1. Reactor pattern

Telefon metafor. Registrer ved en udbyder (*Reactor*). Når nogen ringer stiller udbyderen opkaldet igennem til dig (*event handler*). Opkaldet besvares og samtalen startes (*handle_event()*).

Problem

- Mange klienter kommunikerer til en central server, eks. en logging server
  - Altså en event-driven applikation
- En løsning er DTC, men det har nogle drawbacks (s. 180)
  - Ineffektivt og kompleksitet
- Vi vil gerne processere events serielt, men modtage dem concurrent
  - Vi skal derfor have en enhed der demultiplexer og dispatcher events
  - Vi skal være opmærksomme på 4 ting (s. 181)
    - Applikationen må ikke blocke på et event
    - Synchronization og contextswitching skal undgås (ingen threads)
    - Det skal være let at tilføje nye service handlers
    - Applikationskoden må ikke indeholde multithreading eller synchronization

Løsning

- Block mens der ventes på indication events (eks. select() systemkaldet)
- Demultiplex disse events og dispatch til eventhandlers

- Det smarte ved select() er at vi kan være sikre på operationer på en handle ikke blocker

Struktur (med UML klassediagram)

- Tegn Reactor med 2 EventHandlers
- Forklar applikationens flow
- Forklar brug af Singleton og Bridge i afleveringen

- Det eneste der skal gøres for at tilføje funktionalitet er at implementere nye eventhandlers

Beskriv anvendelse i server og klient

Server	Som sagt lyt på concurrent events på en synkron måde (simplicitet, effiktivitet)
Client	Mange åbne connections kræver mange tråde venter på respons. Med Reactor pattern bliver dette synkront, og derved simplere at implementere
Fordele	Separation of concerns. Event handlers skal kun koncentrere sig om funktionalitet Moduler er løst koblet og eventhandlers kan ændres uafhængigt af reactoren Portability. Det er let at udskifte hivlket demux systemkald der benyttes (*bridge pattern*)
Ulemper	Eventhandlers skal være hurtige, da hele systemet ellers bliver langsomt

Reactor vs. Proactor

- Reactor skalerer ikke godt. Den kan kun bearbejde en request ad gangen, og dette skal være hurtigt

Hvornår skal Reactor anvendes og hvornår skal Proactor anvendes

- Når der er OS support for asynkrone kald, som Windows NT er Proactor fordelagtig
- Proactor er mere komplex end Reactor, derfor kan Reactor være god til mindre projekter

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**

Ref. POSA2 p. 179-214 + slides

4 Proactor and ACT Patterns

4.1 Proactor

Problem

Vi vil gerne processere requests asynkront
Når et asynkront job er færdigt sendes en completion event fra OSet

For at opnå høj effektivitet skal fire regler overholdes
- Completion events skal håndteres samtidigt
- Undgå så vidt muligt context switching og synkronisering
- Let at tilføje og redigere services (service handlers)
- Applikationskode skal ikke indeholde multithreading eller synkronisering

Løsning

Split services op i to dele
- Langvarige operationer, disse håndteres asynkront af in *iniaiator*
- Completion handlers, der processerer resultatet af operationerne når de er færdige

Alle asynkrone jobs returnerer et completion event
Dette completion event fungerer som et indication event til Reactoren
Proactoren dispatcher til en *completion handler* der behandler resultatet (*gemt i en completion event*)

Struktur (med UML klassediagram)

Proactor patternet har ni aktører
1. Handles (OS feature)
2. Asynchronous operations (OS feature)
3. Completion handler (interface)
4. Concrete completion handlers
5. Asynchronous operation processor (OS feature)
6. Completion event queue (OS feature)
7. Asynchronous event demultiplexer (OS feature)
8. Proactor
9. Initiator (= concrete completion handler)

Fordele Separation of concerns. Completion handlers skal kun koncentrere sig om funktionalitet

