Systémová integrace

Systémová integrace (SI) je pojem, který se v praxi používá ve dvou významech:

1. V užším technickém pojetí se jedná o integraci různých technických částí informačního systému, tedy aplikací (systémů) do jednoho celku. Cílem je taková architektura informačního systému jakožto celku, která efektivně podporuje business procesy v organizaci.
2. V širším významu pojem systémová integrace označuje celý proces, který je nezbytný pro efektivní fungování zejména rozsáhlejších informačních systémů

Důvody integrace

Správně navržené řešení obvykle představuje:

- Zjednodušení stávající architektury integrace (snížení počtu rozhraní mezi systémy)
- Nižší náklady na modifikaci stávajících systémů a aplikací
- Nižší náklady na implementaci a integraci nových systémů a aplikací
- Větší automatizaci business procesů, což přináší nižší náklady a větší rychlost zpracování
- Možnost snadnější integrace se systémy externích subjektů

Typy integrace

Z pohledu architektury IS lze integraci realizovat na několika úrovních:

- Integrace na úrovni uživatelského rozhraní (prezentační vrstvy)
- **Integrace na úrovni aplikační vrstvy**
- Integrace na úrovni datové (perzistentní) vrstvy
- Integrace mezi rozdílnými vrstvami architektury IS

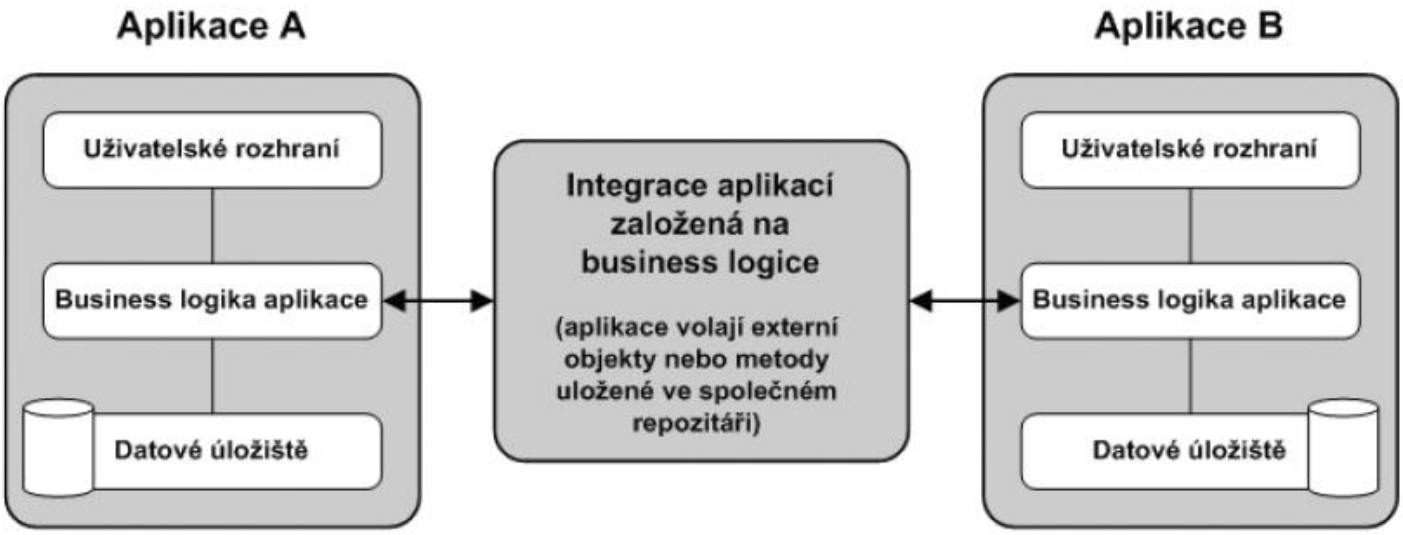
Integrační styly - představují obecné přístupy k integraci informačních systémů či aplikací, resp. představují možné způsoby komunikace mezi jednotlivými systémy. Z tohoto pohledu existuje několik základních přístupů:

