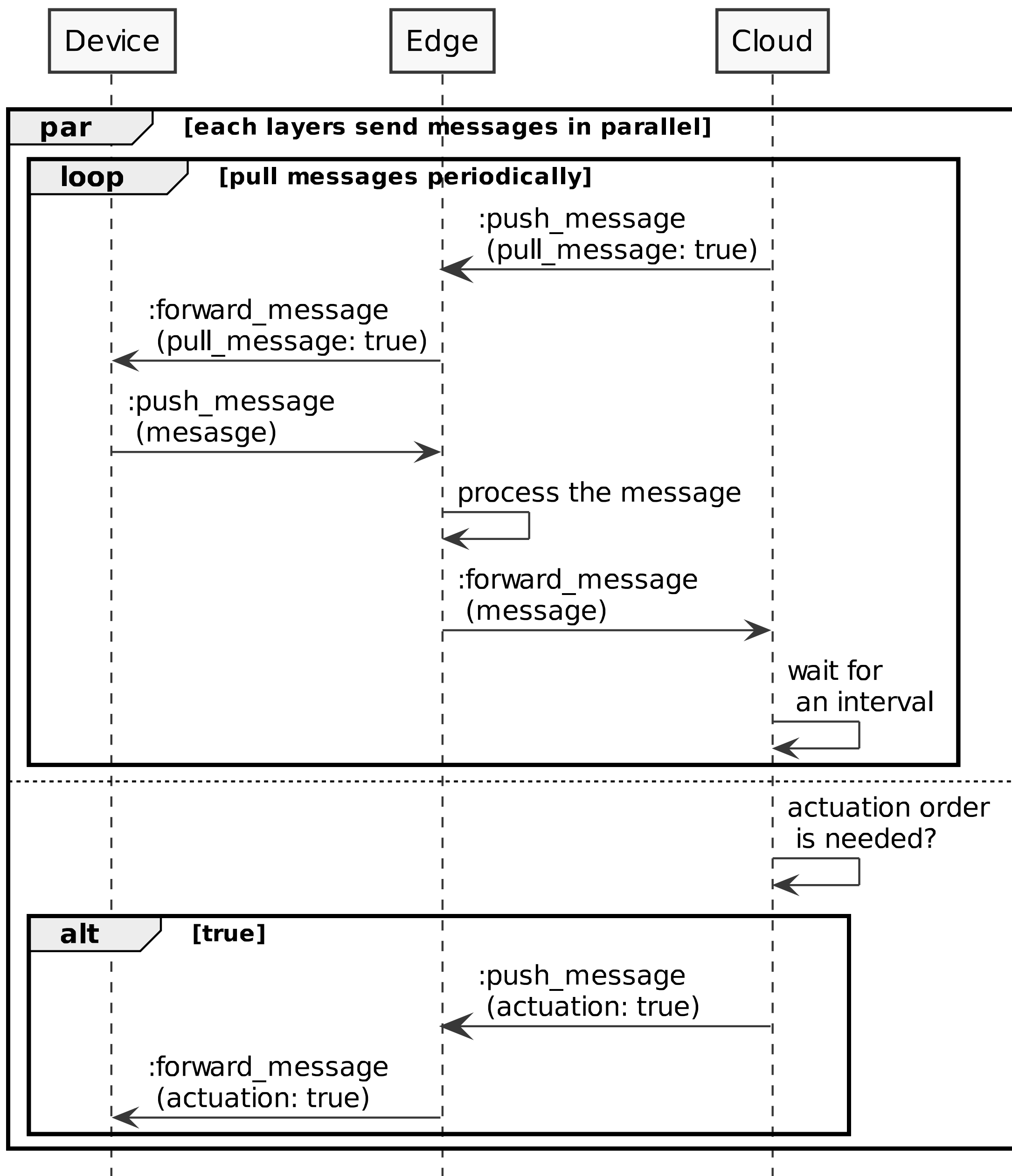

図 3.5: pull モードのシーケンス図

3.2.4 demand モード

demand モードのシーケンス図を図 3.6 に示す.

