ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1999
Seria: INFORMATYKA z. 37 Nr kol. 1421

Piotr FABIAN
Politechnika Śląska, Instytut Informatyki

SYSTEM ROZPOZNAWANIA MOWY W SIECI LOKALNEJ

Streszczenie. Artykuł opisuje implementację dostępu do systemu rozpoznawania mowy w lokalnej sieci komputerowej. Rozdzielenie zadania akwizycji i wstępnego przetworzenia sygnału od właściwego rozpoznawania pozwala przeznaczyć dedykowaną stację roboczą do rozpoznawania (wymagającego znacznej mocy obliczeniowej i pamięci), pozostawiając do wykonania stacjom - klientom tylko mało obciążające wstępne przetworzenie sygnału.

SPEECH RECOGNITION SYSTEM IN A LOCAL NETWORK

Summary. This article presents an implementation of a client-server architecture of a speech recognition system in a local network. Separating the task of collecting and front-processing the raw speech data from the proper recognition lets the recognition system (which is time- and memory consuming) run on a dedicated host. Clients have to do only speech recording and a small piece of initial processing, which does not require huge resources.

1. Wprowadzenie

Artykuł przedstawia implementację dostępu do systemu rozpoznawania mowy opartego na systemie ISADORA, działającą w lokalnej sieci komputerowej. System rozpoznawania mowy ISADORA został opracowany w ramach projektu SQEL-Copernicus 1634, dotyczącego automatycznego rozpoznawania i rozumienia pytań w różnych językach. Stosując technikę HMM rozpoznaje mowę ciągłą, przy ograniczeniu tematyki pytań do określonego, definiowalnego zakresu. Został on zaadaptowany przez autora do przetwarzania wypowiedzi w języku polskim [4]. W przykładowym systemie informacyjnym osiąga poprawność rozpoznawania ok. 99% przy kilkuset rozpoznawanych słowach i wypowiedziach (zdaniach)

o długości od kilku do kilkunastu słów. System przeznaczony jest zarówno do rozpoznawania jednego, jak i wielu mówców. W drugim przypadku konieczne jest „nauczenie" systemu brzmienia głosu co najmniej pięćdziesięciu różnych osób. Rozpoznawanie wypowiedzi trwa w przybliżeniu tyle czasu, ile wypowiedź na komputerze o mocy obliczeniowej porównywalnej z Pentium 200 MHz. Wymagania pamięciowe programu: od 10 MB do kilkuset MB, zależnie od stopnia komplikacji zbudowanego na podstawie danych uczących modelu mowy.

Jeśli rozpoznawanie mowy ma być tylko pomocniczym procesem pomagającym w obsłudze komputera, to rozsądne jest minimalizowanie zajętości zasobów przez tego rodzaju program. W lokalnej sieci można przeznaczyć jeden z komputerów do wykonywania zadania rozpoznawania, uruchamiając na innych tylko aplikację sterującą przesyłaniem danych do tej stacji i odbierającą wyniki. W opisywanej implementacji komunikacja między serwerem działającym w systemie Linux a klientem działającym w środowisku Windows realizowana jest poprzez gniazdka (ang. *sockets*).

2. Ogólne informacje o automatycznym rozpoznawaniu mowy (ARM)

Przez pojęcie „automatyczne rozpoznawanie mowy" (ang. *automatic speech recognition*) rozumie się proces przekształcający akustyczną reprezentację wypowiedzi w języku naturalnym na zapis w postaci symbolicznej (zwykle w postaci słów lub zdań jako tekstów ASCII), dokonywany bez udziału człowieka. Proces ten realizowany jest przez odpowiednie programy realizujące dosyć złożone algorytmy cyfrowego przetwarzania sygnałów i analizy procesów stochastycznych [6]. Dopiero od kilku lat odnoszone są sukcesy w postaci praktycznych realizacji ARM dla dużych słowników (kilkadziesiąt tysięcy wyrazów) i mowy ciągłej. Proces ARM wymaga bowiem znacznej mocy obliczeniowej i dużych ilości pamięci. Wcześniej wymagania te nie mogły być spełnione ze względu na zbyt wysoki koszt odpowiedniego sprzętu. Równolegle z rozwojem sprzętu powstają coraz doskonalsze algorytmy rozpoznawania mowy.

