Универзитет у Новом Саду
Технички факултет "Михајло Пупин", Зрењанин

Проф. др Жељко Стојанов

Дистрибуирани софтверски системи

Скрипта за предавања

Зрењанин
- 2018 -

Садржај

Листа слика

Листа табела

Листа листинга

Листа коришћених скраћеница

Предговор

Текст скрипте је припремљен као основни материјал за теоријски део испита студентима који слушају предмет *Дистрибуирани софтверски системи* на студијском прогаму *Информационе технологије - Софтверско инжењерство* који се реализује на Техничком факултету "Михајло Пупин" у Зрењанину, Универзитета у Новом Саду.

Прво поглавље представља основне концепте дистрибуираних система. Друго поглавље уводи основне концепте инжењеринга дистрибуираних софтверских система. Након тога су у трећем поглављу приказани основни принципи софтверског инжењерства базираног на компонентама, док четврто поглавље приказује сервисно оријентисане архитектуре са фокусом на Веб сервисе.

Упркос вишегодишњем раду у реализацији наставе и пројеката софтверских решења, као и припреми материјала за студенте који је послужио као основа за ову скрипту, аутор је свестан да постоје пропусти и техничке грешке. Аутор ће бити захвалан свакоме ко укаже на уочене грешке и недостатке и тиме допринесе квалитету наредних верзија овог текста. Све сугестије које могу унапредити квалитет овог текста су такође добродошле.

Слава Богу, сада, увек и у векове векова. Амин.

Зрењанин, 2018. лета Господњег.

Жељко Стојанов

Поглавље 1

Основни концепти

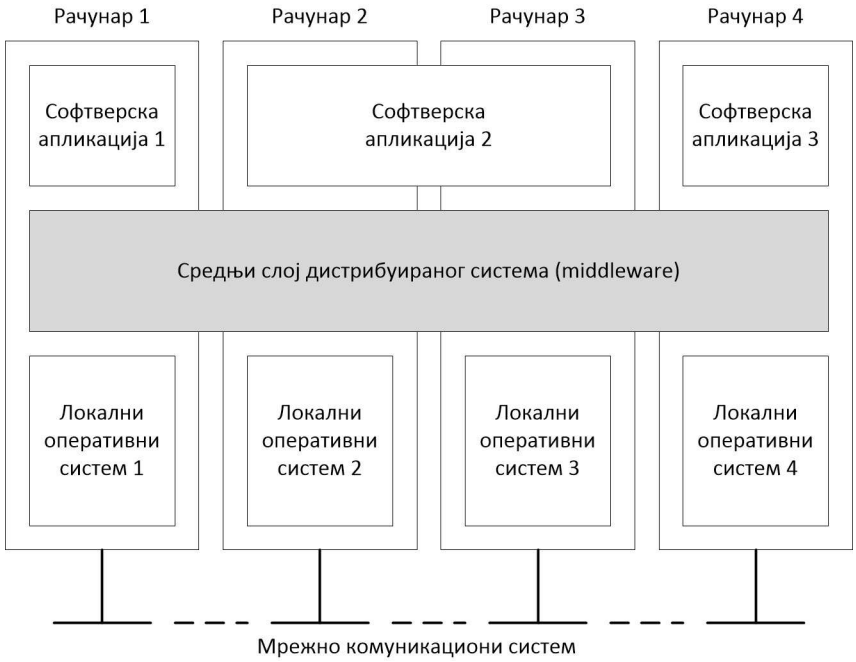
Развој дистрибуираних система је последица два основна тренда технолошког развоја у области рачунарства: (1) развој микропроцесора и (2) развој рачунарских мрежа. Ови трендови омогућују да се велики број микропроцесорски базираних рачунарских јединица повеже помоћу рачунарских мрежа и тако настају дистрибуирани рачунарски системи. У литератури се може пронаћи велики број дефиниција дистрибуираног система, али ћемо овде усвојити следећу дефиницију [Tanenbaum and Van Steen, 2007]:

> **Дистрибуирани систем** је колекција независних рачунара који се корисницима приказују као јединствен кохерентан рачунарски систем.

Основни аспект који наглашава ова дефиниција је да корисник има утисак да користи јединствен систем при чему је од њега скривена архитектура система и начин повезивања рачунара. Следећа важна карактеристика дистрибуираних система је да корисник на исти начин користи систем без обзира преко којег рачунара му приступа. Дистрибуирани системи су у принципу континуирано доступни корисницима, што подразумева да корисник не примећује да ли се нака компонента у систему мења или поправља.

Софтверски системи су апстрактни и нематеријални. С обзиром да немају физичких ограничења, софтверски системи могу бити веома комплексни, захтевни за разумевање и врло компликовани и скупи за одржавање. Софтверски системи због своје сложености и различитих начина употребе могу имати веома комплексне поступке инсталирања и конфигурисања. Тако долазимо и до уопштене дефиниције софтверског система

> **Софтверски систем** или софтвер, је скуп повезаних рачунарских програма, подаци које ти програми обрађују, одговарајућа документација и конфигурациони подаци који су потребни за исправно подешавање и употребу софтвера.



Слика 1.1: *Организација дистрибуираног рачунарског система са средњим слојем [Tanenbaum and Van Steen, 2007]*

Имајући у виду претходно наведено, можемо увести следећу дефиницију дистрибуираног софтверског система:

> **Дистрибуирани софтверски систем (ДСС)** је колекција независних софтверских компоненти (подсистема) које се корисницима приказују као јединствен софтверски систем, при чему су софтверске компоненте рапоређене у дистрибуираном рачунарском систему.

Дистрибуирани софтверски системи су о основи велике већине данашњих социо-техничких система као што су уграђени и первазивни системи (*embedded and pervasive systems*), центри за рад са великим количинама података (*big data centers*), веб сервиси (*Web services*), системи у облаку (*cloud systems*), и системи система (*systems of systems*).

Да би обезбедио употребу хетерогених софтверских система, рачунара и мрежних система, дистрибуирани систем се најчешће организује тако да садржи **средњи слој** (*middleware*) који је смештен између виших слојева који чине корисничке софтверске апликације и нижих слојева који чине оперативни системи и мрежно-комуникациони системи. Организација дистрибуираног система са средњим слојем је приказана на слици 1.1. Средњи слој је распоређен на више рачунара обезбеђујући исти интерфејс за корисничке апликације без обзира на ком рачунару су покренуте.

1.1 Циљеви дистрибуираних система

Циљеви које је потребно остварити пројектовањем и изградњом дистрибуираних система су:

- **Доступност ресурса** (*Resources availability*). Основна идеја је да корисници система могу да деле дистрибуиране ресеурсе на ефикасан и контролисан начин, као и да се олакша сарадња корисника на заједничким пословима. Дељени ресурси могу бити стандардни уређаји као што су рачунари, штампачи, дискови, базе података, веб странице итд.
- **Транспарентност система** (*Transparency*). Овај циљ подразумева да корисник не треба да зна на ком делу система се реализују процеси (послови) које је он покренуо, већ да има утисак да се све одвија на рачунару преко којега је он приступио систему. Транспарентност се огледа у аспектима који су приказани у табели 1.1.
- **Отвореност система** (*Openness*). Систем нуди сервисе у складу са стандардним правилима који описују синтаксу и семантику сервиса (нпр. форма, садржај и значење порука током комуникације). Сервиси се обично описују помоћу интерфејса који су обично писани језиком за опис интерфејса (Interface Definition Language (IDL)).
- **Скалабилност** (*Scalability*). Скалабилност система се односи на могућност проширивања система додавањем ресурса и нових корисника, без обзира на њихову локацију. Кључни проблеми који утичу на скалабилност система су централизација сервиса, података и алгоритама.

Табела 1.1: *Аспекти транспарентности у дистрибуираним системима*

Аспект	**Опис**
Приступ	Скривање разлике у презентовању података и приступу ресурсима
Локација	Скривање локације ресурса
Миграција	Скривање могућности промене локације ресурса
Релокација	Скривање могућности да ресурси мењају локацију током употребе
Репликација	Скривање могућности репликације ресурса на више локација
Конкурентност	Скривање конкурентне (истовремене) употребе ресурса од стране више корисника
Откази	Скривање отказа и опоравака делова система

У пракси је немогуће обезбедити потпуну транспарентност система пошто су корисници свесни да раде у дистрибуираном систему. Због тога је само питање како најбоље приказати или сакрити дистрибуираност система корисницима.

Основни проблеми који се морају размотрити приликом пројектовања дистрибуираних система се односе на: поузданост, безбедност и хомогеност

рачунарских мрежа, променљивост топологије система, кашњења, пропусне опсеге, трошкове транспорта (преноса) и администрирање система.

1.2 Типови дистрибуираних система

Приликом разматрања дистрибуираних система, треба јасно разликовати дистрибуиране рачунарске системе (*distributed computing systems*), дистрибуиране информационе системе (*distributed information systems*) и дистрибуиране уграђене системе (*distributed embedded systems*).

1.2.1 Дистрибуирани рачунарски системи

Овакви системи се користее за сложене и захтевне послове рачунања, или послове рачунања са високим перформансама (*high-performance computing tasks*). На основу конфигурације система, овде се могу јасно разликовати следећа два типа система: (1) кластерски системи, и (2) *grid* системи.

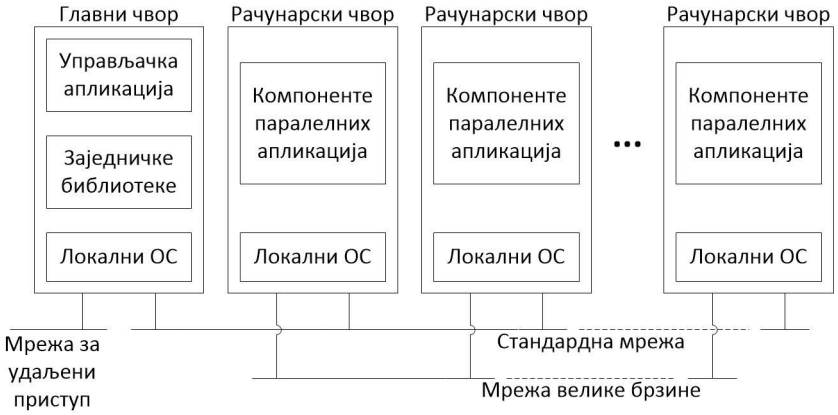
Кластерски рачунарски системи

Кластерски рачунарски систем се састоји од колекције сличних радних станица које су повезане локалном рачунарском мрежом. Радне станице се називају чворови. Свака радна станица има исти оперативни систем. Овакви системи имају одличан однос цене и перформанси па се врло често користе за паралелно извршавање програма (послова). Пример опште конфигурације кластерског система је приказан на слици 1.2. У оваквој конфигурацији свим чворовима у систему управља главни чвор (*master node*), који врши алокацију чворова за специфичне послове, одржава ред послова који треба да се изврше, и обезбеђује интерфејс за кориснике система. Главни чвор практично покреће средњи слој (*middleware*) који је задужен за покретање програма и управљање кластером. Остали чворови само извршавају додељене послове.

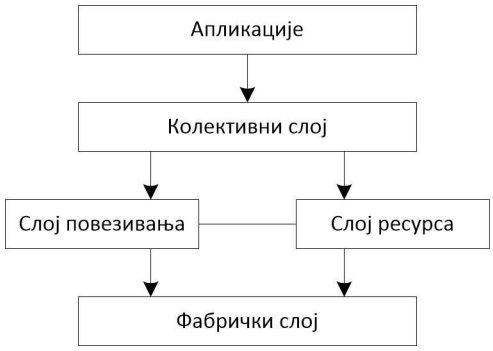
Средњи слој чине софтверске библиотеке које обезбеђују паралелно извршавање програма на више чворова. На пример, ове библиотеке обезбеђују комуникацију програма базирану на порукама (*message-based communication*). Примери кластерских система су Linux базирани системи Beowolf и MOSIX.

Грид рачунарски системи

Појам грид (*grid*) је настао средином 90-тих година прошлог века са циљем да означи дистрибуирану рачунарску инфраструктуру за напредна научна и инжењерска рачунања. Основна одлика грид система је хетерогеност, што значи да нема никаквих претпоставки у вези храдвера,



Слика 1.2: *Пример кластерског рачунарског система*



Слика 1.3: *Слојевита архитектура грид рачунарског система [Foster et al., 2001]*

оперативних система, рачунарске мреже, администрације домена и правила безбедности. Концепт грида се пре свега односи на координирано дељење ресурса и решавање проблема у динамичком окружењу које чине више **виртуелних организација** (*virtual organizations*). Дељење података не подразумева једноставно дељење датотека, већ директни приступ рачунарима, софтверима, подацима и другим ресурсима у складу са стратегијама дељења ресурса између више организација. Дељење ресурса је контролисано у смислу ресурса који се деле, ко има право приступа, као и услова под којима се ресурси деле.

Софтверски системи у грид рачунарским системима имају основну улогу да обезбеде приступ ресурсима у различитим административним доменима, и то пре свега корисницима који припадају специфичним виртуелним организацијама. Због тога је фокус на архитектури ових система. Пример слојевите архитектуре грид рачунарског система је приказан на слици 1.3.

Архитектура грид система се састоји од четири слоја. Најнижи слој, фабрички слој (*Fabric layer*) обезбеђује интерфејс ка локалним ресурсима неког окружења. Ови интерфејси омогућују упит стања и карактеристика ресурса, али и управљање ресурсима (нпр. закључавање ресурса).

Слој повезивања (*Connectivity layer*) се састоји од комуникационих протокола који подржавају трансакције за употрубу ресурса. У овом слоју се налазе и безбедносни протоколи за утентификацију корисника и ресурса. Најчешће се кориснице не аутентифицирају, већ се то делегира програмима које они користе.

Слој ресурса (*Resource layer*) је одговоран за управљање ресурсима, при чему користи функције слоја повезивања и директно користи интерфејсе фабричког слоја за приступ ресурсима. Овај слој обезбеђује приступ конфигурационим информацијама специфичних ресурса.

Колективни слој (*Collective layer*) обезбеђује руковање приступу ресурсима. Слој садржи функције за откривање, алокацију и распоређивање послова на више ресурса, као и за репликацију података. Овај слој садржи велики број протокола различите намене, што одсликава широк спектар сервиса које може понудити нека виртуелна организација.

Слој апликација (*Application layer*) се састоји од апликација које се користе у виртуелној организацији, а које омогућују коришћење грид система.

Колективни слој, слој ресурса и слој повезивања чине "срце" грид система и често се називају средњи слој (*middleware layer*).

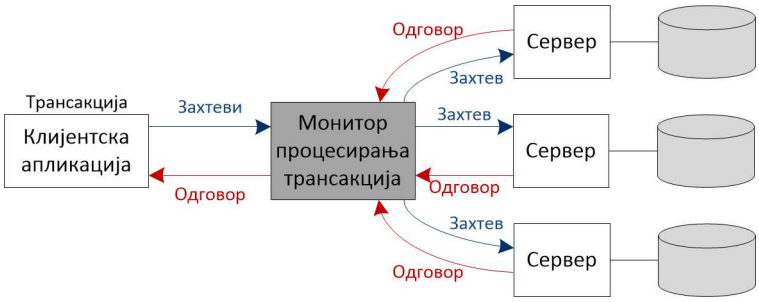
1.2.2 Дистрибуирани информациони системи

Веома значајна класа дистрибуирани система су дистрибуирани информациони системи који се користе као подршка пословања у организацијама различитог типа. Овакви системи се реализују тако што интегришу различите типове апликација, или се базирају на јединственом средњем слоју. У највећем броју случајева овакви системи имају посвећене сервере који помоћу комуникационе инфраструктуре пружају услуге клијентима (корисницима). У таквим системима клијенти обично шаљу серверу захтев за неком услугом, сервер изврши одговарајућу операцију, и потом врати одговор клијенту.