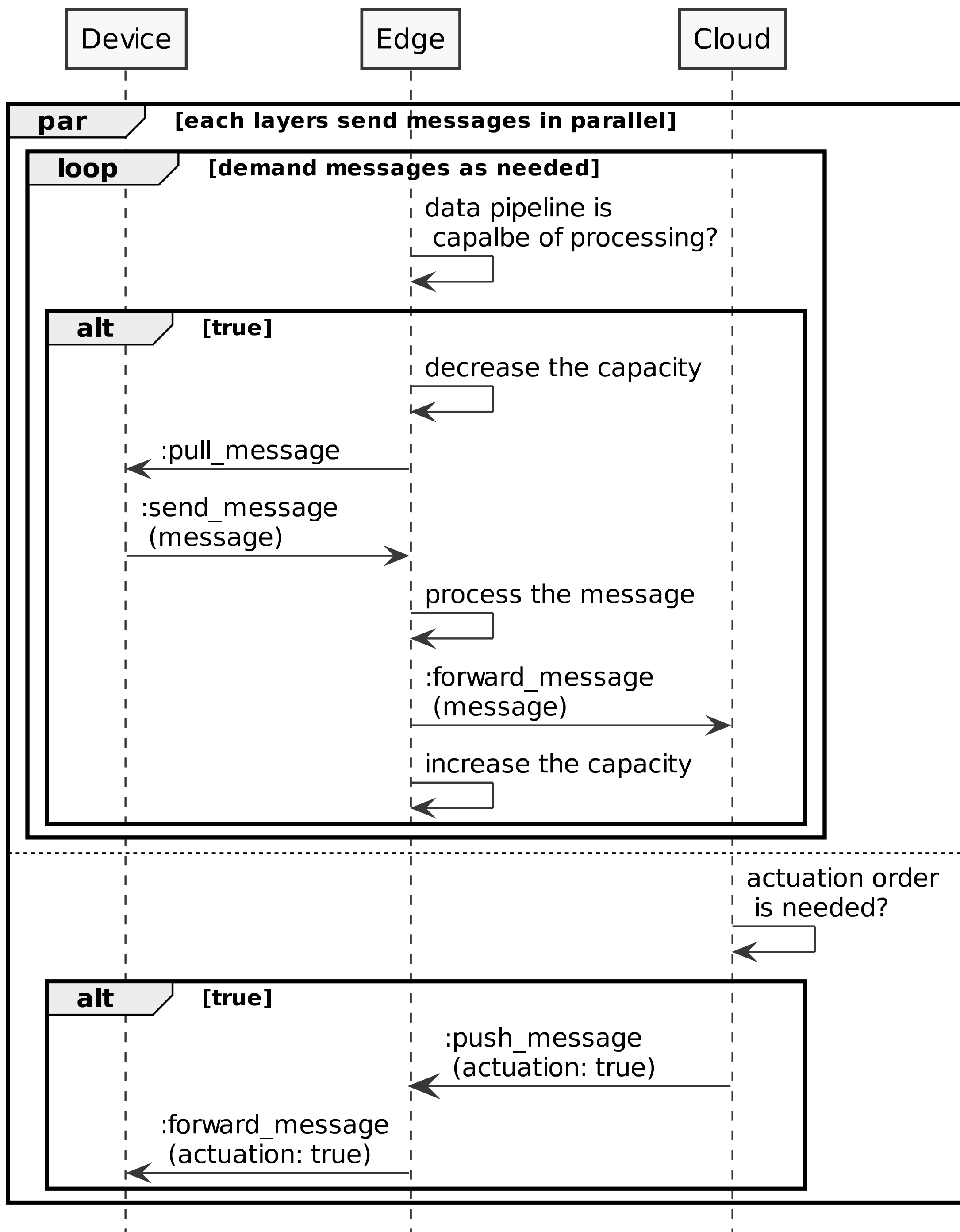

図 3.6: demand モードのシーケンス図

各層のプログラムがそれぞれ独立に動作しているのは前述 2 つの方式同様である．demand 方式において，データフローに送り込まれるメッセージ数を決定するのはエッジ層である．順方向のデータフローにおいて．エッジ層は現在処理されているメッセージ数があらかじめ定められた限度内に収まっているかどうかを確認す

る．もし限度内に収まっていたら処理の余力があると判断し，処理可能なメッセージ数の余力を引き下げた上で，デバイス層へ`:pull_message`によってメッセージの送信指示を行うことでデータフローが開始される．デバイス層は，`:send_message`によってエッジ層へメッセージを送信する．エッジ層は，受け取ったメッセージに対して変換・情報の加除・集約等を施した上で，`:forward_message`によりクラウド層へ送信する．処理が終わったら，エッジ層は処理可能なメッセージ数の余力を引き上げる．逆方向のデータフローについては，前述2つの方式同様である．