- **Integrace na aplikační vrstvě:**

1. Vzdálené volání procedur
    - RPC, RMI, REST, webservice, atd.
2. Zasílání zpráv
    - ESB, sběrnice, brokery

Integrace na aplikační vrstvě

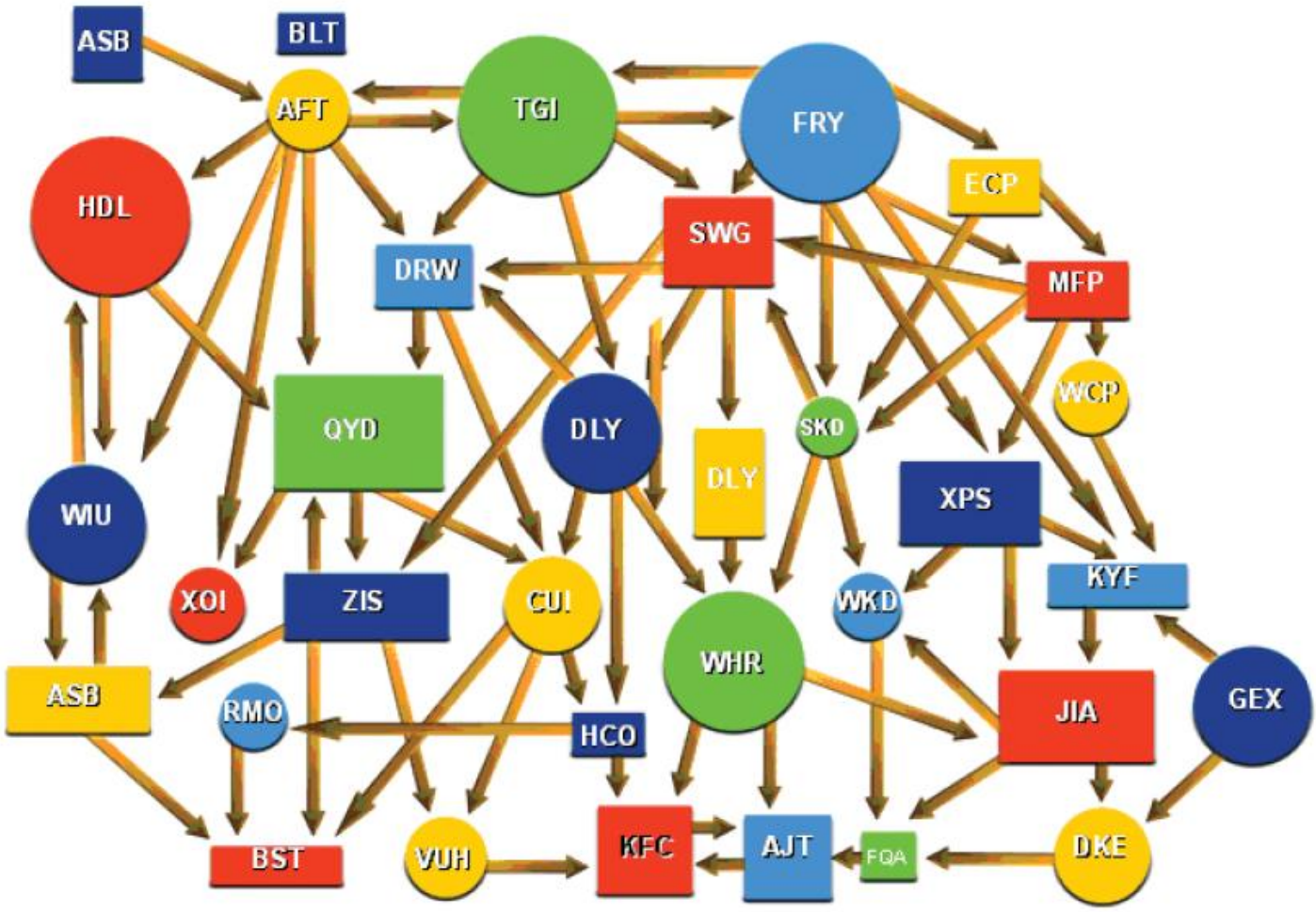
Střední cestou, která ovšem v tomto případě není zlatá, je integrace na bázi komponent aplikační vrstvy. Spočívá v integraci prostřednictvím vzájemného volání mezi objekty (komponentami) ve střední vrstvě aplikací (viz obrázek 1), jde v podstatě o „drátování“ jednotlivých aplikací mezi sebou.



