15. Модели на разпределени софтуерни архитектури. Среди и протоколи за разпределени приложения. (РСА)

Анотация: Изложението по въпроса трябва да включва следните по-съществени елементи:

1. Параметри на паралелната и разпределена обработка. Метрика. Методи за анализ.
2. Модели на разпределените софтуерни архитектури – процедурни, обектни, потокови, контекстни, йерархични, асинхронни и интерактивни модели на софтуерната архитектура. Структури, организация, компоненти, приложение.
3. Организация на разпределените приложения – клиент-сървър, двуслойни, трислойни и n-слойни модели. p2p. Сървъри за приложения и web-сървъри. Метасистеми и грид. Сервизно-, модело- и аспектно-ориентирани архитектури. Софтуерни агенти.

1. Параметри на паралелната и разпределена обработка

1.1. Паралелна обработка

Паралелната обработка наричаме обработка на данни, която се извършва от няколко процеси или нишки, които общо ще наричаме работници. Тя е разпределена, ако процесите са разположени на няколко машини, които общо ще наричаме изчислителни възели или само възели. При паралелната обработка целта е ускорение, а при разпределената – освен ускорение и функционално разслояване.

1.2. Параметри на паралелната и разпределената обработка

1.2.1. Паралелизъм

Паралелизмът обикновено го отбелязваме с p и с него означаваме броя на работниците, които работят по време на изпълнение на дадена програма. Той се дели на два вида – софтуерен и хардуерен. Софуерният паралелизъм е характеристика на програмите – колко работници се стартират. Хардуерният паралелизъм зависи от броя процесори (вкл. ядра), върху които се изпълнява задачата. Ускорението от софтуерния паралелизъм е ограничено от хардуерния – не може да очакваме 8 пъти ускорение от 8 нишки, ако машината има по-малко от 8 ядра, дори и ако едно ядро планира повече от една нишка.

1.2.2. Грануларност

Грануларност е степента на декомпозиция на данните, т.е. на колко подзадачи се разбива цялата задача. Различават се три вида грануларност, които се разграничават не по числови параметри, а по ефекта които постигат (тъй като числата са различни за различни те задачи и машини):

- **едра** – малък брой подзадачи. Малко време за разпределение на задачите на работниците, за сметка на лошо балансиране в средния случай;
- **фина** – голям брой подзадачи. Задачите са достатъчно голям брой, за да се счита, че всички работници ще получат по равен брой задачи в средния случай, но за сметка на много време по разпределението им;
- **средна** – компромис между горните два вида. Това е златната среда, към която алгоритъмът за паралелна обработка трябва да се стреми.

1.2.3. Балансиране

Балансирането е такова разделение на подзадачите, при което всеки процес да получи равен на брой задачи в очакване. Доброто балансиране ще гарантира, че в очакване подзадачите ще завършат по едно и също време. Това е желана цел, тъй като времето за работа на един паралелен алгоритъм е равно на времето за работа на най-бавния работник в него. Следователно не е желано един работник да приключи работа много по-рано от друг.

Балансирането може да се раздели по няколко оси:

- Време на осъществяване
  - статично – подзадачите се дефинират като предварителна операция на алгоритъма, след което се стартира паралелизма.

    - динамично – При постъпването на нова подзадача, динамично се определя кой работник да я поеме. Препоръчително е, когато нови задачи постъпват по време на изпълнение.
- Структура на разбиване
    - централно
    - разпределено (например верига, решетка, хиперкуб)
    - йерархично

1.2.4. Синхронизация

Синхронизацията описва обема на информация, който процесите използват за да си комуникират помежду си и чрез какви механизми се осъществава тази комуникация. Според начина ѝ на осъществяване разглеждаме три вида алгоритми:

- **Асинхронни** – не обменят данни между различните си работници. Никой от процесите не знае за съществуването на останалите процеси.
- **Локално синхронни** – всеки работник си има дефинирани съседи, с които обменя информация. Приема информация от едни и предава на други, като тези от които приема и тези на които предава са точно дефинирани.
- **Глобално синхронни** – всеки работник може да си комуникира с всеки останал (работниците образуват клика – пълен граф).