3.2.5 実装

上述したプロトコルの実装に際して，Pratipadはデータ処理パイプラインを提供するElixirのライブラリであるBroadway [9] を活用している．Broadwayは，Producer-Consumerパターンを実装しており，さまざまな種類のメッセージに対応したProducerを提供している他，Producerを独自に実装することで必要なプロトコルに対応できる．そこで，Erlang/OTPの分散ネットワーク経由でメッセージを取得できるProducerとして`off_broadway_otp_distribution`を実装した [27]．また，順方向と逆方向とでそれぞれBroadwayのパイプラインを用いることで，データフローの双方向性を実現している．これらを用いて実現した双方向データフローを，図3.7に示す．

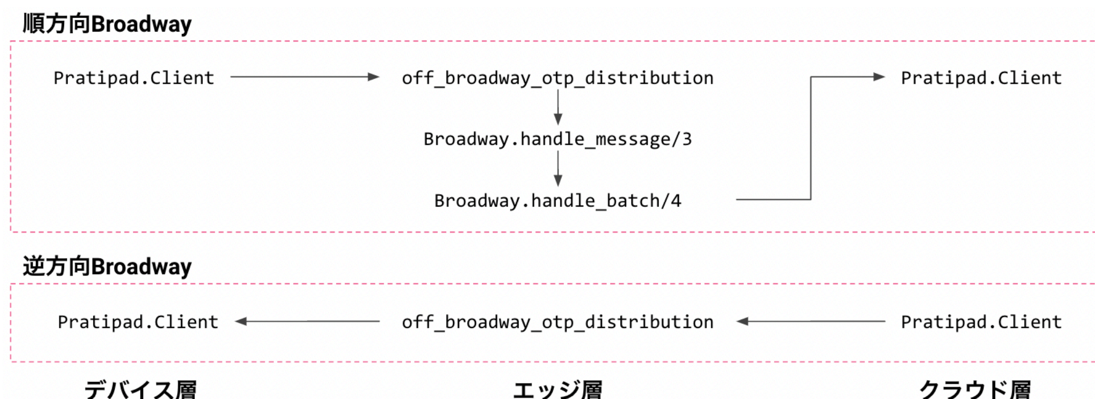

図3.7: 双方向データフローの実装

順方向のデータフローは，前述したErlang/OTPの分散ネットワークプロトコル経由でデータを取得できるProducer実装の`off_broadway_otp_distribution`が，Pratipadのクライアント実装である`Pratipad.Client`を通じてデバイス層からデータの取得することで始まる．取得したデータは，Broadwayを経由して3.3節で後述する処理が行われた後，クラウド層へ送られる．逆方向のデータフローについては，データ処理を行わないことをのぞけば，順方向と同様である．

3.3 提案（3）3 層からなるデータフローを一望のもとに把握できる記法

課題（3）は，課題（2）に加えて，データフローの記述にはエッジ層におけるデータ処理の記述も必要であることから，IoT システムの全体を通じたデータフローの記述が複雑なものとなり，見通しが悪くなるということであった．そこで，本節では 3 層からなる IoT システムのデータフローを宣言的に記述した上で，処理と分離して記述できる記法を提案する．

3.3.1 データフロー記法

提案する記法は，前述の Pratipad によって実装されており，Elixir のコードとして記述，実行できる．記法の一覧は表 3.1 の通りである．これらの記法を用いることで，データフローの全体を一望のもとに把握することが可能となる．

表 3.1: IoT システムのデータフローを記述するために Pratipad が提供する記法

種別	記法	概要
入力	`Push`	push モード
	`Pull`	pull モード
	`Demand`	demand モード
処理	`P`	単一のプロセッサー
	`[P1, P2, ... PN]`	順次適用する複数のプロセッサー
	`{P1, P2, ... PN}`	並列適用する複数のプロセッサー
方向	`~>`	データフローは単方向
	`<~>`	データフローは双方向

3.3.2 記法の種別

提案する記法には，入力，処理，方向の 3 つの種別がある．

入力記法は，前述の提案（2）で述べた 3 つのデータ取得方式を記述する記法であり，それぞれ push モード，pull モード，demand モードに対応する．

処理記法は，エッジ層におけるデータ処理を，処理を担当するモジュール（以下，プロセッサーと呼ぶ）の名前を記述することで表現する記法である．単一の処理のみの場合は，対応するプロセッサー名をひとつだけ記述する．順次適用される処理内容の場合は，プロセッサー名を Elixir のリストによって列挙する．並列適用される処理内容の場合は，プロセッサー名を Elixir のタプルによって列挙する．順次適用は，メッセージ内容への変換・情報の加除等が以前の処理に依存

する一連の処理に用いられ，処理後のメッセージが出力される．並列適用は，メッセージ内容を外部へ送信するといった，メッセージ内容への副作用のない独立して並列的に実行できる一連の処理に用いられ，入力されたメッセージがそのまま出力される．

