Umeå Universitet
Institutionen för datavetenskap

HT 2015

Chattserver

Innehåll

1 Introduktion

Datornätverk används för att överföra information mellan olika datorer. Den överförda informationen kan i stort sett bestå av vad som helst som kan uttryckas med ettor och nollor. En typ av information är textmeddelanden som är av typen "från många till många" och en typ av applikation som hanterar sådana meddelanden brukar kallas en chatt. En chatt består i regel av en serverdel och en klientdel. Serverdelen tar emot ett meddelande från en klient och vidarebefordrar det till alla klienter som är anslutna till chattservern. Klientdelen skickar textmeddelanden, inskrivna av användaren, och presenterar den fortgående konversationen för användaren. Ibland används även en namnserver för att hjälpa klienter att hitta servrar att ansluta till.

När ett system blir komplext inför man olika abstraktioner. Ett abstrakt objekt fångar vissa intressanta aspekter av systemet, har ett eller flera gränssnitt som kan användas av andra objekt, samt gömmer detaljer om hur objektet är implementerat. Det finns två stora fördelar med att låta ett system bestå av flera abstrakta objekt. För det första delar det upp problemet med att bygga ett system i mindre och mer hanterbara komponenter. För det andra ger det en mer modulär design, dvs det blir mycket lättare att byta ut och lägga till nya objekt.

Specifikationen för kommunikationen i ett nätverkssystem kallas protokoll. Ett protokoll definierar vilka medelanden som kan skickas mellan olika komponenter i systemet, de medelanden som en komponent som implementerar protokollet förstår kan benämnas komponentens gränssnitt. Implementationer av protokoll är ofta standardiserade till att följa vissa API:er t.ex. BSD sockets vilka definierar ytterligare ett gränssnitt, som används av program som ska använda protokollet. Detta interna API-gränssnitt kan också kallas servicegränssnitt. Protokoll som komunicerar över nätverket använder paketerar datan i paket, PDU (Protocol Data Unit). Det räcker inte att bara skicka över datat utan oftast måste ytterligare information skickas med som beskriver hur datat ska behandlas. Denna ytterligare information stoppas oftast i en header, som sätts före datat i paketet. Datat kallas body eller payload. Denna inkapsling upprepas för varje använt protokoll i ett nätverkssystem.

I den här laborationen ska ni implementera en chattserver och en chattklient. Målsättningen är att er server och klient ska kunna kommunicera med andra gruppers servrar och klienter samt den namnserver vi tillhandahåller. Chattklienten ska ha ett grafiskt användargränssnitt

2 Syfte

Det övergripande syftet med laborationen är att öka förståelsen för datakommunikation och datornät. Laborationen har följande delsyften:

- Träna socketprogrammering.
- Förståelse för protokoll. Varför och hur de används, samt implementering av protokoll.
- Förståelse för abstraktion i komplexa system.
- Trådprogrammering.
- Dialog med andra grupper.

2.1 Föväntade studieresultat

De förväntade studieresultat som examineras med laborationen är:

2.1.1 5DV166 – Färdighet och förmåga

- Tillämpa socket- och trådprogrammering. (FSR 6)
- Utforma en design av och implementera en tillämpning som är relaterad till lagerkonceptet. (FSR 7)
- Genomföra tester, däribland vissa stresstester, av den egna implementationen. (FSR 8)
- Använda specifika protokoll som exempelvis TCP, IP, HTTP, ARP och CDMA. (FSR 9)

2.1.2 5DV167 – Färdighet och förmåga

- Tillämpa socket- och trådprogrammering. (FSR 6)
- Utifrån en given design implementera en tillämpning som är relaterad till lagerkonceptet. (FSR 7)
- Utföra viss testning av den egna implementationen. (FSR 8)
- Använda specifika protokoll som exempelvis TCP, IP, HTTP, ARP och CDMA. (FSR 9)

3 Beskrivning av systemet

Chattjänsten består av tre huvuddelar: en klientdel, en serverdel och en namnserver. Er huvuduppgift i laborationen är att implementera de två förstnämnda delarna i detta system.

3.1 Chattserver

När chattservern startar ska den registrera sig hos namnservern så att klienter kan hämta information om chattservern från namnservern.

Chattservern har två huvuduppgifter: dels att hålla koll på vilka chattklienter som är uppkopplade mot chattservern inklusive deras nicknames och adresser och dels att distribuera meddelanden från klienter till samtliga anslutna klienter.