1.2.5. Тип на декомпозицията

Налични са два вида декомпозиция на задачата:

- **По данни** – SPMD (Single Program, Multiple Data)
- **По управление** – MPMD (Multiple Program, Multiple Data)

При SPMD ползваме една и съща програма, която стартираме върху различни данни, а при MPMD имаме няколко различни програми.

1.3. Метрики

1.3.1. Ускорение и ефикасност

Ускорение (acceleration) пресмятаме чрез формулата $S_p = \frac{T_1}{T_p}$, където T_1 е времето за последователна обработка, а T_p е времето за паралелна обработка при паралелизъм p. Поради наличие на комуникационни и синхронизационни забавяния, обикновено $S_p \in (1, p)$, но има и изключения.

Ефективност (efficiency) е нормализирано ускорение, което пресмятаме чрез формулата $E_p = \frac{S_p}{p}$, като обикновено стойностите са от интервала $(0, 1)$ и за това е удобно да се измерват в проценти.

1.3.2. Суперлинейна аномалия

Този тип аномалия имаме когато $S_p > p$. Причината за суперлинейната аномалия е такава фрагментация (декомпозиция на данните), която позволява обработката на алгоритъма ни да се изпълни без скритите операции по подготовка на кеша. Тази аномалия се дължи на управлението на кеша. При пускане на повече нишки може да се включат повече хардуерни ядра, като всяко от тях има собствен L1 кеш. Допълнително, ако областите от данни се припокриват между нишки, то дадено парче от данни се зарежда само веднъж от бавната оперативна памет и след това се достъпва бързо от споделения L2 кеш.

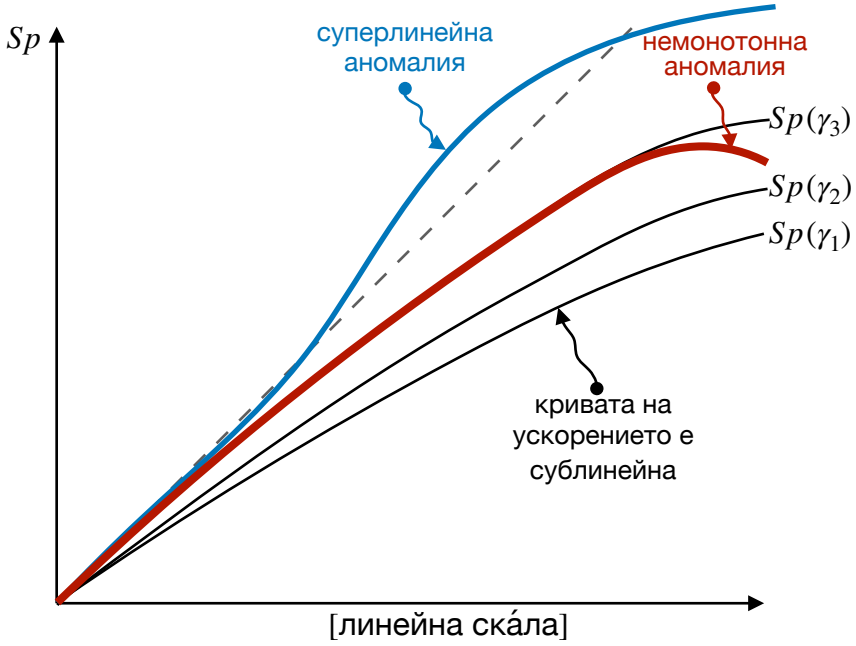
1.3.3. Нелинейна аномалия

Този тип аномалия имаме когато $S_{p+1} < S_p$ за някое p. Тоест програмата се забавя при увеличаване на паралелизма, което е контра-интуитивно. Причините за това може да са:

- алгоритмични;
- надвишаване на хардуерния паралелизъм, заради което трупаме единствено време за смяна на контекста;
- балансиране и разпределяне, но работата не се върши по-бързо.

