ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – ВАРНА
Факултет по изчислителна техника и автоматизация
Софтуерни и интернет технологии

ДИПЛОМАНА РАБОТА

за придобиване на ОКС „Магистър”

Тема:

„Разработване на библиотека за изграждане на софтуерни системи работещи в реално време“

Изготвил: Велислав Василев Колесниченко

Ръководител: доц. Хр. Вълчанов

Специалност: Софтуерно инженерство

Факултетен номер: 61562005

ТУ Варна, 2017 г.

Съдържание

Глава 1. Постановка на дипломното задание. Цели и задачи на разработката. Необходимост от решаване на дипломната задача

Работата в реално време се различава от другите форми на обработка на данните с изричното включване на размерността време. Това се изразява чрез основните потребителски изисквания за *своевременна реакция* и *паралелна обработка*, които системите за работа в реално време удовлетворяват. В много системи входните данни се въвеждат от голям брой източници и в повечето случаи това въвеждане на данни може да се припокрива във времето. Паралелната обработка обикновено включва корелативна обработка на данните от повече от един входни източници в същия интервал от време. Изискването за паралелна обработка се получава от вътрешните характеристики на самия проблем и е различно от изискването за припокриваща се обработка на транзакции.

Операционния режим на компютърната система работеща в реално време, предоставя механизми за обработка на данните, пристигащи отвън, който да са в постоянна готовност, така че резултатите от работата им да са налице в предварително определен период от време. Моментите на постъпване на данните могат да бъдат случайно разпределени или да бъдат определени предварително в зависимост от различни приложения.

Поради изискванията на външната среда, придобиването на данните, изчисляването и съответните реакции трябва да бъдат изпълнени на време. В строгите системи за работа в реално време това изискване за своевременна реакция трябва да бъде изпълнено, и в условия на екстремално натоварване и независимо от грешки и неизправности. Това води до по-специфични изисквания за *предсказуемост* и *надежност*. Данните, които трябва да се обработват могат да пристигат случайно разпределени във времето, според дефиницията на режима за работа в реално време. Този факт е довел до широко разпространеното заключение, че поведението на системите за работа в реално време не може и не би могло да бъде детерминирано. Това заключение е базирано на концептуална грешка! Наистина, техническият процес може да бъде толкова сложен, че неговото поведение да изглежда случайно. Реакциите, които трябва да се изпълняват от компютъра обаче, трябва да бъдат прецизно планирани и напълно предсказуеми. Това е в сила особено в случай на едновременно настъпване на няколко събития, водещи до ситуация на състезания за услуга; това също включва преходни претоварвания и ситуации на възникване на неизправности. Независимо от доброто планиране на системата, винаги има вероятност за преходно претоварване във възел, като резултат от

аварийна ситуация. За да се осигури адекватно поведение на системата в такива случаи, са създадени схеми за разпределяне на товара, които дават възможност задачите да мигрират между възлите на разпределени системи.

Фигура 1: Диаграма на условията за система в реално време

1.1. Цели и задачи на разработката

Разработване на софтуерна библиотека предназначена да допълни средата за проектиране .NET Framework, като използва наличните програмни средства и се интегрира и разширява функционалността, за да подпомогне разработването на ефективни и надеждни софтуерни системи.

Основните приноси в тази теза включват представянето на компонентен библиотечен проект за изграждане на софтуерни системи в реално време, който да бъдат приложими в широк диапазон от софтуерни приложения. Тази библиотека се фокусира основно върху проектирането на високо ниво с цел достигане до по голям диапазон от електронно-изчислителните машини. Комуникационния модел за изграждане на компонентите и осигуряване механизма за управление на тези компоненти, използва набор от дизайнерски модели, които са интегрирани, за ефективна употреба.

Библиотеката има за цел да осигури практическа структура за взимане на проектни решения, които са насочени към ускоряване на софтуерното проектиране на системи, които се занимават с обработка на данни в реално време.

Цели се да се осигури рамков модел за изграждане на софтуер със способността да извършва или изпълнява задачи точно в момента, в който се казва, че са изпълнени. По този начин с библиотеката ще може да се създават програми, които могат да изпълняват зададените им задължения и функции точно когато са поискани, а не след изпълнението на програмата или на определена дата. След конфигурирането на софтуерните програми в реално време посредством библиотеката се цели те да изпълняват своите задачи автоматично и да се приспособяват към промените, които се правят от тях в момента.

1.2. Необходимост от решаване на дипломната задача

Софтуерните приложения в реално време са популярни днес, тъй като те осигуряват по-бързо изпълнение на задачите, операциите и дейностите на електронно-изчислителните машини. Софтуерът в реално време позволява на потребителя да изпълнява различни задачи и дейности едновременно, докато програмите се поддържат отворени. В компютърните системи операционните системи в реално време прихващат множество програми, които работят, дори ако потребителят е съсредоточен само върху едно приложение. Някои от тези софтуерни програми също са предназначени да изпълняват планирани задачи, така че дори и да не са отворени, те автоматично отговарят на часовника на компютъра и изпълняват задачите, които им се дават. Тези програми включват медии, строителни инструменти, приложения за изчисления и анализ. Това ги прави изключително важни, тъй като се използват за различни цели и дейности разчитат на софтуер в реално време, поради което те трябва да се актуализират, за да се избегнат забавяния или потенциални злополуки. След навлизането на множество хардуерни устройства с различни сензори, който имат измервателна цел се появява нуждата те да обменят данни помежду си в реално време. Други електронни устройства и уреди, също така използват добре програмите в реално време за собственото им усъвършенстване и иновации, тъй като тези характеристики ги карат да изглеждат по-привлекателни за целевия пазар. Голямото разнообразие от устройства поражда необходимостта тяхното проектиране да се типизира, за да може да се подобри времето за проектиране и да се отговори на нарастващото потребителско търсене.

Архитектурите на повечето конвенционални софтуерни продукти, не предлагат предвидимостта, необходима за поддържане на поведението в реално време, или преконфигурирането, изисквано за тези решения да бъдат приложими в широк диапазон от софтуерни системи.

Глава 2. Описание на използваните програмни средства и технологии

2.1. Програмния език C#

Езикът за програмиране C# е съвременен, обектно-ориентиран и типово обезопасен език за програмиране, който е наследник на C и C++. Той комбинира леснотата на използване на Java с мощността на C++. Създаден от екипа на Андерс Хейлсбърг, архитектът на Delphi, C# заимства много от силните страни на Delphi – свойства, индексатори, компонентна ориентираност. C# въвежда и нови концепции – разделяне на типовете на два вида – стойностни (value types) и референтни (reference types), автоматично управление на паметта, делегати и събития, атрибути, XML документация и други. Той е стандартизиран от ECMA и ISO. C# е специално проектиран за .NET Framework и е съобразен с неговите особености. Той е сравнително нов, съвременен език, който е заимствал силните страни на масово използваните езици за програмиране от високо ниво, като C, C++, Java, Delphi, PHP и др.

2.2. Програмния език JavaScript

JavaScript е съвременен език за програмиране за World Wide Web. Той не само осигурява възможност за разработване на интерактивни Web страници, но и представлява основно средство за интегриране на контроли, plug-in модули за браузъри, сървърни скриптове и други.

JavaScript може да влияе на почти всяка част от браузъра. Браузъра изпълнява JavaScript кода в цикъла на събития т.е. като резултат от действия на потребителя или събития в браузъра.

- Основни задачи в повечето JavaScript приложения са:
    - Зареждане на данни чрез AJAX.
    - Ефекти с изображения и HTML елементи: скриване/показване, пренареждане, влачене, слайд шоу, анимация и много други.
    - Управление на прозорци и рамки.
    - Разпознаване на възможностите на браузъра.
    - Използване на камерата и микрофона.
    - Създаване на 3D графики WebGL.
    - По-добър и гъвкав потребителски интерфейс
- Какво не може да се прави с помощта на JavaScript:
    - Не може да се записва информация на потребителския компютър или отдалечения сървър.

- Не може да се запазва информация директно в отдалечена база данни.
- Не може да се стартират локални приложения.

2.3. Реактивно програмиране

В компютърните науки, реактивното програмиране е парадигма за асинхронно програмиране, свързана с потоците от данни и разпространението на промените. Това означава, че става възможно излъчването на статични или динамични потоци данни с помощта на използвания език за програмиране и че съществува изведена зависимост в рамките на съответния модел на изпълнение, което улеснява автоматичното разпространението на промяната, свързана с потока от данни.

В императивното програмиране изразът $a := b + c$ означава, че на променливата a се присвоява резултата от $b + c$ в момента в които израза се изпълни. В последствие стойността на b или c може да се промени което няма да повлияе на стойността на a. Въпреки това, в реактивен програмиране, стойността на a се актуализира автоматично всеки път, когато стойностите на b или c се промяна. В програмата не се налага повторно изпълнението на израза $a := b + c$,за да се преизчисли настоящата стойност на a.

Реактивното програмиране е приложимо като начин за опростяване на създаването на интерактивни потребителски интерфейси, би могъл да се използва при комуникацията на системите в реално време, тези възможности поставя реактивното програмиране, като обща парадигма за програмиране.

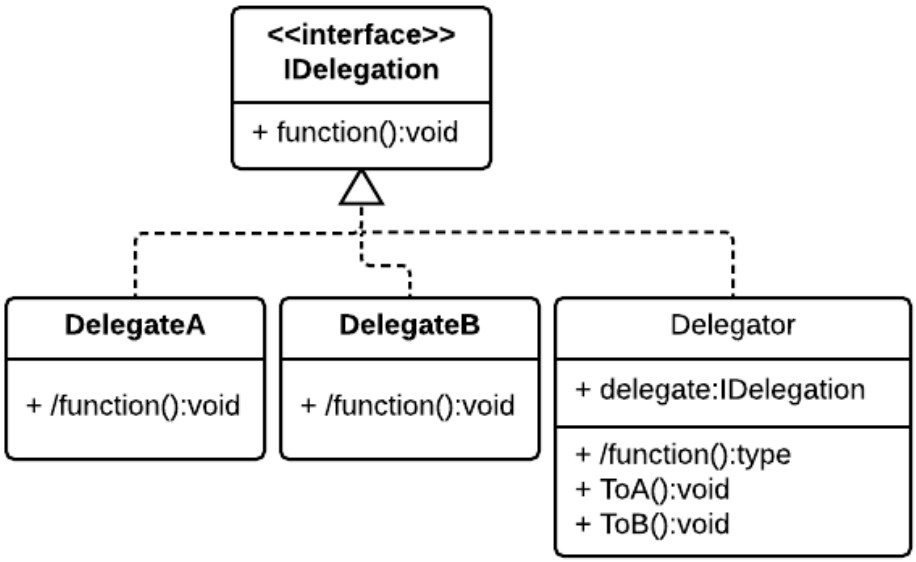
2.3.1. Реактивни шаблони за дизайн

2.3.1.1. Реализиране в средата на C# .NET

В обектно-ориентираните езиците за програмиране реактивният подход е не естествен, при изграждането на софтуерни решения. Възможностите на C# като много-парадигмен език за програмиране предоставят софтуерни конструкции за реализирането на не типични за обектно-ориентираните езици програмни структури.

Използването на реактивен шаблон за дизайн в C# може да се реализира посредством делегати. Делегатът е клас, който съдържа референция към методи или списък от референции към методи с предварително определена сигнатура. Делегатите позволяват методите да се третират като класове и съответно техните инстанции да се предават като параметри към други методи. Тези свойства на

делегатите им позволяват да се използват за реализиране на методи за обратно извикване.



Фигура 2: UML диаграма представяща делегати

При използването на делегат той се дефинира в сървърния клас и се имплементира от клиентския клас. Клиента подава имплементирания делегат към сървъра и при нужда сървъра изпълнява метода за обратно извикване. За да се осъществи извикването на клиентския метод от сървъра в точно определен момент в C# съществува конструкцията събитие.

Събитието представлява съвкупност от условия, които могат да се идентифицират в даден обект, при изпълнението на които да се уведомява регистриран за събитието абонат. Осъществяването на изисква да са на лице следните условия:

- По време на изпълнение трябва да може да се определи резултата от условията за настъпване на събитие;
- По време на изпълнение абонатите трябва да могат да се записват и отписват от събитието.

Събитията са обвързани със следните обекти:

- Пораждащ събитието – обект в който се идентифицират условията на събитието се нарича пораждащ събитието обект. В общия случай в него могат да възникнат произволен брой събития;
- Абонат – обект който получава информация за възникналото събитие (получатели обработващи събитието);
- Пораждане на събитие – обект съдържащ съвкупността от условия за пораждане на събитието;

- Информация за събитието – обект съдържащ допълнителна информация за възникналото събитие;
- Абониране – обект позволяващ съм събитие да се добавят и премахват динамично абонати.

2.3.1.2. Реализиране в средата на JavaScript

JavaScript поддържа взаимодействие с браузъра посредством събития възникващи от действията на потребителя. Това е възможно от възможностите на браузърите да регистрират функции, като манипулатори за конкретни събития.

Всеки манипулатор на събитие в браузъра се регистрира в контекст. Когато се извика addEventListener той регистрира, като метод за целия прозорец, защото в браузъра глобалният обхват е еквивалентен на window обект. Всеки DOM елемент има свой addEventListener метод, който позволява да се регистрират слушатели специално за този елемент.

Можем да прикрепим манипулатор към елементите DOM. По този начин, кликването върху елемент причинява стартиране на съответния манипулатор, докато кликванията върху останалата част от документа не го правят. Функцията addEventListener регистрира втория си аргумент да се извиква всеки път, когато събитието описано в първия аргумент възникне.

