Tracter: A Lightweight Dataflow Framework

Philip N. Garner, John Dines

Idiap Research Institute, Martigny, Switzerland

pgarner@idiap.ch, dines@idiap.ch

Abstract

Tracter is introduced as a dataflow framework particularly useful for speech recognition. It is designed to work on-line in real-time as well as off-line, and is the feature extraction means for the Juicer transducer based decoder. This paper places Tracter in context amongst the dataflow literature and other commercial and open source packages. Some design aspects and capabilities are discussed. Finally, a fairly large processing graph incorporating voice activity detection and feature extraction is presented as an example of Tracter's capabilites.

Index Terms: Dataflow, speech recognition, open source.

1. Introduction

Tracter is a lightweight software framework for doing signal processing. It grew out of a requirement at Idiap, and within the AMI consortium [1], for a real-time capability. Whilst it remains fundamentally a real-time system, it also works perfectly well offline, and has been used to solve a number of other problems associated with collaborative work in signal processing and automatic speech recognition (ASR).

Signal and image processing usually involve chains of processing steps. The results of one processing step are simply passed to one or more subsequent processing steps until the required form of result is obtained. For example, analysis of speech might involve sampling, sample rate conversion, filtering and spectral analysis before being passed to a coder or speech recogniser. Analogously, visual analysis of faces might involve video capture followed by illumination normalisation and feature extraction before being passed to a face recogniser.

Working on an individual component or a small chain of components is easy. Typically a student might implement them as subroutines and simply call them in turn with file input and output. Embedding such components into a larger system, however, can be difficult. This is especially true when the system is required to run online. If it is then required that other standard libraries are used, and that other developers, perhaps at other institutions, contribute, then the system can become cumbersome.

Dataflow is a programming paradigm that can solve these and other problems. In dataflow, individual components known as actors are linked together as vertices in a directed graph. Each actor processes data received at an input and produces data at an output. The arcs of the graph serve as buffers, routing data to other actors. Dataflow is exemplified by several commercial packages, notably Cantata (part of Khoros[1]) and Simulink[2]. In the academic world, there is the Ptolemy project [2], and the CLAM system [3]. All these tools build on the visual nature of the directed graph.

[1] http://www.khoral.com/

[2] http://www.mathworks.com/products/simulink/

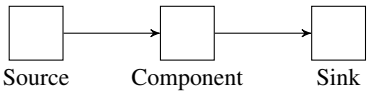

Figure 1: A minimal Tracter graph.

Dataflow has been used in ASR before; the concept goes back to at least 1985 [4], and is clearly used in the hardware implementation of Gómez *et al.* [5]. More recently, in SPHINX-4 Lamere *et al.* [6] define a chain of processing stages connected by queues. This in turn inspired a similar implementation by Dixon *et al.* [7], although those designs apparently allow only chains rather than graphs. The ATK wrapper for HTK [8] is also dataflow influenced.

In its most tangible form, Tracter defines a library of components that can be linked together into a directed graph in the spirit of the dataflow paradigms described above. Data enters one or more sources, is propagated through the graph, and processed data is made available at a sink. The graph nature of Tracter has allowed easy implementation of at least two otherwise quite difficult to implement aspects of ASR processing in the context of the Juicer ASR decoder [9] and the AMIDA system [10]: Tandem features and voice activity detection (VAD).

In the remaining sections, some design and implementation aspects of Tracter are discussed, and illustrated by example. Tracter is available as an open-source library under a permissive (BSD style) licence.

2. Architecture

2.1. Overview

Tracter is a data-flow framework for signal processing in the sense of, for instance, Lee and Messerschmitt [11]. It defines a library of components that each typically do a small amount of computation, but can become nodes in a directed graph where they work together (although independently) to do something more useful. Although Tracter contains several implementations of basic algorithms, it has developed into a wrapper for libraries of basic algorithms.

The components generally run serially, not in parallel; Tracter does not do concurrent dataflow in the sense of Kahn process networks [12]. It is not a language either. However, the components do have some things in common with process networks. For instance, reads are blocking but writes are not.

