Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund
Kommunikation und Multimedia
Prof. Dr. L. Wolf

Praktikum Kommunikationssysteme im SS06

– Aufgabenbeschreibung –

Betreuer: Zefir Kurtisi, Habib-ur-Rehman, Alexej Beresnev

Aufgabe 3: Audio-Video Synchronisation (AVS)

Termine

Konzept	06.07.06
Abgabefrist	27.07.06

1 Überblick

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, welche zwischen der Auslieferung zweier aufeinanderfolgender Datenpakete vergeht, variiert[1]. Die Synchronisation, die

[1] Diese Variation bezeichnet man als *Jitter*.

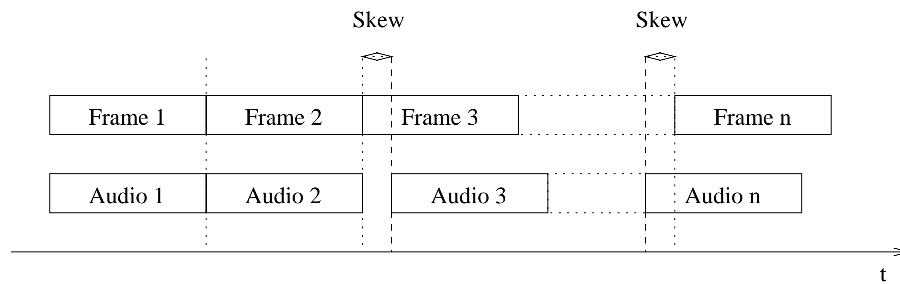

Fig. 1: Skew bei der Präsentation von Audio- und Videodaten

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 Lippensynchronität 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 Bereitgestellte Daten und Programm-Gerüst

Im für diese Aufgabe bereitgestellten Archiv sind ein Programmgerüst und eine Musterlösung enthalten. Aufgrund ihrer Größe liegen die Videodateien in einem freigegebenen Ordner, der im Quellcode referenziert wird. Im Archiv sind enthalten in Ordner

`template`
: das Programm-Gerüst mit Funktionen zur Audio- und Video-Ausgabe.
Client und Server benutzen gemeinsame Definitionen aus dem `./include`-Verzeichnis.
Für den Client ist im `./client`-Ordner das Programmgerüst in der Datei `avs-client.c` enthalten. Die Bibliothek `libavs-client.a` enthält die Funktionen zum Zugriff auf Audio und Video. Sie wird bei Verwendung des beiliegenden Makefiles eingebunden.
Im `./server`-Verzeichnis liegen Quellcode und Bibliotheken zum Erzeugen des Servers.

`muster`
: die Binaries einer auf das Programm-Gerüst basierenden Musterlösung, um die Machbarkeit zu dokumentieren. Beispielhafter Aufruf des Servers mit `./avs-server 5000` und des Clients mit `./avs-client <server-ip> 5000 naked` sollte ein Video versenden und abspielen.

3 Format der Filmdateien

Die Filme sind im IBR-Video-Format (`ibrv`) abgelegt. Neben anderen sind enthalten

- `naked`
  Szene aus 'Die nackte Kanone' (Referenzfilm: sehr schön sichtbar, ob synchron oder nicht synchron)

- `alice`
  längerer Werbefilm, mit dem man die langfristige Lippensynchronität prüfen kann

Jede Videodatei beginnt mit einem Film-Header, der wichtige Parameter wie Auflösung, Anzahl der Frames oder Grösse der Buffer für Audio und Video enthält. Es schliessen sich die Frames an, in denen wiederum mehrere Video-Blöcke und ein Audio-Block enthalten sind.

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

In der bereitgestellte Datei `./include/avs.h` ist die Definition des Formates selbsterklärend enthalten.

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

4 Zu implementierende Funktionen

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 5).

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 implementierende Lösung sollen dementsprechend 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 (intra-frame) codiert sind. Üblicherweise werden beim Kodieren zeitliche Redundanzen entfernt, so daß Folgebilder voneinander (inter-frame) abhängig sind und daher in jedem Fall Fehlerbehandlungen vorzusehen sind.

4.1 Der Server

Der Server besteht aus einem Programm, das 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 versenden, um zu gewährleisten, dass diese zuverlässig übertragen werden (s.u.).

4.2 Der Client

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

`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, ist das mit einer Ausgabe zu protokollieren. 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 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 Übertragungsrate reagieren, mit der er Audio-/Video-Pakete versendet. Eine Lösung ohne Flusskontrollprotokoll nur durch ein 'geschicktes' Timing des Versendens 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 können nicht alle Pakete auf einmal versendet werden, da eine Begrenzung der beim Client gepufferten Bilder auf 80 eingehalten werden muss. Hier ist also ein Mittelweg zu finden.

4.3 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.

Für die Ausgabe von Audiodateien sind zwei Funktionen vorgesehen, mit denen sich ein Audio-Block in den Audio-Eingangsbuffer schreiben und die Anzahl der geschriebenen Audio-Blöcke abfragen lässt. Damit ist eine Synchronisation der Anwendung auf das Audio realisierbar.

Die zu verwendenden Funktionen sind im Einzelnen:

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

Diese Funktion initialisiert das Bildfenster und die dazugehörigen Ressourcen über SDL. Das Audiodevice wird über PortAudio initialisiert.
Als Parameter erhält diese Funktion einen Zeiger auf eine `movie_header`-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.

`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.

`int play_audio_block(char *data, int length)`

Mit dieser Funktion werden *length* Bytes der im Buffer *data* übergebenen Daten dekodiert (µLaw nach PCM) und in die internen Buffer des Audiodevice kopiert. Gelingt dies, wird 1 zurückgegeben, sonst 0. Werden die Buffer nicht schnell genug nachgefüllt, sind deutlich hörbares Knacken und ähnliche Störgeräusche wahrnehmbar.

`int get_completed_audio_blocks(void)`

Diese Funktion gibt die Anzahl der Audioblöcke zurück, die in den internen Puffer des Audiodevice geschrieben wurden. Der interne Puffer beträgt bei der Initialisierung mit *PortAudio*

2048 Bytes, so dass etwa drei Audio-Blöcke der hier verwendeten Länge von 640 Bytes vorgehalten werden. Mit diesem Rückgabewert ist ein Rückschluß auf die Anzahl abgespielter Audiosamples und damit auf die Laufzeit möglich.

5 Die Libnetbug

Da auf dem lokalen Netz nur mit wenigen Übertragungsfehlern zu rechnen ist, existiert eine Bibliothek, die diese künstlich erzeugt (Libnetbug), welche zum Server hinzugelinkt wird. Durch Verändern der Parameter im Skript `./server/setupnetbug.sh` und anschließendem Ausführen des Skriptes mittels:

```
source setupnetbug.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 `avs-server` ausgeführt wird.

6 Abnahmekriterien

Für eine erfolgreiche Abnahme ist das Verhalten des Zusammenspiels von Server und Client auch unter schlechten Bedingungen (d.h. bei hohen Paketverlustraten etc.) maßgebend. Abnahmekriterium für die Aufgabe ist daher, dass bei einer Paketverlustrate von 20% (eingestellt in der libnetbug) das Audio noch fehlerfrei und kontinuierlich zu hören ist. Bei einer Paketverlustrate von 0% muss auch das Video flüssig und nahezu fehlerfrei zu sehen sein (nahezu weil selbst im lokalen Netz ein vereinzeltes Paket verloren gehen kann). Eine Lippensynchronität muss über die gesamte Spielzeit und unabhängig von Übertragungsfehlern sichergestellt werden.

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.