Системи базирни на процесирању трансакција

Појам трансакције (*transaction*) потиче из области база података, где су трансакције уобичајен начин извршавања операција над базом података. Ако се више захтева за операцијама на више различитих сервера упакује у јединствен захтев, тада се тај захтев може извршити као **дистрибуирана трансакција** (*distributed transaction*). Основна идеја је да ће се у оквиру дистрибуиране трансакције извршити или сви упаковани захтеви или ни један. Основне карактеристике трансакција су:

- **Атомичност** (*atomic*) - гледано споља трансакција је недељива, што значи да се извршавају или све операције или ни једна.
- **Конзистентност** (*consistent*) - систем или неки његови сегменти остају непроменљив након извршавања трансакције.



Слика 1.4: *Улога монитора дистрибуираних трансакција у ДСС*

- **Изолованост** (*isolated*) - трансакције се извршавају независно или изоловано једна од друге.
- **Трајност** (*durable*) - када се трансакција потврди, промене које је направила су трајне.

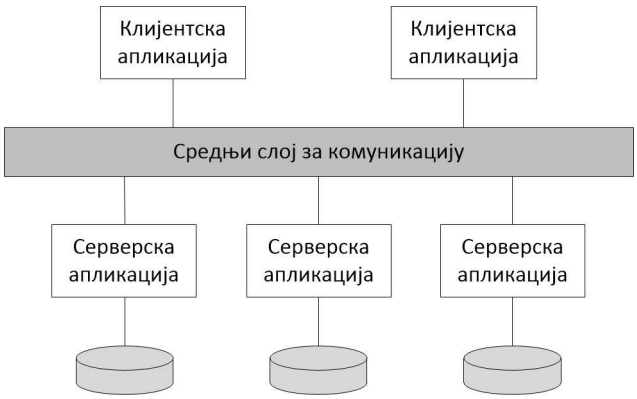
Дистрибуирана трансакција се може извршавати на више сервера, што значи да може имати угњеждено више подтрансакција. Трансакције се могу угњеждавати на више нивоа, што захтева сложене алгоритме управљања трансакцијама. Основно правило код оваквих система је да ако се неуспешно заврши трансакција, она ће поништити све резултате свих њених подтрансакција које су се успешно завршиле. За управљање дистрибуираним трансакцијама се користи компонента монитор процесирања трансакција (*transaction processing monitor*, *TP monitor*) који омогућује приступ ка више сервера кроз модел програмирања трансакција (слика 1.4).

Интеграција пословних апликација

У сложеним пословним системима је потребно интегрисати различите апликације које комуницирају не само помоћу модела дистрибуираних трансакција већ и другим методама (поруке, позив удаљене процедуре). Различите апликације имају потребу да директно комуницирају, па се такав модел комункације базира на средњем слоју за комуникацију (*communication middleware*), што је приказано на слици 1.5.

Комуникација у средњем слоју се реализује применом различитих метода. Најчешће се користе:

- **Позив удаљене процедуре** (Remote Procedure Call (RPC)). Софтверска компонента која има потребу да пошаље захтев другој софтверској компоненти то ради позивом локалне процедуре која пакује тај захтев и шаље га као поруку другој компоненти. Резултат се пакује и враћа софтверској компоненти као резултат позива процедуре.
- **Позив удаљене методе** (Remote Method Invocation (RMI)). RMI је објектно оријентисана верзија Remote Procedure Call (RPC) при чему комуницирају објекти који позивају методе других објеката.



Слика 1.5: *Интеграција пословних апликација помоћу средњег слоја за комуникацију*

- **Средњи слој базиран на порукама** (Message-Oriented Middleware (MOM)). Објекти конуицирају тако што поруке шаљу тачки логичког контакта објекта који се позива. Свака апликација истиче које типове порука може да обради, па комуникациони средњи слој тада само такве поруке прослеђује апликацијама. Овакав систем се назива **објави/пријави** (*publish/subscribe*) и све се чешће користи у дистрибуираним софтверским системима.

Основни недостаци RMI и RPC су: (1) објекти, односно софтверске компоненте, које комуницирају морају бити покренуте (активне), и (2) објекти морају знати начин међусобног позивања (обраћања).

1.2.3 Дистрибуирани первазивни системи

Дистрибуирани системи базирани на мобилним (*mobile*) и уграђеним (*embedded*) уређајима имају у принципу нестабилно понашање због тога што се њихов положај може мењати и зависи од окружења. Заједнички назив за овакве системе је **первазивни дистрибуирани системи** (*pervasive distributed systems*). Уређаји у оваквим системима су мали, мобилни, напајају се батеријама и користе бежичну конекцију на мрежу система. Овакви системи су део нашег животног и радног окружења. Њихова основна карактеристика је да их конфигурише власник или се сами подешавају према окружењу где се налазе (користе).

Основни захтев за дизајн первазивних дистрибуираних система је да лако прихватају промене контекста. То значи да систем прати промене окружења и реагује тако да и даље врши намењену улогу (нпр. ако уређај препозна да је подразумевана мрежа недоступна покушаће да се преконфигурише да користи неку од доступних). Овакво понашање је подржано могућношћу ад-хок конфигурисања да би се систем прилагодио окружењу. Најчешће примене су у домену кућних система, електронских система у медицини, пољопривреди и индустрији. У зависности од примене, процесирање података се може радити директно у окружењу (нпр.

Слика 1.6: *Сензорска мрежа са централизованом базом података*

индустријске примене или примене у пољопривреди), или се подаци само прикупљају у окружењу и потом шаљу удаљеном серверу на обраду (нпр. сензорска мрежа која се користи за праћење стања пацијента).

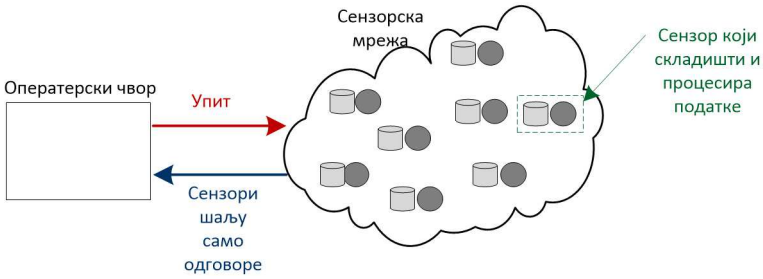
Сензорске мреже

Сензорске мреже имају значајну примену у дистрибуираним системима пошто имају могућност процесирања информација, што проширује аспект примене у односу на традиционално посматрање мреже као комункационог система. Сензорске мреже се састоје од великог броја (десетине или стотине) релативно малих чворова који имају сензорни уређај, користе бежичну комуникацију и напајање на батерије. Ограничења која намећу бежична комуникација и ограничени ресурси због напајања батеријама постављају захтев за великом ефикасношћу као примарни циљ приликом дизајна.

Због могућности да сензорски чворови процесирају и складиште податке дистрибуирани системи овог типа се могу посматрати и као дистрибуиране базе података. Постоје два приступа у дизајну сензорских мрежа као дистрибуираних база података. Први приступ подразумева да сензори прикупљају податке и шаљу их удаљеном оператерском чвору на којем је централизована база података, што је приказано на слици 1.6.

Други приступ подразумева да сваки сензор чува и процесира своје податке, а када добије упит од оператерског чвора даје одговор. Оператерски чвор има апликације које су задужене за агрегацију одговора (података) са различитих сензорских чворова, што је приказано на слици 1.7.

Оба ова приступа имају одређених предности али и недостатака. Први приступ са централиованом базом података на оператерском чвору подразумева да сензори шаљу све податке оператерском чвору што може бити проблем за мрежне ресурсе и потрошњу енергије, док други приступ где сваки сензор има способност складиштења и процесирања података има недостатак да се оператеру не шаљу сви подаци већ само подаци према упиту. Оптимално решење је да се уведе процесирање података у самој мрежи (*in-network data processing*) помоћу додатног чвора који би радио прослеђивање упита и потом агрегацију података прикупљених са сензора. Агрегација се може радити и на више чворова у мрежи, при чему се сваком чвору може доделити подскуп сензорских чворова за које врши агрегацију



Слика 1.7: *Сензорска мрежа са централним оператером и сензорским чворовима који складиште и процесирају локалне податке*

података. На тај начин се добија агрегација на више нивоа, где се подаци добијени агрегацијом прослеђују на виши ниво, а на највишем нивоу је чвор који комуницира са оператерским чвором. На овај начин се мрежни ресурси много ефикасније користе и мања је потрошња енергије.

Поглавље 2

Инжењеринг дистрибуираних софтверских система

Инжењеринг дистрибуираних софтверских система (ДСС) обухвата следеће области:

- Основна питања и принципи дизајна и имплементације ДСС.
- Клијент-сервер (*client–server*) модел рачунања и слојевита архитектура клијент-сервер система.
- Типични шаблони (*patterns*) архитектура дистрибуираних система и специфичности њихове примене.
- Разумевање концепта софтвер као сервис (*software as a service*) и веб базирани приступ удаљеним софтверским системима.

2.1 Основа питања дизајна и имплементације ДСС

Основни проблем у дизајну и имплементацији ДСС јесте хетерогеност система са аспекта софтверских компоненти, али и хардверске и комуникационе инфраструктуре. Због проблема са предвидљивошћу понашања, управљање оваквим система је сложено. Због тога су основна питања која треба размотрити у инжењерингу ДСС:

- **Транспарентност** (*Transparency*) се односи на начин приказа ДСС корисницима као јединственог система.

- **Отвореност** (*Openness*) се односи на дизајн базиран на скупу стандардних протокола који подржава интероперабилност различитих компоненти или подсистема.
- **Скалабилност** (*Scalability*) се односи на дизајн система тако да он буде скалабилан, тј. да се његов капацитет може повећати у складу са захтевима.
- **Безбедност** (*Security*) се односи на начин дефинисања и имплементације безбедносних правила (политика) у управљању системом или појединим деловима система.
- **Квалитет сервиса** (*Quality of Service, QoS*) се односи на начин спецификације одређеног нивоа квалитета сервиса који се испоручује корисницима система.
- **Управљање отказима** (*Failure management*) се односи на начине детекције и отклањања отказа тако да они имају минималан утицај на функционисање система

ДСС су много комплекснији од централизираних система, па је због тога и њихов дизајн, имплементација и тестирање знатно сложеније. Поред тога треба узети у обзир и комплексност интеракције између различитих компоненти ДСС, као и компоненти и инфраструктуре. На пример, на извршавање кода неке софтверске компоненте могу утицати и брзина рачунарске мреже, пропусна моћ комуникационих уређаја и канала, и брзина свих осталих рачунара у систему.

С обзиром на ограничења која има комуникациона инфраструктура, а која утичу на пренос или простирање сигнала, кашњење одзива дистрибуираних софтверских компоненти је неизбежно. Због тога је немогуће обезбедити потпуну транспарентност система, пошто ће корисник увек бити свестан различитих перформанси извршавања појединих софтверских сервиса, што је последица њихове локације у систему. Такође, отказ удаљеног мрежног чвора може изазвати престанак рада софтвера, а корисник и даље има приступ систему на свом чвору, па и таква ситуација нарушава транспарентност система.

Отвореност система се пре свега односи на могућност да се софтверске компоненте (системи) развију применом различитих технологија и потом интегришу у јединствени ДСС. Услов је да софтверске компоненте користе скуп протокола који је предвиђен за комуникацију у систему.

Скалабилност треба да обезбеди квалитет и доступност софтверских сервиса без обзира на повећане захтева од стране корисника. Три димензије скалабилности су:

- *Величина* - могуће је додати нове ресурсе у систем да би се пружиле услуге повећаном броју корисника.
- *Дистрибуција* - могуће је географски распоредити софтверске компоненте а да се не деградирају перформансе система.
- *Управљивост* - могуће је ефикасно управљати системом без обзира на повећање броја ресурса система и њихов распоред.

Безбедност је знатно теже обезбедити код ДСС него код централизованих система. Проблем може настати ако нападач "продре" у једну компоненту система, па онда њу користи за приступ осталим компонентама (*back door*). Основни проблем је обезбедити безбедносна правила која се једнако примењују у свим компонентама система.

Квалитет сервиса се односи на могућност система да сервис испоручи на време и на начин који је прихватљив корисницима.

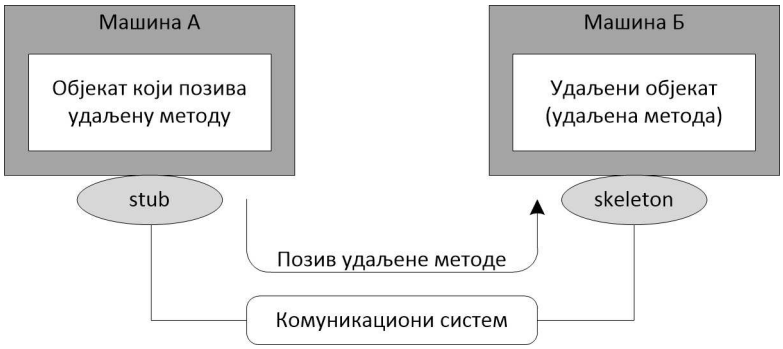
Откази компоненти у ДСС су неибежни, па систем треба пројектовати тако да то не утиче на његов рад. Постоје разне технике побољшање система по питању отказа компоненти. ДСС треба да обезбеди механизме откривања компоненти које су отказале, као и аутоматски опоравак компоненти и система.

2.1.1 Модели интеракције у ДСС

Два основна модела интеркције у дистрибуираним системима су базирана на процедурама (*procedural interaction*) и порукама (*message-based interaction*). Интеракција базирана на процедурама подразумева да софтверска компонента на једном рачунару позива познати сервис који нуди друга софтверска компонента на другом рачунару, и потом чека на испоруку сервиса. Интеракција базирана на порукама подразумева да једна софтверска компонента дефинише шта се очекује у поруци и потом то шаље другој софтверској компоненти на другом рачунару. Порука у нтеракцији базираној на порукама у једној интеракцији може пренети више информација него један позив процедуре у процедуралној интеракцији.

Процедурална комуникција се обично имплементира као позив удаљене процедуре (Remote Procedure Call (RPC)), где једна компонента позива другу другу компоненту тако да то изгледа као локални позив. Средњи слој система прихвата позив и прослеђује га удаљеној компоненти. На *Java* плтформи се овакав процедурални позив реализује као позив удаљене методе (Remote Method Invocation (RMI)).

Приликом позива удаљене процедуре, позивалац (клијентска страна) мора имати **stub** од удаљене процедуре компоненте која даје услугу (серверска страна) приликом позива. **Stub** је код који конвертује параметре које прослеђује компонента која позива процедуру у параметре које може да користи позвана удаљена процедура у другој компоненти. На тај начин позив удаљене процедуре изгледа као позив локалне процедуре. **Stub** библиотека мора бити инсталирана и на клијентској и на серверској страни. Проблем са позивом удаљене процедуре је да и позивалац и позвани морају бити доступни сво време. Код интеракције базиране на порукама пошиљалац и прималац поруке не морају бити доступни сво време пошто се поруке отпремају у њихове редове за поруке. На страни компоненте која пружа услугу, тј. методу која се позива, користи се **Skeleton** који је објекат која се понаша као пролаз (*gateway*) ка серверском објекту. Сви позиви удаљене процедуре пролазе кроз **Skeleton** који при томе реализује следеће послове: (1) чита параметре удаљене методе која се позива, (2) позива методу, и (3) прослеђује (*marshals*) резултате објекту који је извршио позив. Пример



Слика 2.1: *Позив удаљене методе помоћу Stub и Skeleton објеката*

комуникације компоненти помоћу **Stub** и **Skeleton** објеката у позиву удаљене процедуре је приказан на слици 2.1.

Интеракција базирана на порукама подразумева да једна компонента креира поруку са детаљима захтеваног сервиса од стране друге компоненте. Средњи слој система шаље поруку одредишној компоненти која нуди сервис, која потом парсира поруку, извршава захтеване операције и креира поруку која се шаље пошиљаоцу. Поруку пошиљаоцу шаље средњи слој система. У ДСС су комуникационе поруке у формату eXtensible Markup Language (XML). Пример 2.1.