Najczęściej stosowanym we współczesnych systemach ARM podejściem do rozpoznawania mowy jest wykorzystanie ukrytych modeli Markowa (ang. *Hidden Markov Models*, HMM). Sygnał mowy jest zamieniany na postać cyfrową przez przetwornik analogowo-cyfrowy. Otrzymany strumień danych (kilkanaście tysięcy próbek na sekundę) dzielony jest na odcinki odpowiadające czasom od kilku do kilkunastu milisekund. Dla każdego takiego odcinka wyznaczany jest pewien zbiór kilku do kilkudziesięciu charakterystycznych parametrów (cech, ang. *features*). Parametry te powinny być zbliżone dla podobnych fragmentów mowy. Oczywiście w praktyce nie będą one identyczne nawet dla kilku takich samych wypowiedzi

jednej osoby; różne będą też czasy trwania wypowiedzi. System rozpoznawania mowy najpierw jest *uczony*: tworzy abstrakcyjną reprezentację różnych wzorców (np. słów), które ma rozpoznawać, w postaci sieci Markowa. Taka sieć może generować różne wersje danego wzorca, którym może być np. jedno słowo. Każdemu wzorcowi odpowiada w *nauczonym* systemie oddzielny model (HMM). Rozpoznanie nieznanej próbki (słowa) polega na odszukaniu w zbiorze zapamiętanych modeli wzorców takiego modelu, który „jest w stanie" wygenerować próbkę najbardziej zbliżoną do badanej.

Proces tworzenia modeli polega głównie na ustalaniu parametrów sieci probabilistycznej na podstawie skończonego zbioru wzorców. Jest to zadanie bardzo trudne do analitycznego rozwiązania. Dlatego stosuje się numeryczne metody znajdowania przybliżonych rozwiązań poprzez iteracyjne „poprawianie" parametrów. Dla dużych słowników otrzymuje się duże sieci, w których liczba wyznaczanych parametrów sięga kilku do kilkudziesięciu milionów. Proces uczenia jest bardzo czasochłonny. Wykorzystanie nauczonej sieci do rozpoznawania zajmuje znacznie mniej czasu i może być realizowane „w czasie rzeczywistym".

Nauczenie sieci wymaga danych uczących, czyli wielu wypowiedzi różnych osób. Zwykle większa liczba próbek wykorzystanych w procesie uczenia daje sieć lepiej rozpoznającą mowę. Przy ok. 50 osobach „uczących" sieć staje się niezależna od konkretnej osoby i rozpoznaje również te osoby, które nie brały udziału w procesie uczenia.

Model HMM można wykorzystać do zapamiętania wzorców pojedynczych słów. System będzie wtedy rozpoznawał izolowane słowa, a w każdym razie będzie wymagał wcześniejszego wyznaczenia granic słowa w dziedzinie czasu przed przystąpieniem do właściwego rozpoznawania (w mówionym języku nie występują odstępy czasowe między kolejnymi słowami, nie jest więc możliwe odnalezienie granic słów w prosty sposób). Rozpoznawanie mowy ciągłej można zrealizować np. przez zbudowanie modeli HMM uwzględniających nie pojedyncze słowa, ale np. ciągi słów, zdania itp. Wpływa to oczywiście na rozmiary modelu i czasy: uczenia i rozpoznawania.

3. Komunikacja przy użyciu gniazdek w systemie Linux

Sposób komunikacji przy wykorzystaniu gniazdek definiowany jest przez standard POSIX. Gniazdko jest uogólnionym kanałem komunikacji między procesami. Jest reprezentowane jako deskryptor pliku, podobnie jak np. potoki. Może jednak służyć do komunikacji między nie związanymi ze sobą procesami, również działającymi na różnych komputerach. Komunikacja może odbywać się w trzech trybach: datagramowym (ang. *datagram*), strumieniowym (ang. *stream*) i „surowym" (ang. *raw*).