1.3.4. Графика на ускорението и ефикасността

Обикновено за дадена система се изобразяват фамилия от криви, които показват какво е нейното ускорение (и/или ефикасност), в зависимост от някакви параметри. Най-често използваният параметър е грануларността. Примери за графики, които включват и двете аномалии:



1.3.5. Закон на Амдал

Всеки алгоритъм се състои от серийна част и част, която може да бъде паралелна. Това налага горната граница на възможното ускорение, която се описва чрез закона на Амдал.

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}, \text{ където:}$$

- $S_{latency}$ е теоритичното ускорение на изпълнението на цялата задача;
- s е ускорението на секцията, в която прилагаме паралелизма;
- p е фракцията от времето на задачата, заемана от паралелизираната секция, при серийно изпълнение на алгоритъма.

1.4. Сравнение между паралелна и разпределена обработка

Параметър / Тип обработка	Паралелна	Разпределена
Инфраструктура	мултипроцесор	мултикомпютър
Обмен на данни	Обща памет и механизъм за синхронизация	Предаване на съобщения чрез middleware
Обичайна декомпозиция	SPMD	MPMD
Примери за синхронни задачи	nBody, WaTor	Client-server, p2p, някои конвейери
Примери за асинхронни задачи	фрактали	някои конвейери

2. Модели на разпределените софтуерни архитектури и техните структури, организация, компоненти и приложение

2.1. Дефиниции

Софтуерната архитектура дефинира какви са съставните компоненти и процеси на една софтуерна система, как те са структурирани и си взаимодействат. Моделите на софтуерни архитектури се представят чрез диаграми, описващи дизайна на системата. Под дизайн се разбира съвкупността от декомпозицията на съставните компоненти, техните функции, прилагания архитектурен стил и качествени атрибути.

2.2. Процедурни модели

Процедурните архитектурни модели се базират на разпределена обработка чрез отдалечено извикване на процедури (remote procedure call (RPC)) или Remote Method Invocation (RMI). Представлява извикване на процедура на един възел от друг възел, като това е прозрачно за извикващия възел, тоест е като локална за него процедура.

2.3. Обектни модели

Обектно-ориентираните архитектурни модели се основават на разделянето на отговорностите на системата в индивидуални повторно използваеми и независими обекти, които си сътрудничат. Основните принципи на тези модели са:

- Абстракция – опростяването на решението на реален проблем чрез дефиниране на подходящи класове;
- Композиция – изграждането на обекти от други обекти;
- Наследственост – автоматичното присвояване на функционалностите на едни обекти на други обекти и тяхното разширение и промяна;
- Енкапсулация – скриването на ненужните за потребителя детайли на някои обекти.
- Полиморфизъм – възможността на даден обект да се държи като друг, запазвайки логиката на собственото си поведение.

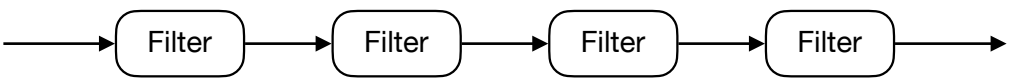
2.4. Потокови модели

Потоковите архитектурни модели разглеждат системата като последователност от трансформации на постъпваща като от линеен поток информация. Всеки компонент трансформира входните си данни в изходни. Топологията на пренос на данните между тях се задава експлицитно чрез блок-диаграми. Модулите поддържат само интерфейс по данни, но не и контролен, като те не се адресират директно, а чрез предаваните данни.