Completion handlers kan kodes uden fokus på concurrency (klares af OSets async funktioner)

Ulemper OSet skal supportere asynkrone operationer
Svært at debugge pga. decupling i tid

Benefits of Proactor in comparison with Reactor with a concurrency pattern

Det er komplekst at implementere concurrency pattern
Multithreading er svært at debugge og kan introducere fejl
Mere overhead i form af context switching
Benyttelse af OS funktioner er mere sikkert
Hvis OS ikke har de ønskede asynkrone kald er Reactor en god erstatning

4.2 Asynchronous Completion Token

Cookies er et eksempel på ACT pattern. De sendes med til klienten, og returneres uændret så serveren kan huske konteksten (*session*) fra før kaldet.

Problem

Når en asynkron service returnerer et svar skal dette demultiplexes og dispatches til en handler
Når et asynkront kald returnerer er kaldkonteksten glemt (*http kald til en server - cookies*)

Tre vigtige egenskaber
- Konteksten for kaldet skal kendes for at kunne demultiplexe svaret
- Kommunikations overhead skal holdes på et minimum
- Demultiplexing skal være effektivt

Løsning

Med asykrone kald sendes en ACT der indeholder information om demultiplexing
Når servicen returnerer sendes ACTen med uændret
Initiatoren kan nu bruge ACTen til at demultiplexe og dispatche til en handler

Struktur

Initiatoren laver en ACT der indeholder information om completion handleren
ACTen sendes med det asynkrone kald
ACTen returneres uændret sammen med resultatet af det asynkrone kald
ACTen bruges til at demultiplexe resultatet og kalde completion handleren

Fordele Simplicitet. Ellers skulle kaldkontekst skulle gemmes i datastruktur
Effektiv og fleksibel. Let at finde completion handler, kan pege på et hvilket som helst objekt

Ulemper Authentication. Hvis servicen ændrer på ACTen (*en bruger ændrer sin cookie*)

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

Ref. POSA2 p. 215-260 + slides
Ref. POSA2 p. 261-284 + slides

3. Acceptor/Connector Pattern

Benyttes af browsere til at connecte til servere asynkront, altså uden at blocke main tråden.

Problem

Vi vil gerne decouple initialisering af services fra servicen i sig selv
Dette tillader fx at benytte forskellige concurrency strategier i handlers uden at ændre acceptoren

Løsning

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

Struktur

1. Transport Endpoint (*sock acceptor*)
2. Transport Handle (*socket*)
3. Service Handler (*user implemented*)
4. Acceptor
5. Connector
6. Dispatcher (*Reactor eller Proactor*)

Fordele	Reusable code. Oprettelse af connections kan puttes i frameworks og genbruges
	Service handlers kan udvikles uden at tænke på connection
	Connection til mange peers over low latency WANs, kan benytte de asynkrone egenskaber
Ulemper	Ved små projekter kan det være overkill

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**

Ref. POSA2 p. 285-322 + slides

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

6.1 Leader/Followers

Analogi Taxaer holder i en kø (thread pool). Den første er leader. Når der kommer en kunde (event) tager den forreste taxa sig af denne. Den næste taxa bliver nu leader. Når den gamle leader kommer tilbage bliver den en follower.

Problem

Event driven applikationer der har mange threads der behandler events fra samme event sources
Half sync/half async kan have stort overhead. Vi søger derfor efter en mere effektiv løsning
Proactor er ikke altid en mulighed, da UNIX fx ikke understøtter asynkron I/O

Løsning

Lav en threadpool og udnævn en leader
Leaderen venter på nye events
Når der kommer et event promoter leaderen en ny leader og bliver en processing thread
Den nye leader lytter på nye events, mens den gamle leader dispatcher til event handlers
Virker nærmest som at have mange Reactors, så nyt IO ikke bliver blocket af lange operationer