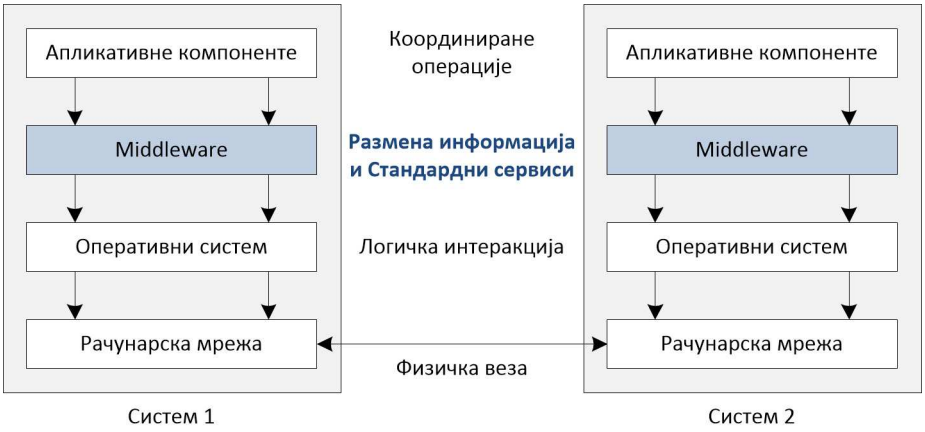
Листинг 2.1: *Пример комуникације помоћу XML порука*

```
Request:
POST /bankinfo HTTP/1.1
Host: www.dss-test.com
<GetBalance>
  <Account>33557799</Account>
  <SessionId>11</SessionId>
</GetBalance>

Response:
<Balance acct="33557799">350.25</Balance>
```

2.1.2 Средњи слој

Компоненте у ДСС се могу имплементирати у различитим програмским језицима и извршавати на различитим типовима процесора. Такође, модели података, приказ информација и комуникациони протоколи могу бити потпуно различити. Због тога ДСС треба да садржи софтвер који управља овим различитим компонентама система, и обезбеђује да оне комуницирају и размењују податке. Овај софтвер се назива **средњи слој** или *middleware*, пошто се он налази између различитих софтверских компоненти у ДСС. Средњи слој се заправо налази између апликативних програма и оперативних система као што је приказано на слици 2.2.



Слика 2.2: *Middleware у дистрибуираном софтверском систему*

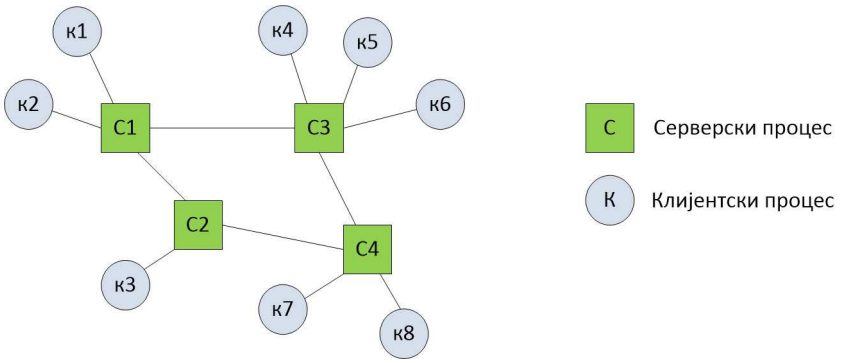
Middleware се имплементира као скуп библиотека који се инсталира на сваком рачунару у дистрибуираном систему, заједно са *run-time* системом који управља комуникацијом. Примери софтвера средњег слоја су софтвери за управљање комуникацијом са базама података, конвертори података, контролери комуникације итд.

2.2 Клијент-сервер системи

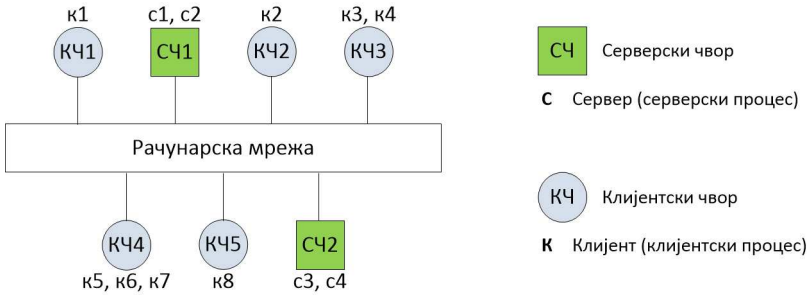
ДСС којима се приступа преко Интернета су најчешће организовани као клијент-сервер системи (*client–server systems*). У оваквим системима корисник је у интеракцији са софтвером који је покренут на његовом локалном рачунару (нпр. веб претраживач или мобилна апликација за телефон), а тај софтвер комуницира са софтвером који се извршава на удаљеном рачунару (нпр. веб сервер). Удаљени софтверски систем пружа сервисе као што су приступ веб страницама, или приступ бази података. Овакав модел је најопштији у дистрибуираним системима, и може се применити чак и када су и клијент и сервер на истом рачунару.

Софтверска апликација са клијент-сервер архитектуром се моделује као скуп сервиса (*services*) које пружају сервери (*servers*). Клијенти (*clients*) захтевају услуге сервиса и резултате приказују корисницима. Клијенти морају бити свесни постојања доступних сервера, али не морају се међусобно знати. Клијенти и сервери се покрећу као независни процеси, као што је приказано на слици 2.3.

Клијенти и сервери се могу покретати на истим или различитим рачунарима који су повезани у мрежи. Пример расподеле клијената и сервера на мрежне чворове је приказан на слици 2.4, где имамо систем са 4 сервера и 8 клијената распоређених на 2 серверска чвора и 5 клијентских чворова. Сервери се обично имплементирају на мулти-процесорским системима у



Слика 2.3: *Интеракција клијената и сервера*

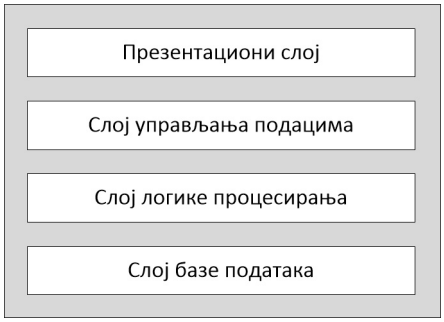


Слика 2.4: *Придруживање клијената и сервера мрежним чворовима*

којима посебан софтвер прати оптерећење свих процесора и распоређује клијентске захтеве тако да се обезбеди равномерно оптерећење.

У клијент-сервер моделу се обрада и приказ података могу реализовати на различитим рачунарима који покрећу одговарајуће процесе. То омогућује да се овакав модел реализује као архитектура са више логичких слојева и јасним интерфејсима између слојева. При томе сваки слој може бити рапоређен на другом рачунару. На слици 2.5 је приказан склијент-сервер модел софтверске апликације са 4 слоја:

- **Презентациони слој** (*presentation layer*) је задужен за презентовање информација корисницима и управљање интеракцијом са корисником.
- **Слој управљања подацима** (*data management layer*) управља подацима који се прослеђују ка клијенту и од клијента. У овом слоју се врши провера података, а такође се може реализовати и генерисање веб страница са садржајем.
- **Слој логике процесирања** (*application processing layer*) је задужен за имплементирање логике коју апликација користи да би обезбедила захтеване функционалности корисницима.
- **Слој базе података** (*database layer*) је задужен за смештање података и пружање сервиса за управљање трансакцијама.



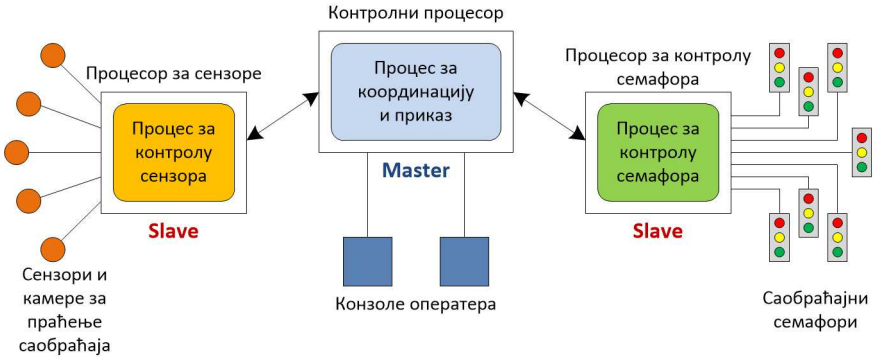
Слика 2.5: *Слојевити модел клијент-сервер софтверске апликације*

Клијент-сервер модел се користи у разним архитектурама у ДСС које користе Интернет за рапоређивање и пружање сервиса, а такође представља и основу на којој је изграђен модел *софтвера као сервиса* (Software as a Service (SaaS)).

2.3 Модели архитектуре у ДСС

Основни циљ пројектовања ДСС је да се обезбеди баланс између перформанси, зависности, безбедности и управљивости системом. Различити модели архитектуре су предложени са циљем да се испуне наведени услови. Приликом пројектовања система је најважније обезбедити испуњеност критичких не-функционалних захтева за систем. Следећи модели архитектуре се најчешће користе:

- **Master-slave архитектура** (*Master-slave architecture*) се користу системима у реалном времену (*real-time systems*) у којима је важно обезбедити интеракцију у задатим временским оквирима.
- **Двослојна клијент-сервер архитектура** (*Two-tier client–server architecture*) се користу у једноставним клијент-сервер системима и у системима где је неопходно из безбедносних разлога имати централизован систем.
- **Вишеслојна клијент-сервер архитектура** (*Multitier client–server architecture*) се користи у системима са великим обимом трансакција на серверима.
- **Архитектура дистрибуираних компоненти** (*Distributed components architecture*) се користи када треба комбиновати ресурсе различитих система и база података, или као имплементациони модел у вишеслојним клијент-сервер системима.
- **Архитектура равноправних чланова** (*Peer-to-peer architecture*) се користи када корисници размењују локално чуване информације, а сервер се користи само да међусобно упознаје клијенте. Сви чланови у



Слика 2.6: *ДСС за управљање саобраћајем са master-slave архитектуром*

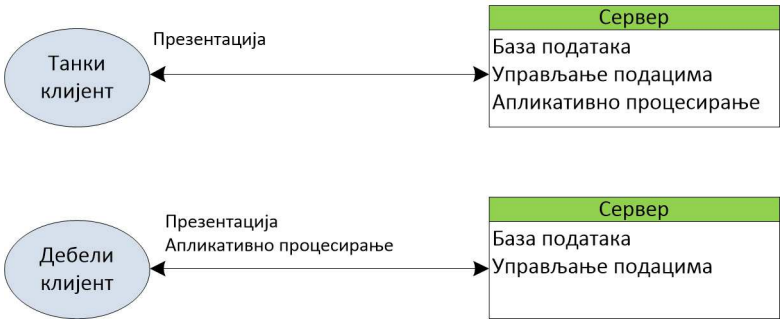
систему су равноправни. Такође се користи и када је потребно извршити велики број независних операција (рачунања).

2.3.1 Master-slave архитектура

Master-slave архитектура се обично користи у системима који раде у ралном времену (индустрија, пољопривреда, медицина, саобраћај). У таквим системима се различити процесори, и процеси, користе за прикупљање података из окружења, процесирање података, рачунање, складиштење података и управљање актуаторима (уређаји који врше промене у окружењу на осноу управљачких сигнала које добијају од софтвера). Глави (*master*) процес је одговоран за израчунавања, координацију, комуникацију и контролу подређених (*slave*) процеса. Подређени процеси се користе за специфичне послове као што су прикупљање података од сензора у окружењу или управљање уређајима у окружењу.

На слици 2.6 је приказан модел *Master-slave* архитектуре за систем управљања у саобраћају са три процеса који су покренути на различитим процесорима. Главни (*master*) процес се покреће у командној соби и има терминале за оператере. Главни процес контролише два подређена (*slave*) процеса који се извршавају на другим процесорима који могу бити физички дислоцирани од главног процесора. Подређени процеси су задужени за прикупљање података са сензорских станица и за управљање семафорима.

Овај модел се користи у ДСС где је могуће предвидети дистрибуирано процесирање које се може лако распоредити на удаљене подређене процесоре. Ово је уобичајено у системима који раде у реалном времену где су операције временски ограничене. Подређени процесори (процеси) се врло често користе за захтевне операције израчунавања и процесирања (обрада сигнала или управљање опремом), чиме се ослобађа главни процесор који је задужен за надзор помоћних процесора.



Слика 2.7: *Модели танке и дебеле двослојне клијент-сервер архитектуре*

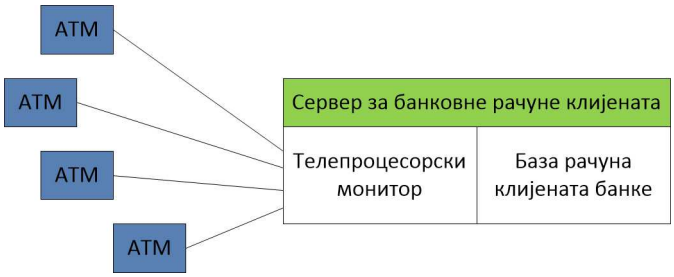
2.3.2 Двослојна клијент-сервер архитектура

Основни модел клијент-сервер система подразумева постојање клијентског рачунара где се извршава клијентски део софтвера и удаљеног сервера где се извршава серверски део софтвера. Двослојна клијент'сервер архитектура је најједноставнији облик клијент-сервер модела који се имплементира са једним сервером и произвољним бројем клијената који користе сервер. Две основне форме овог модела, приказане на слици 2.7, су:

- **Модел са танким клијентом** (*thin-client model*) је модел код којег је само презентациони слој имплементиран на страни клијента, док су сви остали слојеви (управљање подацима, апликативно процесирање података, складиштење података) имплементирани на серверу. Клијентски софтвер може бити специјални писани софтвер за руковање представљањем, а најчешће је то веб претраживач.
- **Модел са дебелим клијентом** (*fat-client model*) је модел код којег је поред презентационог слој на страни клијента имплементиран и део апликативног процесирања, док су управљање и складиштење података имплементирани на серверу.

Основна предност танких клијената је лако управљање. Озбиљан проблем може представљати инсталирање комплексних софтвера на великом броју клијентских рачунара. Врло често се као танки клијент користи веб претраживач, па тада нема потребе за инсталирањем клијентског софтвера. Недостатак система са танким клијентима је велико оптерећење сервера на којем се врши комплетно процесирање, као и оптерећење комуникационе инфраструктуре. Неко минимално процесирање се може одрадити и у веб претраживачима, нпр. применом *JavaScript*-а, што може ослободити сервер дела процесирања података (нпр. провера типа унетих података у веб страници).

Модел са дебелим клијентом пребацује део или комплетно апликативно процесирање на клијентски рачунар и тиме троши његове процесорске капацитете. Сервер се тада понаша као трансакциони сервер који управља подацима. Проблем са дебелим клијентима настаје због сложеније испоруке и одржавања клијентских софтвера.



Слика 2.8: *Архитектура ATM система са дебелим клијентима*

Пимер система са дебелим клијентом је банкомат (Automated Teller Machine (ATM)) који испоручује кеш и пружа друге банкарске услуге корисницима (нпр. трансакције између рачуна, провера стања). ATM је клијентски рачунар који процесира акције корисника, док је сервер обично удаљени *mainframe* рачунар на којем се трансакционо управља подацима. Пример дистрибуираног ATM система је приказан на слици 2.8. ATM клијенти не комуницирају директно са базом података већ са телепроцесорским монитором (*teleprocessing monitor*), који је реализован као софтвер средњег слоја (*middleware*) који организује комуникацију са удаљеним клијентима и врши серијализацију клијентских трансакција према бази података.

Процесирање је у системима са дебелим клијентима много ефикасније него у системима са танким клијентима, али је управљање системом значајно комплексније. Апликативне функционалности су распоређене на различите удаљене рачунаре, па то знатно отежава одржавање система. На пример, када се клијентски апликативни софтвер мења, понекад је потребно и рестартовати клијентске рачунаре, што уноси значајне потешкоће у одржавању, а исто тако и повећава трошкове одржавања. Системе треба пројектовати да имају подршку за удаљену аутомаску надоградњу софтвера (*remote software upgrades*).