Давайки на елемента onclick атрибут има подобен ефект. Но един елемент може да има само един onclick атрибут, така че може да се регистрира само един манипулатор на елемент по този начин. Метода addEventListener позволява да се добави произволен брой манипулатори, така че да не можете случайно да замести манипулатор, който вече е бил регистриран. Метода removeEventListener се извиква с аргументи, подобни на addEventListener, за премахване на манипулатор.

Функцията на манипулатора на събитие приема аргумент: обект на събитието. Този обект дава допълнителна информация за събитието. Например, ако искаме да знаем, кой бутон на мишката е натиснат, можем да определим това от обекта на слушателя. Информацията съхранена в обекта на събитието се различава за всеки тип събитие той съдържа свойството type носещо информация за идентифициране на събитието (например, "click" или "mousedown").

Използвайки механизмите за регистриране на слушатели в JavaScript можем да създадем собствени събития с които да следим промените по обектите в контекста, за да създадем връзка между тяхното изменение с което да се удовлетвори изискването за работа на реактивната парадигма.

2.4. Шаблон за дизайн модел изглед контролер (MVC)

Шаблонът за дизайн служи за изграждане на структурирани софтуерни приложения с разделяне на логическите слоеве на софтуера. Модел изглед контролер модела се състои от три основни компонента:

- Модел - отговаря за абстракцията на данните в приложението;
- Изглед - отговаря за визуалната презентация на приложението;
- Контролер - отговаря за реакцията на потребителския интерфейс към потребителските команди.

Разделението на логиките ни дава възможност да променяме всяка една от тях, независимо от останалите, което ни осигурява по-голяма гъвкавост. Например, ако сменим конкретната визуална презентация на данните, ще се наложи да променим само View компонента, а останалите части ще останат същите.

Поради това, че Модела и Изгледа компонентите нямат директна връзка, имаме възможност по едно и също време да поддържаме различни изгледи на един и същи модел, т.е. едни и същи данни можем да ги показваме по различни начини на потребителя.

Друго предимство на този шаблон е това, че трите компонента могат до известна степен да съществуват независимо един от друг, което повишава тяхната преизползваемост.

MVC е фундаментален шаблон за дизайн, поради това той е в основата на други шаблони и има множество различни варианти и модификации. Това негово предимство го прави подходящ за използване в разпределени и мрежови приложения.

Моделите използвани за разпределени приложения трябва да отговарят на изискванията, на съвременните нужди за разпределените приложения:

- Междуплатформена комуникация – отдалечените програмни компоненти трябва да са достъпни за клиенти с различни операционни системи, изградени върху различни софтуерни платформи;
- Базирана на отворени Интернет стандарти и технологии – различните компоненти на разпределените приложения трябва да са лесно достъпни през Интернет и да се възползват изцяло от предимствата на глобалната мрежа;
- Самоописание – архитектурата за разпределени приложения трябва да предоставя възможност за самоописание на програмните компоненти, което да позволява тяхното използване без да е необходимо

предварително познаване на структурата им и интерфейсът за достъп до тях.

Възможностите на шаблона модел изглед контролер го поставят в списъка на софтуерните модели за създаване на разпределени приложения. Това прави шаблона подходящ за използване в системи работещи в реално време, които използват мрежова комуникация.

Глава 3. Описание на реализацията. Описание на библиотечните класове.

Текущото приложение е системна библиотека разделена в два аспекта за реализиране на сървърни модули и уеб базирани клиентски модули. Разработена за средата на .NET, библиотеката цели да подпомагане разработката на качествен софтуер и улесняване на разработчиците при създаване на нови проекти и решаване на проблемите в текущи разработки свързани с работата на софтуера в реално време. Библиотеката набляга на използването на договорени форматни съобщения между двата модела за клиентска и сървърна реализация.

Наличието на интерфейс, където единственият начин да се разбере дали се е възбудил даден обект, е да се прочете текущото състояние на този обект. За да може да се реагира на настъпилите промени в обекта, ще трябва постоянно да се следи състоянието на обекта, така че да можете да се разбере промяната в обекта преди да е настъпила друга промяна. Това би било опасно за извършване на разнородни и интензивни времеви изчисления, тъй като се рискува да се пропусне преминаването на обект от едно в друго състояние.

Това е редът, по който се обработва входа с по-примитивни машини. Този проблем се решава на една по горна стъпка за хардуера или операционната система която трябва да забележи промяната на събитието и поставянето му в режим на изчакване. Една програма може след това периодично да проверява чакащите на опашката обекти, за нови събития и да реагира на това, което намира там.

Този подход изглежда възможен при предаването на съобщения между две или повече програми. Нека разгледаме следната постановка, имаме сървърна и клиентска програма които общуват през мрежа посредством HTTP протокол. Понастоящем има три основни опции, които се използват за комуникация клиент-сървър: запитване, продължително запитване и потокова. Тези опции използват HTTP протокола с половин дуплекс, за да се симулира поведението в реално време[1].

3.1. Програмни средства за реализиране на комуникация

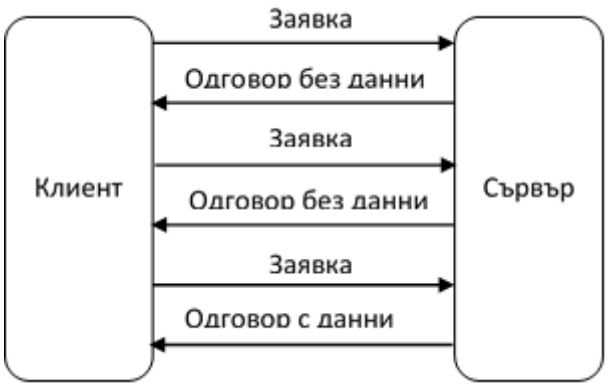
3.1.1. Запитване

При използването на този подход е необходимо клиентският код да изпраща заявки до сървъра въз основа на предварително зададен интервал.

[1] В реално време – способността да реагираме на някое събитие, като се случи

Някои от отговорите на сървъра ще бъдат празни, ако поисканите данни още не са готови, както е показано на фигура 3. Докато данните не бъдат актуализирани на сървъра клиента ще изпраща заявки без отговор.

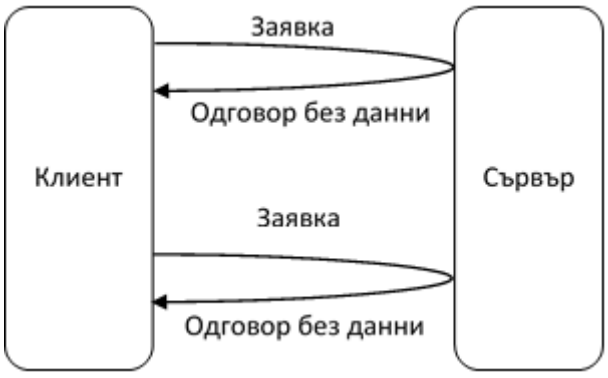
Няма нищо ценно за клиента в тези отговори, но все още ще получава "метаданни". Този подход води до получаване на подробни заглавни части на HTTP отговора, които нямат търсените данни, тези запитвания водят до отделяне на ресурси от сървъра за изпращане на отговори за сметка изпълнение на други задачи, което при големи системи може да доведе до забавяне работата на сървъра и за изпълнението на изискванията от една система работеща в реално време представени в глава 1.



Фигура 3: Протокол заявка отговор

3.1.2. Продължително запитване

Реализирането на този подход започва по подобен начин на първия: клиентът изпраща HTTP заявката до сървъра. Но в този случай, вместо да изпрати празен отговор обратно, сървърът чака, докато данните за клиента станат достъпни. Ако исканата информация не е налична в рамките на зададен интервал от време, сървърът изпраща празен отговор на клиента, затваря и възстановява връзката.

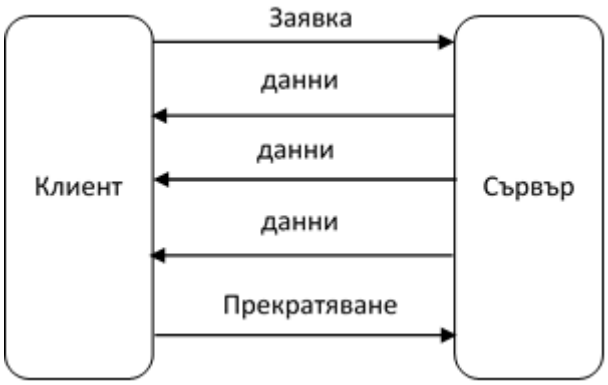


Фигура 4: Удължено запитване

При постъпване на заявка към сървъра той изчаква определен интервал от време преди да върне отговор към клиента. В случай че търсената информация е налична на сървъра или е достъпна преди изтичане на интервала за изчакване към клиента се връща отговор със заявените данни. Когато сървъра не може да отговори на запитването на клиента, той изпраща празен отговор, след което клиента изпраща нова заявка и сървъра изчаква докато има данни за отговор. От гледна точка на спецификацията HTTP, това е нормално поведение този подход може да изглежда така, сякаш се справя с бавно реагиращ сървър, но този процес за изчакване на данни за клиента, заема ресурсите на сървъра за време в което той може да изпълнява други задачи.

3.1.3. Потоков подход

При този подход клиентът изпраща заявка за данни. Веднага след като сървърът получи данните готови, той започва да ги изпраща по отворен поток за данни, без да закрива връзките. Сървърът изпраща данните към клиента, преструвайки се, че отговорът никога не завършва. Например искането на видео води до поточно предаване на данни, без да се затваря HTTP връзката.



Фигура 5: Потоково предаване

Този подход позволява изпращане на всички данни до клиента без прекъсване и подновяване на връзката, но при не последователни данни този подход наподобява предишните в изчакване подготовката на данните за клиента което може да отнеме време през което комуникационния канал между клиента и сървъра ще е зает със отворена заявка.

Обработващата програма[2] трябва да помни, да прегледа опашката и да го прави често, защото всяко време между натиснатия клавиш и забелязването му от програмата, ще накара софтуера да чувства липса на реакция.

[2] Обработваща програма – клиент при избирателен подход и сървър при продължително избирателен и потоков подход

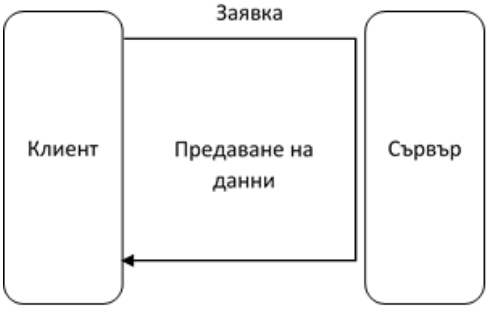
Тези подходи за реализиране на мрежова комуникация не отговарят изцяло на изискванията за система в реално време. За да може да отговори на изискването за предсказуемост и своевременна реакция не може да се разчита на времеви периоди за които не е известно дали ще са достатъчни за постъпването на данните и колко пъти ще се повторят докато данните са налични на сървъра. Решението на този проблем е да се осигури механизъм за мигновено уведомяване на клиента при наличие на готови данни.

Използването на сокет комуникация може да осигури възможност за уведомяване на клиента за постъпване на данни предназначени за него без да се извършва активно изчакване а де се уведоми посредством порт към който е свързан.

3.1.4. WebSocket протокол

WebSocket е стандартен протокол за прехвърляне на двупосочни данни между клиента и сървъра. WebSockets протоколът не се изпълнява по HTTP, вместо това е независима реализация върху TCP. Използване на WebSocket протокола ще позволи да се реализира отваряне на сокет за очакване на данни от сървър. WebSocket протокола се характеризира с:

- Установява се връзка между клиент и сървъра, използвайки се HTTP за първоначалното установяване на връзка,
- Превключване на комуникационния протокол от HTTP към протокол, базиран на сокет,
- Изпращане на съобщения в двете посоки едновременно,
- Изпращане на съобщения независимо. Това не е модел на заявка отговор, защото и сървърът, и клиентът могат да инициират предаването на данни, което позволява в реално време да се уведомяват и двете страни за промяна.
- Както сървърът, така и клиентът могат да инициират прекратяване на връзката.



Фигура 6: Предаване на данни през WebSocket

3.1.4.1. Формат на адреса при WebSocket протокол

WebSocket протоколът дефинира две нови схеми на URI, „ws“ и „wss“, съответно за не криптирани и шифровани връзки. Схемата за URI на „ws“ (WebSocket) е подобна на URI схемата „HTTP“ чрез нея се идентифицира, че ще бъде установена WebSocket връзка чрез използване на TCP/IP без шифроване. URI схемата на „wss“ (WebSocket Secure) идентифицира, че трафикът по тази връзка ще бъде защитен чрез TLS (Transport Layer Security). TLS връзката осигурява като предимства пред TCP връзката, конфиденциалност на данните, целостта и удостоверяване на крайната точка.

3.1.4.2. Управление на WebSocket протокол

Всички мрежови комуникации, които използват WebSocket протокола, започват с отваряне на връзка. Процеса по отваряне на връзка приключва във прехвърляне от HTTP към WebSocket протокол. Това е подобряване на HTTP комуникацията базирани на съобщения.

Причината да се преминава през HTTP, вместо директно да се осъществява връзка с TCP протокол е, че WebSocket работи на същите портове (80 и 443), както и HTTP и HTTPS. Това е важно предимство, защото заявките на браузъра се маршрутизират през едни и същи портове, тъй като поради причини, свързани със сигурността, корпорациите на защитните стени не могат да разрешават произволни връзки на сокети. Също така, много корпоративни мрежи позволяват само определени изходящи портове. Портовете на HTTP и HTTPS обикновено се включват в бели списъци на сигурни канали за комуникация.