Servern ska för varje klient hålla reda på nicknamet och TCP-anslutningen för klienten. Under själva körningen väntar chattservern på att någon chattklient ska skicka ett meddelande. Beroende på typen av meddelande så ska servern sedan skicka information om detta meddelande till alla anslutna klienter.

3.2 Klient

När chattklienten startas så ska den hämta en lista med chattservrar inklusive deras IP-adresser och portnummer från en namnserver. Sedan ska den ansluta till en av dessa chattservrar genom att adress och port för servern anges. När den är ansluten så ska den kunna skicka och ta meddelanden, byta nickname och lämna chatten. Om en klient försöker ansluta med ett upptaget nickname så ska servern skicka ett meddelande som förklarar detta och låta klienten försöka ansluta igen. På samma sätt ska ett meddelande skickas om en klient försöker byta till ett upptaget nickname. För detaljer kring servermeddelanden se Avsnitt 4.4.

Chattklienten ska ha ett grafiskt användargränssnitt som ska uppfylla följande krav:

- Adress och port till namnserver ska kunna anges och listan som tas emot ska kunna visas.
- Adress och port till den chattserver som klienten ska ansluta till ska kunna anges.
- När klienten är ansluten till chatten så ska meddelanden visas och det ska finnas möjlighet att skriva och skicka egna meddelanden.
- Det ska gå att byta nickname genom användargränssnittet.
- En lista med anslutna klienter ska visas, och denna lista ska uppdateras dynamiskt när andra klienter ansluter till eller lämnar chatten.

För de som inte vill skapa ett eget så finns ett färdigt användargränssnitt att använda.

3.3 Namnserver

Trots att implementationen av denna del inte ingår i laborationen kommer viss information ändå att ges angående namnservern. Detta för att underlätta kommunikationen med densamma.

Alla chattservrar ska registrera sig hos en namnserver för att chattklienter ska kunna hitta dem. Sedan ska chattservrarna kontinuerligt meddela att de fortfarande är aktiva (ett s.k. *heart-beat*). Namnservern har en lista med aktiva chattservrar som uppdateras kontinuerligt. Chattklienter kan hämta listan med aktiva chattservrar från namnservern.

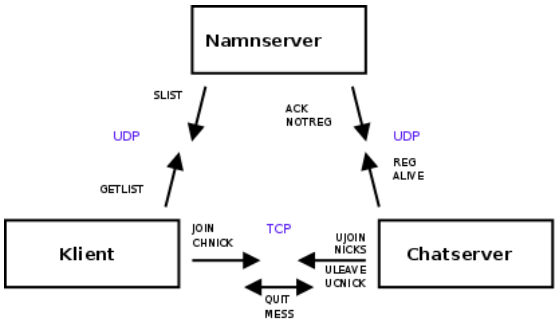
En central namnserver finns uppsatt på `itchy.cs.umu.se` på port `1337`. Denna server rekommenderas ni använda så att andra studenter kan ansluta till er server.

Den centrala namnservern har även ett webbinterface vilket visar den aktuella serverlistan. Det nås på `http://itchy.cs.umu.se:8080/`.

För ostörd testkörning och debugging finns binären till en namnserver (Java 8).

4 Protokoll

Kommunikationen i systemet består av tre delar: klient-server, server-namnserver och klient-namnserver. En översikt över kommunikationen visas i Figur 1. En kort förklaring för varje PDU finns i Tabell 1. För en detaljerad beskrivning av alla PDU:er, se Appendix D.



Figur 1: Översikt över protokollen mellan klient, chattserver och namnserver.

4.1 Server-namnserver

All kommunikation mellan chattserver och namnserver sker via UDP. När en server ska registrera sig hos en namnserver så skickar den först en `REG`-PDU. Som svar får den sedan en `ACK`-PDU med ett unikt ID. Därefter ska servern regelbundet skicka en `ALIVE`-PDU med samma ID. Om namnservern inte får

