ストリーム処理を考慮したレイヤ化されたデータ転送管理システム

新平　和礼† 　善明　晃由† 　斎藤　貴文†

† 株式会社サイバーエージェント　〒 101–0021 東京都千代田区外神田 1 丁目 1 8 − 1 3 秋葉原ダイビル 1 3 階
E-mail: †{shinhira_kazunori,zenmyo_teruyoshi,saito_takafumi}@cyberagent.co.jp

あらまし　Web サービスを提供するシステムは日々様々なデータを出力する. これらのデータは BI ツールによるデータ分析や効果測定, 推薦や検索などの他のシステムとの連携など様々な用途で利用される. データを活用するためには大量のデータを目的のシステムに転送しなければならない. その際, データの種別によって利用目途が異なるため, データ毎に適切なシステムへ転送する必要がある. また, 転送中のデータに対し, 不正なフォーマットのデータを排除するフィルタリングや転送先システムに応じたデータ加工などの処理が求められる. 適用する処理についてもデータを生成したシステムやデータの種別, 用途などの条件によって異なるためデータに応じた設定が求められる. データ転送用のミドルウェアを用いることでデータ転送や転送中のデータに対する処理の適用が可能となるが, 転送設定が異なる複数の経路の混在や, 転送と処理が密に結合するなどの理由から転送設定が複雑化するという課題がある. 本稿ではこれらの課題を解決するためデータ転送の責務を分離する階層構造を提案し, 本階層構造に基づくデータ転送を提供する基盤ついて述べる.

キーワード　ストリーム処理, データマネジメント, データ転送

1 はじめに

システムが生成するデータは BI ツールによるデータ分析や効果測定, 外部システムとの連系など様々な用途で利用される. また, システムが出力するデータの利用目的は多岐にわたり, 利用目的によってリアルタイム処理やバッチ処理など異なるシステムでデータを処理する必要がある. そのため, データを活用するためには様々なシステムが出力したデータを目的に応じたシステムに転送しなければならない.

データ転送や転送中の処理の適用を実現するソフトウェアとして Apache Flume や Fluentd などのデータ転送ミドルウェアがあげられる. これらのミドルウェアは到達保証設定に応じたデータ転送の信頼性確保や, ミドルウェアを起動するマシンの追加による処理性能のスケールアウトの機能を提供する. ミドルウェアの機能を利用することで, 各転送経路の信頼性やスケーラビリティを向上できる.

一方, 複数の経路が混在する場合, 経路の増加と共にデータ転送ミドルウェアの設定が複雑になる. その要因として, データの形式や経路選択の設定方法, 適用する処理が異なる点があげられる. データ転送の経路には異なる用途, 形式のデータが混在するため, それぞれのデータから転送先の情報を抽出し適切な宛先に転送する必要がある. また, データ転送においては転送先システムへのデータ投入前に不正な形式のデータを排除したり, 重複したデータを排除するなどの前処理が必要となるケースも存在する. これらの条件が異なる複数の経路が同一のミドルウェアの設定に混在することで設定が複雑化する.

また, 転送ミドルウェアにおける設定では, 転送設定と処理が密に結合する課題がある. 適用する処理内の関数の返り値毎に転送先を設定するなど, 転送先の設定と処理の実装が混在し, 転送経路の把握が困難となる.

本課題を解決するために我々はデータの転送情報を抽象化してレイヤ構造として定義する取り組みを行っている. データ転送のレイヤ化ではデータ転送の責務を 4 つの階層で定義することで, データ転送における経路の選択方法を共通化すると共に, アプリケーションレベルの処理とデータ転送を分離する. レイヤ構造に基づくデータ転送処理基盤を試作し, 本基盤上でデータ転送を行う事例を紹介する.

本稿の構成を以下に示す. 2 章で既存のデータ転送における課題について述べる. 3 章では本稿で提案するデータ転送の定義について説明し, 4 章で開発したデータ転送処理基盤のアーキテクチャについて述べる. 5 章で本システムの適用事例について述べ, 6 章で関連研究について記載する. 7 章で本稿をまとめる.

2 背　　景

本節ではデータ転送の設定が複雑になる要因について述べる.

2.1 転送設定の異なる経路の混在

転送設定の異なる経路が混在することによる複雑化するデータ転送の構成例を図 1 に示す. 本構成では, 拠点 1 に構築されたシステム A, B が生成するデータをストレージとメッセージキューの二箇所へ転送する. 拠点 1 に配置されたシステム A, システム B はそれぞれ異なる運用が行われ, データの形式も異なるものとする.

