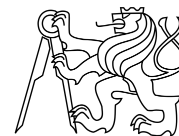
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	REST a transak ní zpracování
Student:	Bc. Jakub Drábek
Vedoucí:	Ing. Pavel Krej í
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Definujte pojmy REST, SOAP, middleware a transak ní zpracování. Analyzujte architekturu REST pro pot eby transak ního zpracování a vyberte konkrétní koncepty, které pracují na middleware a storage vrstv . Diskutujte výhody a nevýhody jednotlivých koncept a vyberte nejvhodn jší metodu pro nasazení ve vybrané bankovní organizaci. Stanovte hypotézy pro použití vybrané metody a ov te je formou prototypové implementace i výzkumu v kontextu IT systém vybrané bankovní organizace. Zhodno te výsledky a stanovte klí ové faktory pro nasazení vybrané metody. Pokud žádná z existujících metod nevyhovuje, navrhn te a ov te své ešení. Navrhn te zadávací dokumentaci pro nasazení vybraného ešení.

Seznam odborné literatury

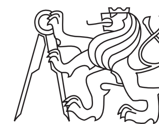
Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
d kan

V Praze dne 25. prosince 2016

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství



Diplomová práce

REST a transakční zpracování

Bc. Jakub Drábek

Vedoucí práce: Ing. Pavel Krejčí

7. května 2017

Poděkování

Děkuji vedoucímu Ing. Pavlu Krejčímu za rady, náměty, čas a prostředky poskytnuté při psaní této práce. Dále chci poděkovat své rodině a blízkým přátelům za neustálou podporu během mého studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 7. května 2017 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2017 Jakub Drábek. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Drábek, Jakub. *REST a transakční zpracování.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2017.

Abstrakt

Tato práce se věnuje problematice využití transakčního zpracování v architektonickém rozhraní REST. Cílem práce je analyzovat a vybrat konkrétní koncepty pro potřeby zavedení transakčního zpracování v architektuře REST.

Dále z vybraných konceptů nalézt a ověřit nejvhodnější metodu pro nasazení v rámci bankovní organizace. V případě nenalezení žádné z vyhovujících metod je cílem navrhnout a ověřit své vlastní řešení.

U vybraných konceptů jsou zde diskutovány jejich výhody a nevýhody. U metody vybrané k implementačnímu ověření jsou popsány klíčové faktory, které vedly pro toto rozhodnutí a závěrečné vyhodnocení ověření. Dále je součástí zaváděcí dokumentace pro nasazení vybraného řešení a služeb realizující příklad použití. Závěrem práce je vyhodnocení a diskuze k zavedení transakčního zpracování do architektonického rozhraní REST.

Klíčová slova REST, transakce, SOA, middleware

Abstract

This thesis deals with the use of transactional processing in the architectural interface REST. The aim of this thesis is to analyze and select specific concepts for transaction processing in the REST architecture.

Then I choose and verify the best method for deploying within the bank organization. If no suitable method is found, the goal is to create and validate a customized solution.

For selected concepts are discuss their advantages and disadvantages. The methods selected for implementation verification describe the key factors that led to this decision and the final evaluation. Further is also part of the deployment documentation for deploying a selected solution and service providing an example of use. The conclusion of the thesis is the evaluation and discussion to the introduction of the transaction processing into the architectural interface REST.

Keywords REST, transaction, SOA, middleware

Obsah

Úvod **1**

1 Úvod do problému **3**
- 1.1 Základní definice . . . 3
- 1.2 REST a transakční zpracování . . . 14
- 1.3 Alternativní řešení . . . 16

2 Nalezená řešení **21**
- 2.1 Transakce jako zdroj . . . 21
- 2.2 Návrh koordinátora . . . 33
- 2.3 Použití Try-Cancel/Confirm návrhového vzoru . . . 49
- 2.4 Další nalezená řešení . . . 58
- 2.5 Shrnutí . . . 58

3 Ověření vybrané metody **61**
- 3.1 Vybraná metoda . . . 61
- 3.2 Implementační detaily . . . 67
- 3.3 Zaváděcí dokumentace . . . 69
- 3.4 Příklad použití . . . 70
- 3.5 Zhodnocení ověření implementací . . . 71

Závěr **73**

Literatura **75**

A Seznam použitých zkratek **79**

B Snímky průchodu příkladu použití **81**

C Obsah přiloženého CD **85**

Seznam obrázků

1.1 Stavový diagram transakce 12
1.2 Diagram 2PC protokolu 13
1.3 Transakční Message Queue [1] 17

2.1 Schéma komunikace - transakce jako zdroj 28
2.2 Schéma komunikace s použitím transakčního koordinátora 33
2.3 Stavový diagram transakce s 2PC protokolem 34
2.4 Schéma TTC protokolu 2.3.1 49
2.5 Schéma protokolu 50
2.6 Schéma úspěšného scénáře pro TCC protokol 53

3.1 Schéma implementovaného řešení 62

B.1 První fáze registračního formuláře 81
B.2 Druhá fáze registračního formuláře - úspěch 82
B.3 Výsledek úspěšné registrace 82
B.4 Druhá fáze registračního formuláře - neúspěch 83
B.5 Zobrazení výskytu chyby 83

Úvod

Systémová integrace, jakožto pojem v technickém pojetí, kde se jedná o integraci různých částí informačního systému (aplikací) do jednoho celku, je v moderním IT světě již zaběhlý pojem. Cílem je mít takovou architekturu informačního systému, která efektivně podporuje podnikové procesy (business processes) v organizaci.

Jedním z typů systémové integrace, která získává stále na větší popularitě, je integrace pomocí služeb (servisně orientovaná architektura), která se nejčastěji využívá pro integraci systémů s uživatelským rozhraním nebo integraci procesů běžících v reálném čase.

Servisně orientovaná architektura se v současné době posouvá mimo své tradiční kořeny ve styl vzdáleného volání procedur, tak aby byly zahrnuty formy interakce, které jsou například obsaženy v `REST`, webově orientovaných architekturách, architekturách řízenými událostmi apod.

Hlavně v době, kdy se začíná objevovat termín `Web` 2.0, dochází k většímu důrazu na zjednodušení programovacího modelu pro jednoduché, obsahově orientované aplikace, které se převážně vyznačují kompozicí obsahu z více nezávislých zdrojů pomocí `API` založených na principu `REST`.

Vzrůstající popularita architektonického rozhraní `REST` sebou však nese některá omezení pro systémy, které pomocí této technologie spolu komunikují. Jedním z takovýchto omezení je transakční zpracování, které z definice architektury `REST` není podporováno (viz. dále). To přináší určité komplikace a zanesení složitosti do konceptu služeb a procesů probíhajících např. v bankovních organizacích, kde pro mnohé procesy je princip transakčního zpracování nezbytný.

Tato práce se právě zaměřuje na analýzu a způsoby zavedení transakčního zpracování do architektonického rozhraní `REST`. Dále se věnuje prototypovému ověření jedné vybrané metody s cílem pro nasazení v rámci bankovní organizace.

Kapitola **1**

Úvod do problému

1.1 Základní definice

Téma této práce, problematika `REST` a transakční zpracování se týká oblasti komunikace v distribuovaných prostředích. Pro seznámení s touto problematikou a lepším pochopení uvedených řešení je nutné nejprve uvést základní definice a pojmy související s tímto tématem.

1.1.1 SOA

První důležitým pojmem, nebo spíše sadou pojmů, je SOA (Service Oriented Architecture), servisně orientovaná architektura, jedná se o sadu principů, metodologií a technologií, které doporučují jak skládat složité aplikace a jiné systémy ze skupiny na sobě nezávislých komponent poskytujících služby.

Službou je zde myšlena komponenta, která poskytuje určitou přidanou hodnotu v rámci celku. Jednotlivé komponenty lze z pravidla různě kombinovat, doplňovala nebo nahradit jednu komponentu druhou. Každá komponenta je tedy diskrétní jednotkou určité funkcionality dostupné vzdáleně.

Jednotlivé služby jsou poskytovány ostatním komponentám pomocí komunikačního protokolu přes síť a jsou nezávislé na konkrétním produktu, dodavateli a technologii. Jedná se o návrhový vzor `request/reply`, kde komunikační protokoly popisují, jak předávat a přijímat zprávy za použití popisných metadat. [2]

Základní hodnoty SOA

Manifest `SOA` byl zveřejněn v říjnu 2009 a přichází se šesti základními hodnotami:

1. **„Bussiness“ hodnota** – podnikové hodnotě je dán větší význam než technické strategii.

2. **Strategické cíle** – strategické cíle mají větší význam než benefity spojené s konkrétním projektem.

3. **Vnitřní interoperabilita** – interoperabilitě je dán větší význam než vlastní integraci.

4. **Sdílené služby** – sdílené služby jsou upřednostňovány před účelovými službami.

5. **Flexibilita** – flexibilita je významnější než optimalizace.

6. **Evoluční zpřesňování** – postupné uhlazování má větší význam než snaha o počáteční dokonalost.

Základní principy SOA

Hlavním principem `SOA` je orientace na služby a použití v různých podmínkách, ten zastřešuje osm základních principů, které jsou:

1. **Standardizovaný kontrakt** – servisní kontrakt poskytované služby je jasně definován. Pro to je nutné již při návrhu SOA komponenty zvážit technická rozhraní a formát data.

2. **Slabé vazby mezi komponentami** – vazby mezi jednotlivými komponentami musí být co nejužší[1]. Závislosti jsou dodefinovány těsně před použitím a externě, tedy mimo vyvinutou službu. To vede k redukci závislostí kontraktu služby, její implementační logiky a konzumentů.

3. **Princip abstrakce** – úkolem principu abstrakce je skrýt implementační detaily služby, nebo komponenty. Tento princip je hlavně určeno pro skládání složitějších systémů, podporuje princip volných vazab a vede na zlepšení granularity systémů.

4. **Znovupoužitelnost** – navrhovaná služba nebo komponenta by měla být využitelná i jinde než pro aktuální projekt. Toto závisí na dobrém architektonickém návrhu.

5. **Nezávislost** – pro dodržení definovaného kontraktu musí služba obsahovat vnitřní nezávislou logiku pro správu zdrojů, ze kterých čerpá. Tento princip vyžaduje dodržení pravidel o izolaci a normalizaci služeb.

6. **Bezstavovost** – princip bezstavovosti umožňuje znovupoužití a možnost škálovatelnosti.

[1] Aplikace, která přijímá data od jiné, musí rozumět cizí reprezentaci dat. To by mělo být zajištěno pokud možno co nejvíce univerzálně.

7. **Princip identifikovatelnosti** – tento princip má převahu spíše v ekonomickém opodstatnění. Služba by měla být lehce identifikovatelná, jakožto i způsob jejího použití. Identifikovatelnost je v tomto případě zajištěna pomocí WSDL [3].

8. **Princip skládání** – princip zajišťující možnost seskládání jednotlivých služeb a zajištění jejich vzájemné synergie.

Implementační přístupy

Nejčastějším způsobem implemetace SOA je pomocí webových služeb. Toho je dosaženo dostupností funkčních stavebních bloků (služeb) přes standardní internetové protokoly, které jsou nezávislé na platformě či programovacím jazyku.

Architektura funguje nezávisle na konkrétní technologii, proto lze SOA implementovat pomocí široké škály technologií, jako jsou například:

- Web services založené na WSDL a SOAP,
- Messaging [4],
- RESTful HTTP s použitím architektonického stylu REST,
- WCF[2] frameworku [5],
- apod.

1.1.2 Representational State Transfer

Representational State Transfer (REST) je architektonické rozhraní, určené pro distribuované systémy poskytující interoperabilitu mezi počítačovými systémy přes internet. Definuje množinu architektonických principů, pomocí nichž lze navrhnout webové služby zaměřené na zdroje daného systému, včetně toho, jak jsou řešeny jednotlivé stavy zdrojů a jejich přenos přes HTTP protokol.

Hlavní principy REST

REST se skládá z množiny architektonických principů. Zde nalezneme hlavní principy tohoto architektonického rozhraní [6]:

1. **Client-Server** – oddělením uživatelského rozhraní od datového úložiště můžeme zvýšit přenositelnost uživatelského rozhraní napříč různými platformami a zlepšit škálovatelnost zjednodušením serverové komponenty.

[2] Windows Communication Framework – sada knihoven pro tvorbu servisně-orientovaných aplikací.

2. **Bezstavovost** – nelze využít jakýkoliv uložený kontext na serveru. Tedy, každý požadavek od klienta musí obsahovat veškeré informace nezbytné pro pochopení požadavku a stav relace je zcela zachován na straně klienta.

3. **Uložitelný do mezipaměti** – tento princip vyžaduje, aby data v odpovědi na žádost byly implicitně nebo explicitně označeny jako `cacheable`, nebo `non-cacheable`. V případě `cacheable` má klient právo znovu použít data odpovědi později.

4. **Jednotné rozhraní** – `REST` je definován čtyřmi omezeními na rozhraní pro uplatnění důsledku principu softwérového inženýrství obecnosti na rozhraní. Tato omezení jsou:

   - identifikace zdrojů,
   - manipulace se zdroji pomocí reprezentací,
   - sebepopisné zprávy,
   - využití „hypermédií“ jako motoru stavu aplikace.

5. **Vrstvený systém** – vrstvený styl sytémů umožňuje strukturu, která se skládá z hierarchické struktury omezování chování komponent tak, aby každá složka nebyla viditelná mimo vrstvu, s kterou je v interakci.

6. **Kód na vyžádání** – volitelný princip, který umožňuje stahovat a spouštět kód ve formě appletů a skriptů na straně klienta.

Aplikace `REST` na webové služby

`API` webových služeb, které odpovídá architektonickým omezením `REST`, se nazývá `RESTful`. `RESTful API` založené na `HTTP` protokolu je definováno následujícími aspekty [7]:

- bezstavové,
- struktura založena na `URI`,
- internetový typ média, který definuje přechod stavů datových prvků, viz. zdroj,
- explitcitní použití standardních `HTTP` metod (`OPTIONS`, `GET`, `PUT`, `POST` a `DELETE`).

Zdroj

Zdroj, klíčová abstrakce informace v REST, rezprezentuje jakoukoliv informaci, která může být pojmenována (dokument, obrázek, pomocná služba, kolekce zdrojů, nevirtuální objekt apod.). Každý zdroj má svůj jedinečný identifikátor sloužící zejména k interakci mezi jednotlivými komponentami.

Stav každého zdroje v konkrétním čase, reprezentace zdroje, se skládá z dat, metadat popisujících data a hypermediálního odkazu (identifikátor, v RESTful reprezentovaný hypertextovým odkazem). Odkaz zde slouží klientovi k dosažení přechodu do dalšího požadovaného stavu.

Reprezentace zdroje musí být sebepopisná, to znamená, že klient nemusí vědět, co kokrétní zdroj reprezentuje, ale musí jednat na základě jeho media-typu. Každý media-typ reprezentuje základní procesní model. [8]

Zdroj v RESTful

V RESTful designu je využito standardních HTTP metod, které jsou mapovány mezi CRUD operacemi (create, read, update, delete). [9] Toto mapování je:

- POST – slouží k vytvoření nového zdroje (reprezentace create),
- GET – slouží k obdržení zdroje (reprezentace read),
- PUT – slouží ke změně stavu zdroje, nebo jeho aktualizaci (reprezentace update),
- DELETE – slouží k odebrání, nebo smazání zdroje (reprezentace delete).

Více o architektonickém rozhraní REST naleznete například v [6].

1.1.3 SOAP

Simple Object Access Protocol (SOAP) je jednoduchý protokol určený k výměně struktorovaných a typovaných informací v decentralizovaném distribuovaném prostředí. SOAP sám o sobě nedefinuje žádnou aplikační sémantiku, spíše definuje mechanismus pro její vystavení poskytnutím zabalení modelu do modulů a kódováním dat uvnitř modulů.

Hlavními charakteristikami tohoto protokolu jsou:

- rošiřitelnost (bezpečnost, WS směrování, ...),
- neutralita – SOAP může pracovat nad různými protokoly jako jsou HTTP, SMTP, TCP, UDP, nebo JMS,
- nezávislost – SOAP umožňuje libovolný programovací model.

SOAP zpráva je založena na XML. Skládá se ze tří funkčně ortogonálních částí:

1. SOAP envelope – konstrukt definující framework pro popis, **co** je ve zprávě, **kdo** by se tím měl zabývat a **zda** je zpráva volitelná nebo povinná.
2. SOAP kódovací pravidla - sady pravidel, které popisují mechanismus serializace, který může být použit pro výměnu aplikačně definovaných datových typů.
3. SOAP RPC reprezenace - RPC[3] reprezentace definuje konvenci, která může být použita pro reprezentaci vzdálených procedur a odpovědí.

Hlavním cílem SOAP je jednoduchost a rozšiřitelnost. Existuje zde několik funkcí z tradičních systémů pro zasílání zpráv a systému pro distribuované objekty, které nejsou součástí SOAP specifikace.

Architektura tohoto protokolu se skládá z několika vrstev specifikace:

- formát zpráv,
- Message Exchange Patterns [10],
- vazby na přenosový protokol,
- model zpracování zpráv,
- protokol rozšiřitelnosti.

SOAP zprávy jsou v podstatě jednosměrné komunikace odesílatele k přijímači, ale často jsou kombinovány k implementaci vzorů jako je request/response [11]. Implementace může být uzpůsobena tak, aby byly využity jedinečné charakteristiky jednotlivých síťových systémů. Příkladem může být HTTP spojení, které poskytuje SOAP odpovědi jako HTTP pomocí stejného připojení, jakým byla přijata příchozí žádost. [12]

Příklad SOAP zpráv

Na následujícím příkladu je zobrazen SOAP požadavek GetLastTradePrice zaslaný na službu StockQuote. Požadavek přebírá textový parametr - burzovní symbol a vrací číselnou hodnotu v odpovědi. SOAP envelope element v hlavičce dokumentu reprezentuje SOAP zprávu.

[3] Remote Procedure Call – vzdálené volání procedur.

