MARVELO – A Framework for Signal Processing in Wireless Acoustic Sensor Networks

Haitam Afifi1, Joerg Schmalenstroeer2, Joerg Ullmann2, Reinhold Haeb-Umbach2, Holger Karl1

Computer Networks Group1, Department of Communications Engineering2, Paderborn University
Email: {haitam.afifi, karl}@uni-paderborn.de, {schmalen, haeb, ullmann}@nt.uni-paderborn.de

Abstract

Signal processing in Wireless Acoustic Sensor Networks (WASN) is based on a software framework for hosting the algorithms as well as on a set of wireless connected devices representing the hardware. Each of the nodes contributes memory, processing power, communication bandwidth and some sensor information for the tasks to be solved on the network.

In this paper we present our MARVELO framework for distributed signal processing. It is intended for transforming existing centralized implementations into distributed versions. To this end, the software only needs a block-oriented implementation which MARVELO picks-up and distributes on the network. Additionally, our sensor node hardware and the audio interfaces responsible for multi-channel recordings are presented.

1 Introduction

Distributed signal processing in sensor networks has attracted a lot of attention in the past years [1, 2]. During the early phase of this research topic, authors usually selected exemplary algorithms, e.g., a beamformer, and investigated how to distribute the computational task on a network of nodes. Investigated algorithms include distributed Kalman filters [3, 4], distributed Delay-and-Sum Beamformer (DSB) [5], distributed Minimum Variance Distortionless Response (MVDR) beamformer [6], acoustic source localization [7] or generalized sidelobe canceller [8].

In later publications, more general approaches were presented which might be less task-specific and thus easier to adapt to other problem statements. Bertrand et al. presented the class of Distributed Adaptive Node-specific Signal Estimation (DANSE) algorithm for Minimum Mean Square Error (MMSE) estimators [9, 10]. A distributed Expectation-Maximization (EM) algorithm was discussed in [11] and distributed compressive sensing is the topic of [12], just to name a few signal processing techniques that could be useful in a variety of scenarios and tasks.

However, these strategies all target the same idea: First they estimate parts of the problem locally and, subsequently, try to find a global optimum by averaging across the local results. For this task, decentralized algorithms like gossiping [13] can be applied. Another interesting idea for distributing an estimation problem was presented by Vlachos et al. in [14] where a correlation matrix was locally estimated in parts and the missing parts were reconstructed.

All publications mentioned above try to optimize their approaches to work on a specific hardware at low data rate, limited memory and computational power. Some consider data rate-preserving strategies, e.g., [9], or transform a bandwidth-consuming task (signal distribution) into a computation power-consuming one (matrix reconstruction), e.g., [14].

The ongoing progress in hardware development for embedded hardware reduces the need for algorithm optimization, since every generation offers more performance at reduced costs.

Power consumption is out of our scope, because these devices are not only constrained to some extent. However, these strategies are only partial and might be less task-specific and thus easier to adapt to other problem statements. Bertrand et al. presented the class of Distributed Adaptive Node-specific Signal Estimation (DANSE) algorithm for Minimum Mean Square Error (MMSE) estimators [9, 10]. A distributed Expectation-Maximization (EM) algorithm was discussed in [11] and distributed compressive sensing is the topic of [12], just to name a few signal processing techniques that could be useful in a variety of scenarios and tasks.

However, these strategies all target the same idea: First they estimate parts of the problem locally and, subsequently, try to find a global optimum by averaging across the local results. For this task, decentralized algorithms like gossiping [13] can be applied. Another interesting idea for distributing an estimation problem was presented by Vlachos et al. in [14] where a correlation matrix was locally estimated in parts and the missing parts were reconstructed.

All publications mentioned above try to optimize their approaches to work on a specific hardware at low data rate, limited memory and computational power. Some consider data rate-preserving strategies, e.g., [9], or transform a bandwidth-consuming task (signal distribution) into a computation power-consuming one (matrix reconstruction), e.g., [14].

The ongoing progress in hardware development for embedded hardware reduces the need for algorithm optimization, since every generation offers more performance at reduced costs.
to the next blocks in the processing queue, as defined by the controller. Hence, they are responsible for performing the algorithmic work on the nodes.

Algorithm blocks are encapsulated in independent processes, enabling parallel processing of multiple blocks controlled by the Linux kernel scheduler.

Communication between controller and clients (Fig. 1) takes place through two different ports. The CMD ports are exclusively used for internal communication between controller and clients. For example, commands are sent through these ports to start/stop a process, and to check reachability of other clients in the network. Additionally, it can be used to collect the system status of the clients. The MSG ports are used to monitor the progress of each processing algorithm and terminalize their output. This is required for debugging or supervision.