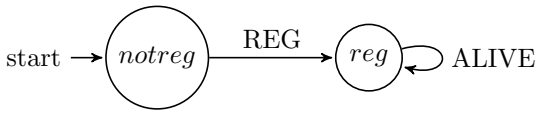
Namn	Syfte
Chattserver-namnserver	
`REG`	Skickas från en chattserver när den vill ansluta till en namnserver.
`ALIVE`	Skickas från chattserver till namnserver för att fortsätta vara registrerad.
`ACK`	Skickas som svar på `REG` eller `ALIVE`.
`NOTREG`	Skickas som svar på `ALIVE` om chattservern inte är registrerad.
Klient-namnserver	
`GETLIST`	Skickas från klient till namnserver för att få en lista med chattservrar.
`SLIST`	Svar på `GETLIST`, innehåller lista med chattservrar.
Klient-chattserver	
`JOIN`	Skickas från klient till chattserver för att ansluta till chatten.
`NICKS`	Skickas som svar på `JOIN`, innehåller lista med anslutna klienter.
`QUIT`	Skickas mellan klient och chattserver för att indikera stängd anslutning.
`MESS`	Chattmeddelande mellan klient och chattserver.
`UJOIN`	Skickas från chattserver till klienter för att indikera att användare anslutit till chatten.
`ULEAVE`	Skickas från chattserver till klienter för att indikera att användare lämnat chatten.
`CHNICK`	Skickas från klient till chattserver för att byta nickname.
`UCNICK`	Skickas från chattserver till klienter för att indikera att en användare bytt nickname.

Tabell 1: En kort förklaring av alla PDU:er och deras syfte.

någon `ALIVE`-PDU på 20 sekunder så avregistreras chattservern och ytterligare `ALIVE` kommer besvaras med `NOTREG`. Då ska chattservern registrera om sig med en ny `REG`-PDU. För att undvika att avregistreras ifall ett UPD-datgram försvinner, och då det kan ta lite tid för meddelandena att komma fram så kan ni skicka en `ALIVE`-PDU var 8:e sekund. För ett tillståndsdiagram över chattserver-namnserver-kommunikationen se Figur 2.

4.2 Klient-namnserver

För att begära en serverlista så skickar klienter en `GETLIST`-PDU till en namnserver via UDP. Som svar kommer en eller flera `SLIST`-PDU:er, också detta via UDP. Eftersom ett UDP-datagram har en max-storlek på 65507 bytes så kan



Figur 2: Tillståndsdiagram för kommunikationen mellan chattserver och namnserver. Alla giltiga PDU:er och deras respektive övergångar visas. Skickas en $ALIVE$-PDU när anslutningen är i tillståndet *notreg* så svarar namnservern med en $NOTREG$-PDU. Alla andra PDU:er ignoreras. Om ingen $ALIVE$-PDU skickas inom 20 sekunder så försätts kommunikationen i tillståndet *notreg*.

det skickas flera svar ifall servrarna inte ryms i ett datagram. Därför innehåller varje PDU ett sekvensnummer som börjar om på 0 för varje `GETLIST`-PDU som skickats. I klienten bör detta hanteras så att när en ny serverlista begärs så töms den nuvarande listan, och servrarna i de mottagna PDU:erna läggs till i listan eftersom de dyker upp, så länge ingen PDU med det sekvensnumret tagits emot.

4.3 Klient-chattserver

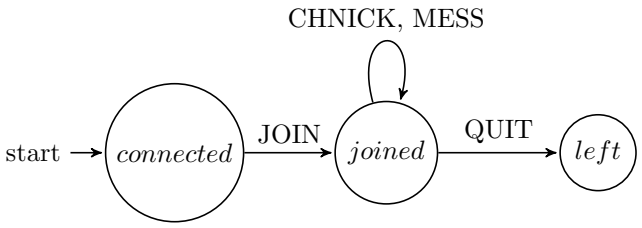
Kommunikation mellan klient och chattserver sker via en TCP-anslutning. När anslutningen öppnats så skickar klienten en `JOIN`-PDU och får som svar en `NICKS`-PDU. Därefter kan klienten skicka `MESS` för att skicka meddelanden och `CHNICK` för att byta nickname. Dessutom kan anslutningen avslutas genom att skicka en `QUIT`-PDU. Alla andra PDU:er är ogiltiga och ska besvaras med en `QUIT`-PDU följt av att chattservern stänger anslutningen. För ett tillståndsdiagram över anslutningen mellan klient och chattserver ur serverns perspektiv, se Figur 3. Om en klient försöker ansluta till en chatt med ett nickname som redan finns så ska detta hanteras genom att klienten ges ett nickname som är en modifikation av det önskade, genom att t.ex. lägga på en siffra på slutet.

När en klient har anslutit till chatten så kan servern skicka `MESS` om någon annan klient skickat ett meddelande, `UJOIN` eller `ULEAVE` om en annan klient anslutit till eller lämnat chatten, `UCNICK` om en klient ändrat nickname samt `QUIT` om servern avslutas.