Актуализацията на протокола се инициира по искане на клиент, който също предава специален ключ с искането. Сървърът обработва тази заявка и изпраща потвърждение за надстройката. Това гарантира, че връзка с WebSocket може да бъде създадена само с крайна точка, която поддържа WebSocket.

```
GET HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: echo.websocket.org
Origin: http://www.websocket.org
Sec-WebSocket-Key: i9ri`AfOgSsKwUlmLjIkGA==
Sec-WebSocket-Version: 13
Sec-WebSocket-Protocol: chat
```

Фигура 7: Заглавна част на заявката

Клиент изпраща GET заявката за актуализиране на протокола. **Sec-WebSocket-Key** е само набор от произволни байтове. Сървърът приема тези байтове и добавя към този ключ специален глобален уникален идентификатор (GUID) низ **258EAFA5-E914-47DA-95CA-C5AB0DC85B11**. След това създава хеш от Secure Hash Algorithm SHA1 и изпълнява Base64 кодиране. Полученият низ от байтове трябва да бъде използван както от сървъра, така и от клиента, този низ не може да се използва от мрежови крайни точки, които не разбират WebSocket протокола. След това тази стойност се копира в полето на заглавието **Sec-WebSocket-Accept**. Сървърът изчислява стойността и изпраща обратния отговор, потвърждавайки надстройването на протокола:

```
HTTP/1.1 101 Web Socket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Accept: Qz9Mp4/YtIjPccdpbvFEm17G8bs=
Sec-WebSocket-Protocol: chat
Access-Control-Allow-Origin: http://www.websocket.org
```

Фигура 8: Заглавна част на отговора

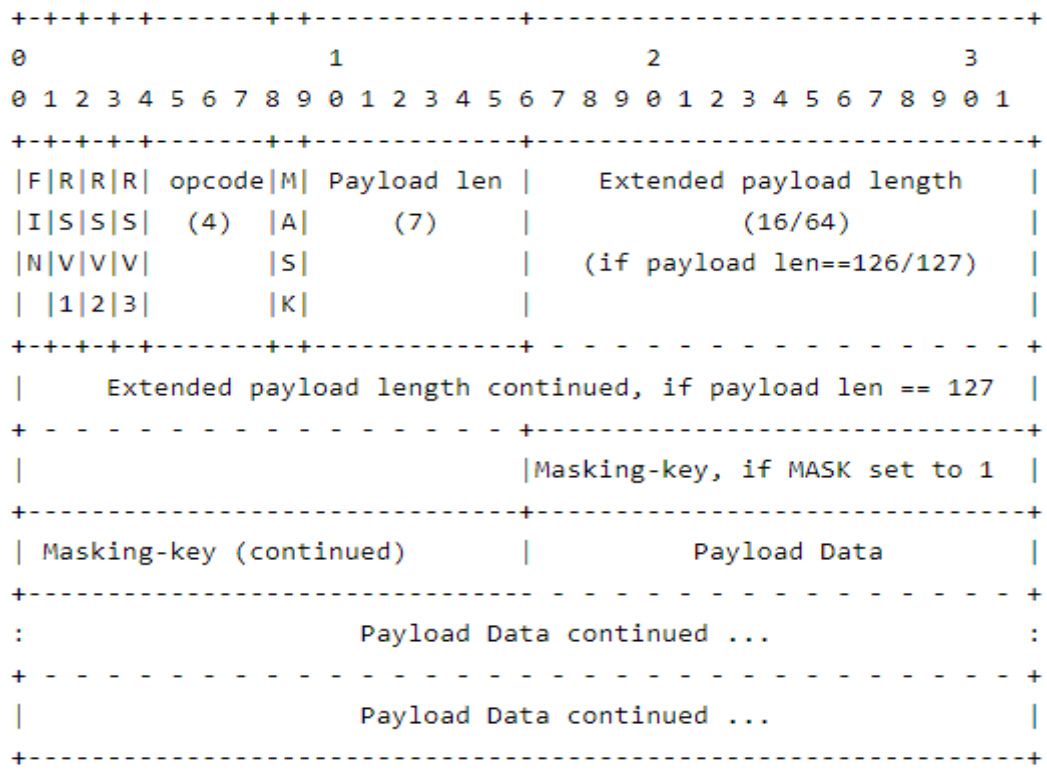
WebSocket протоколът използва HTTP код за грешка 400, за да сигнализира за неуспешно свързване. Първоначалното свързване може да включва и заявка за под-протокол и информация за версията на WebSocket протокола, но не можете да включите други произволни заглавни части. Не можем да предадем информацията за упълномощаване. Има два начина за това. Можете или да се предаде информацията за упълномощаване като първа заявка или да бъде поставен в URL адреса като параметър на заявката по време на първоначалното свързване.

Тъй като протоколът WebSocket създава двупосочна връзка сокет към сокет, сървърът има достъп до сесията за разговори, свързана с такава връзка. Тази сесия може да бъде свързана с клиент и да се съхранява на сървъра.

3.1.4.3. Структура на WebSocket пакета

Управлението на WebSocket е първата стъпка при преминаването към протокола за пакетиране на съобщения, който ще бъде приложен чрез TCP. WebSocket не е протокол, базиран на потоци, като TCP той е базиран на съобщения. При TCP програмата изпраща поток от байтове, който трябва да има конкретна индикация, че прехвърлянето на данни завършва. Спецификацията на WebSocket опростява това, като поставя всяка част от данните в пакет, а размерът

на рамката е известен. JavaScript може лесно да се справи с тези пакети при клиента, защото всеки пристига опакован в обекта на събитието. Но сървърната страна работи малко по-трудно, защото трябва да обвие всяка част от данните в пакет, преди да я изпрати на клиента.



Фигура 9: Формат на WebSocket пакета

Частите на рамката са:

- Първият бит показва дали тази рамка е последната в полезния товар на съобщението. Ако съобщението е под 127 байта, то се побира в един пакет и този бит винаги ще е зададен;
- Следващите 3 битова са запазени за бъдещи промени в протокола и подобрения. Те трябва да съдържат нули, защото в момента не се използват;
- Следващите 4 бита служат за оказване типа на съобщението с opcode:
  - **0x00** Тази пакет съдържа полезния товар;
  - **0x01** Този пакет съдържа текстови данни;
  - **0x02** Този пакет съдържа двоични данни;
  - **0x08** Този пакет прекратява връзката
  - **0x09** Този пакет и пинг;
  - **0xA** Този пакет е понг.
- Следващия бит показва дали рамката е маскирана;
- Следващите 7 бита или 7 + 16 бита или 7 + 36 бита показват дължината на полезния товар в пакета;

- Последния блок съдържа действителните данни.

Разглеждайки четирите метода за реализиране на комуникацията се отличава WebSocket протокола като мощна и ефективна техника за обработка на информация в реално време. Което е подчертано от пълният дуплекс комуникационен канал в WebSocket протокола осигурен чрез една TCP връзка.

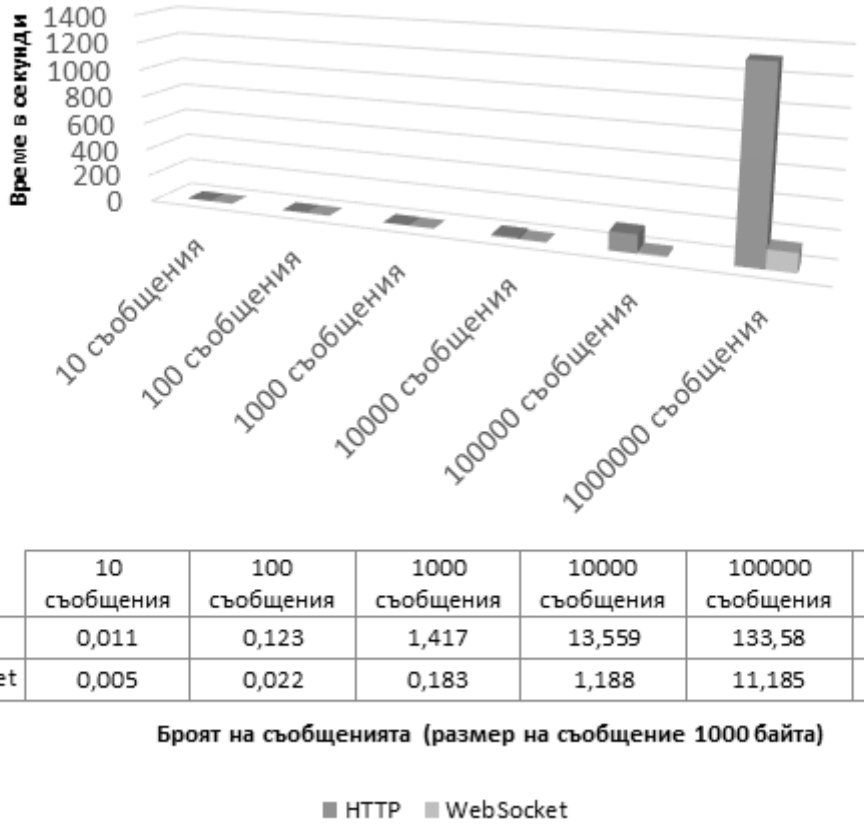
Протоколът WebSocket дава възможност за по-голямо взаимодействие между клиента и сървъра, улесняване на съдържанието на живо и създаването на игри в реално време. Това става възможно, като се предостави стандартизиран начин сървърът да изпраща съдържание на браузъра, без да се търси от клиента, и позволява на съобщенията да се предават назад и напред, докато връзката е отворена. По този начин може да се осъществи двупосочен разговор между клиента и сървъра. Подобен ефект се постига и по не стандартизирани начини, като се използват технологиите за спиране на процесите като Comet. Освен това комуникациите се осъществяват чрез TCP порт 80, което е от полза за онези среди, които блокират нестандартни Интернет връзки, използващи защитна стена , Протоколът WebSocket в момента се поддържа в най-разпространените браузъри, включително Google Chrome, Internet Explorer, Firefox, Safari и Opera. WebSocket също така изисква уеб приложения на сървъра да го поддържат.

Съпоставяйки подходите за предаване на съобщения можем да отличим един като подходящ за използване за реализиране на комуникацията.

	Запитване	Продължително запитване	Потоков подход	WebSocket
своевременна реакция	Предварително определен интервал от време	Предварително определен интервал от време	Непрекъснато предаване при отворена връзка	При наличие на данни
паралелна обработка	Едновременно обработване на няколко заявки	Едновременно обработване на няколко заявки	Едновременно обработване на няколко заявки	Едновременна двупосочна комуникация от клиента към сървъра.
предсказуемост	Заявки без отговор	Заявки без отговор	Заявки без отговор	Предаване на пакети само при нужда
надеждност	Сигурност при предаването на данни	Сигурност при предаването на данни	Сигурност при предаването на данни	Точно определен формат на предаваните данни

Фигура 10: Таблица на предимствата на различните подходи за обработка на съобщения в реално време.

Подходите базирани на HTTP протокола, осигуряват добра надеждност при предаване на данните, между клиента и сървъра. Когато от сървърната страна не са налични данни за съответния клиент, които изпраща запитвания комуникацията е натоварена със празни съобщения от сървъра към клиента. При WebSocket протокола се предават пакети между клиентската и сървърната страна когато са настъпили условия за предаване на данни, през другото време не се натоварва комуникационната средата.



	10 съобщения	100 съобщения	1000 съобщения	10000 съобщения	100000 съобщения	1000000 съобщения
HTTP	0,011	0,123	1,417	13,559	133,58	1324,73
WebSocket	0,005	0,022	0,183	1,188	11,185	135,83

Фигура 11: Хистограма на производителността на избирателен подход и websocket

Резултатите от проведените тестове показват реализацията на WebSocket протокол в библиотеката, като подходящо средство за реализиране на комуникацията. Той отговаря на голяма част от изискванията на системите работещи в реално време. Като част от приложния слой на ISO стандарта той е платформено независим и може да комуникира с различни програмни среди.

3.2. Механизъм за предаване на съобщения

Съобщенията са базовият метод за споделяне на данни между участниците в архитектурите с разпределени ресурси. Текущия проект представя подход за предаване на информация между множество участници в система работеща в

реално време, където данните се разпространяват между участниците. Явното управление на потока на информацията позволява контрол до краен предел на паралелното изпълнение на задачата.

Всяка заявка към сървъра със данни и тяхното описание се пакетира с информация за дестинацията и източника. Така получени пакетите са точно определени за кого се отнасят и на какви събития активират. След получаването им те се обработват от сървърен процес и в момента в които са готови се изпращат към абонираните процеси, за настъпилите промени.

Всеки клиенти има свой уникален адрес уникален идентификатор образуван в момента в който се е присъединил към системата. Възможно е обединяването на клиенти в групи, между които да има общи условия за предаване и приемане на данни. Източник на съобщението е винаги един участник в системата. Това може да е клиента които информира сървърната страна за възникнали промени в неговата защитена среда. Получател може да е един или група от клиенти, в зависимост от броя на регистрираните към текущия процес участници.

Транспортът на съобщенията е пряко свързан с типа и връзките между участниците в системата. Различните архитектури имат различна мащабируемост, латентност и сложност на реализацията.

Основните видове архитектури, който могат да бъдат реализирани с библиотеката са базирани на комуникацията между адресант и адресат. Където двете страни във връзката са динамично изменяеми, както и броят на участниците в системата.

