1. Reactor

1. **Explain the "Reactor" pattern mechanism**
2. **Describe the use of the Reactor in a Server and in a Client node**
3. **Describe the differences between the Proactor and the Reactor pattern and when to use the Reactor or the Proactor in a node**

Problem

Eks. Client/Server baseret systemer
Mange klienter kommunikerer til en central server, en event-driven applikation
Vi vil gerne processere events serielt, men modtage dem concurrent
Vi skal derfor have en enhed der demultiplexer og dispatcher events

1.
Implementation
Inddrag komponenterne, Reactor, EventHandler, konkrete EventHandlers, Handles, Demultiplexer
Hvad sker der (uformelt flow)?

2.
Client/Server sammenhæng?
En server kan benytte reactor til at vente på socket connections
På den måde kan one thread pr. connection undgås
Kan evt. sammenkobles med Connector/Acceptor
Kan nemt sammenkobles med en concurrency strategi som half sync/half async
For en klient, kan vi vente på at få svar på requests sendt til servere
Browser eksempel, har mange connections på en gang, reactoren demultiplexer disse

3.
Reactor og Proactor patterns
Problemet er at event handlers kører i samme tråd som Reactoren
Vi kan garantere at læsning/skrivning er nonblocking (*pga. select()*) men det kan stadig være sløvt
Hvis en EventHandler tager lang tid, bliver systemet unresponsive
Dette problem kan fikses med Proactor
Proactoren gør brug af asynkrone systemkald, til at udføre tidskrævende operationer
Giv eksempel med at skrive en meget stor fil (tager lang tid selvom det er nonblocking)
Med asynkront systemkald blocker dette intet
For at benytte Proactor skal OS dog supportere asynkrone systemkald (WinNT gør, UNIX gør ikke)
Derfor er Proactor ikke altid en mulighed

Proactor er dog mere kompliceret at implementere da asynkrone kald er dekoblet i tid

2. Paradigms for distributed communication

1. **Explain the basic three coupling mechanisms: time, space and flow coupling**
2. **Explain these couplings in relation to RMI and Publish/subscribe communications**
3. **Give examples of other communication paradigms**

Time, Space og flow coupling

Publish/Subscribe er dekoblet på alle tre måder, hvilket gør strukturen meget fleksibel

Time coupling - afsender og modtager er online samtidig. Requests er transient, og ikke persistant

Space coupling - parterne kender hinanden og benytter point-to-point kommunikation

- Publish/Subscribe kommunikerer gennem en proxy (*Event Service*)

Flow coupling - kommunikationen mellem to parter er synkron.

- Flow coupling kan både være på klienten og serverens side
- Publish/Subscribe kalder blot notify asynkront (*forventer ikke svar fra Event Servicen*)
- Publish/Subscribe forker en process (*eller tråd*) der venter på notifikation (*signal el. interrupt*)

Kobling af RMI

Målet med RMI er at agere som var metoden lokal

Flow coupling - Normale metodekald er ikke asynkrone, derfor er RMI flow koblet

- Dette kan til dels rettes op på med futures

Space coupling - Klienten bliver nødt til at kende severen som metoden skal kaldes på

Time coupling - Da requesten er transient (*ingen queue lag imellem*) skal begge services køre samtidigt

Andre kommunikationsparadigmer

Message Passing

Message Queuing

Tuple spaces

3. Acceptor/Connector Pattern

1. **Explain the "Acceptor/connector" pattern mechanism**
2. **Describe when to use an acceptor, a connector or both patterns in a node**
3. **Describe the difference between a synchronous and an asynchronous connector**

Problem
Ofte har initialisering og oprettelse af connnections intet med servicen der tilbydes at gøre
Vi vil derfor gerne dekoble disse ting
- Nemt at ændre connection roller uden at ændre servicen
- Nemt at tilføje nye services
- Dette tillader fx at benytte forskellige concurrency strategier i handlers uden at ændre acceptoren

Løsningen - step by step flow
Decouple, så en connector factory laver en service handler og giver den en handle
En acceptor factory laver en service handler og giver den en handle
Efter initialisering kommunikerer disse parter ikke mere

Anvendelse
- Connector → opretter forbindelser
- Acceptor → accepterer inkomne forbindelser

