Att koppla ihop grafiska applikationer över nätet med TCP och UDP

Vi kan nu skapa grafiska bilder i en applikation som kör på en dator, vårt mål när vi vill utveckla en nätbaserad applikation (tex ett spel) blir att alla användare får relevant information om systemets tillstånd och en konsistent, grafisk presentation av detta. Det är också viktigt att den grafiska presentationen är aktuell och synkroniserad med andra användares grafiska presentationer.

Ett problem som vi stöter på (och ska presentera en lösning på här) är att vi vill skriva applikationen som en *client-server*-lösning. Det betyder att det mest effektiva upplägget är att servern sparar *all* tillståndsinformation om systemet och skickar bara det som klienterna behöver för att representera tillståndet på ett korrekt sätt för sina respektive användare. Vidare skickar klienterna uppdateringsönskemål kontinuerligt till servern under den tid som applikationen är igång.

Vi skickar alltså aldrig grafik över nätet! Bara styrdata. Klienterna har hand om all grafik. Men en klient *får* absolut inte ha hand om tillståndsdata. En klient kan ju gå ner och då ska inte övriga klienter drabbas. Det betyder att det som relaterar till grafik måste separeras från det som handlar om tillstånd. De ska ju hanteras på två olika maskiner (eller i alla fall två olika delar av applikationen: tilståndet hanteras av servern, den grafiska presentationen hanteras av klienten.)

Hur gör vi då praktiskt?

I ett tidigare program med kvadrater som rörde sig över skärmen delades programmet upp i tre delar: `square.h`/`square.c`, `timer.h`/`timer.c` och `mousecontrolled_squares.c` där `mousecontrolled_squares.c` var huvudprogrammet och `square` och `timer` var två stödjande bibliotek. Programmet var baserat på *OpenGL* och två funktioner användes för att hantera positionen av en kvadrat och den grafiska representationen, de hette `square_move()` och `square_show()`. Ett uttryck för att servern ska hantera tillstånd blir att det är servern som ska utföra uppgifterna i funktionen `square_move()` och sedan tala om för alla anslutna klienter att en kvadrats position har förändrats. Då ska klienterna, på egen hand, använda sig att något som liknar funktionen `square_show()` för att ge den grafiska representationen av systemet till sina respektive användare.

Vi ska nu studera hur man kan distribuera ett grafiskt program över nätet. När jag började programmera kring det här för att kunna presentera något bra tänkte jag mig att kvadratprogrammet skulle ha en server som håller reda på alla kvadraters positioner och att ett antal klienter skulle ansluta till servern och att varje enskild klient skulle äga en kvadrat och klicka på sin fönsteryta och då ändra riktningen på sin kvadrat. Alla klienter skulle se alla kvadrater samtidigt.

Det visade sig dock bli ett väldigt stort program som tog för mycket tid att skriva och det är då bättre att ni får bli insatta i de svårigheter som man löser och hur man arbetar för att lösa dem och själva tillämpa det arbetssättet på era problem snarare än att vänta på ett exempel som jag ska skriva som tar lång tid. Jag har nu kommit så långt att jag kan beskriva ett arbetssätt och veta att detta arbetssätt kommer att fungera.

En noggrannare beskrivning av en grafisk client-serverapplikation:

Vi börjar med att beskriva det allmänna förloppet då klient och server kör. Vi väver in konkretiseringar för fallet med kvadratprogrammet. Det första som måste göras är att klienterna gör sig kända för servern, man startar först servern som väntar på anslutningar och varje klient kan sedan starta. Klientprogrammet kallas *squareclient* och servern kallas *squareserver*. Det som etableras först i dialogen mellan klient och server är starttillståndet. I fallet med kvadraterna är det endast vilken färg som kvadraten ska ha som den klienten ska äga. Andra saker etableras automatiskt internt inne i serverprogrammet som till exempel vilken ip-adress klienten har. Nästa steg är att servern utifrån transaktioner med klienter upprättar interna datastrukturer som ska beskriva tillståndet i systemet. Dit har jag inte hunnit. Det som ska göras är att upprätta något i stil med

```
struct client_square
{
  int x;
  int y;
  int color;
};
```

och se till att när en klient ansluter så läggs en kvadrat till en array.