2.3.3 Вишеслојна клијент-сервер архитектура

У вишеслојној клијент-сервер архитектури су различити слојеви система (презентациони, управљање подацима, апликативно процесирање и складиштење података) имплементирани као независни процеси који се могу извршавати на различитим процесорима. На тај начин се повећава скалабилност, перформансе и лакоћа одржавања система. Пример трослојног система система са танким веб клијентима је приказан на слици 2.9.

На слици 2.9 је клијент реализован као веб претраживач који се извршава на корисниковом рачунару и омогућава приказ и елементарно процесирање података. Веб сервер, који је други слој у архитектури, обезбеђује услуге управљања подацима и апликативно процесирање. Трећи слој је реализован на рачунару који има сервер базе података. Овакав систем је скалабилан и лако се може проширити у случају повећања броја клијената.

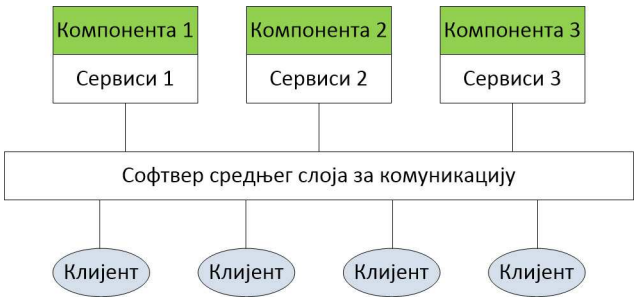
Слика 2.9: *Трослојна клијент-сервер архитектура са танким веб клијентима*

Оваква трослојна архитектура обезбеђује ефикасну комуникацију између веб сервера на које се извршава *middleware* који подржава упите у Structured Query Language (SQL), док се комуникација са клијентима обавља помоћу протокола HTTP Secure (HTTPS).

Модел трослојне клијент-сервер архитектуре се може проширити на вишеслојну архитектуру додавањем нових сервера у систем. Најпре се веб север може реализовати на два сервера, један за управљање подацима а други за апликативно процесирање. Такође се може користити и више сервера база података ако је потребно користити различите типове сервера за базе података, у ком случају би се интеграција података прибављених са различитих сервера вршила на апликативном веб серверу који би то проследио презентационом слоју. Значајан број фактора треба разматрати приликом дизајна и имплементације вишеслојних клијент-сервер архитектура. У табели 2.1 су приказане типичне ситуације примене вишеслојних клијент-сервер архитектура.

Табела 2.1: *Модели примене вишеслојних клијент-сервер архитектура*

Архитектура	**Примена**
Двослојна клијент-сервер архитектура са танким клијентима	(1) Пословне апликације у којима није практично раздвајати апликативно процесирање и управљање подацима. (2) Рачунарски захтевне апликације као што су компајлери. (3) Апликације са интензивним радом са подацима али без захтевног апликативнохг процесирања (нпр. претраживање веба).
Двослојна клијент-сервер архитектура са дебелим клијентима	(1) Апликације у којима се процесирање података врши већ постојећим (*off-the-shelf software*) софтвером, као што је на пример Microsoft Excel. (2) Апликације са интензивним процесирањем података, као што су апликације за визуелизацију. (3) Мобилне апликације у случајевима када доступност Интернета може бити проблем (логичко процесирање и кеширање података на клијенту).
Вишеслојна клијент-сервер архитектура	(1) Велике апликације са великим и променљивим бројем клијената. (2) Апликације у којима се интегришу подаци из различитих система (извора). (3) Апликације у којима је и управљање подацима и апликативно процесирање подложно честим променама.



Слика 2.10: *Архитектура дистрибуираних компоненти*

2.3.4 Архитектура дистрибуираних компоненти

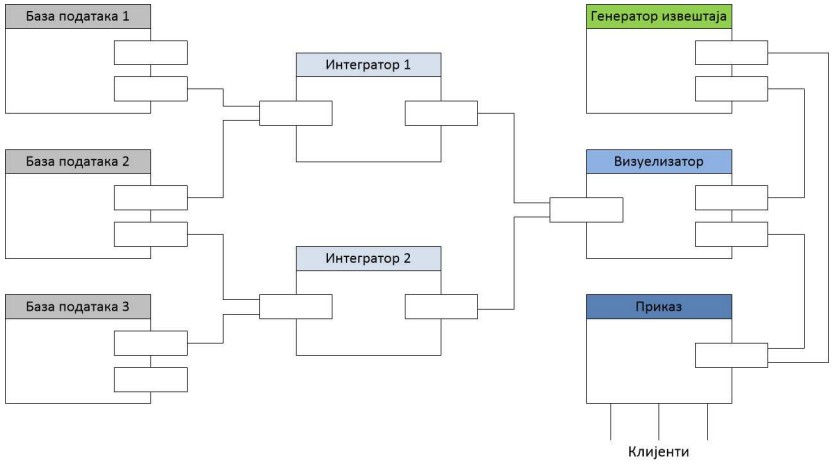
Вишеслојни клијент-сервер модели имају ограничење флексибилности приликом дизајна система које подразумева да пројектанти морају током пројектовања стриктно одлуичити који сервис ће бити имплементиран у којем слоју система. У пракси врло често није увек јасно да ли ће се неки сервис користити за апликативно процесирање или управљање подацима, или као сервис базе података.

Генералнији приступ дизајну дистрибуираних система се базира на дизајну система као скупу сервиса, без покушаја да се сервиси стриктно имплементирају у неком слоју система. Сваки сервис, или група сервиса се имплементира као независна компонента. У оваквој архитектури ДСС је организован као скуп међусобно повезаних компоненти које обезбеђују интерфејсе ка сервисима као што је приказано на слици 2.10.

ДСС базирани на компонентама се ослањају на средњи софтверски слој који управља интеракцијама између компоненти, усаглашава разлике између типова параметара који се прослеђују између компонената, и обезбеђује скуп заједничких сервиса које користе софтверске компоненте.

За реализацију средњег софтверског слоја је средином деведесетих година прошлог века дефинисан стандард Common Object Request Broker Architecture (CORBA), који је развила Object Management Group (OMG) са циљем да подржи комуникацију у ДСС који се имплементирају на различитим платформама. CORBA омогућује комуникацију компоненти писаних у различитим програмским језицима, и које се извршавају на различитим оперативним системима и хардверским платформама. Данас у пракси пројектовања средњег софтверској слоја доминирају Enterprise Java Beans (EJB) и .NET. Употреба архитектуре ДСС базиране на дистрибуираним компонентама има следеће предности:

- Омогућује пројектанту да одлуку о томе где ће се компонента поставити у систему донесе у произвољном тренутку током пројектовања ДСС, тј. у ком слоју или на којем рачунару ће бити инсталирана.
- Потпуно отворена архитектура за проширивање додатним ресурсима. Нови сервиси се могу додавати без поремећаја у раду система.



Слика 2.11: *Архитектура дистрибуираних компоненти за data mining систем*

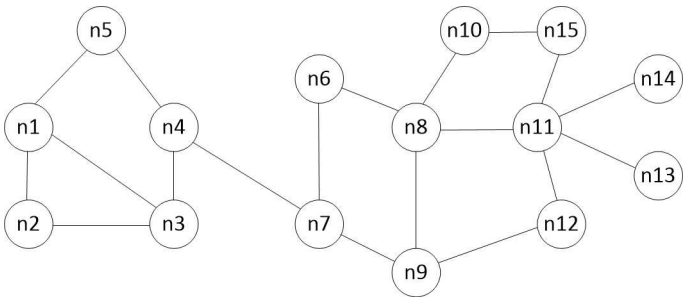
- Систем је флексибилан и скалабилан. Нове компоненте се могу додавати ако се повећа оптерећење система без утицаја на функционисање.
- Систем је могуће динамички реконфигурисати тако да се компоненте мигрирају у мрежи према потреби, чиме се могу битно побољшати перформансе система.

Дистрибуирани систем се може посматрати као логички модел који обезбеђује структуру за организовање система. Функционалности система се посматрају искључиво у смислу сервиса или комбинације сервиса који се реализују комбинацијом дистрибуираних компоненти. Системи за интензиван рад са подацима (нпр. *data mining* системи) су типичан пример где се користи архитектура са дистрибуираним компонентама. На слици 2.11 је приказан дистрибуирани систем за *data mining* са више сервера база података. Компоненте за интеграцију података из различитих база података обезбеђују да се подаци припреме за одговарајућу употребу (извештаји, визуелизација, приказ).

Типични примери би били малопродајни системи за храну, књиге, пољопривредну опрему итд. Код оваквих система се лако врши проширење додавањем нових сервера за базе података или нових функционалности у компонентама за интеграцију или приказ резултата корисницима.

Основни недостаци архитектуре са дистрибуираним компонентама су:

- Дизајн је значајно сложенији него код клијент-сервер архитектура.
- Значајно је тежа за разумевање и визуелизацију у односу на клијент-сервер архитектуру.
- Стандардизован средњи софтверски слој никада није прихваћен у пракси, већ различити произвођачи (нпр. *Microsoft* и *Oracle*) имају своје некомпатибилне верзије.
- Средњи софтверски слој је у принципу веома сложен што доприноси укупној сложености целог ДСС.



Слика 2.12: *Децентрализована P2P архитектура*

Као одговор на ове проблеме све више се користи архитектура оријентисана на сервисе (Service-Oriented Architecture (SOA)). С обзирома да се комуникација у ДСС базираним на компоенентама ослања на RPC који је значајно бржи од комуникације базиране на порукама у SOA, ДСС базирани на компонентама се користе у системима који имају захтев за великом пропусном моћи и брзинама у којима се обрађује велики број трансакција.

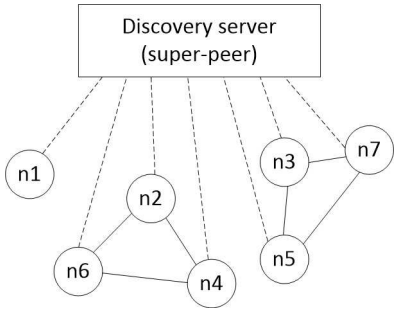
2.3.5 Архитектура равноправних чланова

Peer-to-Peer (P2P) системи су децентрализовани системи код којих се процесирање и рачунање могу равноправно извршити на било ком чвору у мрежи. У оваквим системима нема разлике између клијената и сервера пошто се P2P системи пројектују тако да користе процесорске моћи великог броја чворова у мрежи. Ови системи се више користе у персоналним системима него у пословним. Примери примене су у системима за дељење фајлова као што су *Gnutella* и *BitTorrent*, инстант системи за поруке као што су *ICQ* или *Jabber*, или Voice over Internet Protocol (VoIP) сервис као што је *Skype*. P2P системи су нарочито погодни за употребу у следећим случајевима:

- Код система који имају велике захтеве за рачунањем које је могуће разделити у одвојене процесе који се могу извршити као велики број независних рачунања.
- Код система где је примарни циљ размена информација између великог броја рачунара у мрежи, а не постоји потреба за централним чувањем информација.

За успешно функционисање P2P мрежа је потребно да је сваки чвор "свестан" постојања свих осталих чворова у мрежи, што омогућује директну размену података између чворова. Међутим, у пракси је то немогуће остварити, па су чворови организовани у локалне групе при чему поједини чворови се понашају као мостови (*bridges*) према другим локалним групама. Оваква децентрализована P2P архитектура је приказана на слици 2.12.

У децентрализованој архитектури чворови у мрежи нису само функционални елементи већ и комуникациони елементи који усмеравају



Слика 2.13: *Полуцентрализована P2P архитектура*

податке и контролне сигнале између чворова. Предност овакве архитектуре је да је она веома редундантна, отпорна на отказе (*fault-tolerant*) и одјављивање чворова из мреже, док је недостатак повећан саобраћај у мрежи.

Алтернативни P2P модел је *полуцентрализовани* модел архитектуре у којем један или више чворова могу функционисати као сервери коју подржавају комуникацију чиме се редукује количина саобраћаја између чворова у мрежи. Полуцентрализовани модел P2P архитектуре је приказан на слици 2.13.

У полуцентрализованој архитектури улога сервера (*super-peer*) је да помогне чворовима у успостављању међусобне конекције или да координира сложено процесирање. Чворови комуницирају са сервером да би открили остале чворове у мрежи (испрекидане линије на слици 2.13), а када су откривени сви чворови тада чворови који су повезани пуним линијама на слици 2.13 могу директно комуницирати без посредовања сервера.

P2P архитектуре омогућују ефикасну употребу ресурса и капацитета у мрежи. Међутим, то за последицу има проблем са безбедношћу и поверењем у друге кориснике. P2P подразумева да чвор дозволи директан приступ локалним ресурсима другим корисницима из мреже. То потенцијално значи да неко може приступити свим ресурсима на чвору, па је потребно посебну пажњу посветити питањима безбедности.

2.4 Софтвер као сервис

Клијент-сервер архитектуре имају основни проблем са великим оптерећењем сервера, нарочито у случајевима са танким клијентима. Савремени веб претраживачи значајно редукују овај проблем употребом веб технологија које омогућују ефикасно представљање веб садржаја (страница) и локално процесирање помоћу различитих скрипт језика (нпр. Asynchronous JavaScript And XML (AJAX) који је скуп веб технологија које се користе на клијентској страни). То значи да се веб претраживач конфигурише и користи као клијент са значајним уделом локалног процесирања. У том случају се апликативни софтвер посматра као удаљени сервис којем се може приступити са било ког уређаја који може покренути стандардни претраживач

интернета. Типични примери оваквих система су веб оријентисани клијенти електронске поште као што су *Yahoo!* и *Gmail*, или пословне канцеларијске апликацијее као што је *Google docs*. Модел софтвера као сервиса (Software as a Service (SaaS)) подразумева удаљено постављање софтверске апликације којој се потом приступа преко Интернета. Кључни елементи SaaS су:

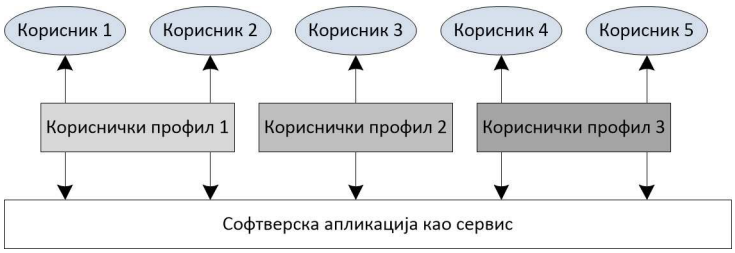
- Софтвер је распоређен на серверу, или групи сервера, и приступа му се помоћу веб претраживача.
- Власник софтвера је софтверска организација која га је произвела и обезбеђује му приступ (провајдер), док га друге организације или појединци само користе.
- Корисници плаћају употребу софтвера у зависности од одабраних сервиса. Ако је софтвер бесплатан тада најчешће корисници пристају на разне огласе помоћу којих се финансира развој и одржавање софтвера.

Користи од употребе SaaS за кориснике софтвера су: (1) трошкови управљања софтвером су потпуно пребачени на провајдера, (2) за одржавање софтвера - исправљање грешака (*fixing bugs*) и инсталирање допуна (*installing software upgrades*) - је одговоран провајдер, (3) одржавање хардвера и оперативног система који обезбеђују рад софтвера су одговорност провајдера, (4) нема трошкова лиценцирања софтвера, па се софтвер може користити на произвољном броју рачунара, и (5) софтверу се може приступити са мобилних уређаја са било које локације у свету.

Недостаци употребе SaaS за кориснике софтвера су: (1) трошкови преноса података према удаљеном сервису, који зависе од брзине и пропусне моћи комуникационе инфраструктуре, (2) време преноса података је лимитирано пропусном моћи мрежне инфраструктуре, (3) нема контроле еволуције софтвера пошто провајдер може мењати софтвер према својим потребама, и (4) проблеми са законом и регулативама (различите земље имају различите законе и регулативе).

