



Praktikum Kommunikationssysteme (SS04)

Aufgabe 4: Audio- und Videosynchronisation

Letzter Abgabetermin Konzept: 09. Juli

Letzter Abgabetermin Programm: 23. Juli

1 Audio- und Videosynchronisation

Multimedia-Systeme werden häufig dadurch charakterisiert, dass Daten von mindestens zwei unterschiedlichen Medien mit Hilfe eines Rechners verarbeitet werden. Die gemeinsame Verarbeitung unterschiedlicher Medien macht es notwendig, die Medien zueinander in Beziehung zu setzen. Als Synchronisation wird dabei die Definition und Einhaltung räumlicher und zeitlicher Beziehungen zwischen Multimedia-Objekten verstanden. In vielen fortgeschrittenen Anwendungen, wie z.B. Videokonferenzsystemen, Video-Servern, Multimedia-Datenbanken oder Anwendungen aus dem Bereich der rechnerunterstützten Gruppenarbeit (CSCW) treten Synchronisationsprobleme auf.

In dieser Praktikumsaufgabe wird eines dieser Probleme, die Lippensynchronisation ([1, 2]), genauer betrachtet. Ziel ist es dabei, die Darstellung eines Video- und eines Audiostroms so zu koordinieren, dass Bild und Ton in dem bei der Aufzeichnung bestehenden Verhältnis zueinander präsentiert werden. Hinkt der Ton bzgl. seines korrekten Abspielzeitpunktes hinterher oder eilt er voraus, wird diese Zeitdifferenz als *Skew* bezeichnet (s. Abb. 1). Der Ausdruck *Lippensynchronisation* rührt daher, dass der Skew besonders leicht bei Kopfansicht eines Sprechers festgestellt werden kann, wenn die Lippenbewegungen nicht mit dem Ton übereinstimmen. Tatsächlich muss die Übereinstimmung nicht perfekt sein, ein kleiner Skew bleibt in der Regel unentdeckt. Allerdings sind bei der Lippensynchronisation die Anforderungen für die zu erzielende Synchronisation besonders hoch. In Experimenten konnte nachgewiesen werden, dass ein Skew im Bereich von etwa ± 80 ms vom Betrachter nicht festgestellt wird. Liegt der Skew im Bereich von 80–160 ms, wird er zwar bemerkt, aber nicht unbedingt als störend empfunden, während darüberliegende Werte den Eindruck nachhaltig beeinträchtigen.

In einer verteilten Anwendung wie z.B. einem Videokonferenzsystem müssen die Datenströme über ein Netzwerk transportiert werden. Idealerweise werden die kontinuierlich auftretenden Daten dazu vom Transportsystem isochron übertragen und können an der Senke ohne weitere Synchronisationsmaßnahmen präsentiert werden. Heutige Transportsysteme unterstützen jedoch häufig keinen isochronen Datenverkehr, so dass die Zwischenankunftszeiten, d.h. die Zeit,

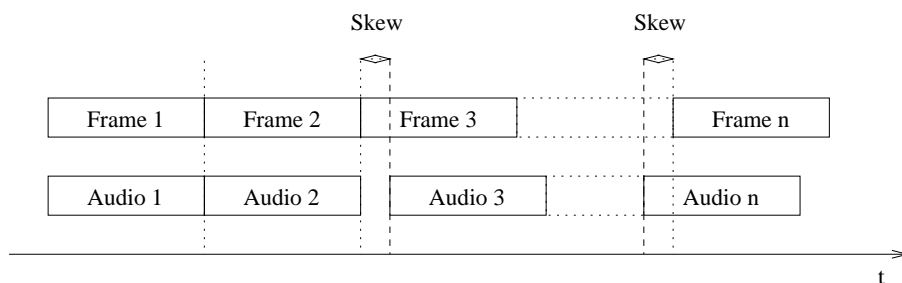


Abbildung 1: Skew bei der Präsentation von Audio- und Videodaten

welche zwischen der Auslieferung zweier aufeinanderfolgender Datenpakete vergeht, variiert¹. Die Synchronisation, die hier notwendig ist, wird *Intra-Objekt-Synchronisation* genannt. Für ein Video bedeutet das, dass die einzelnen Frames möglichst gleichmäßig angezeigt werden können.

