СОФИЙСКИ УНИВЕРСИТЕТ “СВ. КЛИМЕНТ ОХРИДСКИ”
ФАКУЛТЕТ ПО МАТЕМАТИКА И ИНФОРМАТИКА
Катедра “Информационни Технологии”

ДИПЛОМНА РАБОТА

за получаване на образователно-квалификационна степен “магистър”

на Шенол Хълми Юсуф, фак. № М-21493
студент от магистърска програма
“Разпределени системи и мобилни технологии”

тема:

“ Система за отдалечено изпълнение на приложения и услуги в йерархичен грид”

Научен ръководител:
доц. д-р Васил Георгиев

София, 2007

Съдържание

Увод

Настоящата дипломна работа представлява част от проекта GrOSD за разработка на лек грид мидълуеър. Изследванията и разработката на грид платформата GrOSD (Grid-aware Open Service Directory) се извършват по проекта СУГрид на Факултета по Математика и Информатика към СУ ”Свети Климент Охридски”, както и в партньорство с европейският изследователски проект CoreGRID (European Network of Excellence CoreGRID). Основната цел е изграждането на лека грид инфраструктура, като приоритет са опростената архитектура и реализация.

Основната задача, заложена в дипломния проект, е разработката на системен модул за отдалеченото изпълнение на приложения и услуги, които в контекста на проекта GrOSD се установяват и работят във възлите на грид-системата. С помощта на този модул се оползотворява предоставения за нуждите на грида изчислителен ресурс за реализирането на разпределени изчисления. За този модул ще използваме наименованието Node Service (или съкратено NS) в съответствие с приетата терминология на платформата GrOSD. Същевременно обаче трябва да отбележим, че NS не е разработен като системен модул, който дефинира своето съществуване единствено в рамките на споменатата грид платформа. Макар че акцентът при неговата разработка е все пак с оглед на интеграцията му в рамките на грид проекта, също така модулът има и своя самостоятелна реализация за неговото автономно използване от отдалечени клиенти през специално създаден за тази цел *приложен програмен интерефейс* (за по-кратко, този термин ще се съкращава оттук нататък в изложението до приетия в програмните среди негов чуждоезичен еквивалент *API – Application Programming Interface*). Тази черта определя необходимостта от една гъвкава архитектура на модула, която да позволява неговото използване в рамките на различни системи, които могат да го надстройват и да използват неговата функционалност.

В настоящия увод ще обобщим функционалните черти на модула, които той осигурява във възлите на грид-системата. Основните типове задания, за които настоящият системен модул е предназначен, са следните:

- Изпълнението на потребителски приложения, написани на програмния език Java, които са компилирани в готов вид като самостоятелни .class файлове или като пакетирани .jar архиви.
- Изпълнението на произволни потребителски програми, написани на Java, със специфиране на входната точка на изпълнение под формата на публично деклариран метод в програмния код. Както в предната точка, програмата се предоставя под формата на компилирани .class файлове или .jar архиви. Дефинирането на класа, метода и аргументите за изпълнение става динамично, в процеса на създаване на заявката.
- Зареждането на програмен код, компилиран на Java, във възела от грид-системата, където се намира модула, с цел да бъде в постоянна готовност за приемането и обработката на набор от входни данни. Тази функция ще наричаме поддръжка и изпълнение на *персистентни*

услуги, за разлика от горните типове задания, които са за еднократно изпълнение на възлите и имат *преходен* характер.

- Поддръжката на изпълнение на *персистенти услуги*, от своя страна, поставя няколко подзадачи пред програмното изпълнение, най-важните от които са осигуряване на пълен *жизнен жикъл на персистентната услуга* (зареждане, стартиране, спиране и отстраняване от възела) и дефинирането на механизми за захранването му с входни данни, като същевременно се осигури максимална гъвкавост за потребителите при формулирането на заявките за изпълнение, подобно на работата с *преходните услуги*.
- Осигуряването на механизми за навременно уведомяването на потребителите за хода на изпълнение на заданията и за събирането и изпращането на получените резултата по предварително дефинирани в заявката критерии.

Изброените дотук функции са мотивирани и отговарят на нуждите и изискванията за осъществяване на разпределени изчисления в контекста на платформата GrOSD. В допълнение, пред програмната реализация бяха поставени и други цели, които макар и да не са задължителни като характеристика на модула в качеството му на част от грид системата, допринасят за по-пълноценното оползотворяване на заложените в него основни функции. Ще обърнем внимание на следните допълнения:

- *Многозадачност* на Node Service – позволява се независимото стартиране на няколко задания едновременно.
- Детайлите по комуникационния протокол за трансфериране на формулираните заявки от потребителската машина до изчислителния възел са скрити зад фасадата на *приложен интерфейс* за зареждането на заявките.
- Предоставя се сравнително мощен интерфейс за формулирането на самите заявки, който от една страна, трябва да е достатъчно *гъвкав* по отношение на дефинирането на входните и изходни параметри на заданието за изпълнение, но също така да позволи потребителските приложения, които го използват, да бъдат *компактни* като обем на програмния код.
- С оглед на търсенето на по-широко поле за неговото функционално приложение, модулът трябва да бъде гъвкав и по отношение на неговото *разширяване* с допълнителни бъдещи типове задания за изпълнение, които могат да се изискат в контекста на други среди.

От технологична гледна точка, модулът трябва да бъде разработен с некомерсиални програмни средства, които да осигурят изпълнението му на широк кръг от платформи, използващи свободен софтуер с отворен код.

В изложението по-долу ще разгледаме в по-големи детайли смисъла и реализацията на всяка една от споменатите функции на системния модул.

Първата глава от изложението акцентира върху основната приложна област на Node Service, а именно грид архитектурата и конкретни имплементации на грид

системи. Тази глава разглежда по-подробно леките грид-системи, като се спира на основни им характеристики и технологиите за тяхното изграждането.

Втора глава продължава с обзор на архитектурата на платформата GrOSD. Мотивира се мястото на модула Node Service в тази архитектура. Дефинират се общите изисквания към модул от този тип и се дават примери как те се изпълняват в други грид системи.

Трета глава започва с описание на концептуалния модел на Node Service в настоящата му имплементация, мотивира се нейната архитектура, след което се преминава към обстоен анализ на структурата, функциите и реализацията на всеки един от основните му елементи.

В четвърта глава се демонстрира как представения модел работи в два основни сценария на приложение – работа на Node Service в самостоятелен режим и интеграцията му със системата GrOSD.

В пета глава се дават кратки указания относно инсталационните пакети на услугата, работата с тях от страна на изпълнителния възел и от страна на клиента.

Шеста глава представлява обзор на Java технологиите, използвани за реализацията на настоящия проект.

Седма глава е специално посветена на разгръщане на възможностите на проекта и обогатяване на функциите на услугата. Посочват се както перспективите за развитие на Node Service в рамките на грид-система, така и един любопитен ракурс на модификация на проекта за самостоятелна работа в peer-to-peer мрежи, отново с цел извършване на разпределени изчисления.

Приложение А представя примерен клиентски код на Java, който демонстрира работата с Node Service API-то за създаването на заявки за изпълнение и получаване на резултата.

1. Обзор

1.1. Грид-системи

Грид изчисленията са едно от най-новите направления в развитието на разпределените системи. Тази дисциплина изучава най-общо свързването на потенциално неограничен брой всевъзможни изчислителни устройства посредством мрежови връзки и услуги в една “мрежа” или “решетка” (grid). Често се използва аналогия между този нов подход към пресмятанията и електрическата мрежа. Днес приемаме електричеството за даденост, но това, което е направило електроенергията широкодостъпна е изграждането на електрическата мрежа. Чрез предоставянето на електричеството еднакво както на обикновените потребители, така и на предприятията, електрическата мрежа е създала предпоставки за създаването на нови устройства и развитието на индустриите, които ги произвеждат. При нужда, потребителите могат да свързват повече устройства към мрежата, както и доставчиците, при недостиг, могат да включат допълнителни генератори. По същия начин грид-системите могат да добавят неограничен брой изчислителни устройства във всяка грид-среда, като по този начин увеличават изчислителните възможности на грид-средата.

За грид започва широко да се говори в края на 90-те години на миналия век. Преди това опитите за координирано използване на широко разпределени ресурси са известни като метакомпютинг. През този етап се свързват суперкомпютри в изследователски центрове с цел решаване на конкретни научни проблеми. От този период датират проекти като FAFNER, който изчислява възможността за откриване на защитния код на RSA криптиращия алгоритъм, и I-WAY, при който се свързват с високоскоростни мрежи паралелни компютри в няколко научни центъра в САЩ. Тези ранни проекти могат да се считат за предшественици на грид.