Požadavek:

Hlavička:

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"
```

Obsah:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
     <m:GetLastTradePrice xmlns:m="Some-URI">
        <symbol>DIS</symbol>
     </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Odpověď:

Hlavička:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
```

Obsah:

```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
     <m:GetLastTradePriceResponse xmlns:m="Some-URI">
        <Price>34.5</Price>
     </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Více informací o SOAP se dozvíte v [13].

1.1.4 Middleware

Middleware je označení pro specializovaný software poskytující ostatním aplikacím služby, které jsou nad rámec slůžeb poskytovaných operačním systémem (označovaný jako `software glue`). Jedná se o software spojující softwarové komponenty, nebo podnikové aplikace a usnadňuje vývojářům vývoj komunikace a vstupů/výstupů.

Middleware představuje vrstvu ležící mezi operačním systémem a ostatním softwarem v distribuovaném prostředí. Řídí interakce mezi odlišnými aplikacemi napříč heterogenními platformami (spojení s aplikačními servery, redakčními systémy a podobnými nástroji, které podporují vývoj a dodávku aplikací).

Typický middleware je založen na technologiích `XML`, `JSON`, `SOAP`, `REST`, webových službách a servisně orientované arhitektuře. Systematická vazba různorodých aplikací, často za použití middleware, je známá jako Enterprise Application Integration [14].

1.1.4.1 Typy middleware

1. **Message Oriented Middleware** – kategorie zahrnující asynchronní uložiště, aplikace pro zasílání zpráv, integrační brokery pro transformaci zpráv, směrování a koordinaci obchodních procesů.

2. **Object Middleware** – kategorie zahrnující převážně Object Request Brokery (`ORB`) [15].

3. **RPC Middleware** – middleware, který poskytuje volání procedur vzdálených systémů. Na rozdíl od Message Oriented Middleware se jedná o synchroní interakci mezi systémy.

4. **Databátový Middleware** – databázový middleware umožňuje přímý přístup do datových struktur a zajišťuje přímé spojení s databází. Jedná se o různé databázové brány s různými možnostmi připojení.

5. **Transakční Middleware** – kategorie zahrnující Transaction Processing Monitory [16] a webové aplikační servery.

6. **Portály** – kategorie enterprise portal serveru. Ty jsou zde zahrnuty z důvodu usnadnění pro `front-end` integraci. Umožňuji interakci mezi stolními počítači, `back-end` systémy a službami.

Nejedná se o kompletní seznam typů middleware. Avšak ostatní typy lze zahrnout do těchto uvedených kategorií. [17]

1.1.5 Transakční zpracování

Transakční zpracování je označení pro skupinu logických operací (označených jako transakce), které musí být vždy provedeny jako jeden celek. V případě, že se při zpracování vyskytne jakákoliv chyba a skupina operací nemůže být dokončena, musí dojít k navrácení všech dílčích operací do stavu před začátkem transakce. Do transakce může být zahrnuto dva či více transakčních zdrojů (databáze, `JMS`, `EIS`, ...) nad kterými dochází k jednotlivým operacím.

Transakci, která ovlivňuje právě více zdrojů označujeme jako distribuovaná transakce. Zde každý z účastníků musí potvrdit a zaručit, že všechny provedené změny budou trvalé. Typickým představitelem transakčního zpracování jsou databázové transakce.

Systém, který podporuje transakční zpracování (transakční systém) je definován pomocí `ACID` vlastností [18]:

- **Atomicita** (Atomicity) – změny uvnitř transakce jsou atomické (buď je provedeno vše nebo nic).
- **Konzistence** (Consistency) – změny v transakci vedou ke korektnímu stavu. Nedochází k porušení z žádných integračních omezení.
- **Izolace** (Isolation) – v případě spuštění více transakcí současně se systém jeví tak, že nějaká transakce T byla spuštěna před anebo po jiné transakci. Tedy, operace transakce jsou skryty před vnějšími operacemi.
- **Trvanlivost** (Durability) – jakmile je transakce úspěšně dokončena, její změny jsou skutečně uloženy.

Pro práci s transakcemi jsou využívány tyto tři základní operace:

- `begin` – začátek transakce,
- `commit` – potvrzení úspěšné transakce, ukončí transakci a uloží dosažené výsledky,
- `rollback` – odvolání změn.

Metodologie

V kontextu transakčního zpracování se setkáváme s následujícími pojmy [18]:

- `Rollback` – v případě, že některá část transakce selže před potvrzením transakce (před provedením `commit`), dojde k obnově původního stavu systému, který se nacházel před započetím transakce. Toho může být dosaženo například pomocí záznamů mezilehlých stavů systému.
- `Rollforward` – možnost držení žurnálu všech separátních modifikací. To je užitečné v případě celkového selhání systému a nutnosti ho obnovit z posledních záloh.

- `Deadlock` – `deadlock` nastává v případě, že dvě a více transakcí se v průběhu jejich zpracování pokusí přistupovat ke stejné části systému ve stejné době, způsobem, který jim brání řízení. V tomto případě typicky dochází ke zrušení a `rollbacku` těchto transakcí a jejich opětovnému vyvolání v jiném pořadí.

- `Compensating transaction` – jedná se o transakci sloužící k odstranění neúspěšných transakcí a obnovení systému do předchozího stavu u systémů, které nepodporují mechanismy `commit` a `rollback`.

Stavy
Stavy, ve kterých se transakce může vyskytovat:

- **Aktivní** (`Active`) – stav po zahájení transakce.

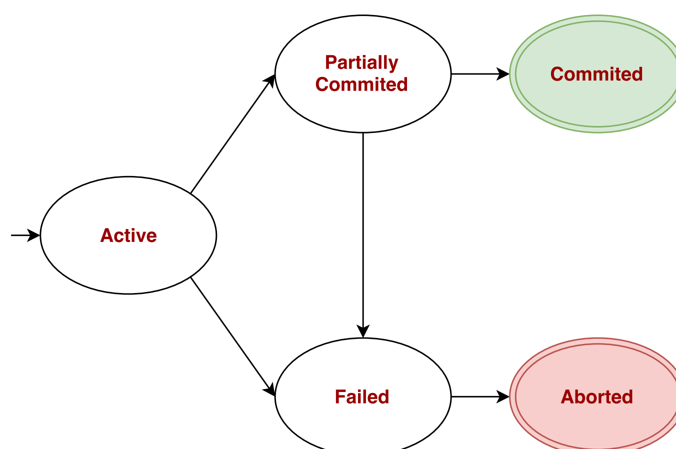
- **Částečně potvrzená** (`Partially Commited`) – stav po provedení poslední operace transakce.

- **Chybná** (`Failed`) – stav, kdy již nelze pokračovat v normálním průběhu transakce.

- **Zrušená** (`Aborted`) – po skončení operace `rollback`, tj. uvedení systému do původního stavu.

- **Potvrzená** (`Commited`) – po úspěšném zakončení, po vykonání operace `commit`.

Přechod mezi stavy je zachycen na následujícím schématu:



Obrázek 1.1: Stavový diagram transakce

1.1.5.1 Two-phase commit protokol

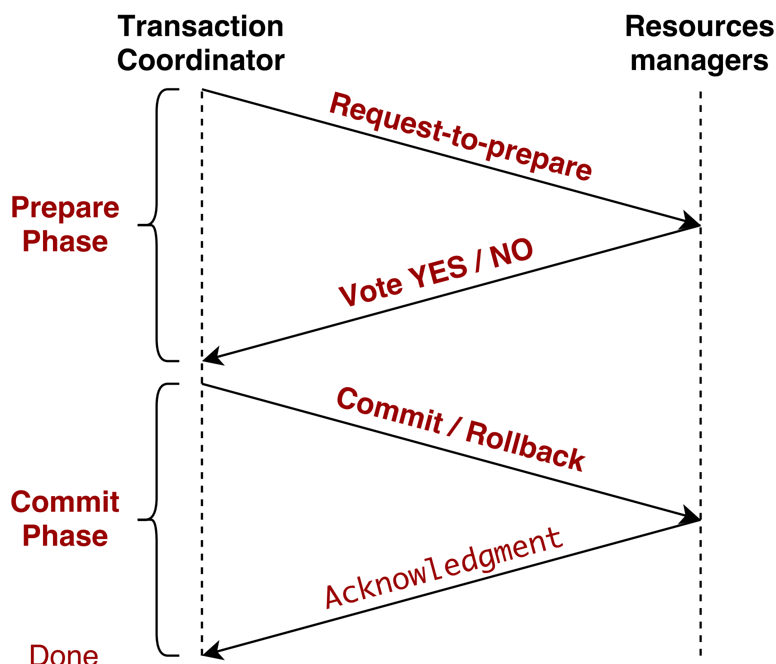
Algoritmus využívaný v transakčních manažeřích, databázových systémech apod. pro zajištění atomicity distribuovaných transakcí. Koordinuje všechny transakční zdroje v rámci transakce.

Tento algoritmus, označovaný jako `2PC`, se skládá ze dvou fází:

První fáze Fáze přípravy (`Prepare Phase`). V této fázi všechny zůčastněné strany, které odeslaly zprávu do transakčního manažera, ho informují, že jsou připraveny a schopny vykonat svou část operace. Všechny strany jsou tedy připraveny buď potvrdit, nebo odvolat své vyvolané změny. V případě, že transakční manažer od každé zúčastněné strany obdrží potvrzení, postupuje se do druhé fáze.

Druhá fáze Fáze potrvzení (`Commit Phase`). V této fázi transakční manažer informuje každou zúčastněnou stranu, zda má provést operaci potvrzení (`commit`), nebo operaci vrácení (`rollback`) všech provedených změn, které byly součástí transakce. Správně provedený `rollback` musí vrátit systém do původního stavu. [19]

Diagram tohoto protokolu:



Obrázek 1.2: Diagram `2PC` protokolu

1.2 REST a transakční zpracování

1.2.1 Transakční zpracování v REST

Architektonické rozhraní `REST` je definováno sadou principů a omezení uvedených v sekci 1.1.2. Jedním z těchto omezení je **bezstavovost**, která je definována:

Na straně serveru, v komunikaci klient-server, nedochází k ukládání kontextu požadavku mezi jednotlivými klientskými dotazy. Každý dotaz od libovolného klienta obsahuje všechny potřebné informace pro vykonání požadované operace a jakýkoliv stav relace se koná na straně klienta. [20]

Toto omezení je jednou z hlavních příčin, proč `REST`, tak jak je definován, nepodporuje transakční zpracování. Důvodem je, že v případě vykonávání nějaké komplexní akce zahrnující více služeb (zdrojů) s definovanými omezeními, je třeba přenášet stav. Ten reprezentuje, v jaké fázi se vykonávaná transakce nachází. To například může ovlivnit viditelnost provedených změn vůči ostatním klientům dotazujících se na jednotlivé služby zúčastněné v transakci. Zde je ale v případě použití rozhraní `REST` stav zdroje spravován na straně serveru a stav relace je uchováván na straně klienta.

Porušení omezení bezstavovosti obvykle vede ke špatné škálovatenosti, zde je poté nutné synchronizovat probíhající transakce mezi jednotlivými instancemi zpracování.

V případě použití transakce je dále potřeba, aby každý účastník transakce podporoval funkcionalitu operací `rollback` a `commit` nebo, aby byl zajištěn způsob pro jejich vykonání. To nemusí být praktické nebo i vhodné pro synchronní způsob komunikace. Dále zde vyvstávají otázky ohledně implementace a provedení dalších operací, které jsou spojeny s transakčním zpracováním, jako jsou například:

- časový limit (`timeout`) pro potvrzení transakce,
- způsob odstranění vzniklých chyb,
- znovu-opakovatelnst operace v transakci,
- apod. [21]

1.2.2 Zavedení transakčního zpracování v architektuře REST

Transakční zpracování, i přes odlišné názory v IT světě a problémy uvedenými výše, má v mnohých případech své opodstatnění a vznikají situace, kdy je vhodné jeho zavedení i vně systémové integrace. Tato potřeba se nachází například v bankovních organizacích, kde jednotlivé systémy stále častěji využívají pro vystavení svých služeb právě rozhraní `REST`.

Avšak, jsou zde kladeny požadavky, které by mělo případné řešení splňovat. Těmito požadavky jsou:

- Zachování `ACID` vlastností.
- Neinvazivní řešení – v případě zavedení řízené transakce zde vzniká úzká vazba mezi službami komunikujícími uvnitř transakce, to zavádí další komplexnost do jejich návrhu. Proto by řešení mělo být co nejméně invazivní, tak, aby nebyla nutnost dále upravovat služby zúčastněné v transakčním zpracování.
- Malá komunikační zátěž – požadavek na co nejmenší nárůst komunikační zátěže z důvodu nárůstu zatížení na jednotlivé systémy a prodloužení doby vykonání celkového procesu. Za přijetalnou hodnotu lze považovat navýšení do 50% provedené komunikace v běžném případě. Horní hranicí lze označit dvojnásobný a vyšší nárůst.
- Oddělení klient – v případě, kdy klient vykonává nad nějakou službou operace, které vedou na transakční zpracování, je zde požadavek, aby klient nevěděl nic o probíhajíci transakci. Tedy, transakční zpracování je automaticky řízeno bez povědomí klienta.
- Rozšiřitelnost – do daného řešení lze snadno bez větších úprav zahrnout další službu, která se může účastnit transakčního zpracování.

1.2.2.1 Scénář využití transakce

Možným scénářem pro využití transakčního zpracování je registrace nového uživatele. V tomto scénáři dochází ke komunikaci mezi `CRM` systémem [22], který je dostupný například na adrese `https://example.com/crm/` a účetním systémem, který je dostupný například na adrese `https://example.com/ais/`. V tomto scénáři mají operace vně transakce následující význam:

- operace `rollback` – dojde k odvolání všech záznamů spojených se založením nového uživatele v systému,
- operace `commit` – dojde k potvrzení všech záznamů spojených se založením nového uživatele, ten se od tohoto okamžiku stane v systému viditelným,
- provedení operace nad službou v transakci – dojde k provedení určité operace, např. vytvoření nového účtu. Úspěšný výsledek této operace není však v systému viditelný do chvíle dokud nedojde k potvrzení transakce.

Tento scénář je zcela uměle vytvořen pro demonstraci použití jednotlivých řešení uvedených dále, jeho průběh je uveden u jednolivých řešení a v nich se může jeho skladba mírně lišit.

V následujících případech se `CRM` systém během dvou kroků snaží založit nového uživatele v účetním systému.

Případ úspěšně provedené akce (úspěšná transakce)

1. `CRM` systém zašle požadavek na účetní systém pro vytvoření nového uživatele. Součástí tohoto požadavku jsou základní kontaktní údaje nového uživatele.

2. `CRM` systém zašle aktualizaci pracovního umístění nově vytvořeného uživatele do účetního systému.

3. `CRM` systém potvrdí nově vytvořeného uživatele.

Případ neúspěchu (odvolané transakce)

1. `CRM` systém zašle požadavek na účetní systém pro vytvoření nového uživatele. Součástí tohoto požadavku jsou základní kontaktní údaje nového uživatele.

2. `CRM` systém zašle aktualizaci pracovního umístění nově vytvořeného uživatele do účetního systému.

3. `CRM` dojde k vypršení `timeout`, nebo je signalizována chyba od účetního systému.

4. `CRM` systém provede `rollback` akce vytvoření nového uživatele.

Zde, bez využití zpracování v transakci by zůstal nevalidní záznam v účetním systému.

1.3 Alternativní řešení

Vedle zavedení transakčního zpracování do rozhraní `REST` existují i jiné způsoby řešení systémové integrace umožňujíci transakční zpracování. V zásadě je zde využito jiných technologií než architektonického rozhraní `REST`. Tato kapitola se krátce věnuje pár vybraným možnostem pro demonstrování alternativních přístupů k problému transakcí a systémové integrace.

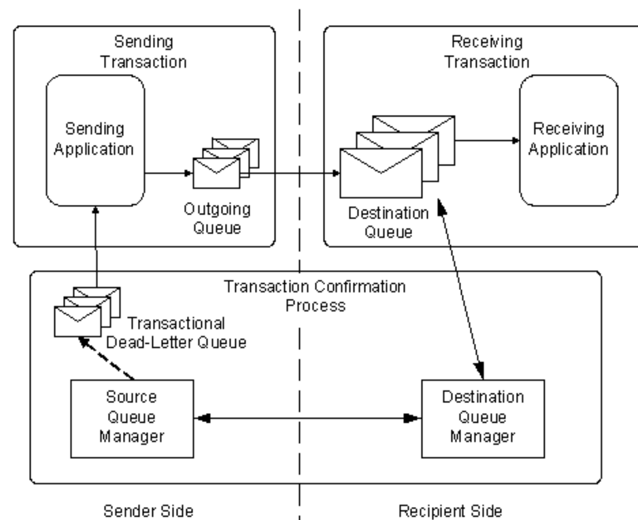
1.3.1 Message Queue

Message Queue je softwarová komponenta používaná pro meziprocesní komunikaci, nebo komunikaci v rámci jednoho procesu mezi jednotlivými vlákny. V této technologii je využito fronty pro poskytnutí asynchronního komunikačního protokolu s konzistentním aplikačním rozhraním.

Komunikace mezi dvěmi komponentami zde probíhá pomocí zasílání zpráv, které jsou řazeny do uložiště fronty. To umožňuje jednotlivým komponentám nezávisle na sobě řadit a vybírat tyto zprávy při různých rychlostech, v odlišném čase apod. Zprávy vkládané do fronty jsou uloženy do doby než dojde k jejich vyzvednutí příjemcem.

Toto paradigma má mnoho společné se vzorem `publisher/subscriber` a je často jednou z částí middleware systému orientovaného na zprávy. [1]

Pro zajištění transakčního zpracování zde existují transakčně orientované fronty. Způsob jedné takové implementace od společnosti Microsoft je zachycen na následujícím schématu:



Obrázek 1.3: Transakční Message Queue [1]

V tomto modelu zasílající komponenta může potvrdit zaslanou transakci, která zasílá zprávu do cílové fronty a obdobně komponenta, pro kterou je zpráva určena, smí potvrdit přijímací transakci, která vyzvedne zprávu z cílové fronty. Pokud dojde k potvrzení celé komunikace v transakci, zpráva je umístěna do výstupní fronty na zdrojové komponentě a dochází k jejímu doručení do cílové fronty.

V případě, kdy dochází k zasílání více zpráv v rámci jedné transakce je využito interního `exactly-once-delivery` protokolu[4]. Více naleznete v [23].

[4]Každá zpráva je spojena s jedinečným identifikátorem, který slouží k filtrování duplicitnch zpráv během jejich přenosu od odesílatele k příjemci.

1.3.2 EJB

Enterprise JavaBeans (`EJB`) je jedno z Java aplikačních rozhraní pro modulární konstrukce enterprise softwaru. Jedná se o komponentu na straně serveru pro zapouzdření aplikační podnikové logiky. Webový EJB kontejner poskytuje běhové prostředí (`Runtime Environment` [24]) pro webově založené softwérové komponenty. [25]

EJB kontejnery jsou transakčními servery, které zpracovávají šíření transakčního kontextu a distribuované transakce. Transakce zde může být řízena pomocí kontejneru nebo pomocí vlastního zpracování v zdrojovém kódu (Java Beans a anotace).

V případě řízení transakce pomocí kontejneru jsou její stavy říženy automatiky za pomocí kontejneru. V případě použítí zdrojového kódu je životní cyklus transakce řízen vývojářem.

Kontejnerem řízené transakce `EJB` 3.0 specifikuje následující transakční atributy, které jsou implementovány v kontejneru.

- **`REQUIRED`** – indikuje, že metoda má být spuštěna uvnitř transakce.
- **`REQUIRES_NEW`** – indikuje, že při spuštění metody má být vytvořena nová transakce.
- **`SUPPORTS`** – indikuje, že metoda by měla být součástí transakce.
- **`NOT_SUPPORTED`** – indikuje, že metoda by neměla být součástí transakce.
- **`MANDATORY`** – indikuje, že metoda musí být spuštěna uvnitř transakce, jinak je vyhozena výjimka.
- **`NEVER`** – indikuje, že pokud je metoda spuštěna uvnitř transakce, je vyhozena výjimka.

V tomto případě je myšlena jakákoliv podniková operace, která může být součástí transakčního zpracování.

Transakce spravované v kódu Transakce spravované ve zdrojovém kódu (`Bean Managed Transactions`) mohou být řízeny výjimkami na aplikační úrovni. Hlavními body o kterých je nutné rozhodnout v tomto způsubu jsou:

- **`Start`** – kdy začít transakci.
- **`Success`** – identifikace úspěchu scénáře, kdy transakce má být potvrzena.
- **`Failed`** – indetifikace chybového scénáře, kdy transakce má být vrácena. [26]

Jednou z nejvýraznějších nevýhod EJB je nutnost použítí technologie Java EE a tedy větší zásah nebo nutnost modifikaces stávajících komponent. Více o EJB se dozvíte například v [25], nebo [26].

1.3.3 Ostatní možnosti

Vedle výše zmíněných existuje mnoho jiných možností řešení. Například místo rozhraní REST použít konkurenční protokol SOAP, který defaultně podporuje atomické transakce (WS-AtomicTransaction) zaručující vlastnosti ACID.

Další možností je využít jednoho z možných návrhů pro moderní SOA systémy, které se snaží vyhnout nutnosti použítí transakcí tak, jak jsou zde definovány. Jedním, z těchto návrhů, je například popsán v tomto dokumentu [27].

Kapitola 2

Nalezená řešení

V Této části jsou rozebrány tři nejčastěji zmiňovaná řešení zavedení transakčního zpracování v `REST` (respektive v `RESTful`) architektonickém stylu. Všechny možnosti jsou principiálně odlišné, ale vyskytují se zde některé společné rysy. První možnost je uvedena ve dvou variantách, ve kterých lze její použití zvažovat.

2.1 Transakce jako zdroj

Prvním řešením je realizace transakce reprezentované formou samostatné `REST` služby, kde transakce vystupuje v roli samostatného zdroje (transakční služba zde reprezentuje funkci koordinátora). Toto řešení vychází z [20]. Jedná se o jednoduchou variantu, kde probíhá komunikace mezi klientem (koncová stanice, server, ...), vytvářejícím transakci a zasílajícím požadavky, a jednou či více službami poskytující požadovanou funkcionalitu, převážně budeme uvažovat *high availability* systémy [28], účastnící se transakčního zpracování.

Uvažujme dostupnost zdroje transakce například na adrese:

`https://example.com/transactions/`.

Komunikace zde může probíhat pomocí standardních internetových protokolů HTTP (v1.0 [29] nebo v1.1 [30]) a HTTPS [31]. V této práci pro prezentované příklady uvažujeme protokol HTTPS. Data předávaná v probíhající komunikaci jsou zde reprezentována pomocí datového formátu `JSON`[5] [32].

U tohoto řešení, použítí transakce jako zdroje, lze uvažovat více verzí, které se liší, jak se s tímto zdrojem dále pracuje. Zde jsou uvedeny dvě možnosti, které mají některé charakteristiky společné.

[5] JavaScript Object Notation – způsob zápisu dat nezávislý na počítačové platformě, určený pro přenos dat, která mohou být organizována v polích nebo agregována v objektech.

Těmito charakteristikami jsou:

Návratové hodnoty
Prvním společným rysem jsou návratové hodnoty odpovídajících `HTTP` odpovědím. Každá operace může skončit buďto úspěchem nebo neúspěchem. V takovýchto situacích server vrací:

- v případě úspěchu `HTTP` odpověď `2xx`, například `HTTP/1.1 200 OK`, a případná data,
- v případě neúspěchu některou z `4xx` HTTP odpovědí s popisem chyby.

Stavy transakce
Dalším společným rysem jsou stavy, ve kterých se transakce může vyskytovat. Tyto stavy jsou:

- `pending` - probíhající transakce, transakce čekající na dokončení,
- `commited` - potvrzená transakce,
- `rollbacked` - vrácená transakce.

Následuje popis jednotlivých verzí řešení pomocí transakce reprezentované jako samostatný zdroj.

2.1.1 Transakce jako zdroj - První verze

2.1.1.1 Popis

Vytvoření transakce
Vytvoření nové transakce provedeme zasláním `POST` požadavku na adresu `https://example.com/transactions/`. Součástí požadavku na vytvoření nové transakce může být nastavení doby platnosti, po kterou může být transakce platná (`timeout` transakce). Po vypršení této doby platnosti bez předchozího vykonání operace `rollback`, nebo `commit` je transakce zrušena a je vyvolána operace `rollback`.

Tento požadavek vytvoří nový zdroj transakce s vlastním identifikačním číslem (`id`), které v případě úspěchu vrací v odpovědi.

Podoba takového požadavku bude například:

Hlavička:

```
1 POST /transactions/ {protocol}
2 Host: example.com
3 Content-Type: application/json
```

Obsah:

```
{
  "timeout": "{TIMEOUT}"
}
```

Úpravy transakce

Úpravy transakce, vykonání operace v transakci pomocí `PUT` požadavku na adresu `https://example.com/transactions/{id}/update`. V datové části požadavku jsou obsaženy všechny potřebné informace:

- `HTTP` metoda, která bude vykonána, ve tvaru `method = GET/POST/...`
- `URL` adresa operace tvaru `url=URL_OPERACE`,
- příznak stavu transakce `state=pending`, nebo `state=commited`,
- volitelně datová část obsahující obsah požadavku ve tvaru `content=DATA`,
- volitelně délka platnosti (`timeout`) do které musí být operace vykonána, jinak dojde k vyvolání operace `rollback` nad transakcí.

Veškeré informace o provedené změně jsou uloženy v objektu transakce. V odpovědi od serveru jsou obsaženy všechny informace o úspěchu, nebo neúspěchu provedení požadované operace. Dále odpověď obsahuje data, která by byla obdržena v případě originálního požadavku. Podoba takového požadavku bude například:

Hlavička:

```
PUT /transactions/{id}/update {protocol}
Host: example.com
Content-Type: application/json
```

Obsah:

```
{
  "update" : {
    "method": "{OPTIONS/GET/HEAD/POST/PUT/DELETE/
   TRACE/CONNECT}",
    "url": "{url_operace}",
    "state": "{pending/commited}",
    "content": "{datta}",
    "timeout": "{timeout}"
  }
}
```

Nastavením `state=commited` vyvolá po vykonání konkrétní operace její okamžité potvrzení.

Potvrzení transakce

Potvrzení transakce (operace `commit`) je proveden pomocí `POST` požadavku na adresu `https://example.com/transactions/{id}/commit`.

Podoba takového požadavku bude například vypadat:

Hlavička:

```
1 POST /transactions/{id}/commit {protocol}
2 Host: example.com
```

To vyvolá potvrzení nad všemi změnami uloženými ve zdroji transakce.

Vrácení transakce

Vrácení transakce (operace `rollback`) je provedeno pomocí `POST` požadavku na adresu `https://example.com/transactions/{id}/rollback`. Nebo za pomoci `DELETE` požadavku na adresu zdroje transakce, tedy požadavku na adresu `https://example.com/transactions/{id}`. Podoba takového požadavku bude například:

Hlavička:

```
1 POST /transactions/{id}/rollback {protocol}
2 Host: example.com
```

nebo:

```
1 DELETE /transactions/{id} {protocol}
2 Host: example.com
```

To vyvolá vrácení všech změn provedených nad touto transakcí.

Získání informací o transaci

Informace o některé z transakcí jsou získány pomocí standardního `GET` požadavku a adresy konkrétní transakce. Podoba takového požadavku bude například:

Hlavička:

```
1 GET /transactions/{id} {protocol}
2 Host: example.com
```

Tento požadavek vrátí všechny potřebné informace o transakci, jako jsou:

- `id`,
- strany zůčastněné v transakci,
- seznam provedených operací v tranakci,
- nastavený `timeout` a případný zbývající čas v milisekundách.

2.1.1.2 Příklad použití

Příklad, který představuje úspěšný scénář z 1.2.2.1. Ve všech uvedených řešení je pro demostraci příkladu použití využit protokol `HTTP 1.1`.

1. **Vytvoření nové transake:**

   Hlavička požadavku:

```
POST /transactions HTTP/1.1
Host: example.com
```

   Přijatá odpověď:

   Hlavička odpovědi:

```
HTTP/1.1 201 Created
Content-Type: application/json
```

   Obsah odpovědi:

```
{
  "transaction": {
    "state": "success",
    "id": 1
  }
}
```

2. **Založení nového uživatele**

   Hlavička požadavku:

```
PUT /transactions/1/update HTTP/1.1
Host: example.com
Content-Type: application/json
```

   Obsah požadavku:

```
{
  "update": {
    "method": "POST",
    "url": "https://example.com/ais/user",
    "state": "pending",
    "content": {
      "firstname": "Jakub",
      "lastname": "Drabek",
      "birthday": "23.02.1992",
      "mail": "drabeja1@fit.cvut.cz"
    }
```

```
12    }
13 }
```

Přijatá odpověď:

Hlavička odpovědi:

```
1 HTTP/1.1 201 Created
2 Location: https://example.com/ais/user/999
3 Content-Type: application/json
```

Obsah odpovědi:

```
1 {
2   "user-id": 999
3 }
```

3. **Aktualizace informací o uživateli**

Hlavička požadavku:

```
1 PUT /transactions/1/update HTTP/1.1
2 Host: example.com
3 Content-Type: application/json
```

Obsah požadavku:

```
1 {
2   "update": {
3     "method": "PUT",
4     "url": "https://example.com/ais/user/999/
     update",
5     "state": "pending",
6     "content": {
7       "department": "IT",
8       "position": "development"
9     }
10   }
11 }
```

Přijatá odpověď:

Hlavička odpovědi:

```
1 HTTP/1.1 200 OK
```

4. **Potvrzení založení nového uživatele**

Hlavička požadavku:

```
1 POST /transactions/1/commit HTTP/1.1
2 Host: example.com
```

Přijatá odpověď:

Hlavička odpovědi:

```
1 HTTP/1.1 200 OK
```

2.1.1.3 Zhodnocení

Výhody

- Služby zúčastněné v transakčním zpracování jsou izolovány od povědomí o probíhající transakci. Je tedy splněn požadavek na bezstavovost zúčastněných služeb.

Nevýhody

- Zvýšení komunikační zátěže. Komunikace probíhá pomocí služby transakce, to vede k přibližne dvojnásobnému nárůstu zaslaných zpráv.
- Transakční služba musí vědět o všech službách, které se mohou účastnit transakce a musí podporovat mechanismus pro odvolání provedených změn, ten se může dle jednotlivých služeb lišit nebo nemusí být splnitelný.
- Každá služba, které je umožněno účastnit se transakčniho zpracování, přináší další komplexnost a nutnost rozšířit službu transakcí.
- Není zde vyřešen požadavek na zajištění izolace provedených změn uvnitř transakce.
- Začátek transakce, konec transakce a vzniklé chyby musí být řešeny v rámci klienta, nebo musí být přidán další mechanismus pro odstínění klienta od povědomí o transakci.
- Špatně rozšířitelné o další zúčastněné služby. Nedefinuje jakým způsobem zahrnout nové služby.

2.1.2 Transakce jako zdroj - Druhá verze

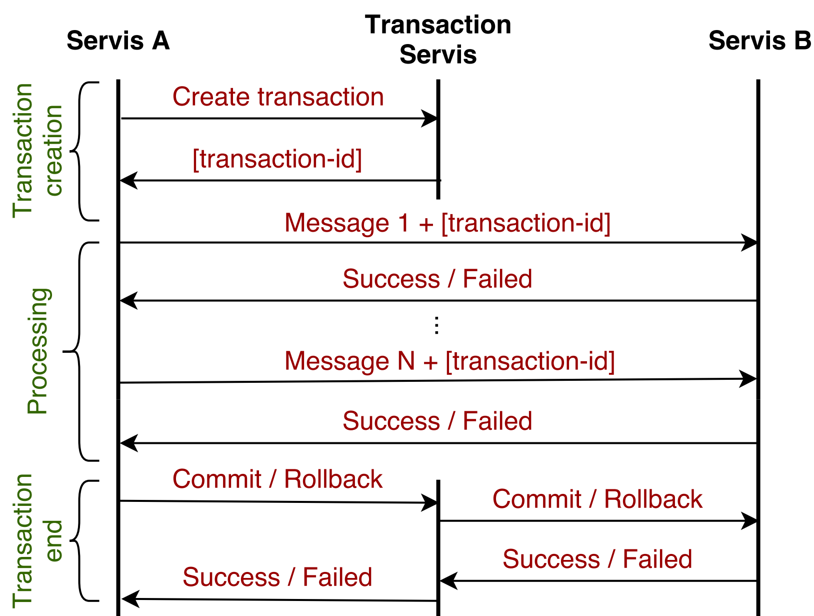
Druhá verze řešení, transakce reprezentovaná pomocí nového zdroje, je modifikací předchozí verzí.

2.1.2.1 Popis

Na rozdíl od pvní verze je v Této variantě jinak zacházeno s operacemi nad transakcí. Každá operace (vytvoření nového zdroje, vyvolání změny, apod.) v transakci, probíhá mimo zdroj transakce. Avšak nese u sebe `id` transakce a stav, ve kterém se transakce nachází. Všechny potřebné informace pro realizaci transakčnosti jsou uvedeny v hlavičkách požadavku, např.:

- `x-transaction-id` - identifikačná číslo transakce,
- `x-transaction-state` - stav transakce,
- `x-transaction-timeout` - stanovený čas pro provedení operace.

Komunikace mezi dvěmi službami bude vypadat:



Obrázek 2.1: Schéma komunikace - transakce jako zdroj

Vytvoření transakce
Obdobně jako v první verzi, vytvoření nové transakce provedeme zasláním `POST` požadavku na adresu transakční služby `https://example.com/transactions/`. Tento požadavek vytvoří nový zdroj transakce s vlastním identifikačním číslem `id`, které v případě úspěchu vrátí v odpovědi. Toto `id` dále používáme

pro prezentaci transakčního zpracování v jednotlivých dotazech na účastníky procesu.

Úpravy transakce
Provedení operace v rámci transakce je zde řešeno tak, že se provede standardní požadavek, jehož hlavička je rozšířena právě o identifikační číslo transakce, její stav a případný `timeout`. Podoba takového příkazu je například:

Hlavička požadavku:

```
1 PUT /account/{account}/add/{amount} HTTP/1.1
2 Host: example.com
3 Content-Type: application/json
4 x-transaction-id: {transaction_id}
5 x-transaction-state: {transtaction_state}
6 x-transaction-timeout: {timeout}
```

Obsah požadavku:

```
1 {
2   {content of request}
3 }
```

Potvrzení transakce
Operace `commit` je opět provedena pomocí `PUT` požadavku na adresu `https://example.com/transactions/{id}/commit`. Tato operace změní stav zdroje transakce na `commited` a dále musí spustit průchod všechn zdrojů s nastaveným příslušným `id` transakce a také i u nich změnit stav na `commited`.

Vrácení transakce
Operace `rollback` je také opět provedena pomocí `PUT` požadavku na adresu `https://example.com/transactions/{id}/rollback`. Tato operace, obdobně jako v případě operace `commit`, musí projít všechny zdroje s nastaveným příslušným `id` transakce a odvolat všechny změny, nebo tyto zdroje smazat.

Obdobného výsledku je dosaženo i pomocí `DELETE` požadavku na adresu konkrétního zdroje transakce.

2.1.2.2 Příklad použití

Transformovaný příklad z verze 1.

1. **Vytvoření nové transakce:**

   Hlavička požadavku:

```
1 POST /transactions HTTP/1.1
2 Host: example.com
```

Přijatá odpověď:

Hlavička odpovědi:

```
HTTP/1.1 201 Created
Content-Type: application/json
```

Obsah odpovědi:

```
{
  "transaction": {
    "state": "succes",
    "id": 1
  }
}
```

2. **Založení nového uživatele:**

Hlavička požadavku:

```
PUT /transactions/1/update HTTP/1.1
Host: example.com
Content-Type: application/json
x-transaction-id: 1
x-transaction-state: pending
```

Obsah požadavku zůstává nezměněn. Následná přijatá odpověď:

Hlavička odpovědi:

```
HTTP/1.1 201 Created
Location: https://example.com/ais/user/999
Content-Type: application/json
```

Obsah odpovědi:

```
{
  "user-id": 999
}
```

3. **Aktualizace informací o uživateli**

Hlavička požadavku:

```
PUT /transactions/1/update HTTP/1.1
Host: example.com
Content-Type: application/json
x-transaction-id: 1
x-transaction-state: pending
```

Obsah požadavku:

```
{
  "update": {
    "method": "PUT",
    "url": "https://example.com/ais/user/999/update",
    "state": "pending",
    "content": {
      "department": "IT",
      "position": "development"
    }
  }
}
```

Přijatá odpověď:

Hlavička odpovědi:

```
HTTP/1.1 200 OK
```

4. **Potvrzení založení nového uživatele**

Hlavička požadavku:

```
POST /transactions/123/commit HTTP/1.1
Host: example.com
```

Přijatá odpověď:

Hlavička odpovědi:

```
HTTP/1.1 200 OK
```

2.1.2.3 Zhodnocení

Výhody

- Nižší komunikační zátěž oproti první verzi. Potřebných zpráv navíc je vyjádřeno vzorcem $3 + x$ kde x je počet zúčastněných služeb.
- Zodpovědnost za odvolání či potvrzení změn přenesena z transakční služby na zúčastněné služby – do transakčního zpracování lze zahrnout více služeb bez nutnosti úpravy transakční služby.
- Snadno rozšiřitelné o zúčastněné služby, které umí nakládat s informací o probíhající transakci.

Nevýhody

- Nutnost upravit všechny služby, které se mohou účastnit transakčního zpracování tak, aby byly schopny přijímat informace o probíhajícím transakčním zpracování – příjímat informace o transakci, podpora operací `rollback` a `commit` apod.
- Není zde vyřešen mechanismus pro zajištění izolace provedených operací v rámci transakce – nutnost opět ošetřit na úrovni účastnících se služeb.
- Porušuje princip bezstavovosti.
- Nárust potřebné komunikace.
- Klient opět není odstíněn od povědomí o transakci.

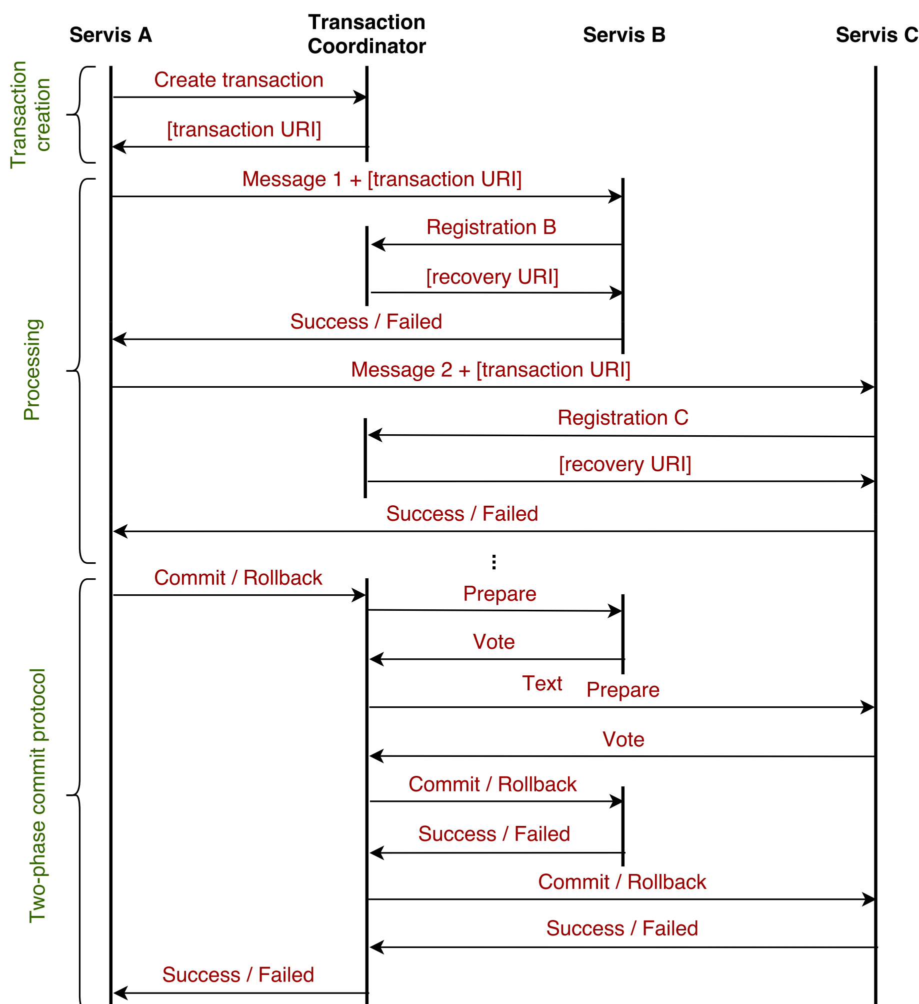
2.2 Návrh koordinátora

2.2.1 Popis řešení

Druhé řešením pro realizaci zavedení transakčního zpracování v `REST` je použití „klasického“ transakčního koordinátora. Jedná se o specializovaný software (např. na úrovni `middleware`, ...) pomocí nějž jsou vytvářeny a spravovány jednotlivé transakce. Transakční koordinátor může být součástí skupiny ostatních koordinátorů poskytujících další přidanou funkcionalitu, a kteří společně reprezentují sérii nějakých aktivit (např. umístěných v integrační vrstvě).

Toto řešení popisuje protokol pro použití transakčního koordinátora využívajícího `HTTP` protokol a vychází z [33].

Pro počáteční ilustraci je zachycena komunikace za použití tohoto protokolu na následujícím diagramu:



Obrázek 2.2: Schéma komunikace s použitím transakčního koordinátora

Zde účastníci, kteří vystupují uvnitř transakce, využívají služeb transakčního koordinátora pro zajištění transakčního zpracování. Ten je reprezentován svou `URI` [34]. Pro příklad uvažujme adresu:

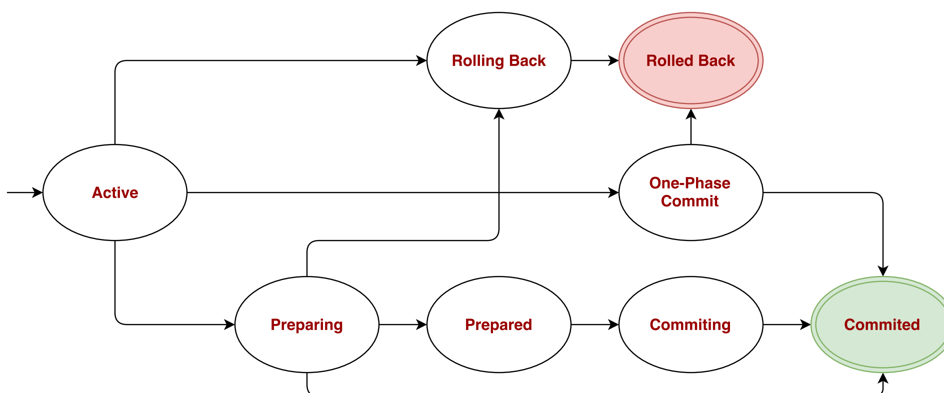
`https://example.com/transaction-coordinator/`.

V tomto řešení je využito upraveného `two-phase commit` protokolu, drobné modifikace jsou popsány jednotlivými operacemi uvedenými níže.

`Two-phase commit` protokol Transakční koordinátor využívá `2PC` protokol s následujícími optimalizacemi:

- **Operace `rollback`** – transakční koordinátor nemusí zaznamenat informaci o účastníku transakce do trvalého úložiště, dokud není proveden `commit`, neboli, dokud po přípravné fázi není dosaženo úspěšného dokončení. Odpověď o neexistující transakci může být použita k odvození, že byla provedena operace `rollback`.
- **První fáze protokolu** – v případě, že je registrován pouze jeden účastník, je vynechána přípravná (`prepare`) fáze.
- **Pouze ke čtení** – účastník představující službu, která žádným způsobem nemění transakční data, může být v přípravné fázi označen jako `read-only` a koordinátor jej dále vynechá z druhé fáze protokolu.

Jednotlivé transakční stavy a přechody mezi nimi, za použití `2PC` protokolu, jsou zobrazeny na následujícím schématu:



Obrázek 2.3: Stavový diagram transakce s `2PC` protokolem

Účastníkům, kteří úspěšně prošli přípravnou fází, je umožněno činit samostatná rozhodnutí kdy a za jakých podmínek provést `commit` nebo `rollback`. Pokud dojde k provedení takové volby, a tím dojde k ovlivnění vlastní transakce, musí dojít k jejímu zaznamenání. V případě, že koordinátor informuje účastníky o osudu transakce (jejím potvrzení nebo vrácení) a ten je stejný jako autonomní rozhodnutí jednotlivých účastníků, nevzniká problém a každý

z účastníků provede příslušnou operaci před samotným koordinátorem. Pokud je ale rozhodnutí v rozporu s některým z autonomních rozhodnutí, vzniká neatomický výsledek – heuristický výsledek s odpovídajícím heuristickým rozhodnutím.

Těmito heuristickými výsledky jsou:

- **Heuristický rollback** - `TransactionHeuristicRollback` značí, že potvrzení operace selhalo z důvodů jednostranného odvolání transakce z jednoho nebo z více účastníků transakčního zpracování.
- **Heuristický commit** - `TransactionHeuristicCommit` značí, že pokus o `rollback` se nezdařil z důvodu potvrzení transakce všech účastníků. Toto může nastat například v případě, kdy koordinátor byl schopen úspěšně připravit transakci, ale došel k rozhodnutí o jejím vrácení (z důvodu vnitřní chyby apod.), ale mezitím došlo k potvrzení od všech účastníků.
- **Smíšený výsledek** - `TransactionHeuristicMixed` značí, že některé aktualizace byly přijaty, jiné zase vráceny.
- **Heuristický hazard** - `TransactionHeuristicHazard` značí že, není znám výsledek jedné či více aktualizací. U zbylých musí dojít buďto k potvrzení nebo k vrácení.

Seznam všech stavů s popisem použitých v tomto řešení je uveden v sekci 2.2.1.5.

2.2.1.1 Vytvoření transakce

Vytvoření nové transakce je provedeno pomocí `POST` požadavku na adresu transakčního koordinátora. V případě, že chceme nastavit `timeout` uvedeme ho jako parametr dotazu, jinak je použita výchozí hodnota.

Při úspěšném vytvoření transakce jsou v hlavičce odpovědi vráceny:

- návratový kód `201 Created`,
- `URI` nově vytvořené transakce (reprezentováno jejím unikátním `id`),
- `URI` správce zodpovědného za transakci (tzv. terminátor), který je schopen s danou transakcí zacházet – typicky vystupující jako nový klient,
- `URI` pro registraci účastníků v transakci.

Podoba dotazu s nastaveným `timeout`:

Hlavička:

```
POST /transaction-coordinator {protocol}
Host: example.com
Content-Type: application/json
```

Obsah:

```
{
    "timeout": "{timeout}"
}
```

Podoba hlavičky odpovědi:

Hlavička:

```
{protocol} 201 Created
Location: /transaction-coordinator/{id}
Link: /transaction-coordinator/{id}/terminator;
rel="terminator"
Link: /transaction-coordinator/{id}/participants;
rel="participants"
```

Obdobnou odpověď dostaneme i v případě zasláním `HEAD` požadavku na adresu transakce `/transaction-coordinator/{id}`. Rozdílem zde bude pouze návratový kód, který bude `200 Ok`. Ale v případě překročení `timeout` dojde k zneplatnění transakce. Všechny následné pokusy dotazu na transakci a příslušná `URI` vrátí kód `410 Gone`.

Pokud není doručena klientu odpověď od transakčního koordinátora (např. z důvodu výpadku sítě), je možnost zasílat další dotazy na vytvoření nové transakce, dokud není dosaženo úspěchu. V tomto případě by předešlé vytvořené transakce měly být nevyužity a jsou zrušeny po překročení `timeout`. V případě fatální chyby transakčního koordinátora je vrácen kód `500 Internal Server Error` s popisem chyby.

2.2.1.2 Informace o transakci

Seznam všech transakcí spravovaných koordinátorem získáme zasláním `GET` požadavku na adresu transakčního koordinátora. Obdobně získáme informace o konkrétní transakci. Tedy zašleme `GET` požadavek na adresu konkrétní transakce. To nám vrátí stav, ve kterém se transakce aktuálně nachází.

Podoba hlavičky požadavku:

Hlavička požadavku:

```
GET /transaction-coordinator/{id} {protocol}
Host: example.com
Accept: application/json
```

Příslušná odpověď:

Hlavička odpovědi:

```
{protocol} 200 Ok
Content-Type: application/json
```

Obsah odpovědi:

```
{
    "tx-status": "TransactionActive"
}
```

2.2.1.3 Zrušení transakce

Zrušení transakce, provedení operací `commit`, nebo `rollback`. To provedeme pomocí `PUT` požadavku na `terminator` příslušné transakce s nastaveným příslušným stavem (`TransactionCommit`, nebo `TransactionRollback`). Tedy:

Hlavička:

```
POST /transaction-coordinator/{id}/terminator {protocol}
Host: example.com
Content-Type: application/json
```

Obsah:

```
{
   "tx-status": "{TransactionCommit/
   TransactionRollback}"
}
```

Pro zrušení transakce je nutné, aby transakce byla v aktivním stavu, jinak není možné požadovanou operaci vykonat a je vrácen kód `412 Precondition Failed`.

V případě úspěchu je vrácen kód `200 Ok`, nebo `202 Accepted`. Pokud se ale jedná o dupicitní požadavek o zrušení a tranksakce již byla zrušena, je vrácen kód `410 Gone` a aktuální stav transakce `TransactionCommited`, nebo `TransactionRolledback`. V ostatních případech je vrácen kód `400 Bad Request`.

Potvrzení transakce (operace `commit`) je vykonáno až po vykonání všech požadovaných změn provedených uvnitř transakce – až když každý z účastníků je hotov se svou prací. V případě, že všechny změny stále nebyly vykonány, potvrzení transakce končí neúspěchem.

2.2.1.4 Propagace transakčního obsahu

V případě, kdy práce se zdrojem má být součástí transakčního zpracování je nutné, aby URI konkrétní transakce byla předána tomuto zdroji. V tuto chvíli se může zdroj registrovat jako účastník transakce za použití své unikání URI pro své jednoznačné identifikování. Uvažujme, že účasník je například rezprezentován pomocí URI /participant-resource.

Pro propagaci kontextu jsou zde uvažovány dva možné způsoby:

- URI transakce je vloženo jako hlavička Link v požadavku pro zúčastněnou službu.
- Služba správy účastníků transakce vrátí odkaz na klienta, který může být použit k registraci účastníků do transakčního procesu.

V Této práci dále budeme uvažovat první variantu, kde během komunikace mezi dvěmi službami bude předáván odkaz na transakci. Zde v případě, že je přijat dotaz s nastavenou transakcí a služba není zatím registrována, musí dojít k její registraci. V případě, že není možné, aby se služba registrovala do transakce, je navrácen chybový kód 406 Not Acceptable.

2.2.1.5 Operace s účastníky

Registrace účastníků druhé fáze protokolu

Jednotlivý účastníci, kteří se účastní druhé fáze 2PC prokolu, jsou registrování pomocí POST požadavku na URI obdržené při vytváření nové transakce. Podoba takového požadavku:

Hlavička požadavku:

```
1 POST /transaction-coordinator/{id}/participant {protocol}
2 Host: example.com
3 Content-Type: application/json
```

Obsah požadavku:

```
1 {
2     "participant": "/participant-resource",
3     "terminator": "/participant-resource/terminator"
4 }
```

To v případě úspěchu vrátí kód 201 Created a URI určenou pro obnovu původního stavu zdroje (stav před započetím transakce). Zde uvažujme URI pro obnovu dostupnou na /transaction-recovery. Podoba takové odpovědi tedy bude:

Hlavička odpovědi:

```
1 {protocol} 201 Created
2 Location: /participant-recovery/{id}
```

Provedením `HEAD` požadavku na `URI` registrovaného účastníka získáme opět odkaz na `terminator`:

Hlavička požadavku:

```
1 HEAD /participant-resource {protocol}
2 Host: example.com
```

Hlavička odpovědi:

```
1 {protocol} 200 Ok
2 Link: /participant-resource/terminator;
3 rel="terminator"
```

Toho dosáhneme ale pouze v případě, kdy je transakce ve aktivním stavu (`TransactionActive`), jinak je vrácen kód `412 Precondition Failed`.

Registrace účastníků neúčastnících se druhé fáze protokolu

Aby se účastník mohl podílet na transakčním zpracování, musí být zajištěn požadavek na konzistenci dat v případě vzniku chyby, nebo musí být garantován souběžný přístup k danému zdroji.

Tyto požadavky však nevedou k potřebě úpravy služby tak, aby každý zdroj měl zřízen `terminator`. Je zde pouze potřeba mít mechanismus, jak informovat transakčního koordinátora, které služby mají s kým komunikovat při realizaci druhé fáze `2PC` protokolu.

V tomto případě, během registrace nové služby do transakce, musí tato služba poskytnout `URI` pro zajištění operací přípravy (`prepare`), potvrzení (`commit`), odvolání (`rollback`) a volitelně potvrzení první fáze protokolu. Ty musí být předány v požadavku na registraci účastníka do transakce. Podoba takového požadavku bude poté vypadat:

Hlavička:

```
1 POST /transaction-coordinator/{id}/participant {protocol}
2 Host: example.com
3 Content-Type: application/json
```

Obsah:

```
1 {
2     "participant": "/participant-resource",
3     "prepare": "/participant-resource/prepare",
4     "commit": "/participant-resource/commit",
5     "rollback": "/participant-resource/rollback"
6 }
```

Předané informace získáme zpět opět provedením `HEAD` požadavku na registrovaného účastníka. Podoba odpovědi:

Hlavička:

```
{protocol} 200 OK
Link: /participant-resource/prepare; rel="prepare"
Link: /participant-resource/commit; rel="commit"
Link: /participant-resource/rollback; rel="rollback"
```

Služba, která registruje jednotlivé účastníky musí tedy definovat vztah účastník a `terminator` nebo vztahy/zdroje potřebné pro realizaci `2PC` protokolu.

Získání stavu účastníka

Stav každého účastníka transakce získáme pomocí `GET` požadavku na jeho `URI`. To vrátí stav účastníka obdobně jako dotaz na stav transakce u transakčního koordinátora, viz. 2.2.1.2.

Vyjmutí účastníka

Transakční koordinátor řídí jednotlivé účastníky pomocí `2PC` protokolu. Ten je realizován pomocí `PUT` požadavku na `terminator` konkrétního účastníka (`URI` poskytnuté během registrace) s konkrétním nastavením stavu (operace) v obsahu dotazu (žádost o změnu stavu). Požadavek může obsahovat následující stavy:

- `TransactionPrepare` – fáze přípravy transakce,
- `TransactionCommit` – potvrzení transakce (`commit`),
- `TransactionRollback` – odvolání transakce (`rollback`),
- `TransactionCommitOnePhase` – potvrzení účastníka, který je součástí jedno-fázového potvrzení.

Podoba požadavku na přípravnou fázi:

Hlavička:

```
PUT /transaction-resource {protocol}
Host: example.com
Content-Type: application/json
```

Obsah:

```
{
    "tx-status": "TransactionPrepare"
}
```

V případě úspěchu je vrácen kód `200 OK`. Jinak jsou vráceny chybové kódy. Ty mohou být vráceny v následujících situacích:

- Kód `409 Conflict` v případě neúspěchu s přiloženým důvodem chyby. Zde, pokud je účastník součástí `2PC` protokolu, musí následně dojít k operaci `rollback` nad transakcí.
- Kód `412 Precondition Failed` v případě, kdy účastník není ve stavu, ze kterého lze provést požadovanou operaci.
- Kód `410 Gone` po již vykonaných operacích `TransactionRollback`, `TransactionCommit` nebo `TransactionCommitOnePhase` a následném `POST` požadavku.

V případě výskytu chyby při operaci `TransactionPrepare` je transakce odvolána.

Po zaslání `PUT` dotazu na `terminator` účastníka s `Forget` parametrem je vyvoláno odstranění jakýkoliv heuristických rozhodnutí učiněných během transakce. V případě úspěchu je vrácen kód `200 OK` a dojde k odebrání účastníka. Následný `PUT` nebo `GET` požadavek způsobí vrácení chybového kódu `410 Gone`.

Obnova účastníka

Obnova zde představuje operaci, kdy koordinátor nebo účastník bude obnoven do stejného stavu jako před vznikem chyby, která způsobila vyvolání obnovy. Pokud tato operace není podporována, koncový bod musí indikovat, že byl přesunut jinam. Například za pomoci navrátového kódu `301 Moved Permanently` nebo `307 Temporary Redirect` v případě dočasného přesunutí.

V případě, že dojde k selhání účastníka, který následně musí být přesunut na jiné `URI`, není transakční koordinátor schopen dokončit transakci. Z tohoto důvodu je zde definován způsob pro obnovení informací serveru vlastněných transakčním koordinátorem jménem jeho účastníků.

V případě obnovy účastníka za `PUT` požadavku na `URI` reprezentované `/participant-recovery`, je navrácen účastník do původního stavu (původní `URI`, ...) dodaného během jeho registrace a dále se spustí pokus o obnovu přidružených transakcí s použitím nového `URI` účastníka. Originální `URI` lze získat provedením `GET` dotazu.

Podoba `PUT` požadavku:

Hlavička:

```
1 PUT /participant-recovery/{id} {protocol}
2 Host: example.com
3 Content-Type: application/json
```

Obsah:

```
{
    "new-address": "{URI}"
}
```

Přehled stavů

Přehled stavů, které mohou být vráceny při `GET` požadavku na `URI` transakčního koordinátora nebo daného účastníka:

- **`TransactionRollbackOnly`** – indikuje, že koncový bod je posléze navrácen do původního stavu,
- **`TransactionRollingBack`** – koncový stav je v procesu vrácení,
- **`TransactionRolledBack`** – nad koncovým bodem byl vykonán `rollback`,
- **`TransactionCommitting`** – koncový bod je v procesu potvrzení (to neindikuje, že transakce bude potvrzena),
- **`TransactionCommitted`** – došlo k potvrzení koncového bodu (potvrzení vykonaných operací),
- **`TransactionHeuristicRollback`** – všichni účastníci provedou `rollback` v situaci, kdy jsou požádání o provedení `commit`,
- **`TransactionHeuristicCommit`** – všichni účastníci provedou `commit` v situaci, kdy jsou požádání o provedení `rollback`,
- **`TransactionHeuristicHazard`** – někteří účastníci provedou `commit`, někteří `rollback` a u zbylých je výsledek neurčitý,
- **`TransactionHeuristicMixed`** – někteří účastníci provedou `commit` a někteří `rollback`,
- **`TransactionPreparing`** – koncový bod je v přípravě,
- **`TransactionPrepared`** – koncový bod je připraven,
- **`TransactionActive`** – transakce je aktivní.

Stavy, které jsou zasílany od transakčního koordinátora jednotlivým účastníkům (koncovým bodům) pro řízení `2PC` protokolu:

- **`TransactionPrepare`** – indikuje, že účastník by se měl připravit na následující potvrzení provedených změn,

- `TransactionCommit` – příjemce by se měl pokusit o provedení operace `commit`. V případě, že tato výzva již byla dříve obdržena, musí příjemce vrátit kód `410 Gone`.

- `TransactionRollback` – příjemce by se měl pokusit o provedení operace `rollback`. Opět, v případě, že tato výzva již byla dříve obdržena, musí příjemce vrátit kód `410 Gone`.

- `TransactionCommitOnePhase` – příjemce by se měl pokusit provést o operaci `commit` bez přípravné fáze. Pokud o toto byl již příjemce dříve požádán, musí navrátit kód `412 Precondition Failed`. V případě, že příjemce byl již ukončen, musí být navrácen opět kód `410 Gone`.

2.2.2 Příklad použití

Příklad realizující scénář z 1.2.2.1. Pro jednoduchost je zde opět uveden pouze scénář provedení úspěšné transakce.

1. **Vytvoření nové transakce**

   Vytvoření nové transakce s `id` 1, kterou vytváří `CRM` systém. Podoba požadavku:

   Hlavička požadavku:

   ```
   1 POST /transaction-coordinator HTTP/1.1
   2 Host: example.com
   3 Content-Type: application/json
   ```

   Příslušná odpověď:

   Hlavička odpovědi:

   ```
   1 HTTP/1.1 201 Created
   2 Location: https://example.com/transaction-coordinator/1
   3 Link: https://example.com/transaction-coordinator/1/
         terminator;
   4 rel="terminator"
   5 Link: https://example.com/transaction-coordinator/1/
         participants;
   6 rel="participants"
   ```

2. **Požadavek na vytvoření nového uživatele**

   Požadavek na vytvoření uživatele od `CRM` do účetního systému je obohacen o odkaz na transakci.

Podoba požadavku:

Hlavička požadavku:

```
POST /ais/ HTTP/1.1
Host: example.com
Content-Type: application/json
Link: https://example.com/transaction-coordinator/1;
rel="transaction"
```

Obsah požadavku:

```
{
  "firstname": "Jakub",
  "lastname": "Drabek",
  "birthday": "23.02.1992",
  "mail": "drabeja1@fit.cvut.cz"
}
```

Po přijetí takového požadavku se musí nejdříve účetní systém registrovat do procesu transakce a poté vykonat požadovanou operaci.

Uvažujme, že uživatelé jsou vedeni na `https://example.com/ais/user/` a `terminator` pro správu uživatelů je dostupný na `https://example.com/ais/user/terminator`. Podoba požadavku na registraci účetního systému do transakce:

Hlavička požadavku:

```
POST /transaction-coordinator/1/participant/ HTTP/1.1
Host: example.com
Content-Type: application/json
```

Obsah požadavku:

```
{
  "participant": "https://example.com/ais/user/",
  "terminator": "https://example.com/ais/user/
   terminator"
}
```

Příslušná odpověď obsahující `URL` zdroje pro obnovu s `id` 1.

Hlavička odpovědi:

```
HTTP/1.1 201 Created
Location: https://example.com/participant-recovery/1/
```

Následně může proběhnout provedení samotné požadované operace a vrácení jejího výsledku zpět do `CRM` systému.

Podoba takové odpovědi opět bude:

Hlavička odpovědi:

```
HTTP/1.1 201 Created
Location: https://example.com/ais/user/999
Content-Type: application/json
```

Obsah odpovědi:

```
{
  "user-id": 999
}
```

3. **Aktualizace informací o uživateli**

Obdobně, jako byl průběh vytvoření nového uživatele, bude probíhat proces aktualizací informací o uživateli. Nejdříve dojde k zaslání požadavku od `CRM` systému do účetního systému. Tedy:

Hlavička požadavku:

```
PUT /ais/user/999/update HTTP/1.1
Host: example.com
Content-Type: application/json
Link: https://example.com/transaction-coordinator/1;
rel="transaction"
```

Obsah požadavku:

```
{
  "department": "IT",
  "position": "development"
}
```

Dále je nutné registrovat nového účastníka – měněného uživatele. Tedy:

Hlavička požadavku:

```
POST /transaction-coordinator/1/participant/ HTTP/1.1
Host: example.com
Content-Type: application/json
```

Obsah požadavku:

```
{
  "participant": "https://example.com/ais/user/
   999",
  "terminator": "https://example.com/ais/user/999
   /terminator"
}
```

Příslušná odpověď s nastaveným `id` pro obnovu na 2.

Hlavička odpovědi:

```
HTTP/1.1 201 Created
Location: https://example.com/participant-recovery/2
```

Následně opět dojde k vykonání samotné operace aktualizace informací a vrácení odpovědi zpět `CRM` systému s podobou:

Hlavička odpovědi:

```
HTTP/1.1 200 Ok
```

4. **Potvrzení transakce**

Potvrzení transakce je realizováno pomocí `2PC` protokolu. Nejdříve je zaslán dotaz na potvrzení od `CRM` systému transačnímu koordinátoru:

Hlavička požadavku:

```
POST /transaction-coordinator/1/terminator HTTP/1.1
Host: example.com
Content-Type: application/json
```

Obsah požadavku:

```
{
  "tx-status": "TransactionCommit"
}
```

Vrácená odpověď:

Hlavička odpovědi:

```
HTTP/1.1 202 Accepted
```

Dále jsou vykonávány jednotlivé fáze `2PC` protokolu. Nejdříve probíhá přípravná fáze, uvažujme, že každý zaslaný dotaz vrátí kód `200 Ok`. Dotaz na zdroj nově vytvořeného uživatele:

Hlavička požadavku:

```
POST /ais/user/999/terminator HTTP/1.1
Host: example.com
Content-Type: application/json
```

Obsah požadavku:

```
{
  "tx-status": "TransactionPrepare"
}
```

Dotaz na zdroj správy uživatelů:

Hlavička požadavku:

```
1 POST /ais/user/terminator HTTP/1.1
2 Host: example.com
3 Content-Type: application/json
```

Obsah požadavku:

```
1 {
2   "tx-status": "TransactionPrepare"
3 }
```

Dále následuje druhá fáze protokolu, tedy dojde k potvrzení všech změn. Opět uvažujme, že každý zaslaný dotaz vrátí kód `200 Ok`.

Dotaz na zdroj nově vytvořeného uživatele:

Hlavička požadavku:

```
1 POST /ais/user/999/terminator HTTP/1.1
2 Host: example.com
3 Content-Type: application/json
```

Obsah požadavku:

```
1 {
2   "tx-status": "TransactionCommit"
3 }
```

Dotaz na zdroj správy uživatelů:

Hlavička požadavku:

```
1 POST /ais/user/terminator HTTP/1.1
2 Host: example.com
3 Content-Type: application/json
```

Obsah požadavku:

```
1 {
2   "tx-status": "TransactionCommit"
3 }
```

2.2.2.1 Zhodnocení

Výhody

- Uvažuje případný vznik heuristických případů. Způsob řešení takového případu je však opět přenechán na konkrétní imlementaci.
- Komplexnější a bezpečnější řešení než předchozí. Primárně uvažuje použití `2PC` protokolu.
- Izolace provedených změn přenechána na zúčastněné služby. Transakční koordinátor nemusí být dále rozšiřován v případě zapojení se nové služby do transakčního zpracování.
- Lze snadno zahrnout další zúčastněné služby bez nutnosti úprav transakčního kordinátora.

Nevýhody

- Nutnost upravit všechny služby tak, aby umožnili práci s transakčním koordinátorem a dále měli zavedený mechanismus zde označený jako `terminator` nebo podporovali příslušné služby pro zajištění potřebných operací `commit` a `rollback`.
- V případě, kdy služba není zahrnuta mezi ty, které podporují transakční zpracování, musí být minimálně schopna reagovat na příchozí požadavek o zařazení do transakčního zpracování chybou.
- Neřeší požadavek na izolaci provedených změn. Ten v případě potřeby musí řešit účastnící se služby.
- Klient není opět izolován od povědomí o transakci. Opět nutnost řešit dalším přidaným mechanismem.
- Komplexnější řešení vedoucí na vyšší složitost a vysokou komunikační zátěž. Je nutné provést přibližně o $1+4x$ požadavků více, kde x je počet zúčastněných služeb.

2.3 Použití Try-Cancel/Confirm návrhového vzoru

2.3.1 Popis řešení

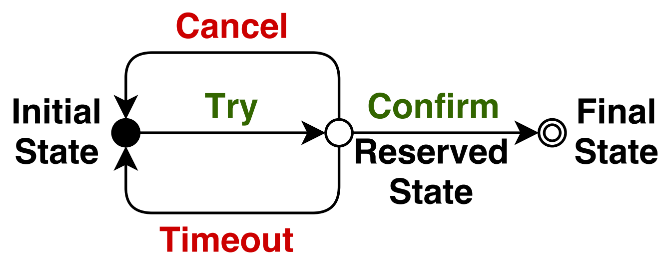
Další možné řešení je založeno na `Try-Cancel/Comfirm` návrhovém vzoru 2.3.1. Jedná se o jednoduchý protokol taktéž zaručující atomicitu a obnovu přes distribuované služby používající `REST`. Cílem tohoto protokolu je minimalizovat požadavky kladené na jednotlivé služby účastnící se transakčního zpracování. Nepožaduje žádné další rozšíření `HTTP` protokolu a jeho použití opět předpokládá využití transakčního koordinátora nebo služby vystupující v jeho roli. Toto řešení vychází z [35].

Vedle zajištění atomicity uvnitř distribuované transakce, tento protokol dále umožňuje zahrnout interakce více služeb do jednoho logického kroku za zajištění konzistence zahrnutých zdrojů uvnitř distribuované transakce.

Avšak výjimkou v tomto protokolu je požadavek na izolaci, který je zde porušen. Příčinou je možnost zobrazení stavu zdroje zahrnutého uvnitř transakce. To může být vhodné pro různé scénáře použítí, například v případě, kdy chceme zamezit zaslání duplicitního požadavku na danou službu.

`Try-Cancel/Confirm` (`TCC`) návrhový vzor

`Try-Cancel/Confirm` je protokol ideální pro použití v podnikových modelech skládajících se ze dvou fází. Jeho průběh je zobrazen na následujícím diagramu:



Obrázek 2.4: Schéma `TTC` protokolu 2.3.1

V tomto protokolu jsou všechny požadakvy „vyzkoušeny“ (`try` – vykonání požadované operace končící úspěchem nebo neúspěchem) a dále zůstávají v „rezervované“ fázi, dokud nedojde k vykonání operace `commit`, nebo operace `rollback`. Toto složení vede k přirozenému, volně spřaženému transakčnímu modelu. Odvolání transakce může být vyvoláno spontalně, např. po vypršení časového limitu (`timeout`), vzniklou chybou nebo některou další vnější událostí.

Protokol

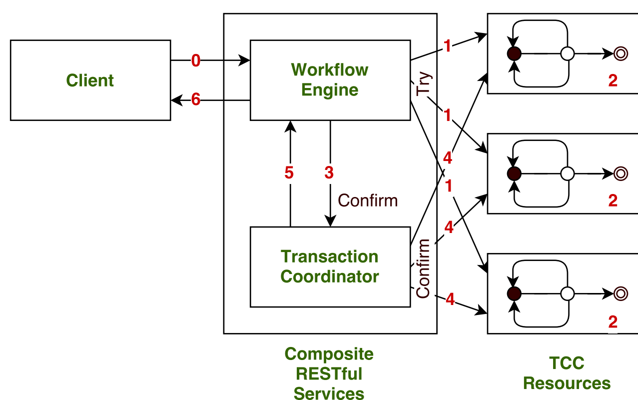
Tento protokol je opět založen na principu využití transakčního koordinátora nebo služby řídící globální transakci. Ta se skládá z jednotlivých autonomních operací, které jsou na sobě nezávislé a které se řídí pomocí `TCC` návrhového vzoru. Jedná se tedy o množinu krátkých transakcí, splňujících `ACID` vlastnosti pro každý ze zdrojů zahrnutých v globální transakci.

Tento protokol zavádí následujcící definice transakce: Transakce T je série volání R_i mezi `RESTful` službami (účastníci transakce) S_i, které buďto musí skončit potvrzením anebo zrušením společně. Všechna volání tedy skončí explicitním potvrzením $R_{i,confirm}$ nebo všechna R_i jsou odvolány.

Úspěšný průběh

1. Transakční zpracování T probíhá interakcí skrz více `RESTful` služeb S_i.
2. Interakce R_i může vést ke změně stavu účastníka S_i identifikovaného dle své jedinečné `URI` – ta odpovídá $R_{i,confirm}$.
3. Jakmile průběh T úspěšně skončí, množina `URI` pro potvrzení a další potřebná aplikační logika je předána transakční službě (nebo transakčnímu koordinátoru).
4. Transakční služba poté zašle idemponentní[6] `PUT` požadavek pro `commit`, nebo `DELETE` požadavek pro `rollback` na všechny $R_{i,confirm}$.

Tento průběh, zobrazaný níže, splňuje požadavek na atomicitu. Každá zúčastněná strana zde dostane konzistentní požadavek na odvolání či potvrzení. Zde všichni účastníci provedou obdobně zrušení své vnitřní transakce.



Obrázek 2.5: Schéma protokolu

[6] Opakovaným požadavkem dosáhneme stejného výsledku.

Proces obnovy
V případě vyskytu chyby nebo překročení `timeout` a následné nutné obnovy zdrojů do původního stavu, je zde explicitně uvažováno, že každá služba sama umožňuje provedení této operace (obdobně jako v druhém řešení, kde je předáván odkaz na `terminator`). Zůstává zde ale problém provedení obnovy do atomického stavu napříč všemi zúčastněnými stranami.

Proces obnovy je zde tedy akce vyvolaná pomocí koordinátora nebo některou zúčastněnou službou. V případě koordinátora se jedná o přirozenou funkcionalitu, na druhé straně se jedná o přirozené chování i z pohledu účastníka transakce. Ten, ačkoliv neví o transakci T, bude chtít uvolnit své vyhrazené prostředky v nejkratším možném čase (jeden z požadavků pro `REST`). Scénáře obnovy pro oba pohledy:

Obnova zúčastněné služby
Obnova zúčastněné služby S_i v transakci má následující podobu, pokud uvažujeme výskyt během scénáře 2.3.1:

1. Pro obnovu do kroku 2 nedělej nic.
2. Pro obnovu po kroku 4 nedělej nic.
3. Pro obnovu mezi kroky 2 a 4 vyvolej $R_{i,cancel}$ autonomně. Toto může být vyvoláno nařiklad překročením `timeout`.

Obnova koordinátora
Obdobně jako jednotlivý účastníci je zde uvažováno, že služba koordinátora je schopná obnovy svého stavu. Dále koordinátor musí zajistit, že dojde k obnově všech zúčastněných služeb do stavu před transakcí T. Nejkritičtějším místem selhání je zde krok 4, kde je zúčastněno více účastníků najednou a v případě selhání se jedná o problematické místo [7]. Naivní protokol pro proces obnovy:

1. Pro obnovu před krokem 2 nedělej nic.
2. Pro obnovu mezi kroky 2 a 4 nedělej nic.
3. Pro obnovu po kroku 4 nedělej nic.
4. Pro obnovu během kroku 4 opakuj $R_{i,confirm}$ s každým účastníkem S_i dokud není $R_{i,confirm}$ úspěšně provedena[8].

[7] Selhání je zde problematické z důvodu nevědomosti účastníka o transakci T.

[8] Pro toto je nutné, aby koordinátor vedl záznam informací o všech účastnících transakce a provedených změnách před krokem 4.

Diskuze k protokolu

Garance atomicity:
I v případě výskytu chyby je za použití tohoto protokolu zachována atomicita. To je důsledkem:

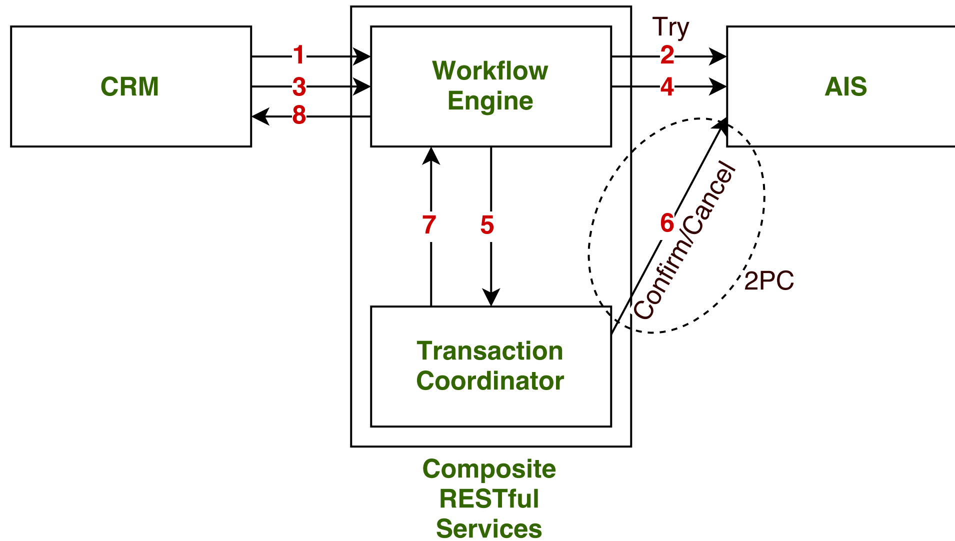
1. Pokud nenastane žádná chyba, pak proběhnou kroky 1 – 4 a každá R_i končí úspěchem.
2. V případě chyby před krokem 2, neexistuje žádná R_i, nic se tedy nestalo.
3. V případě chyby během kroku 2, nebo po něm do kroku 4, jsou všechny R_i autonomně odvolány nějakou S_i (žádná změna nebyla přijata).
4. V případě chyby během kroku 4 koordinátor opakuje každou $R_{i,confirm}$, dokud neuspěje. Zde se ale může opět vyskytnou jeden z heuristických případů obdobně jako u druhého řešení. To je diskutováno níže.
5. V případě chyby po kroku 4 jsou již všechny $R_{i,confirm}$ vyřízeny. Tedy atomicita je již zaručena.

Heuristický případ:
Tento protokol obsahuje jedno slabé místo, které je zmíněno v diskuzi o garanci atomicity. Jedná se o případ, kdy během kroku 4 některá, či více služeb, odvolají provedené změny z důvodu překročení `timeout` během situace, kdy koordinátor provádí potvrzování. To vede k situaci, kdy někteří účastníci potvrdí své změny a jiní je naopak odvolají.

Řešením tohoto problému může být částečně využití `2PC` protokolu (1.1.5.1) během kroku 4. Další variantou, jak tento případ vyřešit, je hlídat překročení `timeout` pomocí služby koordinátora. Ta hlídá časové limity a v případě překročení nějakého limitu platnosti, již neumožní zaslat $R_{i,confirm}$ a provede `rollback` nad všemi účastníky.

2.3.2 Příklad použití

Příklad použití 1.2.2.1 je zachycena na následujícím diagramu:



Obrázek 2.6: Schéma úspěšného scénáře pro TCC protokol

Tento diagram zachycuje průběh:

1. `CRM` systém zašle požadavek na vytvoření nového uživatele.
2. Požadave je předán do účetního systému. Ten vytvoří nového uživatele a vrátí `201 Created` společně s `URI` na nově vytvořeného uživatele, které je vráceno `CRM` systému.
3. `CRM` systém zašle požadavek na aktualizaci uživatele.
4. Ten je opět předán účetnímu systému a je provedena aktualizace údajů. Účetní systém vrátí úspěch `200 Ok`.
5. Transakčnímu koordinátoru je předána `URI` pro potvrzení provedených změn – $R_{confirm}$.
6. Transakčnční koordinátor provede potvrzení provedených změn na $R_{confirm}$ za použití `2PC` protokolu.

7.–8. Výsledek o úspěchu je předán zpět do `CRM` systému.

Příklad návrhu provedené komunikace pro úspěšně provedený scénář:

1. **Požadavek na vytvořené nového uživatele:**

   V prním kroku je učiněno zaslání požadavku na vytoření nového uživatele od CRM systému složené REST službě realizující zpracování transakce. Zde uvažujme, že daná služba se skládá z transakčního koordinatora dostupného na https://example.com/trcoordinator a integrační vrstvy přebírající požadavky od CRM systému dostupné na http://example.com/integration/users.

   Podoba požadavku:

   Hlavička požadavku:

```
POST /integration/users HTTP/1.1
Host: example.com
Content-Type: application/json
```

   Obsah požadavku:

```
{
  "firstName": "Jakub",
  "lastName": "Drabek",
  "birthday": "23.02.1992",
  "mail": "drabeja1@fit.cvut.cz"
}
```

2. **Vytvoření transakce a předání požadavku AIS:**

   Integrační vrstva po obdržení požadavku na vytvoření nového uživatele od CRM systému, ověří, zda požadovaná akce vede na transakční zpracování. V případě že ano, zašle požadavek na vytvoření nové transakce transakčnímu koordinátoru. Požadavek bude vypadat:

   Hlavička požadavku:

```
POST /trcoordinator HTTP/1.1
Host: example.com
Accept: application/json
```

   Odpověď obsahující odkaz na nově vytvořenou transakci:

   Hlavička odpovědi:

```
HTTP/1.1 201 Created
Location: https://example.com/trcoordinator/1
```

   Po tomto kroku je původní požadavek na vytvoření nového uživatele rozšířen o hlavičku obsahující id transakce a následně předán AIS systému. Ten vytvoří nového uživatele a vrátí integrační vrstvě odpověď

s odkazem na nově vytvořeného uživatele. Ta tuto odpověď předá zpět CRM systému. Podoba takové odpovědi:

Hlavička odpovědi:

```
HTTP/1.1 201 Created
Location: https://example.com/crm/users/1
```

3. **Aktualizace informací o uživateli:**

   Požadavak na přiřazení pracovní pozici nově vytvořenému uživateli jde od CRM systému integrační vrstvě, ta k požadavku přidá id příslušné transakce a přepošle jej na AIS systém. Podoba obohaceného požadavku:

   Hlavička požadavku:

```
PUT /ais/user/1/update HTTP/1.1
Host: example.com
Content-Type: application/json
Transaction-Id: 1
```

   Obsah požadavku:

```
{
  "department": "IT",
  "position": "development"
}
```

   To vrátí odpověď o úspěchu provedení operace s návratovým kódem 200 Ok.

4. **Potvrzení transakce:**

   Přiřazením pracovní pozice novému uživateli je ukončen transakční průběh a provedené změny musí být potvrzeny. Potvrzení vyvolá integrační vrstva zasláním požadavku k provedení operace commit na transakčního koordinátora. To provedeme pomocí POST požadavku na adresu transakce s nastavenými účastníky. Možná podoba takového požadavku:

   Hlavička požadavku:

```
POST /trcoordinator/1 HTTP/1.1
Host: example.com
Content-Type: application/json
```

   Obsah požadavku:

```
{
  "participants": ["https://example.com/ais/user/
    1"]
}
```

To vyvolá průchod 2PC protokolem mezi transakčním koordinátorem a AIS systémem. Pro tento účel uvažujme, že k potvrzení vykonaných změn v AIS lze dosáhnout pomocí provedení POST požadavku s požadovaným stavem na adresu: `https://example.com/ais/users/commit`.

V první fázi protokolu bude od transakčního koordinátora zaslán požadavek:

Hlavička požadavku:

```
1 POST /ais/users/commit HTTP/1.1
2 Host: example.com
3 Content-Type: application/json
```

Obsah požadavku:

```
1 {
2   "transactionState" : "PREPARE"
3 }
```

A po navrácení odpovědí `200 Ok` od AIS systému je vykonána druhá fáze potvzovacího protokolu. Podoba požadavku:

Hlavička požadavku:

```
1 POST /ais/users/commit HTTP/1.1
2 Host: example.com
3 Content-Type: application/json
```

Obsah požadavku:

```
1 {
2   "transactionState" : "COMMIT"
3 }
```

V případě navrácení `200 Ok` již víme, že změny byly úspěšně potvrzeny.

2.3.3 Zhodnocení

Výhody

- Účastník transakce musí pouze podporovat operace dle `TCC` vzoru. Neví nic o probíhající transakci.
- Mezi účastníky transakce není sdílen žádný transakční kontext.
- Uvažuje odstínění klienta od transakčního zpracování.
- Menší komunikační zátěž než v případě druhého řešení. Pro více zúčastněných slůžeb je zvýšení přibližně o $2 + 3x$ požadavků navíc, kde x představuje počet zúčastněných služeb.
- Snadněji rozšiřitelné o další služby.

Nevýhody

- Předpokládá nesplnění požadavku na izolaci.
- Účastníky transakce je nutné upravit tak, aby byly splněny požadavky `TCC` vzoru – podpora operací `rollback` a `commit`.
- Při použití dvou-fázového potvrzení vyšší komunikační zátěž.

2.4 Další nalezená řešení

Další zajímavé možnosti, které v této práci nejsou dále rozebírány, ale můžeme se s nimi setkat jsou:

- `RETRO RESTful` transakční model [36],
- použití `publish/subscribe` mechanismu [37].

2.5 Shrnutí

Všechna zde uvedená řešení přímo nezavádí konkrétní způsob implementace podpory transakčního zpracování v technologii `REST`. Jedná se o návrhy protokolů a možností, lépe či hůře popsaných, jak toho docílit. Každé z řešení má své výhody a nevýhody. Zde nejvýraznější nevýhodou, která se vyskytuje u většiny zmíněných řešení, je nutnost, aby každá služba vystupující uvnitř transakce podporovala mechanismy pro operace `commit` a `rollback`. Tedy, aby každá služba věděla o přítomnosti v transakci, nebo měla zajištěn mechanismus pro odvolání provedených změn (to opět vede k nutnosti informovat službu o zahájení transakce). Tím dochází k porušení požadavku na bezstavovost.

Druhým porušeným principem z pohledu transakce, který je spojen s prvním, je izolace. Uvedené možnosti neřeší, jak zajistit izolaci provedených změn uvnitř zúčastněných služeb anebo v případě třetí varianty, kde je použit `TCC` protokol, berou tento nedostatek na vědomí a dále se mu nevěnují.

2.5.1 Srovnání nalezených řešení

V následující části uvažujme toto pojmenování jednotlivých řešení:

- Řešení 1 - Transakce jako zdroj, první varianta,
- Řešení 2 - Transakce jako zdroj, druhá varianta
- Řešení 3 - Návrh transakčního koordinátora,
- Řešení 4 - Řešení s použitím `TCC` protokolu.

2.5.1.1 Splnění ACID vlastností

	Řešení 1	Řešení 2	Řešení 3	Řešení 4
Atomicita	✓ Splněno	✓ Splněno	✓ Splněno	✓ Splněno
Konzistence	≈ Částečně	≈ Částečně	✓ Splněno	✓ Splněno
Izolovanost	× Nesplněno	× Nesplněno	× Nesplněno	× Nesplněno
Trvanlivost	✓ Splněno	✓ Splněno	✓ Splněno	✓ Splněno

V případě požadavku na konzistencí používají první dvě řešení pouze jedno-fázový potvrzovací mechanismus, který vede na vyšší možnost výskytu jednoho z heuristických případů. Zbylá řešení již uvažují použití bezpečnějšího dvou-fázového potvrzení.

2.5.1.2 Zavedené požadavky

V následující tabulce jsou porovnány jednotlivá řešení vůči zavedeným požadavkům na transakčná zpracování v `REST`.

	Řešení 1	Řešení 2	Řešení 3	Řešení 4
Neinvazivní ř.	✓ Splněno	× Nesplněno	× Nesplněno	× Nesplněno
Izolace klienta	× Nesplněno	× Nesplněno	× Nesplněno	✓ Splněno
Kom. zátěž	× Nesplněno	✓ Splněno	× Nesplněno	≈ Částečně
Rozšiřitelnost	× Nesplněno	≈ Částečně	✓ Splněno	✓ Splněno

Srovnání potřebných požadavků, které je potřeba provést navíc, je zachyceno v následné tabulce (přibližné horní hranice).

	Řešení 1	Řešení 2	Řešení 3	Řešení 4
Požadavků navíc	$\approx 2x$	$\approx 3 + x$	$\approx 1 + 4x$	$\approx 2 + 3x$

Jednotlivé body jsou podrobněji rozebrány v sekcích „Zhodnocení“ u každého rozebíraného řešení.

Kapitola 3

Ověření vybrané metody

3.1 Vybraná metoda

Pro implementační ověření vybrané metody bylo zvoleno třetí uvedené řešení, tedy protokol popsaný v 2.3 používající `Try-Cancel/Confirm` vzor, do kterého byly zavdeny drobné úpravy. Hlavní odlišností je informování transakčního koordinátora o účastnících transakce, které je provedeno v počáteční fázi při vytváření transakce a použití některých principů z 2.2, hlavně mechanismu `terminator` pro ovládání stavu transakce v zúčastněných službách.

Hlavní důvody pro zvolení tohoto řešení jsou:

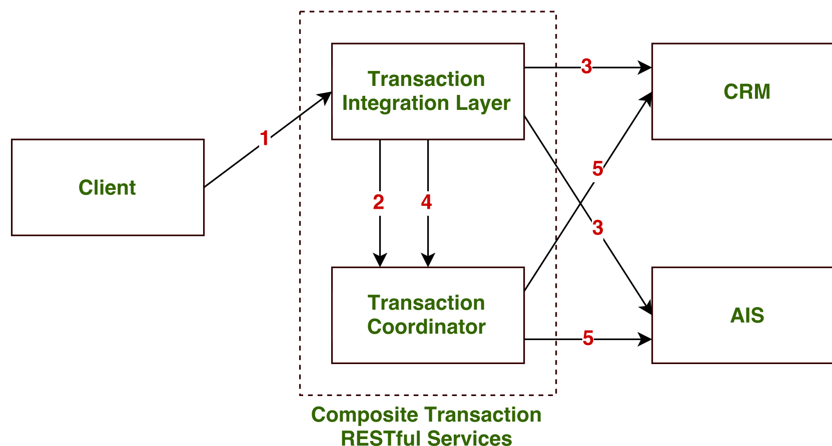
- jednoduché, snadno implementovatelné řešení,
- splnění požadavku na oddělení klienta od povědomí o probíhající transakci,
- přiměřený nárůst přidané komunikace,
- snadno rozšiřitelný a konfigurovatelný (viz. dále),
- v rozšířené verzi splňuje požadavky `ACID` (za cenu porušení principu bezstavovosti).

3.1.1 Implementovaný příklad

V rámci ověření je použit rozšířený příklad použití z 1.2.2.1. Zde vystupuje klient jako služba registrující nového uživatele, systém pro řízení vztahu se zákazníky (`CRM`) a účetní systém (`AIS`). Zde klient zasílá požadavky na vytvoření nového uživatele do `CRM` systému, ale zároveň je požadováno, aby nový uživatel byl současně založen v `AIS` a vše probíhalo v rámci transakčního zpracování.

To je zajištěno za použití složené `RESTful` služby skládající se z integrační vrstvy (na úrovni `middleware`, separátní `REST`-ové služby, . . .) a transakčního koordinátora.

Vše je zachyceno na následujícím diagramu:



Obrázek 3.1: Schéma implementovaného řešení

Diagram zachycuje následující průběh:

1. Klient zasílá požadavky na integrační vrstvu, která z venčí vystupuje v roli požadované služby (`CRM` systému). Implementuje tedy návrhový vzor Fasáda [38]. A dále je od integrační vrstvy přijata příslušná odpověď – očekávaný výsledek, chyba, apod.

2. Integrační vrstva se na základě požadavku od klienta rozhodne o vytvoření nové transakce. V případě, že požadovaná operace vede na transakční zpracování, integrační vrstva zažádá transakčního koordinátora o vytvoření nové transakce s daným nastavením, která je ji v případě úspěchu vrácena.

3. Integrační vrstva dále zpracovává požadavky od klienta a ty dále deleguje na jednotlivé zúčastněné služby (`CRM` systém a `AIS` systém). V případě transakčního zpracování je k požadavkům zasílaným zúčastněným službám připojena hlavička `Transaction-Id` s hodnotou identifikačního čísla dané transakce.

   Zúčastněná služba v případě obdržení takového požadavku musí sama zajistit další příslušné kroky pro zajištění transakčního zpracování.

4. Integrační vrstva se na základě požadavku od klienta rozhodne zda ukončit probíhající transakci (operace `commit` a operace `rollback`), nebo v případě výskytu chyby v komunikaci se zúčastněnými službami, je zaslán konkrétní požadavek na transakčního koordinátora o provedení operace `commit`, nebo `rollback`.

   Transakční koordinátor po obdržení takového požadavku dále provede danou akci nad všemi zúčastněnými službami transakčního zpracování,

při čemž pro potvrzení provedených změn je použit 2PC protokol. Pro operaci `commit` je zasílán `POST` požadavek na `terminator` dané služby s nastaveným požadovaným stavem (pro první fázi protokolu stav `PREPARE` a pro druhou fázi stav `COMMIT`). Operace `ROLLBACK` je provedena zasláním `DELETE` požadavku opět na `terminator` dané služby.

3.1.2 Úprava zúčastněných služeb

Jak je zmíněné výše, tato varianta počítá s faktem, že služby zúčastněné v transakčním zpracování umí pracovat s informací o probíhající transakci a vystavují rozhraní `terminator`. Zde jsou příslušné služby, `CRM` a `AIS`, upraveny tak, aby při příchozím požadavku s nastaveným `Transaction-Id` a dosavaď nenastavenou transakcí, došlo k vytvoření záznamu o nově probíhající transakci.

V tuto chvíli je, pro zjednodušení, služba uzamčena pro modifikující požadavky, které neobsahují příslušné `id` transakce. Změny vykonané uvnitř transakčního zpracování jsou ukládány do separátního pomocného úložiště, které umožňuje oddělit stav před transakcí a stav během probíhající transakce. Operace pro čtení zůstavají volně dostupné a vrací výsledky dle přítomnosti `id` transakce v příchozím požadavku. V případě přítomnosti validního `id` transakce jsou vráceny výsledky z pomocného úložiště, opačně je použito úložiště původní.

Implementované služby vystavují jednotné rozhraní pro ovládání uživatelů vedených v systému. Pro příklad uvažujme rozhraní dostupné na `http://example.com/users`. Možné operace jsou:

- přečtení všech uživatelů vedených ve službě pomocí `GET` požadavku na `http://example.com/users` – tato operace vrátí kolekci všech uživatelů v systému
- přidání nového uživatele do systému pomocí `POST` požadavku na `http://example.com/users` – tato operace zpět vrátí uživatelská data a odkaz na nově vytvořeného uživatele v rámci systému v hlavičce `Location`,
- přečtení jednoho konkrétního uživatele pomocí `GET` požadavku na `http://example.com/users/{id}` – to vrátí v případě validního uživatelského `id` vrátí data požadovaného uživatele,
- smazání uživatele ze systému pomocí `DELETE` požadavku na `http://example.com/users/{id}`,
- aktualizace uživatele pomocí `POST` požadavku na adresu aktualizovaného uživatele `http://example.com/users/{id}`,
- změna pracovního umístění uživatele pomocí `PUT` požadavku na `http://example.com/users/{id}/position`,

- potvrzení provedených změn uvnitř transakce provedením `POST` požadavku s nastaveným požadovaným stavem transakce a nastavenou hlavičkou `Transaction-Id` na `http://example.com/users/terminator`,
- odvolání provedených změn uvnitř transakce pomocí `DELETE` požadavku s nastavenou hlavičkou `Transaction-Id` na `http://example.com/users/terminator`.

3.1.3 Integrační vrstva

Implementovaná integrační vrstva vystavuje stejné rozhraní jako `CRM` systém. Příchozí požadavky na založení nového uživatele dále předává do samotného `CRM` a dále do `AIS` systému. Mezi tím dochází k případnému řízení transakčního zpracování za použití transakčního koordinátora.

Rozhodnutí o započetí transakce a jejím následném ukončení, ať už potvrzením nebo odvoláním, je vykonáno v integrační vrstvě transakční služby. Ta pro učinění takového rozhodnutí musí být nakonfigurována. Zde je konfigurace dosaženo pomocí `XML` [39] konfigurace ze souboru. Ten kromě údajů o započetí a ukončení transakce obsahuje i další potřebné informace, jako jsou účastníci transakčního zpracování a `timeout`, které jsou dále předány transakčnimu koordinátoru.

Příklad použité konfigurace v implementovaném příkladu, která obsahuje základní data pro nastavení transakce:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <routes>
    <route>
      <name><![CDATA[/crm/users]]></name>
      <from><![CDATA[http://localhost:9090]]></from>
      <to><![CDATA[http://localhost:9191:/users]]></to>
      <transaction_settings>
        <timeout>3600</timeout>
        <begin_action>
          <method>POST</method>
          <path><![CDATA[/]]></path>
        </begin_action>
        <end_action>
          <method>PUT</method>
          <path><![CDATA[/{userId}/position]]></path>
        </end_action>
        <participants>
          <!-- CRM -->
            <participant>
              <uri>
                <![CDATA[http://localhost:9191/users]]>
              </uri>
              <terminator_uri>
                <![CDATA[http://localhost:9191/users/
    terminator]]>
              </terminator_uri>
            </participant>
            <!-- CRM -->
            <participant>
              <uri>
                <![CDATA[http://localhost:9292/users]]>
              </uri>
              <terminator_uri>
                <![CDATA[http://localhost:9292/users/
    terminator]]>
              </terminator_uri>
            </participant>
        </participants>
      </transaction_settings>
    </route>
  </routes>
</configuration>
```

Kořenovým element je zde element `configuration` s podelementem `routes`. Ten zapouzdřuje klíčové elementy `route`, které definují „cestu“. Ta představuje určitý průchod akcemi, který se skládá z počáteční akce (začátek cesty), mezi akcemi a ukončující akci (konec cesty).

