一般社団法人 人工知能学会
Japanese Society for
Artificial Intelligence

人工知能学会研究会資料
JSAI Technical Report
SIG-Challenge-057-12 (11/20)

ロボット聴覚オープンソースソフトウェアHARK用ミドルウェア HARK middleware の紹介

HARK middleware: Middleware for open-sourced robot audition software HARK

木下智義[1*]　中臺一博[2,3]
Tomoyoshi Kinoshita[1]　Kazuhiro Nakadai[2,3]

[1] 株式会社ネットコンパス
[1] NetCOMPASS Ltd.
[2] （株）ホンダ・リサーチ・インスティチュート・ジャパン
[2] Honda Research Institute Japan Co., Ltd.
[3] 東京工業大学
[3] Tokyo Institute of Technology

Abstract: 本稿では，ロボット聴覚オープンソースソフトウェア HARK のミドルウェアとして開発した HARK middleware について紹介する．HARK では，従来，モジュール統合のオーバヘッドが小さいという実時間信号処理の要件を備えた flowdesigner/batchflow をミドルウェアとして採用してきたが，複数のデバイスの利用が難しい，分散処理ができないという制約があった．そこで，flowdesigner/batchflow の利点を踏襲しつつ，こうした問題を解決する HARK middleware を 2018 年にリリースした．しかし，HARK に関する文献の多くは，ロボット聴覚の主要課題である音源定位・分離・認識に関する信号処理アルゴリズムやその実装にかかるものであり，もう一つの特長である HARK middleware に関する文献は存在していない．今回，HARK middleware の pybind11 化，分散処理サポートを行ったので，これを契機に，HARK middleware の概要を紹介する．

1　はじめに

著者らは，これまでに，ロボット聴覚オープンソースソフトウェア HARK (Honda Research Institute Japan Audition for Robots with Kyoto University) を開発し，提供してきた [1, 2]．HARK を用いることで，ロボット聴覚システムをはじめとした音響信号処理を簡便に構築することが可能となり，応用システムも紹介されている [3]．

その一方で，従来の HARK では，C++ベースで記述されているため開発の難易度が高く，データフローがいわゆる pull 型となっているために同時に複数デバイスの I/O 処理を扱うことが難しいなどの制約があった．また，処理が同一プロセス上で完結することが想定されており，別ホストと連携した処理を実現するには，ユーザが連携処理を記述する必要があった．

本稿では，これらの問題を解消しつつ，従来の HARK 機能を実現してきたソフトウェア資産を引き続き利用可能なミドルウェア HARK Middleware（以下 harkmw と呼ぶ）を紹介する．

*連絡先：〒 103-0024 東京都中央区日本橋小舟町 1 番 3 号 6 階
E-mail: kino@netcompass.co.jp

2　関連するミドルウェア

ロボットシステムの構築にこれまで多くのミドルウェアが発表されてきたが，その主要なものとして，ROS および OpenRTM-aist が挙げられる．

ROS[4] は，player/stage [5] を源流に持つ 2007 年にリリースされたミドルウェアで，その草創期は，Willow Garage 社によって開発が推し進められて，ロボット分野のデファクトスタンダードとしての地位を確立した．roscore と呼ばれるブローカを導入し，各ユーザが ROS ノードの作法に従って機能を開発することにより，比較的容易に ROS ノード間結合を実現することができる．しかし，ROS ノード間の連携には常にプロセス間通信を用いる必要があり，またそのためのシリアライズ処理においてもオーバーヘッドが存在する．これに対して harkmw では，ノード間の連携は単一プロセスで動作する限り関数コールのオーバーヘッドしかなく，

データの授受もポインタの受け渡しに相当する形で行われるため，低コストである．

一方，OpenRTM-aist[6] は，国際標準化団体 OMG (Object Manegement Group) [7] で標準化された RT コンポーネントインターフェース仕様に準拠した RT ミドルウエアの参照実装として，2008 年にリリースされ，その後も産総研を中心に研究開発が継続されている．OpenRTM-aist も ROS 同様，ブローカーベースでモジュール間通信を行うアーキテクチャを採用しており，ブローカには CORBA を採用している．CORBA も OMG 標準として汎用的に使われている技術であるが，信号処理ではフレームレベルの数十 ms 単位でのデータを連続的に処理を行っていく点を考慮すると，ROS で挙げた点と同様の問題がある．これに対して harkmw では，先述の通りノード間の連携のオーバーヘッドは小さなものとなっている．