Понятие за грид

Според една от първите дефиниции, дадена от Фостър и Кеселман в [1], изчислителен грид е апаратно-програмна инфраструктура, предлагаща надежден, консистентен, широкодостъпен и евтин достъп до система от високопроизводителни изчислителни ресурси. Надеждността е много важна за потребителите на грид – те трябва да са сигурни, че ще получат предсказуема, постоянна, а често и висока производителност от грид-системата. Системата трябва да предлага консистентни услуги, в смисъл, че услугите трябва да са стандартни, достъпни през стандартни интерфейси, използващи стандартни параметри. Само така може да се постигне широко възприемане на новата технология. Широката разпространеност гарантира на потребителите, че грид услугите са достъпни винаги, в каквато и да е среда – както е достъпна електроенергията. Тази инфраструктура трябва да предлага евтин достъп, защото в противен случай не би могла да намери широко възприемане.

Тази дефиниция набляга основно на изчислителните възможности на грид. По-късно авторите я променят, като наблягат на идеята, че основната задача на грид е споделянето на ресурси, и по-конкретно – “координираното споделяне на ресурси и решаване на задачи в динамични, мултиинституционални виртуални организации” [2]. Споделянето се отнася до директен достъп до компютри, програмно осигуряване, данни и други ресурси. То е стриктно контролирано, като доставчиците и потребителите на ресурси ясно дефинират какво точно се споделя, кой има право да споделя и при какви условия се извършва споделяне. Група индивиди и/или институции, обединени от подобни правила за споделяне, съставляват виртуална организация (ВО). Освен координираното споделяне на ресурси и формирането на виртуални организации, от съществено значение за грид-системата е наличието на отворени стандарти. Последните осигуряват средства за взаимодействие и интеграция, и трябва да се използват при откриването, достъпа и координацията на ресурсите.

Архитектура на грид-системите

От съществено значение за грид архитектурата е тя да се базира на общи отворени протоколи, дефиниращи основните механизми, чрез които членовете на една ВО и ресурсите договарят, установяват и извършват споделянето. Отворената архитектура, базирана на стандарти, улеснява разширяемостта, взаимодействието, преносимостта и споделянето на код, докато общите протоколи позволяват дефинирането на стандартни услуги.

Фостър и Кеселман предлагат в [2] слоест модел на грид архитектура, който се състои от няколко слоя и следва принципа на “пясъчния часовник” - в тясната

част на “часовника“ се дефинират малък брой основни абстракции и протоколи, които предлагат функционалност на множество услуги от по-високите слоеве, и от своя страна се базират на множество технологии, които се намират в по-ниските слоеве.

В предложената архитектура най-ниското ниво предоставя ресурсите, до които грид протоколите осигуряват достъп. Това могат да са изчислителни ресурси, ресурси за съхранение на данни, мрежови ресурси, сензори. Това ниво се нарича “тъкан” (Fabric) и представлява апаратната инфраструктура на грид-системата. Слоят над него има два подслоя и съставя тясната част на “часовника”. Единият подслой се нарича съобщителен (connectivity) и дефинира основни протоколи за комуникация и идентификация (authentication). Комуникационните протоколи позволяват пренос на данни между ресурсите, изграждащи грид инфраструктурата. Идентификационните протоколи надграждат комуникационните услуги и осигуряват криптографски сигурни механизми за установяване идентичността на потребители и ресурси. Ресурсният подслой се базира на протоколите на съобщителния подслой и дефинира протоколи за сигурно договаряне, наблюдение, контрол, счетоводство и заплащане за операциите върху ресурсите. Тези протоколи са насочени напълно към индивидуални ресурси. Следващият слой предлага протоколи, услуги и приложни програмни интерфейси (API), които не са свързани с конкретен ресурс, а са глобални и реализират взаимодействието между ресурсите в инфраструктурата. Затова този слой се нарича колективен (collective). Той увеличава разнообразието от услуги като комбинира малкия брой протоколи от долния слой. Последният слой от тази архитектура е приложният – тук се изпълняват грид приложенията. Те използват услугите, предоставени от всеки от долните слоеве.

Така описаната архитектура намира пълна реализация в Globus Toolkit [3]. Той предоставя програмна инфраструктура за свързване на хетерогенни възли във виртуален компютър. Globus Toolkit поддържа групи от услуги, като комуникация, защита, регистрация и управление на ресурсите, като услугите са със строго дефинирани интерфейси. Пакетът от инструменти поддържа съвместно ползване на ресурси от множество организации.

С нарастване на интереса към грид системите се увеличава необходимостта от стандарти, които да се използват в грид архитектурата. Един от най-важните стандарти в това отношение е Архитектурата за отворени грид услуги (Open Grid Services Architecture – OGSA) [4], [5]. Той е издаден от Global Grid Forum (GGF) и представлява спецификация, която дефинира обща, стандартна и отворена архитектура за грид приложения. Тази спецификация цели да стандартизира почти всички услуги, които грид приложенията могат да използват. OGSA специфицира архитектура, ориентирана към услугите (Service-Oriented Architecture – SOA) за грид, която създава модел на изчислителна система, състоящ се от множество разпределени изчислителни шаблони. Като технология за реализация на последните се предвиждат уеб услугите. OGSA стандартът дефинира интерфейси на услугите и определя протоколите за използване на тези услуги, но не предоставя конкретна реализация.

Класове грид приложения

Грид-системите могат да се използват за решаване на разнообразни задачи. Обособяват се пет основни класа от грид приложения [2], въз основа на изискванията и предназначението на приложенията.

- *Разпределен суперкомпютър* – този тип приложения използват услугите на грид-системата, за да получат достъп до голям брой изчислителни ресурси, с цел решаване на задача, която изисква интензивни изчисления и не може да се реши от единствена система. Примери за такива приложения са военни симулации, субатомни, атомни и молекулярни задачи, задачи от небесната механика и др.
- *Високоефективна фонова обработка (High throughput computing)* – в този случай грид-системата насрочва и изпълнява голям брой слабо свързани или независими задания, с цел да се оползотворят неизползвани ресурси. Подобни приложения се използват за дизайн на чипове, решаване на криптографски задачи и др.
- *Обработка по заявка (On-demand computing)* – тези приложения използват възможностите на грид, за да посрещнат кратковременна нужда от ресурси, които не би било финансово ефективно да се подържат локално. Такива ресурси могат да са изчислителни ресурси, програми, хранилища за данни, специализирани сензори и др. Пример за подобни грид приложения е обработката на данни, генерирани от сложна медицинска апаратура.
- *Обработка на големи обеми от данни (data intensive computing)* – при тези приложения фокусът се поставя върху синтезиране на информация от данни, намиращи се в географски разпределени хранилища, цифрови библиотеки и бази от данни. Често тази обработка е съпроводена с интензивни изчисления и комуникации. Такива приложения са нужни при обработка на огромното количество данни, натрупано при експерименти от физика на високите енергии, от телескопи, или метеорологични спътници.
- *Съвместна обработка (Collaborative computing)* – този клас приложения са основно ориентирани към улесняване на взаимодействието между хората и изграждане на сложни колективи за решаване на определени задачи. Могат да се използват за симулации в реално време, интерактивни игри, и др.

Като цяло, грид представлява сравнително ново и многообещаващо направление в развитието на разпределените системи. Развитието на грид и широкото му разпространение могат да променят начина, по който понастоящем се използват изчислителните ресурси. Към момента грид-системите се използват предимно в научните и инженерни среди. Основна задача пред грид в бъдеще е повсеместното му разпространение сред всички групи потребители.

1.2. Лек грид – характеристика и примери

Много от проектите в областта на грид са ориентирани към определена приложна област, или дори към изпълнение на специфични приложения – пример за това е проектът EU DataGrid, който е насочен към създаване на грид инфраструктура за анализ на големи обеми от данни, натрупани при научни експерименти и формиране на научни екипи за сътрудничество. От друга страна, повечето платформи, които предлагат грид мидълуеър, като например Globus Toolkit, са твърде сложни – те имат много богата функционалност и голям брой инструменти, което ги прави трудни за инсталация и администрация. Като следствие от това възниква необходимостта от грид платформи, които са опростени и общи – не са ориентирани към конкретна задача и предлагат базова функционалност, която може да се разширява при необходимост. Тези грид платформи са известни като “леки” (lightweight) грид-системи.

По своята архитектура, реализация, предлагани протоколи и услуги леките грид-системи се различават значително помежду си. Основно ги обединява това, че те са проектирани за бързо внедряване и минимални разходи за експлоатация и поддръжка. Те са подходящи за използване от (и дори предимно са насочени към) индивидуални потребители или по-малки организации. Този клас потребители не се нуждае от всички услуги и протоколи, които сложните грид платформи предлагат, и за тях лекият грид предлага базова функционалност, която е достатъчна за изпълнение на техните задачи, като при това могат да се използват предимствата на грид изчисленията като споделяне и прозрачно използване на ресурси. Обикновено се предвижда такива грид-системи да се ползват за решаване на задачи с по-скромни мащаби, но това не е задължително.

