Abstract

Current voice recognition systems tend to be implemented as a PC desktop facility. This model is not suitable for the growing complexities of present and future developments: It is single-user, it is non portable, and it assumes the workstation model, where all the CPU resources are supposed to be locally available. This work researches how a high performance speech recognition system can be redesigned and implemented as a time-critical network service shared through ordinary data transmission media with three main design goals: Scalability, predictability and POSIX portability. The whole idea has been tested by rebuilding IVORY, a well known robust desktop voice recognition methodology, as a distributed component.

1. Introduction

While Speech Processing and Recognition is a field experiencing a rapid and promising expansion, the operating-system environments for the desktop PC still typically lack of true real-time support. To overcome this limitation, current speech recognition systems are confident on the workstation principle: all the CPU resources are always available to the application where they are embedded. This approach shows a main limitation: Its growing computational complexity. IVORY ([1], [7]), a stand-alone speech recognition system of isolated words, gives figures of computational complexity around 21 Mflop/s. Though this load is easily assumed by current CPU’s, continuous speech can raise the computing power demand one order of magnitude. Noise cancellation demands up to five or six times the power of the recognition itself. Furthermore, new applications of speech processing demand much more computing power. For instance, tracking a single speaker by the Microphone Arrays technique shows a computational complexity near 166 Mflop/s ([6]). Though today’s PC microprocessors claim peak execution rates exceeding 1 Gflop/s, regular DSP algorithms rarely result in such a high performance. In our view, desktop speech processing is-and will always be- strongly limited by its computational complexity, nowadays constrained to the computing power of the average personal computer.

Distributed computing should change this scenery. Ongoing developments on component based software engineering makes possible to envision a remote service of DSP computing power for speech processing. It would allow to bring both to the current desktop PC and to the future internet appliances the more advanced developments on the field. This work investigates the distribution of speech recognition in the context of DIARCA, a research project whose aim is two-fold. Firstly, to distribute IVORY with three design goals: Scalability, predictability and POSIX portability. Secondly, to extend the results in order to support microphone array developments. This work is about the first goal.

2. Going distributed

Figure 1 shows the implementation of IVORY on a PC hosted TI C32 based PCI carrier board ([7]). Noise canceller, extractor of signal parameters, template coding, template normalization and a Markov parsing operate in pipeline. This approach, based on low cost DSP coprocessors for a stand-alone PC, is too rigid. It shows a single-user, monolithic and closed implementation that results very hard to port to another platform. It is also difficult to change and to extend with new features, and it lacks of any scalability property.
In contrast, we conceive voice processing in terms of the client-server model: As a time-critical component simultaneously accessed by many people through ordinary data transmission media, as shows the Figure 2. This intended component should be easily scalable, both in hardware and software, in order to smoothly suit current and future voice processing algorithms and applications. DIARCA was born to pursue this goal.

3. Real-time: Choosing a solid foundation

Enough CPU is not enough. The main non-functional requirement of DIARCA servers and clients is timing. Timeliness is as important as functionality on voice recognition: The client must guarantee that voice frames are put on the cable at a constant rate. Similarly, the server should put codes of recognized words at the pace they are pronounced. This feature is impossible to guarantee in ordinary desktop operating system environments. Our experience makes clear, for instance, that plain LINUX can not cope with this hard real-time requirement. That's why we shifted to what we consider the right foundation: A real-time operating system (RTOS). DIARCA meets the requirement of predictability by adopting in both client and server the support of a well known RTOS, QNX RTP ((2)). QNX is a microkernel implementation of the POSIX specifications - Portable Operating System based on UNIX- ([3]). POSIX not only enables portability between POSIX compliant applications, but also predictability, because its real-time extension (POSIX 1003.1b) provides a well defined set of real-time facilities such as thread priorities and scheduling protocols.

Each client machine attaches to the DIARCA server through an associated session, a per-client bi-directional stream that carries speech frames at a constant data rate and recognition words (Figure 3). Currently, the session is implemented using Berkeley Sockets over the UDP protocol for the sake of performance. Once open, the session guarantees quality of service to the client: Enough CPU resources have been reserved in the server until the client closes the session.

4. Interface and architecture

A client application program uses DIARCA as a conventional library that provides the following C interface:

```c
int Drc_Open ();
int Drc_Get(int df, char *buf, int sz);
int Drc_Close(int df);
```

Drc_Open allows a client to initiate a session. This primitive is supposed to find the closer DIARCA server in the network. Once found, it invokes Open, that builds and sends a session request message. Drc_Open returns a session handler, a kind of file descriptor for further operations on the session. Drc_Open makes the client operating system to deliver a continuous stream of sound packets to the server. When the client user speaks, his/her voice is carried to the server. On silence periods, the sequence just carries the environmental noise. Drc_Get does not interact with the server. It suspends the invoking thread until either a word code or an error code arrives. Drc_Close closes the session that takes as parameter.

![Figure 4. The architecture of DIARCA](image)

Figure 4 shows the architecture of DIARCA. Any server, distributed or not, comes defined by what is referred to as its interface. For example, the interface of an hypothetical file server would be the prototypes of procedures such as open, read, write, seek and close. The advantage of this approach is that the interface hides to the client the complexities associated to the implementation of the service. The DIARCA server has been conceived as a virtual UNIX-like device that exports three methods, Open and Close to start and to terminate sessions on client demand, and Sample, that provides a block of 256 samples that operates on the pipeline. Expressed in a CORBA IDL style, the specification of the DIARCA component would be:

```idl
interface Diarca {
    int Open (void);
    int Close (int session);
    void Sample(int session, voice_t vce);
}
```

As Figure 4 shows, the DIARCA library internals use this interface. Open starts a pipeline in the server and returns a session handler to Drc_open. Close liberates the resources of the pipeline associated to the session. Sample feeds the server with more speech samples.