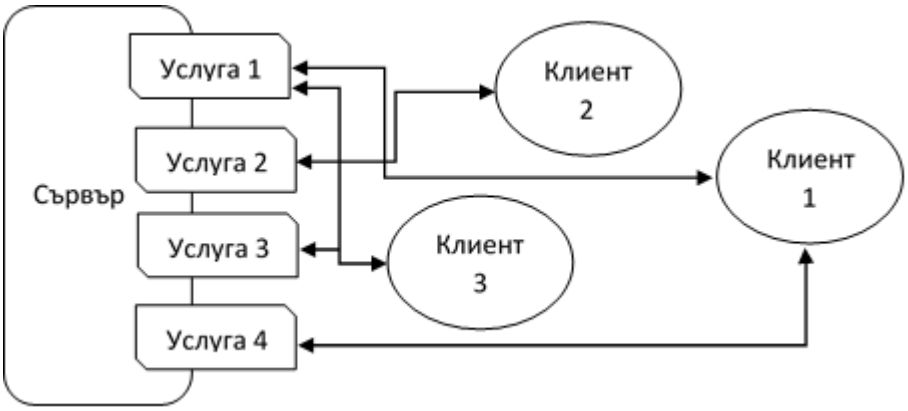
3.2.1. Архитектура Клиент-сървър с двама участници

Архитектурата клиент сървър с двама участници представлява най-простия случай на приложение. Приложението се моделира като набор от услуги, предоставяни от сървъри, и клиент ползващ тези услуги. Клиентите познават сървърите, те възбуждат началото на комуникацията. При осъществена връзка сървърът помни кои клиенти са се обърнали към него и къде се намират. Всеки клиент, който се обърне към сървъра се закача за сървърно събитие, което служи за реактивно опресняване на данните при клиента, въз основа на условия описани предварително от проектанта на системата. Така осъществената връзка отваря WebSocket при клиентската страна където сървърът изпраща настъпилите промени в събитията за които клиента е закачен. Клиентите и сървърите са логически процеси на различни компютърни конфигурации в общия случаи отдалечени един от друг. Топологията на мрежата между тях е не известна

поради не обвързаността на библиотеката с конкретна, реализация на средата за комуникация и операционната среда.

3.2.2. Архитектура Клиент-сървър с множество участници

Архитектурата клиент сървър с множество участници представлява надграждане на архитектурата от предходната точка. Както при участник от един клиент тук сървърът също предоставя набор от услуги. Тези услуги са моделирани с компонентите на библиотеката като събития със зададени правила за активиране. При тази реализация на сървърния процес към него могат, да се обръщат множество клиенти като изискват услуга от него. В реализацията която предоставя библиотеката за изграждане на системи работещи в реално време един клиент обръщайки се съм услуга от сървъра той се регистрира за нейната



Фигура 12: Архитектура клиент сървър в условията на библиотеката

употреба. При свързването си клиента отваря WebSocket, който е известен на сървъра и при настъпване на промяна по услугата за която е регистриран клиент, той се уведомява в момента на възникване на промяната. Както към един сървър могат да се обръщат много клиенти така към една услуга могат да се регистрират различни клиенти.

Както е показано на фигура 12 един клиент може да е свързан към една конкретна услуга към сървъра и да получава съобщения от нея, също така може да е регистриран към много услуги и всяка да изпраща съобщения в различен момент от време. Към една услуга на сървъра, от примера се вижда, че могат да са свързани множество клиенти, които в различен момент от време да изпращат съобщения към сървърна услуга и да получават съобщения на базата на комуникацията на останалите участници в системата и на вътрешно сървърни промени.

Така организирана системата позволява естествено да се получи паралелното предаване на съобщения между сървъра и клиентите както и

паралелната работа на цялата система. Предаването на съобщения между двете страни на комуникация се осъществява по два различни и независими канала.

3.2.3. Архитектура разпределени обекти

Моделът клиент-сървър не пасва добре на идеята за приложения работещи в реално време в разпределена среда, защото с нарастване на сложността им нараства и нуждата от включване на повече от два процеса. Остава възможността да се използва модел за отдалечена комуникация, които позволява изграждането на многослойни разпределени приложения.

Архитектурата разпределени обекти предоставя точно такава възможност където всеки обект има представлява отделен процес в цялостната система в реално време. Стандартния подход за изпълнение на този модел предоставя възможност за отдалечен достъп до обекти, които са разположени върху отдалечен сървър и се извикват от външни за сървъра процеси.

Библиотеката за проектиране на приложения работещи в реално време, разглежда този архитектурен модел като приложение разпределено на различни процеси. Тези процеси може да са както на една физическа платформа, така и на отдалечени устройства от различен характер.

Фигура 13: Последователна връзка на разпределени обекти

Този механизъм на проектиране позволява, да се създадат разпределени приложения с широк набор от инфраструктури, в който предаването на съобщения се реализира с помощта на разработената библиотека. Фигура 13 представя модел на разпределени обекти в които връзките са последователни между отделните процеси всеки е свързан с предишния и следващия. В този случай съобщенията се предават от процес на процес като промяната в единия краен процес преминава последователно през всички процеси за да достигне до другия краен процес. Възбуждането на промяна в централен обект води го разпространение на съобщения към всички свързани обекти, което актуализира процесите непосредствено след времето за възникване на промяна.

Фигура 14: Разпределени обекти свързани всеки към всеки

Организирането на архитектура, в която всеки от възлите да е свързан до всеки от другите възли в архитектурата чрез пълен дуплекс, прави възможно едновременното предаване на данни от даден възел до всички останали в двете посоки. Фигура 14 показва този архитектурен модел, в който Процес 2 изпраща съобщения към Процес 1 и Процес 3, като в един и същи момент те могат да изпращат съобщения към него.

Механизмът за предаване на съобщения използва обръщане на контрола като абстракция, описваща един аспект то софтуерните архитектури, в който потокът на контрол в система е обърнат в сравнение със стандартното прилагане на архитектурните модели. В стандартните имплементации един процес, управлява потока на съобщенията едностранно. Процеса заявил ресурса изчаква да се достави отговора след което прекратява връзката. Последвалото обновяване на данните изисква ново свързване и прекратяване на връзката всеки път дори в случаите когато данните не са променени и инициализиращия процес няма какво да получи, той отново трябва да отвори и затвори връзката. Създавайки възможност при сървъра да помни местоположението на клиентите свързани към него. Формата за комуникация се обръща вече не е нужно клиентския процес инициирал първоначалното извикване да повтаря действието си да се е сигурен, че ще получава обновление на данните. Знаейки кой клиент се интересува от тези данни сървърът може да извърши изпращането на съобщения само ако има настъпили промени в данните.

3.2.4. Формати за предаване на съобщения

Архитектурата определя пътя по който ще текат съобщенията между софтуерът работещ в реално време. Важно условие, за да могат да функционират съвместно отделните модули в системата е те да разбират съобщенията, които си пращат. Начинът по които се решава този проблем е като има общи правила за съобщенията за всички участници в системата.

Форматът на съобщенията може да бъде представен в двоичен вид или чрез текстово форматиране.

- Двоично форматиране – представяне на вътрешните обекти в двоичен код. Полученият в резултат на форматирането поток е много компактен. Този формат съдържа информация за самото съдържание на данните така и допълнителна информация за типовете и класовете на обектите. При този вид форматиране двете страни трябва да разпознават обектите които са представени е двоичен вид и да могат да разчетат кога на кодиране, за да може да преобразуват данните от формата за предаване на съобщението в транспортния клас преди да го трансформират във вътрешно системни обекти.
- Текстово форматиране – представяне на вътрешните обекти с помощта на текстови символи. Предаденият в резултат на форматирането поток съдържа информация за данните които се предават без да се интересува за конкретните обекти които представляват. При този вид форматиране двете страни трябва да могат да разпознават конкретния формат на съобщението което се предава а не обекта който стой зад него. Получено съобщението може да се трансформира директно във вътрешно системен обект
    - Разширяем маркиращ език – е стандарт, дефиниращ правила за създаване на специализирани маркиращи езици, както и синтаксисът, на който тези езици трябва да се подчиняват
    - Текстово базиран отворен стандарт – е стандарт за предаване на прости структури от данни и асоциативни масиви. Използва се за предаване на структурирани данни през Интернет.

Възможността всички участници в системата да разбират какви съобщения се изпращат по време на комуникацията изисква те да могат да ги преобразуват от формат за предаване на съобщения във вътрешни обекти на процеса и обратно, за да могат да изпратят вътрешно обработените обекти до необходимия възел в системата. При използването на двоично форматиране е нужно всеки участник да очаква точен обект по предавателният канал и да има необходимите свойства да го преобразува. Това намалява възможността за динамично променяне на системата, всеки новодобавен модул първо трябва да имплементира обектите които се предават като съобщения и да отделя време за тяхната трансформация в полезен обект. Време което може да се окаже важно във последвалата обработка на данните. Текстово форматираните данни предоставят поток, който притежава ключови символи по които всеки участник в системата може да разпознае съобщението и да го представи като собствен обект, без да се преминава през междинно преобразование.

3.2.5. Образуване на съобщение в клиентската библиотека

Контекстно свободният дизайн на JavaScript носи предимства в разработването на потребителски интерфейс, подобряване на потребителската работата с приложения разработени за работа в браузър. Това предимство на езика се превръща в недостатък когато се налага да взаимодейства с други приложения в обща разпределена система. Отделните модули в една система трябва да могат да си предават съобщения, които са с точен формат и тип, за да могат без проблемно да се разбират отделните модули.

Налага се библиотечният модул разработен върху JavaScript да може да предава точно формирани съобщения към сървърния модул, за да се осигури съвместимост в системата. Голямото разнообразие от приложения и методологии за работа с JavaScript причиняват неудобство при представянето на крайни решения за обща употреба.

Така представените формати на съобщения очертават избирането на текстов формат като предимство пред конкретизирането на формата и кодирането му в съобщение. Текстовият формат за съобщенията позволява да се представят обекти в текстово базиран отворен стандарт (фигура 15). Този формат използва ключове и стойности при разпознаването на данните, и тяхното безпроблемно категоризиране. Обемът на файловете са значително малък, което води до по-бързо предаване и обработка. Така представени обектите използват по-малко ресурси и време за трансформиране в сравнение с други подобни формати за представяне на обекти. Важно условия, за да може да се отговори на критериите за времеви диапазон в системите обработващи данни в реално време.

```
“човек”: {      "име": "Иван Иванов",
                "години": 59,
                "адрес": {
                        "улица": "Студентска 1",
                        "град": "Варна",
                        "пощенски код": 9010
                },
                "телефон": [ {  "тип": "домашен",
                                "номер": "052 12 -34- 56"
                        }, {  "тип": "факс",
                                "номер": "056 555-1234"
                        }
        ]
}
```

Фигура 15: Текстово базиран отворен стандарт

Използването на разширяем маркиращ език позволява да се предава допълнителна информация към данните. Силното разпространение на този формат го прави задължителна възможност за формат на съобщенията, за да е съвместима библиотеката с настоящи проекти.

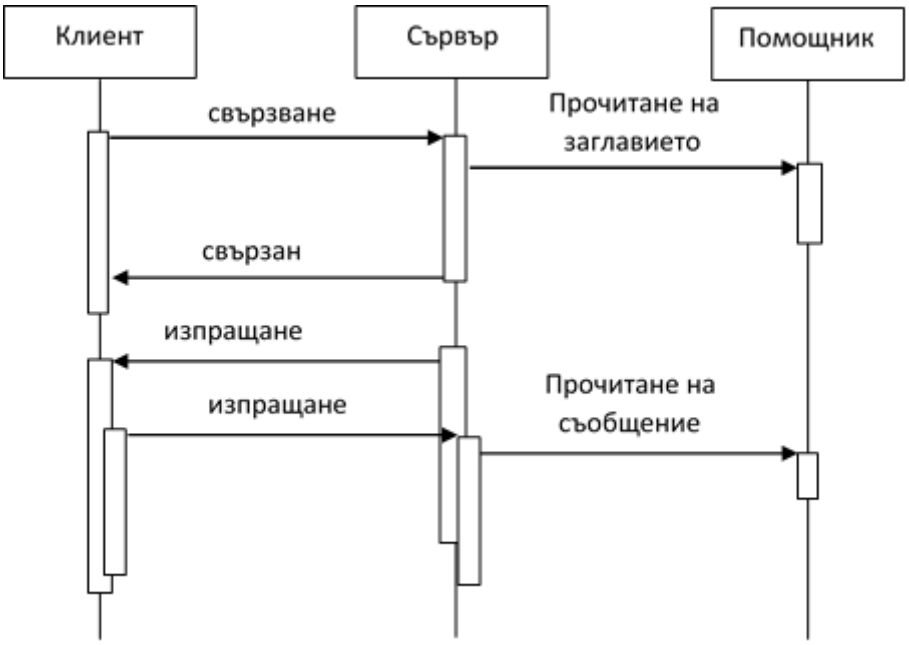
```
<person>
  <name>Иван Иванов </name>
  <age>59</age>
  <address>
     <street>Студентска 1</street>
     <city>Варна</city>
    <postalCode>9010</postalCode>
  </address>
  <phoneNumbers>
         <phoneNumber>
           <type>домашен</type>
           <number>052 12 -34- 56</number>
         </phoneNumber>
         <phoneNumber>
           <type>факс</type>
           <number>056 555-1234</number>
         </phoneNumber>
 </phoneNumbers>
</person>
```

Фигура 16: Разширяем маркиращ език

3.2.6. Образуване на съобщение в сървърната библиотека

След като разгледахме форматите на съобщенията и определихме възможните такива за клиентската библиотека те трябва да могат да бъдат разбираеми и за сървърната част от библиотеката.

Получаването на съобщения от сървърният модул зависи от разбирането на протокола за комуникация определен от програмните средства за реализиране на комуникация. Получавайки WebSocket пакета на модула извлича съобщението от данновата част на пакета от фигура 9 описан в глава 3. Рамката за проектиране .NET и програмният език C# позволяват използването на валидните за съобщенията формати, което прави лесно използването на съобщенията.