За разлика от сложните грид системи, леките грид платформи са лесни за инсталиране и администриране – повечето дори не изискват специални администраторски права или задълбочени познания. Простотата на използване ги прави подходящи за разнообразни потребители и организации. Освен това те обикновено не са насочени към решаване на конкретна задача, и затова са подходящи както за научни цели, така и за приложение в корпоративна среда, или дори сред масови потребители.

Характерно за този тип грид-системи е динамичната промяна на инфраструктурата, лежаща под мидълуеъра – динамичното включване и отпадане на ресурси и потребители. Това позволява подобни системи да се използват в динамична среда, като например сред индивидуални потребители, които се включват и изключват спорадично в системата и използват и предоставят временно ресурси. Подобна динамика е присъща и на самия мидълуеър – той предлага само най-базовата функционалност, а ако са необходими допълнителни услуги, те се добавят динамично. Обикновено в леките грид-системи управлението на потребителите и на ресурсите е разпределено, няма напълно централизиран контрол, което допринася за увеличаване на динамиката на системата.

В [6] са разгледани основните изисквания, на които трябва да отговаря един лек грид с общо предназначение и компонентна архитектура. Следват някои от тези изисквания:

- Лек и с архитектура, която не е ориентирана към конкретна задача – повечето съществуващи грид-системи разчитат на множество ресурси и инструменти, което им пречи да бъдат достатъчно общи. Общата грид платформа трябва да е “лека”, с минимална, но съществена функционалност, като може да бъде разширявана с допълнителни характеристики. Това би позволило грид технологиите да се използват от всякакви потребителски устройства.
- Статични и динамични метаданни – статичните метаданни са особено важни в компонентна среда, тъй като те дават информация за компонентите, която е нужна за правилното им използване и комбиниране, като например версия, производителност, проблеми със съвместимостта и др. Динамичните метаданни предоставят информация за динамичните свойства на компонентите и са важни например за удовлетворяване на качеството на услугите, оптимизация, възстановяване след грешки и др.
- Динамично внедряване на компоненти – системата трябва да може динамично да включва компоненти, например като реакция на промени или нараснали нужди.
- Преконфигуриране и адаптиране – системата трябва да е в състояние да се адаптира към променящата се среда, като по този начин се гарантира отказоустойчивост.
- Поддръжка на клиент/сървър и P2P споделяне на ресурси – при клиент/сървър модела на споделяне съответствието между заявки и ресурси се прави централизирано от брокер, докато при P2P доставчиците и ползвателите на услуги комуникират без посредник. При първия подход могат да се наложат конкретни политики на употреба на ресурсите, докато при втория се намалява времето за отговор и се увеличава производителността. Затова и двата модела трябва да се поддържат.
- Предоставяне на услуги при нужда – платформата трябва да поддържа създаване на услуги при поискване (on demand), когато са нужни на потребителите.
- Минимален но достатъчен модел на сигурност – високото ниво на сигурност, високата производителност и простотата на платформата са взаимно противоречащи си цели. Затова между тях трябва да се намери правилното съотношение. Подходящо е да се реализира минимално ниво на сигурност, но да се предвиди поддръжка на допълнителни механизми за сигурност, които да се използват при нужда.
- Разпределено управление – разпределеното и същевременно координирано управление е в основата на функционирането на платформата.

1.3. Примери за леки грид платформи.

ALiCE (Adaptive and scaLable Internet-based Computing Engine) [7], [8] представлява лек грид мидълуеър за създаване и внедряване на общи грид приложения. ALiCE е завършена грид-система. Тя улеснява натрупването и

виртуализацията на ресурси в интранет и използването на свободни ресурси в Интернет. Платформата се състои от няколко ясно обособени слоя. В най-ниския слой се намират базовите услуги, предлагани от системата. Те включват откриване, заделяне и управление на ресурси, управление на данни, управление на сигурността, комуникация между обектите, счетоводство и мониторинг. Следващият слой поддържа разработката и внедряването на грид приложения. Той скрива детайлите на паралелното програмиране като предлага библиотека с шаблони за изграждане на грид приложения, които улесняват разработката. Най-горният слой съдържа инструменти и приложения, които се използват от потребителите на системата.

Заданията в грид-системата се подават от компютър, наричан “консуматор”, и се изпълняват на свободни компютри, които се наричат “производители”. Заданията се предават от консуматорите на производителите посредством ресурсен брокер, който управлява процесите и ресурсите. Брокерът съдържа и планировчик (scheduler), който се грижи за правилното разпределяне на заданията по производители. На всеки консуматор има потребителски интерфейс, които улеснява подаването на заданията, докато на производителите работи модул, управляващ изпълнението на заданията.

Друг проект реализиращ лека грид платформа е ***H2O*** [9], [10]. Той е ориентиран към по-малки организации и индивидуални потребители. В H2O се набляга на силната разпределеност на ресурсите, преконфигурирането и адаптивността. Платформата е компонентна и ориентирана към услуги. Базира се на идеята всеки ресурс да се представи като софтуерен компонент, който предоставя услуги чрез ясно дефинирани отдалечени интерфейси. Собствениците на ресурсите предоставят среда за изпълнение под формата на контейнер за компоненти, който се изпълнява върху ресурса. Тези контейнери се наричат ядра (kernels). В тези контейнери могат да се стартират услуги, също под формата на компоненти. Тези услуги се наричат плъглети (pluglets). Докато при повечето компонентни платформи собствениците на контейнерите са също и тези, които предоставят услугите, които се изпълняват в контейнерите, при модела на H2O това не е задължително. Възможно е трета страна, която няма нищо общо със собственика на контейнера, ако има необходимите права, да стартира услуга в контейнера. Този модел на споделяне прехвърля тежестта на внедряване на услугите в контейнерите от собствениците на ресурси към доставчиците на услуги, което може да насърчи повече потребители да споделят ресурси. Платформата осигурява разнообразие от протоколи за комуникация между компонентите, както и различни механизми за сигурност.

OurGrid [11], [12] е лека грид-система, разработвана от Федералния университет на Кампина Гранде, Бразилия, със съдействието на Hewlett Pacard. Тази платформа, за разлика от разгледаните дотук, не е съвсем обща – тя е предназначена за решаване на асинхронни паралелни задачи (т.е., такива паралелни задачи, които могат да се разделят на независими подзадания). Въпреки, че този тип задачи са по-прости за програмиране, те намират голямо приложение, например при Монте Карло симулации, фрактални изчисления, изчисления в биологията, компютърната обработка на изображения и др. Основните компоненти на системата са три:

- *MyGrid* – представлява централна точка на грида. При изпълнение на задания той координира обработката, като извършва планиране на

заданията, трансфер на данни, наблюдение на изпълнението. Чрез него се подават задания в грида. Този модул се изпълнява на “home” машина. Всички останали машини в грида се наричат грид машини.

- *Peer* – изпълнява се на “peer” машина. Неговата задача е да организира и предоставя грид машините в същия административен домейн. Той се използва от MyGrid модула за осигуряване на грид машини.
- Потребителски агент (*User Agent*) – работи на всяка грид машина, като осигурява достъп до нея за изпълнение на подадените в грида задачи.

Потребителите на тази грид платформа се обединяват в P2P общност, като всеки в общността използва свободните ресурси на останалите. При този сценарий е възможно потребители само да използват ресурси, без да споделят. За да се избегне това, OurGrid реализира механизъм за заделяне на ресурси, който гарантира предимство на потребителите, които споделят своите ресурси, като по този начин стимулира споделянето.

В заключение може да се каже, че леките гридове са тип грид-системи, които биха могли да притежават различни характеристики, но основните им качества са опростената, олекотена архитектура, общото им предназначение и наличието само на най-основни услуги. Тези техни черти биха могли да допринесат за разпространението им сред по-широки групи потребители.

1.4. Лек грид - технологии

За реализацията на леките грид платформи се прилагат разнообразни технологии. Повечето като език за програмиране използват Java, от където следват и технологиите, поддържани от платформата Java. Използването на този език гарантира платформена независимост на лекия грид, което е от съществена важност, тъй като се предполага, че грид мидълуеърът ще работи на машини с различни архитектури и операционни системи. Не на последно място стои и фактът, че платформата е безплатна. Леките грид-системи използват и разнообразни протоколи, среди и др.