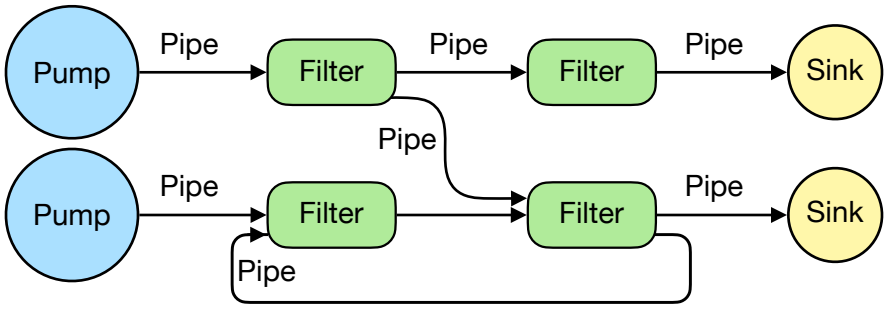
2.4.1. Видове

Можем да разграничим няколко вида потокови архитектури в зависимост от това кога и как се извършва обработката на данните от всеки модул:

- Пакетна обработка (Batch Sequential) – Модулите се извикват в последователен ред един след друг, като изходът от един модул се предава като вход на следващия. Пакетите от данните, които се обработват от всеки модул, са записани под формата на временни файлове, които служат за комуникация между два последователни в изпълнението си модули. Ключовото е, че обработката от следващия модул може да е отложена във времето. Най-често този модел се използва за асинхронна обработка на данни, като се цели оптимално използване на ресурси за сметка на паралелизъм и интерактивност. Примери: профилактика на системата; безплатна обработка, която се случва само когато системата не е натоварена от платени потребители.



- Филтриране и канали (Pipe and Filter) – Обработката се случва в онлайн режим. Всеки процес (отговарящ на модул) е активен и е свързан директно със съседите си. Всеки процес започва да предава изходните данни към следващия филтър преди да е свършил с обработката на всичките данни. Този начин за обработка на данни позволява паралелизъм (конвейеризация).



Pipes and Filters

- Контролни процеси (Process Control) – като Pipe and Filter, но с добавени ограничения върху времето за изпълнение (realtime constraints). Понякога между модулите се добавя и пряк физически интерфейс. Както при всяка система в реално време, целта е да се гарантира време за изпълнение. Примери: управление на физически процеси, например при автомобилите и самолетите.

2.5. Контекстни (Data Centric) модели

Контекстните архитектурни модели се характеризират с централизирано хранилище за данните, от където те са достъпни за всички останали компоненти на системата. Декомпозицията на модулите се състои от модул за управление на достъпа до данните и агенти, които извършват операции върху тях.
Интерфейсът между агентите и данните може да бъде явен (при RMI и RPC) или имплицитен (например транзактивен). Оригиналният модел не предвижда комуникация между агентите.

2.5.1. Типове

Съществуват два основни контекстни модела:

- Хранилище (Repository), където агентите са активни, т.е. те инициират взаимодействието;
- Черна дъска (Black board), където инициативата е на модула с данните, а агентите се абонират за събития (event listeners). Събитие настъпва при промяна в данните. Използват се при AI и мултидисциплинарните системи.

2.6. Йерархични модели

Йерархичните архитектурни модели декомпозират системата на йерархични модули, като функциите се групират по йерархичен принцип на няколко нива. Координацията обикновено е между модули от различни нива (вертикална свързаност) и се базира на явни (т.е. „заявка-отговор“) обръщения. Слоевете делегират своята работа на услуги от съседни по-ниски нива. Пълна прозрачност между нивата се постига при запазване на свързващите интерфейси.

Ще разгледаме два основни типа йерархични модели:

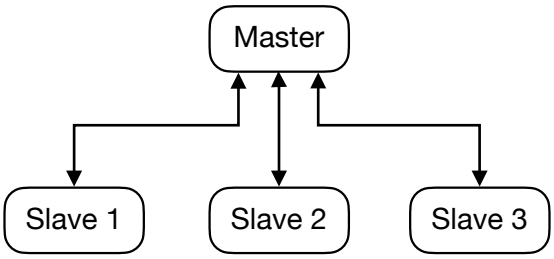
2.6.1. Слоест модел

При този модел, имаме последователност от слоеве, като всеки предоставя услуги на по-долните. Използван при много ОС като Unix, GNU и Windows. Също така протоколните стекове OSI и TCP/IP са слоести архитектури. Целта на този тип архитектура е бързо проектиране на (преносими) приложения.