W trybie datagramowym dane przesyłane są do odbiorcy w pakietach, z których każdy zawiera informację o adresie docelowym i porcję danych. Tryb ten nie gwarantuje dotarcia pakietów do odbiorcy w takiej kolejności, w jakiej zostały wysłane, a nawet nie gwarantuje dotarcia ich kiedykolwiek. Implementacja protokołu realizującego bezbłędny transfer danych w tym trybie wymaga dodatkowej kontroli odbioru pakietów, np. przez odsyłanie potwierdzeń otrzymania kolejnych porcji danych. Wysłanie każdego pakietu jest niezależne od pozostałych (nie jest jawnie nawiązywane stałe połączenie między komunikującymi się stacjami).

W trybie strumieniowym komunikację poprzedza nawiązanie połączenia. Dane mogą być przesyłane w obu kierunkach i dostarczane są we właściwej kolejności. Tryb ten jest asymetryczny. Serwer tworzy gniazdko przyjmujące z zewnątrz żądania połączenia. Takie żądanie może wysłać klient. Wtedy serwer tworzy gniazdko dla tego klienta i może się z nim komunikować. Jeden serwer może utworzyć na żądania klientów wiele takich połączeń i obsługiwać je „jednocześnie".

Tryb „surowy" pozwala wykorzystać niskopoziomowe funkcje protokołów sieciowych i jest rzadko używany.

4. Komunikacja przy użyciu gniazdek w systemie MS Windows

Specyfikacja gniazdek w systemie MS Windows oparta jest na implementacji *UNIX Berkeley Software Distribution 4.3*. Obejmuje ona zarówno funkcje zgodne z BSD, jak i charakterystyczne dla Windows. Również tu możliwa jest komunikacja w trybie datagramowym i strumieniowym.

Biblioteka *Microsoft Foundation Class Library* (MFC) zawiera implementacje klas ułatwiających wykorzystanie gniazdek. Są to klasy `CAsyncSocket`, `CSocket` oraz `CSocketFile`. Klasy te nie realizują jednak wszystkich niezbędnych operacji; programista musi zadbać o podobną organizację programu, jak w przypadku „czystych" funkcji BSD. Na potrzeby komunikacji między komputerami zaimplementowana została klasa `CSo2`, pozwalająca w łatwy sposób zrealizować dwustronne połączenie przez gniazdka w trybie strumieniowym.

5. Obsługa wejścia akustycznego w systemie MS Windows

Obsługa wejścia akustycznego, tj. przetwornika analogowo-cyfrowego karty dźwiękowej, jest w środowisku MS Windows realizowana przez sam system operacyjny, który wykorzy-

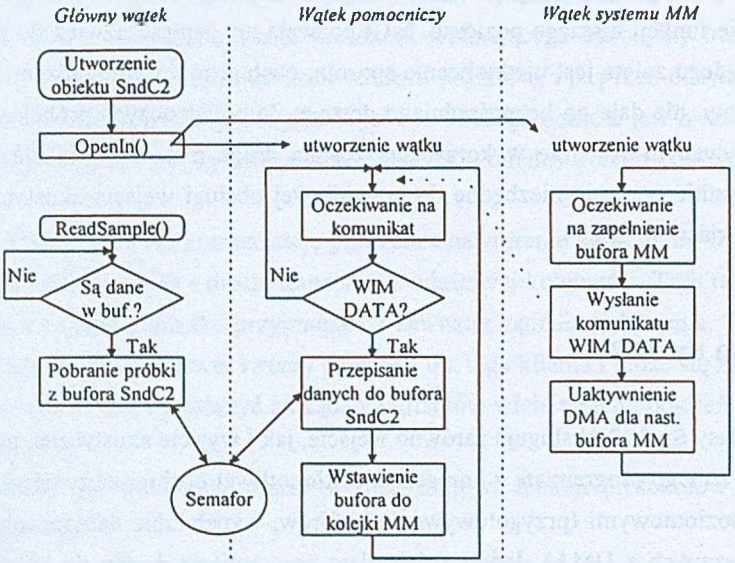
stuje sterowniki dostarczane przez producentów różnych kart. Dzięki temu różne karty mogą być obsługiwane w identyczny sposób. Windows oferuje programistom dwie metody obsługi wejścia akustycznego: przez wysyłanie komunikatów *Media Control Interface* (MCI) oraz wykorzystanie funkcji niskiego poziomu. MCI pozwala np. zapisać dźwięk do pliku, odtworzyć plik itp. Jego zaletą jest ujednolicenie sposobu obsługi różnych urządzeń „multimedialnych". Niestety, nie daje on bezpośredniego dostępu do pojedynczych próbek sygnału. Dlatego w opisywanym systemie wykorzystana została druga metoda - funkcje niskiego poziomu. Wszystkie operacje niezbędne do prawidłowej obsługi wejścia akustycznego wykonują metody klasy `SndC2`.

