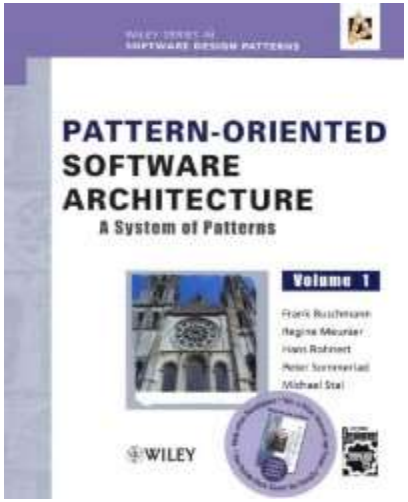
Pattern Oriented Software Architecture

Pattern Oriented Software Architecture



Pattern Oriented Software Architecture

Vol. 1 - A System of Patterns

2. Architectural Patterns

- **2.2 From Mud to Structure**
  - Layers, Pipes and Filters, Blackboard
- **2.3 Distributed Systems**
  - Broker
- **2.4 Interactive Systems**
  - Model-View-Controller, Present.-Abstraction-Control
- **2.5 Adaptable Systems**
  - Microkernel, Reflection

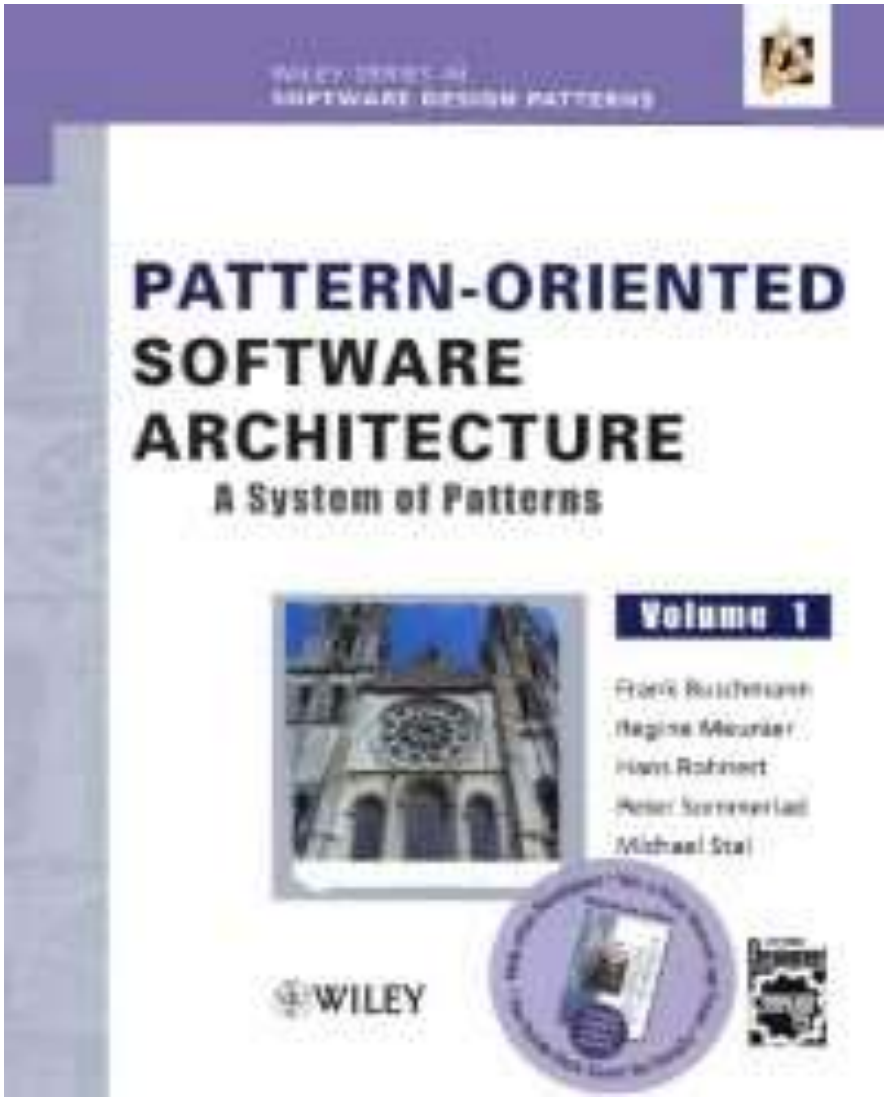
3. Design Patterns

- **3.2 Structural Decomposition**
  - Whole-Part
- **3.3 Organization of Work**
  - Master-Slave
- **3.4 Access Control**
  - Proxy
- **3.5 Management**
  - Command Processor, View Handler
- **3.6 Communication**
  - Forwarder-Receiver, Client-Dispatcher-Server
  - Publisher-Subscriber

Vol. 2 - Patterns for Concurrent and Networked Objects

- **2. Service Access and Configuration Patterns**
  - Wrapper Facade
  - Component Configurator
  - Interceptor
  - Extension Interface
- **3. Event Handling Patterns**
  - Reactor
  - Proactor
  - Asynchronous Completion Token
  - Acceptor-Connector
- **4. Synchronization Patterns**
  - Scoped Locking
  - Strategized Locking
  - Thread-Safe Interface
- **5. Concurrency Patterns**
  - Active Object
  - Monitor Object
  - Half-Sync/Half-Async
  - Leader/Followers
  - Thread-Specific Storage

Vol. 1 - A System of Patterns



2.4 Interaktivní systémy
2.5 Adaptovatelné systémy

- **Interaktivní systémy**
  - Popis problému
  - Vzor MVC
  - Vzor PAC
- **Adaptovatelné systémy**
  - Popis problému
  - Vzor Microkernel
  - Vzor Reflection

Interaktivní systémy

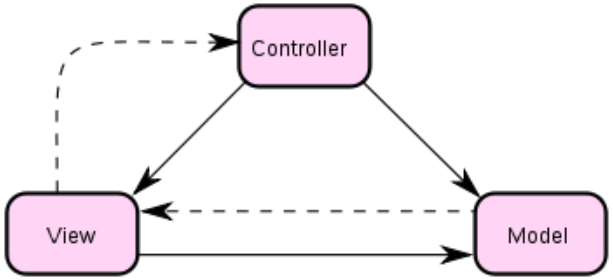
Vzory MVC a PAC

Interaktivní systémy – popis problému

- **Chování řízeno vstupy uživatele**
  - GUI
  - Webové aplikace
- **Součásti systému**
  - Funkční jádro
  - Prezentační vrstva
  - Vstupy uživatele
- **Požadované vlastnosti**
  - Nezávislost součástí
  - Možnost používání různých front-endů

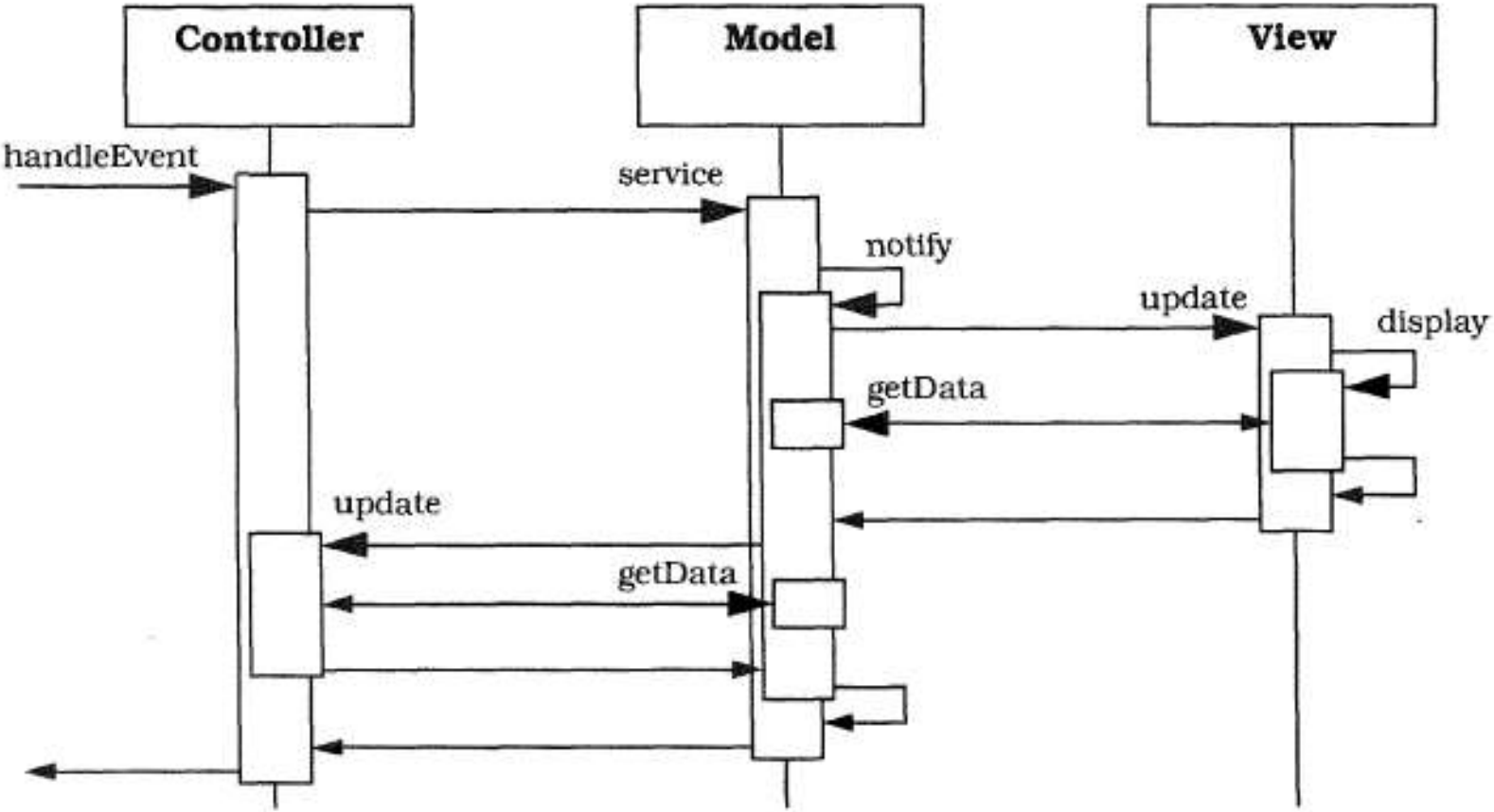
MVC

- **Model-View-Controller**



- **Model**
  - Data + logika aplikace
- **View**
  - Pohled na model prezentovaný uživateli
- **Controller**
  - Přijímá vstupy uživatele a zajišťuje následné změny v modelu a pohledech
- **Při vytváření se všechny pohledy a controllery zaregistrují u modelu**
  - NV Observer

MVC – chování

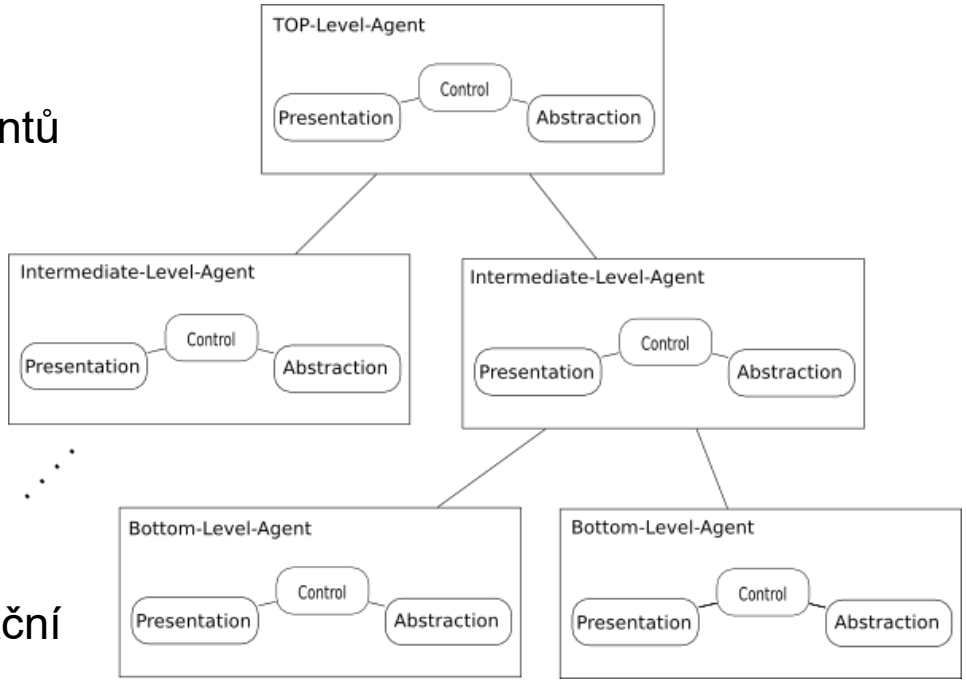

Controller
Model
View
handleEvent
service
notify
update
display
getData
update
getData

MVC – varianty, použití

- **Document – view**
  - MFC
- **Implementace MVC**
  - Java EE – model, JavaServer Page, servlet
  - Swing
  - Spring MVC
  - Zend Framework
  - ASP.NET MVC Framework

PAC

- **Presentation-abstraction-control**
  - Stromová hierarchie kooperujících agentů
  - Prezentační a abstrakční části agentů zcela oddělené
- **Abstrakční část**
  - Data a aplikační logika
- **Prezentační část**
  - Zobrazení dat uživateli
- **Controller**
  - Komunikace s ostatními agenty
  - Komunikace prezentační a abstrakční části
  - Zpracování vstupů uživatele
- **Příklad:** řízení letového provozu



Zdroj: wikipedia

PAC - vlastnosti

- **Dynamická tvorba hierarchie**
- **Distribuované prostředí**
- **Je třeba najít kompromis mezi jemností dekompozice systému a efektivitou komunikace mezi agenty**
  - Komunikace probíhá jen s přímo propojenými agenty
  - Ještě závažnější, pokud jsou agenti distribuovaní
- **V praxi se používá mnohem méně než např. MVC**
  - Komplexita návrhu
  - Příliš velká režie spojená s komunikací
  - Drupal – redakční systém

Adaptovatelné systémy

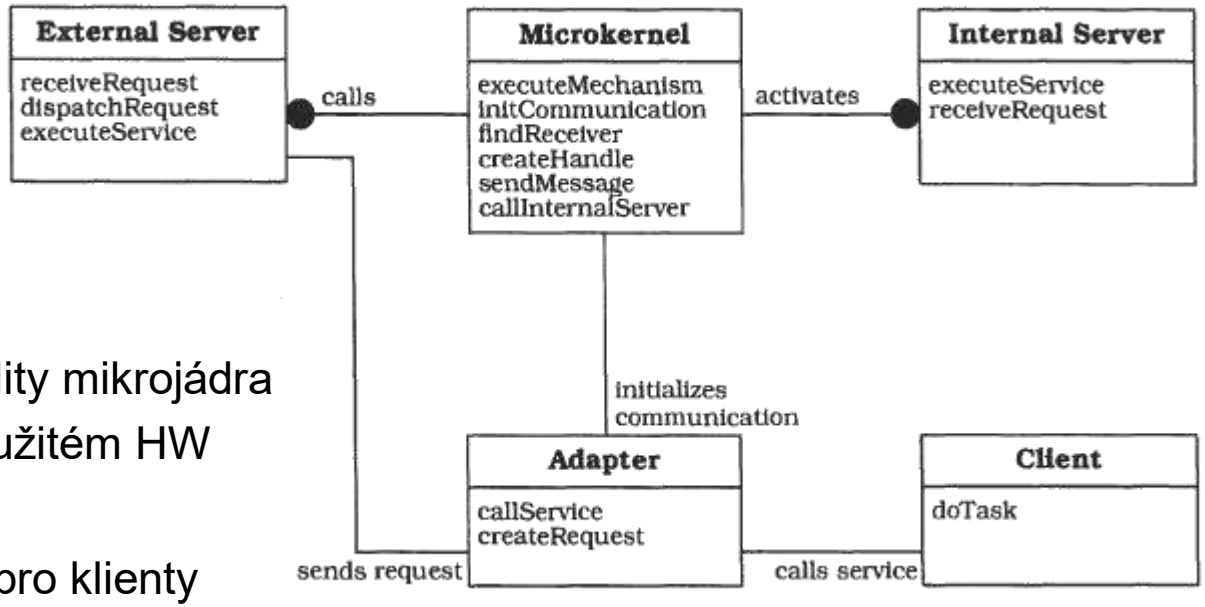
Vzory Microkernel a Reflection

Adaptovatelné systémy – popis problému

- **Systémy se obvykle v průběhu času vyvíjí**
  - Nová funkcionalita
  - Podpora novějších verzí OS
  - Přechod na novou verzi GUI, knihoven,...
  - Podpora nového hardwaru
- **Různé požadavky uživatelů**
- **Požadované vlastnosti**
  - Relativně snadné modifikace a rozšíření aplikace
  - Neuvažujeme modifikace základního návrhu systému

Microkernel

- **Použití zejména pro operační systémy**
- **Mikrojádro**
  - Základní služby
  - Komunikace
  - Správa zdrojů
- **Interní server**
  - Rozšíření funkcionality mikrojádra
  - Často závislá na použitém HW
- **Externí server**
  - Konkrétní prostředí pro klienty
  - Abstrakce nad vrstvou mikrojádra a interních serverů
- **Klient**
  - Svázaný s konkrétním externím serverem
- **Adaptér**
  - Přenositelné rozhraní pro klienty



Microkernel – použití, výhody, nevýhody

- **Operační systémy**
  - Windows NT
  - Chorus
- **Přidání nové funkcionality**
  - Interní server
- **Nový pohled na logiku implementovanou mikrojádrem**
  - Externí server
- **Použití v distribuovaném prostředí**
- **Nevýhody**
  - Komplexní návrh aplikace
  - Rychlost v porovnání s monolitickými aplikacemi

Reflection

- **Idea: programu umožníme přístup ke své vlastní struktuře**
  - 2 vrstvy – metavrstva a skutečný kód aplikace
- **Vrstva metaobjektů**
  - Aspekty systému, u kterých požadujeme možnost změny
  - Strukturální metaobjekty
  - Funkční metaobjekty
  - Informace o stavu vrstvy s aplikačním kódem
- **Příklady funkcionality poskytované metavrstvou**
  - Typové informace
  - Volací konvence
  - Vyhledávání komponent systému
  - Komunikace mezi komponentami systému
  - Transakční protokoly
- **Vrstva s aplikačním kódem**
  - Vlastní logika aplikace
  - Činnosti, u kterých předpokládáme možnost změny, vykonává pomocí metavrstvy

Reflection

- **MOP – metaobject protocol**
  - Provádí změny v metavrstvě
  - Poskytuje interface umožňující vyvolání změn
  - Přístupný aplikační vrstvě a/nebo systémovému administrátorovi
- **Modifikace některého aspektu aplikace**
  - Specifikace nového metaobjektu
  - Úprava stávajícího metaobjektu
  - Aktualizace v místech použití v metavrstvě
  - Může vyžadovat přeložení/přilinkování k aplikaci
- **Reflection – výhody**
  - Při změnách neměníme kód, pouze voláme metody MOP
  - Používání metod MOP je typicky snadné
- **Reflection – nevýhody**
  - Nutná podpora programovacího jazyka
  - Nižší efektivita
  - Modifikace mohou být destruktivní

3.3 Master-Slave

3.4 Command processor

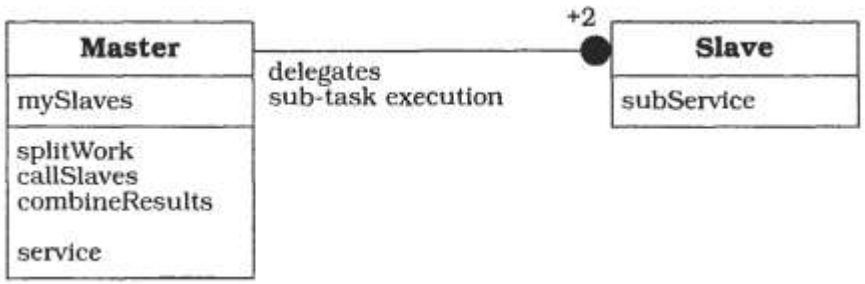
3.5 View handler

Master-Slave

- **Účel**
  - Rozdělení velké úlohy na více menších podúloh a výpočet finálního výsledku
  - Definuje způsob rozdělení úlohy, rozhraní pro komponenty Master a Slave
  - Aplikace principu „Rozděl a panuj“
- **Kategorie**
  - Organization of work – Organizace práce
- **Využití**
  - Paralelní programování
    - Každá podúloha se vykoná paralelně jako samostatné vlákno/program
  - Odolnost vůči chybám
    - N-modular redundancy
- **Struktura**
  - Podúlohy – obvykle identické algoritmy
    - Někdy vhodné použít najednou různé algoritmy, které však řeší stejnou úlohu