A minimal Tracter graph is shown in figure 1. Tracter distinguishes sources and sinks from other components; it allows an arbitrary graph of actors to be assembled, with an arbitrary number of sources and sinks. It is mostly synchronous [11] in the sense that a given actor firing normally consumes and produces predictable amounts of data. However, this is not true for

the VAD gate (see section 4.2). The major difference in comparison with Lee and Messerschmitt is that the actors do not run concurrently. They are scheduled sequentially using the pull formalism described below, and by Manolescu [13]. The same formalism has been used in the design of SPHINX-4 by Lamere *et al.* [6], and by Dixon *et al.* [7]. Manolescu [13] notes that much the same programming pattern is used by UNIX streams; that in turn is the method used by the SPTK[3] toolkit to similar ends.

2.2. Marshalling

Marshalling is taken here to be the means by which data is guided (marshalled) through the graph. In the original dataflow literature, summarised by Lee and Parks [12], the actors in the network were all separate processes. Calculations were triggered by data arriving at actor inputs. This marshalling method can be referred to as *push*, in that processing is driven from the source (input) of the graph. Push processing lends itself to calculations that are certain to run in real-time, and is exemplified by analogue to digital converters (ADCs), where a clock drives the ADC, and subsequently interrupts the host CPU.

The opposite, *pull*, method is driven from the sink (output) of the graph. A request for data is sent from the sink and propagates through the graph, returning when data is available. Pull lends itself to processing that does not necessarily run in real time, since actors are not asked for more data until they have fulfilled a given request.

In writing a (real-time) ASR system, the focus tends to be on the decoder. Certainly, the decoder is the most CPU intensive component. By contrast, the front-end or feature-extraction components are comparatively simple. Indeed, from the decoder author's point of view, the features tend to exist precalculated in a file. This naturally leads to a request-driven *pull* decoder architecture of the general form:

```
feature* f;
while(f = frontend.Read())
    decode(f);
```

In this form, the `Read()` method amounts to reading the next line of a file.

Tracter is, in its simplest form, an interface between the decoder's pull architecture and the ADC's push architecture. More generally, Tracter allows a directed graph of components that are all pull driven[4]. In an ASR system, this allows the design to be decoder-centric:

- The decoder is a Tracter *sink* that calls `Read()` on some input component.
- That input component (which may simply be a *source*) calls its inputs, which in turn call their inputs.
- Eventually, one or more sources are called, which read the raw data.

In the ADC case, it is the sources that interface the push and pull mechanisms using an appropriate buffer. If, as is the case in AMIDA, the source is a TCP socket, it simply calls the BSD socket API function `recv()`, which is itself a pull-driven method.

The *pull* marshalling has one other advantage: processing is only done on data for which the processing result is required.

[3] http://sp-tk.sourceforge.net/

[4] This is the origin of the name. The misspelling is deliberate, in the spirit of Juicer

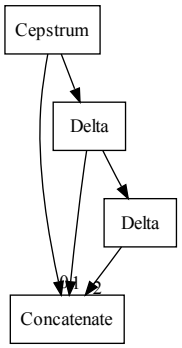

Figure 2: Tracter sub-graph to append dynamic features.

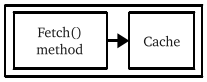

Figure 3: A cached component.

This is particularly useful in the context of VAD, where certain features and certainly decoding only need to be done on speech frames. Reciprocally, noise processing can be restricted to purely noise frames.

2.3. Capabilities

Figure 2 shows the sub-graph that is used to append dynamic features to cepstra for ASR. Any order of dynamic features can be added. Notice that the first delta component will need to read a few frames behind and ahead of the concatenation components in order to calculate a derivative. On the subsequent frame, all but one of the previously calculated frames will be required again. This leads naturally to a caching requirement. In dataflow, the cache is implemented as a buffer on each arc. In Tracter, this concept is simplified into the concept of a cached component. This is illustrated in figure 3. When data is requested from the component, the cache is first checked for those data. If any are not present then the `Fetch()` method is called. The signal processing functionalty is therefore implemented as a `Fetch()` method.