システム A はヘッダの system フィールドに queue と設定し, システム B は use-case フィールドに batch と設定する. 拠点 1 のミドルウェアではデータのヘッダ領域に含まれる値を確認し, system フィールドが queue のデータをメッセージ

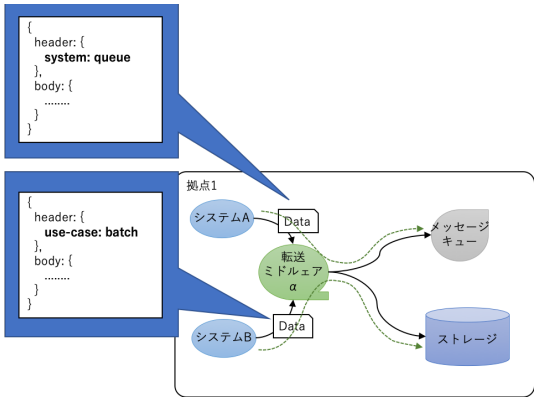

図 1　データ転送の構成例

キューへ, use-case フィールドが batch となっているデータをストレージへ転送する.

```
1  agent.sources = avro1 avro2
2  agent.channels = queue-ch  storage-ch
3  agent.sinks = kafka-sink hdfs-sink
4
5  agent.sources.avro1.type = avro
6  agent.sources.avro1.port = 41414
7  ... 省略
8
9  # 追加設定
10 agent.sources.avro2.type = avro
11 agent.sources.avro2.port = 41415
12 ... 省略
13
14 agent.sources.avro1.selector.type = multiplexing
15 agent.sources.avro1.selector.header = system
16 agent.sources.avro1.selector.mapping.queue = queue-ch
17 ... 省略
18
19 agent.sources.avro2.selector.type = multiplexing
20 agent.sources.avro2.selector.header = use-case
21 agent.sources.avro2.selector.mapping.batch = storage-ch
22 ... 省略
23
24 ... 省略
25 ## channel - file
26 agent.channels.queue-ch.type = file
27 ... 省略
28 agent.channels.storage-ch.type = file
29 ... 省略
30
31 # sinks - kafka
32 agent.sinks.kafka-sink.type =
33     org.apache.flume.sink.kafka.KafkaSink
34 agent.sinks.kafka-sink.channel = queue-ch
35 agent.sinks.kafka-sink.kafka,bootstrap.servers =
36     kafka1.server:9092
37 agent.sinks.kafka-sink.kafka,topic.servers = my_topic
38 ... 省略
39 # sinks - hdfs
40 agent.sinks.hdfs-sink.type = hdfs
41 agent.sinks.hdfs-sink.channel = storage-ch
42 agent.sinks.hdfs-sink.hdfs.path =
43     /flume/events/%y-%m-%d/%H%M/%S
44 ... 省略
```

図 2　拠点 1 の flume-config

転送ミドルウェアに Flume を利用する場合の拠点 1 の転送設定例を図 2 に示す. Flume の設定はデータの入力となる Source, データをバッファリングする Channel, データを出力する Sink の組み合わせによって記述する. また, Flume では Selector と呼ばれる機能を用いることで異なる Channel, Sink にデータを振り分けることができる.

上記例ではシステム A, システム B 用に二つの Source を用意し, それぞれに Selector の設定を行う. 14 行目から 16 行目でシステム A の Selector の設定を行い, データ内の system の値によって異なる Channel を選択する設定を行うことでデータ毎の転送先の振り分けを実現する. システム B に対する Selector 設定は 19 行目から 21 行目に記述され, use-case フィールドの値を参照する.

図 2 の例では振り分けに利用するフィールドによって異なるポートが設定されており, データ形式の違いがネットワーク設定にまで波及している. このため, 送信側でデータの形式に応じてポートを選択する必要があるなど, データ転送の管理が複雑化する.[1]

2.2　複数の拠点をまたぐ転送経路の管理

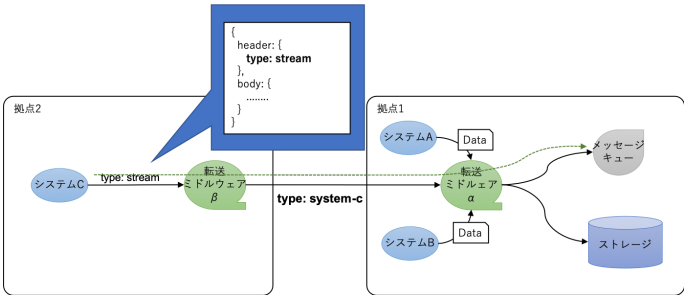

図 3　経路を追加する場合のデータ転送の構成例

複数の拠点をまたぐ転送経路の管理が複雑化する例を図 3 を用いて説明する. 図 3 では図 1 の例に拠点 2 のシステム C からのデータ転送が追加されている. 追加する経路では, システム C のデータを拠点 1 のメッセージキューに格納する. 対象となるシステム C が送信するデータには, ヘッダの type フィールドに stream と設定されているものとし, 拠点 1, 2 のミドルウェアは type ヘッダの値を確認し, stream と設定されているデータのみをメッセージキューへ転送する.