Sedan går som sagt servern in i en loop där den löpande skickar information till alla klienter om systemets tillstånd så att de var och en kan skapa grafiska representationer av detta. Konkret för kvadratprogrammet betyder det att arrayen innehåller alla kvadraters data gå igenom löpande av servern och positionsdata skickas till samtliga klienter.

Klienterna tar också löpande emot styrsignaler från sina respektive användare och skickar in dessa styrsignaler till servern som återigen uppdaterar systemets tillstånd och återigen informerar klienterna om vad som hänt etc. etc. I fallet med kvadrater lyssnar varje klient efter ett klick och detta klick indikerar en ny destination för just den klienten. Det innebär att en av de kvadrater som rör sig kommer att byta riktning. Serverns uppgift är att se till att detta sker och att alla får information om detta på det konsistenta synkroniserade sättet som vi bekrivit ovan.

Både klient och server har någon form av loop, hos klienten kallas det event-loop och på serversidan är det mycket troligt att det finns loopar på flera nivåer, troligtvis minst en tråd per klient och dels något slags huvudloop som sköter systemet på ett översiktligt sätt, till exempel hantering av att ansluta nya klienter eller koppla bort klienter som inte längre ska arbeta mot servern. I ett spel kan det hända att nya spelare kommer till efter hand och att spelare som spelar förlorar eller “dör” som man säger i spelsammanhang. När det gäller kvadratprogrammet tänkte jag mig att en eller två trådar per klient skulle startas, men jag hann som sagt inte så långt. Man kan se i källkoden vart detta skulle börja läggas in. Det finns en kommentar där som lyder: *The stuff below should go into a client-specific thread*.

TCP och UDP

När det gäller kommunikation över nätet finns det flera aspekter att beakta. För det första ska inte all kommunikation ske på samma sätt. Vi kommer att studera två sätt, *connection-oriented* (TCP) och *connection-less* (UDP). Både TCP och UDP är två protokoll hörandes till transportlagret i OSI-modellen, TCP står för *Transmission Control Protocol* och UDP står för *User Datagram Protocol*. Båda protokollen hanteras programmeringsmässigt av sockets och dess sockets kan således vara av TCP-typ eller UDP-typ. TCP-sockets måste ha en körande anslutning (*connection*) för att kunna användas och UPD-sockets har ingen anslutning. Vi har tidigare sett en introduktion av TCP och vi tittar nu på SDL_net:s motsvarighet och vi ska även se hur SDL_net hanterar UDP-sockets.

Jag baserar mycket av denna framställning på följande tutorial:
http://gpwiki.org/index.php/SDL:Tutorial:Using_SDL_net

Vi har tidigare även programmerat med TCP-sockets i operativsystemkursen och då är gången som bekant så här:

Klient:

Skapa en socket. (`socket`)
Gör connect mot en server. (`connect`)
Läs/skriv från/till socketen. (`send`/`recv`)
Stäng socketen. (`close`)

Server:

Skapa en socket med anropet `socket`.
Bind den till en specifik port med `bind`.
Börja lyssna efter anslutningar med `listen`.
Acceptera aslutningar med `accept`. (Anropet `accept` levererar en ny deskriptor att läsa/skriva med, detta är för att möjliggöra flera parallella anslutningar.)
Läs/skriv från/till socketen. (`send`/`recv`)
Stäng socketen. (`close`)

Detta har ni gjort i *C*, i operativsystemskursen, det var TCP-sockets ni använde. Alla dessa funktioner har motsvarigheter i SDL_net och de heter nästan likadant, namnen är ofta `SDLNet_TCP_Recv()` till exempel för att läsa från en socket.