3 harkmw の機能

HARK は，音響信号処理を主として様々な処理を，その構成の変更が容易な形で実現するフレームワークである．HARK は，個々の機能を「ノード」として管理し，ノードを必要に応じて組み合わせることで，任意の処理を実装することが可能である．

各ノードは，1 つ以上の出力と，任意数の入力を「端子」として持っており，ノードの端子を別のノードの端子に接続することで，データの流れを表現することになる．

旧来（バージョン 2.5 以前）は，フレームワークとして flowdesigner/batchflow(以降，flowdesigner と記述する) [8] をベースとしていた．バージョン 3.0 からは，flowdesigner に代わり，新規開発した harkmw を HARK のミドルウェアとして採用している．

harkmw は後述するように C++ で記述された既存の低レベル処理と連携して動作し，またプロセス間通信を用いた高レベルでの連携も同時に行う必要があることから，それらが比較的容易に実現できる Python により記述した．なお，Python による処理は必要最小限に留められており，処理速度の面でも従来のものと遜色なく動作している．

本節では，これらの harkmw の特徴のうち，特に harkmw におけるデータフローと，その実現方法について概説する．

3.1 HARK における処理の構成

HARK では，ノードと呼ばれる処理単位を組み合わせてネットワークを形成することで，処理を定義し実行することができる．

ノードには,「マイクから音声信号を取得する」といった入力機能を備えたもの,「信号に対して FFT を実行して出力する」といった加工等の機能を備えたもの,「WAV ファイルとしてファイルに出力する」といった出力機能を備えたもの，等が用意されている．

また，ノードを組み合わせたものをサブネットとして定義し，あたかもそれが複雑な機能を有したノード (DynamicNode) であるかのように，ネットワーク上に配置することも可能である．

DynamicNode には，subnet と iterator の 2 種類がある．subnet は，ノードを組み合わせてグループ化したものに相当し，ネットワーク内においてより複雑な機能を提供する．

iterator は，一定の条件を満たすまでループする機能を持ったサブネットであり，ループの最後の出力が，サブネット全体の出力となる．信号処理を実装するケースでは，MAIN ネットワーク（ミドルウェアが直接データを要求するネットワーク）に，iterator を 1 つ配置して一連の処理を実行することが多い．

次項では，これらを実現するために従来の HARK が採用していたデータフローと，harkmw におけるデータフローについて述べる．

3.2 従来の HARK におけるデータフロー

本項では，harkmw におけるデータフローの理解のために，従来の HARK におけるデータフローについて述べる．

3.2.1 pull 型アーキテクチャ

flowdesigner におけるデータフローは，いわゆる pull 型であった．すなわち，ネットワークの出力に相当するノードに対して，データを出力するよう要求し，ノードは必要に応じてその前段に相当するノードにデータを要求する．具体的には，各ノードを実装するクラスには `getOutput()` というメソッドが提供されており，下流のノード（あるいは flowdesigner）がこのメソッドを呼び出していた．

この方法では，各ノードは，要求に応じて上流からデータを取得して下流に返すという処理を行えばよく，実装はシンプルにすることができる．一方で，各ノードは自身が処理を開始するタイミングを知ることができず，特に外部との I/O に関わるノードでは問題となることがあった．

3.2.2 `count` 値に基づく時間管理

flowdesigner には `count` という状態値，および `lookAhead`，`lookBack` というパラメータがノードに

備わっていた．

count は，時間方向のフレーム番号を管理する値であり，処理開始時に 0 となり，以降順次インクリメントされる値である．すなわち，前述の getOutput() の引数は count であり，getOutput(count) は，時刻 count のデータを取得するためのメソッド呼び出しということになる．