新規経路の転送を追加した拠点 1, 拠点 2 の Flume の設定をそれぞれ図 4, 図 5 に示す. 拠点 1 の設定では拠点 2 から転送されたデータを受信するため 14 行目に新規に Source の設定を追加し, 41416 番のポートでデータを受信する. 追加した Source では type フィールドによる Selector の選択を行うことでメッセージキューとストレージへのデータ格納を実現する.

拠点 2 においても同様に Selector による転送先の選択を行う. 拠点 2 の設定では Selector により type フィールドが stream のデータのみを拠点 1 の 41416 番ポートへ転送する.

このような設定を行うことで複数拠点にまたがったデータ転送経路の構築が可能となる. ただし拠点 2 の転送設定では拠点 1 の 41416 ポートに送信したことしかわからず, 最終的な宛先について把握するためには拠点 1 の転送設定を確認する必要がある. また, 拠点 1 の設定では複数拠点にまたがった経路が混在することで転送経路の増加と共に複雑さが増し全体の転送設定の把握が困難となる.

2.3　処理と転送の結合

データ転送中にフィルタリング処理や圧縮・解凍処理などをストリーム処理で実現することは, 後続のデータ活用を効率化

1：カスタムの selector を実装することでネットワーク設定への影響を軽減できるが, 後述する転送と処理の一体化の問題がある

```
agent.sources = avro1 avro2 avro-dc2
agent.channels = queue-ch  storage-ch
agent.sinks = kafka-sink hdfs-sink

agent.sources.avro1.type = avro
agent.sources.avro1.port = 41414
... 省略

agent.sources.avro2.type = avro
agent.sources.avro2.port = 41415
... 省略

# 追加設定
agent.sources.avro-dc2.type = avro
agent.sources.avro-dc2.port = 41416
... 省略

agent.sources.avro1.selector.type = multiplexing
agent.sources.avro1.selector.header = system
agent.sources.avro1.selector.mapping.queue = queue-ch
... 省略

agent.sources.avro2.selector.type = multiplexing
agent.sources.avro2.selector.header = use-case
agent.sources.avro2.selector.mapping.batch = storage-ch
... 省略

# 追加設定
agent.sources.avro-dc2.selector.type = multiplexing
agent.sources.avro-dc2.selector.header = type
agent.sources.avro-dc2.selector.mapping.stream = queue-ch
... 省略

... 省略
## channel - file
agent.channels.queue-ch.type = file
... 省略
agent.channels.storage-ch.type = file
... 省略

# sinks - kafka
agent.sinks.kafka-sink.type =
    org.apache.flume.sink.kafka.KafkaSink
agent.sinks.kafka-sink.channel = queue-ch
agent.sinks.kafka-sink.kafka,bootstrap.servers =
    kafka1.server:9092
agent.sinks.kafka-sink.kafka,topic.servers =
    my_topic
... 省略
# sinks - hdfs
agent.sinks.hdfs-sink.type = hdfs
agent.sinks.hdfs-sink.channel = storage-ch
agent.sinks.hdfs-sink.hdfs.path =
    /flume/events/%y-%m-%d/%H%M/%S
... 省略
```

図 4　修正後の拠点 1 の flume-config

```
agent.sources = avro
agent.channels = dc2-ch
agent.sinks = avro-sink

agent.sources.avro.type = avro
agent.sources.avro.port = 41414
... 省略


agent.sources.avro.selector.type = multiplexing
agent.sources.avro.selector.header = type
agent.sources.avro.selector.mapping.stream = dc2-ch
... 省略

... 省略
## channel - file
agent.channels.dc2-ch.type = file
... 省略

# sinks - kafka
agent.sinks.avro-sink.type = avro
agent.sinks.avro-sink.channel = queue-ch
agent.sinks.avro-sink.hostname = dc1.server
agent.sinks.avro-sink.port = 41416
... 省略
```

図 5　拠点 2 の flume-config

するために有効である．データ転送におけるストリーム処理ではデータの変換処理をはじめとして，不適切なデータを除くフィルタリングや転送先の分岐などの処理を行う．適用するストリーム処理の例を図 6 に示す．

図 6 の処理では入力データに対してデータを生成したシステムの判定を行い，システム A のデータのみを抽出して後続の処

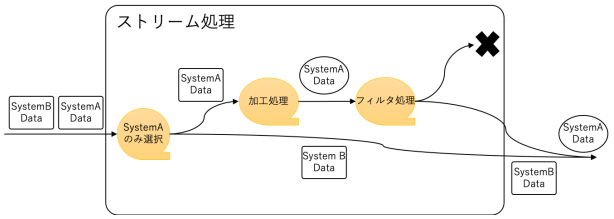

図 6　データ転送の構成例

理を適用し，システム A 以外のデータは無加工のまま出力する．システム A のデータは加工処理を適用した後にフィルタ処理を行い，フィルタを通過したデータのみを出力する．フィルタ処理で False 判定となったデータは本ストリーム処理内で排除し，いずれの宛先にもデータを転送しないものとする．本処理を適用する Flume の設定例を図 7 に示す．

```
## source - avro
agent.sources.avro.type = avro
... 省略

## interceptors
agent.sources.avro.interceptors = myinterceptor
agent.sources.avro.interceptors.myinterceptor.type =
    com.example.flume.interceptor.MyOwnInterceptor$Builder

... 省略
```