Master-Slave

- **Struktura**



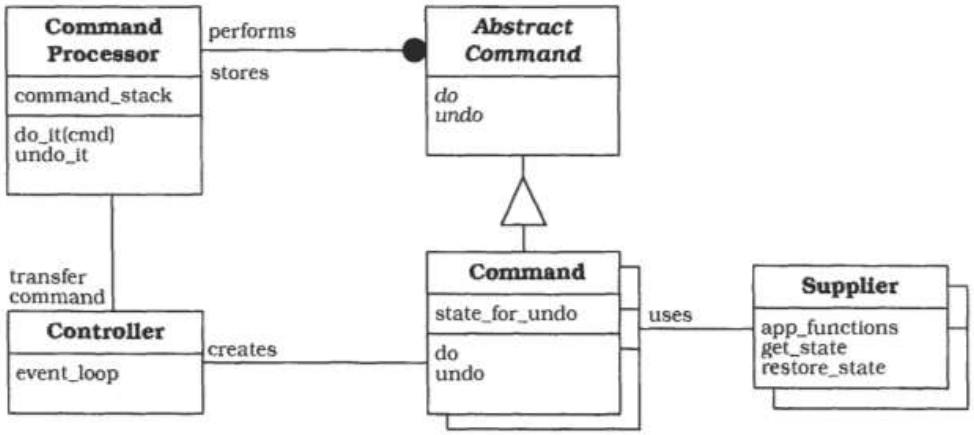
- **Návrh použití**
  - Klient si vyžáda službu od mastera
  - Master rozdělí úlohu na podúlohy
  - Master deleguje vykonání podúloh na „otroky“, spustí je a čeká na výsledky
  - „Otroci“ vypočtou výsledky a odešlou je
  - Master zpracuje obdržené výsledky od „otroků“ do finálního výsledku
  - Master odešle výsledek klientovi

Command processor

- **Účel**
  - Zapouzdření složitějších úkonů do jednoho objektu, se kterým lze snadno pracovat
- **Kategorie**
  - Management – zpracování objektů/služeb/komponent obdobného druhu
- **Související vzory**
  - Command processor staví na vzoru Command
    - Navíc se stará i o správu command objektů
  - Intepreter
- **Využití**
  - Tvorba flexibilního UI
  - Undo-Redo operace

Command processor – undo/redo

- Struktura



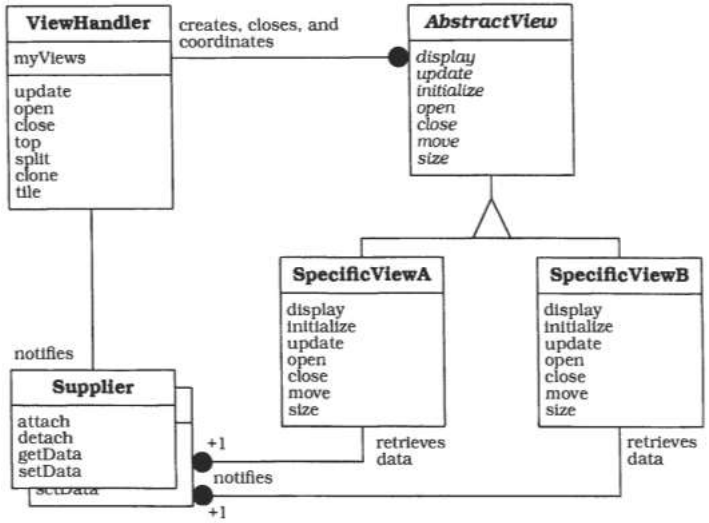
- Návrh použití
  - Controller dostane požadavek na akci a k němu přiřadí konkrétní Command
  - Controller předá objet Command na Command processor
  - Command processor spustí Command a zařadí ho do zásobníku provedených akcí
  - Command se provede
  - Klient zavolá příkaz „Undo” - Controller zavolá „Undo“ přímo na Command processoru
    - Obdobně třeba pro Redo
  - Command processor provede Undo na Commandu z vrcholu zásobníku provedených akcí a smaže tento Command ze zásobníku

View handler

- **Účel**
  - Poskytuje uživateli různé pohledy nad stejnými daty
  - Úpravy, synchronizace a management jednotlivých pohledů
  - Jednotlivé pohledy by na sobě měly být navzájem nezávislé
- **Kategorie**
  - Management
- **Související vzory**
  - View handler je vlastně
    - Abstract factory – vytváří pro klienta pohledy
    - Mediator – sám se stará o koordinaci mezi pohledy
- **Využití**
  - Oddělení prezentační vrstvy od funkční
    - MVC
  - Textový editor
    - Více oken pro jeden dokument najednou

View handler

- Struktura



- Kroky
  - Klient požádá View handler o vytvoření nového pohledu
  - Viev handler inicializuje nový pohled a předá mu jeho Supplier
  - Nový pohled se zařadí do seznamu pohledů a otevře ho
  - View načte data ze svého Supplier a zobrazí je
  - Opakuje pro další pohledy
  - Pohled při každé změně předá nová data přes Supplier
    - Viev handler zavolá update pro všechny pohledy a ty se aktualizují

3.6 Komunikační návrhové vzory

Forwarder-Receiver

Client-Dispatcher-Server

Publisher-Subscriber

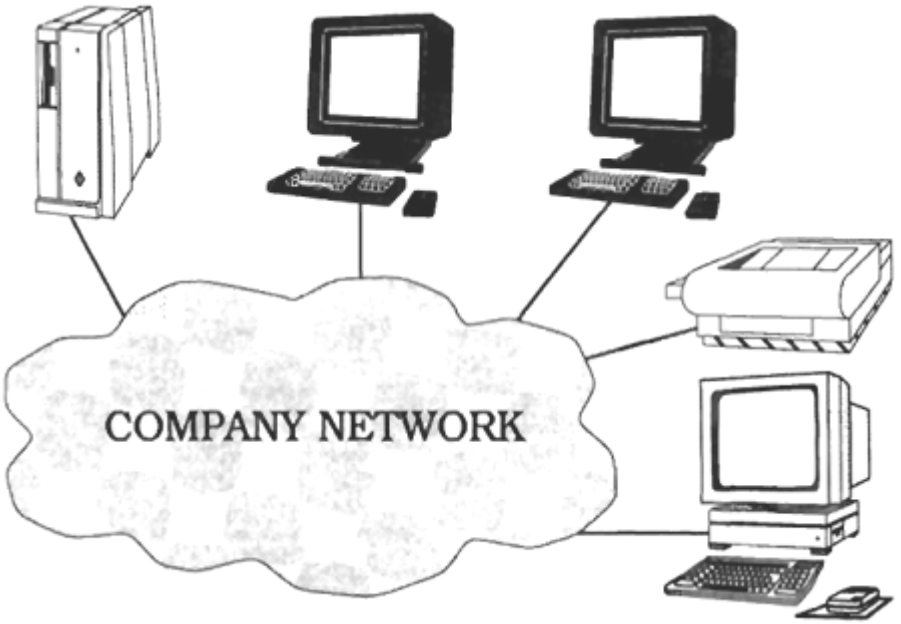
Komunikační návrhové vzory

- **Cíl přednášky**
  - **Obeznámení se skupinou komunikačních NV**
  - **Prezentace hlavních myšlenek**
  - **Motivace pro další (samo)studium**
  - **Cílem není podrobný výklad všech možností a implementace**

- **Obsah**
  - **Příklad**
  - **Forwarder-Receiver**
  - **Client-Dispatcher-Server**
  - **Publisher-Subscriber**
    - **Varianty**
  - **Shrnutí**

Komunikační návrhové vzory

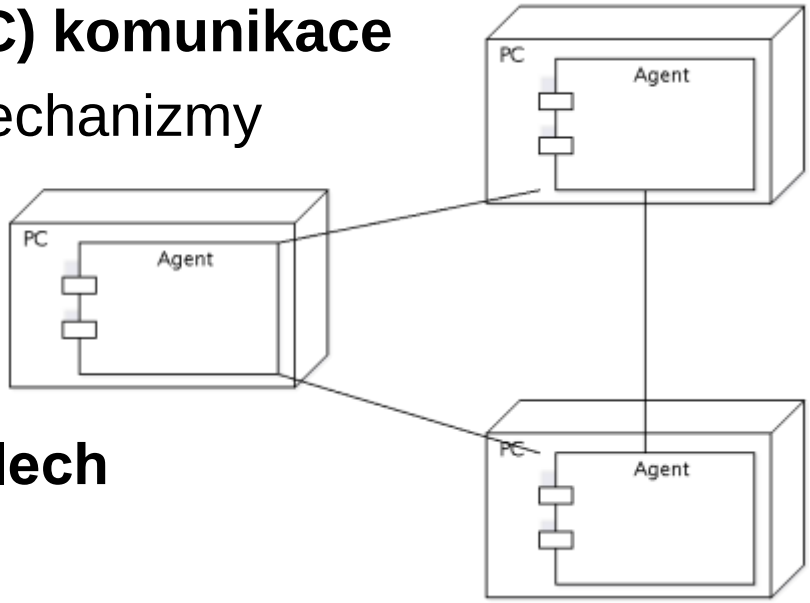
- **Příklad**
  - **Systém pro správu počítačové sítě**
    - Monitorování událostí a zdrojů
    - Konfigurace sítě
    - Peer to peer komunikace mezi uzly sítě
    - Podpora různého hardware



Zdroj: PATTERN-ORIENTED SOFTWARE ARCHITECTURE, A System of Patterns, p. 307

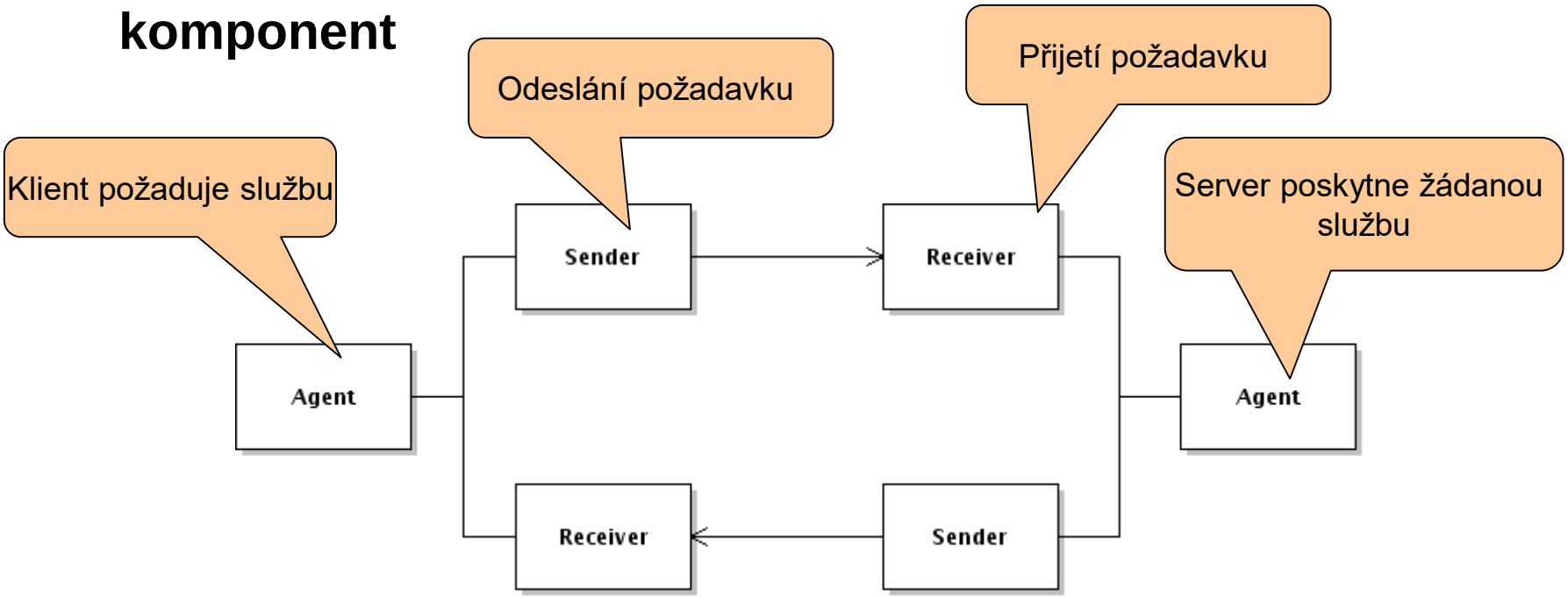
Komunikační návrhové vzory

- **Realizace**
  - **Vytvoření agentů běžících na uzlech v síti**
  - **Low-level mezi-procesová (IPC) komunikace**
    - Efektivnější než high-level mechanizmy
- **Problémy**
  - **Závislost na OS**
  - **Závislost na síťových protokolech**
  - **Omezení přenositelnosti**
  - **Omezená podpora heterogenních prostředí**
  - **Pozdější změna IPC mechanizmu**



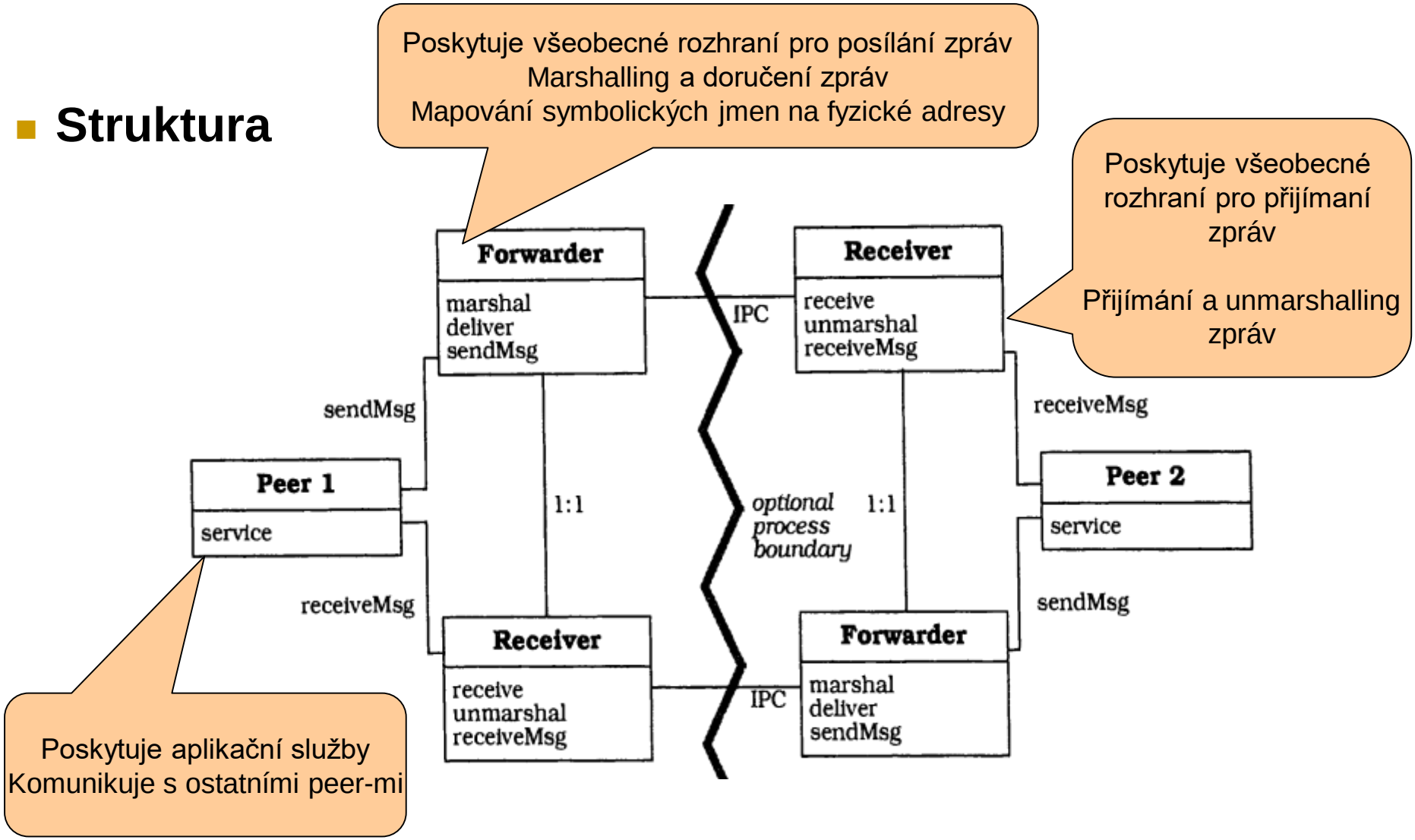
Forwarder-Receiver

- **Řešení**
  - **Spolupráce mezi agenty**
  - **Agent figuruje jako klient i jako server, nebo oboje**
  - **Zapouzdření IPC mechanizmu do samostatných komponent**



Forwarder-Receiver

- **Struktura**



Zdroj: PATTERN-ORIENTED SOFTWARE ARCHITECTURE, A System of Patterns, p. 311

Forwarder-Receiver

- **Použití návrhového vzoru**
  - **Kontext**
    - Peer to peer komunikace
  - **Řešený problém**
    - Vyměnitelnost komunikačního mechanizmu
    - Spolupráce komponent na základě symbolických jmen
    - Komunikace bez vlivu na výkon aplikace
  - **Řešení**
    - Ukrytí komunikačního mechanizmu mimo Peer-ů: vytvoření forwarder a receiver komponent
  - **Důsledky použití**
    - Efektivní mezi-procesová komunikace
    - Zapouzdření IPC prostředků
    - Žádná podpora pro rekonfiguraci komponent

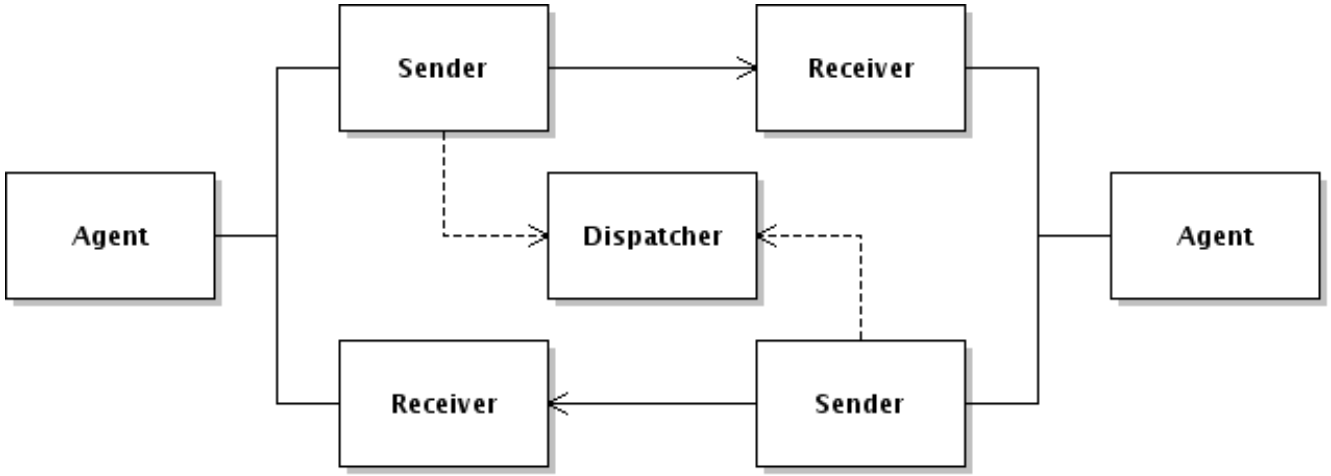
Client-Dispatcher-Server

- **Problémy se systémem pro správu počítačové sítě**
  - **Vyřešení předcházejících problémů**
    - Závislost na OS, omezená přenositelnost, změna IPC mechanizmu apod.
  - **Zavlečení nového problému**
    - Problémy s přizpůsobením se změnám distribuce Peer komponent za běhu
- **Řešení**
  - **Vytvoření mezivrstvy mezi komunikujícími Peer-mi, resp. mezi Forwarderem a Receiverem**
    - Dispatcher komponenta

Client-Dispatcher-Server

- **Úlohy Dispatcher komponenty v systému pro správu počítačové sítě**
  - **Implementace name service služby**
    - Transparentní lokalizace Peer-ů

