Pós-Graduação em Computação Distribuída e Ubíqua

INF628 - Engenharia de Software para Sistemas Distribuídos
Design Patterns para Sistemas Distribuídos

Sandro S. Andrade
sandroandrade@ifba.edu.br

Motivação

- Ampla expansão e popularização dos sistemas distribuídos
- Métodos e técnicas de desenvolvimento para sistemas single-threaded são inadequados
- Contínua reinvenção e redescoberta de conceitos e técnicas
- Design patterns para sistemas distribuídos:
  - Capturam soluções promissoras
  - Direcionam o foco do projetista para aspectos de mais alto nível

Desafios

- Benefícios do uso de sistemas distribuídos:
  - Colaboração e conectividade
  - Melhor desempenho, escalabilidade e tolerância a falhas
  - Menor custo

Desafios

- Problemas não encontrados ou menos problemáticos no desenvolvimento de sistemas single-threaded:
  - Estabelecimento de conexão e inicialização de serviço
  - Demultiplexação de eventos e direcionamento de eventos a manipuladores
  - IPC e protocolos de rede
  - Gerenciamento de armazenamento primário/secundário e cache
  - Configuração estática e dinâmica de componentes
  - Concorrência e sincronização

Desafios

- DESAFIO 1: Acesso e Configuração de Serviço
  - Componentes em sistemas distribuídos se comunicam via IPC, protocolos (ex: FTP, HTTP) ou mecanismos de middleware (ex: COM+, CORBA, EJB)
  - O acesso a serviço requer o uso de APIs de acesso a serviços de concorrência (ex: processos UNIX, threads POSIX, threads Win32, etc) e serviços de IPC (ex: sockets)

Desafios

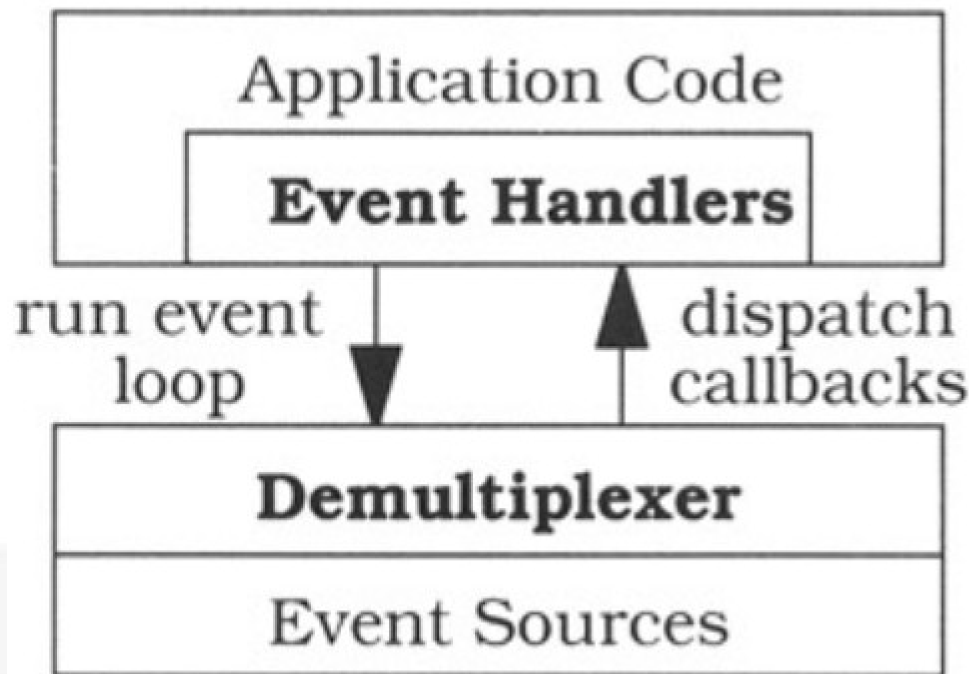
- DESAFIO 1: Acesso e Configuração de Serviço
  - Problemas: muitos detalhes de baixo nível, redescoberta contínua de abstrações de alto nível incompatíveis, alto potencial de erros, falta de portabilidade, alta curva de aprendizado, baixa escalabilidade em relação a complexidade
  - Evolução estática e dinâmica do serviço

