

# Vad är speciellt med implementering av kommunikationsprotokoll?

---

# Översikt

- **Inledning**
- **Hårdvaru-vy och mjukvaru-vy**
- **Mjukvaruarkitektur**
- **“Rules of thumb”**
- **Minnesbandbredd**
- **TCP-implementering**
- **ILP – Integrated Layer Processing**

---

# Kommunikationsprestanda

**Två mått:**

- **genomströmning (throughput)**
- **fördröjning (delay)**

---

# Kostnader

**Två mått:**

- **per byte**
- **per paket**

# Faktorer som påverkar protokollstacksprestanda

**(Givet en viss hårdvara)**

- **strukturen: hur protokollstacken partitioneras mellan OS och tillämpning, hur hårdvaran accessas, hur (om) skikten separeras i implementationen**
- **OS-funktioner: kontext-byte, schemaläggning, minneshantering, avbrottshantering**

## **Ex: paket anländer till väntande tillämpning**

- **Avbrott som startar avbrottshanteraren**
- **Avbrottshanteraren sköter protokollprocessning och överlämnar data till tillämpningen**
- **OS:et måste kopiera data till tillämpningens buffert och göra kontext-byte**

**OS-funktionerna, avbrott, kopiering och kontext-byte, kostar minst lika mycket som protokoll-processningen. Protokollprestanda beror till stor del på implementationsomgivningen.**

---

# Strukturen på en protokollstackimplementation

## partitionering:

- monolitisk i OS:ets kärna
- monolitisk i en tillämpningsprocess (server)
- per process i tillämpningens adressrymd

## gränssnitt:

- single-context
- tasks
- upcalls

## Partitioneringsstrategier

**Vilken funktionalitet skall vara i OS:et (kernel space), som vi litar på, och vilken kan vara i tillämpningens adressrymd (user space), som vi inte litar på?**

**Balans mellan:**

- **software engineering (utveckling, test, underhåll)**
- **möjlighet att skraddarsy**
- **säkerhet**
- **prestanda**

---

## **Monolitisk implementation i OS:ets kärna**

**Allt utom ev. tillämpningsprotokoll i OS:ets kärna**

**Enklaste och mest vanligt förekommande valet**

**Har högst säkerhet**

**Svårare att implementera och debugga**

**Svårt att skraddarsy för speciella tillämpningsbehov**

**Ex. Unix, OS/2 och Windows NT**

## **Monolitisk implementation i tillämpningsprocess (server)**

**I princip hela protokollstacken implementeras i en pålitlig (“trusted”) server-process.**

**Tillämpningen anropar denna server-process i stället för OS:et.**

**Säker**

**Lättare att implementera och debugga**

**Dålig prestanda pga många kontext-byten**

**Ex. Mach, men också WINSOCK DLL i Windows**

## **Per-process i tillämpningens adressrymd**

**Implementeras som ett bibliotek som länkas in i tillämpningen.**

**Bra prestanda (om rätt implementerat)**

**Möjlighet att skraddarsy**

**Lätt att debugga och underhålla**

**Dålig säkerhet – kan lösas med “registry server”**

**Stor binär (om ej delat bibliotek)**

**Multi-trådning nödvändigt för många protokoll**

## Jämförelse

| <b>Teknik</b>  | <b>Skräddar-<br/>sybarhet</b> | <b>Prestanda</b>                        | <b>Säkerhet</b>                           | <b>Utveck-<br/>lingskost-<br/>nad</b>           |
|--|-------------------------------|---|---|---|
| <b>monolitisk i<br/>kärnan</b>                                   | <b>dålig</b>                  | <b>bra</b>                              | <b>bra</b>                                | <b>hög</b>                                      |
| <b>monolitisk i<br/>server</b>                                   | <b>dålig</b>                  | <b>dålig</b>                            | <b>bra</b>                                | <b>medel</b>                                    |
| <b>per-protokollstack i<br/>server</b>                           | <b>medel</b>                  | <b>dålig</b>                            | <b>bra</b>                                | <b>låg</b>                                      |
| <b>bibliotek per-<br/>process, per-<br/>protokoll-<br/>stack</b> | <b>mycket bra</b>             | <b>bra, med<br/>noggrann<br/>design</b> | <b>bra, men<br/>svårare att<br/>uppnå</b> | <b>låg, förut-<br/>satt multi-<br/>trådning</b> |