Vi ska se nu på hur vi gör UPD i SDL_net. Som sagt tidigare så finns ingen pågående anslutning mellan två noder som kommunicerar med UDP, vi skickar försändelser (alltså några bytes data) utan garantier om att de kommer fram. Gången är följande:

1. Skapa en UDP-socket. (Görs en gång, representeras av en deskriptor.)
2. Forma ett UDP-paket, här läggs data in.
3. Sätt adress och port på paketet.
4. Skicka paketet över socketen.

Det finns alltså ingen `close`-funktion här, vi skickar ett paket och sedan vet vi ingenting mer om det. Här följer lite exempelkod från det stora (ofärdiga) programmet som jag skrev:

```
1   p->address.host = client_address.host;
2   p->address.port = client_address.port;
3   p->len=12;
4
5   for(i=0;i<10;i++)
6   {
7     rx = rand()%10; rand()%10 + rand()%10 - 15;
8     ry = rand()%10; rand()%10 + rand()%10 - 15;
9
10    sprintf((char *)(p->data),"%d %d %d", csm_click_at_pdu.x+rx,
11                                          csm_click_at_pdu.y+ry, 1);
12    SDLNet_UDP_Send(udp_sd,-1,p);
13    SDL_Delay(50);
14  }
```

Innan denna kodsnutt har ett UDP-paket skapats. Pekaren `p` pekar på det. På rad 1 och 2 ser vi hur adressen och porten läggs på paketet, sedan upprättas de data som ska finnas i paketet, här skickar vi 10 paket med olika innehåll, vi tar en punkt, (`x`, `y`), och lägger på slumptal på dess koordinater, skriver in det i paketet med `sprintf()` och skickar det sedan med `SDLNet_UDP_Send()`. Vi har tidigare fått en upplysning från klienten om var anändaren klickat någonstans, det finns lagrat i (`x`, `y`). Det servern gör är att skicka tillbaka dessa koordinater, lite modifierade, till klienten och klienten ritar sedan ut kvadrater på dessa positioner, jag valde att göra så här för att först få igång en enkel kommunikation över UDP. Planen var ju senare att detta skulle läggas i en tråd och så skulle kvadraten flyttas successivt över skärmen etc.

På klientsidan har vi följande mjukvaruarkitektur:

Eventloop
{
    Hantera händelser och skicka till servern
    Ta emot UDP-paket från servern som styr hur grafiken ska se ut
    SDL_Delay
}

En viktig skillnad mellan mottagningsfunktionerna `SDLNet_TCP_Recv()` och `SDLNet_UDP_Recv()` är att TCP-varianten blockerar, det vill säga om man läser på en TCP-socket i SDL så väntar körningen där tills det kommer något. Det är *inte* fallet med UDP, om det finns något att läsa så läses detta annars så kör programmet vidare. Det betyder att läsningen på klientsidan är klar direkt om det inte finns några paket. Om det inte heller finns några händelser att hantera, som musklick eller tangenttryckningar så bli ovanstående loop helt tom. Därför måste man ha en Delay i den (jag hade ingen Delay i den först och då hängde sig hela mitt system.)

Endianness

Olika datorer har olika upplägg på sin interna hårdvarumässiga representation av det som finns i minnet. Det finns två huvudsakliga varianter: *big-endian* och *little-endian*. Ni behöver absolut läsa er till en förståelse av detta, men vi ska ge några enkla beskrivningar av vad det är. Som sagt handlar

det om hur en maskin rent fysiskt hårdvarumässigt representerar data. Det finns ju olika storlekar av data, det kan lagras i enskilda *bitar*, *nibbles* (4 bitar), *bytes* (8 bitar), *words* (16 bitar) *long word* (32 bitar) eller *long long word* (64 bitar). I nätverkssammanhang är det relevant att studera i vilken ordning de ingående bytesen som bygger upp word och long word förekommer. På en big-endian-maskin kommer det som är mest signifikant först och det som är minst signifikant kommer sist. Som exempel betraktar vi talet `0x1A0FAFAA` som binärt ges av 00011010 00001111 10101111 10101010. I en big-endian-maskin lagras det talet alltså så här:

Bitar	00011010	00001111	10101111	10101010
Adresser	23	22	21	20

alltså som man är van att tänka sig det. Nu är adressen kanske inte exakt 20-23, det var bara ett exempel för illustrera principen: De fyra bytesen har stigande adresser med stigande signifikans. På en litte-endian-maskin är det dock tvärt om. Om samma tal skulle lagras på en little-endian maskin skulle dess fysiska layout se ut så här:

Bitar	10101010	10101111	00001111	00011010
Adresser	23	22	21	20

alltså helt omkastat. De flesta persondatorer är little-endian. Det hänger på processorn.

Här kommer kruxet: När vi ska sända saker från en maskin till en maskin med en annan endianness, så måste vi omvandla. Problemet då är ju att vi inte vet om vår klient är little eller bigendian. Hur göra? Lösningen är att så fort vi skickar något utanför datorn så ser vi till att det är big-endian och när vi tar emot någonting så utgår från att det är big-endian. Därför har big-endian formatet även kallats *Network byte order*. Vi skiljer då på Network byte order som är big-endian, och Host order som kan vara antingen big-endian eller little-endian beroende på vilken maskin vi sitter vid. POSIX-plattformen erbjuder omvandlingsfunktionerna `ntohl()` och `htonl()` som man anropar innan man tar emot saker från nätet respektive skickar saker till nätet. Beroende på plattformen så vänder dessa funktioner på de ingående argumentens bytes eller inte. I SDL_net finns funktionerna

```
Uint32 SDLNet_Read32(void *area);
void SDLNet_Write32(Uint32 value, void *area);
```

som alltså omvandlar mellan Host byte order och Network byte order. Det finns också motsvarande funktioner för word, alltså ord med 16 bitar, de heter i POSIX `ntohs()` och `htons()` och i SDL heter de samma som ovan fast med `16` istället för `32` i namnen. Ett exempel på användningen kommer här, från exempelprogrammet:

```
remote_ip = SDLNet_Read32(&remoteIP->host);
ch0 = remote_ip>>24;  ch1 = (remote_ip & 0x00ffffff)>>16;
ch2 = (remote_ip & 0x0000ffff)>>8; ch3 = remote_ip & 0x000000ff;

sprintf(remote_ip_str,"%d.%d.%d.%d",(int)ch0,(int)ch1,(int)ch2, (int)ch3);
printf("Remote host IP in string form: %s\n", remote_ip_str);
```

Detta kodavsnitt föregås av kod där en klient ansluter till servern, då får `remoteIP` sitt värde och därifrån kan man gräva med bitvisa operatorer på detta sätt för att få fram delar av en IP-adress. Det

är troligtvis väldigt nyttigt att göra dessa operationer för att får en förståelse av vad som sker, till exempel så nollställer satsen `0x0000ffff` alla bitarna 16-31, detta skickas sedan till ett så kallat "högerskift, >>" och bildar `ch2 = (remote_ip & 0x0000ffff)>>8` som alltså skiftar ner innehållet 8 bitar, det sammantagna blir att byten på plats 1 hamnar i `ch2`.

En av anledningarna till att Network byte order har de mest signifikanta bitarna för är att det går bättre vid routing. En IP-adress är ju (klassiskt sett) fyra bytes och om man läser början av en IP-adress så hittar vi de mest signifikanta bitarna där, IP-protokollet är då utformat (med Netmask och sådant) att så fort man känner början så kan man utföra routing. Principen är densamma då man lyfter en telefonlur för att ringa, då man slagit riktnumret och börjar slå de andra siffrorna i numret så är redan maskineriet igång med att koppla samtalet till rätt riktnummerområde.