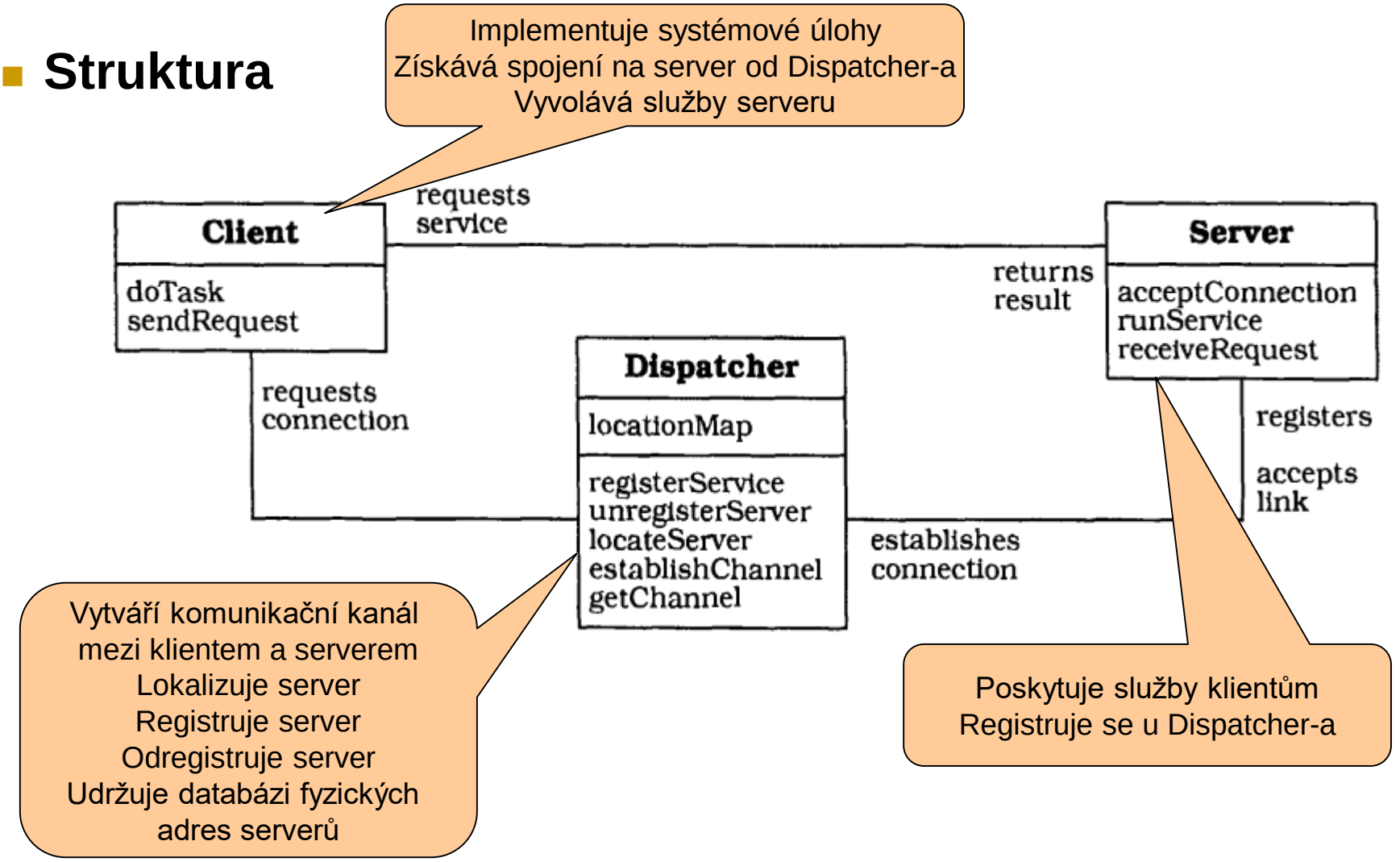
Client-Dispatcher-Server with communication managed by clients



- **Jiné úlohy Dispatcher-a**
  - **Navázání spojení, (od)registrace serverů a pod.**

Client-Dispatcher-Server

- **Struktura**



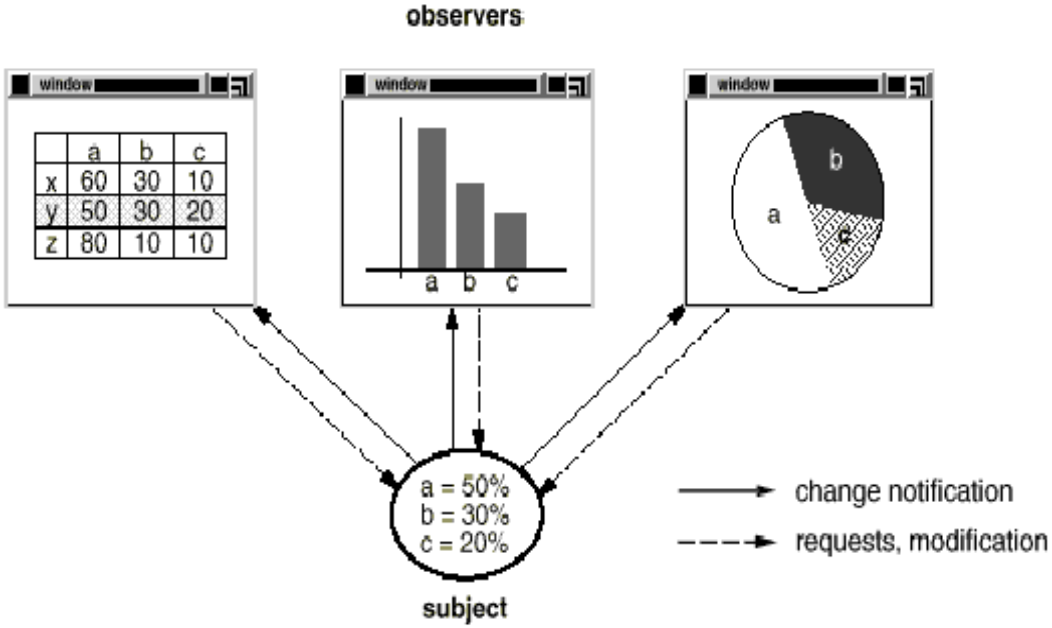
Zdroj: PATTERN-ORIENTED SOFTWARE ARCHITECTURE, A System of Patterns, p. 326

Client-Dispatcher-Server

- **Použití návrhového vzoru**
  - **Kontext**
    - Systém integrující množinu distribuovaných serverů
  - **Řešený problém**
    - Použití služeb bez závislosti od jejich umístnění
    - Oddělení implementace konzumenta služby od navazování spojení se službou
  - **Řešení**
    - Vytvoření vrstvy mezi klientem a serverem poskytující transparentní vyhledávání služeb a navazování spojení
    - Odkazování se na server podle jména
  - **Důsledky použití**
    - Vyměnitelnost serverů, transparentní umístnění a přesun serverů, rekonfigurace, odolnost vůči poruchám, neefektivní navazování spojení, citlivost na změny dispatcher-a

Publisher-Subscriber

- **Aliasy**
  - **Observer, Dependents**
    - Součást základních návrhových vzorů (behavioral patterns)
- **Připomenutí**
  - **Odstranění těsné vazby**
  - **One to many závislost**
  - **Subject (Publisher), Observer (Subscriber)**
  - **Notifikace, aktualizace**



Zdroj: MFF UK, předmět Návrhové vzory, prezentace Obeserver pattern, Miroslav Baron

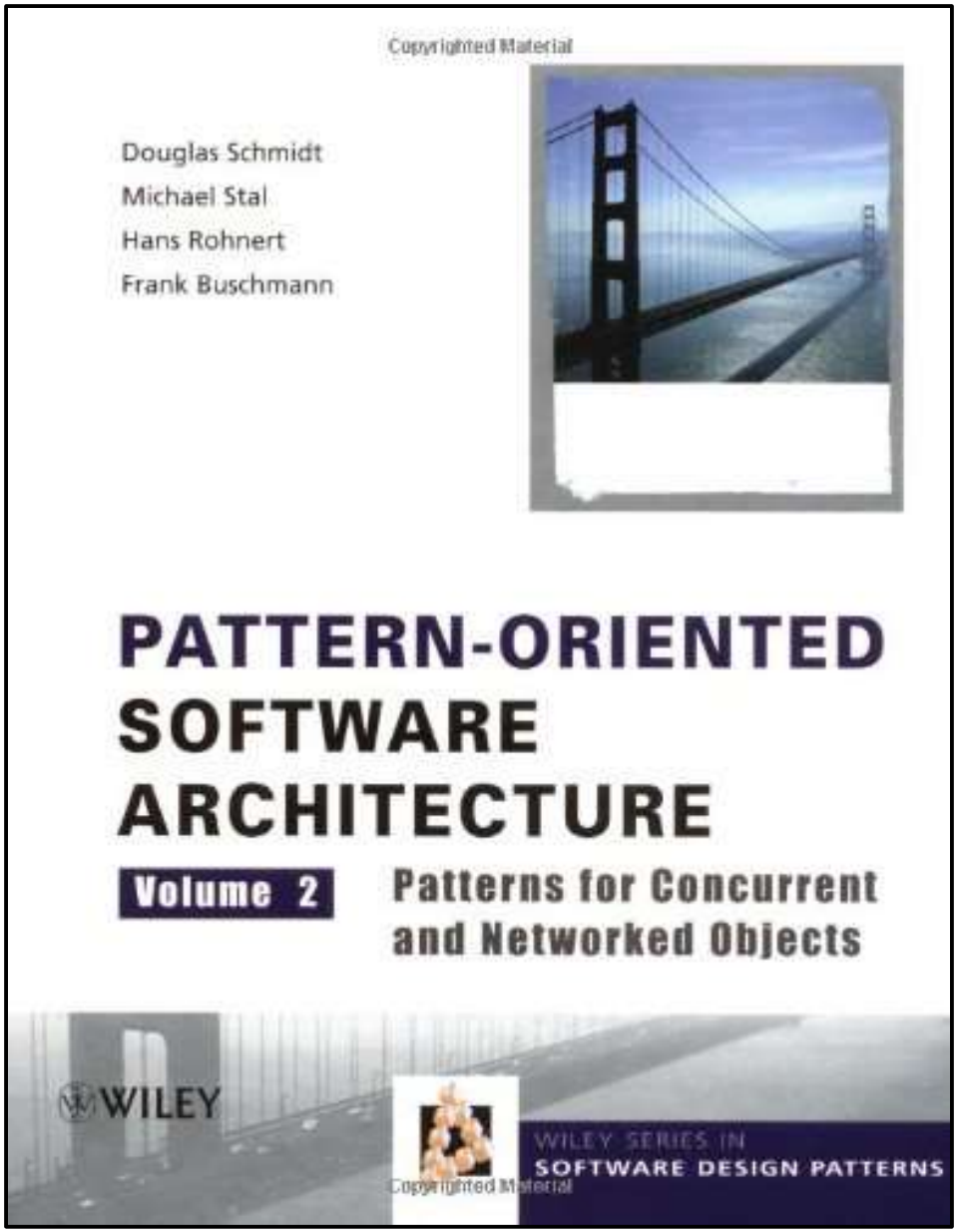
Publisher-Subscriber

- **Varianty**
  - **Gatekeeper**
    - Distribuované systémy
    - Vzdálená notifikace
  - **Event Channel**
    - Distribuované systémy
    - Silné oddělení Publisher-a a Subscriber-a
    - Kanál pro zachycení událostí mezi Publisher-om a Subscriber-om, Proxy Publisher/Subscriber
  - **Producer-Consumer**
    - Oddělení Producer-a a Consumer-a vyrovnávací pamětí
    - Obvykle vztah 1:1

Komunikační návrhové vzory

- **Shrnutí**
  - **Forwarder-Receiver**
    - Transparentní mezi-procesová komunikace
    - Peer to peer model
    - Oddělení Peer-ů od komunikačních mechanizmů
  - **Client-Dispatcher-Server**
    - Zavádí vrstvu mezi klientem a serverem – Dispatcher
    - Transparentní vyhledávání serverů podle jmen
    - Navazování spojení
  - **Publisher-Subscriber**
    - Udržuje stav spolupracujících komponent synchronizovaný
    - Jednosměrná propagace změn

Vol. 2 - Concurrent and Networked Objects



2. Service access and configuration patterns

- Wrapper facade
- Component configurator
- Interceptor
- Extension interface

Wrapper facade

- **Fasáda zakrývá (komplexní) vztahy mezi objekty**
- **Wrapper fasáda zakrývá nízkoúrovňová rozhraní**
- **Low-level API:**
  - Nízká úroveň abstrakce
  - Nekompatibilita mezi platformami
  - Náročné na užívání
  - Náchylné na chyby

Abstrakce, zapouzdření, sjednocení

- **Zvýšit úroveň abstrakce**
- **Funkce převést na třídy a rozhraní**
- **Jednotné rozhraní na všech platformách**
- **Zadní dvířka**

Jak na to

- **Seskupit funkce pracující se stejnými datovými strukturami**
- **Identifikovat průnik funkcionalit na podporovaných platformách**
- **Skupiny + průniky = třídy wrapper fasády**
- **Ponechat přístup k nízkorúrovňovým datovým strukturám (handle, ukazatel)**

Component configurator

- **Motivace – konfigurovatelnost aplikace za jejího běhu**
- **Statická logika aplikací**
- **Změna implementace znamená rekompilaci celého programu**
- **Změna konfigurace znamená restart celé aplikace**
- **Různá prostředí – různé konfigurace**

Komponenta s životním cyklem

- **Využití služeb OS nebo runtime platformy**
- **Umožnění dynamického připojení komponent k aplikaci**
- **Komponenta s životním cyklem – inicializace, zastavení, opětovné spuštění, deinicializace**
- **Změna stavu komponenty neovlivní celou aplikaci**

Jak na to

- **Vytvořit rozhraní pro připojení komponent**
- **Repozitář komponent – udržuje jejich seznam**
- **Konfigurátor – správa životního cyklu**
- **Náročné na implementaci**
- **Existující řešení:**
  - OSGi framework (základ Eclipse)
  - Windows NT service control manager

Interceptor

- **Motivace – rozšiřitelnost systému o nové služby**
- **Neznalost všech potřeb klienta v době vývoje**
- **Řešení:**
  - Monolitický systém obsahující vše
  - Úplná otevřenost systému
- **Nevýhody:**
  - Velké a neflexibilní
  - Nebezpečné

Události a callbacky

- **Přidávání služeb do systému na přesně určených místech**
- **Místo = událost (příjem zprávy, spracování dotazu, …)**
- **Služba = callback (logování, kryptování, …)**
- **Částečné otevření a zpřístupnění vnitřní funkcionality systému**

Jak na to

- **Změna systému na stavový automat**
- **Přechod mezi stavy – potenciální místo pro událost**
- **Definování rozhraní pro callback zpracování události**
- **Vytvoření kontextu pro událost:**
  - Informace o události
  - Modifikování chování systému
- **Vztah událost – callback je 1 ku n**
- **EJB, CORBA components, COM+ (proxy varianta)**

Extension interface

- **Motivace – evoluce rozhraní komponent**
- **V počátcích vývoje těžké předpovědět, jak a kam se systém rozroste**
- **Přidávání funkcionalit – přehuštění rozhraní metodami**
- **Těžké udržovat zpětnou kompatibilitu**

Mám více tváří

- **Cíl – rozdělit jedno velké rozhraní na vícero malých**
- **Jedna komponenta implementuje několik rozhraní (tváří)**
- **Výběr rozhraní pro přístup ke komponentě je na uživateli**
- **Jednotný přístup ke všem rozhraním**

Jak na to

- **Vytvořit jedno základní všeobecné (root) rozhraní:**
  - Implementuje každá komponenta
  - Poskytuje přístup k ostatním rozhraním
  - Může obsahovat společnou funkcionalitu všech komponent
- **Každé rozhraní má jedinečný identifikátor a rozšiřuje root rozhraní**
- **Komponenta implemetuje všechna rozhraní, která podporuje**
- **Klient nemá přímý přístup ke komponentě**
- **COM, CORBA component model**

3. Event Handling Patterns

Úvod

Účel:

- **Poskytují způsob jak inicializovat, přijmout, demultiplexovat, dispatchovat a spracovat události (eventy) v síťově orientovaných systémech**

Související návrhové vzory:

- **Reactor, Proactor, Asynchronous Completion Token a Acceptor-Connector**

Reactor - úvodem

- **Také známý jako Dispatcher nebo Notifier**
- **Architekturální návrhový vzor poskytující event-driven aplikacím vyvolávat požadavky jednoho či více klientů podle Hollywood principu**
  - „Don't call us, we'll call you"
- **Jeho úkolem je převzít veškerou zodpovědnost za požadavky odeslané klienty a převést je na požadované služby tak, aby se aplikace už nemusela o nic starat**

Reactor - Motivační příklad

Distribuční logovací služba

- **Máme aplikaci, která potřebuje pravidelně ukládat svůj současný stav na server v distribuovaném systému.**
- **Logovací služba má za úkol tyto data uložit do databáze (nebo vytisknout)**
- **Klient může vyvolat pouze dvě události**
  - Connect – žádost o připojení k serveru
  - Read – žádost o přečtení logu

- **Hloupé řešení:**
  - Vytvořit pro každé připojení nové vlákno
- **Problémy:**
  - Neefektivní, neškálovatelné – nelze měnit kontext
  - Vyžaduje složitou správu vláken
  - Vlákna nejsou podporována na všech systémech

Reactor - Myšlenka

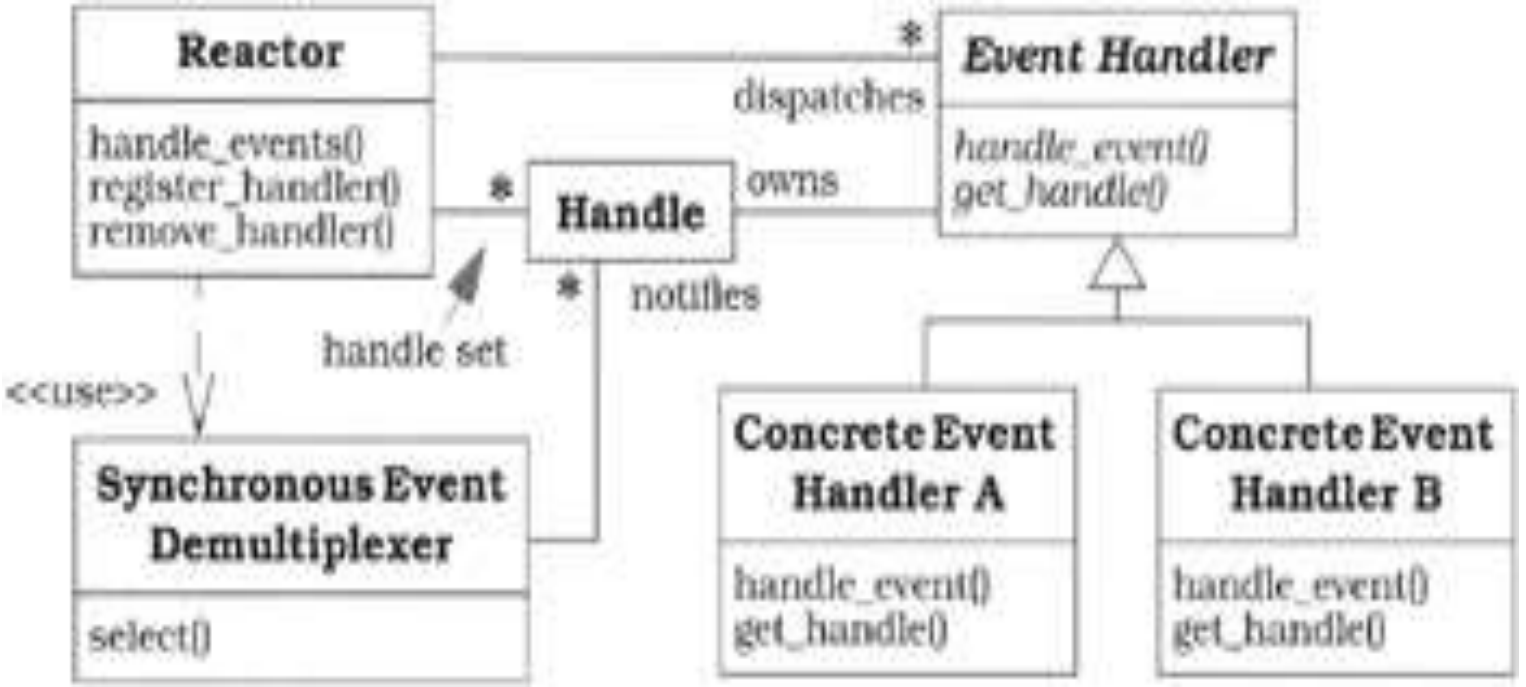
Vytvoření event-driven aplikace, která bude schopná přijímat více požadavků zároveň a bude je schopna je postupně synchronně spracovat.