Applikation tilbyder flere services, vil gerne have generel måde at oprette forbindelser
Eksempel med client server (*browser → server*)
- Connector er hos klienten
- Når website requestes oprettes masse forbindelser, denne kode kan være centralt sted
- Derefter oprettes handlers til de forskellige connections

Server - mange klienter connecter til en server
- Central kode til at acceptere forbindelser og oprette handlers

Torrentnetværk eller andre P2P systemer
- Knuder er både server og klient (*servents*)
- Har både connector og acceptor

Synkron og asynkron connector
Synkron connector blocker indtil connection er oprettet
- Der oprettes forbindelse til serveren
- Ny ServiceHandler oprettes og connection gives til denne

Asynkron connector
- Starter asynkront connection job (*typisk et OS kald*)
- Bliver notificeret af dispatcher når connection er oprettet
- Laver ny ServiceHandler og giver connection til denne
- Ville fungere godt med Proactor som dispatcher

4. Proactor

1. **Explain the "Proactor" pattern mechanism and its interaction with the Asynchronous Completion Token (ACT) pattern**
2. **Describe for what and when these pattern are used**
3. **Describe the benefits in comparison with the use of a Reactor pattern in combination with a concurrency pattern**

Problem og løsning
Vi vil gerne processere tidskrævende operationer asynkront og proaktivt

1. Eks. Hvis vi vil læse på en socket sættes et asynkront recv() kald i gang.
2. Når der kommer data og det er læst genereres et completion event
3. Proactoren dispatcher til en CompletionHandler (*dette sker synkront i samme tråd*)

Hvad er en ACT?
Ligesom cookies på internettet. De gemmer information om det asynkrone kald
Proactoren benytter disse til at demultiplexe completion events

1. Når initiatoren laver et asynkront kald laves en ACT der peger på en CompletionHandler
2. ACTen sendes med det asynkrone kald
3. ACTen returneres sammen med returværdien af det asynkrone kald
4. Proactoren benytter ACTen til at demultiplexe completion eventet og dispatche til den korrekte CompletionHandler

Muliggør effektiv demultiplexing
Meget simpel løsning, kan give sikkerheds problemer

Hvor bruges Proactor pattern?
Proactor benyttes ofte som en optimeret version af Reactor pattern - dvs. client/server sammenhæng
Kan kun bruges når OS understøtter asynkrone kald (*ellers mistes fordelen*)

Forskel fra Reactor med concurrency
Da Proactoren benytter OSets asynkrone API undgås contextswitching - højere effektivitet og sikkerhed
Det er dog mere besværligt at arbejde med asynkrone kald (*da de er dekoblet i tid*)
Reactor EventHandlers kan kodes uden at tage hensyn til concurrency strategien
Proactor CompletionHandlers skal håndtere ACTen
Multithreading er svært at debugge og kan introducere race conditions

5. Architecture Tradeoff Analysis Method

1. **Describe for what and when the ATAM method is used**
2. **Explain the ATAM method steps**
3. **Describe the purpose of architecture view models and the relation to ATAM**

Hvad er ATAM

Et framework til tradeoff analyse
Lav informerede valg omkring tradeoffs i en designprocess
    Process prioritet kan have indflydelse på availability og performance
    Nå så tæt på system requirements indenfor prisrammerne
ATAM hjælper med at strukturere disse tradeoff overvejelser
Benyttes i starten af designprocessen af software systemer
    Hvor ændring af arkitekturen er billigt

ATAM steps

Lav scenarier i samarbejde med stakeholders
Find requirements (eks. soft-realtime), constraints (eks. net forbindelse) og environment (eks. interaktion)
Beskriv architectural views - benyt eks. 4+1 architectural view model
Analyser attrubutter (*security, performance*) individuelt
Ændr en attribut og se hvilke, hvis en værdi ændrer sig meget er dette et "*sensivity point*"
Find tradeoff points - sensitivity points der ændrer mere end en attribut

Architecture view models

Problematisk når mange dele af arkitekturen tegnes i samme diagram
Opdeling af arkitekturen i flere views
4+1 architectural view model er et eksempel (logisk, udvikling, fysisk, process + use cases)
Når arkitekturen skal beskrives i ATAM kan dette bruges

6. Leader/Followers and Half sync/half async Patterns

1. **Explain the "Leader/followers" pattern mechanism**
2. **Explain the main structure of the "half sync/half async" pattern**
3. **Describe the advantages of the Leader/Followers in relation to half sync/half async**

