Henelius, Andreas; Torniainen, Jari

MIDAS

Published in:
SoftwareX

DOI:
10.1016/j.softx.2018.04.004

Published: 01/01/2018

Document Version
Publisher's PDF, also known as Version of record

Please cite the original version:
MIDAS: Open-source framework for distributed online analysis of data streams

Andreas Henelius a,b,*, Jari Torniainen a,c

a Finnish Institute of Occupational Health, Helsinki, Finland
b Department of Computer Science, Aalto University, Helsinki, Finland
c Department of Applied Physics, University of Eastern Finland, Kuopio, Finland

ABSTRACT

Data streams are pervasive but implementing online analysis of streaming data is often nontrivial as data streams can have different, domain-specific formats. Regardless of the stream, the analysis task is essentially the same: features are extracted from the stream, e.g., to employ machine learning and data mining methods. We present the Modular Integrated Distributed Analysis System (MIDAS) for constructing distributed online stream processing systems for heterogeneous data. The MIDAS framework makes it possible to process raw data streams, extract features, perform machine learning and make the results available through an HTTP API for easy integration with various applications. MIDAS is agnostic with regard to the type of data stream and is suitable for multiple domains.

© 2018 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction and significance

Devices in the Internet of Things (IoT) are found in various fields, e.g., in healthcare [1] or in environmental and agricultural applications [2]. IoT is also relevant in Human–Computer Interaction (HCI) (e.g., biosignals for controlling user interfaces).

Many IoT data streams are time series signals with a volume ranging from one channel sampled at 1 Hz (e.g., a temperature sensor) to 16 channels of electroencephalographic (EEG) data sampled at 500 Hz. Time series must be processed sequentially sample-by-sample in contrast to batch processing of data, where the processing order of data items is less important. Regardless of the source of data streams, potentially important information can be extracted from them. Data streams in different domains have varying properties, but the data processing task is essentially identical: features of interest are extracted from the streams and used in decision making. There is hence a need for stream processing systems for online extraction and analysis of streaming signals, that can handle various data formats and high data rates. Since a typical task is to simultaneously collect data from multiple sensors this suggests a distributed system for balancing the computational load.

Current stream processing systems are either domain-specific (e.g., Brain–Computer Interface (BCI) frameworks such as BCI2000 [3], OpenViBE [4] or BCILAB [5]) or automation systems (e.g., QIVICON (https://www.qivicon.com/) or EPICS (http://www.aps.anl.gov/epics/)). Such systems are monolithic and optimised for processing a small number of parallel data streams arriving at a high...
rate in a particular domain. There are also generic data processing frameworks for handling huge amounts of incoming data at a high rate, e.g., Twitter data streams, at massive scales (e.g., Storm (http://storm.apache.org/), Samza (http://samza.apache.org/) and Spark Streaming (https://spark.apache.org/streaming/)). Finally, there are dedicated IoT platforms (e.g., General Electric Predix (http://https://www.predix.com/), PTC Thingworx (https://www.thingworx.com/) and Amazon AWS IoT (https://aws.amazon.com/IoT/) or Kaa (https://www.kaaproject.org/)). Although the distributed data processing systems and the IoT systems are highly capable they are in many cases unnecessary complex to set up for smaller-scale analyses with few data streams, e.g., for prototyping.

The focus of this paper is the cross-platform midas (Modular Integrated Distributed Analysis System) framework, primarily engineered to handle high-velocity time series originating from wearables and IoT sensors for use in HCI applications. The philosophy of the midas framework is to help researchers create and manage setups with multimodal signal sources, enabling them to focus on the signal processing and machine learning aspects of the system. The key design consideration was to create a flexible stream processing system with a distributed architecture for load balancing that is easy to set up and which scales to different types of hardware, such as laptops, desktops and single-board computers.

Midas was engineered with the following design goals in mind, common to systems for online analysis of data streams regardless of the domain.