Applikationsprotokoll och Protocol Data Units

När vi skapar en applikation med en klient och en server så måste ju de kommunicera på ett strukturerat och synkroniserat sätt. De måste vara överens om vilken information som ska skickas och tas emot när och vad den betyder. Översiktligt kallas detta "applikationsprotokoll" och det kan förstås liknas vid ett språk som utvecklas för just er applikation. Det är då viktigt att era delprogram kan tala detta språk. De minsta delarna i ett språk kallas "ord" och ord sätts samman till meningar. Då vi talar om applikationsprotokoll är orden så kallade *protocol data units* och det är en term som finns i flera sammanhang. Vi använder den termen här också.

I en strävan att skriva ett program som var hyfsat strukturerat deklarerade jag ett par typer i en header-fil som heter `protocol.h`:

```
//Client requesting entry. Sent via TCP.
struct csm_request_entry
{
  int requested_color;
};

//Client notifying server of a click. Sent via TCP.
struct csm_click_at
{
  int x, y;
};


//Server granting entry to a user. Sent via TCP. (-1 indicates failure.)
struct scm_grant_entry
{
  int success;
};

//Server requesting client to draw a square at x, y. Sent via UDP.
struct scm_draw_square_at
{
  int x, y;
  int square_color;
};
```

I dessa structar har jag valt att lägga till prefixet `csm` för meddelanden som går från klienten till

server, alltså *client-server-message* – `csm`, och prefixet `scm` för meddelanden som går i andra riktningen, alltså *server-client-message* – `scm`. I programmen har jag sedan lagt deklarationer protocol data units som ser ut så här:

```
struct csm_request_entry csm_request_entry_pdu;
struct csm_click_at csm_click_at_pdu;
struct scm_grant_entry scm_grant_entry_pdu;
struct scm_draw_square_at scm_draw_square_at_pdu;
```

och här har vi alltså deklarationer av fyra stycken protocol data units som kan användas vid kommunikation mellan server och klient. Man kan dock inte bara skicka dessa structar helt rått över nätet, man behöver omvandla dem först, ett bra medium att omvandla till är någon form av märkspråk som XML eller liknande. När man alltså skickar ett musklick från klienten till servern är det alltså bra att skicka det i textform, så här:

```
<click><x>150</x><y>266</y></click>
```

Detta skulle alltså vara en representation av en `struct csm_click_at` pdu och det skulle troligtvis behövas encode och decode funktioner för att omvandla mellan textform och structform på pduerna. Det här är produktionsfrågor som vi får se hur varje grupp ställer sig till. Det är inte komplicerat men det är en del att tänka på. Jag gör inte detta, jag skickar hela pdu:n direkt på socketen, det är inte att rekommendera av skäl som jag inte känner till så väl... Anders? På serversidan skickar jag koordinater som text, till exempel som vi såg innan/ovan:

```
10    sprintf((char *)(p->data),"%d %d %d", csm_click_at_pdu.x+rx,
11                                          csm_click_at_pdu.y+ry, 1);
12    SDLNet_UDP_Send(udp_sd,-1,p);
```

När man skapar textsträngar så är `sprintf()` en gudagåva, eller en *Ritchie*-gåva kanske man skulle kunna säga. Den formaterar enkelt strängar som vanligt med formatsträng som `printf()`, men målet är inte `stdout` utan resultatet läggs i en sträng istället. Ja, vi har ju sett den flera gånger.

Ja, då är det bara att kolla upp `squareserver.c`, `squareclient.c` och `protocol.h`. Klientprogrammet är skrivet i *Windows*-miljö och tyvärr lyckades jag inte få igång projekthanteringen i Code Blocks, men ni måste absolut ordna det så att projektet har uppdelningar i `.c` och `.h`-filer. Programmen finns på Bilda. Där finns också de fyra grundprogrammen, `udpc.c`, `upds.c`, `tcpc.c` och `tcps.c`, alltså klienter och servrar för UDP och TCP från den tutorial som jag baserade mycket av det här på. Det var:

> http://gpwiki.org/index.php/SDL:Tutorial:Using_SDL_net.

Två andra användbara referenser är:

> http://www.ibm.com/developerworks/aix/library/au-endianc/index.html?ca=drs-
>
> http://en.wikipedia.org/wiki/Endianness