方向記法は，データフローが順方向のみの単方向であるか，逆方向にも対応した双方向であるかを記述する記法である．

3.3.3 データフローの記述例

図 3.8 に，提案手法を用いたデータフローの記述例を示す．

```
Push <~> P1 <~> P2 <~> P3 <~> Output
```

図 3.8: 提案手法を用いたデータフローの記述例

この例では，データフローへの入力は Push で示される push モードであり，<~> で示される双方向のフローを表現している．入力されたデータに対して，P1 で示されるプロセッサーの処理が適用される．後続の P2 および P3 についても同様に，プロセッサーによって順次処理が適用される．データフロー記述の最後にある Output は，エッジ層で処理されたメッセージがクラウド層へ送信されることを示す．

3.3.4 プロセッサーの記述例

データフローは，図 3.8 のようにひとつのファイル内に記述される一方で，処理記法で記述されたプロセッサー名に紐づく実装は，それぞれ個別のファイル内に記述される．図 3.9 に，提案手法を用いたデータ処理の記述を示す．この例では，図 3.8 のデータフロー内に記述されている P1 モジュールの処理の実装例を記述している．

```
defmodule P1 do
  alias Pratipad.Processor
  use Processor

  @impl GenServer
  def init(initial_state) do
    %{:ok, initial_state)
  end

  @impl Processor
  def process(message, state) do
    # do something with the message
  end
end
```

図 3.9: 提案手法を用いたデータ処理の記述例

プロセッサーは，Pratipad.Processor ビヘイビア[2]の要求する process 関数を実装する必要がある．メッセージの処理時にこの関数が呼ばれることで，処理が行われる．このように，データフローの宣言と処理の詳細の記述を分離することで，複雑なデータフローを一望のもとに把握できる記法を実現できている．

3.3.5 実装

データフローを宣言的に記述できる記法を実現するために，Elixir のマクロを用いて実装を行った．図 3.10 にマクロを用いてデータフロー記法を実装したコードの抜粋を示す．

ここでは，順方向データフローの記法である~>を実装するコードを示している．Elixir のマクロを実装するための defmacro を用いて実装される~>には，データの入力元としてのエッジ層，データを処理するプロセッサ，データの出力先としてのクラウド層が，それぞれ Elixir のモジュール名として引数に渡され得る．関数 handle_unidirectional_op は，引数として渡されたモジュール名をチェックし，引数に即したデータフローを生成する．~>の連鎖によって記述されるデータフローは，再帰的に引数をチェックしながらデータフローを表現する構造体に変換される．

[2]ビヘイビア（Behaviour）とは，モジュールの中で実装しなければならない関数を定める機能である．Java のインターフェイスに相当する．

```
defmacro left ~> right do
  quote do
    handle_unidirectional_op(unquote(left), unquote(right))
  end
end

defp handle_unidirectional_op(left, right) when left == Push or
   left == Demand do
  %Dataflow{
    mode: @input_mode_map[left],
    forward: %Forward{
      processors: [right]
    },
    backward_enabled: false
  }
end
```

図 3.10: データフロー記法のマクロを用いた実装（抜粋）

このようにして作られたデータフローを用いて，データの処理が行われる．前述の 3.2.5 項のデータフローの実装の示す図にデータフローと処理の流れを追加したのが図 3.11 である．

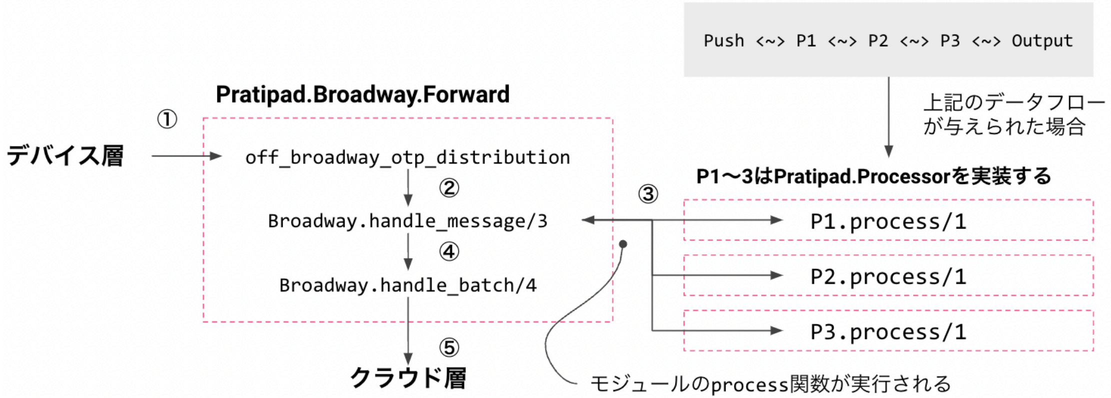

図 3.11: 双方向データフローの実装

前述のマクロで定義された記法を用いて記述されたデータフロー内には，データ処理を行うプロセッサーが複数記述されている．図内の③において示されている通り，プロセッサーの process 関数が次々と実行されることで，データの処理が行われる．

