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[1,2,a)]　三宅 悠介[1]　力武 健次[3,1]　篠田 陽一[2]

概要：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システムにおいては，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．デバイス層，エッジ層，クラウド層の3層からなるIoTシステムのアーキテクチャーモデルにおいては，設計・実装における構造的な複雑さが課題となる．その要因として（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，（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つがある．本研究は，課題のそれぞれに対して（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する．提案のそれぞれに対して（1）提案手法を用いて3層からなるIoTシステムを実際に統合的に設計・実装できること，（2）提案手法を用いるとデータの取得方式のいずれにも容易に対応できること，（3）提案する記法がデータフロー全体を十分に表現できることを評価することで，提案手法の有効性を示す．

Proposal to Eliminate the Complexity of Design and Implementation in The Bidirectional Dataflow of IoT Systems

Kentaro Kuribayashi[1,2,a)]　Yusuke Miyake[1]　Kenji Rikitake[3,1]　Yoichi Shinoda[2]

Abstract: In IoT systems, the configuration of bidirectional dataflow between physical space and cyberspace is an important issue. In the architectural model of IoT systems, which consists of three layers (device layer, edge layer, and cloud layer), there is a structural complexity. It is caused by three factors: (1) various options of programming languages and communication protocols, (2) various and bidirectional data acquisition methods, and (3) poor visibility of the dataflow throughout the IoT system. We propose three methods for each of these problems: (1) a method that can be designed and implemented in an integrated manner using the same programming language and communication protocol for the three layers, (2) an infrastructure that can support push, pull, and demand methods, and (3) a notation method that can grasp the dataflow consisting of the three layers under a single view. For each of the three proposals, we show that (1) the proposed method allows us to design and implement a three-layer IoT system in an integrated manner, (2) the proposed method can easily handle any of the data acquisition methods, and (3) the proposed notation can sufficiently represent the entire dataflow.

[1] GMO ペパボ株式会社 ペパボ研究所
Pepabo R&D Institute, GMO Pepabo, Inc., Fukuoka City, Fukuoka 810-0001, Japan
[2] 北陸先端科学技術大学院大学
Japan Advanced Institute of Science and Technology, Nomi City, Ishikawa 923-1292, Japan
[3] 力武健次技術士事務所
Kenji Rikitake Professional Engineer's Office, Setagaya City, Tokyo 156-0045 Japan
[a)] antipop@pepabo.com

1. はじめに

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．

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** にまとめた．

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．（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．その目的として [4] 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．

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 [5]．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

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．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 [6]，また，必要となる分だけのデータを取得する demand 方式 [7,8] 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．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．

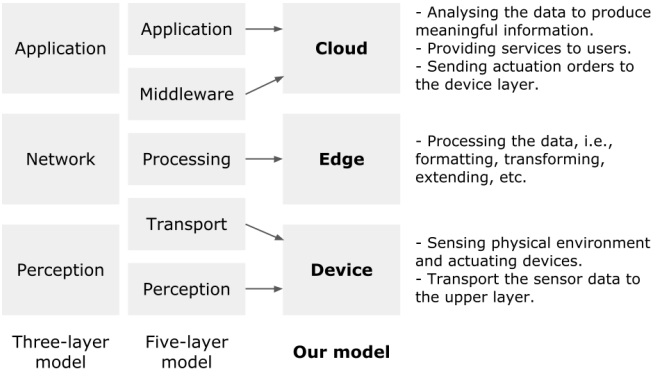

図 1 IoT システムのアーキテクチャモデル
Fig. 1 Architectural models of IoT systems

本研究は，3 層からなる IoT システムに複雑さをもたらす前述の課題を解決するために，3 つの課題にそれぞれ対応する以下の解決策を提案する．

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

提案（1）を実現するために，プログラミング言語 Elixir [9] を用いて，3 層を統合的に設計・実装できる手法を示す．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 [10] を用い，エッジ層の実装には筆者らの開発したデータフロー基盤 Pratipad [11] を用いる．また，各層は Erlang/OTP [12] の分散基盤上に構築し，Pratipad が定めるプロトコルを用いて通信する．提案（2）を実現するために，Pratipad は課題（2）にあげられているどのデータ取得方式にも対応し，双方向通信も可能とする．提案（3）を実現するために，Pratipad は 3 層を通じた双方向のデータ入出力と処理の流れを同一ファイル内で宣言的に記述できる記法を提供する．

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．

2. 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 節で述べる．そして，2.4 節で本研究を位置づける．

2.1 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

[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．[4] 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．

[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）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

[6] 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．[7] 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．[8] 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．