Obrázek 1: Schéma integrace na úrovni obchodní logiky

Ačkoliv tento přístup na první pohled vypadá poměrně koncepčně, je jeho praktická realizace z mnoha důvodů velmi obtížná:

- Vyžaduje si zásah do aplikací, což s sebou často přináší nastudování velkého množství dokumentace a prozkoumávání bílých míst, složitou koordinaci s autorem či dodavatelem aplikace, problémy s podporou vzniklého řešení a další.
- Vzájemná závislost a těsná vazba aplikací. Vzhledem k tomu, že dochází ke vzájemnému volání aplikací, stává se dostupnost a správná funkce volající aplikace do velké míry závislou na všech volaných aplikacích – tento nedostatek lze řešit používáním asynchronního volání, což je ale velmi pracné a často pouze jednoúčelové a komplexní řešení, které musí být podporováno řadou servisních aplikací, které celý systém komplikují. Vzájemná závislost a těsná vazba jsou též brzdou aktualizace aplikací – změna v aplikaci si vyžaduje analýzu dopadu změn v ostatních aplikacích, implementaci změn a znovuotestování všech závislých aplikací
- Syndrom N^2 – integrace na úrovni obchodní logiky vede k růstu složitosti s počtem možných vazeb mezi aplikacemi, přičemž tento počet roste s kvadrátem počtu integrovaných systémů (viz obrázek 2). Přidání každé další integrované aplikace je tak vždy pracnější a nákladnější než té předchozí – rozhodně ne dobrá strategická vyhlídka.
- Heterogenita – ačkoliv by si to všichni uživatelé přáli, nikdy nedošlo k většinovému používání jediné objektové a komponentové technologie. Ani DCOM/COM+ od Microsoftu, ani komponentové technologie z rodiny J2EE, ani žádná nastupující a „tentokrát už zaručeně ta pravá“ technologie nedosáhly a pravděpodobně nikdy nedosáhnou nadkritického rozšíření. Proto je heterogenita technologií střední vrstvy všudypřítomnou a obtížně překonatelnou realitou – ještě obtížněji než se může na první pohled zdát. Existují sice různé nástroje pro překonání technologických rozdílů (tzv. *bridge*), ale tyto zpravidla nesplňují očekávání zákazníků a jsou navíc často produkovány malými firmami s nejasnou podporou i budoucností. Nejambicióznějším krokem na tomto poli byl pokus o zavedení standardu CORBA. Přestože projekt zaznamenal dílčí úspěchy, dny jeho slávy jsou sečteny z obvyklých důvodů – nadměrná složitost, vysoká pořizovací cena i implementační náklady, nekompatibilita a nedostatečná podpora klíčových výrobců softwaru.



Obrázek 2: Syndrom N^2 při integraci aplikací

SOA (Service Oriented Architecture)

Service Oriented Architecture je sada principů a metodologií, která doporučuje skládat složité aplikace a jiné systémy ze skupiny na sobě nezávislých komponent poskytujících služby. Zkratka SOA znamená v překladu z angličtiny architekturu orientovanou na služby.

Služba

Službou zde myslíme něco, co poskytne přidanou hodnotu, v rámci firmy tedy jde o jasně definovanou podnikovou činnost jako například objednání zboží od dodavatele. V rámci aplikace to může být dílčí úkol, který je nutno splnit při obsluze uživatele. Jako příklad může posloužit aplikační mini modul, jenž zaokrouhluje čísla zobrazující se uživateli jako výsledek výpočtu. Při správném návrhu aplikace orientované na služby lze poté tuto komponentu odpojit a zapojit jinou, která bude zaokrouhlovat odlišně, aniž bychom nějak zasahovali do zbytku aplikace. Jednotlivé komponenty lze zpravidla různě kombinovat, doplňovat, případně nahrazovat jednu za druhou. Tímto přístupem se prvky v systému stávají samostatnými, systém je tedy stabilnější a také je zde výhodné rozložení zátěže na více komponent. Při výpadku jedné komponenty může být nahrazena jinou, funkční. Systém se pak lépe spravuje (při poruše komponenty lze jednoznačně určit, kde se vyskytla chyba), ale také vylepšuje, jelikož je potřeba zasahovat jen do určité komponenty a ne do celého systému. Při dodržení principů SOA ve fázi návrhu lze výsledný systém přirovnat ke známé stavebnici Lego, jejíž součásti představují jednotlivé komponenty.