第4章　評価

本章では，3 章で述べた提案手法について，それぞれ 4.1 節，4.2 節，4.3 節で評価を行う．

4.1　提案（1）の評価

3.1 節では，3 層を同一のプログラミング言語と通信プロトコルを用いて統合的に設計・実装できる手法を提案した．本節では，提案手法を用いて実際に 3 層にわたる IoT システムを構築できることを示す．

4.1.1　提案手法を用いた IoT システムの構築例

提案手法を用いた IoT システムの構築例を図 4.1 に示す．

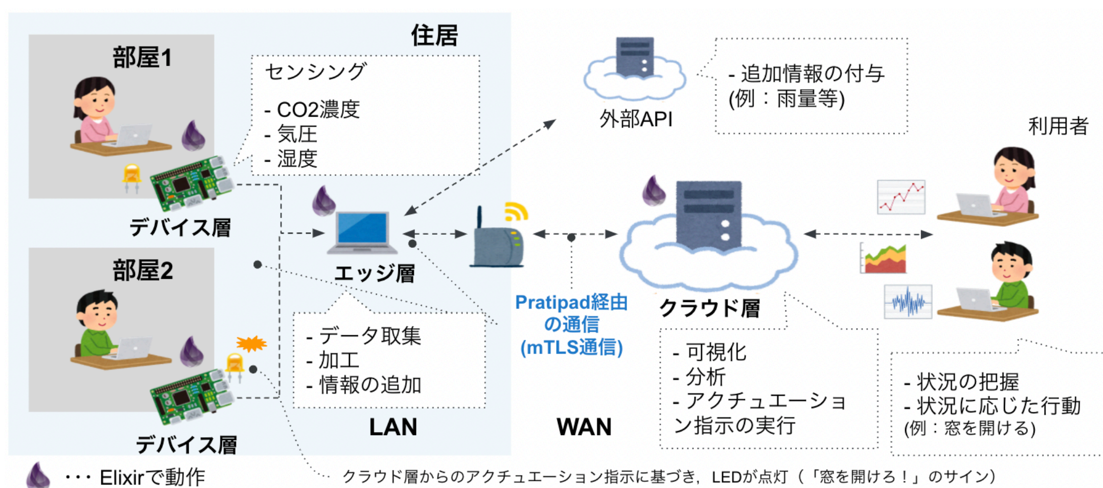

図 4.1: 提案手法を用いた IoT システムの構築例

このシステムは，住居の室内環境を計測し，作業の遂行に不適切な状況である場合に，利用者に対して窓を開けるよう促すためのものである．本システムは，デバイス層，エッジ層，クラウド層の 3 層からなるため，処理の流れを 3 層に沿って説明する．

1. **デバイス層** 住居内の複数の部屋において，センサーを用いて室内環境（二酸化炭素濃度，温度，湿度，気圧）を計測し，エッジ層へ送信する．また，クラウド層からのアクチュエーション指示によって，利用者に室内環境の悪化と窓を開けることの必要を知らせるために，LED を点滅させる．
2. **エッジ層** デバイス層から送信されたセンサーデータに対して，外部の Web API[1]から取得した雨量データを付与した上で，クラウド層へ送信する．室内環境は部屋ごとに異なるが，雨量は住居単位で計測すれば事足りるため，エッジ層で付与することが効率的である．
3. **クラウド層** エッジ層から送信されたセンサーデータに基づき，二酸化炭素濃度が閾値を超える場合で，かつ，強い雨が降っていない場合に，利用者に対して窓を開けるよう促すためにデバイス層に対してアクチュエーション指示を行う．また，直近数時間のセンサーデータの推移をグラフ表示することで，利用者が室内環境の変化を確認できる機能も提供する．

4.1.2 評価

本システムは，3 層をそれぞれ異なるハードウェア，OS によって構成している（表 4.1）（クラウド層については，AWS[2]等のパブリッククラウドを用いたシステム構成が望ましいが，実験環境のネットワークの都合により同一ネットワーク内に配置したハードウェアを用いてクラウド層に見立てている）．