複数のフレームをまとめて処理を行うブロック処理を行う場合には，時刻 count のデータを生成する場合に必要となる上流データは，時刻 count のものに限定することはできず，処理対象となっている複数フレームにアクセスできるよう時間的に幅を持ったスコープを設定する必要がある．この幅はノードにおける処理に依存しているため，パラメータとして持つこととなる．flowdesigner では，この値を lookAhead, lookBack という値によって管理している．すなわち，lookAhead が 0 でないノードでは，lookAhead フレーム分先のデータを用いて処理をするということであり，lookBack が 0 でないノードでは，lookBack フレーム分過去のデータを用いて処理するということである．

なお，flowdesigner では，lookAhead, lookBack の値は定数であることを想定しており，前後可変長時刻，特に「処理開始からすべて」「データの末端まですべて」という範囲に依存した処理は想定していない．

3.3 harkmw におけるデータフロー

これに対し，harkmw では push 型のデータフローを採用している．この場合，ネットワークの出力から「引き出す」データの取得ではなく，入力側から「流し込む」形で処理が進むこととなる．

harkmw の開発をスタートした時点で，HARK の提供するノードは多くの種類が既に実装され，それらは flowdesigner 用の pull 型を前提としたインタフェース設計となっていた．そこで harkmw では，push 型のデータフローに移行することとしながらも，各ノードの実装を変更するコストを抑制するため，ノードの実装は現状を維持したままとすることが求められた．

また，flowdesigner のオーバヘッドが小さいという利点を踏襲するために，ノード間のデータの送受は原則として従来同様 C++ ポインタの受け渡しとすることとした．データフローの変更により，各ノードの getOutput(count) は同一の count について複数回呼び出されることとなったが，既存のノード実装は内部にバッファ機構を持つものがほとんどであり，繰り返しの処理に対してはキャッシュされた結果を返すことができるため，速度面で不利となることはない．

3.3.1 push 型データフローの実現方法

harkmw では，全体として push 型のデータフローを用い，個々のノードにおいては旧来の pull 型を想定した呼び出し形式を維持することとなった．

これらを両立させる目的で，harkmw においては以下のようなアルゴリズムで各ノードの処理を実行することとした．

1. ネットワーク内で，他のノードに依存しない処理を行うノードを探す．
2. 当該ノードについて，ノード自身の処理を実行した後，以下 3. の処理を実行する．
3. ノードの下流にあるノードについて，以下を実行する．
   (a) ノードの各入力に接続されている上流ノードのそれぞれにつき，ノードの処理で用いるデータが準備できているかを確認する．
   (b) 準備ができていれば当該ノードの処理を実行し，当該ノードの下流にあるノードについて 3. の処理を実行する．
   (c) 準備ができていなければ当該ノードの処理の実行は保留して 3. を終了する．

例えば，図 1 のようなネットワークにおいて，従来の HARK におけるシーケンスは図 2 のようになっていた．harkmw においては，図 3 のようになる．

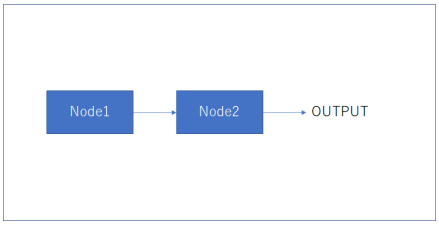

図 1: HARK ネットワークの例（1）

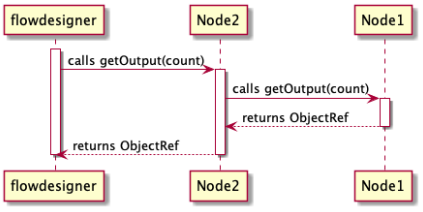

図 2: 従来の HARK におけるシーケンス

ところで，ノードの処理の実行タイミングについて，従来はノード間の getOutput() の呼び出しに依存していたのに対し，harkmw では harmkw が個々のノー

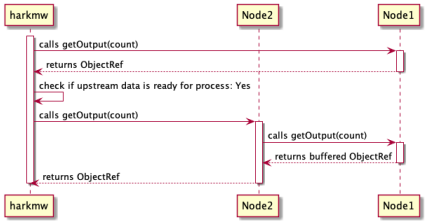

図 3: harkmw におけるシーケンス

ドの getOutput() を呼び出しているため，そのタイミングを制御する余地がある．今後のバージョンでは，I/O 処理を非同期的に実行するなど，より柔軟なタイミング制御機能を追加することも検討している．