2.4. Cache size

In the original dataflow literature, the buffers on the graph arcs were able to grow indefinitely. This presents a run-time overhead associated with memory allocation. ADCs, on the other hand, use fixed size circular buffers. In Tracter, the caches are there to buffer input data for the downstream component(s). Tracter caches are allocated at construction time, and do not change thereafter. This is achieved by a message passing algorithm that functions as follows:

1. The sink tells each of its input components the size of the maximum request it will make for data, plus an offset in time (to facilitate reading ahead or behind).
2. Each input component keeps a record of the maximum read ahead and read behind over all components reading from it.
3. Based on these maxima, the component adjusts its cache size.
4. Based on these maxima, the component iterates over its input components as in step 1.

The algorithm is not explicitly multi-pass, but upstream subgraphs may be iterated over several times in response to messages from components with multiple downstream connections. The algorithm is sufficient but not optimal; the final cache sizes are never too small, but can be larger than necessary.

3. Programming

3.1. API

Tracter is written in C++. It has a hierarchical API that distinguishes the following layers:

Factories define a graph of components that constitute some useful block. A factory is actually a very thin layer; it just instantiates a graph of components, so the overhead after calling the factory is zero.

Components are the main level. A component is a vertex in a directed graph of processing elements. Components necessarily have inputs and outputs. Components implement dataflow.

Objects in Tracter are things that have a name and can hence receive parameters. They don't necessarily take part in dataflow operations. Factories are objects, as are components.

The most important level in Tracter, and the only one discussed in this paper, is the component level.

Various standard components are implemented in Tracter. These include sources for reading from files, sockets and various sound APIs. Several standard graphs are also available as factories for computation of common ASR features such as MFCCs and PLPs.

3.2. Third party libraries

Although some functionality is implemented natively in Tracter, it is really a framework, not a function library. For instance, it contains FFT components, but not an FFT implementation. This is left to function libraries. Similarly, the nature of a collaborative project is that many modules are written by many different people in different institutions. Tracter components present a fairly simple internal interface that has allowed wrapping of several other component packages. Notably,

FFT libraries Kissfft[5] and FFTW[6] provide portable and reasonable quality Fourier transform packages.

Resampling libraries libresample[7] and SRC[8] provide resampling.

Torch [14] is a machine learning package developed at Idiap. It is used to implement a multi-layer perception (MLP) that is used in the AMIDA segmenter.

BSAPI is a speech API developed at Brno University of Technology. It is the prefered development medium for the Brno speech group, and implements some of the more advanced features used in the AMIDA system.

HTK [15] is a commonly used toolkit in ASR, and is used in AMIDA to extract features and for adaptation. Tracter provides wrappers for HTK modules HCopy and HParm.

SPTK is a library very much in the spirit of Tracter. SPTK processing steps use UNIX pipes to communicate; these processing steps can easily be re-wrapped as Tracter components. SPTK notably contains the reference mel-generalised cepstrum implementation.

4. Example

Figure 4 shows a fairly complicated Tracter graph to calculate features, with VAD similar to that in [10]. The main feature extraction steps are implemented using BSAPI wrapper components. The two main aspects are discussed below.

4.1. Library wrapping

The graph illustrates how different libraries can be combined into the same chain:

1. The `MLP` component is a wrapper for the Torch3 MLP implementation.
2. The `BSAPIFrontEnd` and `BSAPITransform` components are wrappers for the BSAPI library.

There is some element of duplication of functionality. For instance, BSAPI also implements MLPs and Tracter has PLP features. However, the versatility is essential for collaboration between groups with different software bases.

4.2. Voice Activity Detection

One of the main features of a real-time ASR system is voice activity detection (VAD). VAD is implemented in Tracter using a gate method. A `VADGate` component distinguishes a downstream subgraph containing the decoder from an upstream graph connected to the actual media. Requests from downstream for indexed data are translated to requests upstream with modified indexes. The indexes are changed by means of a second input to the VAD gate that indicates speech activity.