Концепт SaaS је уско повезан са Service-Oriented Architecture (SOA) која представља приступ у структурирању софтверског система као скупа одвојених сервиса који не зависе од стања података (*stateless services*). Овакви сервиси могу бити дистрибуирани и може их понудити више провајдера. Трансакције су типично врло кратке. SaaS је начин испоруке софтверских функционалности, док је SOA скуп имплементационих технологија. Ако је SaaS имплементиран помоћу SOA тада апликације могу користити сервисне апликативне програмерске интерфејсе (Application Programming Interface (API)) да приступе функционалностима других апликација. На тај начин се креирају комплексни системи, тзв. ***mashups*** системи, који представљају другачији приступ брзом развоју и поновној употреби софтвера (употреба садржаја из више извора да би се креирао и обезбедио сервис).

Захтеви за пројектовање софтвера као сервиса најчешће су базирани на представи провајдера софтвера шта би било корисно корисницима софтвера, а не на њиховим стварним захтевима. Због тога овакви софтвери морају брзо еволуирати када се добију повратне информације од корисника софтвера о стварној употреби. Због тога се овде најчешће користе агилне методе развоја



Слика 2.14: *Конфигурација софтверског система као сервиса*

са инкременталном испоруком софтвера. Због потенцијално веома широког спектра корисника из различитих окружења (организација) потребно је водити рачуна о:

- **Конфигурабилност** (*Configurability*) - како конфигурисати софтвер с обзиром на специфичности захтева различитих организација које га користе.
- **Подршка за више корисника који су организације** (*Multi-tenancy*) - како обезбедити да сваки корисник има утисак да ради са личном копијом софтвера, а да при томе има ефикасно коришћење дељених ресурса.
- **Скалабилност** (*Scalability*) -како дизајнирати систем да се он може проширити да задовољи потребе непредвиђено великог броја корисника.

Конфигурабилност треба уградити у софтвер приликом пројектовања тако да се обезбеди конфигурациони интерфејс који омогућује корисницима да прилагоде софтвер својим потребама. То омогућује динамичко конфигурисање понашања софтвера током употребе. Корисницима се омогућује да изграде свој кориснички профил који им омогућује да сервисе користе на специфичан и њима одговарајући начин, што је приказано на слици 2.14.

Конфигурисање софтвера треба да обезбеди следеће опције:

- **Брендирање** (*Branding*) - омогућује да се корисницима из различитих организација прикажу интерфејси који рефлектују специфичности тих организација.
- **Пословна правила и токови** (*Business rules and workflows*) - омогућује да свака организација дефинише и управља употребом софтвера и података на специфичан начин.
- **Проширења база података** (*Database extensions*) - свака организација може да дефинише како се основни генерички модел података може проширити и прилагодити њеним специфичним потребама.
- **Контрола приступа** (*Access control*) - свака организација може да управља приступом сервисима за своје чланове.

Подршка за више корисника на систему (*Multi-tenancy*) омогућује различитим корисницима и корисничким групама да приступе систему.

Архитектура система треба при томе да обезбеди ефикасно дељење системских ресурса, а да сваки корисник има утисак да само он користи системске ресурсе. То подразумева дизајн система у којем су потпуно одвојене системске функционалности од података у систему, што значи да операције у систему не зависе од стања података (*operations are stateless*). За управљање подацима у оваквим системима су најпогодније релационе базе података. Обично систем има једну заједничку базу података, а сваки корисник користи виртуелно изоловану базу (јасна идентификација корисника у сваком тренутку).

Приликом дизајна система треба водити рачуна о следећим аспектима скалабилности:

- Све компоненте треба пројектовати као сервисе који немају стање (*stateless service*) и који се могу покренути на било ком серверу.
- Систем треба да има подршку за асинхрону интеракцију тако да компоненте не морају чекати на резултате других компоненти, већ увек могу да ефикасно раде без чекања.
- Управљање свим ресурсима на јединствен начин, што обезбеђује да се не може десити да неки сервер остане без ресурса.
- Базу података треба пројектовати тако да се обезбеди "фино" закључавање које ће омогућћити да се закључавају само они делови који се директно користе (нпр. само део записа).

Поглавље 3

Софтверско инжењерство базирано на компонентама

Софтверско инжењерство базирано на компонентам (Component-Based Software Engineering (CBSE)) се почело развијати крајем 90-тих година прошлог века као приступ развоју софтвера базиран на компонентама које се могу више пута употребљавати (*based on reusing software components*). Основни разлог за развој и прихватање CBSE је сагледавање недостатака објектно-оријентисаног развоја софтвера са аспекта поновне употребе конструката који настају током развоја софтвера.

Софтверске компоненте се дефинишу помоћу интерфејса преко којих им се приступа и имају виши ниво апстракције од објеката. Компоненте су обично комплексније и веће од објеката и њихова унутрашња имплементација је скривена од осталих компоненети у систему које јој приступају само преко дефинисаног интерфејса. CBSE је скуп процеса које се односе на дефинисање, имплементацију и интеграцију слабо повезаних независних компоненети у систем. Основни принципи CBSE су:

- Независне компоненте се специфицирају помоћу интерфејса, при чему постоји јасно раздвајање интерфејса од имплементације компоненте.
- Интеграција компоненти у софтверске системе је подржана стандардима који су уграђени у моделе компоненти. Стандарди дефинишу како треба да изгледају интерфејси компоненти и начин комуникације између компоненти.
- Постоји средњи слој који омогућује комуникацију компоненти. Средњи слој поред тога пружа подршку за алокацију ресурса, управљање трансакцијама, безбедност и конкурентност.
- Процес развоја софтвера подржава рад са компонентама, тј. мапирање софтверских захтева у софтверске компоненте. На тај начин се добија софтвер који се лако одржава.

Слика 3.1: *Интерфејси софтверске компоненте*

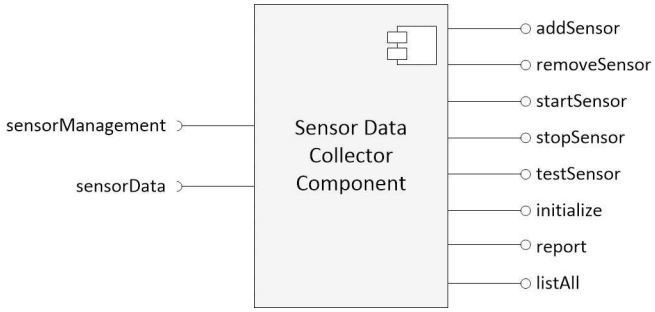
Стандарди који су развијени са циљем да обезбеде подршку за CBSE су CORBA, EJB и .NET. Основни проблем са овим стандардима је да компоненте које се развијају на различитим платформама, ка што су EJB и .NET не могу директно да комуницирају (сарађују). Као правац развоја који је требало да реши овај проблем појављује се развој и употреба софтвера као сервиса (Software as a Service (SaaS)).

3.1 Софтверске компоненте

Најопштије посматрано софтверска компонента је независна софтверска јединица која се комбинује са другим компонентама у циљу креирања софтверског система. Основне карактеристике софтверских компоненти према Сомервилу (2011) су:

- **Стандардизованост** (*Standardized*) - компонента је имплементирана у складу са дефинисаним стандардом за дефинисање и имплементацију компоненти, што подразумева: дефинисање интерфејса, метаподатке о компоненти, документацију, композицију и распоређивање компоненти.
- **Независност** (*Independent*) - компонента се може пројектовати и распоредити независно од других компоненти.
- **Композитност** (*Composable*) - све интеракције компоненте са окружењем се одвијају преко јасно дефинисаног интерфејса, при чему компонента јасно назначава своје јавно доступне делове.
- **Распоређивање** (*Deployable*) - компонента се испоручује као самостална и спремна за извршавање у систему где се интегрише без додатног компајлирања.
- **Документованост** (*Documented*) - детаљно документовање компоненте помаже у њиховој употреби од стране независних корисника.

Имајући у виду наведене карактеристике, компонента се може посматрати као понуђач специфичног скупа услуга са следећим особинама: (1) компонента је независна и извршива, па се може користити без познавања њеног кода, и (2) све сервисе компонента обезбеђује помоћу интерфејса. Компонента има два типа интерфејса који указују на сервис које компонента нуди (*provides interface*) и на сервисе који су потребни компоненти да би исправно функционисала (*requires interface*), што је приказано на слици 3.1.



Слика 3.2: *Софтверска компонента која прикупља податке од скупа сензора*

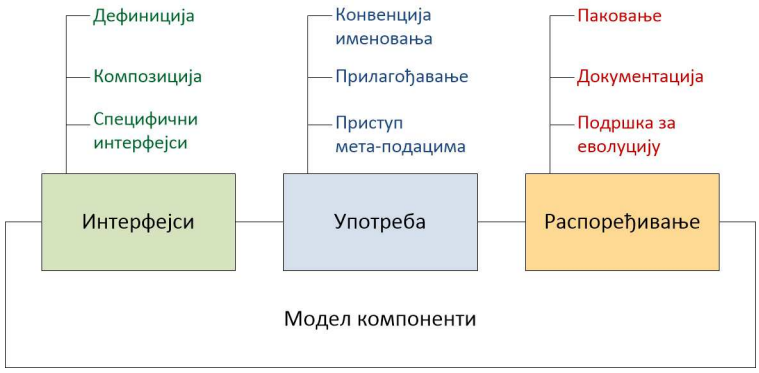
На слици 3.2 је приказан модел компоненте која опслужује скуп сензора. Компонента прикупља податке од сензора и пружа обрађене податке другим компонентама које то захтевају.

3.2 Модели софтверских компоненти

Модели софтверских компоненти дефинишу стандарде за развој и имплементацију, документовање и распоређивање софтверских компоненти. Ови стандарди треба да обезбеде да компоненте које су пројектоване у складу са препорукама стандарда могу сарађивати у оквиру система. најзначајнији модели за компоненте су CORBA, EJB и .NET.

Основни елементи модела компоненти, приказани на слици 3.3 су:

- **Интерфејс** (*Interface*). Модел указује како се специфицира интерфејс, и како се у дефиницију интерфејса укључују називи операција, параметри и изузетци. Модел такође специфицира језик за дефинисање интерфејса компоненти. Примери језика за дефинисање интерфејса су Web Services Description Language (WSDL) за веб сервисе, *Java* за EJB компоненте, или Common Intermediate Language (CIL) за .NET компоненте. Понекад је потребно дефинисати и спеецифичне интерфејсе за компоненте и начин њихове композиције у ДСС.
- **Употреба** (*Usage*). Употреба компоненти подразумева да компонента има јединствено име или идентификатор за руковање (*handle*). На пример, EJB компоненте имају имена базирана на хијерархији домена где су лоциране, док сервиси имају јединствени идентификатор ресурса (Uniform Resource Identifier (URI)). Мета-подаци о компоненти су подаци о самој компоненти и они се користе да би се разумело које сервисе компонента пружа или захтева (нпр. то се у компонентама писаним у језику Java ради применом *reflection interface-a*). Такође је важно да модел опише како се компоненте прилагођавају специфичним окружењима.
- **Распоређивање** (*Deployment*). Модел укључује и спецификацију како се компоненте пакују и распоређују као независни извршиви објекти.



Слика 3.3: *Основни елементи модела компоненти*

Приликом распоређивања компоненте је потребно испоручити и документацију која помаже у конфигурисању и употреби. Начин реализације промена које подразумевају замену компоненте новијом верзијом такође је специфициран моделом, чиме се обезбеђује поуздан рад ДСС независно од актуелне верзије компоненте.

Сервиси које софтверским компонентама обезбеђује средњи софтверски слој (*middleware*) се могу поделити у две категорије (слика 3.4):

- **Платформски сервиси** (*Platform services*). Омогућују компонентама да комуницирају и сарађују у дистрибуираном окружењу. Поред тога омогућују управљање изузетцима.
- **Сервиси подршке** (*Support services*). Ово су уобичајени сервиси које користе компоненте, чиме се убрзава развој софтвера и избегавају потенцијални проблеми због компатибилности компоненти. Ови сервиси омогућују управљање ресурсима, компонентама и трансакцијама, а такође су задужени за конкурентност, безбедност и перзистентност софтверских система базирани на компонентама.

Средњи софтверски слој имплементира сервисе компоненти па се може посматрати као **контејнер за распоређивање компоненти** (*component container*). Контејнер обезбеђује имплементацију сервиса подршке и дефинише интерфејсе које компоненте морају имплементирати да би се могле интегрисати у контејнеру. Укључивање компоненте у контејнер омогућује компоненти да приступи сервисима подршке које пружа контејнер, али и контејнеру да приступи интерфејсу компоненте. Компоненте које су у контејнеру не комуницирају директно већ уз помоћ контејнера који позива компоненте преко њихових интерфејса. Поједини контејнери имплементирају одређене заједничке сервисе, али врло често апликације користе специфичне (наменски развијене) библиотеке које имплементирају неке сервисе као што су безбедносни сервиси или сервиси за управљање трансакцијама.

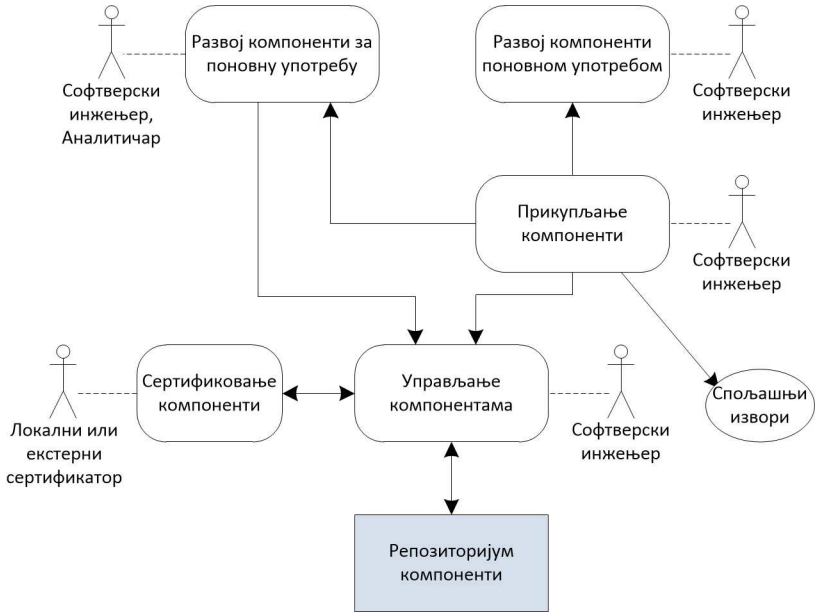
Слика 3.4: *Сервиси средњег софтверског слоја у моделу компоненти*

3.3 Процеси у софтверском инжењерству базираном на компонентама

Процеси у софтверском инжењерству базираном на компонентама (CBSE processes) су процеси који подржавају развој и еволуцију софтверских система базираних на компонентама. Ови процеси се односе на могућност поновне употребе компоненти и различите активности у процесу развоја компоненти које се могу поновно користити. Преглед процеса у софтверском инжењерству базираном на компонентама је приказан на слици 3.5.

Процеси који се појављују у софтверском инжењерству базираном на компонентама су:

- **Развој компоненти за поновну употребу** (*Development for reuse*). Односи се на развој компоненти и сервиса које ће користити друге апликације. Врло често се базира на механизму генерализације већ постојећих компоненти. Изворни код компоненти је доступан током развоја.
- **Развој компоненти поновном употребом** (*Development with reuse*). Односи се развој нових апликација употребом већ постојећих компоненти. Да би се користиле већ постојеће компоненте, треба их пронаћи и употребити их на најбољи могући начин. При томе се обично нема приступ коду одабраних постојећих компоненти.
- **Прикупљање компоненти** (*Component acquisition*). Односи се на прикупљање компоненти које се могу користити у развоју нових компоненти или у развоју апликација. Компоненте се могу пронаћу у локлном репозиторијуму или у спољашњим изворима (нпр. доступне на сајтовима произвођача).
- **Управљање компонентама** (*Component management*). Обезбеђује ефикасно управљање развојем, складиштењем и поновном употребом



Слика 3.5: *Процеси у софтверском инжењерству базираном на компонентама*

компоненти. Такође обезбеђује доступност компоненти другим софтверским организацијама и појединцима.