[6] 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．

また，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．

2.3 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

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 [15] 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．Node-RED [16] 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，[5] は Node-RED を拡張した Distributed Dataflow（DDF）を提案する．DDF 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．

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．

3. 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 [9] 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．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 [17] 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 [10] を用いる．エッジ層の実装には，筆者らが 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 [18] や BeagleBone [19] 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．また，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 [20]．

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．

3.2 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．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．

push モードのシーケンス図を**図 2** 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．

pull モードのシーケンス図を**図 3** 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

[*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．

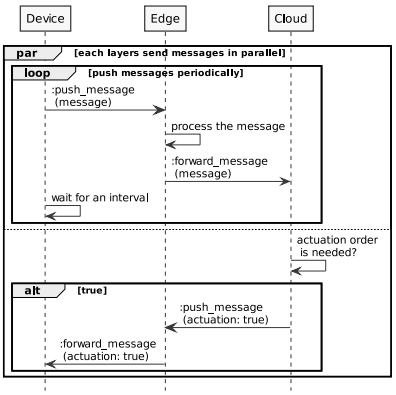

図 2 push モードのシーケンス図
Fig. 2 Sequence diagram of the push mode

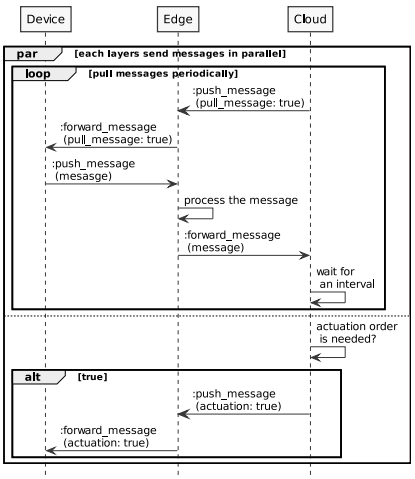

図 3 pull モードのシーケンス図
Fig. 3 Sequence diagram of the pull mode

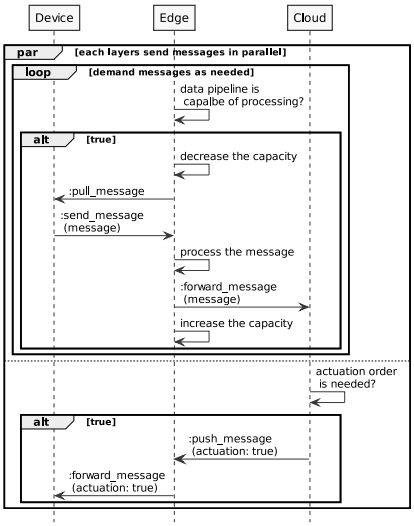

図 4 demand モードのシーケンス図
Fig. 4 Sequence diagram of the demand mode

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．

demand モードのシーケンス図を**図 4** 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．

3.3 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．そのことで，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．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 1** の通りである．

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．

図 5 に，提案手法を用いたデータフローの記述例を示す．この例では，データフローへの入力は Push で示され

表 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
Table 1 Notations supported by Pratipad to describe the dataflow patterns of IoT systems

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

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

図 5 提案手法を用いたデータフローの記述例
Fig. 5 A dataflow example using the proposed method

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．データフローは，図 5 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 6** 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 5 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．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．