図 7　ストリーム処理を適用する flume config

Flume では Interceptor と呼ばれる機能を用いることで転送中のデータに対する処理を適用可能である．本設定では，6 行目において Source に対する Interceptor の適用を行う．ここで，MyOwnInterceptor を図 8 の様に実装することで，変換およびフィルタリング処理が可能となる．

```

public class MyOwnFilter implements Interceptor {

  @Override
  public Event intercept(Event event) {
    if ( !isSystemAData(event) ) {
      return event;
    }

    Event ret = transformFunction(event);
    if (filterFunction(ret)) {
      return ret;
    } else {
      return null;
    }
  }
}
```

図 8　flume-interceptor 実装例

図 8 の実装を行うことで，システム A のデータに対してのみデータの加工とフィルタ処理の適用が可能となる．一方でフィルタの条件や処理は Interceptor の実装として記述されるため，転送中の処理でデータが排除されることを転送ミドルウェアの設定から読み取ることが困難である．

3　データ転送処理のレイヤ化

データ転送における課題の要因として，共通的な転送先の決定手順や宛先情報の定義が存在しないこと，転送中のデータの加工やフィルタ，分岐といった処理内容と転送が密に結合して

いることがあげられる．

これらの課題を解決するため，データ転送の責務を 4 層に分離するレイヤ構造 (以降，データフローレイヤ) を提案する．

データフローレイヤではデータ転送における手続きの共通的な規定と責務の分離を行う．本章では，データフローレイヤの階層構造とレイヤ化による課題解決について説明する．

3.1 データ転送におけるレイヤ構造

データフローレイヤは下位レイヤから順にインフラストラクチャレイヤ，リンクレイヤ，ルーティングレイヤ，アプリケーションレイヤによって構成される．データフローレイヤにおけるデータ転送処理のフローとデータの変化を図 9 に示す．

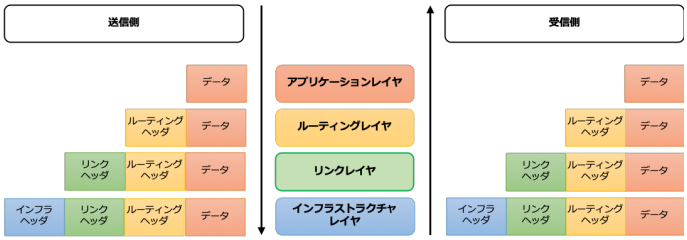

図 9　データフローレイヤの送受信処理イメージ

送信処理では上位レイヤから下位レイヤに向かって処理を行い，処理したデータを下位レイヤに引き渡す．また，各レイヤは上位レイヤから受け取ったデータに対し，処理時にそれぞれのレイヤで必要となる転送情報を付与する．ここで各レイヤで付与する情報をヘッダと呼ぶものとする．

受信処理では下位レイヤから上位レイヤに向かって処理を行い，処理したレイヤを上位レイヤに引き渡す．上位レイヤの定義変更が下位レイヤの転送処理に影響を与えてはならない．受信処理では各レイヤは上位レイヤにデータを引き渡す際，それぞれのレイヤのヘッダ情報を削除することで上位レイヤに必要なデータのみを伝達する．各レイヤの責務の詳細について以下で述べる．

アプリケーションレイヤ アプリケーションレイヤはアプリケーション固有の規定を定める．サービスや用途に応じたデータ形式の規定はアプリケーションレイヤの責務となる．データ転送中のデータの変換やフィルタなどのロジックについてもアプリケーションレイヤで処理される．

ルーティングレイヤ ルーティングレイヤはデータ転送における経路選択の責務を担う．本レイヤで管理する経路情報に従いデータの宛先に対する経路を選択し，次の送信先を決定する．

リンクレイヤ リンクレイヤでは隣接するデータ転送装置間のデータ送受信を管理する．上位レイヤから受領したデータおよび送信先を解釈し，ミドルウェアレベルの宛先情報に変換した上でインフラストラクチャレイヤーにデータと宛先を渡す．

インフラストラクチャレイヤ インフラストラクチャレイヤはデータ転送に利用するミドルウェアを管理する責務を担う．ミドルウェア実装によるデータの送受信を担当し，ミドルウェアレベルでの送達管理を行う．