Traditionally, clients are perceived as active objects because they make decisions on their own. In contrast, servers are considered reactive objects because they only take actions when invoked. The DIARCA server synchronously starts pipelines and operate on them but also produces word codes when recognition events asynchronously happen. In order cope with these events, the DIARCA server incorporates a client -active- subcomponent charged to send these word codes to the client. Thus an internal server has been embedded on the client library. This server exports the method Word, dedicated to receive the recognized word code and to make a callback that delivers it to the DIARCA library. The callback acts as the interrupt service routine of the virtual
device. It, in turn, delivers the code for the consumption of the client application.

5. Library implementation

Fortunately, QNX has available a predictable and easy to use sound driver, based on LINUX’s ALSA. Figure 5 shows how the DIARCA library spawns three POSIX threads of control named Recorder, Sender and Receiver. Recorder continuously captures data from the sound card by using the ALSA driver, Sender sends the data to the server, hence its name, and Receiver waits for new messages coming to the server.

Figure 5. The implementation of the DIARCA library

Drc_Open initializes the library and creates the three threads. Then, it wakes up Sender, that builds a session request message (Figure 3) that is sent to the DIARCA server by invoking its interface function Open. After that, Sender gets suspended again. If the DIARCA server has enough CPU resources to support a new session, it replies with a message, carrying the session handler on success and an error code in other case. It is Receiver who receives this handler. Then it wakes up Recorder and goes to sleep. Recorder, from now on, will continuously loop to capture new 16 bits stereo, 2 channels, 11025Hz sampled data from the sound driver and put them on a limited buffer. Sender also loops to get data from this buffer and to put them on the output stream.

When the server recognizes a new word, it will send a message to the client bearing the code of that word. Each incoming message makes Receiver to wake up in order to signal the application thread that invoked Drc_get. Ready again, the application thread could now show an alert on the user interface, deploy a menu, etc. acting on the recognition event. Its asynchronous nature is the main reason why we have chosen this concurrent design. Drc_Close invokes the Close interface function of server and terminates the three threads.

6. Server implementation

Figure 6 shows the internals of a DIARCA server. As they are accepted, sessions are assigned to available pipelines until they get exhausted. Sessions run in concurrency (session concurrency). Inside each session, the different stages of the pipeline also run in concurrency (pipeline concurrency). The server manages a pool of threads following the boos/worker model, being each “worker” a whole pipeline.

Every incoming message of voice samples contains a field with its session identifier. The boss uses this field to route the frame to its assigned pipeline. The session identifier is propagated through the pipeline with the flow of data.

A new thread is introduced known as the assistant. It is charged to collect the results of all the pipelines -the recognized words- and to send them to the proper client, also in a service loop. In the client, it is the thread Receiver who receives the word (Figure 5). The boss runs a service loop that implements the DIARCA interface as follows:

```c
Boss() {
    Init_Sessions();
    Launch_Assistant();
    while(1) {
        Get_Request(&rqst);
        switch(rqst.type) {
            case OPEN:
                r = Open_Session();
            case CLOSE:
                r = Close_Session(rqst.session);
            case SAMPLE:
                r = Feed_Session(rqst.session,
                                rqst.data);
        }
        rqst.type = r;
        Put_Reply(&rqst);
    }
}
```

As the boss and the assistant, the stages run a service loop on its own execution context and communicate by means of mailboxes built as limited buffers:

```c
while (!end_of_session) {
    Receive(mbox_i, msg_i);
    msg_o = Do_Work(msg_i);
    Send(mbox_o, msg_o);
}
```
7. Work in progress

We have been working with Texas Instruments™ C40 and C6000 DSP processors for some time ([4]). Our goal is to achieve the hardware scalability of DIARCA by using the TIM standard. The new generation of Sundance™ SMT.350 C6000 multicomputer carrier boards are our chosen target platform for DIARCA. In this sense, we are developing PONNHI ([5]), a POSIX 1003.13 compliant kernel for C6000 processors. The boss/assistant paradigm allows to split the whole DIARCA component as a set of subcomponents, being a subcomponent one or more stages of the IVORY pipeline as Figure 7 shows. This approach enables the decoupling of stages for further distribution and/or parallelization of DIARCA over TIM modules.

Networks of DSP multicomputers as the manufactured by Sundance seem to be the natural solution for hardware scalability. This kind of DSP resources, however, are expensive, so we plan to open the service via CORBA. We expect CORBA help to DIARCA to become a component of today and tomorrow larger systems, potentially written in any language and running on any platform. TAO, a real-time CORBA implementation on QNX is the foundation of this research.

8. Conclusions

This work is mainly a proof of concept. The current implementation of DIARCA demonstrates that it is possible to extract a desktop speech recognition system as IVORY from the desktop environment and to rebuild it as a distributed component of a local area network, able to pay service to many clients simultaneously. Also, the work has identified, applied and tested some design principles that can be extended to other DSP problems. We have learned that:

- The use of a RTOS as QNX in both client and server has been determinant in the distribution of IVORY. In particular, rightly synchronized simultaneous registration and reproduction of voice has been achieved in the client by the close control of the sound card that QNX allows, something we failed to achieve with non real-time operating system environments.
- The POSIX standard has eased the development of the whole system and made it very portable.
- The architecture of the system, based on the execution of concurrent threads and the boss-worker model, has proven to be a good way to achieve the scalability of the DIARCA service as well as its multi-user capability.

9. References