När en klient skickar ett meddelande till en chattserver så ska meddelandet inte ha någon tidsstämpel eller någon avsändare. Detta läggs till i chattservern så att klienter inte ska kunna ange ett annat namn eller en annan tid än den korrekta. När servern tar emot meddelanden så är det viktigt att de skickas till alla anslutna klienter i samma ordning.

4.4 Felhantering

Om en klient eller server tar emot en PDU med felaktig padding (bytes som ska ha värdet 0 har ett annat värde) så ska anslutningen som PDU:n skickades över stängas. Om en chattserver tar emot en felaktigt paddad PDU av en klient så ska ett meddelande skickas till klienten som informerar om felet innan anslutningen



Figur 3: Tillståndsdiagram för kommunikationen mellan klient och chattserver. Alla giltiga PDU:er och deras respektive övergångar visas. Alla ogiltiga PDU:er resulterar i att anslutningen övergår i tillståndet $left$ och anslutningen stängs.

stängs. Ni behöver inte verifiera att strängarna som skickas är korrekta UTF-8-strängar, utan det som skickas kan visas som det är. Ni uppmuntras dock att kontrollera detta, och hantera felet på valfritt sätt. Om en PDU börjar med en ogiltig OP-code så ska också då anslutningen stängas efter att ett felmeddelande skickats.

Servern skickar också ut egna meddelanden. Detta händer när:

- Någon skickar korrupta meddelanden till servern och blir utkastad.
- Servern avslutas.

Servern skickar då ut meddelanden som berättar detta till samtliga anslutna klienter. En konvention som gäller är att meddelanden utan nickname och med fältet NICKLEN satt till 0 är systemmeddelanden från chattservern. Det är då lämpligt att chattklienten presenterar detta meddelande på ett speciellt sätt, tex med speciella tecken före/efter texten, för att skilja servermeddelande från själva konversationen. Felaktiga meddelanden ska slängas och uppkopplingen till denna klient bör stängas. Vidare bör som tidigare nämnts ett systemmeddelande om detta skickas av chattservern.

5 Övriga krav

Alla delar av systemet ska kunna hantera fragmentering av meddelanden över TCP-anslutningar. Det innebär att PDU:er kan delas upp i flera datagram, och när de anländer hos mottagaren så läses inte hela PDU:n från strömmen. Implementationen ska då upptäcka detta och vänta in resten av PDU:n innan den hanteras.

Omvänt så kan också flera PDU:er skickas i ett enda datagram. Det kan resultera i att när programmet läser från socketen så läser det in flera PDU:er samtidigt, men hanterar bara den första. Detta ska också upptäckas och hanteras så att inga PDU:er försvinner.

Inga timeouts ska behövas för att läsa från öppna TCP-anslutningar. Det behövs därför en tråd per sådan anslutning. Det är dock OK att ha timeouts för UDP eller för att öppna TCP-anslutningar. Det är också OK att ha en timeout när klienter skickar `JOIN`-meddelanden till chattservrar, och stänga anslutningen om inget sådant meddelande skickas inom några sekunder från att anslutningen öppnades.

6 Redovisning

- Uppgiften ska lösas i grupper om två personer. Studenterna skapar själva dessa grupper. Båda studenterna ska gå samma kurs (5DV166 eller 5DV167). Kontakta någon på kursen om ni inte hittar någon att jobba med.
- Grupperna ska lämna in en gemensam lösning.
- Lösningen ska innehålla implementationen av en chattserver och en klient.
- På kursen 5DV166 så ska klienten implementeras i Java och servern i C eller C++. På kursen 5DV167 så ska både servern och klienten implementeras i Java.
- Inlämningen ska också innehålla en fullständig rapport. Bland annat ska följande ingå:
  - Ett försättsblad med:
    - Kursnamn
    - Kurskod
    - Fullständingt namn
    - CS-användarnamn
    - Handledares namn (alla på kursen)
    - Datum
  - En användarhandledning som förklarar hur programmen kompileras och körs, och hur de används när de startats. Använd gärna bilder på grafiska användargränssnitt.
  - För implementationer i C ska det finnas en make-fil. För implementationer i Java så ska det om de använder andra externa verktyg än JUnit använda något bygg-script, t.ex. Maven eller Ant, så att de externa verktygen inte behöver hämtas eller läggas till i projektet manuellt. Detta ska också beskrivas i rapporten.
  - En systembeskrivning.
    - Beskrivningen ska ha en lämplig nivå. Till exempel behöver inte enskilda metoder beskrivas. Beskrivningen ska ge en överblick över lösningen och förklara mer komplicerade delar.
    - Tänk på dispositionen, se till att alla delar sätts i ett sammanhang innan de förklaras.
    - En utförlig beskrivning av trådlösningarna. Använd gärna figurer som visar vilka trådar som skapar vilka, och hur information flyttas mellan trådarna. För 5DV166 är det också viktigt att redovisa trådsäkerhet.