ALiCE е реализирана на Java и прилага Java технологии като Jini, JavaSpaces, GigaSpaces и JNI.

Jini [13] представлява мрежова технология, която улеснява динамичното свързване и взаимодействие на услуги и устройства в мрежа. Обединението на услуги и устройства се нарича Jini федерация. Услугите могат свободно да се присъединяват към федерацията и да я напускат. Във всяка федерация има поне една директорийна услуга, в която се регистрират останалите услуги. Когато една услуга се присъединява към федерация, тя се регистрира в директорийната услуга като изпраща информация за себе си и свой посредник (proxy) – обект, чрез който е достъпна съответната услуга. Клиентите могат да търсят услуги по техните описания. Ако клиент намери услуга, която иска да използва, директорийната услуга му изпраща съответния обект посредник, чрез който клиентът може директно да комуникира с услугата.

Jini използва ***JavaRMI*** за предаване на обекти между Java виртуални машини. Това позволява на Jini устройствата да пренасят изпълним код из мрежата, към която са свързани, като по този начин се създават мрежи от устройства, които не изискват предварително планиране, инсталиране и човешка намеса.

JavaSpaces [14], която е част от Jini, представлява проста, но мощна технология за изграждане на разпределени приложения. В приложение, използващо тази технология, всички процеси са свързани посредством мрежа, като комуникират и синхронизират действията си посредством хранилище за обекти, наречено пространство (space). Всички комуникации в ALiCE се извършват посредством JavaSpaces.

Като алтернатива на JavaSpaces в ALiCE може да се използва технологията ***GigaSpaces*** [15]. Тя представлява платформа за синхронизация и комуникация, осигуряваща програмна инфраструктура за информационно взаимодействие на корпоративни разпределени приложения и уеб услуги. Основната разлика между JavaSpaces и GigaSpaces е, че първата технология осигурява логическа разпределена обща памет, докато втората реализира разпределена обща памет като обединява няколко пространства на различни машини.

ALiCE използва JNI за да направи възможно извикване на не-Java код по време на изпълнение.

H2O също е реализиран на Java, като при това използва технологиите, предлагани от Java платформата. За комуникация се използва модифициран вариант на RMI – RMIX [16]. Той добавя към стандартния RMI възможност за асинхронни извиквания, както и еднократни извиквания, които могат да се използват за реализация на предаване на съобщения. Всъщност, RMIX представлява обща платформа и семантика над множество протоколи. Различни протоколи се поддържат чрез добавянето на съответните модули , което може да стане дори по време на изпълнение. Понастоящем са реализирани два протоколни модула – RMIX-JRMPX, който е базиран на JavaRMI и Java сериализация, и RMIX-XSOAP, базиран на XSOAP и използващ SOAP като протокол за пренос на данни. Въпреки, че RMIX е протоколът по подразбиране за H2O компонентите, то е напълно възможно компонентите да използват други средства за комуникация.

Други технологии, които се използват в H2O, са JNI, която се използва за да се стартират услуги, които не са писани на Java, в Java контейнери, както и уеб услугите – възможно е интерфейсът на приложна услуга, работеща в контейнер, да се опише посредством WSDL.

Разнообразието от технологии за разработката на леки грид-системи дава голяма свобода за тяхната реализация. От направения преглед се вижда, че основно се използват Java платформата и различни Java технологии, което се обуславя от платформената независимост на езика и множеството средства, които платформата предлага.

2. Инфраструктура и системна архитектура на GrOSD

2.1. Обзор на архитектурата на GrOSD.

GrOSD се отличава с йерархична многослойна архитектура [17], [18], [19]. Тази грид-система е изградена от свързани “виртуални клъстери” от компютри. Под виртуален клъстер следва да се разбира свързани чрез мрежа компютри, принадлежащи на обща административна област. На тези компютри работи грид мидълуеърът, който предлага хомогенни системни услуги и обединява

машините във “виртуалния клъстер”. Последният е единица за организиране на възли, ресурси и потребители, като в действителност представлява по-скоро логическа единица, отколкото физическа. В GrOSD не е нужно компютрите, които са част от един “виртуален клъстер” да се намират физически на едно и също място. По-нататък за краткост “виртуалния клъстер” ще се нарича само клъстер. Архитектурата е йерархична и има три нива. Първото от тях е локалното ниво. На това ниво се намират различните клъстери, които съставят системата. Второто ниво се изгражда чрез свързване на клъстерите и представя самия грид. Последното ниво е “междугридово” – на него е възможна връзка с други грид-системи. Всеки клъстер разполага със собствени системни услуги, като на грид нивото има системни услуги, общи за грида.

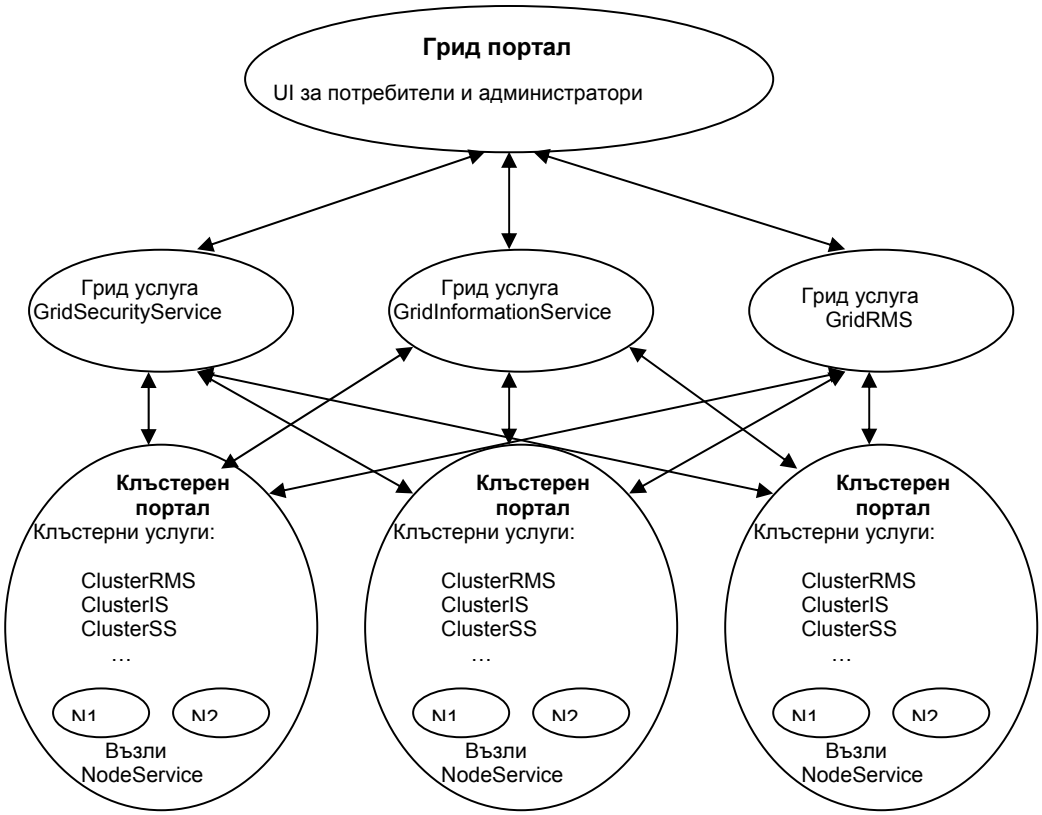
Една от целите на проекта е платформата да бъде изградена от прости и ефективни компоненти. Архитектурата на GrOSD е базиран на модела на архитектура, ориентирана към услугите (Service Oriented Architecture). Входната точка към системата е грид порталът. Грид порталът и клъстерните портали предлагат сходна функционалност за потребителите. Системните услуги, на които се базира платформата са следните:

- Услуга за сигурността и управление на потребителите (Security Service, SS).
- Услуга за управление на ресурсите (Resource Management Service, RMS).
- Информационна услуга (Information Service, IS).
- Услуга за наблюдение и контрол (Monitoring Service, MS).
- Комуникационна услуга (Communication Service, CS).
- Услуга за управление на възлите и изпълнение на приложения върху тях (Node Service, NS).

Всяка услуга има както клъстерна, така и глобална версия. Фигура 1 илюстрира общата архитектура на системата, както и услугите на клъстерно и глобално ниво. Всяка услуга изпълнява своите функции локално в клъстера, или глобално за грида, в зависимост от вида си. Клъстерните системни услуги взаимодействат помежду си в рамките на клъстера и могат да комуникират със съответните услуги от грид нивото. Глобалните грид системни услуги взаимодействат помежду си, както и с клъстерните услуги, реализирайки по този начин грид инфраструктурата.