Werden Audio- und Videodaten getrennt übertragen, wie dies häufig der Fall ist, tritt zudem das oben bereits angesprochene Problem auf, dass die Übertragungsverzögerungen zusammengehöriger Audio- und Videodaten im Allgemeinen unterschiedlich sind und so keine Lippenynchronität mehr gewährleistet ist. Diese Synchronisationsaufgabe wird als *Inter-Objekt-Synchronisation* bezeichnet.

Aufgabe ist es nun, kontinuierlich Audio- und Videodaten aus entsprechenden Filmdateien auszulesen, über das lokale Netz an einen anderen Rechner zu senden und dort in zeitlich korrektem Verhältnis an den Bildschirm und das Audio-Device weiterzugeben. Besonderer Wert wird auf eine präzise Synchronisation von Bild und Ton und eine möglichst unterbrechungsfreie Tonwiedergabe gelegt.

2 Format der Filmdateien

Die Filme sind im Verzeichnis `/home/pks-public/AudioVideo/movies` abgelegt und bestehen aus einer Menge von Dateien der Form `Name.Nummer`.

`Name` kann die folgenden Werte annehmen:

- `mcg`: Vorspann der Serie 'McGyver'
- `men`: Szene aus 'Men in Black'
- `naked`: Szene aus 'Die nackte Kanone' (Referenzfilm: sehr schön sichtbar, ob synchron oder nicht synchron)
- `roberto`: Waschmittelwerbung, bei der man Lippenynchronität testen kann

¹Diese Variation bezeichnet man als *Jitter*.

Jede Datei enthält genau ein Bild, welches wiederum aus mehreren Bildblöcken mit Steuerinformation und den dazugehörigen Audio-Informationen (genau ein Audio-Block pro Datei) besteht.

Jede Bilddatei beginnt mit folgendem Dateikopf:

```
/* Bildheader */
int msec;           /* Zeit seit Start des Filmes    */
                    /* in Millisekunden          */

/* Format-Block */
int marker;         /* DGRAM_INFO = 2          */
int width, height;  /* Bild-Breite und Höhe     */
                    /* (im Film konstant)       */
int is_bw;          /* Bild ist schwarz-weiß    */
                    /* (z.Zt. immer FALSE)      */
                    /* wird nicht mehr benötigt, */
                    /* hat aber historischen Wert :) */
int block_anz;      /* Anzahl Bild-Blöcke pro Bild */
int bytes_per_block; /* Größe eines dekomprimierten */
                    /* Bild-Blockes (in Bytes)    */
int audio_buffer_size; /* Bytes pro Audio-Block    */
```

Auf diesen Header folgt eine Anzahl von Video-Blöcken und ein Audio-Block. Jeder dieser Blöcke beginnt, wie schon der Format-Block, mit einem Marker. Ein Videoblock hat das folgende Format:

```
/* Format des komprimierten Bild-Blockes */
int marker;         /* DGRAM_VIDEO = 0          */
int block_nummer;    /* laufende Nummer des Blockes */
                    /* im Film                  */
int block_size;      /* Größe des komprimierten    */
                    /* Blocks in Bytes          */
char daten[VID_SIZE]; /* Video-Daten, 8 Bit Farbe  */
                    /* Anzahl wie in            */
                    /* block_size angegeben     */
```

Ein Audioblock hat folgende Gestalt:

```
/* Format des Audio-Blockes */
int marker;         /* DGRAM_AUDIO = 1          */
char daten[AU_SIZE]; /* 8-bit-ulaw, Anzahl wie in */
                    /* audio_buffer_size        */
```

Die Idee hinter diesem Format ist, einzelne Format-, Video- und Audioblöcke als jeweils eigene UDP-Datagramme zu versenden.

Aus dem Kopf einer Bilddatei können die maximal nötige Puffergröße ermittelt und Informationen gewonnen werden, die zur korrekten Rekonstruktion der Bilddaten benötigt werden.

Die Daten sind gemäß Sparc-Architektur big-endian codiert. Das muß berücksichtigt werden, wenn auf little-endian Systemen (wie Intel) entwickelt und getestet werden sollte. Im Zusammenhang mit der Sparc-Architektur ist auch zu beachten, daß Speicherzugriffe type-aligned erfolgen. So können beispielsweise 32bit-Werte nur an durch vier teilbare Speicheradressen stehen.