3.2 レイヤ化による課題解決

3.2.1 転送設定の異なる経路の混在

2.1 節では転送設定が異なる経路が混在する構成を例に，ミドルウェアの待ち受けポート番号を送信側が意識する必要があることを示した．また，これによりミドルウェアの増設やポート番号変更などの設定変更が送信元の設定にも影響を与えることを述べた．

データフローレイヤではこれらの課題を解決するため隣接する転送装置間のデータ転送をレイヤ 1 とレイヤ 2 に分離する．レイヤ 2 では隣接する転送機能間のデータ転送の責務を担い，レイヤ 1 は具体的なミドルウェア実装による転送処理を管理する．

レイヤ 2 はミドルウェアの IP アドレスやポート番号で識別される送信先を抽象化する．同一設定で実行される複数のミドルウェアをレイヤ 2 では一つの宛先としてみなすことが可能となり，レイヤ 1 のミドルウェアにおける設定変更や増設などの変更をレイヤ 2 から隠蔽することができる．

3.2.2 複数拠点間の経路の混在

2.2 節では経路選択の方法が標準化されていないことによる転送設定の複雑化の例を示した．本課題の要因としては転送先を選択する条件がデータ間で統一されていないことやまた，データの最終的な宛先情報および宛先に対する次の転送先の情報が管理されいないことからミドルウェアの設定に異なる条件の経路設定が混在し複雑化することがあげられる．データフローレイヤではレイヤ 2 からレイヤ 3 でデータ転送における経路選択に必要な情報と責務を分離し，共通化することで本課題を解決する．

レイヤ 3 は経路情報を管理し，最終的な宛先と宛先に対する次の転送先の対応関係を保持する．レイヤ 2 は隣接する転送機能間のデータ転送を管理し，レイヤ 1 は具体的なミドルウェアを利用したデータ転送を行う．データ転送に関わる情報と責務を定義することで，データ転送の経路全体を通して共通的な設定方法を提供し，設定の複雑化を防ぐことができる．

3.2.3 転送と処理の密結合

2.2 節で転送設定と処理の結合に関する課題を例示した．本課題の原因はデータ処理における分離や加工が転送設定と密に結合することにある．例えば，処理によって設定される値をミドルウェアの転送設定で参照するなど，ミドルウェアの設定において実装によって決定される値の記述が求められるケースが存在する．データフローレイヤではアプリケーションレベルの処理をレイヤ 4 として定義し，転送処理を行うレイヤ 3 以下と分離する．これにより，アプリケーションレベルの加工や分岐などの処理と転送処理を分け，転送設定の複雑化を解消する．

4 データ転送処理基盤

当社では 3 章で述べたモデルに基づいたデータ転送処理基盤の開発を行っている．本章では試作した基盤のシステム構成について記載する．

4.1 データ転送処理基盤の概要

図 10 に本転送処理基盤によるデータ転送の概要を示す．本基盤は各拠点に配置され, それぞれのデータ転送をストリーム処理として実行する．本基盤で実行する転送処理はレイヤ 2 からレイヤ 4 の処理を行い, 自身が接続するレイヤ 1 のストレージやミドルウェアへデータの書き込みを行う．基盤上のそれぞれの転送処理はレイヤ 1 のミドルウェアによって接続される．

本基盤では, 転送経路やストレージの差異を吸収するため, レイヤ 2 において Source, Sink と呼ばれるインターフェースを提供する．Source はデータの入力を定義するものであり, Sink はミドルウェアやストレージへのデータの出力を定義する．Source, Sink によって転送経路やミドルウェア, ストレージの実装の詳細が隠蔽されるため, 上位レイヤを特定の実装に依存しない形で定義できる．

レイヤ 3 における経路の選択やレイヤ 4 におけるアプリケーション固有の処理の実装を定義するため, 本基盤は Interceptor と呼ばれるインターフェースを提供する．Interceptor は入力データに対する出力データを返す関数として定義される．

4.2 データ転送処理モデル

本節では本基盤において定義できるデータ転送処理のモデルについて述べる．データ転送処理の構成例を図 11 に示す．

本基盤では Source, Interceptor, Sink の各機能の総称を Task と呼称し, 転送処理 は Task とポートの組み合わせで定義する．

ポートは Task の入出力を抽象化した概念であり, Source はミドルウェアから受信したデータ列を出力ポートに書き込み, Sink は自身の入力ポートに書き込まれたデータを接続するストレージやミドルウェアに書き込む．

Interceptor は入力ポートで受信したデータを処理し, 結果に応じて出力するポートを選択する．なお, 入力データに対し常に一つの出力ポートを選択する Interceptor を Transform と呼び, 入力データに対し複数の出力ポートを選択可能な Interceptor を Router と呼称する．また, Router において出力ポートを 2 ポート保持する Interceptor を Filter と呼ぶこととする．

図 12 に本基盤におけるデータ転送処理の設定例を示す．ここで, 本転送処理を実行する基盤では図 13 に示す Source, Sink の設定がなされているものとする．図 13 では Kafka を入力とする Source, および Kafka へ出力する Sink を定義する．これらの定義では Kafka のホスト名, ポート番号, トピック名を定義し, 基盤へ登録する．