Основните функции, които предоставя GrOSD платформата са следните:

- библиотека от неактивни услуги
- директория от активни услуги
- структурирано описание на услугите
- управление на потребителите
- статистика за цялостното състояние на системата
- потребителски и административен графичен интерфейс, подържан от порталите



Фигура 1. Обща архитектура на грид-системата GrOSD

Всеки ресурс в системата се представя чрез услуга и всяко задание за изпълнение в системата представлява услуга с входни данни и ресурси, на които да се изпълни. Това прави моделът на услугите фундаментален за архитектурата на системата.

Услугите в GrOSD могат условно да се разделят на следните типове:

- Постоянни системни услуги (Persistent System Services, PSS) – това са изброените по-горе системни услуги, които реализират грид инфраструктурата.
- Постоянни приложни услуги (Persistent Application Services, PAS) – това са услуги, които могат да се използват директно от потребителите (за разлика от системните услуги) и които са постоянно активни.
- Временни приложни услуги (Transient Application Services, TAS) – този тип услуги представят потребителско задание, което се подава на портала за еднократно изпълнение.
- Временни библиотечни услуги (Transient Repository Services, TRS) – това са услуги, които потребителите добавят в грид-системата, и които могат да се ползват от други потребители и услуги. Тези услуги не са активни постоянно, а се активират когато някой потребител иска да ги използва.

Всяка услуга (без системните) притежава структурирано описание в XML формат. То се задава от доставчика на услугата и се използва от RMS, когато услугата трябва да бъде стартирана. Описанието съдържа следните данни:

- предназначение на услугата
- типове данни, които услугата поддържа за входните си параметри
- типове данни, които услугата поддържа за изхода си
- апаратни и програмни изисквания на услугата (например операционна система, вид процесор, количество памет и др.)
- каква е цената (натоварването върху системата) за изпълнение на услугата и от какво зависи тя
- роли, необходими за изпълнение на услугата
- роли от които се нуждае самата услуга, за да се изпълни

GrOSD позволява на потребителите да създават нови услуги като свързват приложни услуги – изходният резултат от една услуга да се използва като вход на друга услуга. Комплементирането на входа и изхода се извършва само въз основа на типа на входните и изходните данни. Платформата предоставя подходящ интерфейс за изграждането и изпълнението на такива услуги.

Следва по-подробно описание на архитектурните компоненти на GrOSD.

Грид портал и клъстерни портали

Порталите представляват входна точка за грида или за конкретен клъстер съответно. През тях потребителят може да получи достъп до системата ако е идентифициран от услугата за сигурност. Грид порталът дава достъп до грид-системата като цяло, а клъстерните – до съответния клъстер. От портала потребителите могат да разглеждат наличните библиотечни и активни услуги и да ги стартират, да добавят собствени услуги и да подават задания за изпълнение.

Услуга за управление на ресурсите (RMS)

Управлението на ресурсите е от съществена важност за функционирането на грид системата. В архитектурата на GrOSD тази услуга [18] заема централно място, като си взаимодейства с всички останали системни услуги. Тя извършва планиране на заданията за изпълнение, откриване, резервиране, заделяне на ресурси, както и мигриране на задания.

На клъстерно ниво RMS (CRMS – Cluster RMS) получава заявки за стартиране на услуги от съответния клъстерен портал. От информационната услуга на съответния клъстер CRMS получава списък с наличните ресурси за изпълнение на заданието. При търсене на ресурсите може да се използва само филтриране по роли, но е възможно и задаване на по-сложни условия за търсените ресурси. RMS приема като критерий при търсенето на ресурси и цената за изпълнение на заданието.

Ако има свободни ресурси, услугата се стартира на тях. В случай, че резултатът съдържа повече ресурси от необходимите, тогава се избират най-подходящите на базата на тяхната употреба и статистиката за натовареността. Ако няма налични ресурси услугата може да се насрочи за по-късно изпълнение. В този случай, както и ако тя първоначално е била зададена за по-

късно изпълнение, се резервират нужните ресурси и тя се поставя в опашка на чакащите задания.

След като ресурсите за заданието са заделени, отбелязва се, че те вече не са свободни и заданието се предава на съответния възел, където се намират ресурсите, за изпълнение. Състоянието на изпълняващото се задание се наблюдава от услугата за контрол и наблюдение (MS).

На грид ниво RMS (GRMS) получава задания за изпълнение от грид портала. При заделяне на нужните ресурси GRMS взаимодейства с клъстерните RMS, като накрая изпраща заданието за изпълнение на съответния CRMS. Възможно е паралелно изпълнение на множество подзадания на ресурси в различни клъстери и тогава GRMS си взаимодейства с няколко CRMS.

Информационна услуга (IS)

Информационната услуга се използва за съхранение на всички данни, необходими за работата на системните услуги. На клъстерно ниво тя управлява данните за услуги и потребители, описанията на услугите, статистически данни за използването на системата, информация за възлите и ресурсите, както и текущото им състояние, но в рамките на клъстера. На грид ниво грид информационната услуга управлява данни, общи за целия грид. В известен смисъл тази услуга може да се разглежда като интерфейс към база от данни. Обособяването на тази функционалност в отделна услуга улеснява работата на системните услуги с данните, като ги изолира от конкретните детайли на достъп до информацията. По този начин се улеснява и използването на различни източници на данни, тъй като услугата скрива конкретния метод на съхранение.

Услуга за наблюдение и контрол (MS)

Тази услуга редовно проверява състоянието на възлите и работещите на тях услуги. Ако има промяна в него това се отразява в съответната информационна услуга. По този начин системата разполага с актуална информация за състоянието на възлите, което улеснява откриването на прекъснали или зациклили задания, повредени възли или ресурси и други проблеми. Другите системни услуги я използват за да събират счетоводна информация или информация за състоянието на системата, която им е нужна за тяхната работа. Работещите услуги и услугите, управляващи възли, могат да информират услугата за наблюдение при промяна в състоянието си.

Комуникационна услуга (CS)

Комуникацията между задания и подзадания е с голямо значение в грид-системите, особено при реализация на паралелни изчисления, където е необходимо взаимодействие между паралелно изпълняващите се процеси. Тъй като архитектурата на GrOSD се базира на услуги, комуникацията между задания се осъществява от комуникационна услуга [20], която, както и останалите системни услуги, има клъстерен (CCS) и грид (GCS) варианти. Всяко задание в системата при стартирането си получава уникален идентификатор, който се състои от идентификатор на заданието (TaskId), уникален в клъстера, и идентификатор на клъстера (ClusterId), уникален в системата. Когато RMS разпредели някое задание за изпълнение на даден възел, тя създава съответствието между възела и идентификатора на

заданието и това съответствие се пази от CCS. Останалите задания знаят само идентификатора и ако искат да комуникират с това задание, те пращат съобщението до CCS, която вече го препраща до съответния възел. Ако е необходима комуникация между задания в различни клъстери, то CCS препраща съобщението до GCS, която знае в кой клъстер да го изпрати. В случай на миграция на задание от един възел на друг, или между клъстери, RMS, която извършва миграцията, актуализира съответствието между идентификатора на заданието и правилния възел в съответната комуникационна услуга (CCS или GCS, в зависимост от това дали миграцията се извършва в рамките на един клъстер или между клъстери).

Услуга за управление на възлите (NS)

Тази услуга се изпълнява на всеки възел в грид-системата. Тя прави един възел част от даден клъстер и съответно от GrOSD. NS следи ресурсите на възела и обновява статистически данни и информация за състоянието му в услугата за наблюдение. NS се грижи за изпълнението на изпратените от RMS задания. Тази услуга предлага възможност на останалите системни услуги да следят изпълнението на задания и да ги прекратяват при необходимост, и да откриват задания, които е нужно да бъдат рестартирани или преместени на друг възел. NS няма варианти за клъстерно и грид ниво – тя работи единствено на възлите.

2.2. Общи изисквания към модула за управление на възлите.

Без съмнение, системната услуга за управление на възлите и изпълнение на заданията е модулът, който носи най-голямата отговорност за осигуряването на изчислителните функции в грида. В една или друга степен, възможностите и характеристиките на този модул определят набора от услуги, които грид-системата може да предложи на своите потребители. От друга страна, той е длъжен да оползотвори по пълноценен начин предоставения изчислителен ресурс, какъвто е възелът, на който работи. В резултат на тези съображения, възникнат различни изисквания пред проектанта на тази услуга, които третират отделни аспекти от функционалността на модула:

- *Разширяемост на услугата за изпълнението на различни типове задачи* – тази характеристика директно определя и какви задачи за изпълнение може да се допускат в грида за изпълнението и е ясно че тяхното по-голямо разнообразие прави грида използваем за по-широк кръг от потребители; тази разширяемост трябва да се осигурява по един лесен и достъпен начин, с преконфигуриране на услугата, без да се засягат модулите от по-горните слоеве в грид архитектурата.
- *Адаптируемост на модула към други грид-системи* – тази характеристика по един или друг начин произтича от горната, но за разлика от нея, където се акцентира върху възможностите на грида да се адаптира към промените в изискванията в рамките на една грид-система, тук се набляга повече върху преносимостта на модула от една система в друга; отново тя трябва да се осигурява с лесно преконфигуриране, което да не засяга модулите от по-високо ниво.

