BARISTA: A FRAMEWORK FOR CONCURRENT SPEECH PROCESSING BY USC-SAIL

Doğan Can, James Gibson, Colin Vaz, Panayiotis G. Georgiou, Shrikanth S. Narayanan

Signal Analysis and Interpretation Lab, University of Southern California, CA 90089

dogancan@usc.edu, jjgibson@usc.edu, cvaz@usc.edu, georgiou@sipi.usc.edu, shri@sipi.usc.edu

ABSTRACT

We present Barista, an open-source framework for concurrent speech processing based on the Kaldi speech recognition toolkit and the libcppa actor library. With Barista, we aim to provide an easy-to-use, extensible framework for constructing highly customizable concurrent (and/or distributed) networks for a variety of speech processing tasks. Each Barista network specifies a flow of data between simple actors, concurrent entities communicating by message passing, modeled after Kaldi tools. Leveraging the fast and reliable concurrency and distribution mechanisms provided by libcppa, Barista lets demanding speech processing tasks, such as real-time speech recognizers and complex training workflows, to be scheduled and executed on parallel (and/or distributed) hardware. Barista is released under the Apache License v2.0.

Index Terms— open source, C++, actor model, concurrency and distribution, real-time speech recognition

1. INTRODUCTION

Modern speech research relies on parallel and distributed hardware for fast and efficient computation, and developing speech processing workflows that can leverage available computational resources is a priority. Barista[1] is a step in this direction. It is an open source framework for concurrent speech processing written in C++ and licensed under the Apache License v2.0. The aim is to provide an easy-to-use, extensible concurrency (and distribution) framework for speech research and applications. An early release of Barista, including example setups, is available on GitHub at the address `http://github.com/usc-sail/barista`. Barista targets both multi-core/processor machines and networked clusters. With Barista, it is possible to schedule and execute different stages of complex speech processing workflows on different threads/cores/processors of a single host machine or on different nodes of a networked cluster.

Barista development started from a need to develop independent software modules that can be painlessly plugged into a larger concurrent speech processing workflow. We wanted these workflows to be fully specified by simple configuration files that define both the parameters for each module and the input/output relationships between them. We had several goals for our system including:

- supporting real-time and online workflows
- easing collaborative development by allowing independent module creation
- utilizing modern hardware, including multi-threaded and distributed systems
- allowing for run time changes in network topology
- remaining compatible with the Kaldi toolkit [1]
- minimizing delay and computational overhead

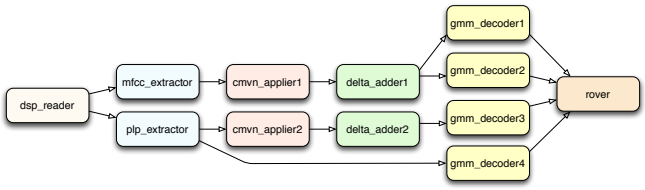

Fig. 1: Example ASR workflow. Modules are developed independently and can be connected as desired by the user for the task. The system is real-time, online, and can be executed on multiple local or distributed cores.

Figure 1 shows an example ASR workflow. Barista aims to ease deployment of such complex workflows through simple network definitions.

Barista is based on the well-known actor model [2] for concurrent and distributed programming. In this model, actors are independent, self-contained modules open to communication with other components through asynchronous messages [3, 4]. Each actor can send/receive messages to/from other actors, create new actors, or destroy existing actors. There is no implicit state sharing between actors and all sharing is done through an explicit message passing mechanism. Since actors do not share states or mutable resources, race conditions are avoided by design in the actor model, unlike other models of concurrent computation that rely on shared state and some form of locking (e.g. mutexes, semaphores, etc.) for coordination. Furthermore, since message passing can readily support network transparency, the actor model applies both to concurrency in the case of parallel threads/processes running on a single multi core/processor

This work was funded by NSF, ONR and DARPA.

[1] Playing on the coffee meme of the Kaldi speech recognition toolkit, we call our contribution "Barista" as it enables the users to mix the ingredients of their ASR systems in a flexible and customizable way.

978-1-4799-2893-4/14/$31.00 ©2014 IEEE

machine and to distribution when actors run on different nodes connected through the network. Most signal processing tasks can be seen as successive independent operations applied on some input data, and this compositional structure is a great match to the actor model at an abstract level, where actors correspond to operations and the flow of data is achieved through message passing. Barista inherits its concurrent and distributed abilities from libcppa, a modern, standards compliant implementation of the actor model in pure C++ [5].