This design is not necessarily the most efficient possibility because the VAD logic must confirm that speech has begun, typically by waiting for some minimum time, before the VAD gate will let the appropriate frames downstream. However, the design is otherwise very flexible and allows the decoder to be developed completely offline and independently.

5. Conclusions

Whilst none of the elements of Tracter are especially novel, the authors believe that the combination of a dataflow architecture, the ability to wrap other packages, and the permissive licence is unique and useful. Tracter has allowed construction of non-trivial ASR based systems across multiple laboratories, and continues to be used for this purpose. It is freely available for download as an open source BSD licenced package[9]. Some GPL licenced parts are packaged separately[10] to avoid licence incompatibility.

6. Acknowlegements

Besides the authors, significant contributions have been made to Tracter by Mike Flynn, Martin Karafiát, Danil Korchagin and Vinny Wan. We are also grateful to Olivier Bornet and Alexandre Nanchen for help with packaging and porting.

[5] http://kissfft.sourceforge.net/

[6] http://www.fftw.org/

[7] http://www-ccrma.stanford.edu/~jos/resample/Free_Resampling_Software.html

[8] http://www.mega-nerd.com/SRC/

[9] http://juicer.amiproject.org/tracter/

[10] http://juicer.amiproject.org/tracter-gpl/

This work was supported by the European Union 6^{th} and 7^{th} Framework Programme IST Integrating Projects "Augmented Multi-party Interaction with Distance Access" (AMIDA, FP6-033812) and "Together Anywhere Together Anytime" (TA2, FP7-214793), and the Swiss National Center of Competence in Research (NCCR) on "Interactive Multi-modal Information Management" (IM2).

7. References

[1] T. Hain, L. Burget, J. Dines, P. N. Garner, A. El Hannani, M. Huijbregts, M. Karafiat, M. Lincoln, and V. Wan, "The AMIDA 2009 meeting transcription system," in *Proceedings of Interspeech*, Makuhari, Japan, September 2010.

[2] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity—the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, January 2003.

[3] X. Amatriain, "CLAM, a framework for audio and music application development," *IEEE Software*, pp. 82–85, January/February 2007.

[4] T. S. Anantharanan and R. Bisiani, "Custom data-flow machines for speech recognition," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, April 1985, pp. 1847–1850.

[5] P. Gómez, A. Álvarez, R. Martínez, M. Pérez-Castellanos, V. Rodellar, and V. Nieto, "A DSP-based modular architecture for noise cancellation and speech recognition," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, vol. V, Monterey, CA, May 1998, pp. 178–181.

[6] P. Lamere, P. Kwok, W. Walker, E. Gouvêa, R. Singh, B. Raj, and P. Wolf, "Design of the CMU SPHINX-4 decoder," in *Proceedings of EUROSPEECH*, 2003.

[7] P. R. Dixon, D. A. Caseiro, T. Oonishi, and S. Furui, "The TITech large vocabulary WFST speech recognition system," in *Proceedings of the IEEE Workshop on Automatic Speech Recognition and Understanding*. Kyoto, Japan: IEEE, December 2007, pp. 443–448.

[8] S. Young, "ATK: An application toolkit for HTK," Machine Intelligence Laboratory, Cambridge University Engineering Department, Trumpington Street, Cambridge, CB2 1PZ, England, Tech. Rep., 2007, version 1.6. [Online]. Available: http://mi.eng.cam.ac.uk/research/dialogue/ATK_Manual.pdf

[9] D. Moore, J. Dines, M. Magimai Doss, J. Vepa, O. Cheng, and T. Hain, "Juicer: A weighted finite-state transducer speech decoder," in *Proceedings of the 3rd Joint Workshop on Multimodal Interaction and Related Machine Learning Algorithms*, 2006.