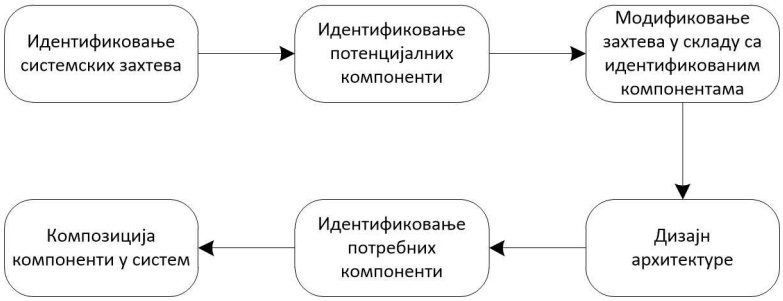
- **Сертификовање компоненти** (*Component certification*). Односи се на проверу да ли компонента одговара спецификацији према којој је дизајнирана.

У репозиторијуму се налазе све компоненте које је организација произвела, заједно са упутствима за њихову употребу.

3.3.1 Развој компоненти за поновну употребу

Развој компоненти за поновну употребу је развој који обезбеђује да софтверске компоненте постану доступне за поновну употребу путем система за управљање компонентама. Овај приступ се најчешће користи у оквиру развоја у софтверским организацијама, а релативно мали број компоненти је доступан јавно. У највећем броју случајева поновна употреба захтева модификацију и прилагођење компоненти специфичним потребама новог развоја. Развој компоненти за поновну употребу подразумева да компоненте имају генеричке особине које се могу прилагодити специфичним потребама. Да би компонента заиста била поновно употребљива потребно је:

- Избацити све методе специфичне за неку апликацију (примену).
- Имена изменити да постану генерална.
- Додати методе које омогућују комплетно покривање функционалности.



Слика 3.6: *Процес развоја базиран на поновној употреби постојећих софтверских компоненти*

- Обезбедити конзистентно управљање изузетцима у свим методама. Компонента не треба сама да обрађује изузетке, већ да дефинише који се изузетци могу појавити и да их достави на специфичном интерфејсу за даљу обраду.
- Додати конфигурациони интерфејс који омогућује адаптирање компоненте различитим ситуацијама.
- Интегрисање потребних компоненти да би се повећала независност компоненти.

Поновна употреба компоненте зависи од домена њене употребе и обезбеђених функционалности. Поновна употребљивост се повећава ако се компонента учини генералнијом, али то подразумева да она има више операција и да је комплекснија, па је због тога и теже разумети њено функционисање и употребу. Са друге стране, употребљива компонента има мали и лако разумљиви интерфејс, што је у контрадикцији са потребом за развојем генералних и комплексних компоненти за поновну употребу. У развоју софтверских компоненти треба пронаћи баланс између особина *употребљивости* и *поновне употребе*.

3.3.2 Развој компоненти поновном употребом

Процес развоја софтвера базиран на употреби постојећих компоненти треба да обезбеди проналажење одговарајућих компоненти и њихову интергацију. Процес развоја базиран на поновној употреби софтверских компоненти је приказан на слици 3.6.

Основне разлике између процеса развоја базираног на употреби компоненти у односу на стандардни процес развоја софтвера су:

- Спецификација софтверских захтева је флексибилна, што значи да су захтеви специфицирани оквирно и без детаља. Сувише специфични захтеви постављају ограничења за компоненте које треба да их реализују. Скуп захтева треба да је комплетан да би се идентификовале све могуће компоненте за њихову имплементацију.

Слика 3.7: *Процес идентификације компоненти*

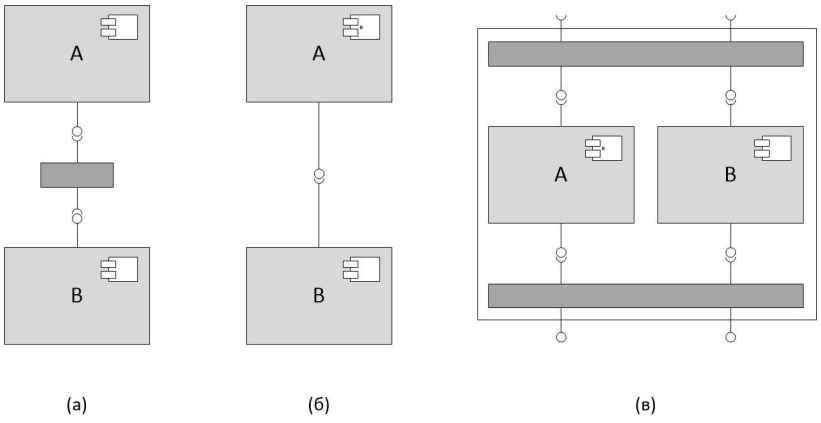
- Захтеви се дефинишу на основу доступних компоненти. Ако се захтеви не могу реализовати доступним компонентама, разматрају се и додатни захтеви, што подразумева да корисници морају бити флексибилни по питању спецификације захтева.
- Након дизајна софтверске архитектуре поново се траже и преиспитују одговарајуће компоненте. Компоненте које су у почетку биле употребљиве могу постати неодговарајуће, што може захтевати нове измене захтева. Процес идентификације одговарајућих компоненти је приказан на слици 3.7.
- Развој се посматра као процес композиције идентификованих компоненти, што подразумева интегрисање компоненти према дефинисаном моделу компоненти. Врло често се морају креирати **компоненте адаптери** (*adapters*) које разрешавају некомпатибилности између интерфејса

Током дизајна архитектуре се бира модел компоненти и имплементациона платформа (нпр. .NET или EJB). С обзиром да платформе имају своје већ предвиђене моделе компоненти, то поставља ограничење у развоју софтвера.

Тражење и избор компоненти подразумева претрагу поузданих извора, тј. испоручиоца компоненти. Први корак је да се претражи локални репозиторијум организације која производи софтвер, пошто компаније обично праве базу својих компоненти што смањује ризик у односу на употребу компоненти од других произвођача. Ако су потребне додатне компоненте, оне се траже на Интернету у добро познатим библиотекама као што су Sourceforge или Google Code, где се може видети и изворни код компоненти. Провера, или валидација одабраних компоненти је потребна да би се увидело да ли се понашају како је наведено у спецификацији. Може се тестирати сама компонента и њена интеграција у систем. За то се могу развити посебни тестови. Провера треба да утврди и да компонента не садржи злонамеран код (*malicious code*) или непотребне функционалности

3.4 Композиција софтверских компоненти

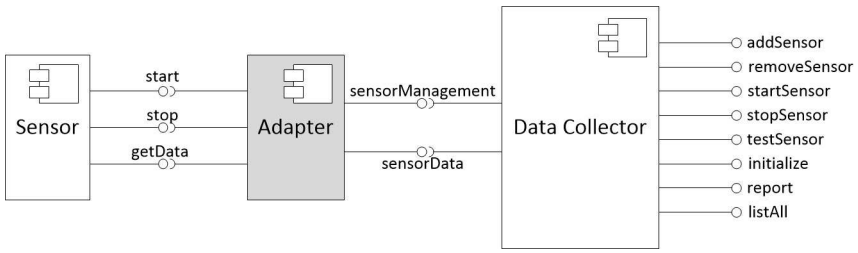
Композиција компоненти је процес интегрисања софтверских компоненти у систем или другу компоненту директно или помоћу "додатног везивног кода" (*glue code*). Начини композиције компоненти су приказани на слици 3.8. Ако претпоставимо да се врши композиција две компоненте **A** и **B**, тада се на слици 3.8 могу уочити следећи типови композиције компоненти:



Слика 3.8: *Типови композиције софтверских компоненти*

- **Секвенцијална композиција** (*sequential composition*) **[слика 3.8 (а)]**. Нова компонента се креира тако што се постојеће компоненте позивају у секвенци, при чему се врши композиција позива интерфејса које пружају компоненте (*provides interfaces*). Прво се позив сервис који пружа компонента **A** и резултат се користи при позиву сервиса који пружа компонента **B**. У секвенцијалној композицији се компоненте не позивају међусобно, већ је потребан додатни везивни код (*glue code*) који позива компоненте у одређеном редоследу и обезбеђује компатибилност резултат који се прослеђују приликом позива различитих компоненти.
- **Хијерархијска композиција** (*hierarchical composition*) **[слика 3.8 (б)]**. Композиција се односи на директни позив сервиса једне компоненте од стране друге компоненте. Позвана компонента обезбеђује сервис који тражи друга компонента позиваоц. То значи да интерфејс за пружање сервиса (*provides*) позване компоненте мора бити компатибилан са интерфејсом за захтевање сервиса (*requires*) компоненте која позива сервис. Компонента **A** директно позива компоненту **B**, и ако су им интерфејси усаглашени није потребан додатни код. Овај тип композиције се не користи код веб сервиса.
- **Адитивна композиција** (*additive composition*) **[слика 3.8 (в)]**. Нова компонента се креира комбиновањем сервиса (функционалности) које пружају компоненте које се комбинују, што значи да се интерфејси нове компонент добијају комбиновањем интерфејса компоненти које се укључују у нову компоненту. Укључене компоненте се позивају независно једна од друге, у зависности чији је сервис позван. Укључене компоненте **A** и **B** су независне и међусобно се не позивају.

Приликом пројектовања софтверских система или софтверских компоненти могуће је користити различите типови композиције у истом пројекту (нпр. неке компоненте повезати секвенцијално, а неке хијерархијски). Када се дизајнирају нове компоненте, тада је могуће пројектовати интерфејсе компоненти тако да буду подржани различити типови композиције компоненти (нпр. пројектовање које обезбеђује компатибилност података или



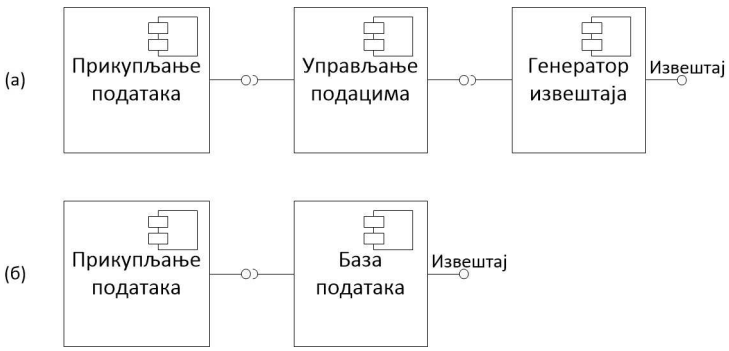
Слика 3.9: *Примена адаптерске компоненте за повезивање сензорске компоненте и компоненте за прикупљање података*

резултата који се добијају на интерфејсима компоненти). Много више проблема се појављује када композицију треба радити са компонентама које се добављају из других извора (компоненте које су произвеле друге софтверске организације), и тада се јавља потреба за писањем више додатног везивног кода којим се решавају проблеми компатибилности компоненти. Проблеми некомпатибилности интерфејса компоненти су:

- **Некомпатибилност параметара** (*Parameter incompatibility*). Операције на интерфејсима имају исти назив али се разликују у броју и типу параметара.
- **Некомпатибилност операција** (*Operation incompatibility*). Називи операција у интерфејсима се разликују.
- **Некомплетност операција** (*Operation incompleteness*). Операција која захтева сервис позива операцију која задовољава само део потреба позиваоца пошто не пружа све захтеване сервисе, или операција која захтева сервис не може да позове и искористи све што нуди интерфејс позване компоненте.

У сва три наведена случаја се обично пише компонента адаптер (*adapter*) која прилагођава операције и њихове параметре у интерфејсима компоненти које комуницирају. Пример адаптерске компоненте (*Adapter*) за повезивање компоненте која повезује сензорску компоненту (`Sensor`) и компоненту за прикупљање података (`Data Collector`) је приказан на слици 3.9. Адаптер обезбеђује форматирање података које добија од сензора тако да одговарају компоненти која прикупља податке, а такође и форматирање управљачких сигнала које компонента за прикупљање података шаље сензорској компоненти.

Поред компатибилности назива операција и броја и типа параметара, важно је да се обезбеди и семантичка компатибилност интерфејса компоненти које се комбинују у сложенију компоненту. Такође је важно решити потенцијалне конфликте између функционалних и нефункционалних захтева, и обезбедити подршку за еволуцију система због промене захтева. Пример конфликтних захтева при дизајну система је илустрован са две опције композиције система приказане на слици 3.10. Прва опција (а) је комплекснија али је флексибилнија за будуће промене због декомпозиције функционалности у различитим компонентама, док је опција (б) бржа и



Слика 3.10: *Пример две опције у композицији компоненти*

поузданија због једноставнијег дизајна, а такође је једноставнија и бржа за имплементацију.

Основно правило у дизајну базираном на композицији компоненти је да свака компонента има јасно дефинисану намену и да се избор компоненти врши на основу постављених захтева. При томе је важно наћи баланс између перформанси, једноставности и цене развоја.

Поглавље 4

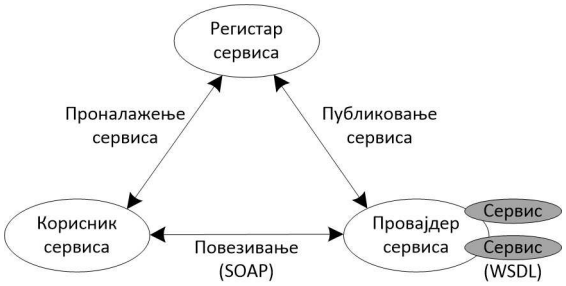
Сервисно оријентисане архитектуре

Сервисно оријентисане архитектуре (Service-Oriented Architecture (SOA)) представљају начин пројектовања дистрибуираних система у којима су компоненте независни и самостални сервиси који се могу извршавати на географски дистрибуираним рачунарима. Комуникација између сервиса користи систем порука базиран на XML-у, и не зависи од програмског језика и оперативног система. Основна одлика сервиса је да је пружање (издавање, публиковање) сервиса независно од његове примене. Провајдер сервиса развија специфичне сервисе и пружа их на услугу различитим корисничким организацијама и појединцима. Софтверски системи базирани на сервисима се конструишу композицијом локалних и екстерних сервиса различитих провајдера, при чему се посебна пажња посвећује интеракцији између сервиса. Размена XML порука између сервиса се може реализовати на један од следећа три начина: (1) XML Remote Procedure Calls (XML-RPC), (2) Standard Object Access Protocol (SOAP), и (3) Hypertext Transfer Protocol (HTTP) GET/POST при чему се шаљу XML документи.

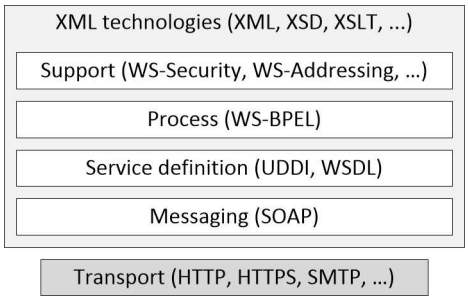
Основни принцип SOA је приказан на слици 4.1. Провајдер сервиса имплементира сервисе и специфицира њихове интерфејсе, и потом публикује информације о сервисима у јавно доступном регистру сервиса. Корисници сервиса откривају спецификацију сервиса у регистру сервиса и лоцирају провајдера сервиса. Након тога корисници могу своје апликације повезати са одабраним сервисима применом стандардних сервисних протокола.

Корисници сервиса су софтверске компоненте или системи који комуницирају са сервисом путем порука, али исто тако могу бити и други сервиси. Једини услов за кориснике сервиса је да се усагласе са интерфејсом (уговором) који дефинише употребу сервиса.

Поред технолошког развоја, значајан утицај на примену сервиса има и процес развоја стандарда и у области хардвера и у области софтвера. Циљ развоја стандарда је да се произвођачи хардвера и софтвера ускладе због све већ броја технолошких иновација и компатибилности различитих верзија



Слика 4.1: *Основни принцип сервисно оријентисане архитектуре*



Слика 4.2: *Стандарди у пројектовању и имплементацији веб сервиса*

производа. Скуп стандарда који обезбеђују подршку развоју веб сервиса је приказан на слици 4.2.