Problémy

- Služba musí být schopna spracovat více požadavků najednou
- Každý z požadavků je označen identifikátorem události a služba musí být schopna tento požadavek demultiplexovat a vyvolat adekvátní událost

- **Řešení**
  - Synchronně čekat na příchozí požadavky
  - Integrovat demultiplexor a dispatcher jako služby, které se starají o požadavky
  - Oddělit dispatcher a demultiplexor od aplikační logiky

Reactor - Struktura



Reactor - Příklad ze života

Telefonní linka

- **Telefonní síť je Reactor.**
- **Vy jste event handler registrovaný telefonním číslem (handle).**
- **Pokud se vás někdo pokouší dovolat, telefonní síť vás na hovor upozorní zvoněním a vy tuto událost spracujete tím, že zvednete telefon.**

Proactor - úvodem

- **Architekturální návrhový vzor, který dokáže efektivně demultiplexovat a dispatchovat požadavky služby spouštěné dokončením asynchronních operací, aby tak dosáhl vyšších výkonů a souběžnosti**
- **Jeho aplikační komponenty (klient, completion handlers) jsou proaktivní**

Proactor - Motivační příklad

WebServer

pokud si uživatel chce zobrazit webovou stránku, musí dojít k následujícím událostem:

- **Prohlížeč naváže spojení se serverem a zašle požadavek GET**
- **Server obdrží událost s žádostí o připojení, přijme spojení a přečte si požadavek**
- **Server otevře a přečte požadovaný soubor**
- **Server odešle obsah souboru zpět prohlížeči a uzavře spojení**

- **Hloupé řešení**
  - Použít návrhový vzor Reactor
- **Problémy**
  - Nedostatečně škálovatelný pro velké množství současně připojených uživatelů
  - Uživatelé by museli čekat než by na ně došla řada

Proactor - Myšlenka

Event-driven aplikace schopná přijímat a spracovávat požadavky asynchronně

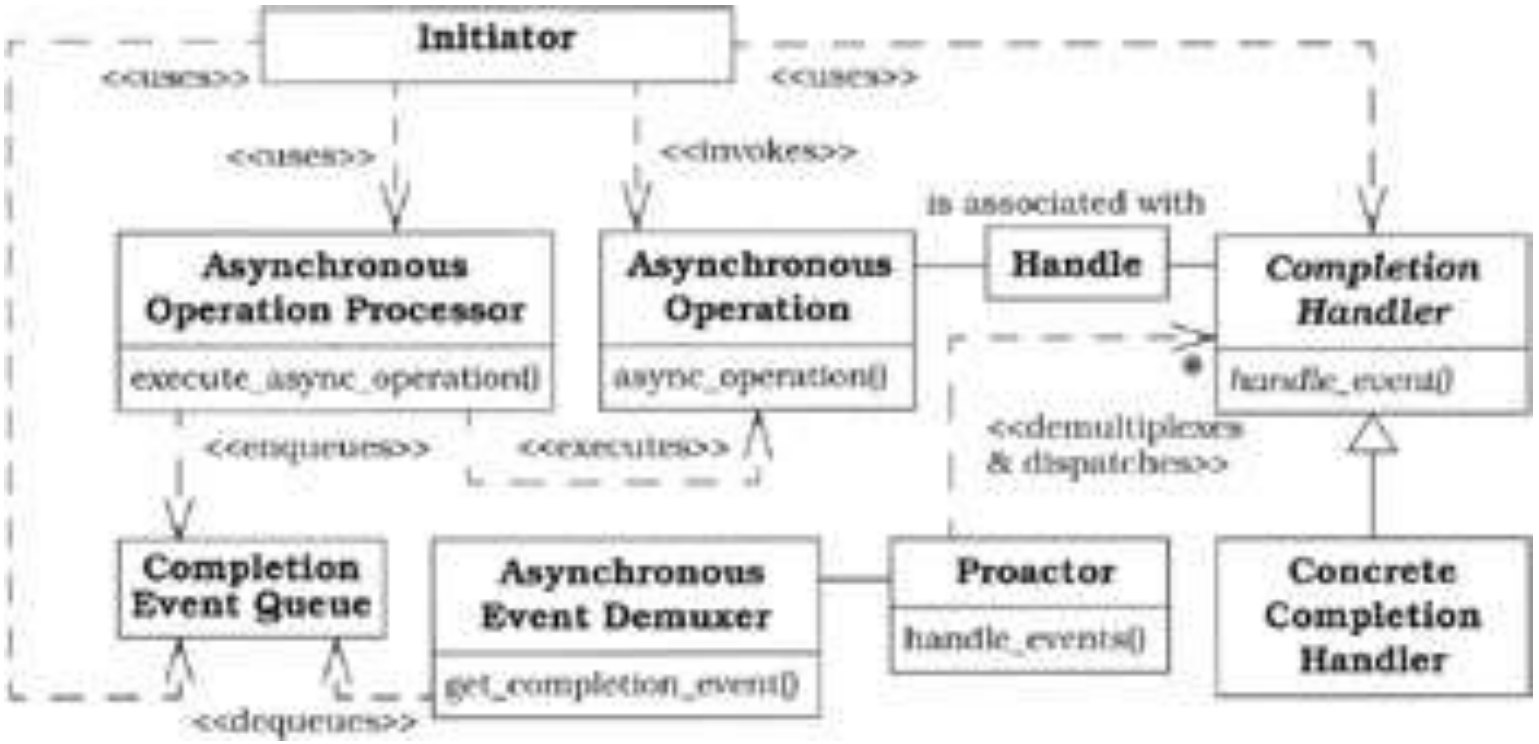
Problémy

- **Po dokončení asynchronního spracovávání požadavku musí být výsledek spracován pomocí dalších příslušných událostí**

Řešení:

- **Rozdělení aplikačních služeb na dvě části**
  - Dlouhotrvající operace
  - Completion handlery
- **Completion handler se stará o spracování vysledků dlouhotrvajících operací po jejich dokončení**

Proactor - Struktura



Proactor – Příklad ze života

Upozornění nepřijatých hovorů

- Pokud zavolám kamarádovi, který je momentálně nedostupný, ale vím, že pokud uvidí upozornění, tak zavolá zpět.
- V tomto případě jsem initiator, který požaduje asynchronní operaci na asynchronous operation processoru (telefon mého kamaráda).
- Mezitím než se kamarád ozve si může zatím procvičovat návrhové vzory.
- Tím že si kamarád přečte upozornění nepřijatých hovorů a zavolá mi zpět se chová jako proactor.
- Tím že ten telefon zvednu se chovám jako completion handler, který právě spracovává callback.

Asynchronous Completion Token (ACT)

- **Další názvy: Active Demultiplexing, Magic Cookie**
- **Umožňuje aplikaci efektivně demultiplexovat a spracovat asynchronní operace závislé na službách aplikace**

ACT – Motivační příklad

Rozsáhlý internetový burzovní systém

- **Nutnost kontroly, aby veškeré aktivity byly prováděny bezchybně – chyba může znamenat ušlý zisk a způsobit žaloby**
- **Vytváření management agentů, kteří dělají analýzy a snaží se detekovat možné chyby**
- **Těchto agentů můžou být stovky a na každý z nich může být zachyceno hned několik událostí, které mají informovat administrátory. Navíc každá z těchto událostí může být vyhodnocena jiným způsobem.**

ACT - Myšlenka

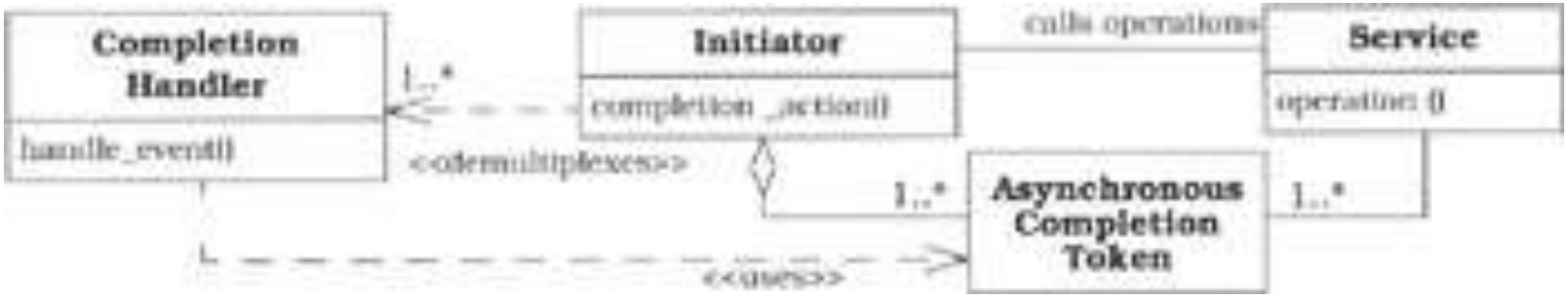
Evet-driven systém, ve kterém aplikace asynchronně vyvolají službu a následně se spracuje výsledek této služby přiřazenou akcí.

Problémy

- **Pokud se vyšle požadavek na více asynchronních služeb najednou, tak služby nemusí vědět který handler na který požadavek použít**
- **Po spracování služby by měla aplikace strávit co nejméně času zjišťováním co s výsledkem udělat**

- **Řešení:**
- **S každou asynchronní službou, kterou initiator vyvolá zároveň pošle informaci, která identifikuje jak by měl initiator spracovat výsledek použité služby.**
- **Po skončení operace se tato informaci vratí zpět initiatorovi , tak aby mohla být efektivne použita k demultiplexovaní odpovědi.**

ACT - Struktura



ACT – Příklad ze života

FeDex

- Poštovní služba má možnost odeslání účtu po uspěšném doručení balíčku, ve kterém je volné pole, do kterého si mohl odesílatel před odesláním balíku napsat libovolnou hodnotu např. vlastní identifikátor balíku, nebo odkaz na další operace, které je po doručení balíku nutno udělat.

Acceptor-Connector

Odděluje připojení a inicializaci spolupracujících peer služeb v síťově orientovaném systému od spracovávání prováděné těmito peer službami po jejich připojení a inicializaci

Acceptor-Connector – Motivační příklad

Rozsáhlá distribuční aplikace monitorující a kontrolující shlukování satelitů.

- **Takováto sít se typicky skládá z multi-service brány na aplikačním levelu, která přepojuje posílání dat mezi různými peery**
- **Aplikace používá TCP-IP protokol s tím, že jednotlivé porty poskytují různé služby**
- **Služby by měli mít přes bránu následující možnosti:**
  - Aktivně vytvořit spojení
  - Pasivně přijímat spojení od jiných peerů
  - Chovat se různě za daných situacích

Acceptor-Connector – Myšlenka

- **Síťově založený systém nebo aplikace, ve které jsou connection-oriented protokoly použity ke komunikaci mezi peery.**
- **Služby těchto peerů jsou propojeny transportními koncovými body.**

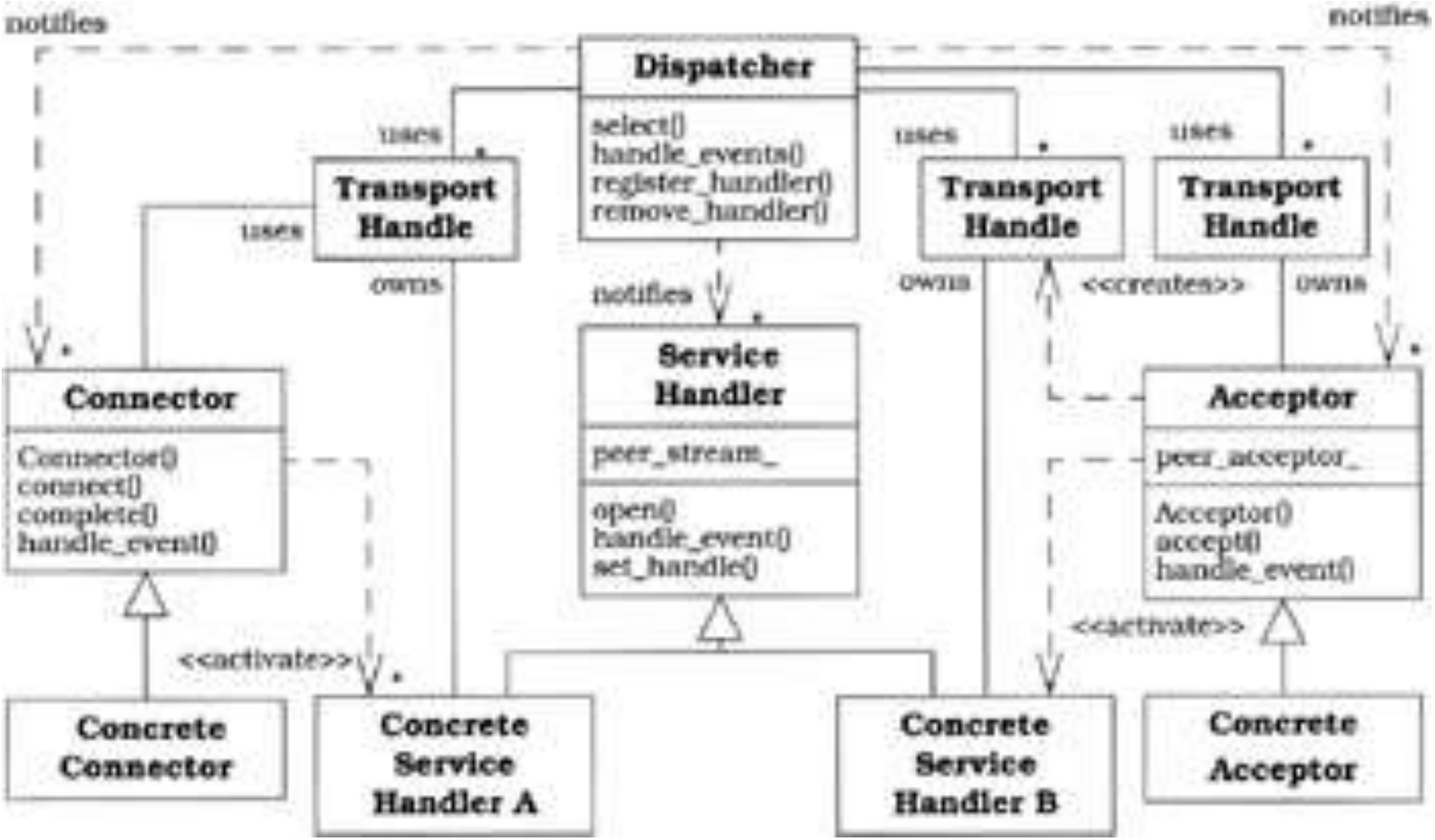
Problémy

- Aplikace v síťově orientovaných systémech typicky obsahují velké množství kódu na konfiguraci připojení a inicializaci služeb.
- Tento kód je často nezávislý spracovávání služeb pro přesun dat.
- Seskupování konfiguračního kódu s aplikačním kódem může vést k problémům.

Acceptor-Connector - Řešení

- **Rozdělení připojujících a inicializačních služeb od ostatních služeb peerů**
- **Zapouzdření aplikačních služeb pomocí peer service handlerů**
- **Vytvoření acceptor factory**
- **Vytvoření connector factory**

Acceptor-Connector - Struktura



Acceptor-Connector – Příklad ze života

Manažeři a sekretářky

- Manažer se chce spojit s jiným manažerem, tak požádá svou sekretářku, aby za něj uskutečnila hovor.
- Sekretářka zavolá druhému manažerovi, ale na telefon odpoví jiná sekretářka.
- Sekretářka, která hovor uskutečnila, je connector.
- Sekretářka, která hovor přijala, je acceptor.
- Manažeři jsou peer service handlers.

4. Synchronization Patterns

Scoped Locking

- **Jaký má účel?**
  - Zajišťuje zamknutí zámku před vstupem do kritické sekce a jeho uvolnění po opuštění této sekce
- **Moderní programovací jazyky už mají tento koncept přímo jako jazykový konstrukt**

```
// C#:
int foo()
{
    lock (lockObject)
    {
       // do critical
       // stuff
    }
    return 0;
}
```

```
// Java:
int foo() {
    synchronized
        (lockObject) {
       // do critical
       // stuff
    }
    return 0;
}
```

```
// C++
int foo()
{
    lock.acquire();
    // do critical
    // stuff
    lock.release();
    return 0;
}
```

Jaký je mezi těmito kódy rozdíl?

Scoped Locking

```
// C#:
int foo(int bar)
{
    lock (lockObject)
    {
       // do critical
       // stuff
    if (bar == 42)
    {
      return -1;
    }
    // do another
    // critical
    // stuff
    }
    return 0;
}
```

```
// Java:
int foo(int bar)
{
    synchronized
        (lockObject) {
    // do critical
    // stuff
    if (bar == 42) {
      return -1;
    }
    // do another
    // critical
    // stuff
    }
    return 0;
}
```

```
// C++
int foo(int bar)
{
    lock.acquire();
    // do critical
    // stuff
    if (bar == 42)
    {
    return -1;
    }
    // do another
    // critical
    // stuff
    lock.release();
    return 0;
}
```

- **Problémy s ručním zamykáním a odemykáním:**
  - Ne vždy programátor promyslí všechny možné toky řízení (control flow) programu
  - Duplikuje se kód
- **Řešení:**
  - Odemykat zámek automaticky, jakmile tok řízení opustí kritický blok

Programátor zapomněl, že se nachází v kritické sekci – zámek zůstal zamknut i po jejím opuštění

Scoped locking - implementace

- **Co se v C++ děje automaticky na konci bloku či při vyvolávání výjimky?**
  - Volají se destruktory lokálních proměnných – toho můžeme využít

```
class MutexGuard
{
public:
    MutexGuard(Lock &lock):
        _lock(&lock)

    {
        _lock->lock();
    }

    ~MutexGuard()
    {
        _lock->unlock();
    }
private:
    Lock *_lock;
    MutexGuard(const MutexGuard &);
    void operator=
        (const MutexGuard &);
};
```

Chceme zabránit kopírování a přiřazování MutexGuardu

MutexGuard je zkonstruován na záčátku synchronizovaného bloku (kritické sekce), uvolněn je pak už automaticky

```
int foo(int bar)
{
    // do non-critical stuff
    {
        MutexGuard guard(lock);
        // do critical stuff
        if (bar == 42)
        {
            return -1;
        }
        // another critical stuff
    }
    // another non-critical stuff
    return 0;
}
```

Scoped Locking - poznámky

- **Explicitní odemykání**
  - acquire() zamkne zámek, pokud je odemčený a zapíše si, že nyní byl zamknut
  - release() uvolňuje zámek jen v případě, že je zamčený, a zaznamená, že zámek byl odemčen
  - konstruktor volá acquire(), destruktor release()
  - příznak uzamknutí se vyplatí i v případě, že zamykání zámku může selhat
  - programátor proto nesmí volat metody zámku přímo!!!

- **Problémy**
  - deadlock při rekurzivním volání metody (bez rekurzivního mutexu)
    - řeší pattern Thread-safe Interface
  - nezvládne systémové věci (abort threadu uvnitř kritické sekce) ani Cčkové longjmp()
  - kompiler vypisuje varování ohledně nepoužité lokální proměnné
    - použijeme makro, které „něco udělá“
- **Použití**
  - všechny „větší“ ucelené knihovny (Boost, Threads.h++, ACE)

- **Související vzory**
  - Strategized Locking (modularita), Thread-safe Interface (rekurze)

Strategized Locking

- **Motivace**
  - chceme mít jednovláknovou verzi systému, která se nezpomaluje zamykáním, a vícevláknovou verzi, která zamyká kritické sekce
  - multiplatformní prostředí s různými synchronizačními primitivy
  - chceme se vyhnout duplikaci kódu

- **Způsoby parametrizace**
  - polymorfismus – konkrétní primitiva jsou známa až za běhu
  - templates – konkrétní primitiva známá už během kompilace

- **Realizace**
  - navrhneme abstraktní rozhraní, které bude systém používat
  - je vhodné použít Guard (Scoped Locking pattern)

Strategized locking - polymorfismus

```
class AbstractLock
{
public:
    void lock() = 0;
    void unlock() = 0;
};

class Guard
{
private:
    AbstractLock *_lock;
public:
    Guard(AbstractLock &lock):
        _lock(&lock)

    {
        _lock->lock();
    }

    ~Guard()
    {
        _lock->unlock();
    }
};
```

```
class MutexLock: public AbstractLock
{
private:
    Mutex *_mutex;
public:
    MutexLock(Mutex &mutex):
                    _mutex(&mutex) {}
    /* override */ void lock()
    {
        _mutex->acquire();
    }
    /* override */ void unlock()
    {
        _mutex->release();
    }
};

class NullLock: public AbstractLock
{
public:
    NullLock() {}
    /* override */ void lock() {}
    /* override */ void unlock() {}
};
```

Strategized locking - templates

Některé překladače umožňují defaultní template argumenty, v tom případě je vhodné nastavit je na nejpravděpodobnější případ

```
template class Guard<class LOCK>
{
private:
    LOCK *_lock;
public:
    Guard(LOCK &lock):
        _lock(&lock)

    {
        _lock->lock();
    }

    ~Guard()
    {
        _lock->unlock();
    }
};
```

```
class MutexLock
{
private:
    Mutex *_mutex;
public:
    MutexLock(Mutex &mutex):
                    _mutex(&mutex) {}
    void lock()
    {
        _mutex->acquire();
    }
    void unlock()
    {
        _mutex->release();
    }
};

class NullLock
{
public:
    NullLock() {}
    void lock() {}
    void unlock() {}
};
```

Strategized locking – hybridní varianta

- **Varianty**
  - pokud někdy víme typ už během kompilace a někdy ne, můžeme zvolit hybrid:

```
class AbstractPolymorficLock
{
public:
    void lock() = 0;
    void unlock() = 0;
};

class PolymorficMutexLock: public
                  AbstractPolymorficLock
{
private:
    Mutex *_mutex;
public:
    PolymorficMutexLock(Mutex &mutex):
                          _mutex(&mutex) {}
    /* override */ void lock()
    {
        _mutex->acquire();
    }
    /* override */ void unlock()
    {
        _mutex->release();
    }
};
```

```
template class Guard<class LOCK>
{
private:
    LOCK *_lock;
public:
    Guard(LOCK &lock):
        _lock(&lock)

    {
        _lock->lock();
    }

    ~Guard()
    {
        _lock->unlock();
    }
};
```

```
PolymorficMutexLock *lock = ...;
Guard<AbstractPolymorficLock>
    guard(*lock);
```