3.3.2 count, lookAhead, lookBack の扱い

先述の通り，HARK の各ノードは lookAhead, lookBack というパラメータを持つ．これらの値は，当該ノードがその処理をするにあたって参照するフレームの範囲を定義するものであり，3.3.1 項に示したアルゴリズムにおいて「データが準備できている」の判定に用いられる．

図 4 に示したネットワークでの処理について考える．このネットワークでは，0 でない lookBack を持つノード（Average）が使われている．Average は，直近の指定したフレーム数の値を平均して出力する機能を持つ．また，Temperature は，時刻 count における周辺温度を返すノードである．(Average, Temperature は実際には実装されていないノードであるが，説明のため導入した)．

この場合，Average の上流にある Temperature では，Average からの getOutput(count) の呼び出しでの count 値が単調に増加せず，lookBack の範囲で前後することとなる．

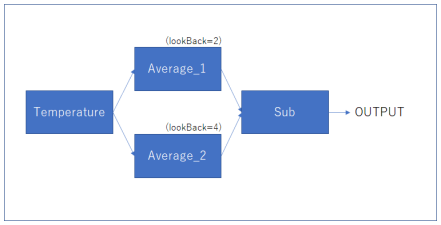

図 4: HARK ネットワークの例（2）

なお，実際のシーケンスは，図 5 のようになる．pull 型のデータフローでは，出力段のノードから引出されるように上流のデータが取得されていることがわかる．

一方，push 型データフローを導入した harkmw では図 6 のようになる．push 型のデータフローでは，上流のノードの処理を最初に実行し，その結果，下流のノードの実行の準備ができた場合に初めて下流の処理が実行されることがわかる．

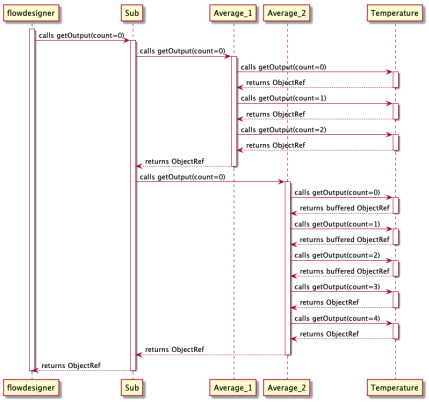

図 5: flowdesigner におけるシーケンスの例（2）

4 harkmw における新機能

harkmw においては，先述のデータフローの変更の他，いくつかの機能が新規に実装されている．本節ではそれらについて説明する．

4.1 複数プロセス実行

従来の HARK では，個々のノードがプロセス間通信の機能を実装するなどの方法を用いない限り，全ての処理は単一プロセスで実行されていた[1]．harkmw では，ネットワークを分割して複数プロセスで実行する機能が追加された．

複数プロセスにて harkmw を実行する場合，処理のプロセスへの割り当ては，ノード単位で行われる．具体的には，flowdesigner においても用いていたネットワーク定義ファイル（.n ファイル）に，どのプロセスで当該ノードを動作させるかを指定する要素を記述することで行う．

先述の通り，harkmw は，ネットワーク内の最も上流にあたるノードを探して処理を開始する．複数プロセスに分割して harkmw を実行する場合，そのようなノードが自プロセスが実行するノードの中には見つからない場合がある．そのような場合には，当該プロセスは，他のプロセスの処理結果に応じて自プロセスにおける処理を開始することとなる．

また，ノードの処理を実行した結果，下流のノードが他のプロセスが実行するものであった場合は，上流

[1]実際に，HARKDataStreamSender, SpeechRecognitionClient, HARK3.2 で新しく導入された HARK-CN などは各ノードで独自にプロセス間通信を実装している