    - Eventuella intressanta algoritmer i lösningen.
    - Om vissa beteenden inte är specificerade i specifikationen för uppgiften så ska detta beskrivas inklusive era val kring hantering.
  - Lösningens begränsningar: vad hanteras inte, vad är odefinierat?
  - Ett testprotokoll: hur vet ni att er lösning fungerar? För ett exempel se Appendix C.
  - Ett diskussionsavsnitt som diskuterar uppgiften. Några förslag på diskussionsämnen:
    - Vad är bra och dåligt med lösningen?
    - Hur det har gått att lära sig nya programmeringskoncept (sockets, trådar etc.)?
    - Vad har varit svårt?
    - Hur har utformningen av uppgiften fungerat?
    - Hur har det gått att arbeta med given kod?
    - Finns det för- och nackdelar med utformningen av protokollet?

- Rapport i `.pdf`-format och kod i ett `.zip`-arkiv lämnas in i Cambro.
- Implementationen ska också demonstreras för handledare. Syftet med denna demonstration är att demonstrera att implementationen är korrekt. Ett eget testprotokoll ska förberedas (för ett exempel se Appendix C). Exempel på delar som ska demonstrerar är:
  - Alla protokolldataenheter
  - Fungerande chatt
  - Hantering av checksumma
  - 5DV166 ska också demonstrera felhantering, t.ex:
    - Fel PDU skickas
    - Felaktig checksumma och padding
    - Illasinnade klienter

A Tips

Alla Javas heltalstyper (byte, short, int, long) är signed, d.v.s. de kan ha negativa värden. Det kan leda till buggar när de castas till datatyper som representeras med fler bitar. Kolla upp hur heltalen representeras (two's complement), prioritet för casting jämfört med andra operatorer, samt hur sign extension fungerar. Se sedan till att skriva tester som kontrollerar att denna hantering fungerar (använd nicknames med större längd än 255 bytes t.ex.).

Använd enhetstester för alla PDU:er. Skriv tester som både kontrollerar att de PDU:er som skapas har rätt format och padding, och tester som kontrollerar att om en PDU är felaktig (har felaktig padding, checksumma etc.) så upptäcks detta.

Bygg in validering i varje enskild PDU så att det är enkelt att kontrollera ifall en PDU är korrekt uppbyggd.

Eftersom specifikationen är väldigt detaljerad så lämpar det sig bra att använda sig av s.k. *test-driven development* (TDD). Det innebär att man först skriver ett test som kontrollerar att programmet fungerar som det ska, och sedan implementerar man det som testas. Sedan ser man till att alla tester lyckas innan nästa test skrivs. Läs gärna mer om TDD som är en väldigt kraftfull utvecklingsmetod.

Försök också skriva automatiska integrationstester. Det innebär att man bygger en automatisk testsvit som t.ex. startar en server och en klient och ser till att klienten kan ansluta till servern. Fördelen med sådana tester är att det går mycket snabbare att kontrollera att allt fungerar än om det måste göras manuellt efter varje liten ändring. Det gör också att man inte riskerar att glömma att köra något test.

Se till att tidigt i designarbetet fundera kring hur en s.k. *total ordning* kan implementeras. Total ordning innebär att alla händelser har samma ordning för alla delar av systemet. Om två personer A och B i chatten skickar varsitt meddelande så ska det inte vara möjligt att något ser A:s meddelande först och någon ser B:s först.

Ni rekommenderas att använda någon form av versionshantering. Detta är bra både för att dela kod med sin labbpartner, men också för att kunna ångra ändringar i koden ifall något slutar fungera. Institutionen tillhandahåller GitLab på adressen `https://git.cs.umu.se`.

B Givet material

För att ni lättare ska komma igång med uppgiften så tillhandahåller vi lite kod. Bland annat tillhandahåller vi en testsvit med enhetstester för några PDU:er. Observera att denna testsvit inte nödvändigtvis kommer hitta alla möjliga fel i PDU:erna. Men den är bra att utgå ifrån när ni bygger tester för resten av era PDU:er.

PDU-klassen innehåller några hjälpmetoder som kan vara bra att använda.