Strategized locking - shrnutí

- **Výhody**
  - flexibilita a snadná rozšiřitelnost
  - jednodušší údržba, není duplicitní kód
  - nezávislost a opětovná použitelnost (reusability)
- **Problémy**
  - při použití templates příliš vyniká strategie zámků
  - někdy až příliš flexibilní – nezkušený programátor může omylem zvolit nevhodné synchronizační primitivum
- **Použití**
  - v jazycích jako C# či Java jsme omezeni na polymorfismus
  - Adaptive Communication Environment (ACE) – opensource framework pro síťové a distribuované aplikace
  - ATL (COM objekty)
  - kernel operačního systému Dynix/PTX
  - Windows HAL.dll – různé spinlocky podle počtu procesorů

Thread-safe Interface

- **Motivace**
  - nemáme reentrantní zámky a synchronizované metody volají jiné (synchronizované) metody téhož objektu
  - s reentrantními zámky způsobuje zamykání příliš velký overhead

```
class HashMap
{
private:
    Lock *_lock;
public:
    void insert(int key, int value) {
        Guard guard(_lock);
        if (value < 0) {
            return;
        }
        if (this->get(key) == -1) {
            // do insert
        }
    }

    int get(int key) {
        Guard guard(_lock);
        // pick up
        return value;
    }
};
```

Při volání synchronizované metody se zamčeným zámkem nastane deadlock

- **Řešení**
  - veřejné metody POUZE zamykají a volají vnitřní metody
  - vnitřní metody NIKDY nezamykají

Thread-safe Interface - implementace

```
class HashMap
{
private:
    Lock *_lock;
public:
    void insert(int key, int value) {
        Guard guard(_lock);
        if (value < 0) {
            return;
        }
        this->_insert(key, value);
    }
    int get(int key) {
        Guard guard(_lock);
        return this->_get(key);
    }

private:
    void _insert(int key, int value) {
        if (this->_get(key) == -1) {
            // do insert
        }
    }
    int _get(int key) {
        // pick up value
        return value;
    }
};
```

Vnější metody jsou synchronizované

Vnější metoda volá vnitřní metodu - OK

Vnitřní metoda volá vnitřní metodu – OK, deadlock nemůže nastat

Thread-safe Interface - varianty

- **Thread-safe Facade**
  - Thread-safe Interface pro celý systém komponent
  - je třeba refaktorovat systém, jinak nested monitor lockout

- **Thread-safe Wrapper**
  - pro třídy, které nepočítají s multithreadingem
  - veřejné metody zamknou zámek, zavolají implementaci a opět uvolní
  - příklad java.util.Collections.getSynchronizedMap()

Thread-safe Interface - shrnutí

- **Výhody**
  - robustnost – snížené nebezpečí self-deadlocku
  - snížení overheadu
  - zjednodušení – oddělení logiky od potřeby synchronizace
- **Problémy**
  - zvýšení počtu metod
  - deadlocku se úplně nevyhneme při volání dalšího objektu (může volat zpět)
  - volání privátní metody na jiném objektu téže třídy
  - pevná granularita zámků (per objekt)
- **Související vzory**
  - Decorator – transparentně přidává funkcionalitu
  - Scoped Locking, Strategized Locking

Double-checked locking

- **Motivace**
  - nějaká část kódu musí být provedena nanejvýš jednou během běhu programu
  - už jsme viděli u Singletonu

Není thread-safe

```
class Singleton
{
private:
    static Singleton
            *_instance = NULL;
public:
    static getInstance()
    {
        if (instance == NULL)
        {
            _instance
               = new Singleton();
        }
        return _instance;
    }
}
```

Zbytečný overhead kvůli zamykání

```
class Singleton
{
private:
    static Singleton *_instance = NULL;
    static Lock _lock;
public:
    static getInstance()
    {
        Guard guard(_lock);
        if (_instance == NULL)
        {
            _instance = new Singleton();
        }
        return instance;
    }
}
```

Double-checked locking

- **Problémy**
  - kompiler může prohodit pořadí instrukcí
  - cache procesorů na některých platformách nejsou transparentní (Intel Itaniu, COMPAQ Alpha)
  - přiřazení pointeru není atomické
- **Řešení**
  - MSVC++: volatile keyword nebo _ReadWriteBarrier()
  - GCC: asm volatile ("":::"memory"); nebo __sync_synchronize()
  - ICC: __memory_barrier() nebo __sync_synchronize()

```
class Singleton
{
private:
    static Singleton *_instance = NULL;
    static Lock _lock;
public:
    static getInstance()
    {
        if (_instance == NULL)
        {
            Guard guard(_lock);
            if (_instance == NULL)
            {
                _instance = new Singleton();
            }
        }
        return instance;
    }
}
```

Double-checked locking – C#, Java

```
// C#
public class Singleton
{
   private static volatile
               Singleton instance;
   private static object syncRoot =
               new Object();

   private Singleton() {}

   public static Singleton Instance
   {
      get
      {
         if (instance == null)
         {
            lock (syncRoot)
            {
               if (instance == null)
                  instance =
                     new Singleton();
            }
         }
         return instance;
      }
   }
}
```

```
// Java version 5.0 and higher

class Singleton {
    private static volatile Singleton
                  instance = null;

    private Singleton() {}

    public static Singleton
                   getInstance() {
        Singleton result = instance;
        if (result == null) {

            synchronized(this) {
                result = instance;
                if (result == null) {
                    instance =
            result = new Singleton();
                }

            }
        }
        return result;
    }
}
```

Řešení pomocí thread-local proměnné

```
public class Singleton
{
   private static object syncRoot = new Object();
   private static Singleton globalInstance = null;
   [ThreadStaticAttribute]
   private static Singleton threadLocalInstance = null;

   private Singleton() {}

   public static Singleton Instance
   {
      get
      {
         if (threadLocalInstance == null)
         {
            lock (syncRoot)
            {
               if (globalInstance == null)
               {
                  globalInstance = new Singleton();
               }
               threadLocalInstance = globalInstance;
            }
         }
         return threadLocalInstance;
      }
   }
}
```

Proměnná soukromá pro každé vlákno

Přístup k datům společným pro všechna vlákna je synchronizován

5. Concurrency Patterns

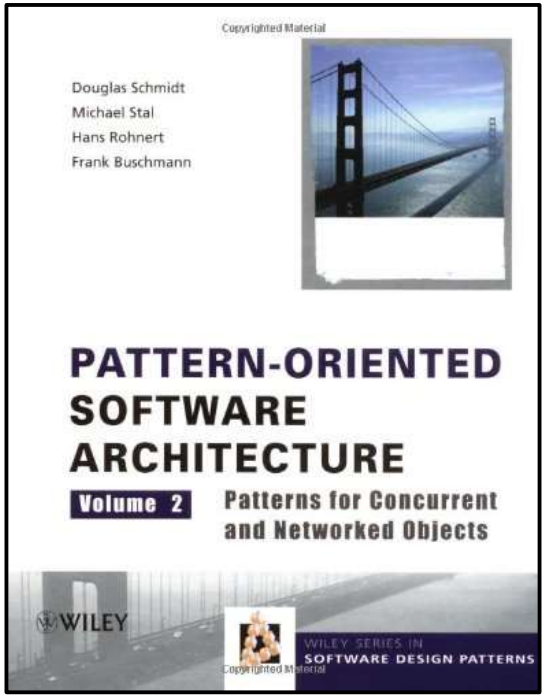
Concurrency

- **Concurrency**
  - více procesů běží současně
  - důvody
    - vyšší výkon
    - méně čekání
    - využití paralelního hardwaru – víc procesorů či jader
  - zdroje problémů
    - sdílení zdrojů mezi procesy
    - nedeterminismus

Když si nedáme pozor, může dojít k poškození dat či deadlocku!

Možné řešení – Concurrency Patterns

- **Active Object**
- **Monitor Object**
- **Half-Sync/Half-Async**
- **Leader/Followers**
- **Thread-Specific Storage**



- **Pattern-Oriented Software Architecture**
  - Patterns for Concurrent and Networked Objects, Volume 2
  - Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann
  - John Wiley & Sons, 2000
  - kapitola 5: Concurrency Patterns

Klasifikace Concurrency Patterns

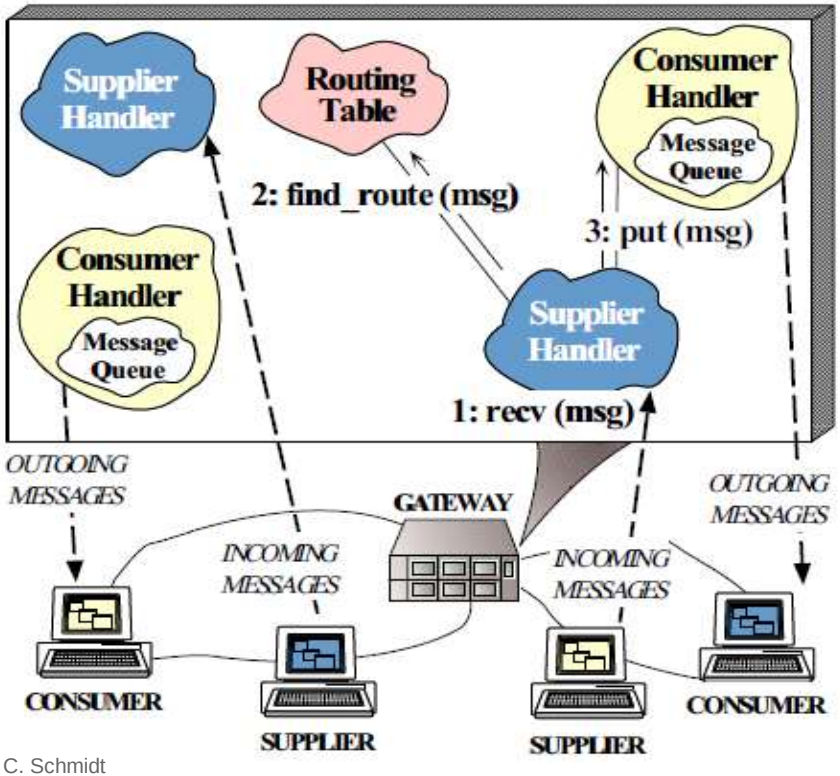
- **Design Patterns**
  - Active Object
  - Monitor Object
  - Thread-Specific Storage
- **Architectural Patterns**
  - Half-Sync/Half-Async
  - Leader/Followers

Active Object

Active Object – návrhový vzor

- **Kontext**
  - více klientů beží v samostatných vláknech a přistupuje ke sdílenému objektu
- **Účel**
  - zjednodušit soubežné přístupy k objektu, který žije ve vlastním vlákně
  - oddělit volání metody na tomto objektu od jejího vykonání

- **Příklad – komunikační brána**
  - procesy ze dvou komponent chtějí mezi sebou komunikovat, ale nechtějí být na sobě přímo závislé



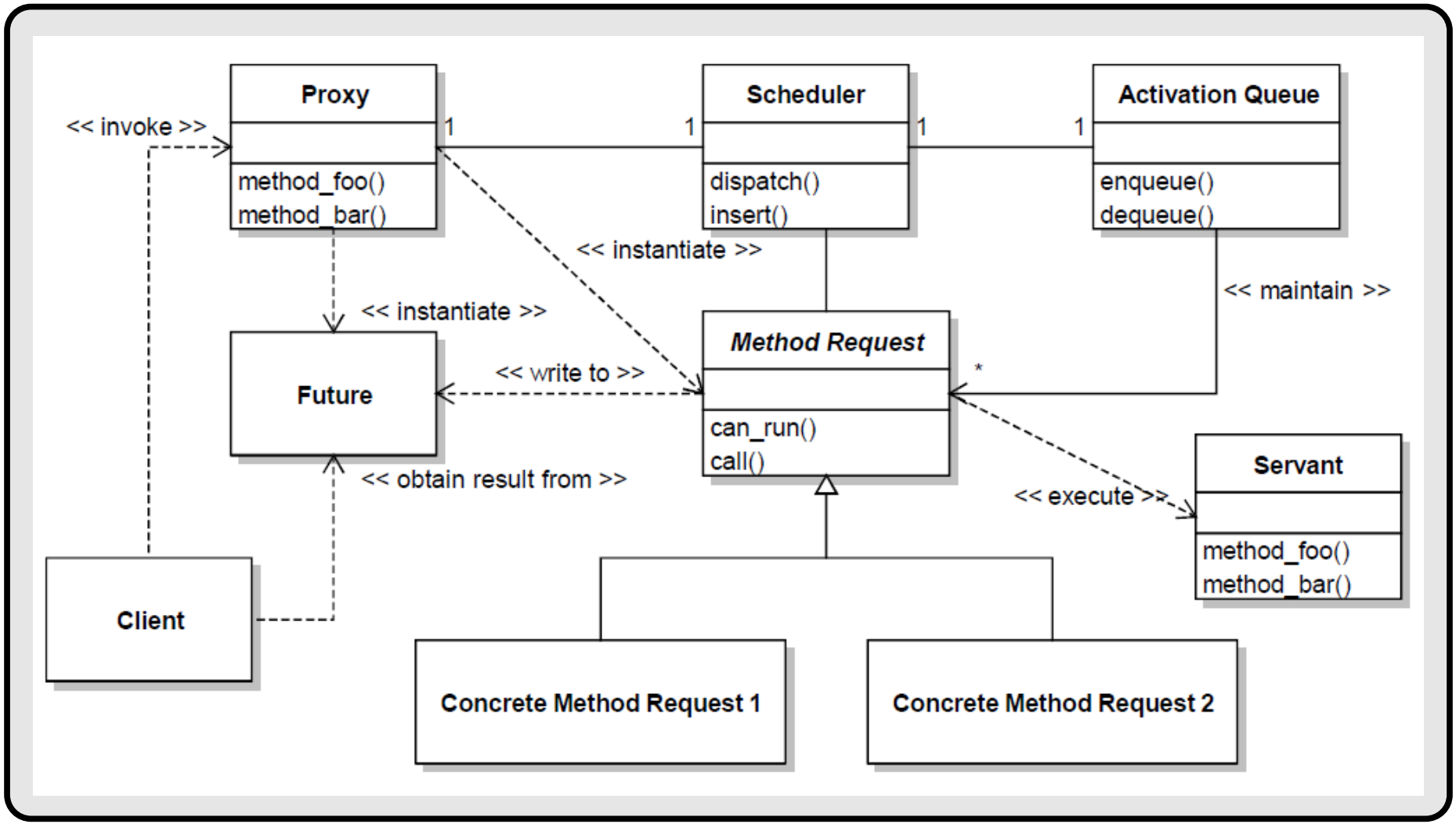
Active Object: An Object Behavioral Pattern for Concurrent Programming, R. Greg Lavender, Douglas C. Schmidt

Active Object – problémy

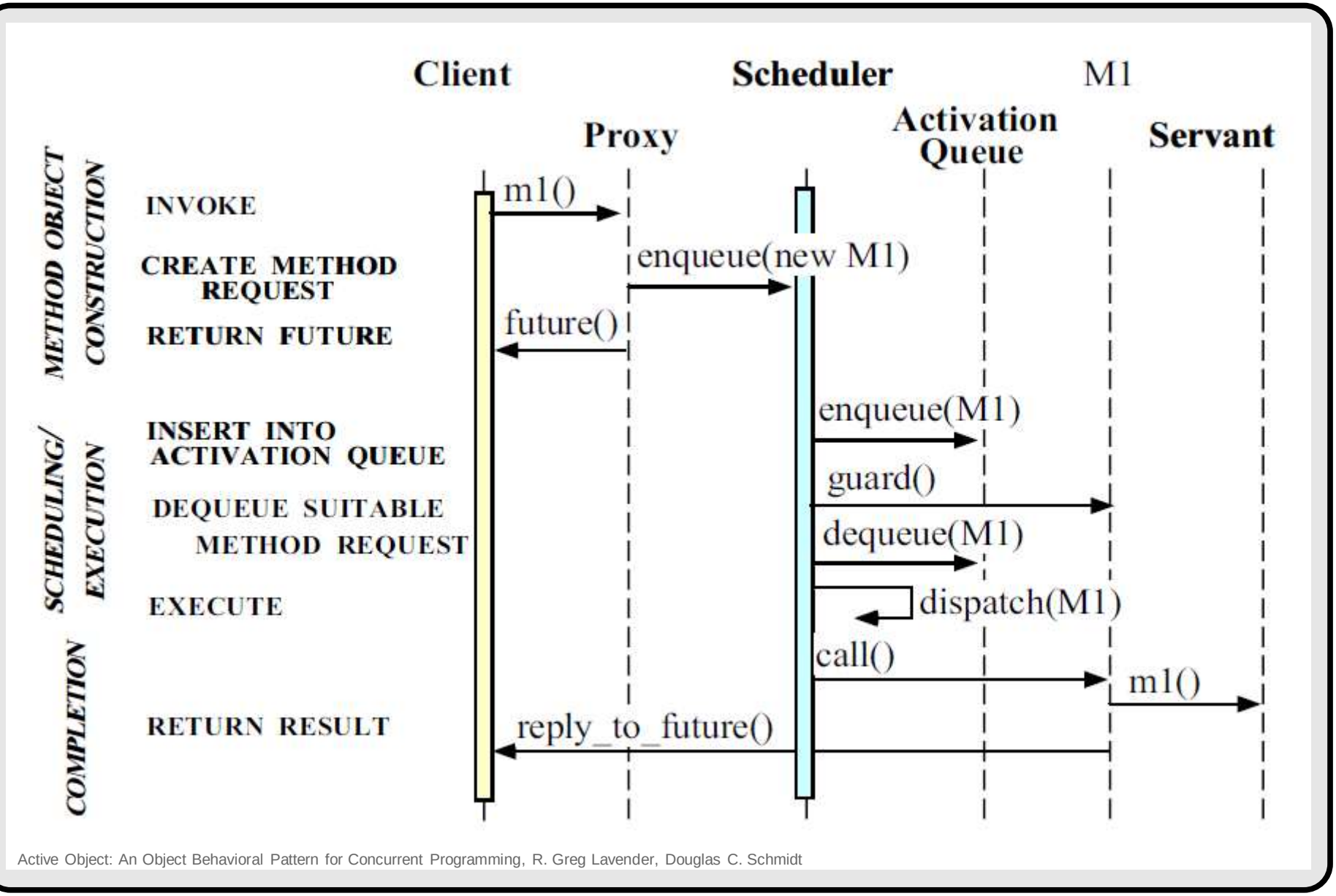
- **Problémy**
  - nechceme, aby náročnější metody blokovaly celý systém
  - synchronizovaný přístup ke sdíleným objektům musí být transparentní
  - chceme využít paralelní hardware – více jader a procesorů
  - chceme, aby celý systém byl škálovatelný
- **Řešení**

Oddělíme volání metody od jejího vykonání!