図 12 では Kafka から受信したデータに対して Transform の処理を適用し Kafka へ書き戻す転送を定義する．tasks セクションでデータの入出力および変換で利用する Task の一覧情報を記述し, その種別や Task 間の接続情報をポートによって定義する．事前定義した Task を参照することで, 接続する Kafka のホスト名やポート番号, トピック名などのインフラストラクチャレイヤの情報をユーザから隠蔽することができる．これによりユーザは抽象化された入出力情報と処理の接続関係のみに着目してデータ転送を記述することが可能となる．

4.3 システム構成

開発した基盤は 3 章で述べたストリーム処理を実行する機能を提供する．本基盤の構成を図 14 に示す．

本基盤はユーザが定義したデータ転送処理を受領し, 構成情報をパース, ストリーム処理に変換し, Apache Flink などの実行エンジン上で実行することでデータ転送処理を実現する．本基盤はユーザからのリクエストを受け付ける API, 実行エンジンの管理を行う Manager, 複数の Manager や実行中の Pipeline を統括管理する Orschestrator で構成される．Manager は実行エンジンを管理しデータ転送処理を実行する機能部であり, 実行エンジン毎の差分を Manager が吸収することで本基盤では複数の実行エンジンの利用を可能とする．

ユーザはデータ転送処理を行う Pipeline 構成を定義し, API にリクエストを送信する．API は Orchestrator に Pipeline の実行を要求し, Orchestrator は管理する複数の Manager からユーザリクエストに応じて適切な Manager を選択する．Manager はユーザが定義した Pipeline 構成を検証し, 実行エンジンに適した処理に変換して転送処理を起動する．

5 適 用 事 例

本章では 4 章で述べたデータ転送基盤によるデータ転送の適用例について記載する．以降の説明では適用例としてデータの品質チェックを伴うデータの転送を用いる．本処理の構成を図 15 に示す．

本処理では Kafka からデータを取得し, 取得したデータが定められたフォーマットに準拠しているかを判定する．判定の結果に基づき, フォーマットに準拠しているデータとフォーマットに準拠しないデータを異なる宛先へ転送する．

一般的な転送設定例として本処理を行う Flume の設定を図 16 に示す．本設定では 12 行目で処理を適用し, 15-19 行目で処理結果が格納されたヘッダの値によってチャネルを選択する．各チャネルに対して Sink を接続することで Kafka の異なるトピックへの書き込みを行う．Interceptor の実装で決定されるヘッダの値を Selector の設定で指定する必要があり, 実装固有の値を理解して転送設定を行う必要がある．

図 15 の転送を試作したデータ転送処理基盤で実行する際の設定を図 17 に示す．なお, ここで利用した Task は簡単のためルーティングレイヤによる経路選択は行わず, Sink による固定的な宛先へのデータ送信を行う処理として実装した．

図 17 の設定では tasks セクションで転送処理の接続関係を定義する．14-19 行目に記述した Filter 処理でフィルタリング処理を行う．Filter の出力ポートである valid_port, invalid_port でポート名を定義することで後続の処理との接続関係を記述する．ValidationFilter は type が Filter であることから, 分岐を伴う処理であることは本設定から明らかであり, ユーザは出力ポートに対して後続処理を設定することで, 実装で決定する値を意識することなく分岐を含むデータ転送を定義することが可能となる．