Webová služba

Webové služby umožňují jednoduchou komunikaci mezi aplikacemi ve velmi heterogenním prostředí, díky komunikaci založené na platformě nezávislých standardech. Komunikace mezi službami probíhá přes SOAP protokol postavený na XML, který pro síťovou komunikaci využívá další protokoly aplikační síťové vrstvy. Nejčastěji HTTP, který bývá povolen na většině firewallů a nasazení je tak méně problematické. Aplikace si mezi sebou posílají XML zprávy, které přenášejí dotazy a odpovědi jednotlivých aplikací. Komunikační rozhraní webových služeb je popsáno jazykem WSDL.

Princip

Různí autoři mluví o lehce odlišných principech, všechny tyto principy jsou spolu ale konzistentní, jedná se tedy o popis dané situaci více způsoby. Dle http://www.soaprinciples.com existuje 8 hlavních principů.

Standardizovaný kontrakt služby

Servisní kontrakt poskytované služby je jasně definovaný. Proto je již při návrhu SOA komponenty třeba zvážit technické rozhraní a formát dat, který bude služba přijímat/odesílat. Kontrakt je brán jako slib dané služby svému okolí, co bude poskytovat.

Slabé vazby mezi komponentami

Vazby mezi jednotlivými komponentami při návrhu SOA mají být co nejtenčí. Tento princip spočívá v tom, že jsou závislosti dodefinovány těsně před použitím a navíc externě, tedy mimo vyvinutou službu, čímž se redukuje závislost kontraktu služby, její implementační logiky a konzumentů.

Princip abstrakce

Jeho úkolem je co nejvíce skrýt implementační detaily služby/komponenty. Tento princip hraje primární roli také v poskládání složitých systémů a také poskytuje podporu pro předchozí princip volných vazeb. Dobře použitý princip abstrakce zlepšuje granularitu systému a hlavně dovoluje a velmi usnadňuje správu a případné pozdější úpravy systému.

Znovupoužití

Při návrhu jakékoliv služby či komponenty by se mělo počítat s jejím využitím i jinde, než pro aktuální projekt, čímž se maximalizuje použitelnost vyvíjené části. Pro zajištění tohoto principu je třeba dbát na dobrý architektonický návrh systému.

Nezávislost

Aby mohla služba zajistit správné fungování, tedy dodržení definovaného kontraktu, musí v sobě obsahovat vnitřní nezávislou logiku pro správu zdrojů, ze kterých čerpá. Tento princip také vyžaduje dodržení určitých pravidel při návrhu, aby bylo možné správné zasazení komponenty do daného implementačního prostředí. V případě SOA se zde mluví o izolaci, normalizaci služeb. Zajištění nezávislost komponenty/služby je důležité pro umožnění principu znovupoužití.

Bezstavovost

Princip bezstavovosti se začal promítat do návrhů systémů s rozvojem podnikových systémů, které obsluhují velký počet uživatelů. Bezstavovost v tomto případě přímo umožňuje znovupoužití, protože komponenty, které si nepamatují stav lze bez jakýchkoliv inicializačních požadavků okamžitě využít i jinde. Tento princip zajišťuje také větší možnosti škálovatelnosti systému v budoucnu.

Princip identifikovatelnosti

Tento princip má spíše své ekonomické opodstatnění. Služba, která je lehce identifikovatelná, jakožto je lehce zjistitelný i způsob jejího použití, má obrovskou výhodu na trhu. Takovéto služby přinášejí větší zisk než služby mnohem kvalitnější, ale málo využívané díky jejich nepřístupnosti široké veřejnosti. Identifikovatelnost v SOA je zajištěna WSDL jazykem.