---

## Single context

**Protokollskikten har en gemensam exekveringstråd som ej kan avbrytas.**

**Endast ett paket processas åt gången, så inga lås behövs för protokollens delade datastrukturer.**

**Koden behöver inte vara reentrant.**

**Effektivt – ger låg fördröjning genom stacken.**

---

## Tasks

**Varje skikt är en “task” som aktiveras från en “task scheduler”.**

**Varje task processar ett paket klart innan den returnerar.**

**En task kan fortsätta processningen av paket genom att anropa en annan task, eller “åter-schedulera” sig själv efter en timeout.**

**Task scheduler har friheten att aktivera tasks med hög prioritet först – ger möjlighet att erbjuda servicekvalitet.**

## Upcalls

**Nyckelidé: en tråd per paket.**

**Tråden tar hand om alla skikts protokoll-processning.**

**Varje skikt registrerar entry-punkter för att sända och ta emot paket hos skiktet under.**

**När ett skikt vill överlämna ett paket, eller ta emot ett paket från skiktet ovan, så anropar den resp. entry-punkt.**

**Måste använda lås till gemensamma datastrukturer därför att flera trådar kan vara aktiva samtidigt.**

## **Tumregel 1: Optimera för det vanliga fallet**

- **Gäller generellt för programutveckling.**
- **Identifiera vad som är den vanliga exekveringsvägen genom koden och optimera denna med avseende på tid.**
- **För kommunikationsprotokoll är det vanliga fallet ett felfritt dataflöde, med generering och mottagning av paket i sekvens.**
- **Kan tex beräkna protokoll-huvud i förväg.**
- **Byt ordning på tester för att passa vanliga vägen.**

---

## **Tumregel 2: Var medveten om flaskhalsarna**

- **Var är flaskhalsen i systemet?**
- **Kan den bytas ut eller användas på ett effektivare sätt?**
- **Genom att åstadkomma en balanserad pipeline, så kan man maximera plattformens prestanda.**

## Tumregel 3: Fin-justera inre loopar

- Det är ofta en liten del av koden som bidrar till en stor del av exekveringstiden.
- De innersta looparna är vanligaste kandidater.
- Överväg att rulla ut och/eller att koda dessa i assembler.
- Exempel: kopieringsloop, checksummeberäkning.

---

## **Tumregel 4: Välj bra datastrukturer**

- **Val av datastrukturer kan dramatiskt påverka prestandan.**
- **Ex: buffrar för protokoll-headers och data, timers och datastrukturer för att lagra förbindelsers tillstånd.**
- **Sökning av datastrukturer i den “vanliga vägen” måste vara mycket snabb.**

## **Tumregel 5: Undvik att “ta på” data**

- **Ett av de värsta misstagen är att kopiera ett paket flera gånger än vad som är absolut nödvändigt.**
- **Gör alla ansträngningar för att undvika att “ta på” data mer än en gång.**
- **Försök kombinera flera data-operationer i samma loop, tex beräkning av checksumma och kopiering.**

## Tumregel 6: Minimera antalet sända paket

- Mottagning och sändning av paket tar resurser i både nätet och slut-systemet.
- “Piggyback” är ett sätt att haka på ytterligare information på paket som ändå skulle skickats, tex ACK på data-paket.
- Exempel på ineffektivitet är HTTP som öppnar en ny TCP-förbindelse för varje objekt, vilket resulterar i minst tre extra paket per objekt: SYN, SYN-ACK och FIN (förutom den extra fördröjningen).

---

## **Tumregel 7: Sänd så stora paket som möjligt**

- **Ju större paket, ju mindre total overhead för paket-huvuden.**
- **Det minimerar antalet paket för en given data-mängd, och därmed summan av den fasta kostna-den för paketen.**

## **Tumregel 8: Cache hints**

- **En cache utnyttjar lokalitet – om någon operation har lokalitet, så kan en cache ofta förbättra prestanda.**
- **Protokoll-kontroll-block är en typisk kandidat. Om ett paket tagits emot på en viss förbindelse är det stor sannolikhet att nästa paket hör till samma förbindelse.**

---

## **Tumregel 9: Använd hårdvara om möjligt**

- **Ibland är det värt att utveckla hårdvara för en funktion.**
- **Exempel är beräkning av TCP-checksumma på nätverksadaptern.**

---

## **Tumregel 10: Utnyttja tillämpningens egenskaper**

- **Gäller tillämpningsprotokoll som tex HTTP**
- **Utnyttja kunskap om tillämpningen för att optimera protokollet.**