Barista relies on Kaldi [1], an open source speech recognition toolkit, for most of the speech processing functionality. Kaldi's highly modular design is an excellent fit for the actor model. Most Barista actors are straightforward adaptations of Kaldi tools, often with identical or very similar functionality. It is in fact possible to emulate many Kaldi tools using their Barista counterparts. One of the design goals of Barista is to support non-blocking concurrent processing of input data. To that end, Barista provides online alternatives for various Kaldi operations, such as feature extraction/transformation that can handle input in an incremental fashion without changing the output. Such exact online implementations are not possible for all Kaldi operations though. For operations which require non-causal processing, such as speaker/utterance normalization, Barista provides approximate online implementations.

A motivating use case for Barista is the real time speech recognition problem, which consists of audio acquisition, several stages of feature extraction, normalization, and transformation, followed by an online decoder. In Barista, each stage of the speech recognition pipeline is implemented as a non-blocking online actor and input data flows through this simple linear network, going through various transformations until it produces a sequence of hypothesis words. Since there are no blocking calls on the network, this network, when scheduled concurrently on a capable multi-core machine, starts to produce partial hypotheses as soon as audio acquisition starts. On top of that, it calculates the final hypothesis with minimal delay because it would have completed most of the decoding by the time it processes the final audio samples. What is more exciting is the possibility of distributing the processing work between a low resource client and a server machine without any change in implementation, leveraging the network transparency provided by libcppa. For instance, a smartphone can run early stages of the recognition pipeline, such as audio acquisition and feature extraction, while a remote server does the processor-intensive decoding [6].

2. SYSTEM DESIGN

At its core, Barista is a collection of libcppa actors adapted from Kaldi tools, along with the utilities to construct and run a network of these actors on parallel and distributed hardware. The user specifies the network topology and the parameters for each actor, and Barista implements the network at runtime. Maybe more importantly, it is easy to implement and add new actors to the Barista framework. Our primary design goal was to develop a simple, reliable, and extensible framework. In this section we will outline aspects of libcppa and Kaldi that are pertinent to Barista and explain the design and implementation choices we made along the way and mention the limitations of the current implementation.

2.1. Barista Modules

In Barista, modules are in a publisher/subscriber relationship, i.e. subscribers listen to the messages of publishers, which is maintained by way of a subscribers list for each module. Each Barista module (or actor) is a C++ class which derives from a common base class `ModuleBase`. `ModuleBase` is a bare bones libcppa actor whose only functionality is to keep a pointer to a subscriber list and provide a configuration mechanism for setting parameters and updating the subscriber list. Each module can publish and receive a variety of messages of possibly different types. Typically, each module processes a particular type of data message but also listens to various other synchronization, configuration, and execution messages flowing through the network. For instance, a feature extraction module might primarily listen for vectors of integers representing audio samples while at the same time handle predefined textual messages signaling utterance boundaries or end of operation. In essence, each Barista module can be thought of as a set of message handlers, one for each type of message it listens to. Continuing with the feature extraction example, one handler might define what to do with a new batch of audio samples while another one defines how to finalize feature extraction for the current utterance when an utterance boundary message is received.

2.2. Communication and Message Processing

Actors communicate with each other using mailbox-based messaging, which was introduced in the original actor model described in [3]. In simple terms, a mailbox is exactly like its physical equivalent. It is owned by an actor and it holds the messages addressed to this actor. More precisely, a mailbox is a FIFO-ordered message buffer. Any actor can enqueue (send) a new message but only the owner can dequeue (read) one. The mailbox is the sole means of communication between actors. In libcppa, each actor processes incoming messages by iterating over its mailbox. Every time the actor decides to process the messages in its mailbox, it starts with the first one but is free to skip messages. The actor dequeues the first message it can handle and performs any operations specified by the message. A message remains in the mailbox until it is processed. Other actor systems based on this type of message processing include Erlang [7] and Scala Actors library [8].