Фигура 17: UML диаграма за предаване на съобщения

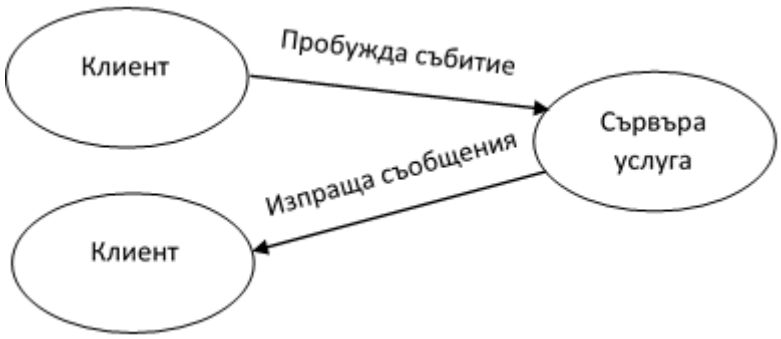
Механизмът за предаване на съобщения е демонстриран във фигура 17. Диаграма на последователностите показва реда в които започва и се обработват съобщенията. Първоначално клиентският процес извършва първоначално запитване за осъществяване на връзка със сървърният процес, адреса на сървъра трябва да съдържа ws:// за означаване на вида връзка и заглавната част на съобщението описан във фигура 7 в точка 3.1.5.

Сървъра изпраща потвърждаващо съобщение за установяване на връзка. След като е осъществена връзка между процесите всеки може да инициализира изпращане на съобщения. Фигура 17 демонстрира предаване на съобщения от сървъра без предварително заявяване от клиента, като докато протича предаването на съобщение от едната страна паралелно може да започне изпращане от другата. Отговарянето на съобщение не е задължително, ако не е необходимо такова не се осъществяват излишни комуникационни действия. Това спестява време при обработка и не натоварва комуникационната среда.

3.3. Пробуждане на събития

Събитията се разглеждат като съобщения за настъпване на конкретно действие. В библиотеката компонентите изпращат събития към своя притежател за да го уведомят за настъпването на интересна за него ситуация. Този модел служи, когато клиентски процес изпрати съобщени, той предизвиква събитие, с което известява, че е настъпила промяна. Тази промяна активира всички зависими процеси, чиито резултати води до изпращане на съобщения с

промените до свързаните клиентски процеси по този начин се постига в реално време обновяване на цялата система.



Фигура 18: Диаграма на използването на събитие

3.3.1. Събития в клиентската библиотека

Клиентската библиотека е създадена върху спецификацията на W3C, който позволява на уеб страниците да използват протокола WebSocket за двупосочна комуникация с отдалечен хост. Спецификацията дефинира четири основни събития.

- Събитие за свързване – събитието за свързване възниква когато се инициализира WebSocket обекта, преди все още да е установено свързване със сървъра. То служи за уведомяване на сървъра за желанието на клиент да се закачи.
- Събитие за осъществяване на връзка – събитието за осъществяване на връзка възниква след събитието за свързване и отговорът на сървъра, че свързването е успешна. След установяването на връзка със сървърът комуникацията е възможна
- Събитие за прекратяване на връзка – събитието за прекратяване на връзка възниква при успешно осъществено свързване и съобщение за разкачване инициализирано от страната на клиента и прието от сървърната страна. Съобщение за прекратяване на връзка може да възникне и в следствие на разпадане на връзката или при не възможност да бъде отворена.
- Събитие за приемане на съобщение – събитието за приемане на съобщение възниква при изпращане на съобщение от сървъра. Това събитие се използва за да може клиентският процес да може да установи кога е пристигнало съобщения от сървъра. При обработката на това съобщение се реализира трансформирането на сървърното съобщение във формат валиден за клиентския процес.

Имплементирането на тези събития позволява при възникването им да има предвидимост при изпълнението на програмата.

3.3.2. Събития в сървърната библиотека

Събитията служат, да се създаде отношение между клиентите и услугите на сървъра. Главните събития които възникват на сървъра са *Свързване*, *Напускане* и *Получаване на съобщени*. Пакетите предавани между клиента и сървъра съдържат кодове, които са описани в точки 3.1.5.3, за определяне вида на съобщенията които се предават в пакета.

Когато даден клиент изпрати заявка за свързване, той активира сървърното събитие Свързване това събитие. Сървъра приема заглавната част на заявката и определя нейната валидност. Валидността на заявката активира образуването на отговор, които се изпраща към клиента и активира клиентското събитие за свързване. Събитието за свързване се извиква еднократно в момента в които клиент заяви връзка. След като е осъществена връзка на сървъра започва да се обработват останалите събития свързани с текущия клиент.

Архитектурата на приложенията проектирани с библиотеката от текущия проект разчитат на предаването на съобщения между отделните компоненти на системата. Предаването на съобщения се осъществява от събитията за приемане на съобщения съответно в клиентската и сървърната реализация. Събитието за получаване на съобщения се изпълнява многократно на сървъра при всяко съобщение идващо от клиента.

При невалидно съобщение за свързване се извиква вътрешно системно събитие за *Напускане*, което изпраща към клиента съобщение за невалидна връзка с което не се осъществява връзка между двата процеса. Други причини за пробуждане на събитие за *Напускане* са разпадната връзка, когато мрежата между клиентския процес и сървърния процес се разпадне. Нормалният начин за извикване на събитието на Напускане и с инициализиране от клиента съобщение за прекратяване на връзката в заглавната част за заявката се изпраща код **0x08** описан в точка 3.1.5.3. Събитието за прекратяване на връзка се изпълнява еднократно, задължително след успешното изпълнение на събитието за осъществяване на връзка и в следствие на валидно условие за неговото изпълнение.

Глава 4. Примерни приложения, използващи библиотеката.

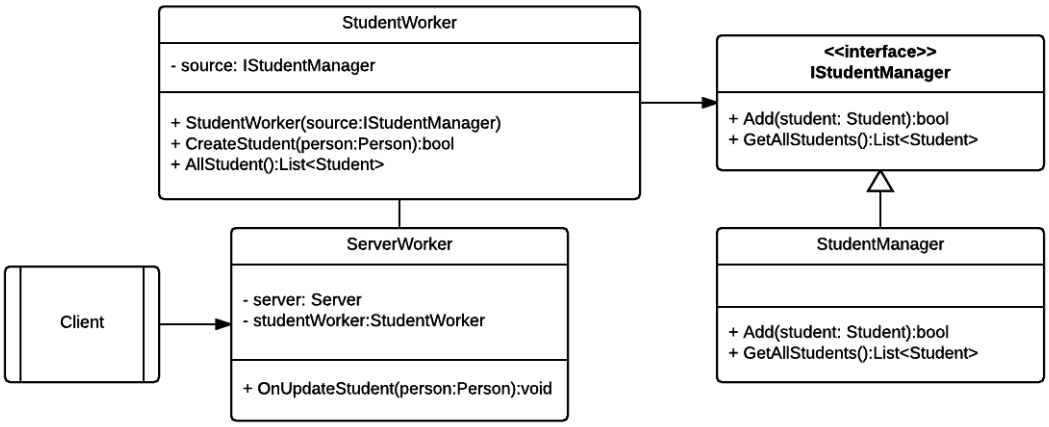
4.1. Информационни системи с бази данни

Базите данни формират критичната част на софтуера, който се изпълнява в информационните системи. Класическата форма на информационните системите включващи съхранение в системи за управление на бази данни, проектирани за предоставяне на данните при поискване от приложенията. Общите характеристики на такива приложения са, че те поддържат голямо количество данни, но операциите, които могат да се изпълняват върху данните са прости.

Съществуват два общи подхода, които предлагат решение на проблема за комбиниране на възможностите за бърз достъп на езика за операции с данни и възможностите за операции върху данни, предлагани от езиците с общо предназначение. Логическият подход използва език, който наподобява логически правила. Обектно-ориентираният подход използва език с възможности за дефиниране на абстрактни типове данни или класове. Тези системи разрешават на потребителя да вгражда структури от данни за бърз достъп в своите класове. Базирайки се на втория подход текущия проект предоставя възможност за двупосочна комуникация между проекта обработващ данните (сървър) и проекта използващ данните (клиент).

Стандартните подходи за извличане на данни се основават на протокола поискване и предоставяне, където сървърната страна предоставя данни само ако са били поискани докато клиентската страна получава данни само ако ги е поискала.

Използвайки библиотеката за обработка на данни в реално време може да се организира предоставянето на данни без те да са неприкосновено поискани преди изпращането. Данните се предоставят на клиенти които предварително са ги заявили, но вместо за да получат актуални данни клиентите да трябва да извършват повторно поискване, сървърната страна извършва изпращане на данни при тяхната актуализация.



Фигура 19:Диаграма насистема за управлвние на студентска информация

Клиентите се свързват със сървърния работник след което при промяна на данните по студентите се активира събитието OnUpdateStudent, което разпространява промените към всички свързани клиенти към системата.

Използването на този подход е възможно за всички конвенционални приложения. С този подход може да се реализират мобилни приложения, при които до момента се е извършвало периодично поискване на данни от сървър да ги получават при промяна с което, да се спести обработка при клиента и да се съхрани енергия на устройството.

4.2. Невронни мрежи

Невронните мрежи могат да боравят с неточни и непълни данни, да се тренират, да се обучават, като се адаптират към изменящата се околна среда, да обобщават в ситуации, които не са им познати, и да се изпълняват много бързо веднъж, след като са тренирани.

Проектите, изградени на базата на невронна мрежа работят върху проблеми, свързани с обработка на естествения език - такива като разпознаване и синтезиране на реч, и преобразуване на текст в реч.

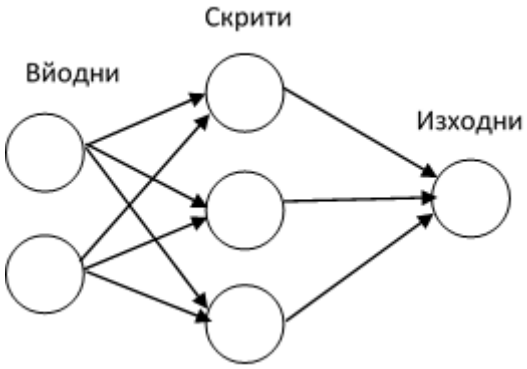
Невронните мрежи функционират много добре в ситуации, където за да се достигне дадена цел, трябва да се направят много стъпки, а изборът на всяка стъпка е сред много възможни. Проблемът е, че има твърде много възможности, които трябва да се опитат в допустимо време на компютърна система с дадена изчислителна мощност.

Невронните мрежи се използват за маршрутизиране на трафика в телефонната мрежа, при проектиране на електронни платки определят мястото на чипа с цел оптимално опроводяване и други подобни проблеми, свързани с

назначаване на ресурси. Освен при маршрутизация за минимизиране на закъснения, понякога те трябва също да могат да отговорят бързо на радикално изменящата се околна среда на системата. Невронната мрежа може да не намери най-доброто решение (което би могло да отнеме значително време дори и на суперкомпютър), но намира достатъчно добър отговор.

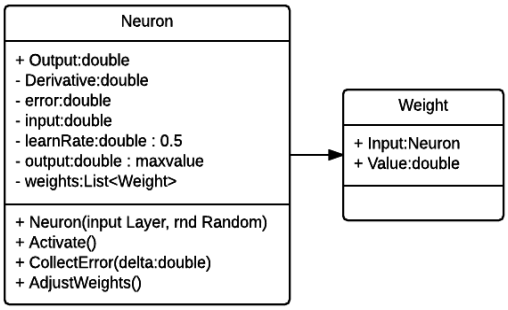
В тази ситуация комбинацията и с разпределена система би допринесло за подобряване на ефективността на мрежата където невроните са разпределени в различни мрежови възли и комуникират в реално време. Тази ситуация би ускорила времето за изчисление заради разпределения характер на такава система и разделената отговорност където всеки възел със собствена изчислителна мощ, отговаря за единична операция, за която получава входни сигнали от предходното ниво и предава изходния сигнал към следващото.

Невронна мрежа с обратно разпределение на грешката има архитектура със един входен и един изходен слои от неврони, като между тях може да има няколко скрити слоя с неврони.



Фигура 20: Трислойна невронна мрежа

На Фигура 20 е показана примерна архитектура на трислойна невронна мрежа с входен, изходен и един скрит слой. На входа на мрежата постъпва входния вектор X.



Фигура 21: Диаграма на един неврон

За активационната функция на невроните се използва:

$\varphi(x) = \frac{1}{1+ e^{-z}}$, чиято първа производна има вида: $\frac{\delta\varphi}{\delta z} = \varphi(z)(1 - \varphi(z))$ Където $z = \sum_{i=0}^{n} w_{kj} O_k$ сума от произведението на теглата на връзките между невроните в два слоя и изхода от свързания неврон. За определяне коректността на изходния резултат се изчислява средно квадратичната грешка $\varepsilon = \frac{1}{2}(t - y)^2$ като при стойност по голяма от избрания екстремум мрежата подлежи на обучение в противен случаи тя е обучена.

Глава 5. Ръководство за програмиста.

5.1. .Net библиотека

Клас Server

Синтаксис

public partial class Server

Публичен клас служеща за създаване на сървър обекти. Този клас служи за организиране на сървър за обработка на данни в реално време.

Полета

private Socket socket
Защитено поле за съхранява сокета на сървъра

private IPEndPoint endPoint
Защитено поле за съхранява крайната точка на сървъра

private List<Client> clients
Защитено поле за съхранява списък от свързаните клиенти към сървъра

Свойства

public Socket Socket
Публично свойство, което връща сокета на текущия сървър.

public IPEndPoint EndPoint
Публично свойство, което връща крайната точка на текущия сървър.

public int ConnectedClientCount
Публично свойство, което връща броя на свързаните клиенти към сървъра.