Desafios

- DESAFIO 2: Tratamento de Eventos
  - Event-Driven x Self-Directed Flow of Control
  - Características:
    - O comportamento da aplicação é disparado por eventos internos ou externos que ocorrem assincronamente (ex: drivers de dispositivos, portas de I/O, sensores, teclado ou mouse, sinais, timers, etc)
    - A maioria dos eventos devem ser prontamente tratados para evitar congestionamento de CPU e para melhorar o tempo de resposta
    - Máquinas de estado podem ser necessárias para detectar transições ilegais. Aplicações event-driven possuem pouco controle sobre a ordem de ocorrência de eventos

Desafios

- DESAFIO 2: Tratamento de Eventos
  - É comum usar "Inversão de Controle" em aplicações event-driven:

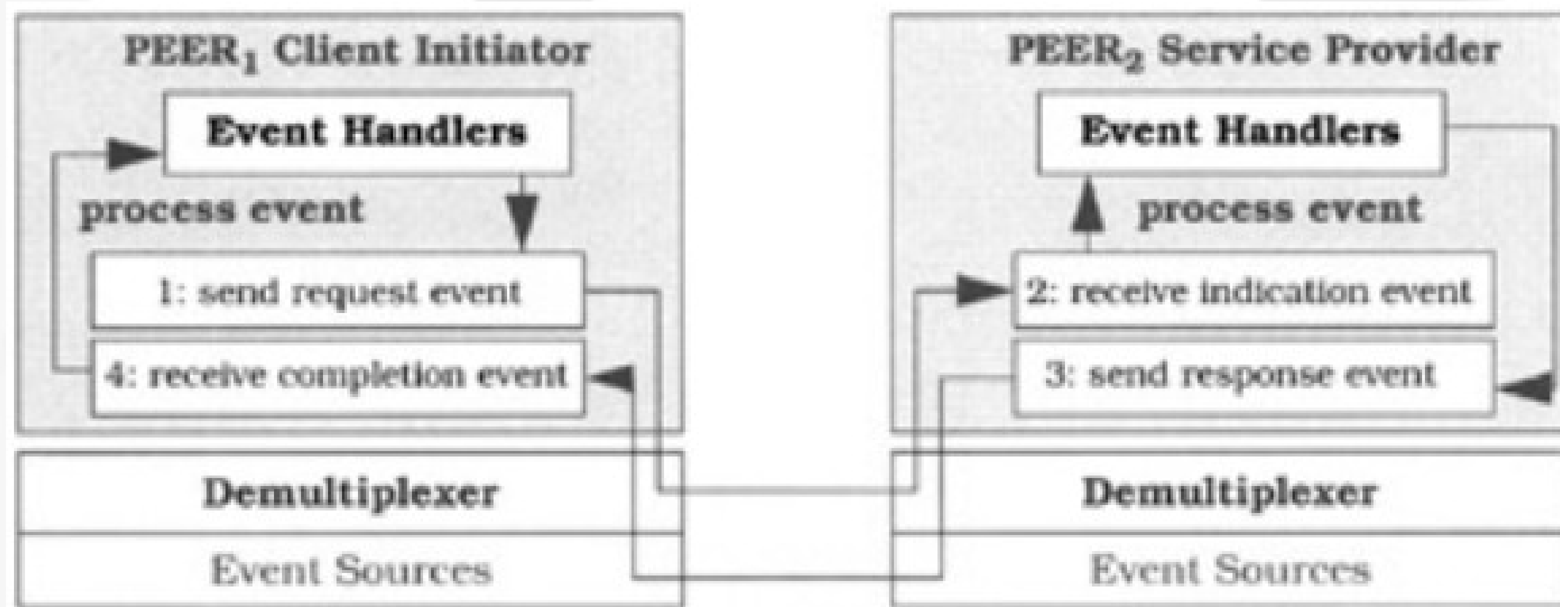


Desafios

- DESAFIO 2: Tratamento de Eventos
  - Benefícios da "Inversão de Controle":
    - Desenvolvedores se concentram na camada de funcionalidade da aplicação
    - As camadas de fontes de eventos e demultiplexação podem ser reutilizadas
    - O tratamento de diferentes eventos fica localizado em diferentes módulos (event handlers)