3 Zu implementierende Funktionen

Für diese Aufgabe sind treibernahe sowie GUI-Programmierung nötig, die eine umfangreiche Einarbeitung erfordern. Daher werden Bibliotheken und ein Gerüst vorgegeben, die im Ordner `/home/pks-public/AudioVideo/template/` abgelegt sind, der von jeder Gruppe in das eigene Verzeichnis zu kopieren ist.

Darin sind enthalten:

`./framework`

Hier ist das Gerüst für den Client und den Server mit Makefile und einem Skript zum Setzen der Netbug-Parameter (s.u.) enthalten, der als Basis für die eigene Implementierung dient. Den Inhalt dieses Ordners bitte kopieren und erweitern.

`./include`

In diesem Verzeichnis sind gemeinsame Headerdateien abgelegt, in denen u.A. die bereitgestellten Funktionen oder der Pfad zu den Videodaten (in `./include/video.h` unter `MOVIE_DIRECTORY`) deklariert sind. Im Makefile des zu nutzenden Gerüsts ist ein entsprechender Include-Pfad gesetzt.

`./gtk_client`

Die GUI des Clients ist hier als Bibliothek abgelegt. Die Linkoptionen sind im Makefile des Gerüsts bereits eingetragen. Die GUI wird über das GTK+ realisiert und ist (bis auf Sparc-spezifische Zugriffe auf das Audiodevice) portabel.

`./Libnetbug`

Hier ist die Bibliothek enthalten, die Netzwerkfehler simuliert (s.u.); auch dafür sind die Linkoptionen im Makefile des Gerüsts gesetzt.

Es ist zum einen der Server zu realisieren, der die Bilddateien liest und diese Daten per Unicast zum Client sendet. Zum anderen ist der Client zu implementieren, der die Daten empfängt, in die korrekte Reihenfolge bringt, für die notwendige Synchronisation zwischen Video- und Audiodaten sorgt und diese anschließend zur Anzeige bzw. zu Gehör bringt.

Bei Konferenzapplikationen, die nicht in einem lokalen Netz ablaufen, müssen Effekte wie Paketverluste, Paketverdopplungen etc. einkalkuliert werden. Derartige Situationen treten im LAN des Instituts normalerweise nicht auf, d.h. die zu erwartenden Probleme bzgl. der Synchronisierung und Darstellung der Video/Audiodaten sind relativ gering. Dennoch ist es mit der sogenannten „Libnetbug“ möglich, solche Probleme zu simulieren (siehe Abschnitt 4).

Bei Videokonferenzen oder Streaminganwendungen wird das Audio höher als das Video priorisiert, da einerseits Fehler im Audiostrom eher und stärker wahrgenommen werden, und andererseits Videodaten meist höhere Datenraten haben. In der zu implementierenden Lösung sollen entsprechend verlorene Audiopakete wiederholt angefordert werden; bei Verlusten im Videostrom kann hingegen auf Korrekturmaßnahmen verzichtet werden.

Anmerkung: Diese Einschränkung ist nur deshalb möglich, weil alle Einzelbilder unabhängig voneinander codiert sind. Üblicherweise werden beim Streamen zeitliche Redundanzen entfernt, so daß Folgebilder voneinander abhängig sind und daher in jedem Fall Fehlerbehandlungen vorzusehen sind.

3.1 Der Server

Der Server besteht aus einem Programm (`vserv.c`), welches die Bilddateien einliest und die in ihnen enthaltenen Video- und Audiodaten per Unicast an den Client sendet. Als Übertragungsprotokoll ist UDP zu wählen, wobei darauf zu achten ist, dass jeweils nur *ein* Video- oder Audioblock pro UDP-Paket gesendet wird.

Allerdings ist es erlaubt, Steuerbefehle wie z.B. Sendewiederholungsanforderungen getrennt über eine TCP-Verbindung zu verschicken, um zu gewährleisten, dass diese zuverlässig übertragen werden (s.u.).