2.6.2. Master/Slaves

Master/Slaves представлява разбиване на главна програма и подчинени програми. Главната програма разпределя работата на подчинените и след това е възможно да ги координира. Този модел може да цели две неща:

- Отказоустойчивост и надеждност: постига се, като главната програма проверява резултата от подчинените, както и статуса им.
- Ускорение: постига се чрез балансиране на натоварването между подчинените програми за по-добро ускорение.



Master-Slave Architecture

2.7. Асинхронни модели

Асинхронните модели се базират на имплицитни асинхронни обръщения между обслужващите процеси, т.е. обмяна на съобшения – message passing.

2.7.1. Producer/Consumer

Асинхронният обмен може да бъде от тип Producer/Consumer, където обменът на данни е 1:1. Има следните разновидности:

- интерактивен (online), където няма буфериране и двата процеса са активни. Често се ползва адресна книга, която позволява процесите да се адресират по име, а не да трябва да знаят своя мрежови адрес – например DNS.
- отложен (offline), където обмена на съобщенията се опростява посредством процес-буфер, който служи като пощенска кутия. Това позволява консуматора да не е активен по време на изпращане на съобщения, както и обратно.

2.7.2. Publish/Subscriber

Publish/Subscriber също е вид асинхронен обмен на съобщения, където обменът на данни е 1:много. Има следните разновидности:

- topic-based, където съобщенията се изпращат до topic-и, които са именувани канали. Издателят е отговорен за дефинирането на topic-ите, а абонатите получават от всички topic-и, за които са се абонирали.
- context-based, където съобщенията се изпращат до абонат, ако атрибутите на съдържанието отговарят на ограниченията дефинирани от абоната. Абонатът е отговорен за класификацията на съобщенията.

2.8. Интерактивни модели

Интерактивните модели поддържат интензивен потребителски интерфейс. За целта декомпозицията на системата е на 3 функционални модула:

- модул за представяне (изглед), имплементиращ потребителския интерфейс. Използва се за представяне (в това число графично и мултимедийно) на изходните данни и намеса на потребителите в обработката (т.е. вход за данни и контрол);
- модул на данни, поддържащ данновия модел заедно с базова функционалност за обработката на данните;
- модул за управление, който отговаря за системни комуникации, управление на процесите, инициализиране и конфигуриране на модули данни, както и управление на изгледи.

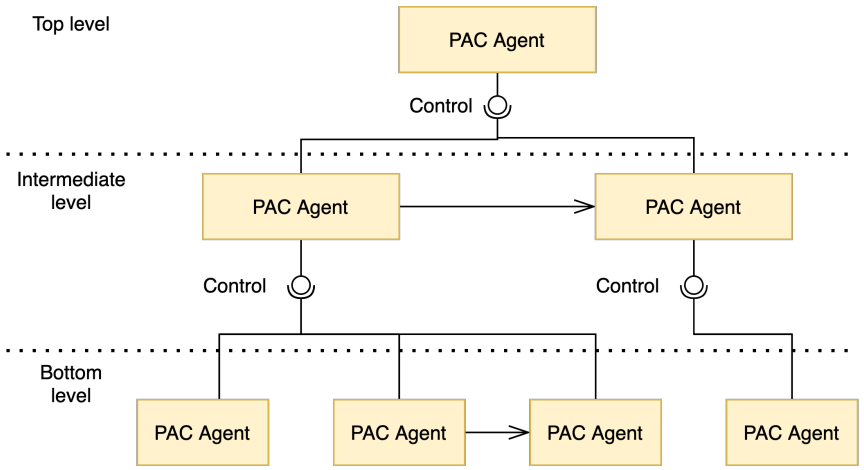
2.8.1. Кратко обяснение

Гореописаните компоненти се имплементират от MVC (Model View Controller). PAC (Presentation-Abstraction-Control) е MVC за многоагентна система, например ако има много източници на данни. Всеки агент изпълнява всеки от PAC атрибутите.

2.8.2. Подробно обяснение

Има две категории интерактивни софтуерни архитектури – PAC и MVC.