Сви аспекти SOA су покривени протоколима за веб сервисе, почевши од основних механизама размене информација помоћу SOAP, до стандарда за језике за програмирање веб сервиса (Web Services Business Process Execution Language (WS-BPEL)). Сви ови стандарди су базирани на XML-у, који је језик за дефинисање структуре података, тако што се текст *тагује* идентификаторима који имају смислено значење. XML је подржан разним технологијама, као што је на пример шема за дефинисање структуре XML докумената (XML Schema Definition (XSD)). Кључни стандарди за архитектуру веб сервиса су:

- **Standard Object Access Protocol (SOAP)**. Стандард који обезбеђује комуникацију између сервиса базирану на размени порука. Стандард дефинише есенцијалне и опционе елементе порука које се прослеђују између сервиса.
- **Web Services Description Language (WSDL)**. Стандард за дефинисање интерфејса веб сервиса. Стандардом се дефинишу операције интерфејса (назив, параметри и њихови типови), као и начин повезивања сервиса.
- **Web Services Business Process Execution Language (WS-BPEL)**. Стандард који дефинише језик за дефинисање процеса програма који укључује више различитих сервиса.

Стандард за откривање сервиса (Universal Description, Discovery and Integration (UDDI)) дефинише компоненте спецификације сервиса које се

могу искористити за откривање и идентификацију сервиса. Информације које се користе се односе на провајдера сервиса, сервис, локацију WSDL описа интерфејса сервиса и информације о пословним релацијама. Намена овог стандарда је да омогући организацијама како да поставе регистре сервиса са UDDI описима сервиса које пружају.

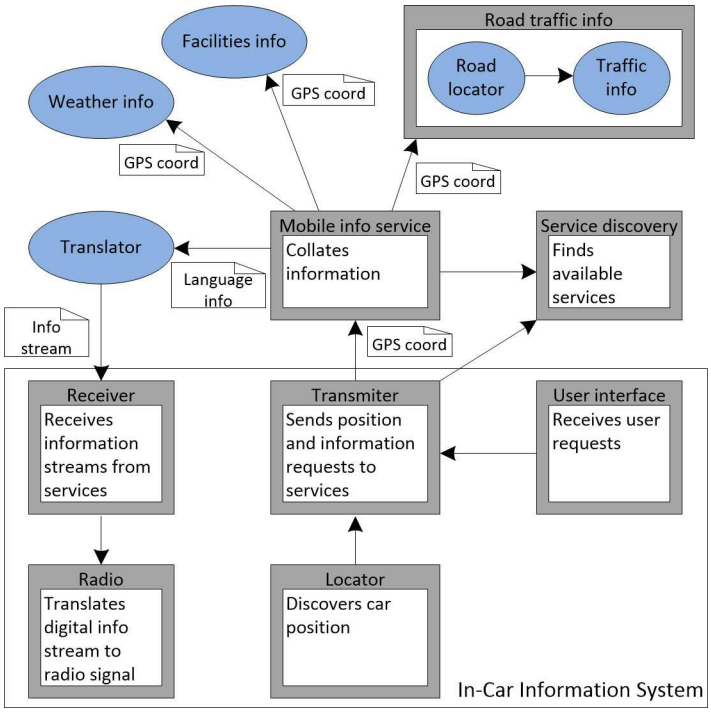
Основни SOA стандарди су подржани значајним бројем додатних стандарда који су фокусирани на специфичне аспекте SOA у различитим пословним применама. Примери таквих стандарда су:

- **WS-Reliable Messaging** - стандард који обезбеђују поуздану испоруку порука (свака порука се испоручује само једном).
- **WS-Security** - скуп стандарда који обезбеђује сигурност сервиса, као што су стандарди за безбедносна правила или стандрди за дигитални потпис.
- **WS-Addressing** - стандарди који дефинишу како ће се адресне информације представљати у SOAP порукама.
- **WS-Transactions** - стандарди који дефинишу координацију трансакција у окружењу са дистрибуираним сервисима.

Текући стандарди за веб сервисе се посматрају као тешки (*heavyweight*), превише генерални и неефикасни за употребу. Имплементација тих стандарда захтева значајно вереме процесирања да би се XML порука креирала, пренела и интерпретирала. Због тога су многе организације почеле да користе једноставније и ефикасније приступе у комуникацији сервиса, као што су **RESTful** сервиси који имају подршку за ефикасну интеракцију сервиса, али немају подршку за својства на пословном нивоу као што су *WS-Reliability* и *WS-Transactions*.

Изградња софтверских апликација базираних на сервисима омогућује организацијама да сарађују и међусобно користе услуге које пружају. Због тога се сервисно оријентисане апликације примењују у размени информација које превазилази оквире организација (нпр. системи испоруке). Апликације базиране на сервисима се могу креирати повезивањем сервиса различитих организација или провајдера применом стандардних језика за програмирање и наменских језика за опис токова сервиса. Овакве архитектуре су слабо повезане што значи да се повезивање сервиса може мењати током употребе. Овакве сложене апликације могу поред веб сервиса интегрисати и локалне апликације у организацијама. Пример овако сложеног система који повезује веб сервисе и локалне софтверске компоненте је информациони систем у аутомобилу (слика 4.3).

Софтверски систем у аутомобилу има модуле за комуникацију са возачем, са GPS пријемником и радиом, као и модуле који комуницирају са екстерним сервисима (време, стање на путевима и локални објекти у региону). Софтвер у аутомобилу открива ове сервисе у региону где се налази и обезбеђује одговарајуће информације возачу. Овај пример илуструје једну од значајних предности система базираних на сервисима: приликом пројектовања софтвера није потребно знати информације о свим доступним сервисима, већ ће сервиси бити идентификовани током употребе софтвера.



Слика 4.3: *Информациони систем у аутомобилу као комбинација софтверских компоненти и веб сервиса [Sommerville2011]*

Развој софтвера базиран на сервисима је нова парадигма у софтверском инжењерству и подразумева употребу разних савремених технологија као што су *cloud* системи, где се сервиси нуде на постојећој инфраструктури коју обезбеђују провајдери сервиса (нпр. *Google* и *Amazon*).

Примена SOA доноси значајне користи у примени и интеграцији дистрибуираних софтверских система, као што су:

- **Поновна употреба** (*Reusability*) - једном пројектовани сервиси се могу интегрисати више пута.
- **Адаптивност** (*Adaptability*) - изолација интерне структуре сервиса и интерфејса омогућује ефикасније руковање и прилагођавање променама.
- **Лако одржавање** (*Maintainability*) - ефикасно одржавање јасно дефинисаних сервиса као независних елемената и у склопу сложених система.

4.1 Сервиси као компоненте

Сервис се може посматрати као софтверска компонента (поглавље 3) при чему се модел компоненти посматра као скуп стандарда за развој веб сервиса. На основу тога се могу увести следеће дефиниција сервиса и веб сервиса:

> ***Сервис*** *је слабо повезана софтверска компонента која се може поново употребити, енкапсулира дискретне функционалности, може се дистрибуирати и програмски јој се може приступити.*
>
> ***Веб сервис*** *је сервис који се може користити применом стандардних Интернет и XML протокола.*

Основна одлика сервиса је да се увек извршава на исти начин без обзира на окружење у којем се извршава. За разлику од класичних софтверских компоненти, сервиси немају интерфејс који дефинише потребе сервисе (*requires interface*), што је последица међусобне независности сервиса.

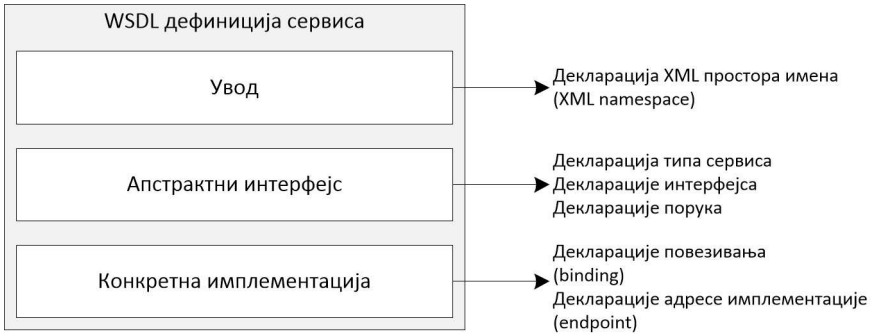
Сервиси комуницирају помоћу XML порука које размењују. Поруке се преносе стандардним Интернет транспортним протоколима као што су HTTP и TCP/IP. Сервиси своје потребе дефинишу у порукама које шаљу другим сервисима. Сервис који прима поруку је најпре парсира, извршава потребне операције, и на крају враћа одговор поново као поруку. Сервис који прима поруку са одговором, парсира примљену поруку и вади потребне информације. сервиси не користе позиве удаљених процедура да би приступили функционалностима других сервиса.

Употреба сервис подразумева да се зна где се сервис налази (URI) и да се знају детаљи његовог интерфејса, што је описано помоћу XML базираног језика WSDL. WSDL опис сервиса се састоји из три дела:

- **"Шта део"** се назива интерфејс сервиса (*interface*), а служи за спецификацију операције сервиса и дефинисање формата порука које може да прима и шаље.
- **"Како део"** се назива повезивање сервиса (*binding*), а служи за мапирање апстрактног интерфејса у конкретан скуп протокола (техничка спецификација како комуницирати са сервисом).
- **"Где део"** описује локацију специфичне имплементације сервиса (*endpoint*).

На слици 4.4 је приказан WSDL концептуални модел који приказује елементе описа сервиса. WSDL спецификација се састоји од три дела који могу бити у одвојеним датотекама, а садржи:

- Уводни део са декларацијом простора имена за сервис.
- Опциони опис типова који се користе у порукама које сервис размењује.
- Опис интерфејса, тј. операције које сервис пружа.
- Опис улазних и излазних порука сервиса.
- Опис повезивања сервиса, тј. комуникационог протокола за комуникацију порукама. Подразумева се SOAP.



Слика 4.4: *Организација WSDL спецификације сервиса*

- Опис физичке локације сервиса (*endpoint*) на интернету као URI.

Комплетне дефиниције сервиса су обично дугачке и тешке за разумевање. Дефиниција простора имена може увести префикс у дефинисању елемената сервиса, што додатно отежава читљивост и разумљивост описа сервиса. Структура WSDL спецификације сервиса је приказана у листингу 4.1.

Листинг 4.1: *Структура WSDL спецификације сервиса*

```
<description xmlns="http://www.w3.org/2004/08/wsdl"
  targetNamespace="..." ...>

  <types>
    <!-- XML Schema description of types being used in messages -->
    ...
  </types>

  <interface name="...">
    <!-- list of operations and their input and output -->
    ...
  </interface>

  <binding name="..." interface="..." type="...">
    <!-- message encodings and communication protocols -->
    ...
  </binding>

  <service name="..." interface="...">
    <!-- combination of an interface, a binding,
    and a service location -->
    ...
  </service>

</description>
```

Основни проблем са WSDL спецификацијом сервиса је да не укључује информације о семантици сервиса или о не-функционалним карактеристикама (перформансе, зависности), већ представља само једноставан опис **потписа**

Слика 4.5: *Процес инжењеринга сервиса*

сервиса. Смислена имена и документација су због тога неопходни да би се разумело функционисање сервиса.

4.2 Инжењеринг сервиса

Инжењеринг сервиса је процес развој сервиса за поновну употребу у сервисно оријентисаним софтверским системима. Сервиси се у том контексту посматрају као апстракције које се могу употребити у различитим системима. Три логичка стања у процесу инжењеринга сервиса, приказана на слици 4.5, су:

- Идентификација сервиса кандидата који могу бити имплементирани и спецификација њихових захтева.
- Дизајн сервиса, што обухвата логички физајн и WSDL спецификацију сервиса.
- Имплементација, испорука и распоређивање сервиса за употребу, након чега сервис постаје доступан корисницима.

4.2.1 Идентификација сервиса кандидата

Сервиси обезбеђују поршку пословним процесима, због чега идентификација сервиса кандидата укључује анализу пословних процеса да би се сагледало које сервисе је потребно имплементирати као подршку тим процесима. Основни типови сервиса су:

- **Услужни сервиси** (*Utility services*) имплементирају генералне функционалности које могу бити коришћене од стране различитих пословних процеса. Пример услужног сервиса је сервис за конверзију новца.
- **Пословни сервиси** (*Business services*) су сервиси придружени специфичним пословним функцијама. Пример пословног сервиса је регистрација студента за слушање курса.

- **Координирајући или процесни сервиси** (*Coordination or process services*) су сервиси који обезбеђују подршку генералнијим пословним процесима који укључују различите учеснике и активности. Пример координирајућег сервиса је сервис за постављање наруџбине, што укључује достављање наруџбине, прихватање услуге или робе по наруџбини и исплата.

Сервиси могу бити оријентисани ка задацима (*task-oriented*) или ентитетима (*entity-oriented*). Примери сервиса оријентисаних ка задацима и ентитетима су приказани у табели 4.1. Услужни и пословни сервиси могу бити оријентисани и ка задацима и ка ентитетима (пословним објектима), док су координирајући сервиси увек оријентисани ка задацима.

Табела 4.1: *Примери класификације сервиса*

	Услужни	**Пословни**	**Координирајући**
Задатак	Конверзија валута	Провера захтева	Захтев за трошкове процеса
	Локација запосленог	Валидација ранга	Плаћање добављачу
Ентитет	Провера стила документа	Форма за трошкове	
	Конверзија типа података	Форма за пријаву студената	

Идентификација сервиса треба да обезбеди идентификовање логички кохерентних и независних сервиса који се могу поново употребити. Селекцијом сервиса се идентификује скуп одговарајућих сервиса и њихове спецификације захтева. Функционални захтеви се односе на то што сервис ради, док се нефункционални захтеви односе на безбедност, различите перформансе и доступност сервиса.

4.2.2 Дизајн интерфејса сервиса

Дизајн интерфејса сервиса подразумева дефинисање операција које су придружене сервису и дефинисање параметара операција. Циљ је да се минимизира број порука потребних да се реализују операције које обезбеђује сервис. Сервиси немају стање, па је управљање спицифичним стањем сервисне апликације задатак корисника сервиса. То подразумева да се информације о стању прослеђују помоћу улазних и излазних порука сервиса. Стања у дизајну интерфејса сервиса су:

- **Логички дизајн интерфејса**. Обухвата идентификацију операција придружених сервису, улазе, излазе и изузетке операција.
- **Дизајн порука**. Обухвата детаљну спецификацију порука које прима и шаље сервис.
- **WSDL развој**. Обухвата трансформацију логичког дизајна и дизајна порука у опис апстрактног интерфејса написан помоћу WSDL-а.

Пројектанти сервиса у принципу не знају како ће сервиси бити коришћени, па је због тога добра пракса да се руковање изузецима остави да реализују корисници. Када се јасно дефинишу сервиси на логичком нивоу прелази се на детаљну спецификацију порука које сервис прима и шаље у

XML формату. Израда интерфејса у WSDL је компликован процес, а и сами WSDL описи су дугачки и често неразумљиви. Већина савремених развојних окружења, као што је на пример *Eclipse* (https://www.eclipse.org/) који се користи за развој сложених апликација на Јава платформи има алат за трансформацију логичкод дизајна интерфејса у WSDL спецификацију.

4.2.3 Имплементација и испорука сервиса

Последња фаза у процесу инжењеринг сервиса је имплементација. Имплементација укључује програмирање сервиса применом стандардних језика Java или C# уз употребу одговарајућих библиотека. Имплментација нових сервиса се такође може реализовати композицијом постојећих сервиса. Тестирање написаних сервиса се базира на изрди улазних порука и проверу да ли ће сервис дати одговарајуће излазне поруке. Савремена развојна окружења омогућују да се на основу WSDL спецификације сервиса генеришу тестови који се користе за проверу да ли интерфејс сервиса задовољава спецификацију.