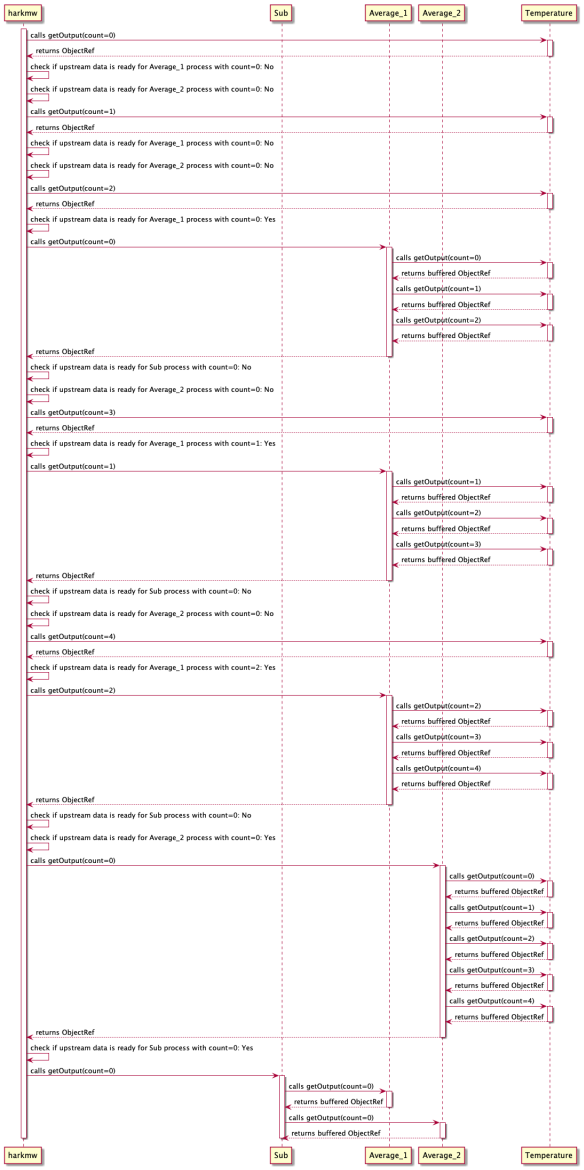

図 6: harkmw におけるシーケンスの例（2）

のノードの結果が得られたことを当該他のプロセスに通知して，他のプロセスにおける処理を開始する必要がある．これらのプロセス間連携処理に，harkmw では MQTT(Message Queuing Telemetry Transport)[9] を用いている．

具体的に，上流のノード A と，下流のノード B が定義され，以下の 2 通りのプロセス割り当ての設定の場合を比較してみる．

1. A，B ともに同一プロセスで実行されるよう設定されている場合
2. A はプロセス P において，B はプロセス Q において実行されるよう設定されている場合

1. の場合，A の処理が完了すると，B の実行可能条件を検証し，実行可能であれば B を実行することは先述の通りである．この時，B の処理の実行時には A の出力データが必要になるが，その受け渡しは，従来の flowdesigner の機構を流用する形で，ObjectRef 型の値を受け渡す実装となっている．これは実質的には C++ のポインタを渡すことに相当し，不要なデータの複製を発生させない効率的なデータの送受と，ノード間の疎結合を両立させている．

一方，2. の場合には，プロセス間でデータの送受を行う必要がある上，各ノードはその上流あるいは下流に存在するノードが同一プロセス上に存在するか，別プロセス上に存在するかを知ることができない．そこで，harkmw では，ノード間にデータの仲立ちを行うノード Proxy を挟むことでこの課題を解決している．すなわち，Proxy の両端に存在するノードが同一プロセス上で処理される場合，Proxy は何もせずに単に ObjectRef 型の値を引き渡すのみである．両端に存在するノードが別プロセス上で処理されている場合，Proxy は制御を harkmw に移し，他プロセスとのデータの受け渡しを実行する．

他プロセスへのデータのを行う際には，各ノードにおいて C++ で生成された ObjectRef 型を，Python 型に変換した上で，pickle したバイナリ列を MQTT の payload に載せる形で送信している．他プロセスからのデータの受信は，逆に，MQTT payload から得た pickle バイナリ列を Python 型に戻した上で，さらに ObjectRef へと変換して用いている．

4.2 harkmw デーモン機能

複数プロセスを用いた分散処理を行う場合，原則としてそれぞれのプロセスをユーザが個別に起動する必要がある．実際の運用を考慮すると，この制約は煩雑になることがあるため，その対処として，harkmw デーモン機能を実装した．