ここで, 図 18 に示す Source, Sink を基盤に事前登録してお

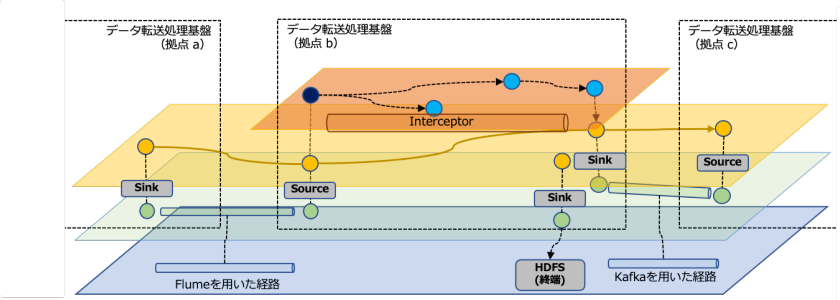

図 10　データ転送処理基盤による転送処理の概要

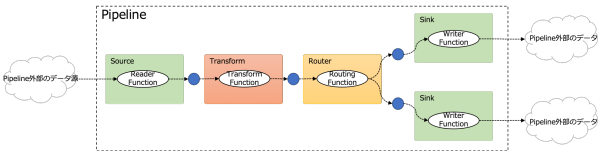

図 11　ストリーム処理の構成

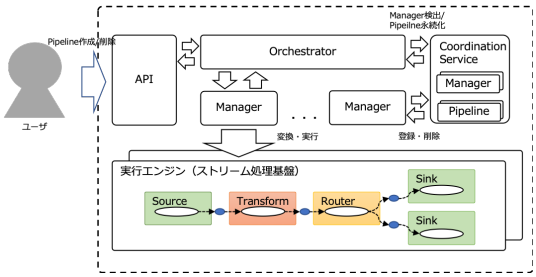

図 14　データ転送処理基板のシステム構成

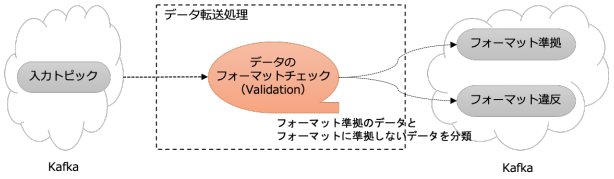

図 15　処理を伴うデータ転送の構成例

```
---
global_option :
  name : "example"
  description : "pipeline for example"
  runner :
    type : FLINK
    parallelism : 3

tasks :
  kafkasource :
    type : Source
    ref: "kafka_source_from_a:"
    output_port : "kafka_source"
  transform :
    type : Transform
    implemented_class : com.example.task.transform.SomeTransform
    input_port: "kafka_source"
    output_port : "transformed"
  kafkasink :
    type : Sink
    ref: "kafka_sink_to_b:"
    input_port : "transformed"
```

図 12　Task を参照するデータ転送処理の定義例

```
---
kafka_source_from_a:
  type: Sink
  implemented_class: com.example.source.KafkaSource
  output_port: "OVERRIDE_AT_RUNTIME"  # 実行時に決定される
  properties:
    topic: "input_topic_a"
    bootstrap_servers: "kafka01.server:9092
---
kafka_sink_to_b:
  type: Sink
  implemented_class: com.example.sink.KafkaSink
  input_port: "OVERRIDE_AT_RUNTIME"  # 実行時に決定される
  properties:
    topic: "output_topic_b"
    bootstrap_servers: "kafka01.server:9092
```

図 13　Task 情報の事前登録

くことでデータ転送処理を記述する利用者からインフラストラクチャレイヤの転送情報を隠蔽することができる．これらは基盤管理者によって設定されるため，データ転送処理を起動する利用者はインフラストラクチャレベルレイヤの接続情報を意識することなく転送設定を記述可能となる．

6　関 連 研 究

6.1　IoT 向けイベント処理基盤

株式会社富士通研究所で開発されている IoT 向けイベント処理基盤 (以降, IoT 向け基盤) [1] [2] は，IoT サービスにおいてデバイスが生成するデータを抽象化し，サービスの容易な変更・改良や異なるサービス間のデータ連携を目的としたイベント処理基盤である．

この基盤では，デバイスから送信されるデータストリームやサービスをオブジェクトとして複数レベルで抽象化することでデバイスとサービスの分離やサービス間の容易な連携，多様なサービスから利用可能なデータの生成を実現している．

本基盤は対象を複数の段階で抽象化し分離するという点で本項で提案するデータフローレイヤの考え方と類似しているが IoT 基盤ではデータの利用を容易にすることを目的とした抽象化であるのに対して本稿ではデータ転送における共通の枠組みを構築することを目的としている点で異なる．

また，IoT 向け基盤は，起動中の処理を停止することなく一貫性を保った処理内容の更新が可能なプラグイン機構を実装しており，運用中の処理の動的変更やスケール性を実現している．本稿で対象とするデータ転送処理はストリーム処理であり，無停止で起動し続けることを前提とすることから IoT 向け基盤で

```
## Name the components on this agent
agent.sources = avro
agent.channels = ok-ch ng-ch
agent.sinks =  ok-kafka ng-kafka

## source - compressed avro
agent.sources.avro.type = avro
agent.sources.avro.channels =  ok-ch ng-ch
... 省略

# Interceptor
agent.sources.avro.selector.interceptors = validation
... 省略
# Selector
agent.sources.avro.selector.type = multiplexing
agent.sources.avro.selector.headers = validation
agent.sources.avro.selector.default = ng-ch
agent.sources.avro.selector.validation.ok = ok-ch
agent.sources.avro.selector.validation.ng = ng-ch


## channel - file
agent.channels.ok-ch.type = file
... 省略

## channel - file
agent.channels.ng-ch.type = file
... 省略

# sinks - ok-kafka
agent.sinks.ok-kafka.type =
    org.apache.flume.sink.kafka.KafkaSink
agent.sinks.ok-kafka.channel = ok-ch
agent.sinks.ok-kafka.kafka.topic = ok-topic
agent.sinks.ok-kafka.kafka.bootstrap.servers =
    kafka01.server:9092
... 省略

# sinks - ng-kafka
agent.sinks.ng-kafka.type =
   org.apache.flume.sink.kafka.KafkaSink
agent.sinks.ng-kafka.channel = ng-ch
agent.sinks.ng-kafka.kafka.topic = ng-topic
agent.sinks.ng-kafka.kafka.bootstrap.servers =
   kafka01.server:9092
... 省略
```