Element `route` dále obsahuje elementy:

- `name` představující jméno cesty, k jednoznačné identifikaci,
- `from` představující počáteční akci, která v případě nastavení transakce spustí transakční zpracování,
- `to` představující ukončující akci, která v případě nastavení transakce ukončí transakční zpracování,
- `transaction_settings` představující nastavení transakčního zpracování, to dále obsahuje prvky:
  - `timeout` reprezentující dobu platnosti transakce v sekundách,
  - `begin_action` představující počáteční akci, která spustí transakční zpracování, s podelementy `method` k identifikaci spouštěcí `HTTP` metody a element `path` identifikující samotnou akci v reprezentaci relativní `URL` cesty,
  - `end_action` představující ukončující akci, která ukončí transakční zpracování se stejnými podelementy jako nastavení počáteční akce,
  - `participants` obsahující elementy `participant`, které reprezentují jednotlivé účastníky transakčního zpracování. Tedy služby, které se během průchodu cestou účastní transakčního zpracování, a které jsou dále transakčním koordinátorem informovány pro provedení operací `commit` a `rollback`.

    Každý element `participant` dále obsahuje `URI` adresu samotné služby v elementu `uri` a `URI` adresu na `terminator` v elementu `terminator_uri`.

3.1.4 Transakční koordinátor

Transakční koordinátor vystavuje rozhraní pro ovládání transakce. Jeho úkolem je řídit zahájení a ukončení transakce. Uvažujme dostupnost této služby na adrese `http://example.com/transactions`. Potom nad transakčním koordinátorem lze vykonat následující akce:

- Vytvoření nové transakce pomocí `POST` požadavku na adresu `http://example.com/transactions` s obsaženým nastavením pro nově vytvářenou transakci. To vytvoří a vrátí objekt nově vytvořené transakce s jejímž `id` poté probíhá následná komunikace.

- Získání informace o transakci provedením `GET` požadavku na adresu dané transakce `http://example.com/transactions/{id}`.
- Vyvolání operace `rollback` nad transakcí lze provést vykonáním `DELETE` požadavku na adresu konkrétní transakce, tedy `http://example.com/transactions/{id}`. To vyvolá proces odvolání změn nad všemi účastníky transakčního zpracování.
- Vyvolání operace `commit` nad transakcí je proveden pomocí `POST` požadavku opět na adresu transakce `http://example.com/transactions/{id}`. To vyvolá průchod dvou-fázovým protokolem, kde jsou nejdříve všichni účastníci vyzvání k přechodu do přípravné fáze, a v případě úspěchu u všech účastníků, je provedo potvrzení provedených změn. Pokud dojde k selhání během přípravné fáze, je vyvolána operace `rollback`.

3.2 Implementační detaily

3.2.1 Vývojové prostředí

Všechny služby jsou implementovány pomocí programovacího jazyka `Java` 1.8 s použitím frameworku Spring Boot [40] ve verzi 1.5.3. Každá služba je vedena jako projekt pro vývojové prostředí Netbeans [41] verze 8.2 s použitím nástroje Maven 3 [42].

Základní projekt byl vygenerován pomocí Spring Initializr [43] s balíky:

- `Web` – obsahuje knihovny pro tvorbu webových stránek, Spring MVC a zabudovaný Apache Tomcat [44],
- `Rest Repositories` – knihovny pro vystavení Spring Data datových uložišť pomocí `REST` skrze `spring-data-rest-webmvc`,
- `H2` – `H2` databáze se zabudovanou podporou,
- `JPA` – JPA Persistence API obsahující `spring-data-jpa`, `spring-orm` a Hibernate,
- `Thymeleaf` – šablonovací systém,
- `Actuator` – nástroje pro sledování a správu aplikace (pro funkčnost není potřeba),

Pro potřeby klienta je ještě navíc využito frameworku `bootstrap` [45] a `jQuery` [46] ve verzi 3.2.0.

Jednotlivé služby jsou implementovány v projektech:

- webová stránka obsahující jednoduchý formulář pro registraci nového uživatele v projektu `dp_client`,

- RESTful služba v roli itegrační vrstvy v projektu dp_integration,
- RESTful služba v roli CRM systému v projektu dp_crm,
- RESTful služba v roli AIS systému v projektu dp_ais,
- RESTful transakční koordinátor v projektu dp_trcoordinator.

Každý projekt je veden pod strukturou:

source files .... zdrojové Java soubory
- domain .... DTO třídy
- exceptions .... výjimky
- repositories ...... JPA repozitáře a repozitáře pro dočasné ukládání provedených změn v transakci
- serviceREST služby a služby sloužící ke komunikaci mezi jednotlivými systémy
- {project_name}Application.java .... hlavní třída spouštěcí aplikaci
- ServletInitializer.java .1 resources .... ostatní zdroje
- static .. statické zdroje pro webové stránky - pouze v případě klienta
- templates .... HTML šablony
- application.properties .... Property soubor pro nastavení zabudovaného serveru a nastavení používaných konstant

V případě integrační služby je v ostatních zdrojích navíc přítomen property soubor s konfigurací samotné vrstvy.

3.2.2 Použité techniky

Služby jsou zde implementovány jen pro splnění náležitostí k ověření funkčnosti průběhu transakčního zpracování. To znamená, že každá služba využívá pouze JPA úložišt [47], která jsou vedeny v operační paměti, pro simulaci perzistentních úložišť.

Pro zajištění izolace provedených změn je použito úložiště simulovaného in-memory seznamem, který kopíruje původní stav před zahájením transakce a z nějž je poté provedena synchronizace s „perzistentním“ úložištěm po skončení transakčního zpracování. Díky uzamčení služby v rámci jedné transakce je nutné řešit pouze synchronizaci provedených změn a oddělit, nad kterým úložištěm je pracováno při nemodifikujících požadavcích, které byly obdrženy během běžící transakce.

Integrační vrstva pro řízení požadavků používá Mapu [48], kde klíčem jsou informace o klientu (adresa spojena s názvem klienta) a hodnotou aktuálně probíhající transakce vytvořené transakčním koordinátorem nebo prázdná hodnota.

3.3 Zaváděcí dokumentace

3.3.1 Pomocí `IDE`

Implementace služeb je vedena jako Netbeans Maven projekt, který obsahuje závislosti pro zabudovaný Apache Tomcat, na kterém lze danou službu spustit. Spuštění služby pomocí vývojového prostředí provedeme:

1. otevřeme projekt dané služby v Netbeans nebo jiném vývojovém prostředí podporující sestavení pomocí nástroje Maven,
2. dále je potřeba zkontrolovat nastavení spuštění služby na zabudovaném aplikačním serveru, to v Netbeans provedeme:
   a) přejdeme do **Properties** daného projektu,
   b) zvolíme záložku **Actions** a vybereme akci **Run project**,
   c) zde v **Execute Goals** musí být vyplněna hodnota:
   `org.springframework.boot:spring-boot-maven-plugin:1.5.2.RELEASE:run`
   v případě použití Spring Boot verze 1.5.2,
   d) pole **Set Properties** necháme prázdné,
3. spustíme sestavení služby, to zajistí stažení všech potřebných knihoven,
4. a poté je již možné službu spustit standardním způsobem v `IDE`.

3.3.2 Pomocí příkazové řádky

Pro spuštění jednotlivých služeb pomocí příkazové řádky je nutné mít minimálně nainstalovaný nástroj pro sestavení Maven 3 a mít cesty k jeho spustitelným souborům zavedených v proměnném prostředí.

Poté lze každou službu spustit pomocí příkazové řádky příkazem `mvn spring-boot:run`. To vyvolá sestavení služby a její spuštění na lokálním prostředí. Stejnou akci provede spuštění přiložených `bat` souborů určených pro systémy MS Windows.

Jednotlivé služby po spuštění běží na lokálním prostředí, přístup k nim je přes adresy:

- klient na adrese `http://localhost:9090/`,
- `CRM` systém na adrese `http://localhost:9191/`,
- `AIS` systém na adrese `http://localhost:9292/`,
- transakční koordinátor na adrese `http://localhost:9393/`,
- integrační vrstva na adrese `http://localhost:9494/`.

Pro změnu portu je nutné upravit hodnotu `server.port` v souboru `application.properties`.

3.4 Příklad použití

Příklad použití implementovaných služeb začíná na registračním formuláři klienta. Zde se vyplní základní osobní informace jako uživatelké jméno, křestní jméno, příjmení a e-mailová adresa. Vše je zachyceno na obrázku B.1. Po potvrzení zadaných údajů dojde k následujícímu průchodu:

1. data od klienta jsou zaslána v požadavku na vytvoření nového uživatele integrační vrstvě,
2. integrační vrstva z konfigurace zjistí, že příchozí požadavek spouští transakční zpracování a zašle požadavek transačnímu koordinátoru na vytvoření nové transakce,
3. transakční koordinátor vytvoří novou transakci, kterou vrátí integrační vrstvě,
4. integrační vrstva dále přepošle požadavek o vytvoření nového uživatele systémům `CRM` a `AIS`,
5. po úspěšném provedení operací nad oběmi službami je klientu vráceno potvrzení s obohacenými uživatelskými daty o umístění uživatele v obou systémech (tyto `URI` adresy konkrétního uživatele slouží pouze pro reprezentační účely).

Dalším krokem je přiřazení pracovní pozice nově vytvořenému uživateli. Zde je fáze kdy lze uměle vyprodukovat chybu. To spočívá ve faktu nekonzistentních záznamů o pracovních pozicích v systémech `CRM` a `AIS`. V `CRM` systému lze uživateli přiřadit všechny pozice, které jsou uvedeny ve formuláři, naproti tomu v `AIS` systému není vedena pozice „Marketing Manager“.

Následující akce jsou tedy rozděleny na úspěšný a neúspěšný (chybou zakončený) průběh.

3.4.1 Úspěšný průběh

V úspěšném scénaři je vybrána pozice, která je obsažena v obou cílových systémech. To je zobrazeno na obrázku B.2. Po potvrzení formuláře dojde k následujícímu průchodu akcí:

1. od klienta je předán požadavek na přiřazení pozice uživali do integrační vrstvy, ta požadavek přepošle na `CRM` a `AIS` systémy,
2. po úspěšném vykonání operace službami, integrační vrstva z konfiguračního nastavení zjistí, že se jedná o operaci ukončující transakci,
3. integrační vrstva zašle požadavek na provedení operace `commit` transakčnímu koordinátoru na danou transakci,

4. transakční koordinátor proveden dvou-fázové potvrzení nad systémy `CRM` a `AIS`.

V případě, že se během tohoto procesu nevyskytne chyba, je klientu vrácen úspěch a je zobrazen přehled nově zaregistrovaného uživatele. To je zachyceno na obrázku B.3.

3.4.2 Neúspěšný průběh

Neúspěšný scénař, zachycený na obrázku B.4, je zakončen chybou značící nenalezení požadované pozice. Zde při předávání požadavku o přiřazení pozice uživateli je od `AIS` vrácena integrační vrstvě odpověď se stavem `404 Not Found`. Integrační vrstva poté zašle požadavek na odvolání transakce transakčnímu koordinátoru a klientu je vrácen obsah chyby, který je následně zobrazen. To zachycuje obrázek B.5.

Mezi tím transakční koordinátor provede odvolání provedených změn v probíhající transakci pomocí mechanismu `terminatora` u každého účastníka transakce.

3.5 Zhodnocení ověření implementací

Zvolené řešení s protokolem `TTC` 2.3 a zavedenými úpravami vedlo ke splnění většiny požadavků na zavedení transakčního zpracování v `REST`. Na rozdíl od ostatních uvedených řešení je splněn požadavek na izolaci klienta, a také po zavedení úprav, je splněn požadavek na izolaci provedených změn v rámci transakce.

Požadavek na komunikační zátěž je zde splněn částečně, samotné transakční zpracování nemá za příčinu strmý nárůst potřebné komunikace. Zde hlavními faktory, které toto ovlivňují, jsou použití integrační vrstvy jakožto prostředníka v komunikaci a použití `2PC` protokolu. To ale přináší vyšší možnosti na řízení komunikace a různá možná rozšíření, a dále zajišťuje lepší bezpečnost pro zajištění konzistence a atomicity provedených změn.

Požadavek izolace, v rámci `ACID` požadavků na transakci, je zde zajištěn vhodnou úpravou zůčastněných služeb za použití oddělených uložišť. Ty navíc musí podporovat mechanismus `terminator` a musí vědět o probíhající transakci. To porušuje požavek na bezestavovost, s čímž bylo ve výběru metody počítáno s faktem, že se jedná o slabší požadavek oproti požadavku na izolaci provedených změn. Toto dále sebou dále nese problém s ukládáním provedených změn v rámci více probíhajících transakcí, kde se mohou následně objevovat synchronizační chyby[9]. To je zde vyřešeno pomocí uzamčení služby pro modifikující operace v rámci jedné transakce. To sebou však pro vyšší hodnoty `timeout` může mít negativní vliv na použitelnost služby a nemusí být vhodné v mnoha případech.

[9] Jedna transakce smazala uživatele, kterého druhá upravovala apod.

Nutnost upravit zúčastněné služby je zde nejkritičtějším bodem. Pro rekapitulaci, oproti standardním službám, jsou zúčastněné systémy rozšířeny o následující prvky:

- pomocné úložiště sloužící k práci v rámci transakce,
- `terminator` sloužící k odvolání nebo potvrzení provedených změn,
- mechanismy, které umožňují pracovat s informací o transakci. To prakticky znamená, že služba před samotným provedením požadované operace musí rozhodnout, zda se jedná o zpracování v rámci transakce nebo mimo ní.

3.5.1 Možnosti úpravy a možná řešení problémů

Některé výše zmíněné problémy, které se objevily během ověřování, lze dále řešit. Pro úplnost jsou zde uvedeny některé možnosti řešeni:

- Pokud budeme chtít minimalizovat komunikační zátěž, lze zvolit použití jedno-fázového potvrzení, to sebou však nese vyšší riziko výskytu jednoho z heuristických případů. Další možností jak částečně snížit komunikační zátěž a rozšířit použitelnost řešení, je možnost konfiguračně povolit účastníky, kteří se neúčastní konečné fáze transakce (operace `rollback` a `commit`) a definování způsobu ukončení u každého účastníka. Tedy, u každého z účastníků konfigurovat zda se jedná o dvou-fázové, jedno-fázové, nebo žádné potvrzení.
- Problém nesplnění požadavku bezstavovosti lze řešit více způsoby. První možností je porušit požadavek na izolaci a připustit viditelnost provedených změn pro klienty, kteří nejsou zahrnuti v transakčním zracování. Dále lze, podobně jako v druhém řešení, konfiguračně definovat inverzní akci k provedené operaci. To ovšem neřeší problém izolace, zvyšuje komplexnost konfigurace a snižuje rozšiřitelnost. Také může nastat situace, kdy provedená akce nemá k sobě příslušnou inverzní. Například, odvolat provedenou aktualizaci uživatelských informací lze pouze pomocí další aktualizace s původními hodnotami. To však přináší další komplexnost buďto do integrační vrstvy nebo transakčního koordináro pro každou zúčastněnou službu.

Závěr

Zavedení transakčního zpracování do architektonického rozhraní REST má v určitých situacích své opodstatnění a to zejména v situacích, kdy jsou vykonávány operace, které v případě výskytu chyby mohou vést na nekonzistenci dat a chybný stav mezi více systémy. S tímto se můžeme setkam v mnoha situacích a jedním z míst, kde se tento problém řeší, jsou bankovní organizace.

Zavedení transakčního zpracování do REST lze dosáhnout více způsoby, které převážně vychází z principu služby (jedné či více složených) v roli transakčního koordinátora, obdobně jako např. v databázových systémech. Tato služba dále spravuje transakci a její průběh. V této práci jsou uvedeny tři často zmiňované způsoby, jak takovou službu specifikovat. Každé z uvedených řešení má své výhody a nevýhody, které převážně vychází z kladených požadavků na transakční zpracování v REST a samotných omezení na použité technologie.

V rámci ověření bylo naimplementováno a testováno řešení využívající systém, složeného z transakčního koordinátora a řídící integrační vrstvy, s použitím TCC protokolu. Řešení bylo ověřováno vůči příkladu, kde vystupuje registrační klient, jenž ovlivňuje dvě separátní služby v rámci transakce.

Po ověření vybraného způsobu lze říci, že lze dosáhnout zavedení transakčního zpracování do služeb využívajících REST vhodným přizpůsobením v transakci zúčastněných služeb. Ověření je však provedeno na modelovém příkladu a všechny vystupující služby byly pro toto vhodně upraveny. Pro reálné případy nemusí být vhodné ani možné upravit každou ze zúčastněných služeb a je nutné dále rozšiřovat navržené řešení, také se zde objevují některé problémy.

Hlavními problémy, se kterými se setkáme při řešení tohoto zadání, je požadavek na bezstavovost REST služeb a požadavek na izolaci provedených změn uvnitř transakce. Tyto požadavky se navzájem kříží a pro zajištění požadavku izolace je nutné zavést „stavovost“ do služeb účastnících se transakce a upravit je tak, aby uměli zpracovávat požadavky související s probíhající transakcí. To v praxi znamená, že každá služba by měla mít definovaný mechanismus pro provedení operací `commit` a `rollback`. Tím požadavek na izolaci přechází

na samotnou zúčastněnou službu.