Problem
Vil gerne behandle mange requests samtidigt
Ved mange requests er DTC ikke mulig (*oprette en thread per requests*)
Skal have en metode at starte threads fra en threadpool

Leader/Followers
Taxa ved banegården analogi
Threadpool med en leader der venter på events
Når et event sker promotes en ny leader og gammel leader bliver “processing”
Bagefter bliver leaderen “idle” og joiner threadpoolen igen
Tegn state diagram (*LEADING, PROCESSING, FOLLOWING*)

Half sync/Half async
Tegn og beskriv de 3 lag

LF fordele
Locking - Minimerer locking overhead da der ikke er interthread kommunikation
En prioritet - Da det ekstra queue lag er væk arbejdes kun med én prioritering (*undgår priority inversion*)
Contextswitch - Minimerer kontekstswitch, da samme thread demultiplexer og handler

7. Interceptor Pattern

1. **Explain the "Interceptor" pattern mechanism**
2. **Describe how the interceptor can be used in developing a framework and the division of responsibilities between developers**
3. **Describe benefits and liabilities**

Problem
Framework arkitekter kan ikke forudse alle usecases
To muligheder
    Lav kæmpe framework der kan alt
    Implementer Interceptor Pattern så frameworket kan udvides

Løsning
Registrer interceptor ved frameworkets dispatcher
Når frameworket når et hookpoint dispatches til interceptoren
Der medsendes et context objekt så interceptoren kan interagere med frameworket
Interceptoren udføres og frameworket kører videre

Opdeling af ansvar
Et hold kan fokusere på strukturen af frameworket
    Behøver ikke at tænke på hvad alle frameworkets use cases
Et hold kan fokusere på implementation af interceptors
    Skal ikke vide hvordan frameworket fungerer, kun kende context interfaces og hook points

Fordele og ulemper
+ Åbner frameworket op så det kan customizes
- Svært at bestemme interception points. For få = ikke fleksibelt, for mange = ineffektivt + bloat
- Fejl i interceptors kan ødelægge hele frameworket

8. JAWS Framework

1. **Explain the overall architecture of the JAWS-framework and how it is designed based on POSA2, POSA1 and GoF patterns**
2. **Describe the reconfiguration in JAWS**
3. **Describe the purpose of the used POSA2 design patterns**

JAWS opbygning
Framework med inversion of control
Kan benytte Reactor eller Proactor til demultiplexing
Skal vælge hvilke strategier der skal bruges (*demultiplexing, concurrency, caching*)
Kan implementere egne løsninger hvis det ønskes vha. Strategy Pattern
Øger driftsikkerhed da mindre applikationsspecifik kode skrives

Hvorfor patterns
Meget fleksibelt
Bekendt for mange og let at forstå
Best practice

Vis diagram og forklar alle patterns og deres formål

Rekonfiguration
Kan rekonfigureres på runtime pga. Stragety Pattern
Eks. skift concurrency strategi når høj load opleves
Ligesom SAFCA (Self-Adaptive Framework for Concurrency Architecture)

9. Component Configurator Pattern

1. **Explain the "Component Configurator" pattern mechanism**
2. **Describe how it can support dynamic reconfiguration at runtime**
3. **Describe benefits and liabilities**

Problem
Konfigurering af et system på runtime
Det kan være for dyrt at genstarte systemet for at opdatere
Eks. skifte caching strategi på billede server

Løsning
Fodbold analogi - Kampen stoppes ikke fordi der sker en udskiftning.
Lav *component configurator* - læser config fil og initialiserer components
Component repository - håndterer forskellige components
Component interface til at starte og stoppe components
Nye components wrappes i DLL
Disse kan loades dynamisk med script givet til configuratoren

Fordele og ulemper
+ Uniformitet. Alle components ligner hinanden, hvilket gør dem lette at arbejde med
+ Reusable code. Konfiguration og kode er ikke koblet. Derfor kan componenten nemt genbruges
+ Kontrol. Dårlig eller fejlagtig kode kan nemt udskiftes uden at stoppe systemet

- Svært at analysere programmet, da komponenter kan udskiftes på runtime
- Mere overhead og indirection
- Stor kompleksitet for små systemer