The system is data agnostic, i.e., different types of data streams can be analysed since streams may provide complementary information. The system is modular and is composed of small autonomous, interconnected units, making it simple to add and remove data streams and analysis components, speeding up the workflow. The distributed architecture ensures scalability and distribution of computational load. Accessibility is provided using an API built on top of standard protocols, e.g., clients to the system can use HTTP. midas is written in Python and is lightweight, e.g., it can be installed in seconds.

Midas does not aim to compete with domain-specific or dedicated high-performing stream processing systems. midas aims to be a free, open-source, lightweight alternative to the other frameworks for creating IoT systems, e.g., setting up real-time analysis pipelines for multimodal time-series data for prototyping or research.

2. Software architecture

Handling data streams essentially consists of (i) data input, (ii) data processing (feature extraction) and (iii) serving data to
clients. Offloading these stages on multiple instances benefits the efficiency and flexibility of the system. This naturally suggests a three-layered design for the high-level architecture of the system (Fig. 1(a)); the input, processing and client layers. The flow of information is from producers on the left to consumers on the right. MIdas comprises the middle processing layer connecting producers and clients.

**MIdas consists of nodes and dispatchers.** The nodes are the processing units for the data streams and contain the signal processing, feature extraction and machine learning functionalities. Nodes process incoming data into metrics i.e., quantities derived from one or several data streams. Data streams can be processed continuously or the node can produce results when requested by a client. Automatic broadcasting of results is also supported. The dispatcher coordinates the communication between clients and nodes.

### 2.1. The input layer

The input layer consists of sensors and streamers. Sensors acquire signals and transmit them using an often vendor-specific protocol over, e.g., Bluetooth, USB, or TCP/IP. Signals must be decoupled from vendor-specific protocols before entering MIdas so generic analysis methods can be applied to the data.

The Lab Streaming Layer (LSL, https://github.com/sccn/labstreaminglayer) is a networked cross-platform protocol for transmission of time series. The LSL allows time synchronisation and discovery of the streams and is the primary data input format from sensors to MIdas.

### 2.2. The processing layer

#### 2.2.1. Node

The nodes receive incoming data streams and perform high-level operations on the data. A node is the main processing unit in MIdas and the most complex component with multiple internal units. All nodes share the architecture of Fig. 1(b).

Nodes can be of two main types, depending on their inputs:

- **Primary nodes** operate on raw signals and receive their data from sensors. Primary nodes are intended for preprocessing signals and feature extraction from data streams.
- **Secondary nodes** utilise the processed information from primary nodes and perform, e.g., machine learning tasks. Secondary nodes may internally gather the data used for analysis, e.g., from user interface events in which case they do not depend on external data streams.

The units inside the node are next described in greater detail. The **receiver** receives incoming data. For primary nodes, the data is typically an LSL stream, whereas secondary nodes request data from primary nodes through the dispatcher. The incoming signal is stored in a data container.

**Data containers** hold a set amount of incoming and processed data, allowing calculations from a time window comprising the past n seconds. Nodes can have two kinds of data containers: primary buffers for incoming raw signal data and secondary buffers for arbitrary refined data, e.g., metrics calculated from data in the primary data buffers. The containers are implemented as multichannel circular buffers with individually configurable parameters.

The **broker** receives messages from clients and performs load-balancing and routes incoming requests to an available responder allowing for concurrency.

A **responder** receives incoming requests and performs different tasks depending on the type of the message. The messaging protocol is described in the MIdas API (https://github.com/bwrc/midas/wiki/API).

Each node has a UDP **beacon** broadcasting status information (IP address, port etc.) for initiating communication with the dispatcher. The dispatcher listens for node beacons and uses the information in the messages to automatically add, remove and reconnect nodes.

The **analysis functions** represent the functional part of the node, e.g., preprocessing, feature extraction and classification. The analysis functions are task-specific and must be implemented by the user. The key design philosophy behind MIdas is to enable the development of analysis methods with minimal effort. The only requirement is that the user must follow the defined format for how data is passed to the analysis function, and it is therefore easy to use existing analysis functions in MIdas. Analysis functions have three modes of operation: (i) metric, (ii) process and (iii) publish. These are described in more detail below.