Princip skládání

V SOA je tímto principem zajištěno možné seskládání jednotlivých služeb a zajištění jejich vzájemné synergie. Kromě již zmiňovaných výhod postupného složení nezávislých komponent/služeb, jde navíc ruku v ruce s odvěkým pravidlem pro řešení komplexních problémů. A sice jejich rozložení na malé, lehce řešitelné podproblémy. Jednotlivá řešení jsou potom seskládána do výsledného systému tak, aby dokázaly zvládnout komplexní problém.

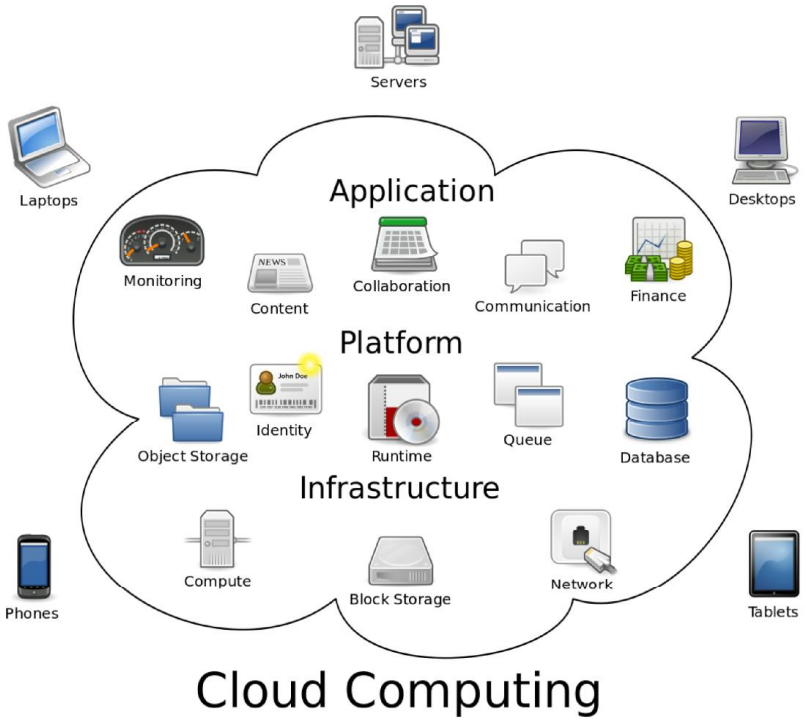
Princip orientace na služby a použití v různých podmínkách

Princip, který by zde neměl chybět a který zároveň zastřešuje všechny již zmiňované, takže je uveden na devátém místě, přestože bylo v nadpisu avizováno 8 principů. Když se nad tím zamyslíme, zjistíme, že každý z principů z předchozí kapitoly napomáhá k dodržení tohoto posledního, zastřešujících principu. Například definovaný servisní kontrakt zajišťuje znovupoužití služby v různých podmínkách, princip slabých vazeb podporuje orientaci na služby a zajišťuje možnost jejich různých kombinací, které si lze vybrat až v momentě použití služeb atd.

Možnosti realizace (implementace)

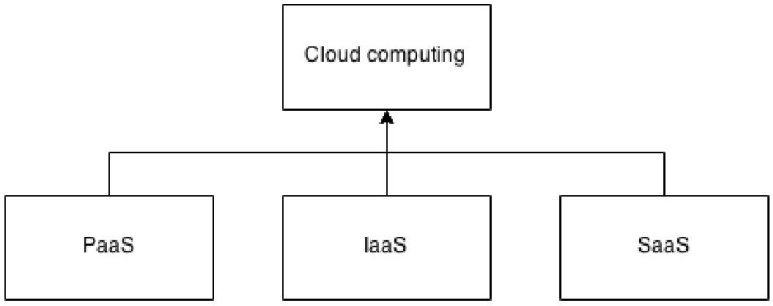
Pokud chceme mluvit o realizaci integrace na aplikační vrstvě, bude vhodné zkráceně představit "hype dnešní doby" a tím je

Cloud computing, je to model využívání služeb / platformy / softwaru pomocí internetu. V tomto modelu většinou zákazník neplatí za software, ale za jeho využívání nebo využívání systémových zdrojů (ať výpočetního výkonu nebo místa).