- `fromInputStream` är en statisk metod som används för att avgöra vilken subklass konstruktor som ska anropas. Det innebär att ni slipper ta reda på vilken PDU som tagits emot medan den läses in. Ni får dock implementera denna metod själva.
- `padLengths` är en metod som tar emot ett antal heltalslängder och paddar dessa så de blir jämnt delbara med 4. Sedan beräknas deras summa. Denna metod är bra för att beräkna hur många bytes som ska läsas från en ström när en PDU:s header lästs in.

- `readExactly` är en metod som läser ett givet antal bytes från en `InputStream`. Denna metod är bra för att `InputStreams` metod kan läsa färre bytes än vad som angetts, vilket kan vara krångligt att hantera.
- `byteArrayToLong` är två metoder som kan användas för att konvertera hela eller delar av byte-arrayer till longs för att undvika sign extension och för att slippa göra detta manuellt varje gång.
- `toByteArray` är en abstrakt metod som behöver implementeras för varje PDU. Den används för att konvertera den interna representationen hos varje PDU till en byte-array som kan skickas genom en ström.

Klassen `CheckSum` kan användas för att beräkna checksummor (se dokumenterad kod).

Klassen `ByteSequenceBuilder` kan användas för att smidigt bygga byte-sekvenser som sedan kan skrivas till strömmar. Se dokumenterad kod för exempel på hur den kan användas.

C Exempel på testprotokoll

Testprotokoll ska både finnas i rapporten och tas med till demonstrationen av lösningarna. För extra tydlighet tillhandahåller vi ett exempel på ett enkelt testprotokoll som ni kan utgå från när ni skapar era egna:

C.1 Test 1

Syfte: Demonstrera att en klient kan ansluta till chatten.

Förutsättningar: Chattserver körs.

1. Starta klient.
2. Ange adress och port till chattserver.
3. Klicka på anslut.
4. Verifiera att klienten anslutit till chatten (meddelande ``` ``Joined server at ...'' ``` visas i meddelandefältet).

D Protokolldataenheter

I all kommunikation är meddelandena indelade i protokolldataenheter (PDU:er). Dessa PDU:er består av ett eller flera words där varje word består av 4 bytes. Värdet på den första byten (PDU:ns OP-kod) i varje PDU beskriver vilken typ av PDU det är, och utifrån denna byte kan formatet på PDU:n bestämmas.

Alla OP-koder visas i Tabell 2. Varje PDU består av flera fält som används i kommunikationen. Det kan vara t.ex. IP-adresser, ID:n eller textmeddelande. Det finns två typer av fält. Den första typen består av ett fast antal bytes och har en på förhandbestämd position i varje PDU. Dessa fält ryms alltid inom ett enda word. Den andra typen är fält med obestämd längd som kan överstiga längden hos ett word. Dessa fält börjar alltid i början på ett word och i slutet fylls de alltid ut med bytes med värdet 0 så att de alltid slutar i slutet på ett word. Deras längd anges alltid i ett annat fält i en PDU. Förutom i slutet på de längre fälten så kan det finnas fasta luckor i PDU:er. Dessa ska "paddas", d.v.s. fyllas ut med bytes med värdet 0.

Alla strängar som skickas i systemet ska kodas med teckenkodningen UTF-8. Strängar ska inte nullermineras, d.v.s. behövs det 4 bytes för att representera en sträng så ska fältet som strängen ryms i vara 4 bytes. Undantaget är Nicks-PDU:n som innehåller en lista av nicknames. Där ska varje nickname nullermineras, inklusive det sista. När det sista nicknamet nullterminerats så ska fältet paddas så det precis slutar i slutet på ett word.

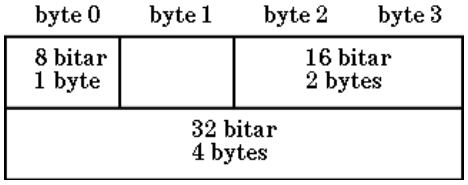
Protokolldataenheter ska vara *big-endian*, d.v.s. de mest signifikanta byten ska vara först. Om ett fält innehåller en short med värdet 256 så ska den första byten ha värdet 1 och den andra värdet 0.

Tidsstämplar anges i Unix-tid, d.v.s. i antal sekunder sedan 00:00:00 UTC första januari 1970. De flesta programmeringsspråk har stöd för att ta fram Unix-tid direkt. Tidsstämplar anges i 32 bitar.