Събития

public event EventHandler<OnSendMessageHandler> OnSendMessage
Публично събитие за уведомяване при изпращане на съобщение

public event EventHandler<OnClientConnectedHandler> OnClientConnected
Публично събитие за уведомяване при свързване на клиент към сървъра

public event EventHandler<OnMessageReceivedHandler> OnMessageReceived
Публично събитие за уведомяване при получаване на съобщение.

public event EventHandler<OnClientDisconnectedHandler> OnClientDisconnected

Публично събитие за уведомяване при прекратяване на връзката към сървъра

Конструктор

Име	Описание
Server(IPEndPoint endPoint)	Инициализиращ полетата на класа.

Методи

Име	Описание
Client GetConnectedClient(int index)	Връща свързан клиент към сървъра
Client GetConnectedClient(string guid)	Връща съответствие на обект
Client GetConnectedClient(Socket Socket)	Връща свързан клиент към сървъра
void Stop()	Спира изпълнението на сървъра
void ReceiveMessage(Client client, string message)	Получаване на съобщение от сървъра
void ClientDisconnect(Client client)	Прекратяване на клиентска връзка
void SendMessage(Client client, string data)	Изпращане на съобщение към клиент

Метод Client GetConnectedClient(int index)

Синтаксис

public Client GetConnectedClient(int index)

Публичен метод, които служи за извличане на клиент свързан към сървъра.

Параметри

index

Тип: int

Индекса на клиента от списъка с клиенти свързани към сървъра.

Връщана стойност

Тип: Client

Връща клиентски обект свързан към сървъра по индекс. При невалиден индекс метода връща null за несъществуващ клиент.

Пример

```
for(int index = 0; index < server.ConnectedClientCount; index++)
{
	Client client = server.GetConnectedClient(index);
	if(client != null)
		Console.WriteLine("Клиент номер: {0}, GuId: {1}", index, client.GuId);
}
```

Метод Client GetConnectedClient(string guid)

Синтаксис

public Client GetConnectedClient(string guid)

Публичен метод, които служи за извличане на клиент свързан към сървъра.

Параметри

guid

Тип: string

Уникален идентификатор на клиента от списъка с клиенти свързани към сървъра.

Връщана стойност

Тип: Client

Връща клиентски обект свързан към сървъра по уникален идентификатор. При невалиден индекс метода връща null за несъществуващ клиент.

Пример

```
	string guid = “0123456789abcdefghijklmnopqrstuvwxyz”;
	Client client = server.GetConnectedClient(index);
	if(client != null)
		Console.WriteLine("Клиент номер: {0}, GuId: {1}", index, client. GuId);
```

Метод Client GetConnectedClient(Socket Socket)

Синтаксис

public Client GetConnectedClient(Socket socket)

Публичен метод, които служи за извличане на клиент свързан към сървъра.

Параметри

socket

Тип: socket

Сокета на клиента от списъка с клиенти свързани към сървъра.

Връщана стойност

Тип: Client

Връща клиентски обект свързан към сървъра по сокета. При невалиден сокет метода връща null за несъществуващ клиент.

Пример

```
Client client = server.GetConnectedClient(socket);

if(client != null)

        Console.WriteLine("Клиент номер: {0}, GuId: {1}", index, client. GuId);
```

Метод void Stop()

Синтаксис

public void Stop()

Публичен метод, които служи за спиране на сървъра, затваря сокета и почиства паметта.

Пример

```
try {
        Server server = new Server(new IPEndPoint(IPAddress.Parse("127.0.0.1"),
8181));
        for(int index = 0; index < server.ConnectedClientCount; index++) {
                Client client = server.GetConnectedClient(index);
                if(client != null)
                        Console.WriteLine("Клиент номер: {0}, GuId: {1}", index,
client.GuId);
        }
} catch(Exaption e) {
        Console.WriteLine("Грешка: {0}", e.Messige);
} finally {
        server.Stop()
}
```

Метод void ReceiveMessage(Client client, string message)

Синтаксис

public void ReceiveMessage(Client client, string message)

Публичен метод, които служи за получаване на съобщение от сървъра.

Параметри

client

Тип: Client

Клиентският обект, от който се получава съобщение.

message

Тип: string

Текстово съобщение което се получава.

Пример

```
Client client = new Client(servet, socket);
client.Server.ReceiveMessage(client, “Здравей сървър аз съм клиент”);
```

Метод void ClientDisconnect(Client client)

Синтаксис

public void ClientDisconnect(Client client)

Публичен метод, които служи за прекратяване на връзка между клиент и сървър.

Параметри

client

Тип: Client

Клиентският обект които ще прекратява връзка.

Пример

```
Server server = new Server(new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8181));
Client client = new Client(servet, socket);
client.Server.ReceiveMessage(client, “Здравей сървър аз съм клиент”);
server.ClientDisconnect(client);
```

Метод void SendMessage(Client client, string data)

Синтаксис

public void SendMessage(Client client, string data)

Публичен метод, които служи за изпращане на данни към клиент свързан със сървъра.

Параметри

client

Тип: Client

Клиентският обект на които ще се изпращат данни.

data

Тип: string

Съобщение което ще се изпраща към клиента.

Пример

```
Server server = new Server(new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8181));
Client client = new Client(servet, socket);
server.SendMessage(e.Client, e.Message);
```

Клас Client

Синтаксис

public partial class Client

Публичен клас служеща за създаване на клиентски обекти. Този клас служи за запазване на данните на свързаните обекти към сървъра.

Полета

private Socket socket

Защитено поле за съхранява сокета на сървъра

private string guid

Защитено поле за съхранява на уникалния идентификатор на клиент

private Server server

Защитено поле за съхранява на сървъра към които е свързан текущия клиент

Свойства

public Socket Socket

Публично свойство, което връща сокета на текущия сървър.

public string GuId

Публично свойство, което връща стойността на уникалния идентификатор на клиента.

public Server Server

Публично свойство, което връща сървърът към който е свързан текущия клиент.

Конструктор

Име	Описание
Client(Server server, Socket socket)	Инициализиращ полетата на класа.

Пример

```
Server server = new Server(new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8181));
Client client = new Client(servet, socket);
client.Server.ReceiveMessage(client, “Здравей сървър аз съм клиент”);
```

Клас OnClientConnectedHandler

Синтаксис

public class OnClientConnectedHandler : EventArgs

Публичен клас служещ за създаване на събитиен обект. Този клас служи за свързване на обработващи методи при свързване на клиент към сървъра.

Полета

private Client client

Защитено поле за съхранява на клиента свързан към сървъра

Свойства

public Client Client

Публично свойство, което връща свързания към сървъра клиент.

Конструктор

Име	Описание
OnClientConnectedHandler(Client client)	Инициализиращ полетата на класа.

Пример

```
Server server = new Server(new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8181));
        server.OnClientConnected += (object sender, OnClientConnectedHandler e) =>
{
        Console.WriteLine("Клиентът с GUID: {0} се свърза!", e.Client.GuId);
};
```

Клас OnClientDisconnectedHandler

Синтаксис

public class OnClientDisconnectedHandler : EventArgs

Публичен клас служещ за създаване на събитиен обект. Този клас служи за свързване на обработващи методи при разкачане на клиент от сървъра.

Полета

private Client client

Защитено поле за съхранява на клиента свързан към сървъра

Свойства

public Client Client

Публично свойство, което връща свързания към сървъра клиент.

Конструктор

Име	Описание
OnClientDisconnectedHandler(Client client)	Инициализиращ полетата на класа.

Пример

```
Server server = new Server(new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8181));
server.OnClientDisconnected += (object sender, OnClientDisconnectedHandler e) =>
        {
           Console.WriteLine("Клиент с GUID {0} се разкачи", e.Client.GuId);
        }
```

Клас OnMessageReceivedHandler

Синтаксис

public class OnMessageReceivedHandler : EventArgs

Публичен клас служещ за създаване на събитиен обект. Този клас служи за създаване на обработващи методи при получаване на съобщение от клиент към сървъра.

Полета

private Client client

Защитено поле за съхранява на клиента свързан към сървъра.

private string message

Защитено поле за съхранява на съобщението получено от сървъра.

Свойства

public Client Client

Публично свойство, което връща свързания към сървъра клиент.

public string Message

Публично свойство, което връща изпратеното съобщение към сървъра.

Конструктор

Име	Описание
OnMessageReceivedHandler(Client client, string message)	Инициализиращ полетата на класа.

Пример

```
Server server = new Server(new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8181));

server.OnMessageReceived += (object sender, OnMessageReceivedHandler e) =>

{

Console.WriteLine("Изпратено съобщени: '{0}' от клиент с Guid: {1}", e.Message , e.Client.GuId);

};
```

Клас OnSendMessageHandler

Синтаксис

public class OnSendMessageHandler : EventArgs

Публичен клас служещ за създаване на събитиен обект. Този клас служи за създаване на обработващи методи при изпращане на съобщение от сървъра към клиента.

Полета

private Client client

Защитено поле за съхранява на клиента свързан към сървъра.

private string message

Защитено поле за съхранява на съобщението изпратено към клиента.

Свойства

public Client Client

Публично свойство, което връща свързания към сървъра клиент.

public string Message

Публично свойство, което връща изпратеното съобщение към клиента.

Конструктор

Име	Описание
OnSendMessageHandler(Client client, string message)	Инициализиращ полетата на класа.

Пример

```
Server server = new Server(new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8181));

server.OnSendMessage += (object sender, OnSendMessageHandler e) =>

{

Console.WriteLine("Изпратено съобщение: '{0}' до клиент с Guid {1}", e.Message, e.Client.GuId);

};
```

5.2. JavaScript библиотека

Обект WebSocketClient

Синтаксис

var WebSocketClient = function (connection)

Публичен обект служещ за създаване клиентска инстанция на WebSocket.

Полета

var url

Поле съдържащо адреса на сървъра.

var wsInitSelector

Поле съдържащо селектор на обекта, който инициализира началото на свързването на клиента към сървъра.

var wsSwndSelector

Поле съдържащо селектор на обекта, който инициира изпращането на съобщение от клиента към сървъра.

var OnOpen

Поле съдържащо метод за обратно извикване при осъществяване на връзка със сървъра.

var OnClose

Поле съдържащо метод за обратно извикване при затваряне на връзка със сървъра.

var OnError

Поле съдържащо метод за обратно извикване при настъпване на грешка по време на комуникацията със сървъра.

var OnMessage

Поле съдържащо метод за обратно извикване при получаване на съобщение от сървъра.

var OnSend

Поле съдържащо метод за образуване на съобщението към сървъра.

Пример

```
  <div>
        <a if="demo" class="demo" >Click here to connect</a>
        <a id="send" >Click here to send</a>
</div>

  <script type="text/javascript">

     document.addEventListener("DOMContentLoaded", function (event) {

        window.ws = new WebSocketClient({

           url: "ws://localhost:8181",

           wsInitSelector: document.getElementById("demo"),

           wsSwndSelector: document.getElementById("send"),

           OnSend: function () { return "Hello World"; }
        });

     });
```

Глава 6. Изводи и заключение. Предложение за развитие

Проектираната библиотека покрива основните функционалности, задължителни за да позволи разработването на приложения целящи да работят в реално време. Основното приложение на библиотеката, осигуряване на независима платформена рамка за проектиране на софтуерните системи работещи в реално време, е постигната посредством мрежовите възможности на библиотеката. Проектирането на софтуер посредством JavaScript, който е стандартизиран език за работа на различни крайни точки. Разработена в контекста на последната рамка на .NET позволява проектирането на приложения за хардуерни конфигурации с голям набор от процесорни архитектури. Библиотеката има приложение както в стандартни десктоп базирани приложения и в мобилните платформи и едноплаткови компютри.

Използваните стандарти при комуникацията определени от ISO и IEEE гарантират сигурност при боравенето с библиотеката заедно с това библиотеката предоставя прозрачен механизъм за проектиране достъпен за навлизащи в сферата на софтуера работещ в реално време разработчици.

При примерните приложения се наблюдава еднотипност на структурата на изграждане на приложенията. Което води до сигурност при започване на софтуерни проекти. Вече известните решения и изпитаният дизайн водят до по бърза и сигурна разработка, с по малка вероятност за критични грешки в софтуера.

Създаването на софтуерни продукти с библиотеката освен предимства създава и неудобствата, въпреки платформената независимост на JavaScript библиотеката е обвързаността си с NET Framework. Изпълнението на програмите в реално време зависи от две неща: правилното развитие на самата програма и надеждността на системата, на която тя работи. Ако операционната система е относително по-бавна от тази на програмата, тогава има по-малък шанс приложението да поддържа точността и предоставя качествени резултати. Междувременно, за програми и системи, които са силно съвместими една с друга, може да се постигне по-добро производителност и изпълнение на възложените задачи.

Бъдещето развитие на библиотеката.

Анализът на текущата реализация на библиотеката и постигнатите резултати от нейната примерна употреба предоставят възможности за подобрение свързани с нейната използваемост.

Планираното бъдещо развитие на библиотеката включва подобряване на класовете за изграждане на клиентски приложения базирани на HTML и JavaScript с добавяне на самогенериращи се модули и атрибути за боравене с JavaScript обектите. Подобряване на механизмите на изграждане на C# обекти с интегрирането на LINQ синтаксиса в библиотеката.

