1. ***Ved bruk av TCP er det viktig å lage en egen protokoll (applikasjonslags-protokoll) på toppen av Berkeley socket laget. Hvilke egenskaper har TCP som gjør denne protokollen så viktig?***

TCP er forbindelsesorientert. Pakker blir sendt på nytt hvis det oppstår feil og pakkene ankommer I riktig rekkefølge.
Protokollen sørger for at TCP kan slå sammen flere beskjeder og rekkefølgen blir riktig. Klient og tjener havner ellers I utakt.

2. ***Hvordan ser din applikasjonslags-protokoll ut?***
Meldingsbasert:
1. <size>:message
2. klient: receive meny – send command – receive status
3. bestemmer hvordan beskjeden ser ut (proto.h)
4. klient sender input fra tastaturet til tjeneren uten noe logikk

Begrunn designet!
1. Bruker size for å finne ut hvor lang meldingen er. Hvis den er lengre, er det en ny beskjed.
2. Bruker det slik for å synkronisere tilstanden mellom klient og tjener. Klienten ble også veldig enkel.
3. Definere størrelse på forespørsler, messages og funksjonene proto_recv og proto_send. Hver beskjed er ren ASCII (gjelder også punkt 4). Gjør protokollen forståelig.

3. ***Ta hensyn til maskinarkitekturen! Hva kan gå galt?***
Ville programmet ditt virke hvis du kompilerer og kjører tjeneren din på en arkitektur med Big Endian, og klienten på en av Linux-maskinene i Ifis terminalstuer?

Beskjeden blir nonsense hvis den tolkes med feil endianness.

- Eksempel, 32-bit number, 0xDEADBEEF:
  - *Big-Endian:*

Memory Location	Value
Base Addr + 0	DE
Base Addr + 1	AD
Base Addr + 2	BE
Base Addr + 3	EF

Little-Endian

Memory Location	Value
Base Addr + 0	EF
Base Addr + 1	BE
Base Addr + 2	AD
Base Addr + 3	DE

Programmet mitt sender bare strings, så jeg har unngått hele den problematikken.
Så, det skal funke med tjener / klient med forskjellig endianness.

4. Programmet ditt bruker forbindelsesorientiert kommunikasjon, men nettfilsystemet NFS bruker hovedsaklig forbindelsesløs kommunikasjon.
Hvilke problemer unngår du i programmet ditt ved å bruke forbindelsesorientiert kommunikasjon?

- Unngår tap av pakker – slipper å skrive kode for å sende på nytt
- slipper å gjenopprette forbindelser
- slipper å ta forbehold om at pakkene kommer ut av rekkefølgen
- slipper å kjøre feilsjekking på kommunikasjonen

5. ***Hvilke muligheter tilbyr Linux og Berkeley socket APIen for å utvikle tjenere som kan opprettеholde flere forbindelser samtidig, og som kan kommunisere med flere klienter samtidig? Hvilken av mulighetene har du valgt og hvorfor?***

Linux

- Fork() per client (easiest, ressurskrevende)
- pthreads() (lightweight, more complex)
- single thread, asynkron (super fast, super complex)

Berkeley:
accept() mange sockets.

Jeg har valgt fork() fordi den var lettest å implementere