図 16　validation を行う Flume の設定例

```
---
global_option:
  name: "example"
  description: "pipeline for example"
  runner:
    type: FLINK
    parallelism: 3

tasks:
  kafkasource:
    type: Source
    ref: "kafka_source"
    output_port: "kafka_source"
  validation:
    type: Filter
    implemented_class: com.example.task.filter.ValidationFilter
    input_port: "kafka_source"
    valid_port: "validation_ok"
    invalid_port: "validation_ng"
  ok_kafkasink:
    type: Sink
    input_port: "validation_ok"
    ref: "ok_kafka_sink"
  ng_kafkasink:
    type: Sink
    input_port: "validation_ng"
    ref: "ng_kafka_sink"
```

図 17　Validation を行う DSL 定義例

実装される処理内容の更新ロジックは我々の基盤を拡張するうえで有用な機能であると考えられる.

6.2　Apache NiFi

Apache NiFi [3] はシステム間のデータフローを構築することを目的にしたソフトウェアである. Processor と呼ばれるコンポーネントを GUI 上で操作し, 接続することでデータフローの構築が可能となる. データフローを流れるデータに適用する処理を Processor として実装することでも可能で, ユーザの任意の加工処理を実装, 適用することが可能である.

```
kafaka_source:
  type: Source
  implemented_class: com.example.source.KafkaSource
  output_port: "OVERRIDE_AT_RUNTIME"  # 実行時に決定される
  properties:
    topic: "test_input"
    bootstrap_servers: "kafka01.server:9092
ok_kafka_sink:
  type: Sink
  implemented_class: com.example.sink.KafkaSink
  input_port: "OVERRIDE_AT_RUNTIME"  # 実行時に決定される
  properties:
    topic: "ok_topic"
    bootstrap_servers: "kafka01.server:9092
ng_kafka_sink:
  type: Sink
  implemented_class: com.example.sink.KafkaSink
  input_port: "OVERRIDE_AT_RUNTIME"  # 実行時に決定される
  properties:
    topic: "ng_topic"
    bootstrap_servers: "kafka01.server:9092
```

図 18　Source, Sink の事前登録

Apache NiFi はデータフローを構築する上で有用な基盤であり, Apache NiFi を実行エンジンとして利用し, 本稿で示したデータフローレイヤに基づく転送処理を構築することで本稿で述べたデータ転送管理の複雑化を防ぐ効果があると考えられる.

6.3　DataFlow Model

Akidau らが提唱する DataFlow Model [4] では, ストリーム処理で扱う無限に続くデータを有限な区間に区切ることでバッチ処理で扱う有限なデータと同一の処理モデルで処理を記述できることを示した.

DataFlow Model では Window の区切りかたや Window の出力タイミング, 同一 Window に対する複数の出力が得られた際の累積計算方法などを調整することでストリーム処理の正確性, 遅延, コストのトレードオフを図る.

DataFlow Model は個々のストリーム処理における正確性, 遅延, コストの調整を目的としており, データ転送経路の複雑化の解消を目的とした本稿の提案手法とは目的が異なる.

7　ま　と　め

システムが生成するデータは形式や用途によって適用する処理や転送先のシステムが異なる. データ転送経路上には様々な形式, 用途のデータが流れ, 既存のデータ転送ミドルウェアを利用した転送管理では管理する経路が増加するにつれて設定が複雑になる.

データ転送管理における設定の複雑化を改善する枠組みとして, データ転送の情報を階層化して定義するデータフローレイヤを提案し, 本定義に基づくデータ転送処理を行う基盤を試作した. また, 試作した転送処理基盤におけるデータ転送処理の適用事例を示し, データ転送における関心の分離ができることを確認した。

今後は, データフローレイヤーの仕様を詳細に定義し, より大規模な転送管理に適用する予定である.

文　　献

[1] 山岡久俊, 中川岳, 板倉宏太, 高橋英一, 金政泰彦, 植木美和, 武理一郎, “組み立て型のサービス開発を実現する IoT 向けイベント処理基盤の提案,” DEIM Forum 2018 C7-1.

[2] 中川岳, 金政泰彦, 山岡久俊, 板倉宏太, 高橋英一, 植木美和, 松本達郎, 武理一郎, “Flink を用いた動的変更可能な IoT 向けイベント処理基盤,,” DEIM Forum 2018 C7-2.
[3] Apache NiFi, https://nifi.apache.org/index.html.
[4] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernandez-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, Sam Whittle, The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing, Proceedings of the VLDB Endowment, vol. 8 (2015), pp. 1792-1803.