Active Object – struktura



Active Object – dynamické chování



Active Object: An Object Behavioral Pattern for Concurrent Programming, R. Greg Lavender, Douglas C. Schmidt

Active Object – varianty

- **Více rolí**
  - různá rozhraní pro více druhů klientů, víc rozšiřitelné
- **Integrovaný Scheduler**
  - práci Proxy a Servanta dělá Scheduler
  - jednoduší implementace, hůř znovupoužitelné
- **Předávání zpráv**
  - logika Proxy a Servanta mimo Active Object
  - víc práce pro programátory aplikace, víc chyb
- **Volání metod s časovým limitem**
- **Polymorfní návratová hodnota (Future)**
- **Distribuovaný Active Object**
  - rozdělení Proxy na Stub a Skeleton
  - podobný je vzor Broker, ale ten pracuje s mnoha Servanty
- **Active Object s Thread Poolem Servantů**
  - lepší paralelismus, muže být nutné synchronizovat

Active Object – příklady použití

- **Komunikační brána**
- **Časovač v Javě**
  - java.util.Timer a java.util.TimerTask
  - zjednodušený Active Object
- **Příklad ze života – restaurace**
  - Client … zákazník
  - Proxy … číšníci a servírky
  - Scheduler … šéfkuchař
  - Servant … kuchař
  - Activation Queue … seznam jídel k přípravě

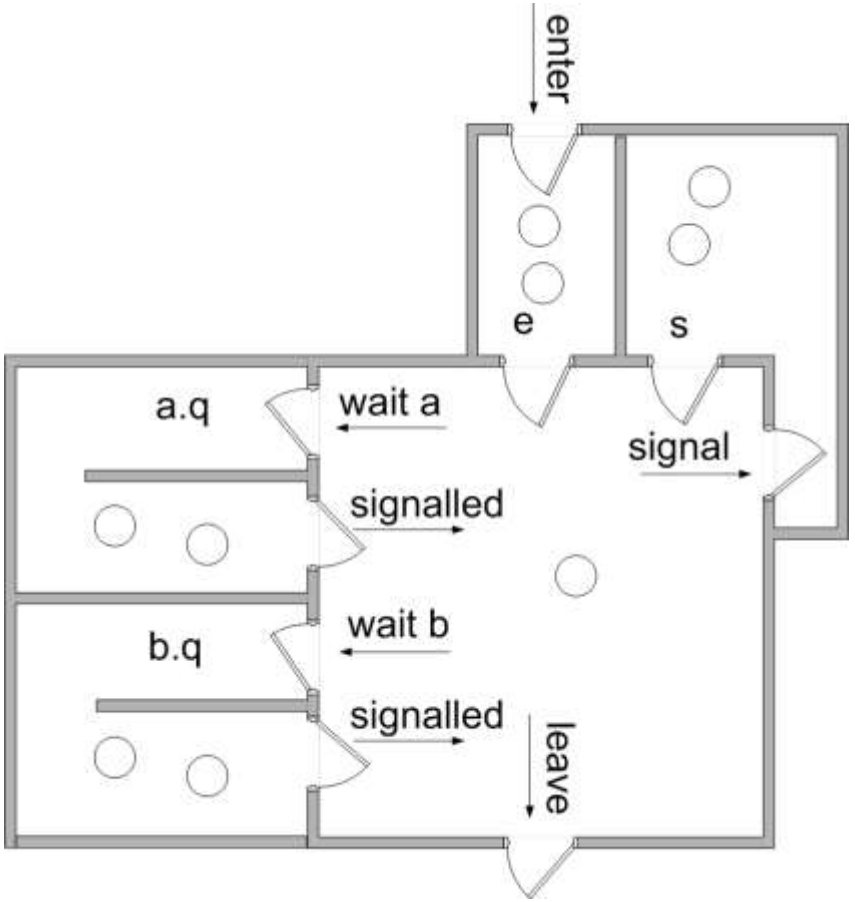
Active Object – souvislosti

- **Method Request**
  - je možno považovat za instaci vzoru Command
- **Activation Queue**
  - může být implementována s pomocí vzoru Robust Iterator
- **Scheduler**
  - je instancí vzoru Command Processor
  - pro více plánovacích politik je možno použít Strategy
- **Future**
  - může být implementována pomocí vzoru Counted Pointer

Active Object – shrnutí

- **Výhody**
  - volání a vykonávání metod probíhá v různých vláknech
  - zjednodušení složité synchronizace
  - metody se vykonávají v jiném pořadí, než byly volány
  - různé strategie pro plánování pořadí
  - je možné rozšířit pro distribuované použití
- **Nevýhody**
  - režie
    - přepínání kontextů
    - synchronizace
    - kopírování dat
    - složitější plánování v Scheduleru
  - složité debugování
- **Kdy je Active Object vhodný?**
  - při práci s relativně velkými objekty
    - jinak použít spíš Monitor Object

Monitor Object



Theodore Norvell, http://en.wikipedia.org/wiki/File:Monitor_%28synchronization%29-SU.png

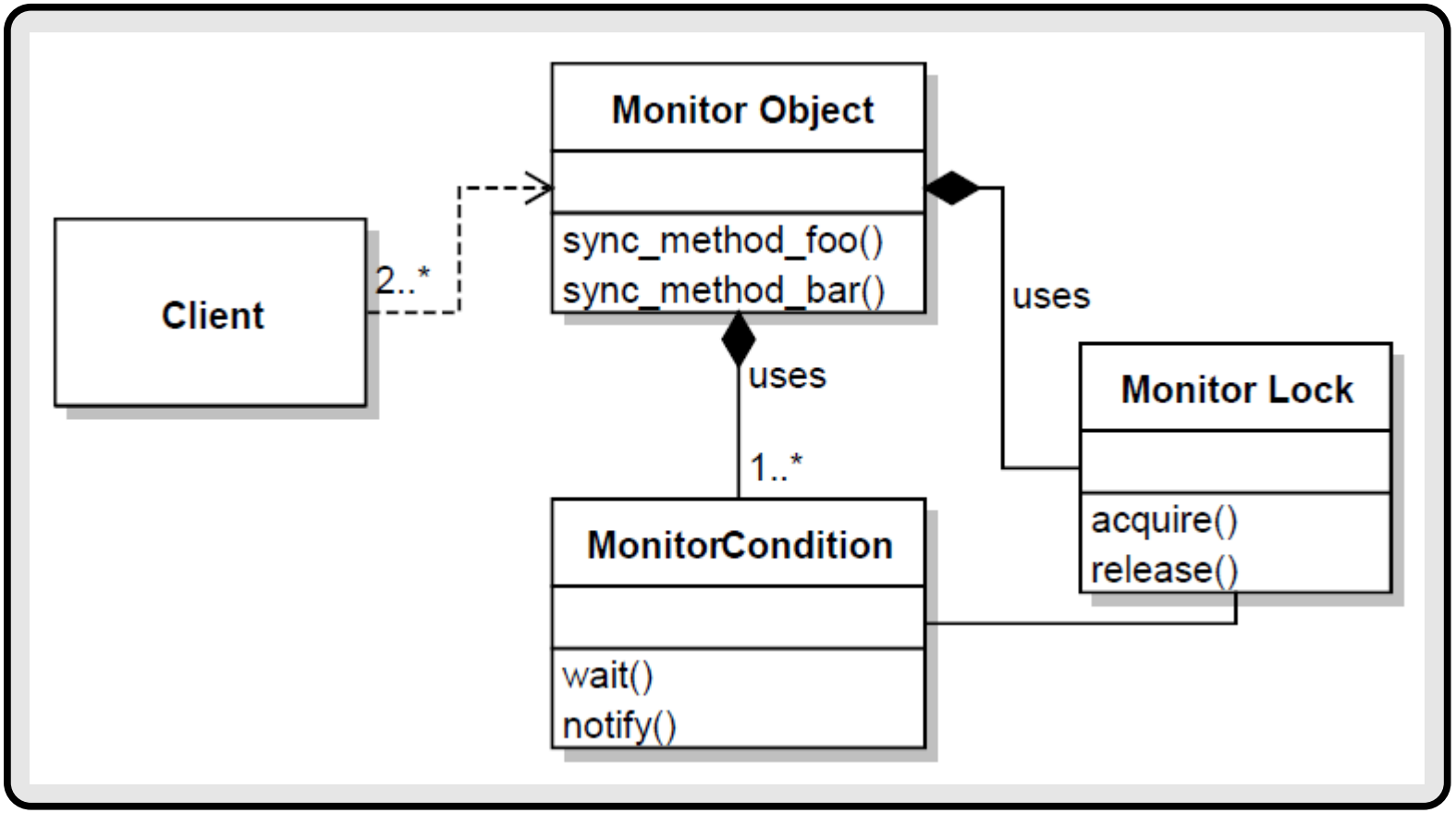
Monitor Object – návrhový vzor

- **Kontext**
  - více vláken volá současně metody na stejném objektu
  - objekt sám žádné vlákno nemá (je pasivní)
  - volání metody probíhá ve klientově vlákně
- **Účel**
  - serializovat souběžné volání metod na objektu
  - tím vynutit, aby s objektem pracovala vždy nejvýš jedna metoda v jediném vlákně
- **Příklad**
  - komunikační brána ze vzoru Active Object
    - pro malé objekty může být overhead Active Objectu příliš velký
    - složitá plánovací strategie nemusí být potřeba

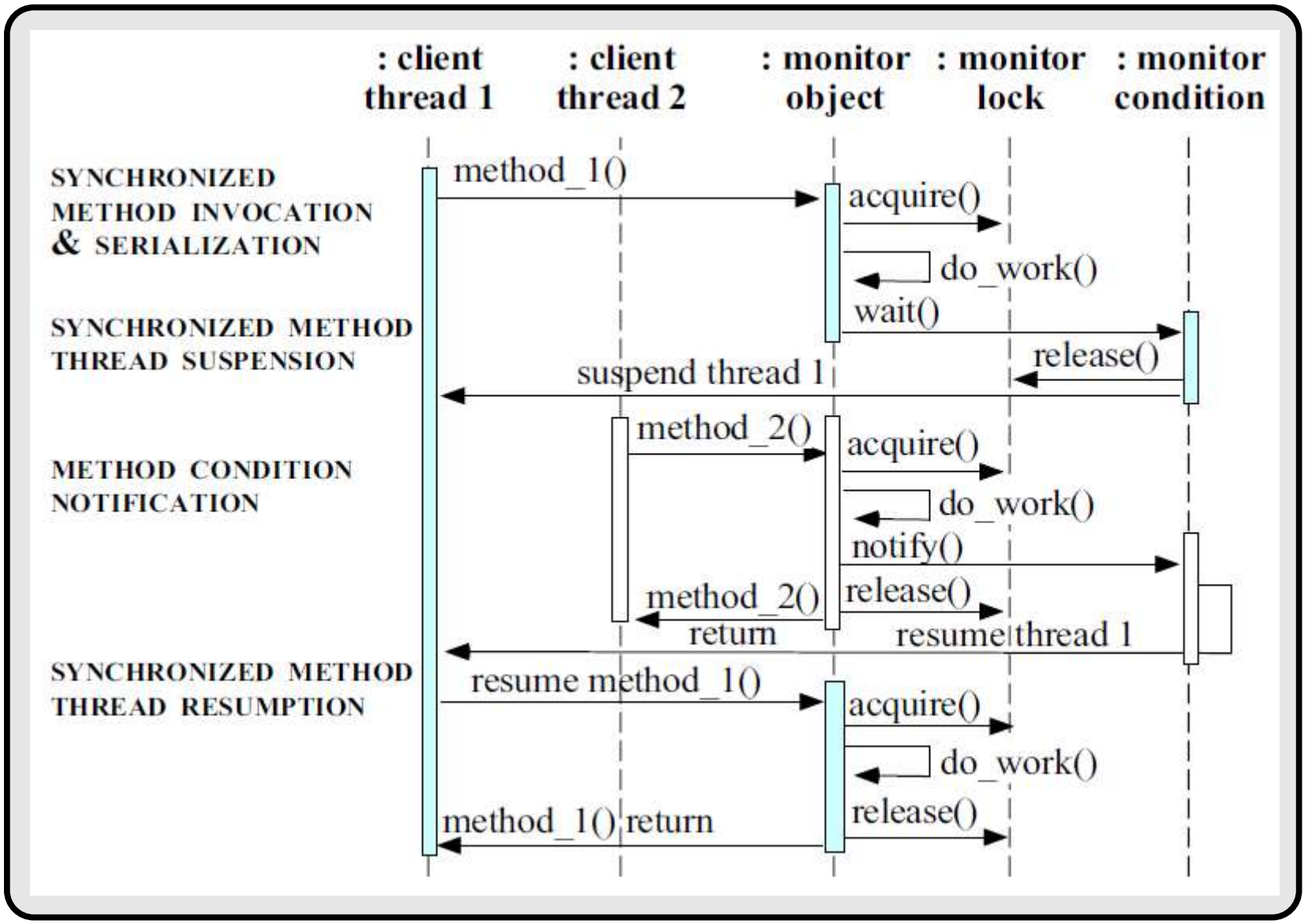
Monitor Object – problémy

- **Problémy**
  - současné volání metod objektu může poškodit jeho vnitřní stav
    - race conditions
  - podobně jako interface je nutné definovat synchronizační hranice
  - chceme transparentní synchronizaci
    - aby klient nemusel používat low-level primitiva
  - má-li metoda blokovat, musí se dobrovolně vzdát řízení
    - ochrana proti deadlocku a zbytečnému čekání
  - před uspáním a probuzením musí být objekt v korektním stavu

Monitor Object – struktura



Monitor Object – dynamické chování

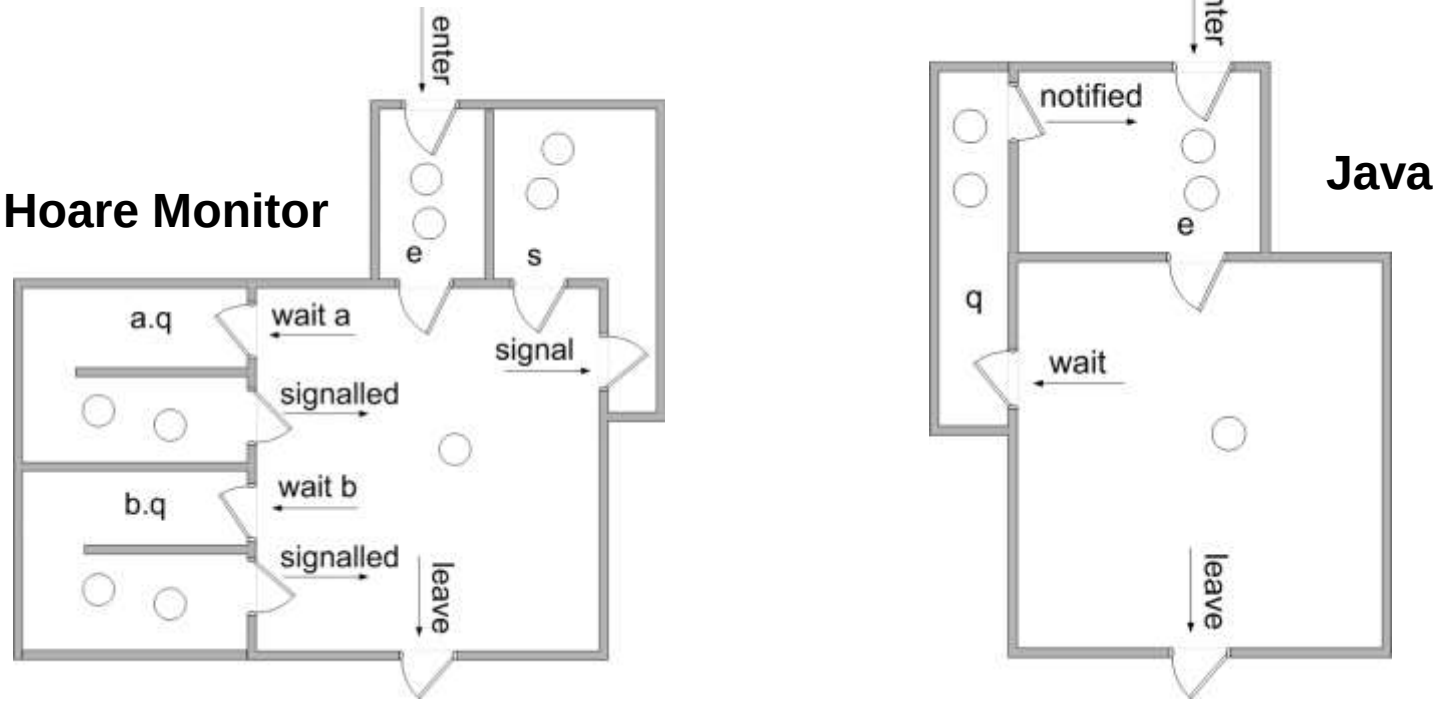


Monitor Object – varianty

- **Timed Synchronized Method Invocations**
  - časový limit na čekání
- **Strategized Locking**
  - flexibilní konfigurace zámku a podmínek
- **Multiple Roles**
  - objekt implementuje více rolí pro různé skupiny klientů
  - klient vidí z objektu jen specifické rozhraní
  - lepší rozšiřitelnost

Monitor Object – příklady

- **Dijkstra Monitor, Hoare Monitor**
- **Monitory na objektech v Javě**
- **Příklad ze života – fast food restaurace**



Monitor Object – shrnutí

- **Výhody**
  - jednoduché řízení konkurence
  - jednodušší plánování, kde se mají metody vykonávat
  - kooperativní plánování
  - použití podmínek

- **Nevýhody**
  - omezená škálovatelnost
  - složitá změna synchronizačních mechanizmů a politik
  - těsná vazba mezi funkčností a logikou synchronizace a plánování
  - nelze znovu použít implementaci s jinými synchronizačními mechanizmy
  - problémy s vnořováním – Nested Monitor Lockout

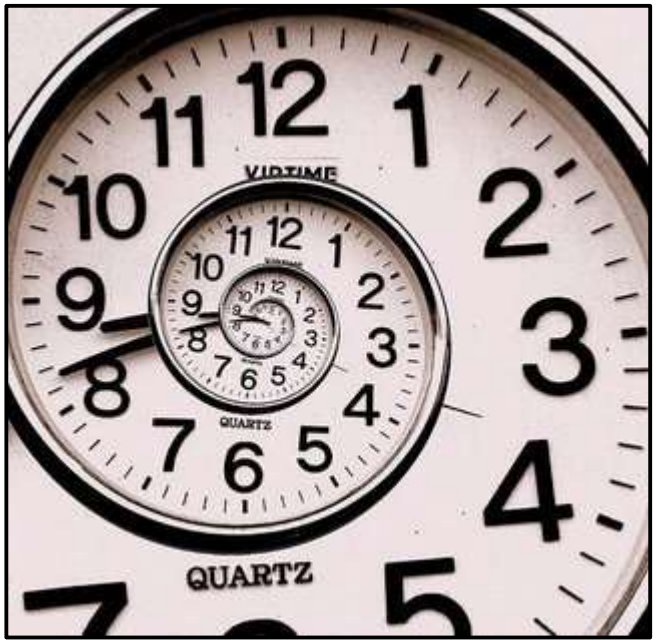
Active Object vs. Monitor Object

- Monitor Object a Active Object dělají podobně věci, ale trochu se liší

- **Active Object**
  - složitější
  - metody běží v jiném vlákně než klient
  - sofistikovanější, ale dražší vykonávání a příjem nových požadavků
  - kvůli větší režii se hodí spíš na větší objekty
  - asynchronní získání výsledků
  - lépe rozšiřitelný

- **Monitor Object**
  - jednodušší
  - metody běží ve vlákně klienta
  - menší režie, hodí se i na menší objekty
  - těsnější vazba mezi funkcionalitou a synchronizační logikou

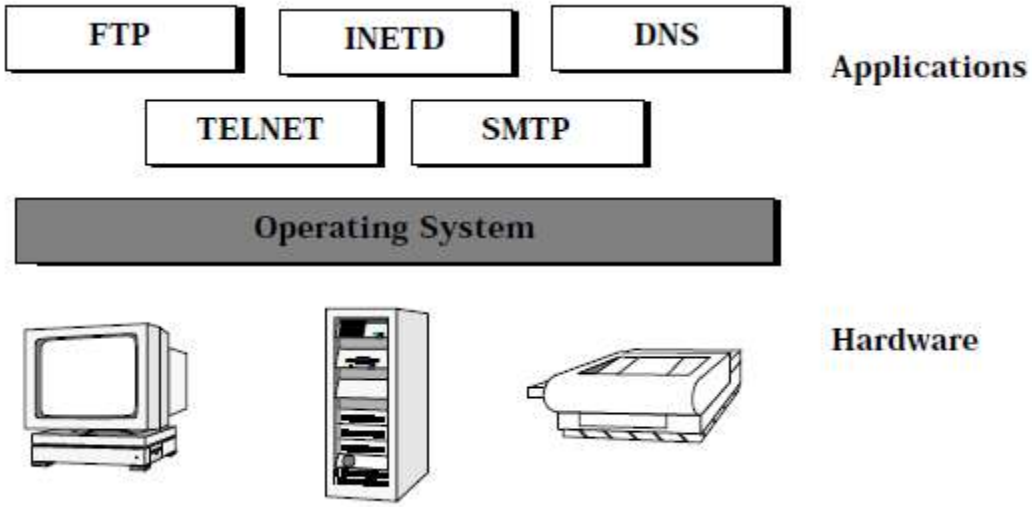
Half-Sync/Half-Async



http://homes.bio.psu.edu/people/faculty/bshapiro/spiral-clock.jpg

Half-Sync/Half-Async – architektonický vzor

- **Kontext**
  - vícevláknový systém s komunikací synchronních a asynchronních služeb
- **Účel**
  - zjednodušit použití takových služeb bez ztráty výkonnosti
- **Příklad**
  - síťování v BSD UNIXu