4. 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．

[*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．

```
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 6 提案手法を用いたデータ処理の記述例
Fig. 6 A data prosessor example using the proposed method

提案手法を用いた IoT システムの構築例を**図 7** に示す．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[*3]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．

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 2**）（クラウド層については，AWS[*4]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

[*3] 「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．
[*4] `https://aws.amazon.com/jp/`

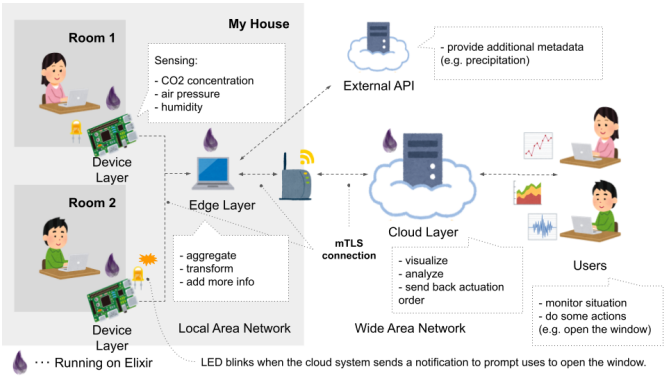

図 7 提案手法を用いた IoT システムの構築例
Fig. 7 An example IoT system using the existing methods

表 2 IoT システムの構築例に用いた構成
Table 2 Configuration used in the example of IoT system

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

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）．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．

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．

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．

図 7 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．

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 3** 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．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** 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 3 に挙げているため，省略している）．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表現できる記法を実現していると評価できる．

5. おわりに

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

表 3 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

Table 3 Description of the dataflow for each data acquisition method using the proposed method

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`

表 4 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

Table 4 Description of the processors in the edge layer using the proposed method

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`

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．

参考文献

[1] Čolaković, A. and Hadžialić, M.: Internet of Things (IoT): A review of enabling technologies, challenges, and open research issues, *Computer Networks*, Vol. 144, pp. 17–39 (2018).

[2] Bansal, S. and Kumar, D.: IoT Ecosystem: A Survey on Devices, Gateways, Operating Systems, Middleware and Communication, *Int. J. Wireless Inf. Networks*, Vol. 27, No. 3, pp. 340–364 (2020).

[3] Xhafa, F., Kilic, B. and Krause, P.: Evaluation of IoT stream processing at edge computing layer for semantic data enrichment, *Future Gener. Comput. Syst.*, Vol. 105, pp. 730–736 (2020).

[4] Nakata, Y., Takai, M., Konoura, H. and Kinoshita, M.: Cross-Site Edge Framework for Location-Awareness Distributed Edge-Computing Applications, *2020 8th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, ieeexplore.ieee.org, pp. 63–68 (2020).

[5] Giang, N. K., Blackstock, M., Lea, R. and Leung, V. C. M.: Developing IoT applications in the Fog: A Distributed Dataflow approach, *2015 5th International Conference on the Internet of Things (IOT)*, pp. 155–162 (2015).

[6] Mitra, K., Ranjan, R. and Åhlund, C.: *Context-Aware IoT-Enabled Cyber-Physical Systems: A Vision and Future Directions*, pp. 1–16, Springer International Publishing (2020).

[7] Hülsmann, J., Traub, J. and Markl, V.: Demand-based sensor data gathering with multi-query optimization, *Proceedings VLDB Endowment*, Vol. 13, No. 12, pp. 2801–2804 (2020).

[8] Dashbit: Broadway: Concurrent and multi-stage data ingestion and data processing with Elixir (online), available from ⟨`https://github.com/dashbitco/broadway`⟩ (accessed 2021-8-18).

[9] Elixir Team: The Elixir programming language (online), available from ⟨`https://elixir-lang.org/`⟩ (accessed 2021-8-18).

[10] The Nerves Project Authors: Nerves Platform (online), available from ⟨`https://www.nerves-project.org/`⟩ (accessed 2021-8-18).

[11] Kentaro, K.: Pratipad: A Declarative Framework for Describing Bidirectional Dataflow in IoT Systems with Elixir (online), available from ⟨`https://github.com/kentaro/pratipad`⟩ (accessed 2021-8-18).

[12] Ericsson AB: Erlang Programming Language (online), available from ⟨`https://erlang.org/`⟩ (accessed 2021-8-18).

[13] Guan, G., Li, B., Gao, Y., Zhang, Y., Bu, J. and Dong, W.: TinyLink 2.0: integrating device, cloud, and client development for IoT applications, *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, MobiCom '20, New York, NY, USA, Association for Computing Machinery, pp. 1–13 (2020).

[14] Al-Masri, E., Kalyanam, K. R., Batts, J., Kim, J., Singh, S., Vo, T. and Yan, C.: Investigating Messaging Protocols for the Internet of Things (IoT), *IEEE Access*, Vol. 8, pp. 94880–94911 (2020).

[15] Johnston, W. M., Paul Hanna, J. R. and Millar, R. J.: Advances in dataflow programming languages, *ACM Computing Surveys*, Vol. 36, No. 1, pp. 1–34 (2004).

[16] OpenJS Foundation: Node-RED (online), available from ⟨`https://nodered.org/`⟩ (accessed 2021-8-18).

[17] Ericsson AB: Erlang – Distributed Erlang (online), available from ⟨`http://erlang.org/doc/reference_manual/distributed.html`⟩ (accessed 2021-8-18).

[18] The Raspberry Pi Foundation: Teach, Learn, and Make with Raspberry Pi (online), available from ⟨`https://www.raspberrypi.org/`⟩ (accessed 2021-8-18).

[19] The BeagleBoard.org Foundation: BeagleBoard.org - community supported open hardware computers for making (online), available from ⟨`https://beagleboard.org/`⟩ (accessed 2021-8-18).

[20] Kopestenski, I. and Van Roy, P.: Erlang as an enabling technology for resilient general-purpose applications on edge IoT networks, *Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang*, Erlang 2019, New York, NY, USA, Association for Computing Machinery, pp. 1–12 (2019).