Desafios

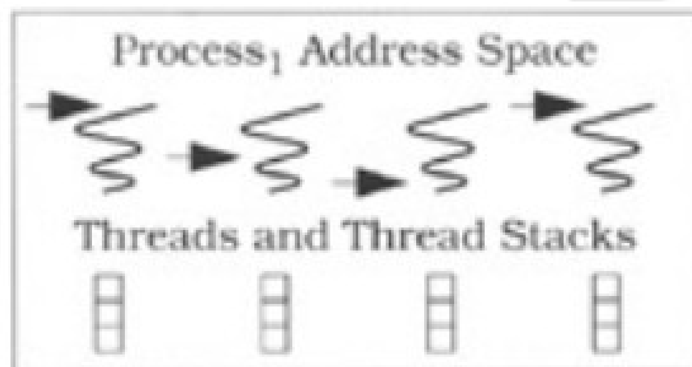
- DESAFIO 2: Tratamento de Eventos
  - Tratamento de eventos em rede:

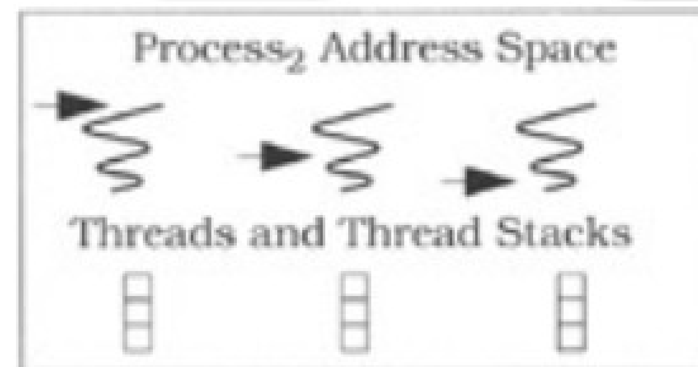


  - Clientes síncronos X Clientes assíncronos

Desafios

- DESAFIO 3: Concorrência
  - Concorrência = termo utilizado para se referir a uma família de políticas e mecanismos que permitem que uma ou mais threads/processos executem simultaneamente suas tarefas de processamento de serviços
  - Desenvolver aplicações concorrentes eficientes, previsíveis, escaláveis e robustas é uma tarefa difícil





Desafios

- DESAFIO 3: Concorrência
  - Problemas:
    - Como projetar uma arquitetura eficiente para um sistema concorrente de modo a minimizar trocas de contexto, sincronização e overhead de cópia/movimento de dados ?
    - Como projetar sistemas concorrentes complexos, com tarefas síncronas e assíncronas de processamento de serviços sem degradar a eficiência ?
    - Como selecionar primitivas de sincronização apropriadas de modo a maximizar o desempenho, prevenir condições de corrida e reduzir custos de manutenção ?
    - Como eliminar threads e locks desnecessários ou simplificar o gerenciamento de recursos sem comprometer a corretude, gerar deadlocks ou bloquear o progresso ?

Desafios

- DESAFIO 4: Sincronização:
  - Exclusão mútua
  - Problemas na implementação de exclusão mútua:
    - Alto potencial de erros
    - Inflexibilidade e Ineficiência
      - Pode-se usar locks mais fracos
      - Locks fortemente acoplados a outros concerns dificultam a modificação
- Outros desafios:
  - Dependability
  - Service Naming

Estudo de Caso

- Projeto de um servidor web concorrente:

HTTP Client

GET /index.html HTTP/1.0

<H1>POSA page</H1>...