表 4.1: IoT システムの構築例に用いた構成

階層	ハードウェア	OS
デバイス層	Raspberry Pi Zero W	Nerves（Linux ベース）
エッジ層	iMac (24-inch, M1, 2021)	macOS
クラウド層	Raspberry Pi 4 Model B	Raspberry Pi OS

一方で，各層を構成するソフトウェアは Elixir を用いて書かれており，各層間の通信は Erlang/OTP による分散ネットワーク基盤を用いて行われている．また，通信経路は TLS で暗号化されており，クライアント証明書による認証を用いたセキュアな接続を実現している．このことから，提案手法は 3 層構造を持つ IoT システムを統合的に設計・実装できる手法として有効なものであると評価できる．

[1]「JJWD - アメダス最新気象データ API サービス」https://jjwd.info/ を利用した．
[2]https://aws.amazon.com/jp/

4.2　提案（2）の評価

3.2 節では，push，pull，demand 方式のいずれにも対応し，使い分けられる基盤を提案した．本節では，4.1 節で取り上げた IoT システムの構築例をもとに，データ取得方式とデータフローの双方向性について検討することで，提案手法の有効性を評価する．

4.2.1　提案手法の提供するデータ取得方式と双方向性

IoT システムに求められる要件次第で，どのデータ取得方式に利点があるかは異なる．そのため，同じ 3 層からなる IoT システムであっても，目的によってデータ取得方式を使い分ける必要がある．

1. **push 方式** デバイスが取得可能なセンサーデータを余すことなく送信できる．IoT システムに対して，デバイス層において取得したデータの時間分解能の高さが求められる場合，この方式は利点がある．

2. **pull 方式** クラウド層が提供するユーザー向けのアプリケーションに必要なだけのデータを取得できる．IoT システムに対して，ユーザーへ提供するサービスレベルに応じてデータの流量を制御できることが求められる場合，この方式は利点がある．

3. **demand 方式** データ処理を行うエッジ層の余力に応じてデータフローの流量を制御しつつデータを取得できる．IoT システムに対して，リソース制約のもとでの高い可用性が求められる場合，この方式は利点がある．

データフローの双方向性についても，IoT システムの要件によって使い分けられる必要がある．単方向のみに対応している IoT システムを，運用開始後に双方向に対応させることには困難が伴う．そのため，要件の変更を見据えて，容易に双方向性に対応できる方式を用いることが望ましい．

4.2.2　評価

図 4.1 の IoT システムの構築例では，3 つのデータ取得方式，および，データフローの単方向・双方向のいずれにも対応できる．それらを使い分けるために必要な作業は，データフロー記述の変更と，デバイス層とクラウド層で利用するメッセージ送受信プロトコルの変更のみである．このことから，提案手法は様々な要件が求められる IoT システムの構築にとって有効な手法であると評価できる．

4.3 提案（3）の評価

3.3 節では，3 層からなるデータフローを一望のもとに把握できる記法を提案した．本節では，提案手法を用いて 3.2 節で示したデータ取得方式および双方向通信，そして 3.3 節で示したエッジ層における処理の順次適用と並列適用を表現できることを示す．

4.3.1 提案する記法の表現能力

提案手法は，push，pull，demand の 3 つのデータ取得方式に加えて，それぞれのデータ取得方式における単方向，双方向いずれについても表現できる．これらのうちで pull 方式は，クラウド層からデバイス層へのデータ送信指示に基づいてデバイス層がデータを送信するため，双方向通信が可能であることが前提である．そのため，データ取得方式と通信の方向の組み合わせは 5 通りとなる．表 4.2 に示す通り，提案手法はその 5 通りについて全て表現可能である．

表 4.2: 提案手法を用いた各データ取得方式に対応するデータフローの記述

方式	方向	記法
push	単方向	Push ~> P ~> Output
	双方向	Push <~> P <~> Output
pull	双方向	Pull <~> P <~> Output
demand	単方向	Demand ~> P ~> Output
	双方向	Demand <~> P <~> Output

また，提案手法は，エッジ層における処理の順次適用と並列適用についても，それぞれ単独であるいは組み合わせて表現できる．表 4.3 では，順次適用のみ，並列適用のみ，それぞれを組み合わせた処理の記述について，それぞれまとめている（単一のプロセッサーの例については，表 4.2 に挙げているため，省略している）．