6. Klasa `SndC2`

Obiekt klasy `SndC2` obsługuje zarówno wejście, jak i wyjście akustyczne, zwalniając korzystającego z niego programistę z konieczności kłopotliwej obsługi karty dźwiękowej funkcjami niskopoziomowymi (przygotowywanie buforów, wypełnianie danymi, obsługa komunikatów związanych z DMA). Jednocześnie daje programiście dostęp do kolejnych próbek sygnału z niewielkim tylko opóźnieniem, zależnym od długości wewnętrznych buforów obiektu. Zmniejszanie ich długości zmniejsza opóźnienie, zwiększając jednocześnie obciążenie procesora. Jeśli sterownik karty dźwiękowej dopuszcza możliwość pracy w trybie dwukierunkowym (ang. *full duplex*), to również klasa `SndC2` może obsługiwać taki tryb. Należy wtedy utworzyć dwa obiekty tej klasy i użyć jednego jako wejściowego, a drugiego jako wyjściowego. Tym sposobem można łatwo implementować algorytmy przetwarzające sygnał w czasie rzeczywistym (np. filtrowanie sygnału). Klasa `SndC2` obsługuje wszystkie częstotliwości próbkowania i formaty danych (8/16 bitów, mono/stereo). Wartości próbek są przeskalowywane do zakresu (-1,1) i traktowane jako liczby rzeczywiste lub pary liczb rzeczywistych. Pozwala to uniknąć w programie korzystającym z `SndC2` konieczności przeliczania próbek zależnie od formatu. Rysunek 1 przedstawia schemat komunikacji między aplikacją, obiektem `SndC2` i pomocniczymi wątkami.

Główny wątek aplikacji tworzy obiekt klasy `SndC2`. Po wywołaniu metody `OpenIn` tworzony jest nowy wątek, przeznaczony do obsługi komunikatów podsystemu multimedialnego (MM). Komunikaty te docierają do aplikacji asynchronicznie w stosunku do operacji wykonywanych przez główny wątek, a muszą być obsłużone niemal natychmiast (inaczej mogą ulec „zgubieniu" próbki sygnału). Dlatego ich obsługą zajmuje się oddzielny wątek - w głównym wątku pętla obsługi komunikatów może działać wolniej. Dodatkowy wątek uru-

chamia sam podsystem multimedialny. Podczas działania obiektu klasy SndC2 uruchomione są więc współbieżnie trzy wątki.



Rys. 1. Schemat obsługi wejścia akustycznego przez obiekt klasy SndC2
Fig. 1. The way how a SndC2 object reads sound samples

Obiekt klasy SndC2 pozwala odczytywać i zapisywać kolejne próbki dźwięku podobnie jak pliki. Programista definiuje podczas otwierania wejścia lub wyjścia częstotliwość próbkowania i format (8/16 bitów, mono/stereo), następnie wykorzystuje metodę `ReadSample()` do czytania lub `WriteSample()` do pisania próbek jak w przykładach poniżej (pominięto testy błędów):

```
// odczyt próbek z wejścia
SndC2 *we;

we = new SndC2;
we->OpenIn(16000,TRUE,TRUE);

x = we->ReadSample();
x = we->ReadSample();
(...)

we->CloseIn();
delete we;
```

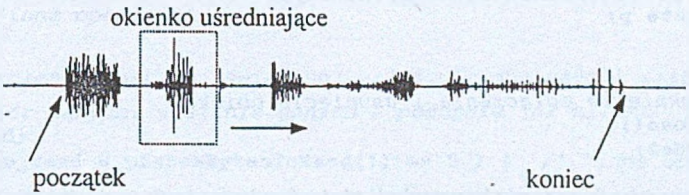
```
// zapis próbek do wyjścia
SndC2 *wy;

wy = new SndC2;
we->OpenOut(16000,TRUE,TRUE);

WriteSample(0.1,-0.8);
WriteSample(-0.45); // lewy=prawy
(...)

wy->CloseOut();
delete wy;
```