HTTP Server
www.posa.uci.edu

GUI

HTML Parser

Requester

File Cache

Protocol Handlers

Event Dispatcher

Graphics Adapter

Transfer Protocol
e.g., HTTP 1.0

OS Kernel & Protocols

TCP/IP Network

OS Kernel & Protocols

Estudo de Caso

- Projeto de um servidor web concorrente:
  - Problemas:
    - Detalhes de programação de baixo-nível
    - Restrições de portabilidade
    - Adoção prematura de uma determinada configuração do servidor
    - Diversidade de alternativas de design:
      - Modelos de concorrência: thread-per-request, variações de thread pools
      - Modelos para demultiplexação de eventos: demultiplexação síncrona ou assíncrona
      - Modelos para caching de arquivos: LRU, LFU, etc
      - Protocolos para entrega de conteúdo: HTTP 1.0, 1.1, NG
    - É importante que a diversidade exista

Estudo de Caso

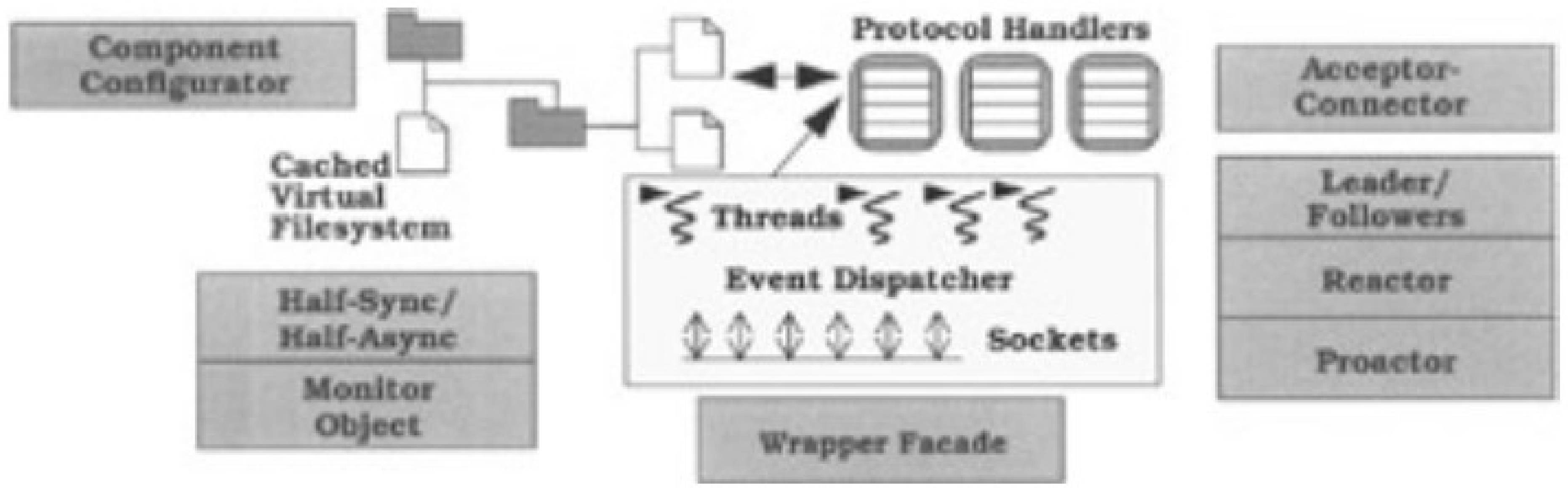
- Principais componentes do JAWS:

Protocol Handlers
Cached Virtual Filesystem
Threads
Event Dispatcher
Sockets

- O Event Dispatcher coordena a estratégia de demultiplexação com a estratégia de concorrência

Estudo de Caso

- 8 patterns são utilizados no JAWS ...