- PAC е йерархично (разслоено) и разпределено управление, при което системата се формира от набор кооперирващи агенти на три нива - базово ниво, състоящо се от агенти работещи върху общи данни и бизнес логика, средно ниво, състоящо се от агенти координатори на изгледите и ниво на изгледите за обработка на локални данни. Всеки агент имплементира P, A и C компоненти.



- при MVC агентите са равнопоставени.

3. Организация на разпределените приложения

При наличие на множество потребители, като системи за масово обслужване, се използва MP-MD декомпозиция.

3.1. Клиент сървър

Клиент-сървър е класически MPMD модел за адаптиране на ограничена клиентска инфраструктура към сложни задачи. Процесите се класифицират като два основни:

- Сървър, който е реактивен процес, имплементиращ приложната логика и данновия модел под формата на конкретен набор от услуги.
- Клиент, който е активен интерфейсен процес, използващ услугите на сървъра.

Сървърите са специално проектирани устройства, направени да обработват определен товар. Те са свързани с опорни мрежи (backbone). Клиентските устройства (които наричаме и клиентска инфраструктура), както и връзките им с мрежата, са произволни.

Има два основни вида клиенти – тънки (thin), които делегират цялата работа на сървъра, и дебели (fat), които изпълняват част от задачата.

Предимства	Недостатъци
технологично специализиране	унификация на клиентската инфраструктура
инфраструктурна гъвкавост (скалиране)	по-трудна защита на достъпа до данните
преизползване	скалируемост на сървъра
еволюция без участието на потребителя	скъпа поддръжка, профилактика и тестване

Клиент-сървър модела е много разпространен, тъй като той е заложен в основата на IP протокола.

3.1.1. Трислойна архитектура

При три-слойната архитектура сървърната част се декомпозира допълнително на сървър на приложението, в което се обработват заявките, и сървър на данните. Основната цел е допълнително функционално разслояване.

3.1.2. N-слойна архитектура

N-слойната архитектура включва надграждане над клиент-сървър архитектурата, където компонентите са разделени на няколко слоя. Можем да си представим, че слоевете са вертикално подредени и компонентите на даден слой са клиенти на сървъри от съседния долен слой.

При N-слойните архитектури редовно се срещат многослойни сървъри, където сървърната част е декомпозирана на поне два слоя, които се класифицират като:

- междинни слоеве, които се намират между интерфейсния и вътрешния слой, имплементиращ бизнес логиката.
- вътрешен слой, най-долния слой, който обичайно е персистентен или се използва за комуникации.

Очевидно щом е надграждане над клиент-сървър архитектурата, то N-слойната архитектура разполага със същите недостатъци и предимства, но по-силно изразени.

3.1.3. Брокерен модел

При този модел се добавя допълнителен модул – брокер. Той предоставя функционално еквивалентни алтернативи на дадени услуги. Позволява и услугите да се класифицират по своите функционални и нефункционални атрибути.

Пример: Брокер за принтиране, на който можем да кажем, че искаме принтирането да е черно бяло (функционално) и да се извърши от принтера, с най-къса опашка (нефункционално).

3.2. Peer-to-Peer архитектури

Peer-to-Peer (p2p) архитектурата се разгръща като наслоена (overlay) мрежа върху съществуваща клиентска част от мрежова инфраструктура. Всеки участник е еднакво привилегирован и заделя част от ресурсите си (cpu, hd и други), която да се използва от останалите участници в мрежата без нуждата от каквато и да е централна координация. За разлика от класическата архитектура клиент-сървър, при p2p всеки участник изпълнява функцията и на клент и на сървър за другите участници. p2p се реализира чрез децентрализирана топология, самоорганизация, инцидентна (ad hoc) и динамична свързаност и наличност на възлите и процесите в тях, скалируемост, отказоустойчивост (fault tolerance), анонимност на споделянето и специална юридическа регулация.

3.2.1. Неструктурирани p2p

При тях топологията на свързване е произволна, както и производителността на комуникационните канали. Често се случва възли да се включват и отпадат. В резултат, общата производителност е също произволна.