Detta protokoll har vissa begränsningar:

- Ett nickname kan inte vara längre än 255 tecken (bytes).
- Ett servernamn kan inte vara längre än 255 tecken (bytes).
- En namnserver kan bara hålla koll på 65535 chattservrar.
- En chattserver kan bara ha 255 klienter.
- Ett meddelande kan inte vara längre än 65535 tecken (bytes).
- Tidsstämpeln fungerar endast till ca år 2036.

Formatbeskrivning:



Meddelandetyp	Op-code
REG	0
ACK	1
ALIVE	2
GETLIST	3
SLIST	4
MESS	10
QUIT	11
JOIN	12
CHNICK	13
UJOIN	16
ULEAVE	17
UCNICK	18
NICKS	19
NOTREG	100

Tabell 2: Op-koder för de olika meddelandetyperna.

D.1 Server-namnserver

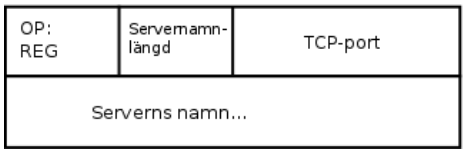
Dessa PDU:er skickas mellan chattservern och namnservern via UDP.

D.1.1 REG

Skickas från chattservern till namnservern för att registrera sig.

- **Servernamn-längd:** Längd på UTF-8-representation av servernamn (ej paddat/nullterminerat).
- **TCP-port:** Port där servern accepterar klienter (0-65535).
- **Serverns namn:** Paddad, ej nullterminerad UTF-8-representation av serverns namn.

CS -> NS: Registrera



D.1.2 ALIVE

Skickas från chattserver till namnserver för att fortsätta vara registrerad.

- **Antal klienter:** Antal chattklienter anslutna till chattservern.

- **ID-nummer:** ID-nummer servern fick som svar på `REG`-meddelandet.

CS–NS: Jag lever

OP: ALIVE	Antal klienter	ID–nummer

D.1.3 ACK

Skickas från namnserver till chattserver som bekräftelse på `REG` eller `ALIVE`.

- **ID-nummer:** ID-nummer som chattservern ska använda i sina `ALIVE`-meddelanden.

NS –> CS: Bekräftelse

OP: ACK	Pad	ID–nummer

D.1.4 NOTREG

Skickas som svar på `ALIVE` om ingen server med det aktuella ID:t är registrerad.

NS –> CS: Ej registrerad

OP: NOTREG	Pad

D.2 Klient-namnserver

Dessa PDU:er skickas mellan klienten och namnservern via UDP.

D.2.1 GETLIST

Skickas från klient till namnserver för att begära en lista med aktiva chattservrar.

Klient–> NS: Hämta serverlista

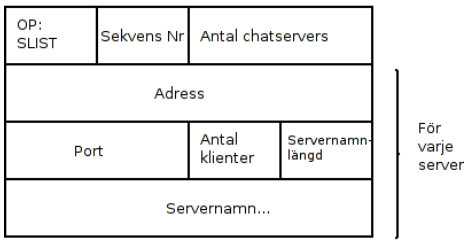
OP: GETLIST	Pad

D.2.2 SLIST

Skickas som svar på `GETLIST`.

- **Sekvensnr:** Nummer på SLIST-PDU:n. Alla PDU:er som skickas som svar på en `GETLIST` har ett uniks sekvensnummer.
- **Antal chattservers:** Antal chattservrar i PDU:n.
- För varje server:
  - **Adress:** IPv4-adress till chattservern.
  - **Port:** Portnummer till chattservern.
  - **Antal klienter:** Antal klienter anslutna till den specifika chattservern.
  - **Servernamnlängd:** Längd på UTF-8-representation av servernamn (ej paddat/nullterminerat).
  - **Servernamn:** Paddad, ej nullterminerad UTF-8-representation av serverns namn.

NS -> Klient: Serverlista



D.3 Klient-server

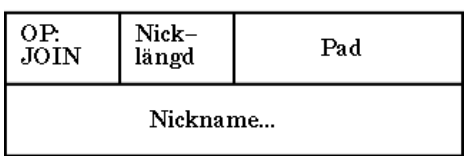
Dessa PDU:er skickas mellan klienten och namnservern via TCP.

D.3.1 JOIN

Skickas från klient till chattserver för att ansluta till chatten.

- **Nicklängd:** Längd på UTF-8-representation av nickname (ej paddat/nullterminerat).
- **Nickname:** Paddad, ej nullterminerad UTF-8-representation av nicknamet.

Klient -> CS: Anslut



D.3.2 NICKS

Skickas som svar på `JOIN` om klienten tillåts ansluta till chatten.