表 4.3: 提案手法を用いたエッジ層におけるプロセッサーの記述

方式	記法
順次	Push ~> [P1, P2] ~> Output
並列	Push ~> {P1, P2} ~> Output
順次＋並列	Push ~> [P1, P2] ~> {P3, P4} ~> Output
並列＋順次	Push ~> {P1, P2} ~> [P3, P4] ~> Output

4.3.2　評価

提案手法は，データフローと処理を分離しつつ，データ取得方式，通信の双方向性，エッジ層における処理の組み合わせを十分に表現できる記法を実現していると評価できる．

第5章　おわりに

5.1　本研究の成果

本研究は，3層からなるIoTシステムに複雑さをもたらす課題として,（1）プログラミング言語や通信プロトコルの選択肢が多様であること,（2）データの取得方式が多様かつデータフローが双方向性を持つこと,（3）IoTシステムの全体を通じたデータフローの見通しが悪くなることの3つを提示した.

そして，課題を解決するために，それぞれに対して（1）3層を同一のプログラミング言語と通信プロトコルを用いて統合的に設計・実装できる手法,（2）push, pull，demand方式のいずれにも対応し，使い分けられる基盤（3）3層からなるデータフローを一望のもとに把握できる記法を提案した.

提案手法について,（1）提案手法を用いて3層構造を持つIoTシステムを構築し,（2）提案手法が3種類のデータ取得方式および双方向通信の全てを使い分けることができることを示し,（3）提案手法の提供する記法がデータフローと処理を分離して記述でき，必要なデータフローを十分に表現できることを示すことで，提案手法の有効性を評価した.

5.2　本研究の社会的な意義

本論文執筆時点（2022年1月）においても収束する気配のないCOVID-19（新型コロナウィルス）による社会情勢の変化は，一方においては，社会の様々な領域においてデジタルトランスフォーメーション（DX）を進展させる結果となった.そのような変化を背景に，今後ますますIoTシステムの適用領域が広がっていくと考えられる.

しかし，本研究が課題としたIoTシステムの設計・実装における複雑性は，さらなるDXの進展に対して障害となりかねない．本研究の提案手法は，IoTシステムの効率的な設計・実装の一助となり得ると考える.

5.3　本研究の学術的な意義

本研究では，IoTシステムの設計・実装における複雑性の課題を3つに分類した.関連研究は，課題のそれぞれを個別に解決し得るものではあったが，提案手法は

そのすべてを同時に解決するものである．そのため，IoT システムの持つ課題を解決し，より効果的に設計・実装を行える手法を，具体的な実装とともに示した点において，学術的な貢献を実現できたと考える．

謝辞

本研究に取り組むにあたり，主指導教員としてご指導くださった篠田陽一教授に感謝いたします．ゼミ等を通じたご指導はもとより，ある時の飲み会の席でおっしゃっていた「研究とは『世界観』を打ち出すべきものである」というお言葉が，ともすれば狭くなりがちな私の視野を広げてくださいました．また，副指導教員のリム勇仁准教授，副研究テーマをご担当くださった丹康雄教授にも，ご指導を感謝いたします．

篠田研究室や東京サテライトでご一緒させていただいた学生の皆様にも感謝いたします．在学中の 2 年間，新型コロナ禍のためほとんどがオンラインでの学生生活となってしまいましたが，Slack を中心に議論したり励まし合ったりすることで，なんとか修士論文を形にするところまで来ることができました．

GMO ペパボ株式会社の皆様にも感謝いたします．ペパボ研究所という社内研究所の所長という立場である以上，自らも研究能力を身につけたいという私の思いと取り組みに対して，さまざまな面においてバックアップをしてくださいました．研究開発という面からも事業成長に貢献していけるよう，引き続き精進します．

最後になりますが，妻の桂以子に感謝いたします．ついつい他のことをそっちのけで研究やプログラミングに熱中して家庭をあまり顧みない不祥の夫をおおらかな心持ちで見守ってくれたことで，私は好きなこと，自分がやるべきと信じることに取り組めています．今後ともよろしくお願いします．

ご支援くださった皆様に重ねて御礼を申し上げ，謝辞とさせていただきます．

本研究に関する発表論文

国内会議（査読あり）

- 栗林 健太郎, 三宅 悠介, 力武 健次, 篠田 陽一, IoT システムの双方向データフローにおける設計と実装の複雑さを解消する手法の提案, インターネットと運用技術シンポジウム論文集, 2021, pp.48-55, Nov 2021.

  (優秀論文賞・優秀プレゼンテーション賞を受賞)

参考文献

[1] Alem Čolaković and Mesud Hadžialić. Internet of things (iot): A review of enabling technologies, challenges, and open research issues. *Computer Networks*, Vol. 144, pp. 17–39, October 2018.