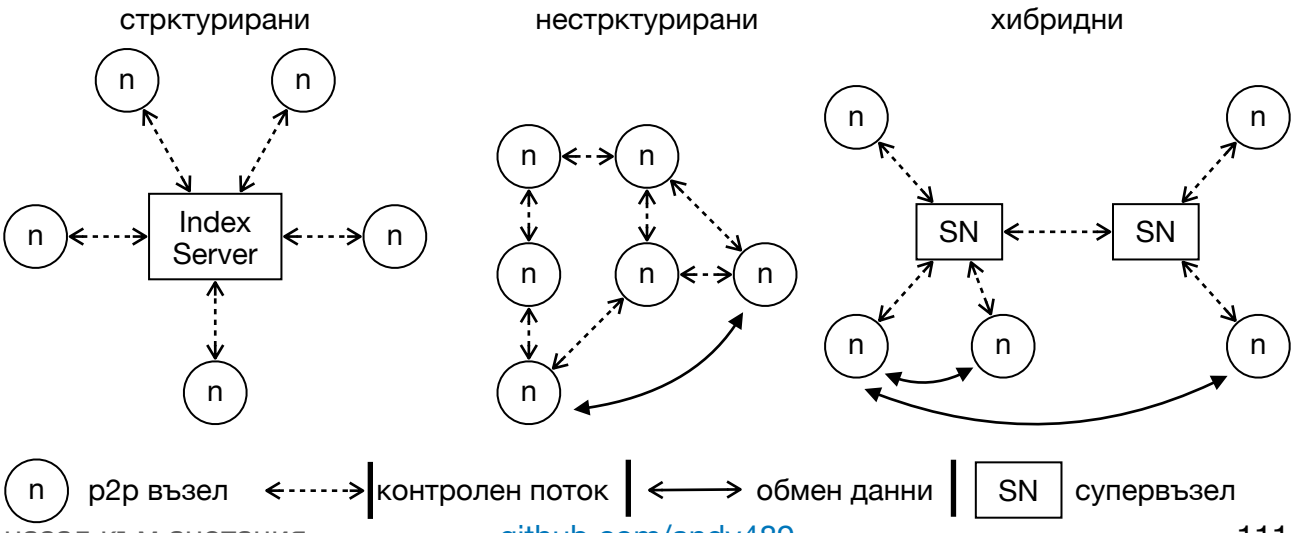
3.2.2. Структурирани p2p

Топологията е изрично зададена, което дава гаранции и за бързодействието. Съществуват няколко топологии, като много популярна е топологията хорда. При нея всеки възел се свързва с фиксиран брой съседи.

3.2.3. Хибридни/йерархични

Ниските нива са структурирани, като всяко ниско ниво играе като супервъзел в неструктурирана мрежа.

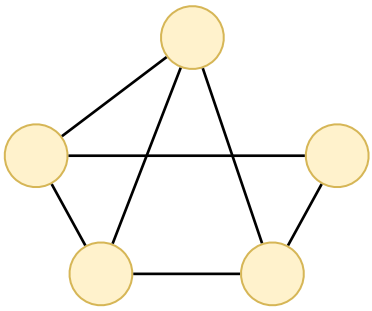
3.2.4. Диаграми



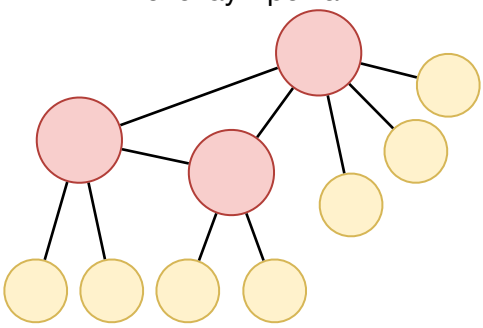
Неструктурирани p2p

Структурирани p2p

Чиста неструктурирана
overlay мрежа



Хибридна
overlay мрежа



Distributed Hash Tables (DHT)
базирана overlay мрежа