One design choice we made early in the development of Barista was to simplify the management of asynchronous messages flowing through the network by limiting inter-module communication to subscription lists. This way, Barista modules do not need to worry about where to send

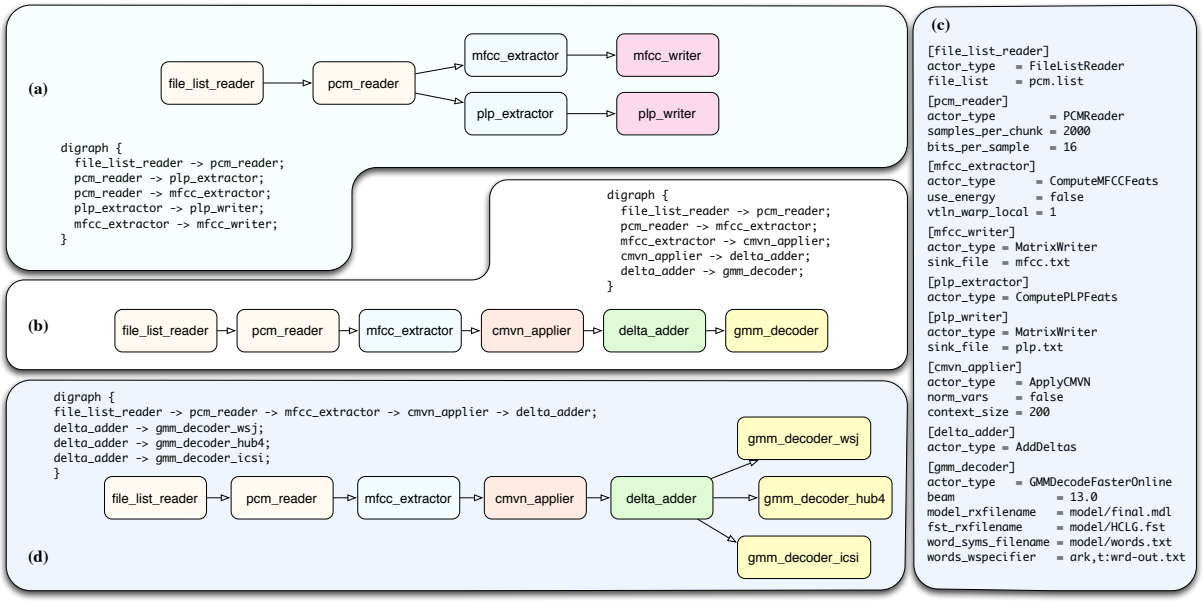

Fig. 2: Example Barista networks. (a) The topography of a feature extraction network and the corresponding network specification file. (b) The topography of a decoding network and the corresponding network specification file. (c) The configuration file shared between networks (a) and (b). (d) The network used in the second case study (see section 3.2).

their messages. All messages are distributed through subscriber lists and all subscribers receive all published messages. Unfortunately, sending separate messages to different subscribers is not directly supported by the Barista framework because of the subscriber lists scheme. Nonetheless, this is not a shortcoming of libcppa and such functionality can be implemented on a per actor basis if needed.

The message passing implementation of libcppa uses tuples with the call-by-value semantic following the "no shared state" mantra that is at the core of the actor model. This implementation keeps the programming model clean and easy to understand, and simplifies development since there is no need to track the lifetime of messages [5]. However, it is common to send the same message to multiple actors, which requires multiple copies of the message with the call-by-value semantic. To avoid unnecessary copying overhead and race conditions, libcppa uses a copy-on-write tuple implementation where a tuple is shared among any number of actors as long as none of the actors tries to overwrite it. If an actor tries to overwrite a tuple, a copy is automatically provided. With this implementation, race conditions are avoided by design and tuples are copied only if necessary.

Since libcppa provides a fully network transparent messaging system, all messages have to be serialized and deserialized to standardize messaging over various network protocols. While libcppa can handle most message types without any extra effort, user defined data types in messages have to be explicitly announced to the system by providing implementations for serialization and deserialization of these types. Barista provides such implementations for Kaldi's custom `Vector` and `Matrix` types.

2.3. Network Specification

Barista network topologies are specified as directed acyclic graphs in GraphViz DOT format [9]. Each node of the network represents a module and directed edges represent the flow of data between modules. In addition to the network specification, Barista requires a configuration file which specifies the module type and parameters for each node in the network. It is okay to have extra nodes, which are not part of the network, listed in this configuration file. However, each node in the network needs to have a corresponding entry in the configuration file. This setup facilitates experimenting with different network topologies; a single configuration file can be shared by multiple Barista setups, with each setup picking only the nodes specified by its network. Figure 2 gives example network specifications and the accompanying configuration file.