Използвани източници

Наков, С. и колектив; „Въведение в програмирането със C#“, София, 2011

Наков, С. и колектив; „Програмиране за .NET Framework – Част 1“, Варна, 2005;

Наков, С. и колектив; „Програмиране за .NET Framework – Част 2“, Варна, 2005;

Garcia-Molina H., Ullman J. D., Widom J.; “ Database Systems: The Complete Book ”, 2008

Gamma E.;„Design Patterns: Elements of Reusable Object-Oriented Software“, САЩ, 1995

Daigneau R.; „Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services“, 2008

Fowler M.; „Patterns of Enterprise Application Architecture“, 2002

Gomaa H.; Real-Time Software Design for Embedded Systems, 2016

Melnikov A.; RFC 6455 - The WebSocket Protocol - IETF Tools, 2011

Приложения

Приложение А: Програмен код

Програмен код на сървърната библиотеката

- Класове дефиниращи събитията
    - Събитие при свързване на клиент

```
namespace RealTimeToolkit.Events
{
   using System;

   /// <summary>
   /// Събитие, когато е свързан клиент
   /// </summary>
   public class OnClientConnectedHandler : EventArgs
   {
      ///<summary>
      ///Клиентът, свързан със сървъра
      ///</summary>
      private Client client;

      ///<summary>
      ///Извличане на клиента, който е свързан със сървъра
      ///</summary>
      ///<returns>Клиентът, който е свързан със сършъра</returns>
      public Client Client => client;

      ///<summary>
      ///Създаване на ново обект за обработка на събития, когато клиент е
свързан със съвъра
      ///</summary>
      ///<param name="client">Клиентът, който е свързан със сършъра</param>
      public OnClientConnectedHandler(Client client)
      {
         this.client = client;
      }
   }
}
```

- Събитие при прекратяване на връзка на клиента

```
namespace RealTimeToolkit.Events
{
   using System;

   ///<summary>
   ///Събитие, когато клиент напусне
   ///</summary>
   public class OnClientDisconnectedHandler : EventArgs
   {
      ///<summary>
      ///Клиентът, който е прекъснал връзката със сървъра
      ///</summary>
      private Client client;

      ///<summary>
      ///Извличане на клиента, който е бил прекъснал връзката със сървъра
      ///</summary>
      ///<returns>
      ///Клиентът прекъснал връзката
      ///</returns>
      public Client Client => client;

      ///<summary>
      ///Създаване на ново обект за обработка на събития, когато клиент е
прекъснал връзката със съвъра
      ///</summary>
      ///<param name="Client">
      ///Клиентът прекъснал връзката
      ///</param>
      public OnClientDisconnectedHandler(Client client)
      {
         this.client = client;
      }
   }
}
```

- Събитие при получаване на съобщение от клиента

```
namespace RealTimeToolkit.Events
{
   using System;

   ///<summary>
   ///Събитие, когато е получено съобщение
   ///</summary>
   public class OnMessageReceivedHandler : EventArgs
   {
      ///<summary>
      ///Клиентът, който изпраща съобщението към сървъра
      ///</summary>
      private Client client;

      ///<summary>
      ///Извличане на клиента, който е изпратил полученото съобщение към
сървъра
      ///</summary>
      ///<returns>
      ///Клиентът, изпраща съобщението
      ///</returns>
      public Client Client => client;

      ///<summary>
      ///Съобщението, изпратено от клиента към сървъра
      ///</summary>
      private string message;

      ///<summary>
      ///Извличане на съобщението, изпратено от клиента към сървъра
      ///</summary>
      ///<returns>
      ///Съобщението, изпратено от клиента
      ///</returns>
      public string Message => message;
```

```
        ///<summary>
    ///Създаване на ново обект за обработка на събития, когато клиент е
изпратено въобщение към сървъра
    ///</summary>
    ///<param name="client">
    ///Клиентът, който е изпратил съобщението
    ///</param>
    ///<param name="message">
    ///Съобщението, изпратено от клиента
    ///</param>
    public OnMessageReceivedHandler(Client client, string message)
    {
      this.client = client;
      this.message = message;
    }
  }
}
```

- Събитие при изпращане на съобщение до клиента

```
namespace RealTimeToolkit.Events
{
  using System;

  ///<summary>
  ///Събитие, когато съобщението е било изпратено на клиент
  ///</summary>
  public class OnSendMessageHandler : EventArgs
  {
    ///<summary>
    ///Клиентът, на когото е изпратено съобщението от сървъра
    ///</summary>
    private Client client;

    ///<summary>
    ///Извличане на клиента на когото е изпратено съобщението от сървъра
    ///</summary>
    ///<returns>Клиентът, на когото е изпратено съобщението</returns>
    public Client Client => client;
```

```
        ///<summary>
        ///Съобщението, изпратено до клиента
        ///</summary>
        private string message;

        ///<summary>
        ///Извича съобщението, изпратено до клиента
        ///</summary>
        ///<returns>
        ///Изпратеното съобщение
        ///</returns>
        public string Message => message;

        ///<summary>
        ///Създаване на ново обект за обработка на събития, когато е изпратено
съобщение
        ///</summary>
        ///<param name="client">
        ///Клиентът, на когото е изпратено съобщението
        ///</param>
        ///<param name="message">
        ///Съобщението, изпратено до клиента
        ///</param>
        public OnSendMessageHandler(Client client, string message)
        {
            this.client = client;
            this.message = message;
        }
    }
}
```

- Класове дефиниращи сървърните обекти
    - Обект на сървъра

```
namespace RealTimeToolkit
{
   using Events;
   using System;
   using System.Collections.Generic;
   using System.Net;
   using System.Net.Sockets;
   using System.Text;

   ///<summary>
   /// Обект за създаване на сървър
   ///</summary>
   public partial class Server
   {      /// <summary>Създаване на обект и стартиране на нов сървър в крайна
точка</summary>
          /// <param name="endPoint">райната точка на сървъра</param>
      public Server(IPEndPoint endPoint)
      {
         //Присвоява краина точка при валидн входен параметър
         if (EndPoint == null) return;
         this.endPoint = endPoint;

         //Създаване на нов сокет за клиента
         this.socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);

         clients = new List<Client>();

         Console.WriteLine("WebSocket Server Started\nListening on {0}:{1}\n",
EndPoint.Address.ToString(), EndPoint.Port);

         //Стартиране на сървъра
         start();
      }

      /// <summary>Сокет на сървъра</summary>
      private Socket socket;
```

```
        /// <summary>Извлича сокета на сървъра</summary>
        /// <returns>сокет на сървъра</returns>
        public Socket Socket => socket;

        /// <summary>Крайната точка на сървъра</summary>
        private IPEndPoint endPoint;

        /// <summary>Извлича крайната точка на сървъра</summary>
        /// <returns>Крайната точка на сървъра</returns>
        public IPEndPoint EndPoint => endPoint;

        /// <summary>Списък със свързаните клиенти към сървъра</summary>
        private List<Client> clients;

        #region Clients Getters

        /// <summary>Извлича свързан към сървъра клиент по номер</summary>
        /// <param name="index">Номер на свързания клиент</param>
        /// <returns>Свързан клиент с този номер или нищо ако не
съществува</returns>
        public Client GetConnectedClient(int index)
        {
            if (index < 0 || index >= clients.Count) return null;
            return clients[index];
        }

        /// <summary>Извлича свързан към сървъра клиент по
идентификатор</summary>
        /// <param name="guid">Идентификатор на свързания клиент</param>
        /// <returns>Свързан клиент с този идентификатор или нищо ако не
съществува</returns>
        public Client GetConnectedClient(string guid)
        {
            foreach (Client client in clients)
            {
                if (client.GuId == guid) return client;
            }
            return null;
        }
```

```
    /// <summary>Извлича свързан клиент по дадения сокет</summary>
    /// <param name="Socket">Сокет на клиента </param>
    /// <returns>Свързан клиент с този сокет или нищо ако не
съществува</returns>
    public Client GetConnectedClient(Socket Socket)
    {
      foreach (Client client in clients)
      {
        if (client.Socket == Socket) return client;
      }
      return null;
    }

    /// <summary>Извлича броя на свързаните клиенти със сървъра</summary>
    /// <returns>Броят на свързаните клиенти със сървъра</returns>
    public int ConnectedClientCount => clients.Count;

    #endregion

    #region Methods

    ///<summary>
    ///Стартира сървъра, след като е създаден сървърен обект
    ///</summary>
    private void start()
    {
      //Свързване на крайната точка
      Socket.Bind(EndPoint);
      Socket.Listen(0);

      //Задаване на метод за обратно извиване при свързване на клиент
      Socket.BeginAccept(connectionCallback, null);
    }

    /// <summary>
    /// Спиране на сървъра
    /// </summary>
    public void Stop()
    {  Socket.Close();
      Socket.Dispose();}
```

```
/// <summary>Метод за обратно извикване когато се свързва клиент към сървъра</summary>
    /// <param name="asyncResult">Състояние на асинхронна операция</param>
    private void connectionCallback(IAsyncResult asyncResult)
    {
        try
        {
            // Приема сокета на клиента които иска да се свържи към сървъра
            Socket clientSocket = Socket.EndAccept(asyncResult);

            // Прочита заявката на клиента които иска да се свържи
            byte[] handshakeBuffer = new byte[1024];
            int handshakeReceived = clientSocket.Receive(handshakeBuffer);

            //Извличане на ключа на заявката за свързване към сървъра и образуване на отговора
            string requestKey = Helpers.GetHandshakeRequestKey(Encoding.Default.GetString(handshakeBuffer));
            string hanshakeResponse = Helpers.GetHandshakeResponse(Helpers.HashKey(requestKey));

            // Изпращане на отговор за свързване на клиента който иска да се свържи
            clientSocket.Send(Encoding.Default.GetBytes(hanshakeResponse));

            // Създаване на нов обект за клиент
            Client client = new Client(this, clientSocket);
            // Добавяне на новия клиент в списъка с клиенти
            clients.Add(client);

            // Извикване на събитието за свързване при успешно свързване на клиент към сървъра
            if (OnClientConnected == null) throw new Exception("Server error: event OnClientConnected is not bound!");
            OnClientConnected(this, new OnClientConnectedHandler(client));

            // Задаване на метод за обратно извиване при свързване на клиент
            Socket.BeginAccept(connectionCallback, null);
        }
```

```
        catch (Exception Exception)
        {
            string error = string.Format("An error has occured while trying to accept a
connecting client.\n\n{0}", Exception.Message);
            Console.WriteLine(error);
            throw new Exception(error);
        }
    }

    /// <summary>Когато се получи съобщение от клиент се извиква събитието
за обработка</summary>
    /// <param name="client">Клиента изпратил съобщението</param>
    /// <param name="message">Съобщението изпратено от клиента</param>
    public void ReceiveMessage(Client client, string message)
    {
        if (OnMessageReceived == null) throw new Exception("Server error: event
OnMessageReceived is not bound!");
        OnMessageReceived(this, new OnMessageReceivedHandler(client, message));
    }

    /// <summary>Когато слиента прекъсне връзката се изпраща съобщение за
напускане</summary>
    /// <param name="client">Клиента които напуска</param>
    public void ClientDisconnect(Client client)
    {
        clients.Remove(client);

        if (OnClientDisconnected == null) throw new Exception("Server error:
OnClientDisconnected is not bound!");
        OnClientDisconnected(this, new OnClientDisconnectedHandler(client));
    }

    #endregion
```

```
#region Server Events

      /// <summary>Изпращане на съобщение до клиент</summary>
      /// <param name="client">Клиент до които се изпращат данни</param>
      /// <param name="data">Данните които се изпращат</param>
      public void SendMessage(Client client, string data)
      {
         // Създаване на пакет и изпращане на съобщението
         byte[] frameMessage = Helpers.GetFrameFromString(data);

         client.Socket.Send(frameMessage);

         if (OnSendMessage == null) throw new Exception("Server error: event
OnSendMessage is not bound!");
         OnSendMessage(this, new OnSendMessageHandler(client, data));
      }

      /// <summary>Събитие за изпращане на съобщение</summary>
      public event EventHandler<OnSendMessageHandler> OnSendMessage;

      /// <summary>Събитие за свързване на клиент</summary>
      public event EventHandler<OnClientConnectedHandler> OnClientConnected;

      /// <summary>Събитие за получаване на съобщение</summary>
      public event EventHandler<OnMessageReceivedHandler> OnMessageReceived;

      /// <summary>Събитие за прекратяване на връзка</summary>
      public event EventHandler<OnClientDisconnectedHandler>
OnClientDisconnected;

      #endregion
   }
}
```

- Обект на клиента

```
namespace RealTimeToolkit
{
   using Common;
   using System;
   using System.Net.Sockets;

   ///<summary>
   /// Обект на свързаните клиенти
   /// </summary>
   public partial class Client
   {

      #region Fields

      ///<summary>Сокет на свързания клиент</summary>
      private Socket socket;

      ///<summary>Извлича сокета на свързания клиент</summary>
      ///<returns>Сокета на клиента</return>
      public Socket Socket => socket;

      ///<summary>Уникален идентификатор на клиента</summary>
      private string guid;

      /// <summary>Извлича уникалния идентификатор на клиента</summary>
      /// <returns>Уникален идентификатор на клиента</returns>
      public string GuId => guid;

      /// <summary>Сървър към които е свързан клиента</summary>
      private Server server;

      /// <summary>Извлича сървъра към който е свързан клиента</summary>
      /// <returns>Сървъра към които е свързън клиента</returns>
      public Server Server => server;

      #endregion
```

```
/// <summary>Създава обект за свързан клиент</summary>
      /// <param name="server">Сървър към които е свързан клиента</param>
      /// <param name="socket">Сокета на клиента</param>
      public Client(Server server, Socket socket)
      {
         this.server = server;
         this.socket = socket;
         this.guid = Helpers.CreateGuid("client");

         socket.BeginReceive(new byte[] { 0 }, 0, 0, SocketFlags.None, messageCallback,
null); }
      #region Methods
      /// <summary>Метод за обратно извикване при изпращане на
съобщение</summary>
      private void messageCallback(IAsyncResult AsyncResult)
      {
         try {
            Socket.EndReceive(AsyncResult);

            byte[] messageBuffer = new byte[255];
            int bytesReceived = Socket.Receive(messageBuffer);

            OpcodeType opcode = Helpers.GetFrameOpcode(messageBuffer);

            if (opcode == OpcodeType.ClosedConnection)
            {
               Server.ClientDisconnect(this); return;
            }

            Server.ReceiveMessage(this, Helpers.GetDataFromFrame(messageBuffer));
            Socket.BeginReceive(new byte[] { 0 }, 0, 0, SocketFlags.None,
messageCallback, null);
         }
         catch (Exception)  {
            Socket.Close();
            Socket.Dispose();
            Server.ClientDisconnect(this);
         } }

      #endregion } }
```