**Metric functions** are triggered by a request–reply pattern initiated by the client. Analysis functions receive the input data as a function parameter and return a value passed back to the client. The API supports passing arguments to metric functions.

**Process functions** run continuously and typically read from primary data buffers, process the data and push processed values into secondary data buffers. Process functions are optimal for continuous computation of a specific variable, e.g., continuously extracting features from incoming data streams. This can be useful if computations are slow, in which case the node may return the last computed metric to the client reducing wait-time.

**Publishing functions** are similar to process functions and broadcast messages using ZeroMQ (http://zeromq.org). These functions can also act as “watchdogs”, publishing messages if certain conditions are met, e.g., if a metric exceeds a set threshold. The broadcasting is realised using a **publisher–subscriber model**, described next.

The **publisher–subscriber** messaging model in MIdas can broadcast a message, e.g., if the heart rate of the subject exceeds 150 beats per minute (the contingent case) or broadcast the average heart rate every five seconds (the continuous case). Clients can then receive data without requests but this requires client support for ZeroMQ. All messages published by nodes are routed to clients through the dispatcher.

#### 2.2.2. Dispatcher

A distributed system is defined as a collection of multiple independent computers that appears as one from the perspective of the user [6]. MIdas follows a client–server architecture in which the dispatcher handles the bi-directional messaging between nodes and clients. The role of the dispatcher in MIdas is to simplify the communication patterns between components in the system, such as node–node or node–client messaging. Hence, although MIdas is distributed it appears to client applications as one system, since all communication between clients and nodes goes through one (or more) dispatcher. Clients only need to communicate with a dispatcher, which simplifies the configuration process since clients only need to be set up with the dispatcher address and can ignore the full network topology. Multiple dispatchers can be used concurrently without inconsistency problems since a message from a client to a node and back is routed through the same dispatcher. The added benefit of a multi-dispatcher topology is increased robustness, load balancing and redundancy against a single-point-of-failure.

Communication between clients and nodes takes place through a request–reply messaging pattern over HTTP initiated by the client. The communication between the nodes and a dispatcher is realised using an internal messaging protocol implemented on
Fig. 2. Distribution of RTTs in a multi-client M
das network using one (blue) and two (lilac) dispatchers. Upper and lower hinges in the Tukey-style box plot denote quantiles 1 and 3 with the median in the middle. Whiskers denote the range of points within 3/2 times the interquartile range from the corresponding hinge. Outliers are marked with circles. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

top of ZeroMQ. These network communication protocols allow a distributed architecture where all M
das components (nodes, clients and dispatchers), can be run on different computers.

2.2.3. Discovery of system topology and fault tolerance

The discovery of the system topology in M
das is automatic and based on the node beacons. Nodes can be added and removed during run-time without reconfiguring the system. Automated discovery also adds reliability as the connection to a node in an offline state can be automatically restored once the node becomes online. The dispatcher can be configured to only communicate with a subset of nodes, which can be used to partition nodes into multiple M
das networks, each with their own dispatchers.

Redundant and load-balanced system topologies where several nodes perform identical tasks can be built using M
das. A client may then request a metric from one of several nodes yielding the same result. Running multiple dispatchers further increases the reliability, redundancy and balances the load.

2.2.4. Synchronisation and time accuracy of data streams

Synchronisation of data streams is especially important for analyses utilising shared information between different data streams. One way of accounting for network-related latencies is the round-trip time (RTT) metric describing the network latency between the components and the time required to process the request. In the LSL protocol each sample is time-stamped at the source and the LSL can constantly measure the RTT between the streamer and the receiving node. Network latencies calculated from the RTT can then be used to synchronise data streams from different streamers. M
das provides methods for determining the RTT between nodes and the dispatcher.

2.3. The client layer

The client layer interfaces with M
das using an HTTP API, making it easy and flexible to request data from the M
das network. Both requests and responses are in the JSON format.

3. System performance

The performance of stream processing systems, such as M
das, is in general difficult to evaluate since performance is affected by factors such as network topology, configuration of individual components and the type and amount of data being processed. These factors all contribute to the overall latency of the system.