Испорука и распоређивање сервиса за употребу је последња фаза процеса инжењеринг сервиса. Сервис се распоређује на сервере који имају једноставну процедуру за инсталирање сервиса. Извршна верзија сервиса се поставља у одговарајући фолдер на серверу, и потом се изврше одговарајућа подешавања да би се сервис пријавио серверу. Након тога је сервис аутоматски доступан за употребу, а потенцијалним корисницима треба доставити одговарајуће информације за ефикасну употребу сервиса, као што су:

- Информације о провајдеру сервиса које доприносе поверењу корисника. Корисници морају бити сигурни да сервис неће имати негативног утицаја на њихове активности.
- Неформални опис функционалности сервиса доприноси да корисници сагледају да ли сервис обезбеђује то што је њима потребно.
- Детаљан опис интерфејса и семантике.
- Информације о претплати на сервис и могућностима ажурирања сервиса.

Проблем са неформалним описом сервиса и његове семантике је неразумевање функционалности које сервис обезбеђује. Најбољи начин да се опише семантика сервиса је употреба описа базираних на онтологији. Онтологије обезбеђују стандардизацију терминологије и дефинишу релације између термина који се јављају у семантичком опису сервиса. Због тога се онтологије користе за опис сервиса свакодневним говорним језиком. За семантички опис веб сервиса се користи онтологија Web Ontology Language for Semantic Web (OWL-S), која је базирана на језику за публиковање и дељење онтологија на вебу Web Ontology Language (OWL).

Слика 4.6: *Приступ старом софтверском систему помоћу сервиса који чине омотач старог система*

4.3 Сервиси и софтверски системи у употреби

Значајан део софтвера који се користи су стари ("*проверени*") софтвери (*legacy systems*), базирани за застарелим технологијама и методама, али су есенцијални за пословање организација. Трошкови замене таквих система су често велики па они најчешће користе истовремено са другим савременијим софтверима. Једна од најзначајнијих употреба сервиса је да сервиси имплементирају **омотач** (*wrapper*) око постојећег софтвера и обезбеде приступ функционалностима и подацима старог софтвера помоћу нових технолошких решења (најчешће приступ из веб окружења).

Као пример се може посматрати компанија која одржава инвентар опреме и користи базу података да евидентира и прати одржавање и поправке опреме. Систем омогућује евидентирање и праћење захтева за одржавањем за различиту опрему, распоређивање послова одржавања, праћење утрошка времена у пословима одржавања итд. Овакав софтвер омогућује креирање листе послова одржавања за запослене у одржавању опреме. Овакви системи се најчешће пројекту са клијент/сервер архитектуром. Ако компанија жели да обезбеди приступ систему са удаљених локација, потребно је проширити систем новим функционалностима. Најефикасније решење је да се направи скуп сервиса који чине **омотач** око постојећег софтвера, а обезбеђују и нове функционалности, као што је приказано на слици 4.6. Друге софтверске апликације могу сада размењивати поруке са сервисима и на тај начин приступити старом софтверском систему. Сервиси су на слици означени применом нотације стереотипа из Unified Modeling Language (UML) језика.

Сервиси који су додати старом софтверском систему приказаном на слици 4.6 су:

- **Сервис одржавања** (*maintenance service*) обезбеђује управљање пословима одржавања опреме, као што су покретање, суспендовање и комплетирање послова.

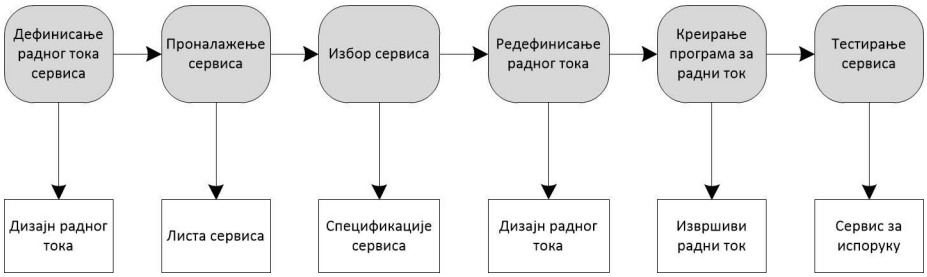
- **Функионалности везане за опрему** (*facilities service*) омогућују управљање информацијама о опреми.
- **Сервис евидентирања** (*logging service*) омогућује управљање захтевима за одржавањем опреме, што укључује пријаву нових захтева, брисање постојећих захтева и упит о стању захтева који су евидентирани.

4.4 Развој софтвера са сервисима

Развој софтвера са сервисима се односи на компоновање и конфигурисање сервиса у нови композитни сервис. Полазни сервиси могу бити пројектовани у оквиру организације или добављени од других произвођача сервиса. Тренд у развоју је превођење стандардних пословних апликација у апликације базиране на сервисима, интеграција сервиса између различитих организација, и коначно развој тржишта сервиса које ће нудити специјализовани провајдери софтверских сервиса (не морају бити организације које су пројектовале и имплементирале сервис).

Процес пројектовања новог сервиса употребом постојећих сервиса, приказан на слици 4.7, је у суштини процес пројектовања софтвера са употребом постојећих модула или компоненти. Овакав приступ пројектовању сервиса подразумева компромисе са захтевима који су иницијално постављени за полазне сервисе, и компромисе са захтевима за нови систем због услуга које нуде постојећи сервиси. Поред функционалних компромиса, веома су важни компромиси који се односе на квалитет сервиса. Фазе у развоју сервиса композицијом су:

- **Формулисање радног тока сервиса** (*Formulate outline workflow*) је иницијална фаза у којој се пројектује идеални дизајн сервиса на основу захтева за композитни сервис. Овако креиран дизајн је апстрактан и подложан је променама када се сазнају детаљи доступних сервиса.
- **Откривање или проналажење сервиса** (*Discover services*) је фаза у којој се траже сервиси у регистрима или каталозима сервиса, са фокусом на детаље о употреби сервиса и начину интеграције сервиса.
- **Избор сервиса** (*Select possible services*) се односи на избор могућих сервиса који би могли имплементирати радни ток сервиса. Критеријуми за избор сервиса су функционалности сервиса, али и цена и квалитет (доступност, одзив, робустност).
- **Редефинисање радног тока сервиса** (*Refine workflow*) се врши на основу информација о доступним сервисима. Редефинисање сервиса укључује додавање детаља у апстрактни опис сервиса, али и преуређење активности у току сервиса (додавање, брисање, измена редоследа). Коначни радни ток се добија када се идентификује стабилан скуп сервиса за укључење у композитни сервис.
- **Креирање програма за радни ток** (*Create workflow program*) се односи на трансформацију апстрактног радног тока сервиса у извршиви програм и коначно дефинисање интерфејса сервиса. За израду



Слика 4.7: *Конструкција сервиса композицијом постојећих сервиса*

извршивог програма се могу користи програмски језици као што су Java или C#, или језик за опис радног тока сервиса WS-BPEL. Интерфејс сервиса се дефинише помоћу WSDL. У овој фази се креира и кориснички интерфејс за приступ сервису (обично веб базирани интерфејс).

- **Тестирање сервиса** (*Test completed service*) подразумева тестирање композитног сервиса пре испоруке.

Радни ток сервиса представља модел пословног процеса, који се уобичајено представљају графичким нотацијама као што су UML дијаграми активности или Business Process Modeling Notation (BPMN). Мапирање BPMN модела омогућује трансформацију модела на нижи ниво XML базираних описа у WS-BPEL, због чега је BPMN усаглашен са стеком стандарда за веб сервисе. Када је завршен финални дизајн сервиса врши се конверзија сервиса у извршиви програм што подразумеве следеће две активности:

- Имплементација сервиса који нису за поновну употребу применом програмских језика Java или C#.
- Генерисање извршне верзије модела радног тока, што подразумева превођење модела у WS-BPEL (аутоматски или ручно).

Тестирање композитних сервиса се не може ослонити на уобичајене технике тестирања програма које се примењују када је изворни код доступан (нпр. инспекција кода) пошто изворни код сервиса добављених од спољних провајдера није доступан. Због тога је кључни проблем у тестирању композитних сервиса разумевање имплементације укључених сервиса. Поред тога јављају се и следећи проблеми:

- Сервиси добављени споља су под контролом провајдера сервиса а не корисника, што значи да све промене сервис зависе од провајдера (па чак и повлачење сервиса из употребе). Решење за овај проблем је развијање стандарда за руковање верзијама сервиса.
- Основна идеја архитектура базираних на сервисима је да се обезбеди динамичко повезивање сервиса, што значи да се не мора увек извршити иста верзија сервиса, па нема гаранције да тестирање може открити све проблеме у функционисању сервиса.

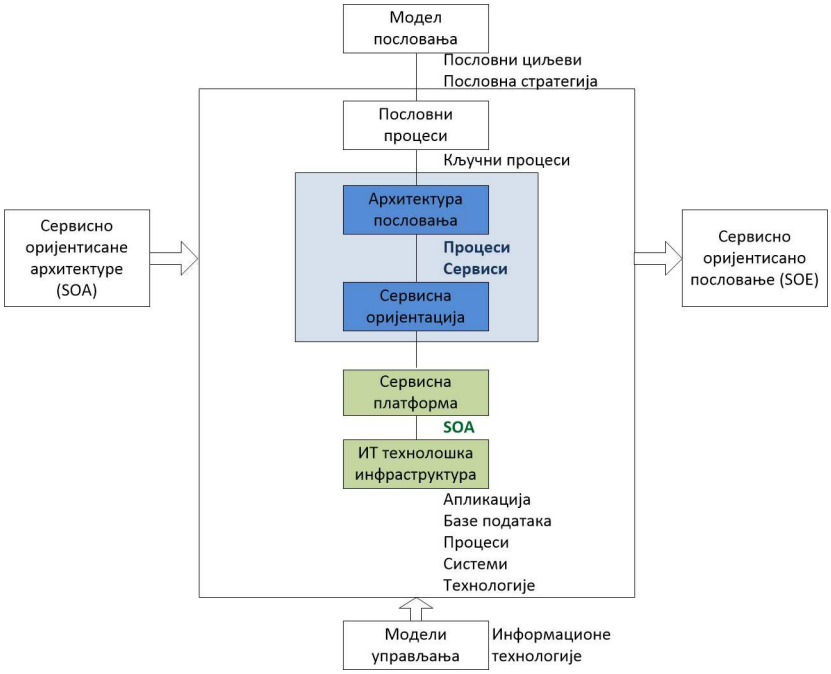
Слика 4.8: *BPM и SOA*

- Тестирање нефункционалних захтева не мора давати поуздане резултате приликом тестирања, пошто сервис не мора бити под пуним оптерећењем као у реалним ситуацијама.
- Модел плаћања сервиса (нпр. претплата или плаћање по употреби) такође може утицати на начин и трошкове тестирања сервиса.
- Употреба сервиса у реалном окружењу може зависти од факктора који се не могу укључити у случајеве тестирања (опрема, непредвиђене акције корисника, итд).

4.5 Сервисно оријентисано пословање

Управљање пословним процесима (Business Process Management (BPM)) је у савременом пословању уско повезано са сервисима које пружају организације, што указује на потребу увођења сервисно оријентисаних архитектура (SOA) у пословање. BPM је приступ у остваривању пословних циљева базиран на управљању и координирању процеса у организацијама, подржан софтверским решењима која се користе као системи за управљање пословним процесима. Примена сервиса у пословању се може посматрати кроз комбиновање сервиса који се тада могу посматрати као процеси којима се приступа према захтеву (*on demand*), а који побољшавају компетитивност пословања кроз континуирано унапређење. Однос BPM и SOA у пословању је приказан на слици 4.8, где се уводи нови концепт пословања базираног на сервисима (Service-Oriented Enterprise (SOE)).

Употреба SOA је базирана на пословном моделу и технолошкој инфраструктури организације. Пословни модел садржи пословне циљеве и стратегију организације, а као подлога за примену SOA у пословним процесима се користе кључни процеси (*core processes*) у организацији. Технолошка инфраструктура утиче на доношење техничких одлука, али је примена SOA у пословању стриктно пословна одлука. На слици 4.9 је



Слика 4.9: *Архитектура пословања заснована на SOA стратегији*

приказана пословна архитектура базирана на SOA, која уз одговарајуће пословне одлуке резултира сервисно оријентисаним приступом пословању.

Сервиси у SOA су пословни модули или техничке функционалности са јасно дефинисаним и изложеним интерфејсом. SOA сервиси се најчешће реализују као веб сервиси који су једноставни за пројектовање и испоруку (веб окружење).

Примена SOA омогућује ефикасније усклађивање пословања и примењених информационих технологија. Техичке и пословне користи које SOA доноси за организацију у којој се примењује су приказане у табели 4.2. Увођење SOA треба треба радити инкрементално да се не би заустављали пословни процеси.

Табела 4.2: *Техничке и пословне користи од примене SOA у пословању*

SOA карактеристике	**Пословне користи**
Једноставно одржавање и замена компоненти	Једноставна замена постојећих пословних компоненти Повећање адаптивности на промене у пословним процесима Брже пласирање на тржишту нових пословних функционалности
Стандардан интерфејс сервиса	Једноставније повезивање нових система Једноставнија интеграција партнера Подршка за аутоматизацију пословних процеса
Аутономија сервиса	Смањење трошкова употребе
Пословна политика	Дефинисање уговора о употреби сервиса Једноставнија интеграција

Значајан допринос сервиса се огледа у интеграцији пословања организација које се састоје од мање-више независних целина. Повезивање или интеграција таквих целина се применом стандардних пословних софтверских решења реализује тако што се повезују поједине тачке у подсистемима, што доводи до компликоване структуре (шпагети стил повезивања), при чему се интеграција врши на више начина:

- Директним повезивањем више база података из различитих подсистема (*ETL, extract, transform, load integration*).
- Директним повезивањем више апликација из различитих подсистема (*Online integration*) базирано на HTTP или TCP.
- Повезивање на нивоу фајл система (*File-based integration*) што омогућује дељење фајлова.
- Директним повезивање апликација и база података из различитих подсистема (*Direct database connection*).

Увођењем добро дефинисане SOA стратегије, са генерализованим интерфејсима сервиса могуће је пружити услуге већем броју анонимних корисника. На тај начин се елимише проблем *шпагети интеграција* и уводи дисциплиновани приступ у комуникацији. Смањење броја повезивања такође олакшава одржавање система и повећава његову флексибилност.

Литература

Pierre Bourque and Richard E. (Dick) Fairley (Editors) (2014) *Guide to the Software Engineering Body of Knowledge, Version 3.0, SWEBOK*. IEEE.

Ethan Cerami (2002) *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL*. O'Reilly Media. Sebastopol, CA, USA.

Wolfgang Emmerich and Nima Kaveh (2002). Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model. *In Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. pp. 691-692. Orlando, FL, USA. DOI: 10.1145/581339.581448

Ian Foster, Carl Kesselman and Steven Tuecke (2001) The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *The International Journal of High Performance Computing Applications*, Volume 15, No. 3, pp. 200-222.

Nicolai M. Josuttis (2007) SOA in Practice: The Art of Distributed System Design. O'Reilly Media. Sebastopol, CA, USA.

James Lawler and H. Howell-Barber (2008). *Service-oriented architecture: SOA strategy, methodology, and technology*. Auerbach Publications, Taylor & Francis Group. Boca Raton, FL, USA.

Savas Parastatidis, Jim Webber and Ian Robinson (2010) *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media. Sebastopol, CA, USA.

Arnon Rotem-Gal-Oz (2012) SOA Patterns. Manning Publications. Shelter Island, NY, USA.

Uttam K. Roy (2015). *Advanced Java Programming*. Oxford University Press. New Delhi, India.

Ian Sommerville (2011) *Software Engineering, 9th edition*. Addison-Wesley, Boston, MA, USA.

Andrew S. Tanenbaum and Maarten Van Steen (2007) *Distributed systems: principles and paradigms*. Prentice Hall. Upper Saddle River, NJ, USA.

Rainer Weinreich and Johannes Sametinger (2001). Component Models and Component Services: Concepts and Principles. In G. T. Heineman and W. T. Councill (eds.) *Component-Based Software Engineering*, pp. 33–48. Addison-Wesley Longman Publishing. Boston, MA, USA.