Estudo de Caso

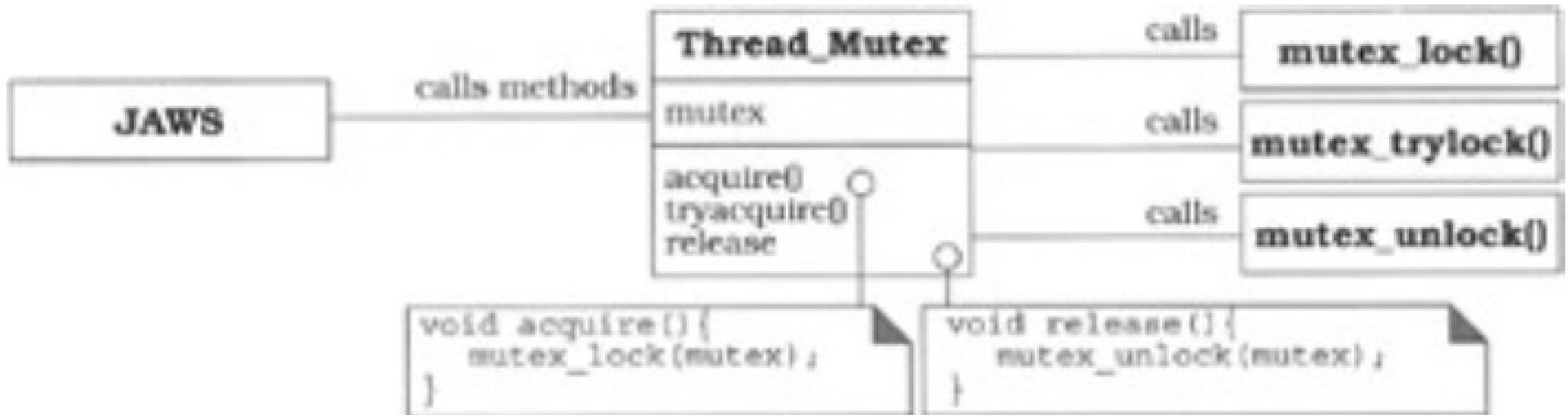
- ... para resolver os seguintes problemas:
  - Encapsular a API de baixo nível do SO
  - Desacoplar a demultiplexação de eventos e gerenciamento de conexão do processamento do protocolo
  - Melhorar o desempenho do servidor via multi-threading
  - Implementar uma fila sincronizada de requisições
  - Minimizar o overhead de threading do servidor
  - Usar I/O assíncrono de forma efetiva
  - Melhorar a configurabilidade do servidor
- Veremos:
  - Uma visão rápida de cada pattern
  - Os principais trade-offs

Estudo de Caso

- Encapsulando a API de baixo nível do SO:
  - Contexto:
    - O servidor web deve gerenciar uma variedade de serviços do SO (processos, threads, sockets, memória virtual, etc). A maioria dos SOs disponibilizam API de baixo nível, escritas em C
  - Problema:
    - É difícil escrever servidores portáveis e robustos usando a API de baixo nível do SO diretamente. As APIs são complicadas, sujeitas a erros e não portáveis
  - Solução:
    - Use o pattern Wrapper Facade: encapsula as funções e dados disponibilizados por uma API não-OO dentro de uma API OO mais concisa, robusta, portável, coesa e de fácil manutenção

Estudo de Caso

- Encapsulando a API de baixo nível do SO:
  - Uso do Wrapper Facade no JAWS:

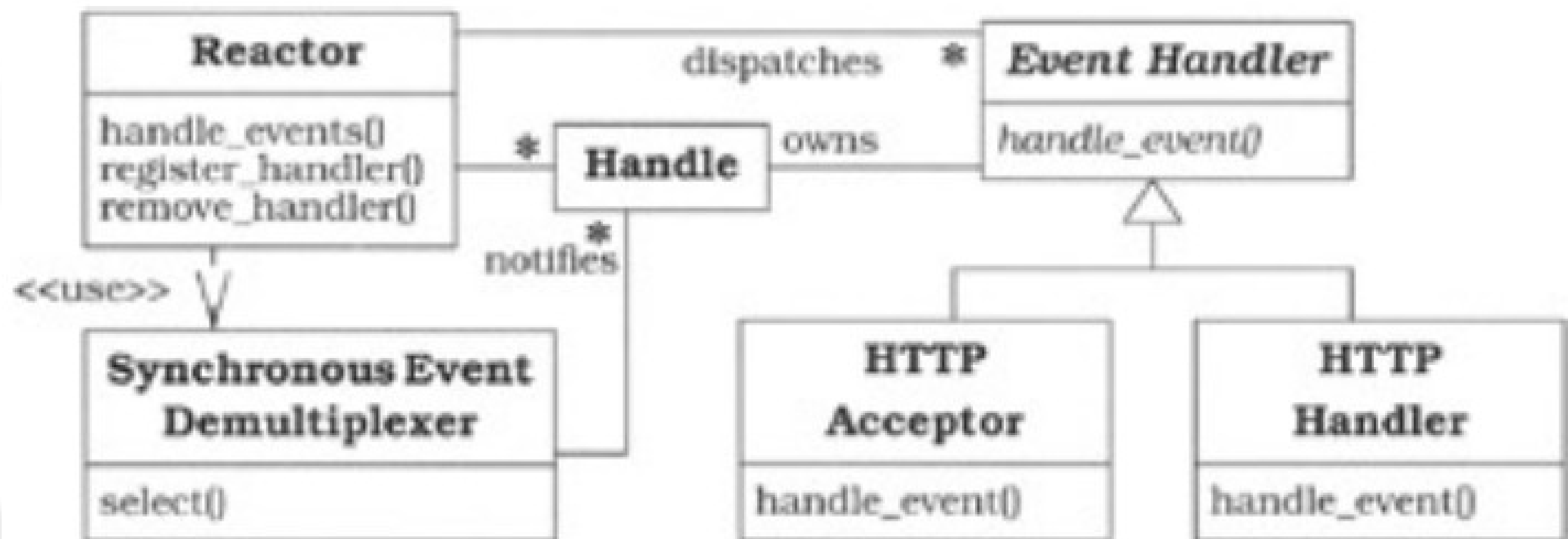


Estudo de Caso

- Desacoplando a demultiplexação de eventos e gerenciamento de conexão do processamento do protocolo
  - Contexto:
    - O servidor é acessado por muitos clientes, cada um contendo sua própria conexão. O servidor deve, portanto, demultiplexar e processar múltiplos tipos de eventos, originados em diferentes clientes
  - Problema:
    - Geralmente a demultiplexação do evento e gerenciamento da conexão está acoplado ao processamento do protocolo, dificultando o reuso
  - Solução:
    - Use os patterns Reactor e Acceptor-Connector para separar a demultiplexação e gerenciamento de conexão, respectivamente, do protocolo de processamento do HTTP

Estudo de Caso

- Desacoplando a demultiplexação e gerenciamento de conexão do processamento
  - Uso do Reactor no JAWS: processar múltiplos eventos síncronos de múltiplas fontes sem bloquear indefinidamente em uma fonte única de eventos