Możliwe jest również sprawdzenie aktualnego stanu bufora wejściowego (liczby próbek do odczytu) oraz liczby buforów wyjściowych czekających w kolejce do odtworzenia. Liczba i długość buforów mogą być ustalane przez programistę. Wybór dużych wartości pozwala na

długie przerwy w komunikacji z obiektem SndC2 (rzędu sekund) bez powodowania nieciągłości dźwięku, dzięki czemu proces przetwarzania próbek nie musi być równomierny w dziedzinie czasu. Tylko średnia prędkość przetwarzania powinna przekraczać wybraną częstotliwość próbkowania.

7. Klasa SndHist

Następnym elementem niezbędnym do uruchomienia klienta systemu rozpoznawania mowy jest moduł dokonujący wstępnego przetwarzania sygnału mowy. Obiekty klasy SndHist automatycznie wykrywają granice wypowiedzi (zdania). Jako kryterium przyjęto wartość energii w okienku o długości ustalanej podczas konstrukcji obiektu lub później oraz krótkookresową analizę wartości sygnału po wykryciu okienka spełniającego kryterium energetyczne. Klasa została zaimplementowana jako „przeźroczysta" dla aplikacji jej używającej – tak aby możliwe było przetwarzanie w czasie rzeczywistym. Kolejne wartości próbek są wpisywane do obiektu tej klasy metodą `double ReWr(double s)`, która jednocześnie zwraca wartość próbki, jednak niekoniecznie z tej samej chwili. Prywatnym polem klasy jest tablica „historii" sygnału, pozwalająca opóźnić sygnał o długość zdefiniowanego okienka. Pozwala to metodzie `IsLoud()` określić, czy zwracana przez `ReWr()` próbka należy do wypowiedzi, czy nie. Opóźnianie sygnału jest konieczne, ponieważ tę decyzję można podjąć dopiero po przeanalizowaniu próbek następujących *po* bieżącej próbce. Po wykryciu średniego poziomu sygnału w okienku przekraczającego ustalony próg dokonywana jest analiza przebiegu czasowego sygnału od początku lub końca tego okienka (zależnie od tego, czy wykrywany jest początek, czy koniec sygnału). Analiza ta zapobiega dołączaniu do wypowiedzi okresów ciszy na początku pierwszego i końcu ostatniego okienka (w których spełnione jest kryterium energetyczne). Natomiast samo uśrednianie energii sygnału w okienku o długości kilkudziesięciu do kilkuset milisekund zapobiega błędnej detekcji krótkiego odcinka ciszy w zdaniu jako końca zdania. Rysunek 2 przedstawia schemat działania klasy SigHist.



Rys. 2. Odszukiwanie granic wypowiedzi przez obiekt klasy SigHist
Fig. 2. Searching for borders of an utterance performed by a SigHist object

8. Klasa CSo2

Założeniem, które wpłynęło na sposób implementacji CSo2, było udostępnienie programiście gniazdek w podobny sposób jak plików. Ważne było, aby komunikacja mogła odbywać się „w tle", bez konieczności okresowego przerywania pracy aplikacji korzystającej z CSo2 przy odbiorze kolejnego pakietu danych. Wykorzystana tu została możliwość uruchamiania współbieżnie wykonywanych wątków w środowisku Windows.

Poniżej przedstawiono założony schemat komunikacji przy wykorzystaniu obiektów klasy CSo2.