この機能を用いると，分散プロセスを稼働させるホストにおいてあらかじめデーモンを立ち上げておくことで，実際の処理時には，1 箇所で harkmw を起動することで分散プロセス全体を動作させることが可能となる．

5 harkmw のソフトウェア構成

harkmw は python モジュールとして実装されている．本節では，各モジュールおよびクラスについて，その機能をこれまでに述べた処理と対応させる形で説明する．

- module **harkmw.main**

プログラムの引数の解釈と，後述する Process の構築および実行を行う．実質的には，コマンドライン起動を想定した Process のドライバ，と言える．

- class `harkmw.process.Process` プロセス共通で用いる機能を提供する．具体的には，プロセスの初期化と終了の処理，MQTT ブローカとの接続およびデータの送受，等である．
- module `harkmw.defs` HARK のネットワーク定義ファイルを読み込んでメモリ上に展開管理する機能を提供するモジュールである．ネットワーク定義ファイルの読み込みと，実行時にその定義情報を参照する際に用いされる．
- module `harkmw.logic` harkmw の実行時の各種制御を担当するモジュール．
- class `harkmw.logic.builder.Builder` ネットワーク定義ファイルを元に構築された `harkmw.defs` の各クラスの情報を元に，ネットワークを構成する `Node`，`Network` 等を生成する．
- class `harkmw.logic.node.Node` `NativeNode`，`MQTTNode`，および，`Network` の基底クラスであり，共通処理を実装する．
- class `harkmw.logic.mqtt_node.MQTTNode` MQTT を介して動作するノードを管理するクラス．
- class `harkmw.logic.network.Network` ネットワーク，あるいは DynamicNode の処理を行うクラス．いわゆる subnet として動作する場合には単一のフレームについて，iterator として動作する場合には条件が成就するまでループしながら各フレームの処理を行う．
- class `harkmw.logic.proxy.Proxy` ノードとノードの間に存在する `Proxy` であり，Python 型のクラスとして各種処理を実行する他，C++ レベルの `Proxy` との連携も担当する．
- class `harkmw.types.dynamic.Dynamic` HARK の各 toolbox をロードするためのクラス．
- class `harkmw.types.native_node.NativeNode` HARK の各ノードを，C++ で実装されたクラス `harkmwnative.Node` と連携しながら Python から操作するためのクラス．
- module `harkmwnative` src ディレクトリ以下にある C++ ソースファイルによって提供されているモジュールである．C++ および pybind11 を用いて，HARK のノード等機能と python との橋渡しを行う．
- class `harkmwnative.Node` HARK ノードを wrap して Python から操作するためのクラス．
- class `harkmwtative.Proxy` HARK ノード間を接続する proxy ノードを wrap して Python から操作するためのクラス．

6 むすび

本稿では，HARK Middleware (harkmw) を紹介した．harkmw を用いることで，従来の HARK ソフトウェア資産を活用しながら，より柔軟な構成でロボットシステムを構築することが可能となる．

参考文献

[1] HARK Official Site https://www.hark.jp/

[2] K. Nakadai, T. Takahashi, H. G. Okuno, H. Nakajima, Y. Hasegawa, and H. Tsujino. Design and implementation of robot audition system HARK. *Advanced Robotics*, Vol. 24, pp. 739–761, 2010.

[3] 「ドローンが耳を澄まして要救助者の位置を検出〜災害発生時の迅速な救助につながる技術を開発〜」https://www.jst.go.jp/pr/announce/20171207-2/index.html

[4] ROS, http://wiki.ros.org/ja

[5] player/stage, http://playerstage.sourceforge.net/

[6] OpenRTM-aist, https://www.openrtm.org/openrtm/

[7] The Object Management Group, http://www.omg.org

[8] C. Côté, D. Létourneau, F. Michaud, J.-M. Valin, Y. Brosseau, C. Räievsky, M. Lemay, and V. Tran. Reusability tools for programming mobile robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004)*, pp. 1820–1825. IEEE, 2004.

[9] MQTT: The Standard for IoT Messaging, https://mqtt.org/