Cloud computingové služby

Cloud computingové služby můžeme dělit podle několika kritérií, ať je to podle modelu nasazení (veřejné, privátní, komunitní), tak (pro nás více zajímavé) podle distribučního modelu.



PaaS

- Platform as a Service - poskytování platformy pro vývoj webových aplikací. Nejvýznamějším hráčem je Google App Engine, na kterém si tento pojem lze snadno vysvětlit. Google App Engine poskytuje zákazníkům možnost vytvářet webové aplikace, aniž by se museli starat o běhovou platformu. Veškeré zákazníkem nahrané aplikace běží v tzv. Sandboxu napříč několika fyzickými / virtuálními stroji a jsou škálovány on-demand (čím víc lidí a větší zátěž, tím víc strojů). U tohoto typu musíme však počítat s tím, že se vždy jedná o uzavřenou platformu. Google App Engine v dnešní době nabízí podporu pouze několika jazyků - Python, Java, JRuby (Ruby), Go, Scala, Clojure a experimentálně PHP.

IaaS

- Infrastructure as a Service - Dalo by se označit jako *pronájem hardwaru*, což ve výsledku znamená, že platíme někomu, že se nám bude o hardware starat (typicky je to virtualizace - VPS apod). Příkladem může být Amazon WS nebo Windows Azure.

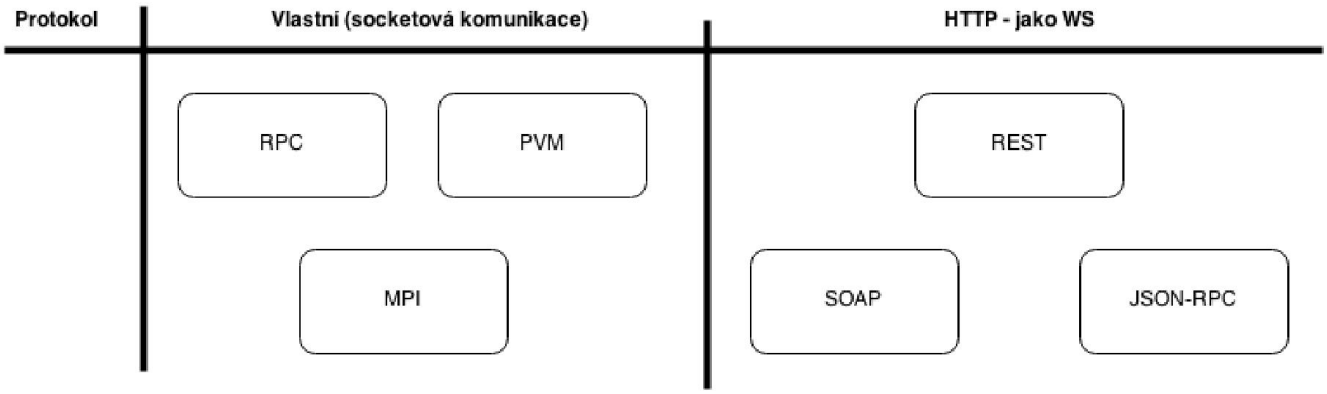
SaaS

- Software as a Service - tzv. *pronájem aplikací*, to znamená, že uživatel si nekupuje krabici s programem (nebo cd) jako

takovou, ale platí si pronájem. Nejvýznamnějším příkladem může být Adobe Cloud (Photoshop a ostatní produkty od Adobe) nebo Google Apps.

Vzdálené volání procedur

Veškerá vzdálená volání probíhají většinou stejně, odlišují se malými detaily, na dalším obrázku naleznete rozdělení:



Zakládním rozdělením by mohla být závislost na nižších vrstvách, zde toto uvádíme jako používání vlastního protokolu. REST a jemu podobné využívají HTTP protokol pro komunikaci se vzdálenými servery, zatímco RPC, PVM, MPI mají vlastní komunikaci.