Další možností, jak předejít porušení požadavku na bezstavovost, je připustit porušení požadavku na izolaci provedených změn a odstínit vědomí zúčastněných služeb o probíhající transakci (to je i jedním z požadavků pro zavedení transakčnosti). V takovém případě je ale potřeba mít zajištěnou inverzní operaci ke každé operaci provedené v rámci transakce, tedy, mít možnost jak odvolat provedené změny od započetí transakce. To s sebou nese nutnost úprav a vyšší komplexnost konfigurace transakčního koordinátora pro každou novou službu, která je přidána do transakčního zpracování.

Dále je zde zanesen problém nárůstu množství provedené komunikace. To ovlivňuje celkovou reakční dobu celého zpracování a zvyšuje možnost výskytu chyby a vyšší latenci systému. Vyvstávají zde i otázky, jak se jednotlivé služby mají chovat v případě výskytu některého z heuristikých případů a podobně, které nejsou v jednotlivých řešení zodpovězeny a povaha jejich řešení závisí na konkrétní situaci a povaze služeb. Tím dochází k zúžení použitelnosti zavedeného řešení.

Závěrem této práce tedy je, že implementace transakčního zpracování do architektonického rozhraní `REST` lze provést za cenu nutných úprav služeb, které chceme využívat v transakčním zpracování, nebo připuštění porušení požadavku na izolaci nebo požadavku na bezstavovost dle priorit v konkrétních situacích použití. Zde možnou cestou je konfiguračně rozšířit ověřované řešení tak, aby připustilo více variant, jak bude zacházet s transakcí a v ní vystupujími službami anebo zvolit některou jinou technologii vhodnější pro řešení tohoto problému.

Alternativním přístupem k tomuto problému může dále být připuštění principů `BASE` s eventuální konzistencí [49]. To může být vhodné pro celou paletu případů a může se jevit vhodnější pro použití než zavádět transakční zpracování do `REST`.

Literatura

[1] IBM: *Introduction to message queuing [online]*. 2017, [cit. 2017-03-18]. Dostupné z: https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_9.0.0/com.ibm.mq.pro.doc/q002620_.htm

[2] Wikimedia Foundation, Inc.: *Service-oriented architecture [online]*. 2017, [cit. 2017-03-11]. Dostupné z: https://en.wikipedia.org/wiki/Service-oriented_architecture

[3] W3Schools: *XML WSDL [online]*. 2017, [cit. 2017-05-04]. Dostupné z: https://www.w3schools.com/xml/xml_wsdl.asp

[4] Oracle Corporation: *SOAP Messaging Models and Examples [online]*. 2017, [cit. 2017-05-04]. Dostupné z: https://docs.oracle.com/cd/E19340-01/820-6767/aeqfx/index.html

[5] Microsoft: *What Is Windows Communication Foundation [online]*. 2017, [cit. 2017-05-04]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms731082(v=vs.110).aspx

[6] Roy Thomas Fielding: *Deriving REST [online]*. 2000, [cit. 2017-03-11]. Dostupné z: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[7] Richardson, L.; Amundsen, M.: *Detection of Interictal Epileptiform Discharges Using Signal Envelope Distribution Modelling: Application to Epileptic and Non-Epileptic Intracranial Recordings. Spike Detector Algorithm*. O'Reilly Media, 2013, ISBN 978-1-449-35806-8.

[8] RESTfulAPI.net: *What is REST API [online]*. 2017, [cit. 2017-03-11]. Dostupné z: http://restfulapi.net/

[9] IBM Corporation: *RESTful Web services: The basics [online]*. 2015, [cit. 2017-03-11]. Dostupné z: https://www.ibm.com/developerworks/webservices/library/ws-restful/

[10] W3C: *Web Services Message Exchange Patterns [online]*. 2002, [cit. 2017-03-11]. Dostupné z: https://www.w3.org/2002/ws/cg/2/07/meps.html

[11] Rob Daigneau: *Request/Response [online]*. 2011, [cit. 2017-03-11]. Dostupné z: http://www.servicedesignpatterns.com/ClientServiceInteractions/RequestResponse

[12] W3C: *Simple Object Access Protocol (SOAP) 1.1 [online]*. 2000, [cit. 2017-03-11]. Dostupné z: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[13] W3C: *Simple Object Access Protocol (SOAP) 1.1 [online]*. 2007, [cit. 2017-03-11]. Dostupné z: https://www.w3.org/TR/soap/

[14] TechTarget: *EAI (enterprise application integration) [online]*. 2007, [cit. 2017-03-12]. Dostupné z: http://searchmicroservices.techtarget.com/definition/EAI-enterprise-application-integration

[15] TechTarget: *Object Request Broker (ORB) [online]*. 2005, [cit. 2017-03-12]. Dostupné z: http://searchmicroservices.techtarget.com/definition/Object-Request-Broker-ORB

[16] OlivierRey: *Transaction Processing Monitor [online]*. 2010, [cit. 2017-03-12]. Dostupné z: http://wiki.c2.com/?TransactionProcessingMonitor

[17] Defining Technology, Inc.: *What is Middleware? [online]*. 2008, [cit. 2017-03-12]. Dostupné z: http://web.archive.org/web/20120629211518/http://www.middleware.org/whatis.html

[18] Wikimedia Foundation, Inc.: *EAI (enterprise application integration) [online]*. 2017, [cit. 2017-03-13]. Dostupné z: https://en.wikipedia.org/wiki/Transaction_processing

[19] Microsoft: *Two-Phase Commit [online]*. 2017, [cit. 2017-03-13]. Dostupné z: https://msdn.microsoft.com/en-us/library/aa754091(v=bts.10).aspx

[20] Stack Exchange Inc: *Transactions in REST? [online]*. 2009, [cit. 2017-03-18]. Dostupné z: http://stackoverflow.com/questions/147207/transactions-in-rest

[21] Stack Exchange Inc: *REST and transaction rollbacks [online]*. 2016, [cit. 2017-03-18]. Dostupné z: http://stackoverflow.com/questions/33060968/rest-and-transaction-rollbacks

[22] Středoevropské centrum pro finance a management: *Customer Relationship Management [online]*. 2012, [cit. 2017-04-01]. Dostupné z: http://www.finance-management.cz/080vypisPojmu.php?IdPojPass=58

[23] Microsoft: *Message Queuing Transactions [online]*. 2016, [cit. 2017-03-18]. Dostupné z: https://msdn.microsoft.com/en-us/library/ms699870(v=vs.85).aspx

[24] Sharpened Productions: *RTE Definition [online]*. 2017, [cit. 2017-03-19]. Dostupné z: https://techterms.com/definition/rte

[25] Oracle: *Enterprise JavaBeans Technology [online]*. 2017, [cit. 2017-03-19]. Dostupné z: http://www.oracle.com/technetwork/java/javaee/ejb/index.html

[26] tutorialspoint: *EJB - Transactions [online]*. 2017, [cit. 2017-03-19]. Dostupné z: https://www.tutorialspoint.com/ejb/ejb_transactions.htm

[27] Helland, P.: *Life beyond Distributed Transactions: an Apostateś Opinion*. Amazon.Com: Seattle, 2016. Dostupné z: http://www-db.cs.wisc.edu/cidr/cidr2007/papers/cidr07p15.pdf

[28] Microsoft: *High-Availability System Architecture [online]*. 2017, [cit. 2017-05-05]. Dostupné z: https://msdn.microsoft.com/en-us/library/cc750543.aspx

[29] Network Working Group: *Hypertext Transfer Protocol – HTTP/1.0 [online]*. 1996, [cit. 2017-03-09]. Dostupné z: https://tools.ietf.org/search/rfc1945

[30] Network Working Group: *Hypertext Transfer Protocol – HTTP/1.1 [online]*. 1999, [cit. 2017-03-09]. Dostupné z: https://tools.ietf.org/html/rfc2616

[31] Network Working Group: *The Secure HyperText Transfer Protocol [online]*. 1999, [cit. 2017-03-09]. Dostupné z: https://tools.ietf.org/html/rfc2660

[32] Network Working Group: *The JavaScript Object Notation (JSON) Data Interchange Format [online]*. 2014, [cit. 2017-03-10]. Dostupné z: https://tools.ietf.org/html/rfc7159

[33] Little, M.: *REST-Atomic Transactions*. google.com, 2011. Dostupné z: http://bit.ly/2pFx8BL

[34] Network Working Group: *Uniform Resource Identifier (URI): Generic Syntax [online]*. 2005, [cit. 2017-03-19]. Dostupné z: https://tools.ietf.org/html/rfc3986

[35] Guy Pardon, C. t.: kapitola Towards Distributed Atomic Transactions over RESTful Services.

[36] Alexandros Marinos, S. t. P. t., Amir Razavi: *RETRO: A Consistent and Recoverable RESTful Transaction Model*. IEEE, 2009. Dostupné z: http://ieeexplore.ieee.org/document/5175822/

[37] Guy Pardon, C. t.: kapitola ATOM Pub/Sub.

[38] tutorialspoint.com: *Design Patterns - Facade Pattern [online]*. 2017, [cit. 2017-04-28]. Dostupné z: https://www.tutorialspoint.com/design_pattern/facade_pattern.htm

[39] W3C: *Extensible Markup Language (XML) [online]*. 2016, [cit. 2017-04-29]. Dostupné z: https://www.w3.org/XML/

[40] Pivotal Software: *Spring Boot [online]*. 2017, [cit. 2017-04-29]. Dostupné z: https://projects.spring.io/spring-boot/

[41] Oracle Corporation: *NetBeans IDE [online]*. 2017, [cit. 2017-04-29]. Dostupné z: https://netbeans.org/

[42] The Apache Software Foundation: *Apache Maven [online]*. 2017, [cit. 2017-04-29]. Dostupné z: https://maven.apache.org/

[43] Pivotal Software: *SPRING INITIALIZR [online]*. 2017, [cit. 2017-04-29]. Dostupné z: https://start.spring.io/

[44] The Apache Software Foundation: *Apache Tomcat [online]*. 2017, [cit. 2017-04-29]. Dostupné z: http://tomcat.apache.org/

[45] Bootstrap.com: *Bootstrap [online]*. 2017, [cit. 2017-04-30]. Dostupné z: http://getbootstrap.com/

[46] The jQuery Foundation: *jQuery [online]*. 2017, [cit. 2017-04-30]. Dostupné z: https://jquery.com/

[47] Pivotal Software: *Accessing Data with JPA [online]*. 2017, [cit. 2017-05-03]. Dostupné z: https://spring.io/guides/gs/accessing-data-jpa/

[48] Oracle: *Interface Map [online]*. 2016, [cit. 2017-05-03]. Dostupné z: https://docs.oracle.com/javase/7/docs/api/java/util/Map.html

[49] DATAVERSITY Education: *ACID vs. BASE [online]*. 2012, [cit. 2017-05-05]. Dostupné z: http://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/

Příloha A

Seznam použitých zkratek

SOA Service Oriented Architecture

XML Extensible markup language

REST Representational state transfer

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IDE Vývojové prostředí

SOAP Simple Object Access Protocol

WCF Windows Communication Foundation

SMTP Simple Mail Transfer Protocol

TCP Transmission Control Protocol

UDP User Datagram Protocol

JMS Java Messaging Services

ACID Atomic, Consistent, Isolated, Durable

EJB Enterprise JavaBeans

WS Web Service

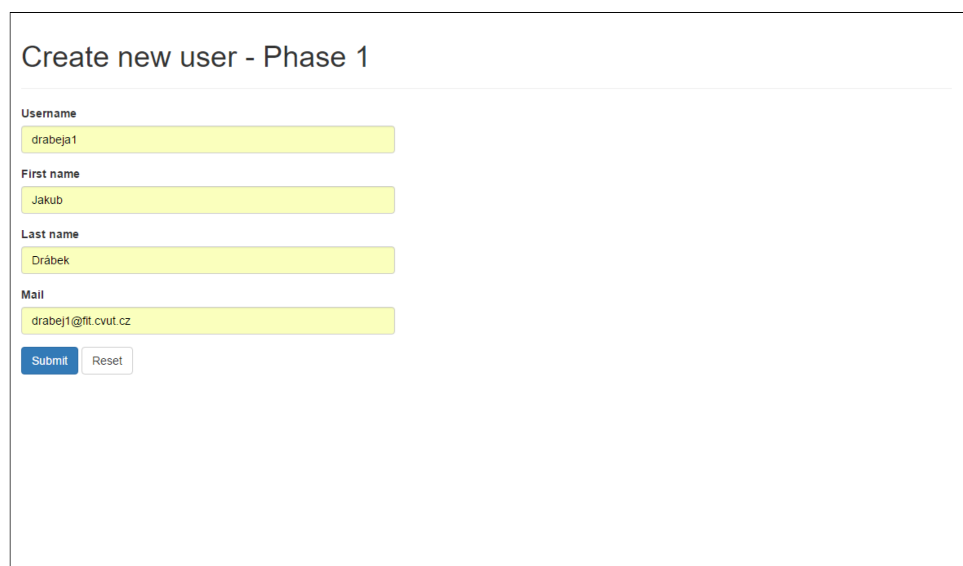
URI Uniform Resource Identifier

URL Uniform Resource Location

Příloha B

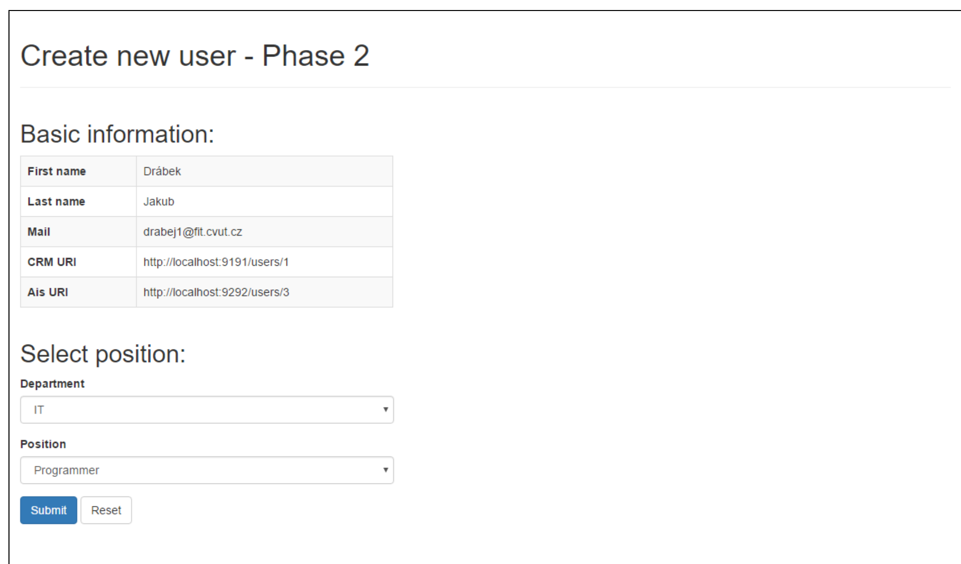
Snímky průchodu příkladu použití

- První fáze registračního formuláře:



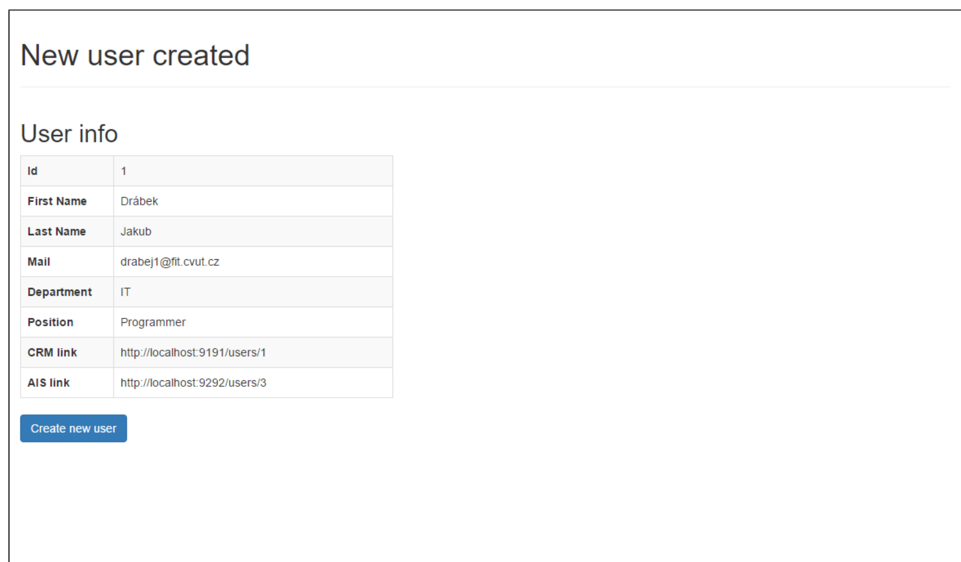
Obrázek B.1: První fáze registračního formuláře

- Druhá fáze registračního formuláře vedoucí k úspěchu:



Obrázek B.2: Druhá fáze registračního formuláře - úspěch

- Výsledek úspěšné registrace:



Obrázek B.3: Výsledek úspěšné registrace

- Druhá fáze registračního formuláře vedoucí k neúspěchu:

Create new user - Phase 2

Basic information:

First name	Drábek
Last name	Jakub
Mail	drabej1@fit.cvut.cz
CRM URI	http://localhost:9191/users/2
Ais URI	http://localhost:9292/users/4

Select position:

Department

Marketing

Position

Marketing Manager

Submit Reset

Obrázek B.4: Druhá fáze registračního formuláře - neúspěch

- Zobrazení výskytu chyby během registrace:

Error while processing request

Obtained error:
Obtained status code 404 : Could not find position 'Marketing Manager'.

Bact to new user form

Obrázek B.5: Zobrazení výskytu chyby

Příloha C

Obsah přiloženého CD

```
exe ............................ adresář s .bat soubory pro spuštění služeb
javadoc ........................ vygenerovaná dokumentace ke zdrojovým kódům
    dp_ais_apidocs ............. Javadoc implementace služby účetního systému
    dp_client_apidocs .......... Javadoc implementace klienta
    dp_crm_apidocs ............. Javadoc implementace služby CRM systemy
    dp_integration_apidocs ..... Javadoc implementace integrační vrstvy
    dp_trcoordinator_apidocs ... Javadoc implementace transakčního koordinátora
src
  impl ......................... zdrojové kódy implementace
    dp_ais ..................... implementace služby účetního systému
    dp_client .................. implementace klienta
    dp_crm ..................... implementace služby CRM systému
    dp_integration ............. implementace služby představující integrační vrstvu
    dp_trcoordinator ........... implementace služby transakčního koordinátora
  thesis ....................... zdrojová forma práce ve formátu LaTeX
text ........................... text práce
  DP_Jakub_Drábek_2017.pdf ..... text práce ve formátu PDF
  DP_Jakub_Drábek_2017.ps ...... text práce ve formátu PS
readme.txt ..................... stručný popis obsahu CD
```