3.2 Der Client

Das bereitgestellte Gerüst ist von den Teilnehmern um die nachfolgend aufgeführten Funktionen zu erweitern (in der Datei `vclient.c`).

Sowohl der Name des Servers als auch der Port müssen `vclient` in der Kommandozeile übergeben werden. D.h. der Aufruf von `vclient` lautet z.B.: `vclient meister 4711`.

```
void init(char *hostname, int port, char *movie)
```

Wird als erste Funktion nach Programmstart aufgerufen, d.h. in dieser sind notwendige Initialisierungen von Daten, das Öffnen von Sockets etc. durchzuführen. Dabei ist der erste Parameter der Name des Rechners, auf dem der Server gestartet wurde und der zweite Parameter der Port, auf dem der Server Anfragen entgegennimmt. Der dritte Parameter ist der Name des Filmes, der abgespielt werden soll. In dieser Funktion darf *nicht* die Funktion `init_audio_video` (s.u.) aufgerufen werden, `init` ist nur für die Initialisierung der Kommunikation zu verwenden!

`void main_loop()`

Wird periodisch aus einer Schleife des Hauptprogramms aufgerufen. Diese Funktion muss zuerst Pakete vom Server empfangen. Dabei sollte sie nicht nur ein Paket pro Aufruf von `main_loop()` annehmen, sondern so viele wie möglich. Ansonsten besteht die Gefahr, dass Pakete verloren gehen.

Innerhalb dieser Funktion sind empfangene Videoblöcke mittels der Funktion `show_video` (s.u.) darzustellen. Es ist wichtig, dass innerhalb dieser Funktion **keine** blockierenden Systemaufrufe getätigt werden oder aktiv gewartet wird, da ansonsten die kontinuierliche Darstellung der Videobilder nicht durchgeführt werden kann.

Sollte trotz mehrmaliger Sendewiederholung ein Audio-Block fehlen, so ist das bei vereinzeltem Auftreten nicht schlimm. Solch ein fehlender Block sollte durch Nullen ersetzt ausgegeben werden.

`void feierabend()`

Diese Funktion wird, wie der Name verrät, unmittelbar vor Beendigung des Clientprogramms aufgerufen. Sie kann benutzt werden, um nicht mehr benötigte Ressourcen, wie z.B. Socket-Deskriptoren, dynamisch allozierten Speicher etc., freizugeben.

Da, wie im folgenden Kapitel beschrieben, ein internes Speichermanagement für die Videodaten existiert, darf für evtl. Fehlerfälle der client nicht einfach mit `exit` oder in der Funktion `main` mit `return` beendet werden. Stattdessen soll die vorgegebene Funktion `graceful_exit()` benutzt werden. Damit desweiteren alle für diese Aufgabe vorgegebenen Teile richtig funktionieren, muss unbedingt sichergestellt werden, dass die verwendeten UDP Portnummern einen größeren Wert als 3000 haben. Damit der Client nicht mit Paketen vom Sender überschwemmt wird, ist als einfache Lösung die Implementierung eines kleinen Flusskontrollprotokolls erforderlich. Der Client sollte also dem Server Rückmeldungen über den Zustand seiner Puffer liefern. Der Server sollte daraufhin mit einer Erhöhung/Verringerung der Rate reagieren, mit der er Audio-/Video-Pakete ausschickt. Eine Lösung ohne Flusskontrollprotokoll nur durch ein 'geschicktes' Timing des Abschickens der Pakete auf dem Server funktioniert im Allgemeinen nicht. Zum Versenden der Nachrichten für das Flusskontrollprotokoll kann TCP verwendet werden.

Erfahrungsgemäß ist es nicht möglich, flüssiges Video und Audio zu erhalten, wenn man jedes Paket einzeln anfordert. Es sollten also kumulative Anforderungen verwendet werden. Andererseits sollten nicht alle Pakete auf einmal verschickt werden. Hier ist also ein Mittelweg zu finden.