- *Отказоустойчивост при грешки* – модулът може да обслужва заявки от различно естество, които си взаимодействат по-различен начин с предоставения изчислителен ресурс на възела. С други думи, не могат да се правят конкретни допускания какво може да е поведението на изпълняваното задание, но във всяко отношение модулът трябва да е способен да толерира възникнали грешки и пропадането на дадено задание не трябва да засяга способността му да изпълнява други задания, които постъпват от горните слоеве.
- *Да оползотворява ресурсите на системата за нуждите на своята работа.* Друг аспект на това изискване е модулът да не поставя някакви съществени изисквания към възела, на който той ще работи. Минималните изисквания увеличават потенциалният кръг от донори на ресурси за грида и правят работата на услугата по-ефективна.
- *Ясно дефиниран и унифициран канал за комуникация с останалите модули в грид-системата, които разчитат на неговите услуги.* Комуникационното звено трябва да даде достъп до пълния набор от функции на услугата, но същевременно трябва да е отделена от имплементацията на тези функции, за да позволи обратна съвместимост на клиентските модули с услугата след нейно разширение. С други думи, вътрешните промени в имплементациите на функциите трябва да бъдат прозрачни за външните модули.

Към тези основни изисквания могат да се дефинират и редица други, които са по-тясно свързани с конкретния дизайн и имплементация на сервиза. По-нататък в изложението ще обърнем внимание как избраните архитектурни решения при проектирането на услугата задоволяват формулираните критерии и каква допълнителна функционалност осигуряват те. На този етап ще отбележим, че модулът за управление на възлите в настоящата грид архитектура трябва да изпълнява две основни групи от функции:

- **Информационни услуги** – да осигурява информация за хардуерните и софтуерните параметри на възела, информация за хода на изпълняваните задачи и за текущата натовареност на възела, което спомага за модулите от горните слоеве да са винаги наясно с разпределението на приетите задания по възлите от грид-системата.
- **Изпълнение на конкретните потребителски задания** – тук влизат обработка на входните параметри на заданието, изпълнение на самото задание, събиране и връщане на резултата; във връзка с информационните функции на услугата, също така се следят етапите на изпълнение и се уведомяват заинтересованите модули за настъпили важни промени в състоянието на заданието.

Настоящата разработка акцентира предимно върху изпълнителната група от функции, задълбочавайки и доразвивайки поставените задачи в проблемната област, докато от информационната група е реализиран вътрешният мониторинг над заданията – на този етап, останалите функции се предоставят на допълнителни модули, които могат да се интегрират благодарение на разширяемата архитектура на сервиза.

2.3. Аналози на модула в други грид системи

Като аналог на ситемния модул за управление на възли, ще разгледаме две известни грид-системи, какви са особеностите им при техния модел на изпълнението на задания и какъв вид имат заданията за обработка в техния случай.

JGrid

При тази система има няколко типа задачи, които могат да се изпълнят [32]. Най-близки до текущата реализация в GrOSD са единичните и паралелни задания в обектен стил. Първите са обекти на Java, които наследяват интерфейса *java.lang.Runnable*, и могат да се изпълнят от изчислителния сервиз в собствена нишка. Основният механизъм за пренос и извикване на изпълнението е посредством технологията JavaRMI. Разликата при втория тип се състои в това, че този клас задания могат да се разбият на множество подзадачи от първия тип, които се пращат на отдалечените изчислителни услуги в грида.

Процесът на изпълнение и при двата вида задачи протича по един и същи начин. На потребителската машина се стартира клиентска програма на Java, която се свързва с грида и извлича *прокси* обект, който служи за връзка към *брокера* на задачи в грида. През *прокситo* клиента изпраща заданието с изискванията за ресурсите, необходими за изпълнение. *Брокерът* проверява дали има услуга на локално ниво в грида, която да отговаря на тези изисквания. Ако няма, се праща заявка за търсене към така наречения локатор на услуги, който прави проверка в по-голяма област от грида. След като се открият подходящи сервизи, единият от тях се избира за изпълнение и неговият локален мениджър на задания се уведомява за получаването на заданието. Той, от своя страна, стартира изпълнението в отделна нишка и ако се изисква, информира мониторинговия модул да наблюдава промените в състоянието на заданието. В същото време, ресурсният модул, който следи натоварването на процесора, паметта и диска на изпълнителния възел, нотифицира регистрираните *брокери* за текущото състояние на възела.

Специфичното за JGrid API-то е, че той разчита подаваните задания да наследяват определени общи интерфейси, които да правят възможен контрола върху техния жизнен цикъл на изпълнение. По-конкретно, налице са интерфейсите:

- **Task** – главен интерфейс на заданието; декларира метод за неговото изпълнение и метод за предаването на контекстен обект за хоста на изпълнение, като по този начин се осигури обратна връзка на заданието с машината, на която работи; този интерфейс наследява следващия:
- **Controllable** – декларира методи за контрол над изпълнението на заданието – неговата отмяна, временно спиране и възобновяване.
- **Checkpointable** - ако клиентското задание наследява и този интерфейс, тогава заданието декларира, че то може да създава точки на съхранение на текущото си състояние, да се спира временно и да се възобновява; двата декларирани метода са за създаване на такава точка на съхранение и за възстановяване на заданието до определено състояние от такава точка.

- **TaskDescriptor** – този допълнителен интерфейс служи за извличане и конфигуриране на параметри за изпълнението на заданието; видовете параметри са: идентификатор на използвания тип комуникация; брой на инстанциите от заданието, които трябва да се изпълнят; списък от параметри за инициализация на инстанциите за изпълнение.

ALiCE

Изпълнението на задания в тази система се характеризира със свой специфичен програмен модел, който се опира върху работата на четири компонента [7]:

- **TaskGenerator** – работи на централна машина с функцията да генерира задания при подадено приложение за изпълнение; тези задания после се разпределят по възлите от ресурсния брокер; програмистът сам определя обстоятелствата за генериране на задания в *main*-метода на компонентата;
- **Task** – самото задание, което работи в изпълнителен възел на системата; програмистът определя хода на изпълнение в метода *execute*;
- **Result** – този обект може да има произволна структура и служи да съхранява изходния резултат от изпълнението на заданието;
- **ResultCollector** – този компонент работи при потребителската машина, която е подала заявката за изпълнение; характеризира се с метод *collect*, където програмистът кодира извличането на резултата и неговата последваща обработка.

Изводът е, че и при двете архитектури се дефинира строго определена структура на заданията и заявките за изпълнение, които отговарят на особеностите на съответните грид-системи.

3. Функционално проектиране на Node Service (NS).

3.1. Общ дизайн на проекта.

В предишната точка направихме въведение към общите изисквания и функционалност на системната услуга. Стана ясно, че тази услуга работи на най-ниското възможно ниво в грид-системата, а именно в неговите ресурсни възли, където се изпълняват заявките. В същото време обаче, като част от грида, Node Service модула е необходимо да взаимодейства със системните услуги от по-високо ниво, които зависят от него, а именно услугата за управление на ресурси (RMS) и централната услуга за наблюдение (MS), които работят в режим на клъстерно ниво (нивото, което е непосредствено над локалното ниво, състоящо се от самите възли в грида). Поражда се нуждата от строго дефиниран канал за комуникация между централните модули и системната услуга по отделните възли. От друга страна, е желателно протоколът за комуникация да скрива възможно най-много детайли от техническата реализация на канала. По този начин директните потребители на

Node Service могат да бъдат по-лесно изолирани от вътрешни промени в комуникационния протокол и да се избегне нуждата от тяхна последваща адаптация към нови изисквания за техническото осъществяване на връзката с даден възел. Дефинира се необходимостта от изграждането на *комуникационен слой* като част от архитектурата на системната услуга, която представлява границата между клъстерното ниво, на което работят централните услуги, които комуникират с Node Service, и локалното ниво на възела, който е част от клъстера.