8.1. Klient

```
    // utworzenie obiektu:
CSo2 *pCSo2;
pCSo2 = new CSo2;

    // próba nawiązania połączenia z portem 5550 serwera
    // melmac.iinf.polsl.gliwice.pl
pCSo2-> ConnectTo("melmac.iinf.polsl.gliwice.pl",5550)
if (pCSo2->Err()) {
    // próba nawiązania połączenia nie powiodła się. Kod błędu
    // i opis można odczytać z odpowiednich pól obiektu *pCSo2.
   delete pCSo2;
   return -1;
  }

    // wysłanie danych
char *s = "Ala ma kota";
pCSo2->Write(s,strlen(s)+1);

    // odbiór danych
int bytes_to_read = pSocSender->BytesToRead()
if (bytes_to_read == 0) {
    // nie ma żadnych danych do odczytu; można wykonywać
    // inne operacje i sprawdzić obecność danych później
}
else {
    // odczyt danych
   char *p = new char[bytes_to_read];
   if (p!=NULL) {
      pCSo2->Read(p,bytes_to_read);
    // ...przetworzenie odebranych danych
      delete p;
   }
}

    // zamknięcie połączenia i usunięcie obiektu
pCSo2->Close();
delete pCSo2;
```

Obiekt klasy CSo2 podczas konstrukcji alokuje własny bufor na odbierane dane i uruchamia wątek obsługujący komunikaty informujące o nadejściu danych. Dane wysyłane i odbierane są w pakietach. Długość pakietu ustala programista każdorazowo przy wysyłaniu da-

nych. Ze względu na specyfikę komunikacji podczas odczytu nie występuje tutaj stan „końca pliku". Można tylko określić, ile bajtów w danej chwili znajduje się w buforze obiektu klasy CSo2. Dane zapisywane są w nim w miarę otrzymywania kolejnych pakietów o długości zwykle większej niż jeden bajt. Koniec każdego pakietu może być identyfikowany przez zapamiętywanie długości kolejnych pakietów w kolejce FIFO, z której usuwany jest początkowy element po odczycie wszystkich bajtów odpowiadającego mu pakietu. Odczyt liczby ważnych bajtów w buforze dokonywany jest przez wywołanie metody BytesToRead(). Jeśli wynikiem jest zero, program może wykonać inne operacje. Dane, które w tym czasie mogą nadejść, zostaną zapamiętane w buforze dzięki współbieżnie działającemu wątkowi i mogą być później odczytane. Operacje czytania i pisania w razie potrzeby blokują wywołujące je funkcje (jeśli nie ma jeszcze wystarczającej liczby bajtów w buforze odbiorczym lub bufor nadawczy jest przepełniony).

Podczas próby łączenia z serwerem mogą wystąpić błędy:

- dostępu do sieci - jeśli protokół TCP/IP nie jest obsługiwany, np. z powodu braku karty sieciowej;
- dostępu do serwera - jeśli serwer o podanym adresie jest niedostępny (nie odpowiada) lub nie oczekuje na połączenie pod podanym numerem portu.

8.2. Serwer

```
   // utworzenie obiektu:
CSo2 *pLst;  // „Listener" - oczekuje na połączenie
pLst = new CSo2;

   // wprowadzenie obiektu *pLst w stan oczekiwania na połączenie
   // pod numerem portu 5550
pLst -> Listen(5550)
if (pLst -> Err()) {
   // nie powiodło się uruchomienie serwera
   delete pCSo2;
   return -1;
  }

   // teraz program może wykonywać inne operacje w oczekiwaniu
   // na przejście obiektu *pLst w stan "połączony"
while ( pLst->m_state != CSo2::CSO2_CONNECTED ) {
   // ..."inne operacje"
}

   // połączenie zostało nawiązane; należy je obsłużyć i zamknąć
   // odbiór danych, wysłanie danych - podobnie jak klient
int to_read;
while ( (to_read = pLst->BytesToRead()) == 0 ) {  /* "inne operacje" */
}
char *p = new char[to_read]; // bufor na dane odebrane
if (p!=NULL) {
   pLst->Read(p,to_read);
   // przetworzenie odebranych danych
   delete p;
```

```
}
char *s = "Ala ma kota";     // dane do wysłania
pLst->Write(s,strlen(s)+1);

   // zamknięcie połączenia i usunięcie obiektu
pLst->Close();
delete pLst;
```

Metoda `Err()` pozwala odczytać kod błędu ostatnio wykonanej operacji. Dodatkowo w odpowiednim polu obiektu zapisywane jest tekstowe objaśnienie błędu.

9. Serwer systemu rozpoznawania mowy

Jak wspomniano wcześniej, wykorzystywany jest system ISADORA, uruchomiony w środowisku Linux. Pracą systemu steruje oddzielny program obsługujący przyjęty protokół komunikacji. Odbywa się ona według schematu przedstawionego w tabeli 1.