2.4. Network Construction

At run-time, Barista spawns each actor in the network specification file and sets each actor's parameters defined in the configuration file. It also sets up the subscription list for each actor defined by the directed edges in the network specification file. Another early design choice was to make network construction a run-time operation to avoid rebuilding the executable every time the network layout changes. Run-time network construction significantly simplifies experimentation and system building since changing the network topology is as simple as editing the network specification file. Another

advantage of this design choice is the possibility of making changes to the network at run-time. We expect this functionality to be an asset both in interactive workflows and in distributed workflows that can scale dynamically with the available hardware resources that we plan to support in the future. The downside of this decision is the added complexity stemming from the need to schedule, execute, and control a dynamic network of actors as opposed to a static one defined at the compilation time.

2.5. Concurrency and Distribution

libcppa supports a variety of actor scheduling strategies. The default is a user-space cooperative scheduling strategy, i.e. executing actors in a thread pool, where each actor's context switches back to the scheduler whenever it tries to process a new message. With this strategy, an actor is not allowed to block while its mailbox is empty and starve other actors by occupying valuable system resources. Instead, the actor is rescheduled again after a new message arrives in its mailbox. Cooperative scheduling is the recommended strategy for most actors, which do not call blocking functions. Blocking actors, e.g. actors responsible for blocking I/O operations, can be executed on their own threads using libcppa to avoid the possibility of starving other actors scheduled cooperatively in a thread pool. Most Barista modules are non-blocking and can be scheduled cooperatively. Resource-hungry modules, such as decoders, or modules that make blocking function calls, such as I/O or network modules, are scheduled on their own threads.

2.6. Fault Propagation

libcppa adopts Erlang's well-established error propagation model [7] based on monitoring, which has been proven to be very effective and reliable in practice [10]. Whenever an actor fails, an exit message is sent to all actors that monitor it, and these messages propagate in the network unless they are trapped and handled. Based on this model, it is possible to build fault-tolerant distributed systems, where failing actors are re-created by dedicated actors monitoring them. The current Barista implementation uses fault propagation to make sure that all modules are either alive or have collectively failed.

3. EVALUATION

We conducted two case studies with the Barista system. These case studies were designed to evaluate the computational efficiency and accuracy of Barista and demonstrate the simplicity of configuring Barista for performing complex tasks. In the first case study, we set up a Barista network and a standard Kaldi pipeline and compared the runtimes and word error rates (WER) of the two systems. In the second case study, we investigated the amount of labor needed to configure Barista to run multiple decoders in parallel. The setups for these case studies are on our GitHub repository under the `egs` directory.

3.1. Case Study I

We evaluated the nominal computational overhead of using Barista compared to a standard Kaldi pipeline by measuring the runtimes for a simple decoding task. Both systems read the audio from file, extracted MFCCs, applied cepstral mean normalization, added delta and delta-delta features, and performed decoding. Figure 2(b) illustrates these processing steps. The models were trained using the Wall Street Journal (WSJ) corpus [11]. Both systems use the CIRS-I test set from WSJ for testing. The difference in runtime[2] was less than 1 second on average (5:20.75 for Kaldi vs. 5:20.09 for Barista) and there was no difference in WER (14.46% for both systems). Thus, Barista did not introduce latency or errors to a standard ASR pipeline. These results are significant because they show that a user can take advantage of the flexibility offered by Barista without sacrificing speed or accuracy.

3.2. Case Study II

The second case study demonstrated the ease of building complex systems with Barista. In this case study, three decoders, each using a different decoding model, shared a single feature extraction stream. The three models were trained on the WSJ corpus, the ICSI meeting corpus [12], and the HUB4 broadcast news corpus [13]. Building this multiple decoder system required only two simple changes to the decoding network from Case Study I. The first change required the feature extraction stream, defined in the network specification file, to be directed to three decoders instead of one. The second change needed the three decoders to be declared in the actors configuration file, with each decoder actor using one of the three models. The network configuration is shown in Figure 2(d). Because the decoders shared the same feature extraction module, Barista performed concurrent processing without introducing the computational overhead of multiple feature extraction streams. The computational savings offered by Barista can help streamline ASR processing and facilitate research in ASR, such as fusing decisions of multiple ASR decoding outputs [14, 15].