RPC

Remote procedure call, umožňuje volat výkonný kód umístěny v jiném místě / na jiném stroji (musíme být k němu připojeni). Princip je vcelku jednoduchý:

- Nejprve proběhne jednoduché zabalení parametrů a identifikátorů procedury do formy vhodné pro přenos mezi počítači (tzv. marshalling).
- Poté se balíček odešle.
- Balíček se na vzdáleném místě rozbalí, zjistí se o jakou proceduru jde (unmarshalling).
- Samotné zavolání a provedení procedury.
- Výsledek procedury se opět zabalí.
- A odešle zpět.
- První počítač výsledek opětovně rozbalí.
- A přijatá hodnota se předá proceduře.

Problém je zde, že vzdálený server musí být neustále připojen a být funkční, jakákoliv chyba není zotavitelná!! Tento problém lze řešit lépe pomocí PVM, MPI nebo webových služeb - REST / JSON RPC.

RMI

Java remote method invocation (**Java RMI**) je technologie programovacího jazyka Java umožňující z jednoho virtuálního stroje (JVM) volat metody objektů na jiném virtuálním stroji, který obvykle běží na jiném počítači (HOSTU). Výhodou RMI je, že programátor zachazí se vzdáleným objektem, jako by byl místní. Rozhraní a třídy, které jsou zodpovědné za funkčnost RMI jsou nadefinovány v balíčku java.rmi.

- založeno na architektuře klient – server
- zjednodušuje komunikaci se vzdálenými aplikacemi na úroveň lokálního volání metod
- podpora pro zabezpečení klienta, serveru i komunikace

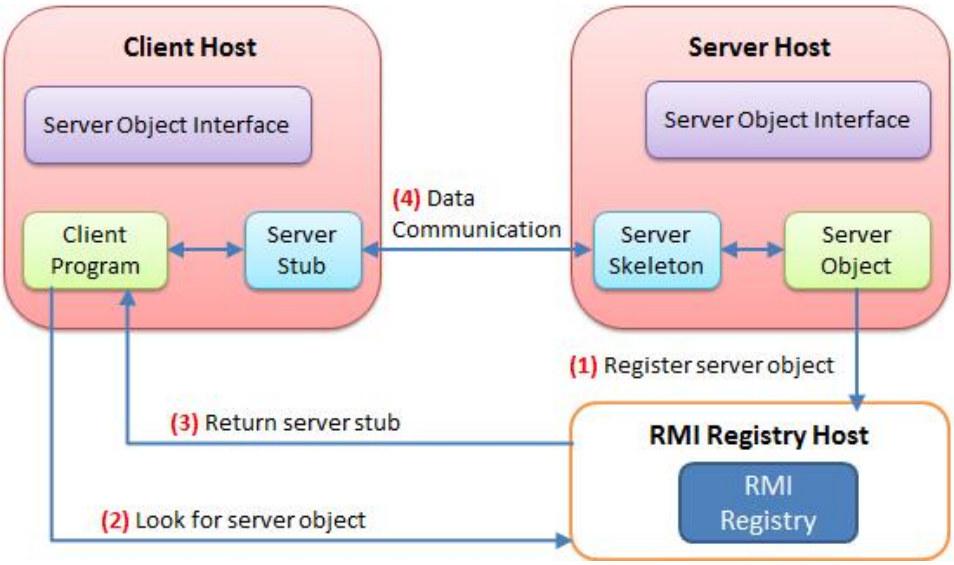
Stub

- stará se o zpracování požadavku klienta ve formě marshall streamu (tzv. marshalling)
- v případě zaslání návratové hodnoty od serveru zpracuje zaslaný marshall stream (unmarshalling)

Skeleton

- protějšek stubu na straně serveru

- rozbaluje marshall streamy do podoby volání metod



SOAP

Simple Object Access Protocol - technologie vyvinutá za podpory Microsoftu (nástupce RPC-XML), nyní ji vlastní W3C konsorcium. Ve výsledku, jenom upravili posílání... nyní využívají HTTP protokol. Jak již název napovídá zprávy jsou formátovány pomocí XML, každá zpráva obsahuje kořenovou část envelope, v ní zanořené části header a body, které obsahují konkrétní informace co chceme. Formát XML byl vybrán díky rozšiřitelnosti a dostupnosti jeho dekódování. To sebou nese i jisté nevýhody, příkladem může být velikost datových zpráv (mezi servery na tom nezáleží, ale do telefonu to nikdo stahovat nechce) nebo složitost parsování zpráv (i když dnes již jsou PC daleko výkonější).