3.3 Funktionen zur Bilddarstellung und Audioausgabe

Bei der Versendung von farbigen Bildern über ein Netzwerk ergibt sich das Problem der Datengröße. Beispielsweise benötigt ein 320x240 Pixel großes Videobild in 8 Bit Farbtiefe 76800 Bytes. Sollen 12 Bilder pro Sekunde in einem Film dargestellt werden, ergibt sich eine Netzlast von etwa 920 kBytes pro Sekunde. Um diese zu reduzieren, sind in dieser Praktikumsaufgabe die Bilder komprimiert im JPEG Format gespeichert. Ein darzustellendes Bild besteht dabei

aus mehreren Bildblöcken, die jeweils separat im komprimierten JPEG-Format vorliegen. Dadurch reduziert sich die Bilddatengröße auf ca. ein Zehntel des ursprünglichen Wertes. Nachteil dieser Methode ist, dass auf den Clients die JPEG-Bildblöcke noch dekomprimiert werden müssen. Da diese Dekomprimierung zeitkritisch ist, stehen für diese Praktikumsaufgabe vorgefertigte Routinen für die Bilddarstellung / -verarbeitung zur Verfügung, die insbesondere zeitaufwendige Kopieroperationen im Speicher vermeiden. Dazu wird ein interner Puffer angelegt, in dem Bildblöcke abgelegt (dekomprimiert) werden können. Aus diesem Puffer können einzelne Bilder bei Bedarf im Videofenster angezeigt werden.

Noch ein Hinweis: Da die Dekomprimierung auf dem Client viel Rechenzeit benötigt, sollten eventuelle Ausgaben des Clients zunächst in eine Datei auf der lokalen Festplatte umgeleitet werden. (z.B.: `vclient <server> <port> <movie> > /tmp/clientout` Die Ausgabe kann anschließend mittels `more /tmp/clientout` oder in einem beliebigen Editor angesehen werden. Werden die Ausgaben nicht umgeleitet, nehmen diese u.U. soviel Rechen- und I/O-Zeit in Anspruch, dass eine synchrone Wiedergabe von Audio und Video trotz korrekter Programmierung nicht mehr möglich ist! Genauso sollte angestrebt werden, möglichst wenig Speicheroperationen wie `memcpy` o.Ä. zu verwenden, da auch diese viel Rechenzeit in Anspruch nehmen. Falls für die Lösung der Aufgabe Timer oder Signale eingesetzt werden, dürfen im Handler keine zeitaufwendigen Aktionen (wie z.B. Grafikausgabe) ausgeführt werden.

Für die Ausgabe von Audiodateien wird zunächst das Audiodevice mit den Parametern der auszugebenden Daten initialisiert. Für die Ausgabe selbst sind zwei Funktionen vorgesehen: die Ausgabefunktion `play_audio_block` erwartet als Parameter einen Audiobuffer und die Länge der zu übertragenden Bytes. Mit der Abfragefunktion `get_completed_audio_blocks` kann die Anzahl der bereits ausgegebenen Audioblöcke abgefragt werden. Gibt man konsequent bei jeder Ausgabe gleich große Audioblöcke aus, ist damit die Anzahl der abgespielten Samples und mit bekannter Samplerate eine zeitliche Zuordnung und damit eine Synchronisation möglich.

Die zu verwendenden Funktionen sind im Einzelnen:

```
void init_audio_video(FORMAT * video, int picnum)
```

Diese Funktion initialisiert das Bildfenster und die dazugehörigen X-Ressourcen über GTK+. Das Audiodevice wird mit den Parametern der auszugebenden Daten (8-bit- μ law, 8 kHz, mono) initialisiert.

Als Parameter erhält diese Funktion einen Zeiger auf eine `FORMAT`-Struktur, aus der u.a. die Bildbreite und -höhe sowie die Größe eines dekomprimierten Bildfragmentes entnommen werden können. Der zweite Parameter *picnum* gibt die Größe des internen Puffers in Bildern zur Speicherung der Videobilder an. D.h. mit `picnum = 1` kann der interne Puffer ein Bild speichern, bei `picnum = 2` zwei Bilder usw. Der maximale Wert für *picnum* beträgt 80, d.h. es ist nicht möglich (und auch nicht erlaubt) den ganzen Film beim Client zu puffern.