4. CONCLUSION

We described Barista, an open source framework for concurrent and distributed speech processing based on Kaldi and libcppa. Barista currently supports concurrent scheduling and execution of independent actors adapted from Kaldi tools on a single host as well as distributed processing via independently executed remote actors. Barista is under active development. The current release should be taken as an early preview of a subset of the functionality described here. In addition, our group is actively working on implementing new Kaldi tools, such as robust feature extraction modules [16, 17] and a module for fusion of diverse experts [14, 15], that will be integrated into the Barista framework. We are also looking into distributed training using the Barista architecture.

[2]Both setups were run on a MacBook Pro with a 2.3 GHz Intel Core i7 processor.

5. REFERENCES

[1] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer, and Karel Vesely, "The Kaldi Speech Recognition Toolkit," in *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. Dec. 2011, IEEE Signal Processing Society, IEEE Catalog No.: CFP11SRW-USB.

[2] Carl Hewitt, Peter Bishop, and Richard Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *Proceedings of the 3rd international joint conference on Artificial intelligence*, San Francisco, CA, USA, 1973, IJCAI'73, pp. 235–245, Morgan Kaufmann Publishers Inc.

[3] Gul Agha, *Actors: a model of concurrent computation in distributed systems*, MIT Press, Cambridge, MA, USA, 1986.

[4] Gul Agha, Ian A. Mason, Scott Smith, and Carolyn Talcott, "Towards a Theory of Actor Computation," in *Proceedings of CONCUR '92, vol 630 of LNCS*. 1992, pp. 565–579, Springer.

[5] Dominik Charousset and Thomas C. Schmidt, "libcppa - Designing an Actor Semantic for C++11," in *Proc. of C++Now*, 2013.

[6] Naveen Srinivasamurthy, Antonio Ortega, and Shrikanth Narayanan, "Efficient scalable encoding for distributed speech recognition," *Speech Communication*, vol. 48, no. 8, pp. 888–902, 2006.

[7] J. Armstrong, *Making reliable distributed systems in the presence of software errors*, Ph.D. thesis, KTH, Sweden, 2003.

[8] Philipp Haller and Martin Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theor. Comput. Sci.*, vol. 410, no. 2-3, pp. 202–220, Feb. 2009.

[9] Emden R. Gansner and Stephen C. North, "An open graph visualization system and its applications to software engineering," *Software - Practice and Experience*, vol. 30, no. 11, pp. 1203–1233, 2000.

[10] J. H. Nyström, P. W. Trinder, and D. J. King, "Evaluating distributed functional languages for telecommunications software," in *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, New York, NY, USA, 2003, ERLANG '03, pp. 1–7, ACM.

[11] Douglas B Paul and Janet M Baker, "The design for the wall street journal-based csr corpus," in *Proceedings of the workshop on Speech and Natural Language*. Association for Computational Linguistics, 1992, pp. 357–362.

[12] Adam Janin, Don Baron, Jane Edwards, Dan Ellis, David Gelbart, Nelson Morgan, Barbara Peskin, Thilo Pfau, Elizabeth Shriberg, Andreas Stolcke, et al., "The icsi meeting corpus," in *Proc. of ICASSP*, 2003.

[13] David Graff, Z Wu, R MacIntyre, and M Liberman, "The 1996 broadcast news speech and language-model corpus," in *Proceedings of the DARPA Workshop on Spoken Language technology*, 1997.

[14] Kartik Audhkhasi, Andreas Zavou, Panayiotis G. Georgiou, and Shrikanth S. Narayanan, "Empirical link between hypothesis diversity and fusion performance in an ensemble of automatic speech recognition systems," in *Proc. of InterSpeech*, 2013.

[15] K. Audhkhasi, A. M. Zavou, P. G. Georgiou, and S. S. Narayanan, "Theoretical analysis of diversity in an ensemble of automatic speech recognition systems," *Audio, Speech, and Language Processing, IEEE/ACM Transactions on*, vol. 22, no. 3, pp. 711–726, March 2014.

[16] Maarten Van Segbroeck and Shrikanth S. Narayanan, "A robust frontend for ASR: combining denoising, noise masking and feature normalization," in *Proc. of ICASSP*, 2013.

[17] James Gibson, Maarten Van Segbroeck, Antonio Ortega, Panayiotis G. Georgiou, and Shrikanth S. Narayanan, "Spectro-Temporal Directional Derivative Features for Automatic Speech Recognition," in *Proc. of InterSpeech*, 2013.