[10] P. N. Garner, J. Dines, T. Hain, A. El Hannani, M. Karafiát, D. Korchagin, M. Lincoln, V. Wan, and L. Zhang, "Real-time ASR from meetings," in *Proceedings of Interspeech*, Brighton, UK, September 2009.

[11] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[12] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–799, May 1995.

[13] D.-A. Manolescu, "A data flow pattern language," in *Proceedings of The 4th Pattern Languages of Programming Conference*, September 1997, Monticello, Illinois, USA. Washington University Technical Report 97-34. [Online]. Available: http://hillside.net/plop/plop97/Workshops.html

[14] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: a modular machine learning software library," Idiap, IDIAP-RR 02-46, 2002. [Online]. Available: http://publications.idiap.ch

[15] S. Young, G. Evermann, M. Gales, T. Hain, D. Kershaw, X. A. Lui, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, and P. Woodland, *The HTK Book*. Cambridge University Engineering Department, December 2006, version 3.4.

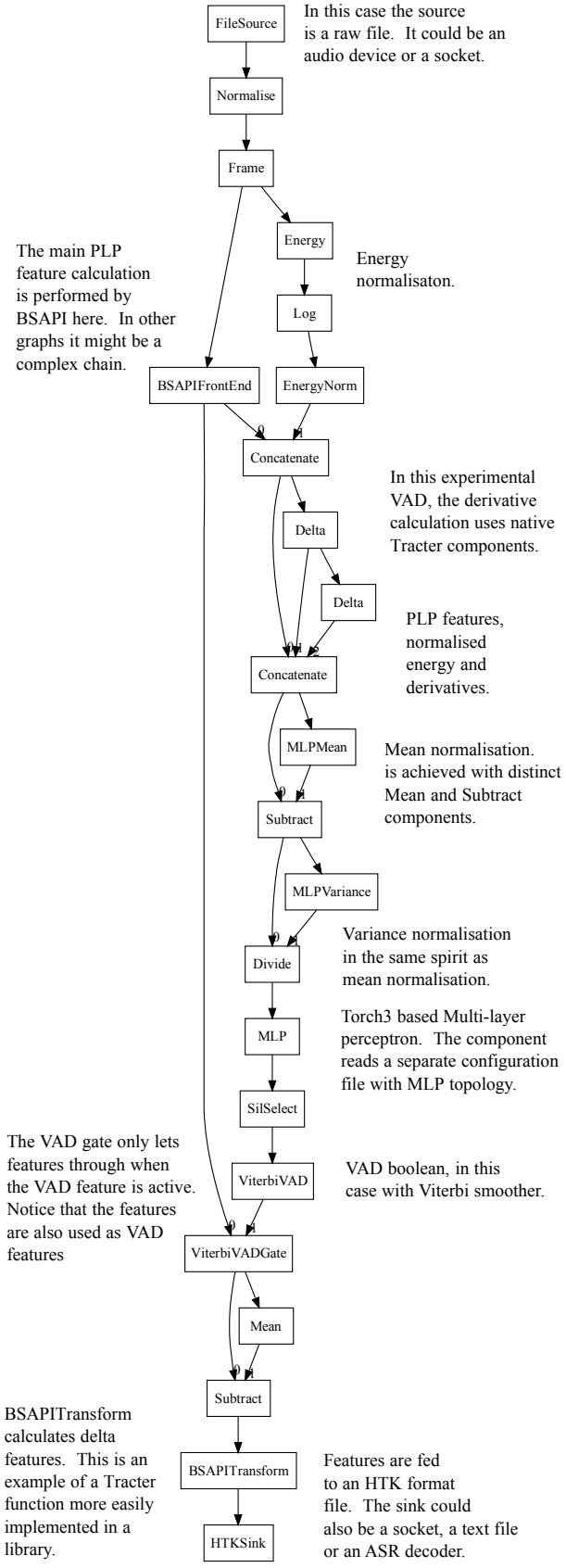

Figure 4: An annotated example graph containing integrated MLP based VAD, gate and feature calculation.