Избраното дизайнерско решение за имплементиране на споменатия комуникационен слой с постигане на абстракция на техническите детайли по осъществяване на връзката с възлите, е с помощта на т.нар. ***Node Service API***, т.е. *приложен програмен интерфейс за работа със сервиза на отделните възли*. Една от функциите на API-то е да предоставя абстракция на логическа връзка с модула на даден възел. Тази абстракция, от една страна, скрива детайлите от техническата реализация на комуникационния канал, а от друга страна играе ролята на *прокси* (представител) на системния модул от даден възел, към който могат директно да се адресират задачите. По този начин, можем да кажем, че API-то надгражда *комуникационния слой*, скривайки го *почти* изцяло от потребителите на Node Service, с друг, *представителен слой*, който служи като фасада за приемането на заявки за изпълнение. Казваме „почти", защото все пак за свързването с представител от даден възел все още е необходимо да се подадат някои детайли за локализацията му като например Интернет адрес *(IP address)* на възела и евентуално, порта, на който работи Node Service.

Стигаме и до най-вътрешния слой на системния модул, а именно *изпълнителния (изчислителния),* който работи локално на възела и обслужва подадените заявки през комуникационното ниво. Вътрешната реализация на изпълнителния слой, разбира се, е изцяло скрита за клиентите на Node Service – това, което е необходимо те да знаят за работата със заявките, е да получават информация за техния ход на изпълнение. Следователно, следващата подсистема в общата схема на NS е тази на неговата същинска сервизна част. Тя е натоварена изцяло с изпълнителни функции и е желателно да бъде изолирана от особеностите на API-то, за да се избегне взаимното влияние на промени в реализацията на една от двете подсистеми.

Така се стига до формулирането на *свързващото звено* между подистемите на сервиза и на клиентсктото API. Ще наречем условно този трети модул ***Node Service Task Framework***, тъй като той изцяло моделира дизайна на заявките за изпълнение и възможностите на сервиза и на негова база се градят останалите два модула. Всъщност, този модул попада изцяло в *представителния слой* от архитектурата. На неговото подробно разглеждане ще се спрем в следващата точка. Тук ще отбележим, че той изпълнява няколко основни функции:

- Дефинира интерфейса за работа със сервизната част от грид услугата – това е информация, която е от интерес предимно за проектанти на API-то.
- Моделира структурата на заявките и заданията за изпълнение – това има пряко отношение към проектирането на изпълнителната част и за

информацията, която се предоставя и изисква от Node Service във връзка с обслужването на заявките.

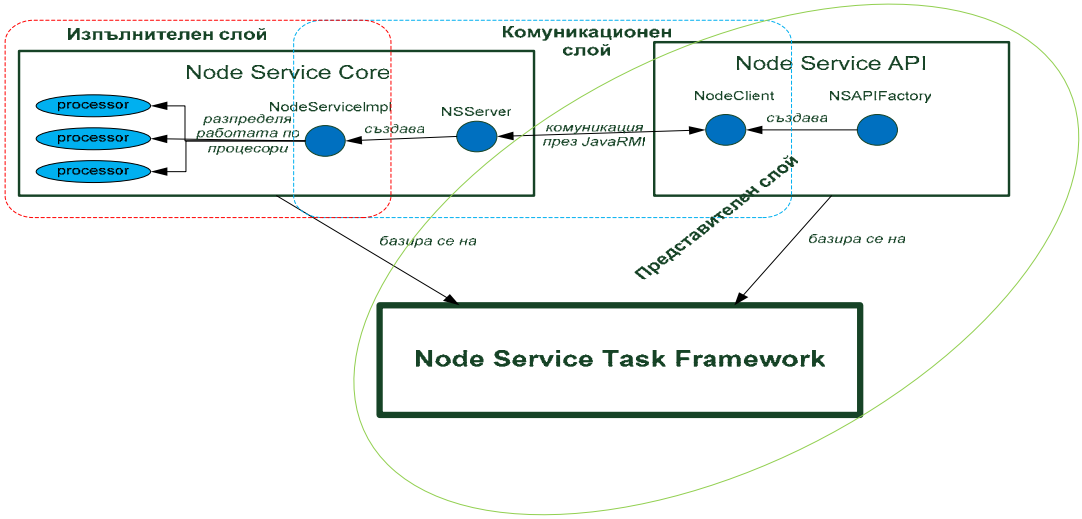
- Очертава цялостната функционалност на предоставените услуги.
- Служи като база за изграждането на подистемите на API-то и на сервизната част. Осигурява *изолационен слой* между имплементациите на двете подсистеми.

Последната характеристика има много ценното свойство, че прави изградената архитектура изключително гъвкава на промени, стига желаната функционалност да не излиза от рамките на дефинирания в *платформената част* модел – конкрентите имплементации на сервизната подсистема и на приложния интерфейс стават напълно заменяеми с други техни реализации, които могат да бъдат по-подходящи при други условия.

Нека да обобщим участието на трите подсистеми в отделните слоеве от дизайна на Node Service-а във *фигура 2*, както и в следната табличка:

Подсистема:	**Участва в:**
Node Service API	Представителен слой. Комуникационен слой.
Node Service Task Framework	Представителен слой.
Node Service Core	Комуникационен слой. Изпълнителен слой.

Таблица 1. Участие на подсистемите на Node Service услугата в слоевето на нейната архитектура.



Фигура 2. Подсистеми и архитектурни слоеве на Node Service.

3.2. Платформена подсистема на модула за управление на възли – Node Service Framework.

В предишната точка подчертахме, че тази подсистема моделира фунционалността на модула и на заявките за изпълнение. От друга страна, тази подсистема е фасада за имплементацията на очертания модел. Технически, подистемата представлява набор от интерфейси, които се конкретизират от API-то и от сервизната част, които се изграждат на тяхна база.

Преди да разгледаме конкретните части на подсистемата, нека да припомним основните типове заявки, които Node Service-а е длъжен да обслужва:

- Изпълнението на потребителски приложения, написани на програмния език Java, които са компилирани в готов вид като самостоятелни .class файлове или като пакетирани .jar архиви.
- Изпълнението на произволни потребителски програми, написани на Java, със специфиране на входната точка на изпълнение под формата на публично деклариран метод в програмния код. Дефинирането на класа, метода и аргументите за изпълнение трябва да става динамично, в процеса на създаване на заявката.
- Зареждането на програмен код, компилиран на Java, във възела от грид-системата, с цел да бъде в постоянна готовност за приемането и обработката на набор от входни данни. Тази функция ще наричаме поддръжка и изпълнение на *персистентни услуги*, за разлика от горните типове задания, които са за еднократно изпълнение на възлите и имат преходен характер.
- Осигуряване на *жизнен жикъл на персистентната услуга* (зареждане, стартиране, спиране и отстраняване от възела) и дефинирането на механизми за захранването му с входни данни, като същевременно се осигури максимална гъвкавост за потребителите при формулирането на заявките за изпълнение, подобно на работата с *преходните услуги*.
- Осигуряването на механизми за навременно уведомяването на потребителите за хода на изпълнение на заданията и за събирането и изпращането на получените резултата по предварително дефинирани в заявката критерии.

Конкретните функционални области на платформената подсистема са следните:

3.2.1. Струкурен модел на заявките за изпълнение.

В посочените в точка 2.2 примери за грид-системи специфичното е, че се поставят определени изисквания към формата и структурата на приложенията за изпълнение. Предимството на този подход е, че логиката за изпълнение разполага с необходимата специфична информация за ефективен контрол над изпълнението и извличането на резултатите. Недостатък е, че всеки програмист, който иска да се възползва от възможностите на грид-системите, трябва да се съобразява с наложения програмен модел и да нагажда своите приложения към него.

Началната мотивация за изграждането на програмния модел в настоящата реализация на подистемата на Node Service платформата, се състои в това, че модулът за управлението на възли трябва да разбира и да се справя с изпълнението на произволни приложения, написани на Java, които да улеснят в максимална степен потребителите на GrOSD. Универсалността на модула за изпълнение идва обаче с недостатъка, че липсва специфичната информация за възможностите на заданията, чрез които да се упражни по-ефективен контрол върху техния жизнен цикъл на изпълнение. Възприетият компромисен вариант е към общият поток на изпълнение да се добавят специализирани разклонения, които се задействат в момента, в който се установи, че заданието притежава някои определени свойства. За Node Service Task Framework това означава, че той може да се разширява с допълнителни интерфейси, наподобяващи тези в JGrid архитектурата, за осъществяването на по-специални задачи. В тази глава обаче акцентът е върху общия поток на обработка на произволно задание и как се структурира заявката в такъв универсален контекст.

3.2.1.1. Структурен модел на заданията за еднократно изпълнение.

В най-общия контекст, изпълнението на едно приложение преминава през три основни етапа – започва се с подаването на входните данни, следва самото стартиране и изпълнение на приложението, и се приключва с извличането на резултата. Грижата на Node Service Task Framework е да осигури модел, в който се дефинират входните и изходни параметри на заданието, да се уточнят детайлите за неговото стартиране и да се определят механизмите за обмена на данни между клиента на Node Service и самото задание, първо, по отношение на подаването на входните аргументи, и, второ, по отношение на събирането и връщането на резулата.