Messages from all processes running on a given node are collected by a local Messaging module. It forwards the messages through its MSG port to the controller, which can display all debugging logs using a special Debug module.

2.2 Management and Configuration

Process management and algorithm configuration are based upon a single Extensible Markup Language (XML) file (compare example in Fig. 2). It is a machine and human-readable representation describing the distribution of the blocks. The XML file specifies the individual nodes on which the processing blocks of the algorithm are hosted, defines the routes between the blocks (i.e., inputs and outputs) and states parameters of the blocks (e.g., number of channels).

The root element of the XML document is network. It contains multiple child elements of type node, each representing a single hardware device defined by the attribute pi_id. This attribute can either be an IP address or a hostname.

Each node has one or multiple children elements of type algorithm. They define and parametrize the processes for encapsulating single processing blocks, including the information from where the block receives the input, and to where it will send its output. Furthermore, the attributes path and executable are set to define the binary to be executed and where to find it. The path attribute refers to a local folder on the controller containing all data required for processing the block, including the binary (attribute executable) and all additional data. Note that MARVELO automatically sends all files and folders including all sub-folders to the respective nodes, so a user does not have to take care about how to push the data to the nodes for synchronization.

A child element algorithm knows three optional children elements input, output and parameter. The latter can be used to specify any input parameter to be passed during the invocation of the binary (e.g., string or int), whereas the chronological order is retained in case of several parameter elements. The input element has an attribute source_pi_id, which specifies the node from where the input is received. The attribute has the same node’s pi_id if the input is received from a process running on the same node. Additionally, the attribute pipe_id is used to define the pipe between data sending and data receiving processes. A corresponding function has the attribute output, where target_pi_id specifies the destination’s pi_id and pipe_id needs to be consistent with the input elements attribute pipe_id of the process running at the destination.

Pseudocode of a process: Initialization method for parsing command line arguments and block oriented data processing method.

2.3 Communication

A process wraps an algorithm with networking parameters for data forwarding, while the Daemon module is responsible for the connections between processes.

In Sec. 2.2 we show how to manage and configure a process. Accordingly, when writing the code for a process (Fig. 3), input and output pipes need to be declared and opened for enabling the data transmission between the processes. A process is created by the local Daemon by executing the executable followed by the parameter attributes and the pipe file descriptors. So each block has to implement a parser for getting the input and output file descriptors that corresponds to the process related pipes. Examples on how to parse arguments can be found in [20] (C language) and [21] (Python) and follow standard best practices.

Then, we read the input data from the input pipes, pass it to the process’s internal function executing the actual algorithm (i.e., runFunction” in Fig. 3), and write the results to the output pipe.

A communication can either take place between processes on
the same node (Fig. 4a) or between processes on different nodes (Fig. 4b). Node’s internal communication is efficiently realized by pipes, whereas inter-node links require networking techniques. In the later case the pipe’s output is forwarded via TCP/UDP connections to communicate with processes on other nodes. The specialized modules netcatConn (send data) and netcatToList (listen for data) employ the Linux netcat utility to realize these functionalities. The decision whether a netcat connection is needed or not is taken by the Daemon in the background, without the need of a user intervention.

One central issue of WASNs remains often unconsidered in hardware design: Time synchronization [29]. Due to the fact of a missing centralized clock signal all devices of a WASN sample data with different rates and start offsets. If a hardware integrated solution (see [30] for example) is not possible data streams have to be resampled in software for synchronization [31] or algorithm selection becomes restricted to Sampling Rate Offset (SRO) independent ones.

In the following subsections we present two compact sensor nodes on Raspberry Pi 3 Model B+ basis. One uses an existing soundcard interface and the other is a completely new development targeting the SRO issue.

### 3.1 Raspberry Pi – Audio Frontend

The Raspberry Pi 3 Model B+ platform offers computational power (ARM A53 Core, 1.4 GHz) at a compact form factor and reasonable costs. Its widespread application in smart home environments has forced the development of multi-channel soundcards, e.g., the Octo soundcard from Audio Injector [32].

We developed a pluggable analog frontend (see Fig. 5, blue Printed Circuit Board (PCB)) hosting a circular microphone array with 6 synchronously sampled audio channels. The Octo soundcard is accessible via the ALSA interface and the required drivers are already integrated in many Linux distributions.

The Raspberry Pi 3 Model B+ (and also the Pi 2 and Pi 3) employs a BCM2837 chip for offering a 4 SP interface. This interface is limited on chip-side to stereo audio [33]. To offer more than two channels the developers of [32] utilizes a Field Programmable Gate Array (FPGA). The 8 channel Time-division Multiplexing (TDM) audio signal from the Analog-to-Digital Converter (ADC) is rearranged by the FPGA to a pseudo dual channel audio signal at four times higher sampling rate. After receiving the data in the Kernel space of the Raspberry Pi the signal is converted back to an 8 channel audio signal.