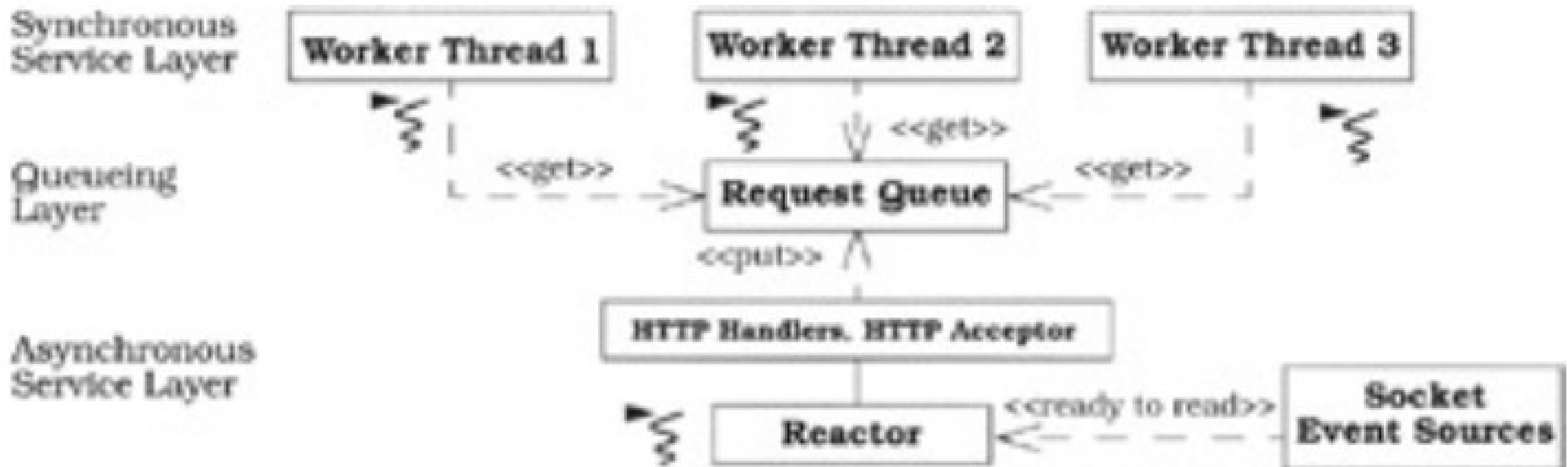


Estudo de Caso

- Melhorando o desempenho do servidor via multi-threading
  - Contexto:
    - O HTTP roda sobre o TCP, que utiliza controle de fluxo para garantir que emissores não produzam dados mais rápidos do que os receptores podem processar. O servidor deve escalar à medida que o número de clientes aumenta
  - Problema:
    - Processar os eventos GET reativamente em uma única thread não escala de forma eficiente pois há muito bloqueio para I/O. De forma semelhante, não é interessante bloquear por serviços de rede
  - Solução:
    - Use o pattern Half-Sync/Half-Async para processar diferentes requisições de forma concorrente em múltiplas threads

Estudo de Caso

- Melhorando o desempenho do servidor via multi-threading
  - Uso do Half-Sync/Half-Async no JAWS:

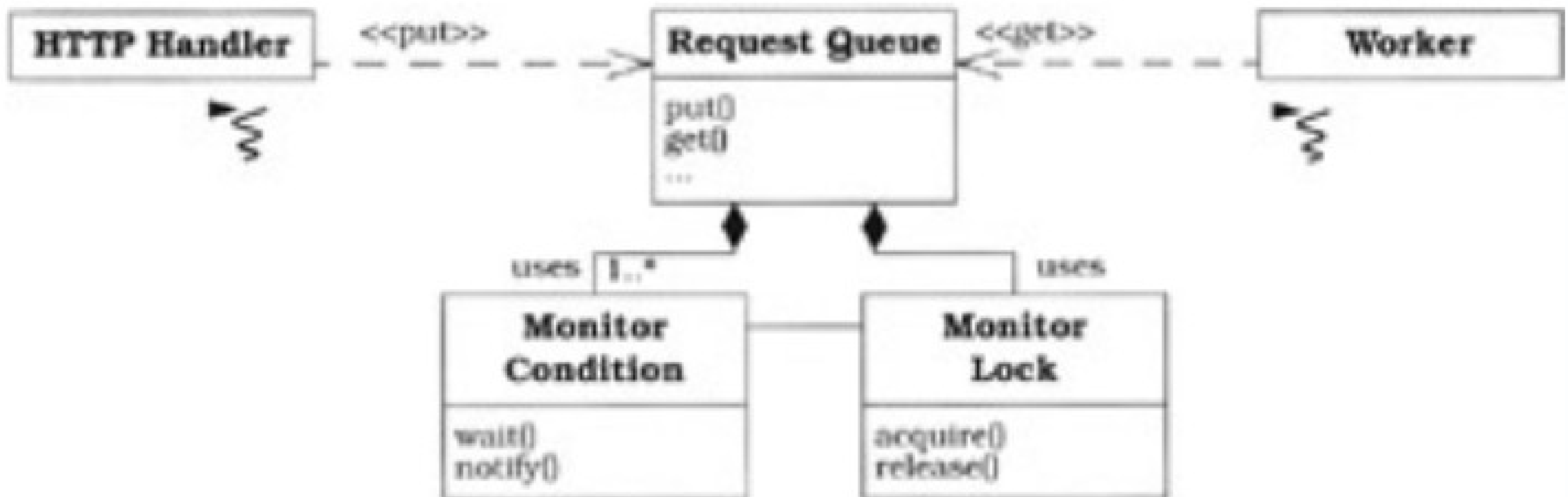


Estudo de Caso

- Implementando uma fila sincronizada de requisições
  - Contexto:
    - O centro do Half-Sync/Half-Async contém uma fila. O Reactor produz e as threads worker consomem requisições da fila
  - Problema:
    - Uma implementação ingênua irá produzir condições de corrida ou espera ocupada quando múltiplas threads inserem ou removem requisições
  - Solução:
    - Use o pattern Monitor Object para implementar uma fila sincronizada de requisições

Estudo de Caso

- Implementando uma fila sincronizada de requisições
  - Uso do Monitor Object no JAWS: um par de variáveis de condição para situações not-empty e not-full



Estudo de Caso

- Minimizando o overhead de threading do servidor
  - Contexto:
    - A cada solicitação de conexão, um novo endpoint é criado e uma thread worker é escalonada para tratar a requisição
  - Problema:
    - Embora escalável o Half-Sync/Half-Async não é a solução mais eficiente, pois requer alocação dinâmica, múltiplas operações de sincronização e troca de contexto
  - Solução:
    - Use o pattern Leader-Followers para minimizar o overhead de threading

Estudo de Caso

- Minimizando o overhead de threading do servidor
  - Uso do Leader-Followers no JAWS:

Worker Thread Pool
synchronizer
join()
promote_new_leader()
demultiplexes
*
Event Handler
handle_event()
Reactor
handle_events()
deactivate_handle()
reactivate_handle()
select()
*
Handle
uses
HTTP Acceptor
HTTP Handler

Estudo de Caso

- Usando I/O assíncrono de forma efetiva
  - Contexto:
    - Multi-thread síncrono pode não ser a melhor alternativa. Quando I/O assíncrono é suportado pelo SO melhores soluções podem ser projetadas
  - Problema:
    - Obter eficiência e escalabilidade através de I/O assíncrono é difícil de implementar (separação no tempo e espaço)
  - Solução:
    - Use o pattern Proactor para fazer uso efetivo de I/O assíncrono

Estudo de Caso

- Usando I/O assíncrono de forma efetiva
  - O Proactor trata eventos do tipo completion, resultados de operações assíncronas
  - O Reactor trata eventos do tipo indication, que disparam operações síncronas

Estudo de Caso

- Usando I/O assíncrono de forma efetiva
  - Uso do Proactor no JAWS:

Estudo de Caso

- Melhorando a configurabilidade do servidor
  - Contexto:
    - A implementação de um servidor web específico depende de aspectos estáticos (ex: #cpus) e dinâmicos (ex: workloads)
  - Problema:
    - Nenhuma configuração é ótima em todos os casos. Algumas decisões só podem ser eficientemente tomadas em runtime.
  - Solução:
    - Use o pattern Component Configurator para melhorar a configurabilidade do servidor web

Estudo de Caso

- Melhorando a configurabilidade do servidor
  - Uso do Component Configurator no JAWS

Component Repository

components *

Component

init()
fini()
suspend()
resume()
info()

<<contains>>

Component Configurator

LRU File Caching Strategy

LFU File Caching Strategy

Estudo de Caso

- Demonstração do JAWS
- Uma descrição mais aprofundada destes e de outros patterns está disponível no POSA, vol 2

Pós-Graduação em Computação Distribuída e Ubíqua

INF628 - Engenharia de Software para Sistemas Distribuídos
Design Patterns para Sistemas Distribuídos

Sandro S. Andrade
sandroandrade@ifba.edu.br