Fordele
Ulemper

Fordele over half sync/half async

Half sync/half reactive kan have stort overhead pga. synkronisering, og context switches
OSer der understøtter asynkrone operationer effektivt kan med fordel benytte Proactor
Lettere at implementere og lettere at debugge

Fra *Consequences* delen

Forbedrer CPU cache affinity
Eliminerer *dynamic memory allocation* og buffer sharing mellem threads
Minimerer locking overhead da der ikke er interthread kommunikation
Da det ekstra queue lag er væk arbejdes kun med en prioritering (threadpoolens prioritet) fremfor flere, den asynkrone tråds prioritering køens prioritering og det synkrone lags prioritering
Det er mere forudsigeligt hvilke tråde der tager sig af hvilke events, dermed lettere debugging
Der skal ikke ske et kontekstswitch for hvert event

Half sync/half reactive er bedre når man gerne vil ændre på rækkefølgen af events (*brug køen*)
Half sync/half reactive er bedre når man gerne vil ændre prioriteten af forskellige events (*brug køen*)

Der står nogle gode pointer i See Also delen om Reactor, Proactor og half sync/half async

6.2 Half sync/half async

Analogi På en restaurant er der en der viser folk hen til et bord eller holder styr på en kø hvis der ikke er bode nok (asynkront lag). Derefter bliver en tjener givet til hvert bord (synkront lag), som godt kan vente på at gæsterne beslutter sig for deres bestilling.

Problem

Et system der både har synkrone og asynkrone operationer der skal samarbejde

Asynkrone operationer hæver effektivitet, men også kompleksitet
Synkrone operationer er simple, men ofte ineffektive

Målet er derfor at gøre det muligt at få asynkron effektivitet hvis det ønskes
Hvis det ikke ønskes skal man ikke bekymre sig om asynkrone ting overhovedet

Løsning

Opdel applikationen i 2 lag, et asynkront og et synkront
Imellem disse skal være en kø der benyttes til at udveksle data mellem lagene

Struktur

1. *Det synkrone lag* - kører i separate threads så de kan blocke
2. *Det asynkrone lag* - kører lower-level services. Må ikke blocke
3. *Queue laget* - Skal notificere når noget bliver lagt heri (*fx sockets i BSD UNIX*)
4. *External event sources* - network interfaces, disk controllers and end-user terminals

Det asynkrone lag får et interrupt eller et signal
Signalet behandles (uden blocking) og lægges i køen til videre behandling
Køen notificerer det synkrone lag
Signalet behandles af en tråd i det synkrone lag

Fordele Simplicitet og performance. Simpelt at kode synkrone services, og bibeholde performance fra asynkrone low level services.
Simpel kommunikation vha. queue laget. Der er ikke brug for at tilgå det andet lags memory

Ulemper Hvis queue laget bliver lagt mellem user og kernel space kan det give efficiency problemer
Svært at debugge

Tjek half sync/half reactive variant

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**

Ref. POSA2 p. 423-446 + slides
Ref. POSA2 p. 447-474 + slides

7. Interceptor Pattern

Fordele Åbner frameworket op så det kan customizes

Ulemper Svært at bestemme interception points. For få = ikke fleksibelt, for mange = ineffektivt + bloat
Fejl i interceptors kan ødelægge hele frameworket

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**

Ref. POSA2 Book p. 109-140 + slides

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**

Ref. Article: "The Many Faces of Publish/Subscribe" + slides

5. Architecture Trade Off Analysis Method (ATAM) and Architecture Documentation

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**

Ref. Article: "The Architecture Trade Off Method" + slides

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**

Ref: “JAWS: A Framework for HighperformanceWeb Servers” + slides

9. Component Configurator Pattern

Analogi Når en fodboldtræner udskifter spillere. Træneren er *component configurator* og spillerne er *components*. Kampen stoppes ikke fordi der sker en udskiftning. Det vedrører kun spillerne der skiftes ud.

Fordele 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

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

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

Ref. POSA2 Book p. 75-108 + slides