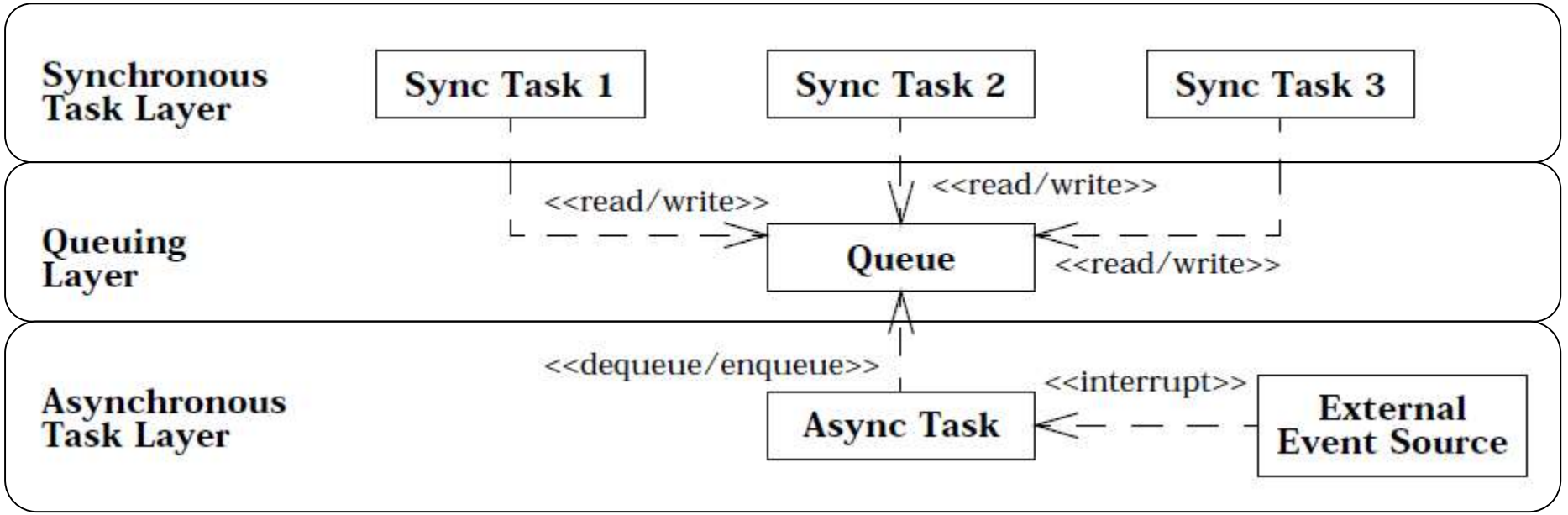


Half-Sync/Half-Async – problémy

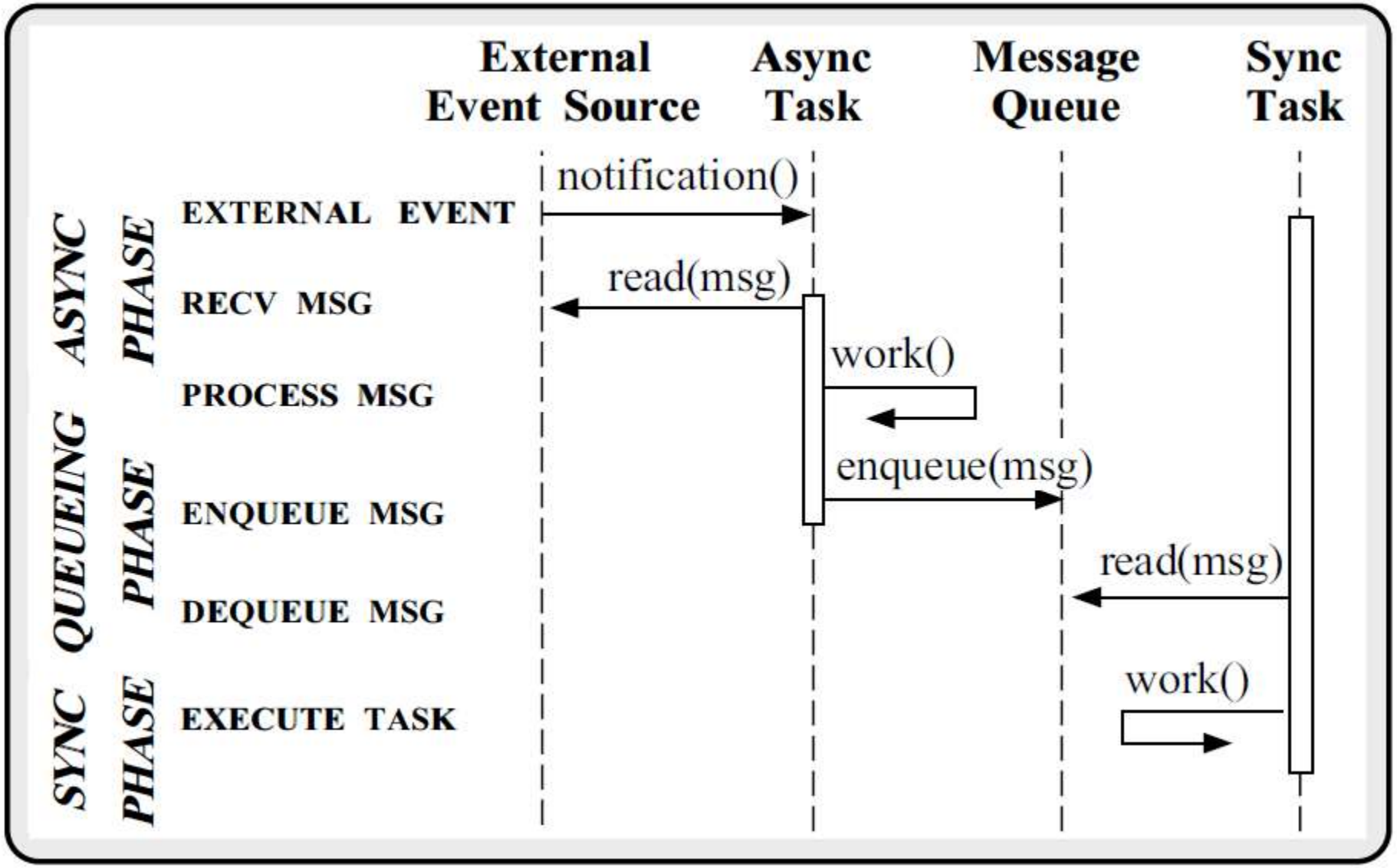
- **Problémy**
  - synchronní zpracování služeb
    - jednodušší programování, ale může dlouho blokovat
    - typicky high-level služby
  - asynchronní zpracování služeb
    - složitější programování, možnost vyššího výkonu
    - někdy je vynuceno přímo hardwarem
    - typicky low-level služby
  - tyto služby spolu potřebují komunikovat
  - jak to vše skloubit?

Half-Sync/Half-Async – struktura

- **Řešení**
  - rozdělit systém na synchronní a asynchronní vrstvu
  - mezi ně vložit komunikační mezivrstvu s frontou



Half-Sync/Half-Async – dynamické chování



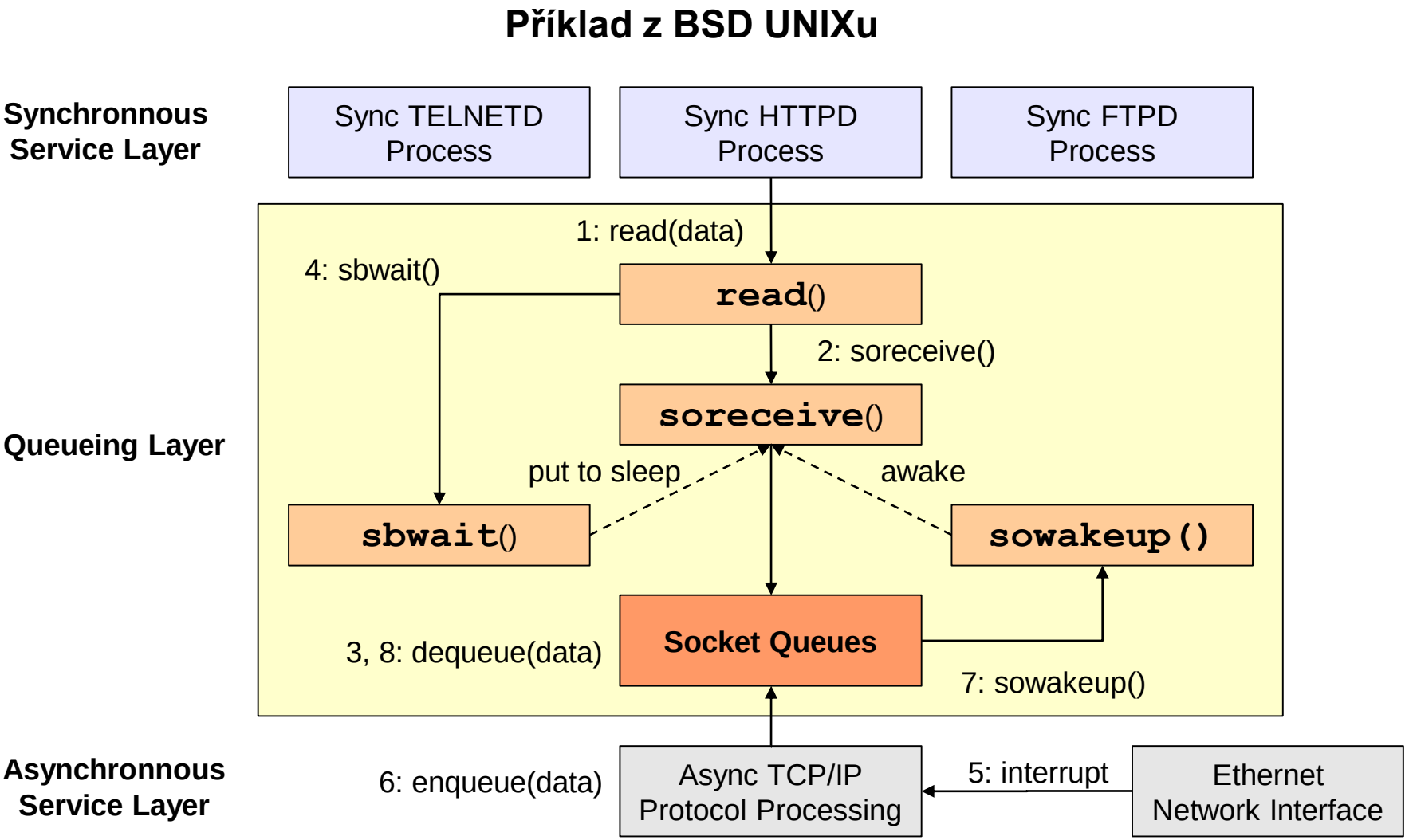
Half Sync/Half Async, Douglas C. Schmidt, 1998

Half-Sync/Half-Async – varianty

- **Varianty**
  - **Asynchronní řízení, synchronní data**
  - **Half-Async/Half-Async**
    - asynchronní zpracování je přístupné i pro high-level služby
  - **Half-Sync/Half-Sync**
    - synchronní zpracování i pro low-level služby
    - více vláken v jádře OS
    - příklady: Mach, Solaris
  - **Half-Sync/Half-Reactive**
    - v objektově orientovaných systémech
    - složení vzorů Reactor a Active Object s thread poolem
    - asynchronní vrstva – Reactor
    - mezivrstva – Activation Queue
    - synchronní vrstva – Servant

- **Souvislosti**
  - vrstvení v Half-Sync/Half-Async je příkladem vzoru Layers

Half-Sync/Half-Async – příklad



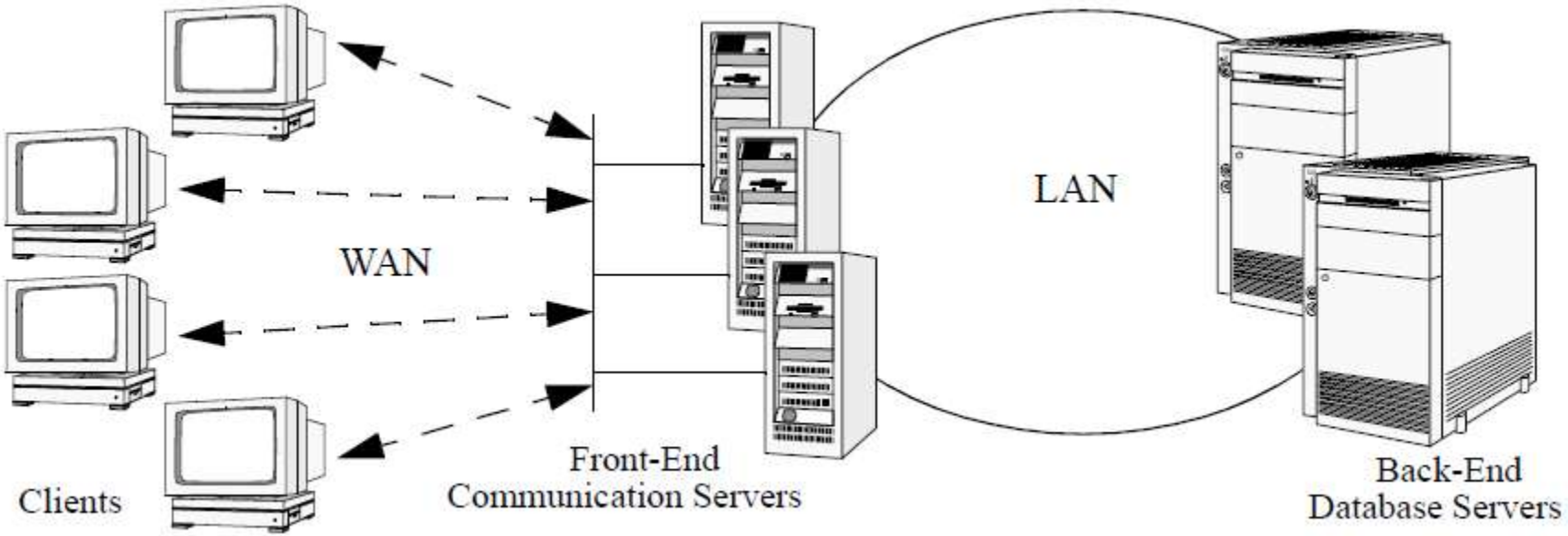
Half-Sync/Half-Async – shrnutí

- **Princip**
  - oddělení synchronních a asynchronních služeb do dvou vrstev
- **Výhody**
  - jednodušší programování synchronních služeb při zachování výkonnosti
  - uzavření složitosti asynchronních služeb do malé části systému
  - oddělení synchronizačních politik
  - centralizovaná komunikace mezi vrstvami
- **Nevýhody**
  - režie za komunikaci mezi vrstvami
  - složitější debugování a testování


Leader/Followers

Leader/Followers – architektonický vzor

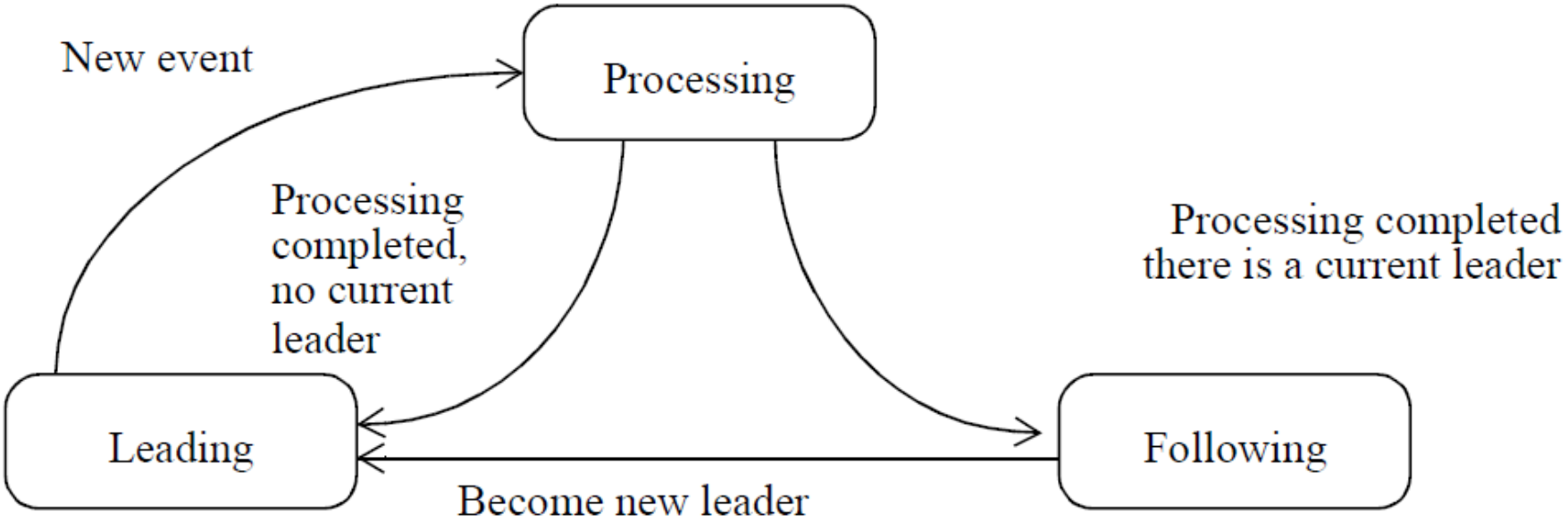
- **Kontext**
  - vlákna musí efektivně zpracovávat události ze sdíleného zdroje
- **Účel**
  - více vláken se střídá v přijímání, demultiplexování a zpracování požadavků, které přicházejí z více zdrojů
- **Příklad**
  - On-line Transaction Processing (OLTP)



Leader/Followers, Douglas C. Schmidt, Carlos O'Ryan, Michael Kircher, Irfan Pyarali, and Frank Buschmann, 2000

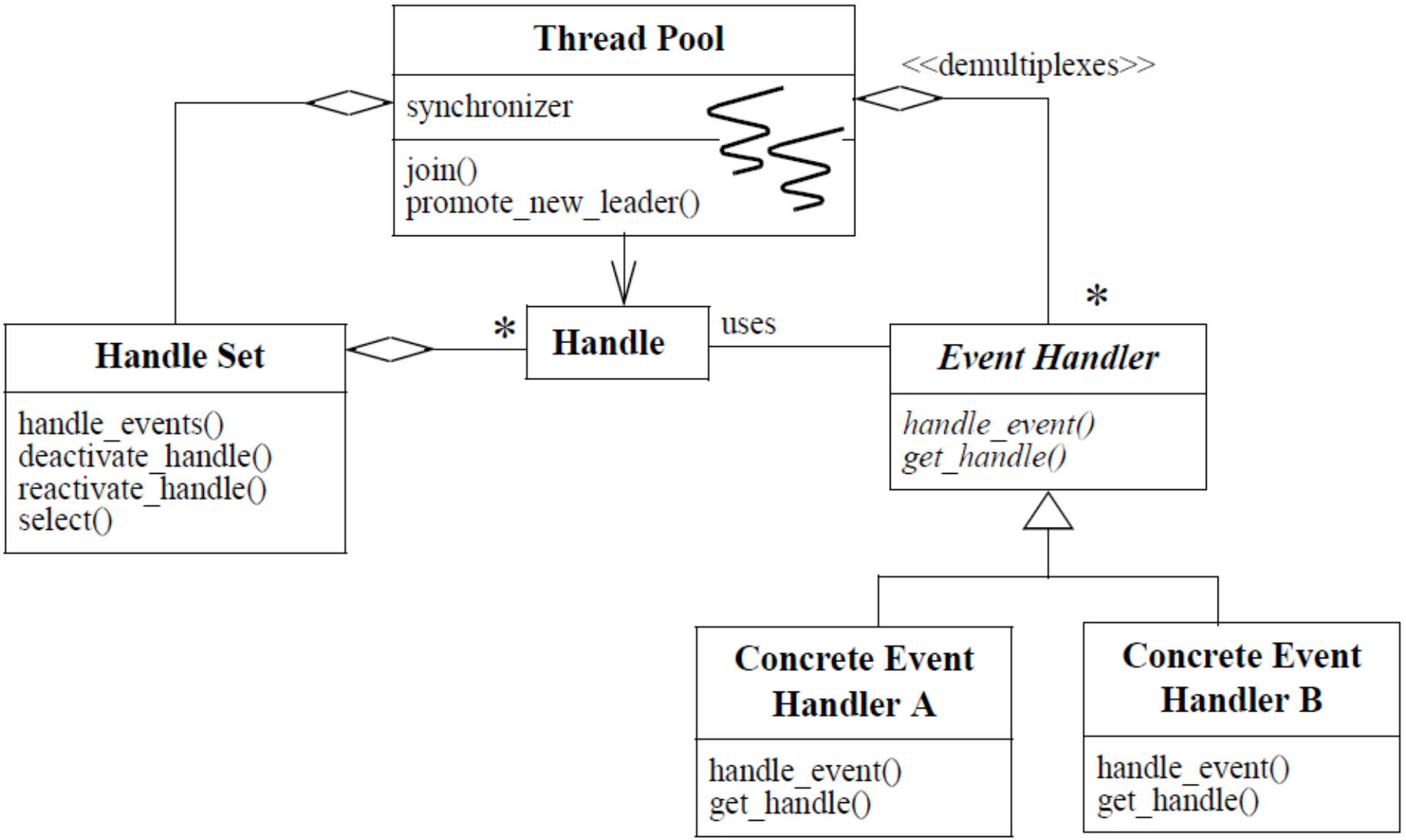
Leader/Followers – problémy a řešení

- **Problémy**
  - chceme efektivní demultiplexování událostí
  - nutné omezit režii – přepínání kontextů, synchronizace, cache, alokace
  - koordinace vláken při demultiplexování – ochrana před race conditions
- **Řešení**
  - události demultiplexuje **více** vláken
  - tato vlákna **se střídají** v demultiplexování událostí
  - přijatá událost je synchronně předána příslušné službě ke zpracování