- **Antal namn:** Antal nicknames i PDU:n.
- **Längd:** Längd på UTF-8-representationer av nicknames inklusive nullterminering av varje nickname (inklusive sista), exklusive padding.
- **Nicknames:** Paddad UTF-8-representation av alla nicknames närvarande i chatten (inklusive klienten som precis anslöt) med en 0-byte efter varje namn (inklusive efter sista namnet).

CS -> Klient: Nicknames

OP: NICKS	Antal namn	Längd
Nicknames...		

Nicknames exempel

OP: NICKS	2	9	
O	P	\0	K
A	L	L	E
\0	\0	\0	\0

D.3.3 QUIT

Skickas från chattserver eller klient innan anslutningen till den andra parten stängs.

CS <-> Klient: Avsluta

OP: QUIT	Pad

D.3.4 MESS

Innehåller ett meddelande, skickas från klient till chattserver eller från chattserver till klienter.

- **Nicklängd:** Längd på avsändarens nickname. Ska vara 0 när PDU:n skickas från klienten.

- **Checksumma:** Checksumma enligt beskrivning ovan.
- **Meddelandelängd:** Längd på ej paddad/nullterminerad UTF-8-representation av meddelandet (0-65535).
- **Tidsstämpel:** Unix-tid (antal sekunder från 1970 till att meddelandet skickades). Ska vara 0 då meddelandet skickas från klienten och sättas av chattservern.
- **Meddelande:** Paddad, ej nullterminerad UTF-8-representation av meddelandet.
- **Nickname:** Paddad, ej nullterminerad UTF-8-representation av nicknamet på avsändaren. Ska inte skickas med från klienten utan läggs till av chattservern.

CS <–> Klient: Meddelande

OP: MESS	Pad	Nick–längd	Check–summa
Meddelandelängd		Pad	
Tidsstämpel (sekunder sedan 1970)			
Meddelande...			
Nickname...			

D.3.5 UJOIN

Skickas från chattserver till klient när en annan klient ansluter till chatten.

- **Nicklängd:** Längd på UTF-8-representation av nickname (ej paddat/nullterminerat).
- **Tidsstämpel:** Unix-tid (antal sekunder från 1970 till att klienten anslöt till chatten).
- **Nickname:** Paddad, ej nullterminerad UTF-8-representation av nicknamet.

CS –> Klient: Någon anslöt

OP: UJOIN	Nick–längd	Pad
Tidsstämpel (sekunder sedan 1970)		
Nickname...		

D.3.6 ULEAVE

Skickas från chattserver till klient när en annan klient lämnar chatten.

- **Nicklängd:** Längd på UTF-8-representation av nickname (ej paddat/nullterminerat).
- **Tidsstämpel:** Unix-tid (antal sekunder från 1970 till att klienten lämnade chatten).
- **Nickname:** Paddad, ej nullterminerad UTF-8-representation av nicknamet.

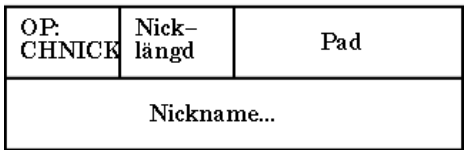
CS -> Klient: Någon lämnade

D.3.7 CHNICK

Skickas från klient till chattserver för att klientens nickname i chatten ska ändras.

- **Nicklängd:** Längd på UTF-8-representation av nickname (ej paddat/nullterminerat).
- **Nickname:** Paddad, ej nullterminerad UTF-8-representation av nicknamet.

Klient -> CS: Byta nickname



D.3.8 UCHNICK

Skickas från chattserver till klient när en annan klient bytt nickname.

- **Nicklängd 1:** Längd på UTF-8-representation av gammalt nickname (ej paddat/nullterminerat).
- **Nicklängd 2:** Längd på UTF-8-representation av nytt nickname (ej paddat/nullterminerat).

- **Tidsstämpel:** Unix-tid (antal sekunder från 1970 till att klienten bytte nickname).
- **Gammalt nickname (1):** Paddad, ej nullterminerad UTF-8-representation av det gamla nicknamet.
- **Nytt nickname (2):** Paddad, ej nullterminerad UTF-8-representation av det nya nicknamet.

CS -> Klient: Någon bytte nick

OP: UCNICK	Nick-längd 1	Nick-längd 2	Pad
Tidsstämpel (sekunder sedan 1970)			
Gammalt nickname (1)			
Nytt nickname (2)			