Achtung: Diese Funktion darf nicht aus `init` aufgerufen werden, sondern muss in `main_loop` beim ersten Durchlauf ausgeführt werden!

```
int decomp_JPEG (char* data, int data_size, int offset)
```

Diese Funktion übernimmt das Dekomprimieren eines Bildblocks (Teilbildes) im JPEG-Format. Dazu muss in *data* ein Zeiger auf den Puffer übergeben werden, in dem der

JPEG-komprimierte Bildblock abgelegt ist, und in *data_size* die Größe des JPEG-Datenblocks angegeben werden. Da diese Funktion die dekomprimierten Daten in den internen Puffer schreibt, muss mittels des dritten Parameters *offset* angegeben werden, an welche Stelle des internen Puffers der dekomprimierte Bildblock geschrieben werden soll. Dieser offset ist absolut, d.h. ein offset von 100 bedeutet, dass das erste Pixel des dekomprimierten Bildblockes an die 100. Stelle (Byte) im Puffer geschrieben wird und die darauf folgenden fortlaufend dahinter.

Der Rückgabewert von `decomp_JPEG` ist 1, falls das Dekomprimieren erfolgreich war, bzw. 0, falls nicht.

void show_video(int y_offset)

Diese Funktion zeigt ein einzelnes Bild an der durch den Parameter `y_offset` angegebenen Stelle im internen Puffer im Videofenster an. Zu beachten ist hier, dass (aus technischen Gründen) als Parameter nur der offset in y-Richtung im Puffer angegeben werden muss. Beispiel: Wurde mittels `init_video` ein Puffer für 5 Bilder der Größe 640x480 Pixeln angelegt, zeigt `show_video(0)` das erste Bild im Puffer an, `show_video(480)` das zweite Bild, `show_video(960)` das dritte Bild usw.

void play_audio_block(char *data, int length)

Mit dieser Funktion werden `length` Bytes der im Buffer `data` übergebenen Daten in die internen Buffer des Audiodevice kopiert, die mindestens 32000 Bytes lang sind. Werden die Buffer nicht schnell genug nachgefüllt, ist deutlich hörbares Knacken und ähnliche Störgeräusche wahrnehmbar. Der Rückgabewert der Funktion gibt die Anzahl der tatsächlich geschriebenen Bytes zurück.

int get_completed_audio_blocks(void)

Diese Funktion gibt die Anzahl der Audioblöcke zurück, die in den internen Puffer des Audiodevice geschrieben wurden. Wurden Blöcke konstanter Größe geschrieben, ist damit ein Rückschluß auf die Gesamtdaten und damit der Laufzeit möglich.

4 Die Libnetbug

Da auf dem lokalen Netz nur mit wenigen Übertragungsfehlern zu rechnen ist, existiert eine Bibliothek zur Erzeugung von künstlichen Übertragungsfehlern („Die Libnetbug“), welche nur zum Server hinzugelinkt wird. Durch Verändern der Parameter im Skript `netb.sh` und anschließendes Ausführen des Skriptes mittels:

```
source netb.sh
```

können u.a. Fehlerwahrscheinlichkeiten beim Aussenden der UDP Pakete durch den Server verändert werden. Dieses Skript muss auf dem Rechner und in der Shell / in dem Terminal ausgeführt werden, in der auch der Server `vserv` ausgeführt wird.

5 Abnahmekriterien

Für ein erfolgreiches Testieren ist auch das Verhalten des Zusammenspiels von Server und Client unter schlechten Bedingungen (d.h. bei hohen Paketverlustraten etc.) maßgebend. Genauer gesagt: Abnahmekriterium für die Aufgabe ist, dass bei einer Paketverlustrate von 40% (eingestellt in der libnetbug) das Audio noch fehlerfrei zu hören ist. Da für das Video keine Übertragungswiederholungen vorgenommen werden, sieht dieses natürlich sehr schlecht aus, was aber der Abgabe nicht im Wege steht. Bei einer Paketverlustrate von 0% muss allerdings auch das Video flüssig und nahezu fehlerfrei zu sehen sein (nahezu weil selbst im lokalen Netz ein einzelnes Paket verloren gehen kann).

Literatur

- [1] R. Steinmetz. *Multimedia-Technologie: Einführung und Grundlagen*. Springer, Berlin, 1993.
- [2] R. Steinmetz and K. Nahrstedt. *Multimedia: Computing, Communications and Applications*. Prentice Hall, Upper Saddle River, 1995.