Tabela 1

Schemat komunikacji między klientem a serwerem

Klient		Serwer
Oczekiwanie na wypowiedź	czas ↓	Oczekiwanie na zgłoszenie klienta
Zapamiętanie wypowiedzi (zdania)		
Nawiązanie połączenia z serwerem		Akceptacja połączenia z klientem
Wysłanie nagłówka pakietu danych		
		Potwierdzenie odbioru nagłówka
Wysłanie zapisanej wypowiedzi		Odbiór wypowiedzi
Oczekiwanie na wynik		Uruchomienie systemu rozpoznawania
		Wysłanie wyniku rozpoznawania
Odbiór wyniku		
Zamknięcie połączenia		Zamknięcie połączenia

10. Wnioski

Przedstawiona organizacja systemu rozpoznawania mowy w sieci lokalnej pozwala w łatwy sposób udostępnić dedykowany serwer wykonujący zadanie rozpoznawania większej liczbie komputerów, przy minimalnym wykorzystaniu ich zasobów. Obecna wersja wymaga przesyłania całego nagrania w postaci cyfrowej do serwera. Pojedyncza wypowiedź, wysyłana raz na kilkadziesiąt sekund, zajmuje zwykle od kilkudziesięciu do kilkuset kilobajtów, co nie stanowi dużego obciążenia dla sieci lokalnej. Przy wykorzystaniu odległego serwera (spoza sieci lokalnej) przesłanie takiej ilości danych może zabierać zbyt dużo czasu. Dlatego prze-

widywana jest modyfikacja systemu: przeniesienie ekstrakcji cech do komputera - klienta, co zwiększa obciążenie tego komputera, ale powoduje zmniejszenie obciążenia sieci.

Zainteresowanym osobom autor udostępni implementację opisywanych klas obsługujących wejście i wyjście akustyczne w środowisku MS Windows oraz obsługę gniazdek (kontakt: PCF@who.net).

LITERATURA

1. Borodziewicz W., Jaszczak K.: Cyfrowe przetwarzanie sygnałów, WNT, Warszawa 1987.
2. Rabiner L., Juang B.: Fundamentals of speech recognition, Prentice Hall PTR, New Jersey 1993.
3. Schukat-Talamazzini E.: Automatische Spracherkennung, Vieweg, Braunschweig 1995.
4. Fabian P.: Adaptation of the SQEL speech recognition system for the Polish language, 2nd SQEL Workshop, Pilzno, 27-29 IV 1997.
5. Petzold Ch., Yao P.: Programowanie Windows 95, Microsoft Press, Warszawa 1997.
6. Czyżewski A.: Dźwięk cyfrowy. Wybrane zagadnienia teoretyczne, technologia, zastosowania, Akademicka Oficyna Wydawnicza EXIT, Warszawa 1998.

Recenzent: Prof. dr hab. inż. Czesław Basztura

Wpłynęło do Redakcji 17 grudnia 1998 r.

Abstract

This article presents an implementation of a client-server architecture applied to a speech recognition system. The recognition "engine" uses ISADORA, a HMM based system developed as a part of the SQEL-Copernicus 1634 project (Spoken Queries in European Languages). The objective of the implementation described here was to separate the initial processing of the speech data from the proper recognition. Speech recognition needs huge resources: tens of megabytes of memory and a fast workstation. If the recognition is only an auxiliary feature added to an application, it shouldn't take all the resources available. Thus, in a local computer network, one workstation may become a dedicated server engaged with speech recognition only (and having enough resources for this task), while other computer run only small client interfaces, performing sound recording, initial processing and communica-

tion. The server uses a workstation running Linux, clients use MS Windows 95. These computers work in a local network (10 Mb/s Ethernet). Special classes have been developed for sound recording (`SndC2`) and socket based communication (`Cso2`). Objects of both classes behave like files, allowing treating sound recording similar as file reading and socket communication like file reading and writing. Since they use multithreading, they work in Windows 9x / NT only. A protocol between clients and the server has been defined and tested.

Interested developers may get the implementation of classes described here (contact address: PCF@who.net).