- Помощни методи

```
namespace RealTimeToolkit.Common
{
   /// <summary>
   /// Информация за пакета на комуникация
   /// </summary>
   public struct FrameMaskData
   {
      public int DataLength, KeyIndex, TotalLenght;
      public OpcodeType Opcode;

      public FrameMaskData(int DataLength, int KeyIndex, int TotalLenght,
OpcodeType Opcode)
      {
         this.DataLength = DataLength;
         this.KeyIndex = KeyIndex;
         this.TotalLenght = TotalLenght;
         this.Opcode = Opcode;
      }
   }
}

namespace RealTimeToolkit.Common
{
   /// <summary>
   /// Енумерация за типовете пакети
   /// </summary>
   public enum OpcodeType
   {
      Fragment = 0,
      Text = 1,
      Binary = 2,
      ClosedConnection = 8,
      Ping = 9,
      Pong = 10
   }
}
```

```
namespace RealTimeToolkit
{
   using Common;
   using System;
   using System.Security.Cryptography;
   using System.Text;

   /// <summary>
   /// Помощни методи за работа с пакетите
   /// </summary>
   public static class Helpers
   {
      /// <summary>Извличана на WebSocket пакет</summary>
      /// <param name="data">Пакета за трансформиране</param>
      /// <returns>Информация за пакета</returns>
      public static FrameMaskData GetFrameData(byte[] data)
      {
         // Get the opcode of the frame
         int opcode = data[0] - 128;

         // If the length of the message is in the 2 first indexes
         if (data[1] - 128 <= 125)
         {
            int dataLength = (data[1] - 128);
            return new FrameMaskData(dataLength, 2, dataLength + 6,
(OpcodeType)opcode);
         }

         // If the length of the message is in the following two indexes
         if (data[1] - 128 == 126)
         {
            // Combine the bytes to get the length
            int dataLength = BitConverter.ToInt16(new byte[] { data[3], data[2] }, 0);
            return new FrameMaskData(dataLength, 4, dataLength + 8,
(OpcodeType)opcode);
         }

         // If the data length is in the following 8 indexes
         if (data[1] - 128 == 127)
         {
```

```
// Get the following 8 bytes to combine to get the data
            byte[] combine = new byte[8];
            for (int i = 0; i < 8; i++) combine[i] = data[i + 2];

            // Combine the bytes to get the length
            //int dataLength = (int)BitConverter.ToInt64(new byte[] { Data[9], Data[8],
Data[7], Data[6], Data[5], Data[4], Data[3], Data[2] }, 0);
            int dataLength = (int)BitConverter.ToInt64(combine, 0);
            return new FrameMaskData(dataLength, 10, dataLength + 14,
(OpcodeType)opcode);
         }

         // error
         return new FrameMaskData(0, 0, 0, 0);
      }

      /// <summary>Извлича кода на пакета</summary>
      /// <param name="frame">Пакета от които ще се извлече код</param>
      /// <returns>Кода на пакета</returns>
      public static OpcodeType GetFrameOpcode(byte[] frame)
      {
         return (OpcodeType)frame[0] - 128;
      }

      /// <summary>Извлича декодирано съобщение</summary>
      /// <param name="Data">пакет за декодиране</param>
      /// <returns>Декодиран пакет</returns>
      public static string GetDataFromFrame(byte[] Data)
      {  FrameMaskData frameData = GetFrameData(Data);
         byte[] decodeKey = new byte[4];
         for (int i = 0; i < 4; i++) decodeKey[i] = Data[frameData.KeyIndex + i];

         int dataIndex = frameData.KeyIndex + 4;
         int count = 0;
         for (int i = dataIndex; i < frameData.TotalLenght && Data.Length > i; i++)
         { Data[i] = (byte)(Data[i] ^ decodeKey[count % 4]);
            count++;
         }
         return Encoding.Default.GetString(Data, dataIndex, frameData.DataLength); }
```

```
        /// <summary>Проверка за валидноста на масив</summary>
        /// <param name="buffer">Масива за вслидиране</param>
        /// <returns>статуса на валидацията</returns>
        public static bool GetIsBufferValid(ref byte[] buffer)
        {
            if (buffer == null) return false;
            if (buffer.Length <= 0) return false;

            return true;
        }

        /// <summary>Трансформиране на съобщене в пакет за
предаване</summary>
        /// <param name="message">Съобщениоето което ще се
трансформира</param>
        /// <param name="opcode">Кода на пакета</param>
        /// <returns>Резултатния пакет за изпращане</returns>
        public static byte[] GetFrameFromString(string message, OpcodeType opcode =
OpcodeType.Text)
        {
            byte[] response;
            byte[] bytesRaw = Encoding.Default.GetBytes(message);
            byte[] frame = new byte[10];

            int indexStartRawData = -1;
            int length = bytesRaw.Length;

            frame[0] = (byte)(128 + (int)opcode);
            if (length <= 125)
            {
                frame[1] = (byte)length;
                indexStartRawData = 2;
            }
            else if (length >= 126 && length <= 65535)
            {
                frame[1] = (byte)126;
                frame[2] = (byte)((length >> 8) & 255);
                frame[3] = (byte)(length & 255);
                indexStartRawData = 4; }
```

```
        else {
            frame[1] = (byte)127;
            frame[2] = (byte)((length >> 56) & 255);
            frame[3] = (byte)((length >> 48) & 255);
            frame[4] = (byte)((length >> 40) & 255);
            frame[5] = (byte)((length >> 32) & 255);
            frame[6] = (byte)((length >> 24) & 255);
            frame[7] = (byte)((length >> 16) & 255);
            frame[8] = (byte)((length >> 8) & 255);
            frame[9] = (byte)(length & 255);

            indexStartRawData = 10;
        }

        response = new byte[indexStartRawData + length];

        int i, reponseIdx = 0;

        for (i = 0; i < indexStartRawData; i++)
        {
            response[reponseIdx] = frame[i];
            reponseIdx++;
        }

        for (i = 0; i < length; i++)
        {
            response[reponseIdx] = bytesRaw[i];
            reponseIdx++;
        }

        return response;
    }
```

```
/// <summary>Извличане на ключ за отговор</summary>
      /// <param name="Key">Ключ на отговора</param>
      /// <returns></returns>
      public static string HashKey(string Key)
      {
         const string handshakeKey = "258EAFA5-E914-47DA-95CA-C5AB0DC85B11";

         string longKey = Key + handshakeKey;

         SHA1 sha1 = SHA1.Create();
         byte[] hashBytes = sha1.ComputeHash(Encoding.ASCII.GetBytes(longKey));

         return Convert.ToBase64String(hashBytes);
      }

      /// <summary>Извличане на http отговор за клинт</summary>
      /// <param name="key">Ключ на отговора</param>
      /// <returns></returns>
      public static string GetHandshakeResponse(string key)
      {
         return string.Format("HTTP/1.1 101 Switching Protocols\nUpgrade:
WebSocket\nConnection: Upgrade\nSec-WebSocket-Accept: {0}\r\n\r\n", key);
      }

      /// <summary>Извличане на ключа от заявката за свързване</summary>
      /// <param name="httpRequest">Http заявката</param>
      /// <returns></returns>
      public static string GetHandshakeRequestKey(string httpRequest)
      {
         int keyStart = httpRequest.IndexOf("Sec-WebSocket-Key: ") + 19;
         string key = null;

         for (int i = keyStart; i < (keyStart + 24); i++)
         {
            key += httpRequest[i];
         }
         return key;
      }
```

```
        /// <summary>Създаване на уникален идентификатор</summary>
        /// <param name="prefix">Префикс на идентификатора</param>
        /// <param name="Length">Дължина на идентификатора</param>
        /// <returns>Резултата от съзгаването на идентификатор</returns>
        public static string CreateGuid(string prefix, int length = 16)
        {
            string final = null;
            string ids = "0123456789abcdefghijklmnopqrstuvwxyz";

            Random random = new Random();

            for (short i = 0; i < length; i++)
                final += ids[random.Next(0, ids.Length)];

            if (prefix == null) return final;

            return string.Format("{0}-{1}", prefix, final);
        }
    }
}
```

Програмен код на клиентската библиотеката

```
var WebSocketClient = function (cnn)
{
    var self = this;
    self.url = cnn.url;
    self.wsInitSelector = cnn.wsInitSelector;
    self.wsSwndSelector = cnn.wsSwndSelector

    var initWS = function (url) {
        var wsImpl = window.WebSocket || window.MozWebSocket;
        ws = new wsImpl(url);

        ws.onopen = (cnn.OnOpen)
          ? cnn.OnOpen
          : function () { console.log('WebSocket connection Open'); }
```

```
    ws.onclose = (cnn.OnClose)
        ? cnn.OnClose :
        function (e) { console.log('WebSocket connection Close'); };

    ws.onerror = (cnn.OnError)
        ? cnn.OnError :
        function (e) { console.log('WebSocket erron: ' + e.data); };

    ws.onmessage = (cnn.OnMessage)
        ? cnn.OnMessage :
        function (e) {
            debugger
            console.log('WebSocket message: ' + e.data);
        };

    return ws;
}
self.OnSend = function send() {
    if (cnn.OnSend) {
        ws.send(cnn.OnSend);
    } else {
        ws.send("Hello World");
    }
};
if (self.wsInitSelector) {
    self.wsInitSelector.onclick = function () {
        self.ws = initWS(self.url);
    };
} else {
    self.ws = initWS(self.url);
}
if (self.wsSwndSelector) {
    self.wsSwndSelector.onclick = function () {
        self.OnSend();
    };
} else {
    self.OnSend();
}
return self; }
```

```
    document.addEventListener("DOMContentLoaded", function (event) {
      window.ws = new WebSocketClient({
        url: "ws://localhost:8181",
        wsInitSelector: document.getElementById("demo"),
        wsSwndSelector: document.getElementById("send"),
        OnSend: function () { return "Hello World"; }
      });
    });
```

Класове от Примерните приложения

```
  public class StudentWerker
  {
    private IStudentManager source;
    public List<Student> AllStudents()
    {
      return this.source.GetAllStudents();
    }
    Public bool CreateStudent(Person person)
    {
      return this.source.Add(new Student {
              FirstName = person.FirstName,
              LastName = person.LastName,
              Egn = person.Egn
      });
    }
  }

public static class StudentManager: IStudentManager
  {
    public static List<Student> GetAllStudent()
    {
      List<Student> students = new List<Student>();
      using (IUnitOfWork unitOfWork = new UnitOfWork(new
StudentSystemDbContext())) {
        Repository<Student> repository = new Repository<Student>(unitOfWork);
        students = repository.Get().ToList();
      }
      return students; }
```

```
      public static bool Add(Student student)
      {
         bool success = false;
         using (IUnitOfWork unitOfWork = new UnitOfWork(new
StudentSystemDbContext()))
         {
            Repository<Student> repository = new Repository<Student>(unitOfWork);
            student.UnitState = UnitState.Added;
            student = repository.Create(student);
            unitOfWork.Commit();
            success = true;
         }
         return success;
      }
   }

private static void Server()
{
         Server server = new Server(new IPEndPoint(IPAddress.Parse("127.0.0.1"),
8181));
         server.OnClientConnected += (object sender, OnClientConnectedHandler e) =>
         {
                  Console.WriteLine("Свързване на клиент {0}", e.Client.GuId);
         };
         server.OnClientDisconnected += (object sender, OnClientDisconnectedHandler
e) =>
         {
         Console.WriteLine("Разкачане на клиент {0}", e.Client.GuId);
      };
         server.OnMessageReceived += (object sender, OnMessageReceivedHandler e)
=>
         {
                  ReadeMessage(e.Client.Server, e.Message);
      };
         server.OnSendMessage += (object sender, OnSendMessageHandler e) =>
      {
         Console.WriteLine("Беша изпратено съобщение до: {1}", e.Message,
e.Client.GuId);
         };}
```

```
public class Neuron
{
   private double error;
   private double input;
   private double learnRate = 0.5;
   private double output = double.MinValue;
   private List<Weight> weights;
   public Neuron(Layer inputs, Random rnd)
   {
      weights = new List<Weight>();
      foreach (Neuron input in inputs)
      {
         Weight w = new Weight();
         w.Input = input;
         w.Value = rnd.NextDouble() * 2 - 1;
         weights.Add(w);
      }
   }

   public void Activate()
   {
      error = 0;
      input = 0;
      foreach (Weight w in weights)
      {
         input += w.Value * w.Input.Output;
      }
   }

   public void CollectError(double delta)
   {
      if (weights != null)
      {
         error += delta;
         foreach (Weight w in weights)
         {
            w.Input.CollectError(error * Derivative * w.Value);
         }
      } }
```

```
    public void AdjustWeights()
    {
        for (int i = 0; i < weights.Count; i++)
        {
            weights[i].Value += learnRate * error * Derivative * weights[i].Input.Output;
        }
    }

    private double Derivative
    {
        get
        {
            double activation = Output;
            return activation * (1 - activation);
        }
    }

    public double Output
    {
        get
        {
            if (output != double.MinValue)
            {
                return output;
            }
            return 1.0 / (1.0 + Math.Exp(-input));
        }
        set
        {
            output = value;
        }
    }
}

public class Weight
{
    public Neuron Input;
    public double Value;
}
```