[2] Sharu Bansal and Dilip Kumar. Iot ecosystem: A survey on devices, gateways, operating systems, middleware and communication. *Int. J. Wireless Inf. Networks*, Vol. 27, No. 3, pp. 340–364, September 2020.

[3] Fatos Xhafa, Burak Kilic, and Paul Krause. Evaluation of iot stream processing at edge computing layer for semantic data enrichment. *Future Gener. Comput. Syst.*, Vol. 105, pp. 730–736, April 2020.

[4] V Hassija, V Chamola, V Saxena, D Jain, P Goyal, and B Sikdar. A survey on iot security: Application areas, security threats, and solution architectures. *IEEE Access*, Vol. 7, pp. 82721–82743, 2019.

[5] Y Nakata, M Takai, H Konoura, and M Kinoshita. Cross-site edge framework for location-awareness distributed edge-computing applications. In *2020 8th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, pp. 63–68. ieeexplore.ieee.org, August 2020.

[6] N K Giang, M Blackstock, R Lea, and V C M Leung. Developing iot applications in the fog: A distributed dataflow approach. In *2015 5th International Conference on the Internet of Things (IOT)*, pp. 155–162, October 2015.

[7] Karan Mitra, Rajiv Ranjan, and Christer Åhlund. *Context-Aware IoT-Enabled Cyber-Physical Systems: A Vision and Future Directions*, pp. 1–16. Springer International Publishing, Cham, 2020.

[8] Julius Hülsmann, Jonas Traub, and Volker Markl. Demand-based sensor data gathering with multi-query optimization. *Proceedings VLDB Endowment*, Vol. 13, No. 12, pp. 2801–2804, August 2020.

[9] Broadway. `https://github.com/dashbitco/broadway`. （参照：2022-01-16）.

[10] Elixir. https://elixir-lang.org/.（参照：2022-01-16）.

[11] Pratipad. https://github.com/kentaro/pratipad.（参照：2022-01-16）.

[12] Erlang. https://erlang.org/.（参照：2022-01-16）.

[13] Gaoyang Guan, Borui Li, Yi Gao, Yuxuan Zhang, Jiajun Bu, and Wei Dong. Tinylink 2.0: integrating device, cloud, and client development for iot applications. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, MobiCom '20, pp. 1–13, New York, NY, USA, April 2020. Association for Computing Machinery.

[14] Eyhab Al-Masri, Karan Raj Kalyanam, John Batts, Jonathan Kim, Sharanjit Singh, Tammy Vo, and Charlotte Yan. Investigating messaging protocols for the internet of things (iot). *IEEE Access*, Vol. 8, pp. 94880–94911, 2020.

[15] Alexey Melnikov and Ian Fette. The WebSocket Protocol. RFC 6455, December 2011.

[16] Wesley M Johnston, J R Paul Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, Vol. 36, No. 1, pp. 1–34, 2004.

[17] Node-red. https://nodered.org/.（参照：2022-01-16）.

[18] Erlang – distributed erlang. http://erlang.org/doc/reference_manual/distributed.html.（参照：2022-01-16）.

[19] Nerves. https://www.nerves-project.org/.（参照：2022-01-16）.

[20] Raspberry Pi. https://www.raspberrypi.org/.（参照：2022-01-16）.

[21] Beaglebone. https://beagleboard.org/.（参照：2022-01-16）.

[22] Igor Kopestenski and Peter Van Roy. Erlang as an enabling technology for resilient general-purpose applications on edge iot networks. In *Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang*, Erlang 2019, pp. 1–12, New York, NY, USA, August 2019. Association for Computing Machinery.

[23] Joe Armstrong. A history of erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pp. 6–1–6–26, New York, NY, USA, June 2007. Association for Computing Machinery.

[24] Alexandre Jorge Barbosa Rodrigues and Viktória Fördős. Towards secure erlang systems. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang*, New York, NY, USA, September 2018. ACM.

[25] pratipad_example_device. `https://github.com/kentaro/pratipad_example_device`.（参照：2022-01-16）.

[26] Kentaro Kuribayashi. Pratipad: A declarative framework for describing bidirectional dataflow in iot systems with elixir. `https://speakerdeck.com/kentaro/pratipad-a-declarative-framework-for-describing-bidirectional-dataflow-in-iot-systems-with-elixir`.（参照：2022-01-16）.

[27] off_broadway_otp_distribution. `https://github.com/kentaro/off_broadway_otp_distribution`.（参照：2022-01-16）.