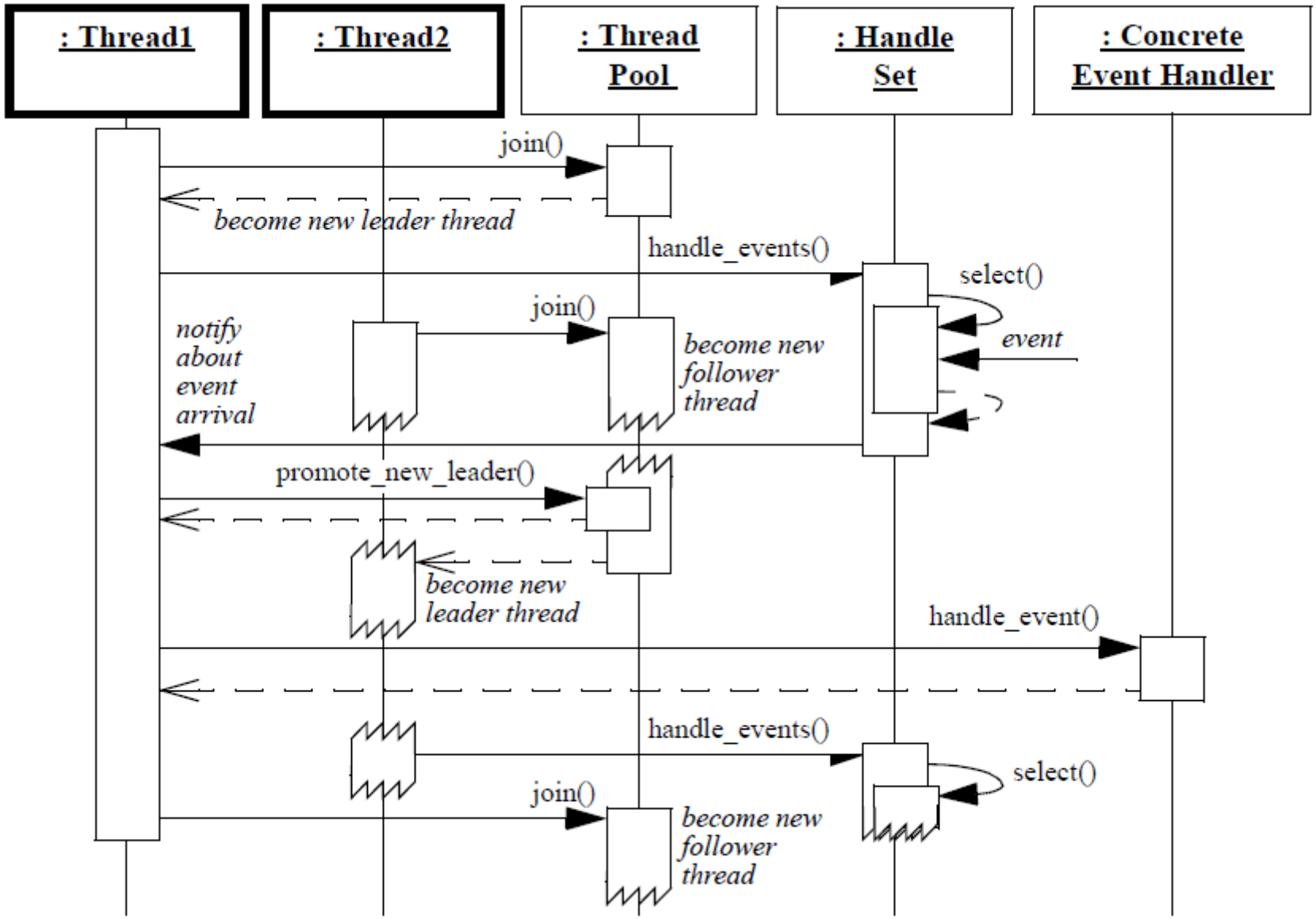


Leader/Followers, Douglas C. Schmidt, Carlos O'Ryan, Michael Kircher, Irfan Pyarali, and Frank Buschmann, 2000

Leader/Followers – struktura



Leader/Followers – dynamické chování



Leader/Followers, Douglas C. Schmidt, Carlos O'Ryan, Michael Kircher, Irfan Pyarali, and Frank Buschmann, 2000

Leader/Followers – varianty

- **Varianty**
  - **Bound Handle/Thread Associations**
    - k vláknům jsou přiřazeny (bound) handles
    - obyčejný Leader/Followers přiřazení nemá (unbound)
  - **Multiple Handle Sets**
  - **Multiple Leaders and Multiple Followers**
  - **Hybrid Thread Associations** – vlákna bound i unbound
  - **Hybrid Client/Servers** – přiřazení vláken se muže měnit
  - **Alternative Event Sources and Sinks**

- **Další příklady**
  - webové servery
  - CORBA Object request brokery
  - příklad ze života – stanoviště taxíků

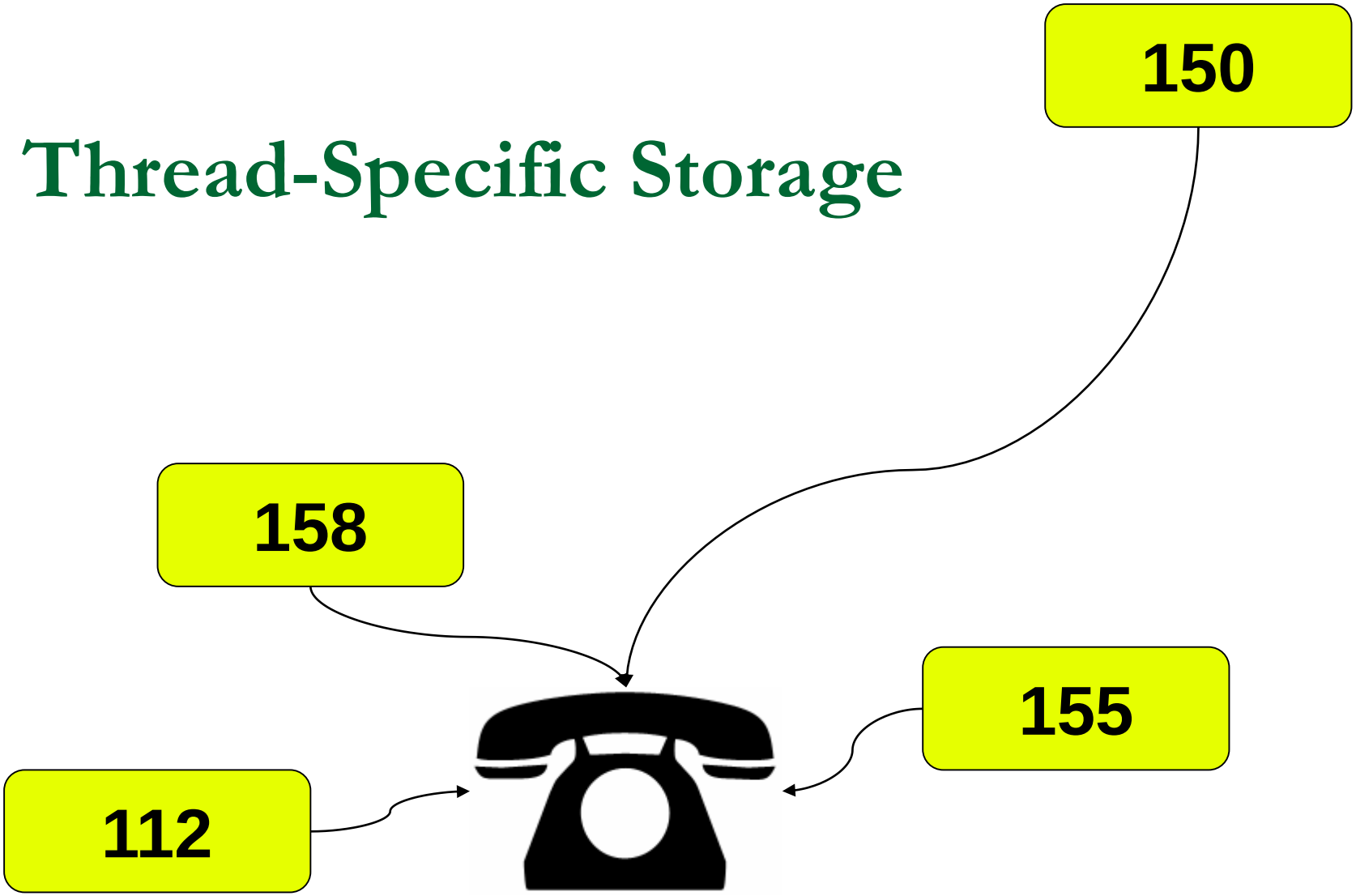
Leader/Followers – shrnutí

- **Výhody**
  - efektivita, vyšší výkon
  - jednoduchost programování
- **Nevýhody**
  - složitější implementace
  - menší flexibilita
  - potenciální bottleneck – pouze jedno vlákno pro I/O

Srovnání s Half-Sync/Half-Reactive

- **Alternativy**
  - **Half-Sync/Half-Async**
  - **Active Object**
  - **Reactor** – když je zpracování událostí krátké
  - **Proactor** – pokud nám nevadí asynchronní zpracování a OS to umí

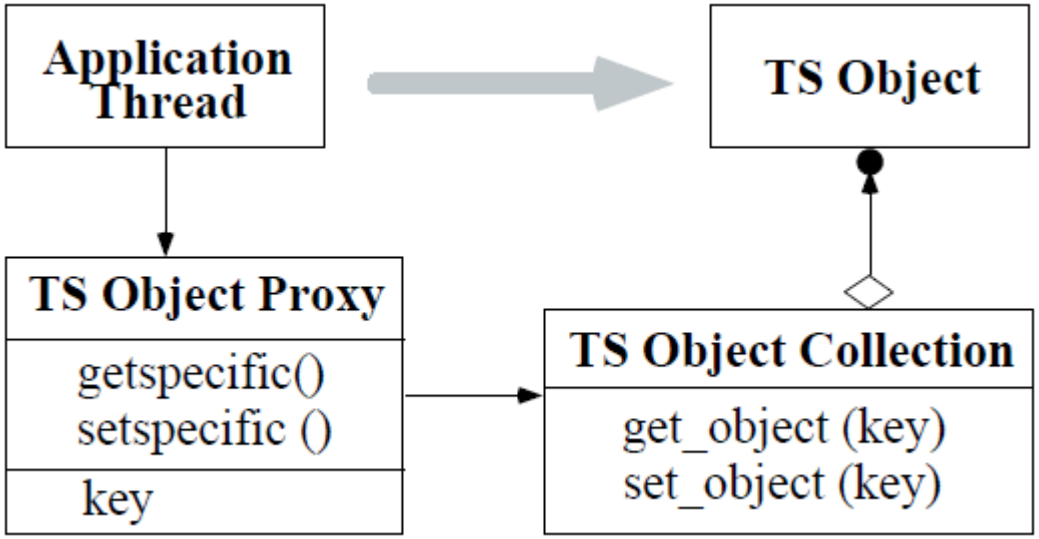
Thread-Specific Storage



Thread-Specific Storage – návrhový vzor

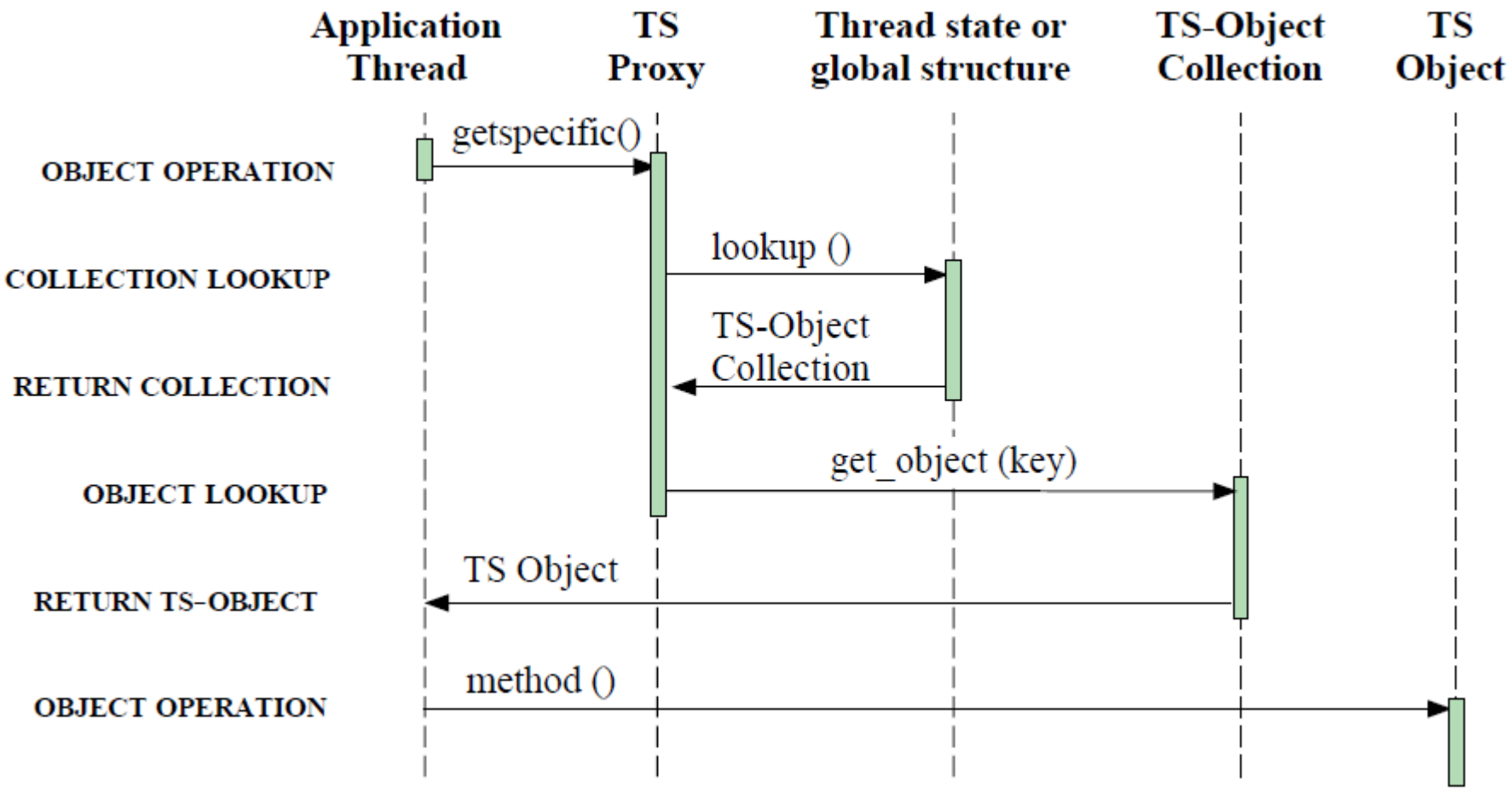
- **Účel**
  - umožnit vícevláknovým aplikacím přistupovat k fyzicky lokálnímu objektu přes logicky globální přístupový bod
  - a to transparentně, bez použití zámků
- **Příklad**
  - errno – error number
- **Problémy**
  - použití fyzicky globálního objektu bez zámků vyvolá race conditions
  - jeho zamykání je pomalé a netransparentní
  - chceme jednoduché a efektivní použití
  - nechceme použít zamykání
  - staré jednovláknové knihovny používají globální objekty
    - není je možné je upravit, bez rozbití dalšího kódu
    - je třeba je nějak ošálit
- **Řešení**
  - mít výhradní globální přístupový bod k objektu
  - objekt ale uložit vůči vláknu lokálně

Thread-Specific Storage – struktura

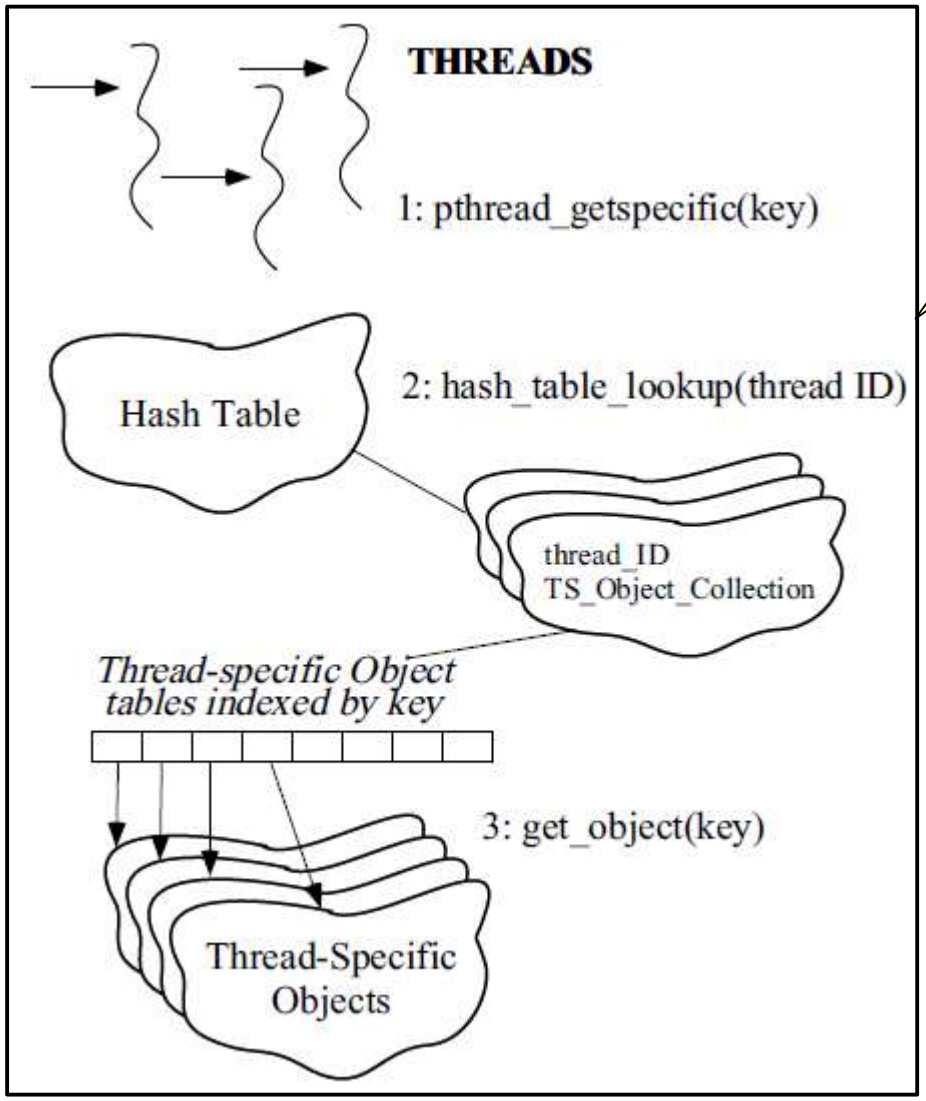


Thread-Specific Storage for C/C++: An Object Behavioral Pattern for Accessing per-Thread State Efficiently, Douglas C. Schmidt, Nat Pryce, Timothy H. Harrison, 1997

Thread-Specific Storage – dynamické chování



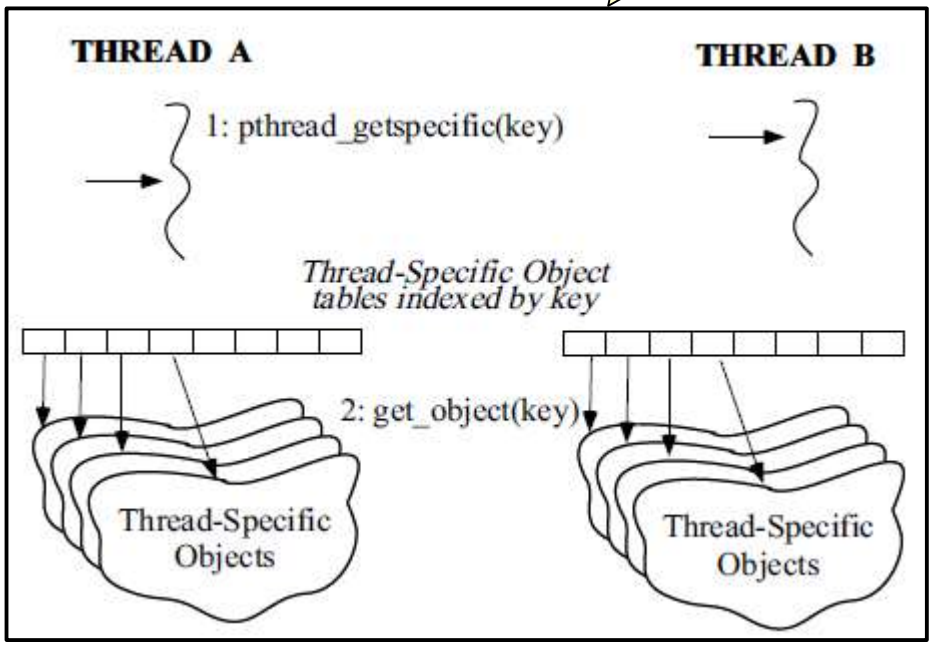
Thread-Specific Storage – varianty



externě

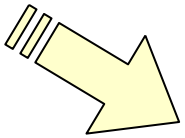
interně

Kam uložit TS Object Collection?



Thread-Specific Storage – příklady použití

- **Operační systémy**
  - implementace TSS přímo v platformě (**Win32**, **Solaris**) nebo knihovně (**pthreads**)
  - **errno** – chybový kód poslední chyby
- **Podpora TSS v různých jazycích**
- **OpenGL**
- **Příklad ze života**
  - telefonní čísla
  - 112, 150,155, 158, ...



GNU C

```
__thread int localNumber;
```

Visual C++

```
__declspec(thread) int localNumber;
```

C#

```
class Foo {
   [ThreadStatic] static int localNumber;
}
```

Java

```
ThreadLocal<Integer> localNumber =
  new ThreadLocal<Integer>(){
    @Override protected Integer initialValue(){
        return 1;
    }
};
```

Thread-Specific Storage – shrnutí

- **Výhody**
  - efektivita
  - znovupoužitelnost
  - jednoduché použití
  - přenositelnost

- **Nevýhody**
  - podporuje používání globálních objektů
  - zamlžuje skutečnou strukturu systému
  - občas omezené možnosti při implementaci

Concurrency Patterns – shrnutí

- **Active Object**
- **Monitor Object**
- **Half-Sync/Half-Async**
- **Leader/Followers**
- **Thread-Specific Storage**