The latency of each request can roughly be attributed to the time for routing the request and reply between the client and the target node, and to the processing time of the node.

To evaluate the performance of M
das under normal operation conditions, we investigated the extreme case where a single node receives requests from multiple clients: 1, 5, 25, 125, 250 and 500. We limit our investigation to the network latency of a "dummy" node with ten responders and no processing capabilities. The performance test was performed using (i) one dispatcher and (ii) two dispatchers. The node and the dispatchers use ten responders and threads for incoming queries, respectively. Each client contacts the node in 100 ms intervals over a time period of 200 s. The distribution of round-trip-times (RTTs) was calculated from the last 100 requests. The network used for testing consisted of four computers (one for the node, two for the dispatchers and one for the clients) connected using wired gigabit Ethernet. The testing setup allowed determination of the network latency which can be considered additive to the actual computational load of the nodes.

The results are displayed in Fig. 2. The median RTT using a single dispatcher is around 3 ms when the number of clients is 250. At 500 clients with one dispatcher, the median RTT and standard deviation increases. Adding a second dispatcher balances the load and the median RTTs remain stable.

4. An illustrative example

We demonstrate the typical usage of M
das using a system for online determination of mental workload. The source code is omitted here for brevity, but the full source code for this and other examples can be found in the supplementary material repository (https://github.com/bwrc/midas-softwarex/). We use synthetically generated data (https://github.com/bwrc/lsltools) with one electrocardiogram (ECG) channel and two EEG channels. The data from about $t = 20$ s–70 s and $t = 120$ s–275 s (see Fig. 3(b)) has a higher average heart rate and a higher brainbeat index [7]. The decision rule for mental workload fuses average heart rate and the brainbeat index: if both metrics exceed a set level the workload is “high”, otherwise “low”. As thresholds for the decision rule we used 1.5 for brainbeat and 65 for the average heart rate. The architecture is shown in Fig. 3(a). The signals are streamed from the sensors to primary processing nodes. The ECG node extracts average heart rate from the ECG signal and the EEG node calculates the brainbeat index. The client requests the average heart rate and
brainbeat index from the past 5 s and outputs the mental workload class (“high” or “low”). The result is shown in Fig. 3(b).

5. Impact

MIDAS has been used in several projects for integrating online signal analysis into different applications. MIDAS has been used to process EEG signals (refer to [8] for a demonstration of this project predated MIDAS) and is a supported input protocol for eye data in an eye-tracking application [9]. MIDAS was used to develop a prototype showcasing adaptation of an information seeking system based on psychophysiological metrics. MIDAS was discussed in a review of psychophysiological methods for HCI [10, Section 3.12]. Section 3.12 in that paper was written by the present authors and discusses solutions for online signal processing. A MIDAS tutorial on online analysis of psychophysiological signals was held in conjunction with the 4th International Workshop on Symbiotic Interaction. In 2015 MIDAS was demonstrated at a booth in the Upgraded Life Festival in Helsinki. The Finnish Institute of Occupational Health participates in the Finnish Defense Force’s Research Programme 2017–2020. The plan is to monitor physiological activity online using MIDAS on wearable computers.

We encourage users of MIDAS to share their processing nodes by issuing pull requests to the repository https://github.com/bwrc/midas-nodes, so users working on similar research tasks can benefit from the open source nature of this project.

6. Conclusions

MIDAS is a modular and generic framework for online distributed analysis of data streams, agnostic with regard to the type of data streams. The system is suitable for multiple domains, i.e., MIDAS has a high generic applicability. This, together with the lightweight and modular nature of MIDAS allows rapid development of distributed applications.

MIDAS provides a high-level abstraction allowing the user to focus on implementing analysis functions without having to consider, e.g., communication patterns and data handling. MIDAS is well suited for research use, e.g., in human–computer interaction.

Acknowledgements

This work was supported by the Revolution of Knowledge Work project, funded by Tekes (grants 40228/13 and 5159/31/2014) and by the Academy of Finland (decision 313513).

References


2 https://youtu.be/6YL2UIH5bcU.