### 3.2 Raspberry Pi – Quad node

The analog frontend from Sec. 3.1 is a 6 channel soundcard with a convenient audio quality. However, the SRO problem remains unsolved on this platform, as the crystal oscillator is neither configurable at ppm precision nor is the sampling process observable in detail.

Hence, we designed a simple pluggable card, called the Quad node (see Fig. 6), with a configurable Any Frequency Oscillators (AFO) (Silicon Labs SiS14) driving the sampling process. Its frequency resolution is configurable with a precision of 0.026 ppb. The ADC chip from Texas Instruments (ADS 1274) samples 4 channel audio signal at four times higher sampling rate. After receiving the data in the Kernel space of the Raspberry Pi the signal is converted back to an 8 channel audio signal.
channels synchronously at 24 Bit and up to 144 kHz. On board we placed four omni directional condenser microphones at 4 cm distance. Audio data is exchanged by a Serial Peripheral Interface (SPI) interface, while the Si514 is controlled via the I₂C interface.

Figure 6: Raspberry Pi with mounted quad node.

3.3 Open Framework & Hardware

All schematics, PCB layouts and Gerber files for the Raspberry Pi based sensor nodes are available from our website [34]. Feel free to modify, enhance or change our hardware to your custom needs.

The MARVELO framework and an installation guide on how to setup a WASN consisting of Raspberry Pi³ Model B+ is available on our project websites [22].

4 Experiments

We evaluate our framework on a wireless 4 nodes (A, B, C, D) Raspberry Pi network using a Python implementation of the sampling rate offset estimator [15], which is depicted in Fig. 7. The algorithm’s implementation does not reach real time processing performance on the Raspberry Pi platform, but it is a compact example to experimentally study MARVELO.

Figure 7: Block diagram example from [15].

It is assumed that only nodes A and B are close to the desired source, hence, they are selected for recording in all scenarios. Furthermore, node D requested the SRO information for other tasks and has to be incorporated. Since the wireless signal between D and the recording nodes is weak (i.e., low wireless links capacities), it utilizes node C for forwarding high data rates. In all tasks, audio recordings of 25 s are processed.

We place the processing blocks as described by Tab. 1. Exemplarily, the third scenario is depicted in Fig. 8. The first three scenarios utilize the same processing blocks in different placements so that the blocks are connected via pipes, while the fourth runs a single bulk script on node D. Hence, Centralized-4 has to process the tasks sequentially while the others can process them in parallel.

In Fig. 9, we show the network and the end to end delays for each scenario. The former is defined as the time required to receive the first input at D from the senders, while the latter is the time required to process the whole recording.

![Network and end to end delays](image)

5 Conclusions & Outlook

We observe that the centralized scenarios (Centralized-1 and Centralized-4) are subject to higher network delay, compared to distributed scenarios (Distributed-2 and Distributed-3), as they need to forward the raw data in the network. Although both centralized scenarios execute the processing tasks on only one node, Centralized-1 has a higher processing delay compared to Centralized-4. This is due to the higher interprocess communication overhead in scenario Centralized-1 which, in our small example, outweighs the possible advantages from executing these blocks in parallel.

Regarding our distributed scenarios, they have a similar network delay, which is in all cases lower than that of the centralized scenarios. This is due to sending only the processed data (e.g., 1 Byte by block 8) compared to raw audio chunks (1 kB). On the other hand, they have different end to end delays. Scenario-2 shows a higher end to end delay (210 s), which is almost equal to that of Centralized-4 (200 s). However, Distributed-3 shows the best distribution in terms of network and end to end delays.

![End to end and network delays](image)

Table 1: Comparison of scenarios for distributed and centralized realizations.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Node A</th>
<th>Node B</th>
<th>Node C</th>
<th>Node D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized-1</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>3,4,5,6,7,8,9</td>
</tr>
<tr>
<td>Distributed-2</td>
<td>1,6,8</td>
<td>2,7</td>
<td>3,4,5</td>
<td>9</td>
</tr>
<tr>
<td>Distributed-3</td>
<td>1,5,4</td>
<td>2,3</td>
<td>6,7,8</td>
<td>9</td>
</tr>
<tr>
<td>Centralized-4</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>[15]</td>
</tr>
</tbody>
</table>

Acknowledgment

This work was supported by Deutsche Forschungsgemeinschaft (DFG) under contract no. KA2325/4-1 and SCHM 3301/1-1 within the framework of the Research Unit FOR2457 “Acoustic Sensor Networks”.

Figure 8: Visualization of scenario “Distributed-3”.

Figure 9: End to end (blue) and network (red) delays.
References