JSON-RPC 2.0 (novější, využívající client-server architekturu)

Json-RPC 2.0 je principiálně stejný jako SOAP, jediným rozdílem je, že místo XML využívá formátu JSON (Javascript Object Notation) a tím i odlišný formát zprávy, která musí obsahovat jméno metody, její parametry, unikátní id zprávy a verzi rpc (v tomto případě 2.0). Ukázka:

```
--> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
<-- {"jsonrpc": "2.0", "result": 19, "id": 1}
```

REST

Representational State Transfer - Byl přestaven roku 2000 a byl vyvíjen společně s protokolem HTTP, na kterém je taky založen. Rest je narozdíl od ostatních zde uvedených orientován datově a ne procedurálně! Každý datový zdroj má přesně definovanou URI a REST definuje 4 metody jak s ním nakládat. Mějmě příkladem kolekci CD:

- **GET** /cd_collection/ - získání seznamu kolekce CD
- **POST** /cd_collection/ - vytvoření seznamu kolekce CD
- **PUT** /cd_collection/ - vyměnění celé kolekce za jinou
- **DELETE** /cd_collection/ - smazání celé kolekce

REST je bezstavový, což v důsledku znamená, že každý požadavek musí obsahovat veškerá data k jeho provedení. Dále také nedefinuje jednoznačně formát výměny dat, mezi nejpoužívanější patří XML/JSON, ale můžeme použít i třeba grafické materiály.

Příklad integrace

Pro představu, jako příklad konkrétní integrace systémů na aplikační vrstvě s využitím výše popsaných technologií a možností realizace, lze uvést následující proces. Ve firmě, která například vyrábí na zakázku nějaké součástky, je běžným obchodním procesem objednávka, přes výrobu až po dodání s fakturou. Do tohoto procesu zasahuje více oddělení a každé používá odlišný software. Na začátku je zpracována objednávka v IS (informační systém), odkud dále získají informace pro vývoj a výrobu. V poslední fázi, když je produkt vyrobený, jsou předány další informace k vyfakturování a dodání zboží zákazníkovi.

Znázornění procesu před integrací, kde každá část využívá jiný software s vlastní databází a informace jsou předávány ručně s tím, že z výroby si požadované informace získají z IS. Po dokončení vývoje a výroby je potřeba informovat účetní oddělení, které potřebuje informace z výroby (např. data z vývoje) a další informace z objednávky z IS.

Pro optimalizaci procesu by bylo vhodné využít komplexní software typu PLM (Product Lifecycle Management), který by zahrnoval všechny potřebné moduly nebo provést integrace dle Obrázku 2 – po integraci systémů. Integrací na aplikační vrstvě se zautomatizují ruční procesy předávání informací mezi systémy, zajistí se zpětná aktualizace/synchronizace dat v případě změn a celý proces se zefektivní.

Popis jednotlivých procedur zajišťující integraci systémů:

- **NewOrder / SynchroOrder –** Při vytvoření objednávky v IS se automaticky založí nový obchodní případ v systému PDM pro vývoj a výrobu. Pokud dojde ke změnám, služba zajistí aktualizaci těchto změn.
- **CreateInvoice –** Při vytvoření objednávky je automaticky připravena faktura pro doplnění po dokončení zakázky. Tato procedura není při integraci nutná a lze fakturu vytvářet až po dokončení celé zakázky. Záleží na informacích potřebných z IS a PDM.
- **OrderCompleted –** Zakázka je dokončena a může se vytvořit faktura a připravit zboží k dodání zákazníkovi.

Prezentace: **8_integrace_na_aplikacni_vrstve.pdf**Zobrazit podrobnosti