Какво е необходимо да знаем, за да можем да изпълним дадено задание написано на Java ? На първо време, необходимо е да разполагаме с компилираните класове, които взимат участие в заданието. Файловете на тези класове може да се предоставят по отделно, но най-често се пакетират в .jar архиви по определена спецификация. Проектирането на платформената подсистема за специфицирането на задания е тясно свързано с възможностите на т.нар. *Java Reflection API* – стандартен модул в JDK, с чиято помощ може да се извлече информация за структурата на класове, предоставени в компилиран вид, и, по-важното, да се извършват операции върху тях на базата на декларираните в тях методи. Следователно, трябва да се уточнят конкретният клас и неговият публичен метод, с чието извикване се стартира заданието. Самият метод се нуждае от входни аргументи, както беше посочено по-горе. В Java името на метода и списъкът му от входни аргументи еднозначно дефинират метода в рамките на класа, в който той участва. Респективно, с това се определя и типа на самият резултат, който се връща от метода след края на неговото изпълнение. Task framework обаче допуска едно разширение на работата с входно-изходни параметри – освен специфицирането им във връзка със сигнатурата на метода за изпълнение, се предполага, че от една страна, част от входните данни могат да се четат от външни файлове (например, като конфигурация на заданието), които лиспват в сигнатурата и трябва да се посочат изрично от клиента на платформата, а от друга страна, изпълнението може да запише резултата си (или поне част от него) също във външни

файлове, които да се открият на даден принцип на хоста на изпълняващия подмодул.

Следват групите от интерефейси, с които става възможно да се изгради структурата на заявката за изпълнение:

1) Специфициране на входните аргументи:

NSTaskArgument – общ интерфейс за дефиниране различните типове входни аргументи; всеки нов тип трябва да го наследява; на този етап той има два подинтерфейса:

- **SerializableArgument –** този интерфейс капсулира аргументи от типа *java.io.Serializable* – тяхно свойство е че могат да се сериализират във вид, който им позволява да се запишат като персистентни данни на файловата система и да се прехвърлят по мрежата от един хост на друг; последното прави това свойство *изискване* за аргументите на заявката с оглед на начина им на използване в една разпределена среда, каквато е грид-системата. Интерфейсът декларира един метод за извличане на името на класа на аргумента (необходимо за избягването на двусмислия при обработката на стойността на аргумента) и друг за извличане на неговата стойност.
- **FileArgument –** файловите обекти като аргументи на заданието имат нужда от по-различен режим на тяхната обработка, тъй като стойността им в Java само задава пътя до физическото местоположение на файла, който обикновено има смисъл само на хоста, на който се намира той; но от гледна точка на това, че изпълнението протича на друго място, е необходим трансфер на истинския файл до хоста на изпълнение. Интерфейсът декларира три метода – за получаване на местоположението на файла във вид на URL (Unified Resource Locator), за уточняване дали полученият URL е от локалната файлова система на клиента или е от публично достъпен мрежови ресурс, и за указване на относителния път на файла, където той да бъде копиран на хоста на изпълнителния подмодул. Ще обърнем внимание, че ако методът за изпълнение приема файлов аргумент, който не изисква физически трансфер (например аргументът може да указва местоположение на файловата система, на която протича изпълнението, където да се създаде и да се запишат някакви данни от него в резултат от работата на метода), то този аргумент трябва да се капсулира като **SerializableArgument** (това в възможно, защото типът *java.io.File* наследява интерфейса *Serializable*).

2) Специфициране на изходния резултат.

NSTaskResult – това е общият интерфейс, през който клиентът може да извлече резултата от изпълнение. Интерфейсът декларира редица методи, чрез които клиентът може да получи резултата в удобен за последващата му обработка вид. Основният метод връща резултата като обект от типа *java.io.Serializable* – последното отново е изискване, за да бъде възможен трансфера на резултата между различните хостове. За случаите, когато резултатът е от примитивен тип, са предвидени методи за извличането му във всеки един от дефинираните в Java примитивни типове. Най-накрая, когато

изпълнението е генерирало файлове, те също могат да се получат като списък от URL-и към локалната файлова система на клиента, където те се копират при извикването на метода за извличане. Трябва да обърнем внимание, че е напълно допустимо част от резултата да се търси като обект, върнат от изпълнението на метода, а друга част от него във вид на изходни файлове. Тази характеристика е допълнително мощно средство, заложено в платформения модул, за справяне с широк кръг от задачи. Декларира се и един допълнителен метод, с който да се вземе коментар за получения резултат, генериран от изпълнителния модул – последното е най-подходящо да се използва при визуализация на резултата от страната на клиента.

NSFileResultLocator и **NSFileFilter –** тези два интерфейса са помощни за претърсване на файловата система на хоста на изпълнение за файлове, които съдържат резултата.

NSFileFilter декларира булев метод, с който се разпознава дали подаденият файл е 'интересен' за клиента като резултат – наследниците на този интерфейс трябва се дефинират изцяло от клиента на NS API-то.

NSFileResultLocator декларира: метод, който връща списъка с относителните файлови пътища, които да се претърсят за изходен резултат; метод, който връща филтъра за разпознаване на желаните от клиента файлове; булев метод, с който се определя дали търсенето да се разпростре и до поддиректориите (това допълнително забавя търсенето).

3) Специфициране на класа и метода на изпълнение.

NSTaskTargetClass – този интерфейс дава необходимата информация за класа, който е обект на заданието за изпълнение; декларира метод за името на класа и два метода, с помощта на които може да се инстанциира обект от класа – единият връща списък с имената на класовете на аргументите за инициализация, като по този списък се намира необходимия конструктор; вторият връща списък от **NSTaskArgument** обекти със самите стойности.

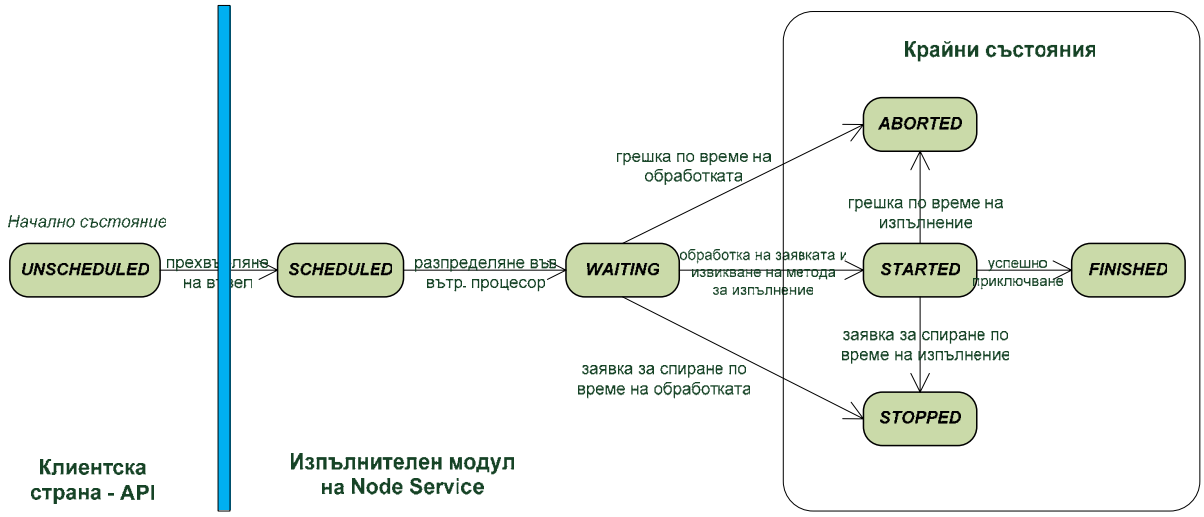
NSTaskMethodPrototype – този интерфейс е абстракция на понятието „сигнатура на метод в Java" – декларира метод, който връща името на метода, и друг, който връща списък на имената на типовете на аргументите. Заедно с информацията от **NSTaskTargetClass**, може точно да се определи с извикването на кой метод от кой клас се стартира изпълнението на заданието. Интерфейсът се разширява от **NSTaskMethod,** който добавя един метод, с който да се вземат стойностите на входните аргументи отново във вид на **NSTaskArgument** обекти.

4) Мониторинг на заданието.

Не е трудно да се види, че с горните интерфейси се структурира почти цялата необходима информация, която ни е необходимо да знаем за изпълнението на заданието. Към тази информация трябва да се добавят единствено файловете с компилиран код на заданието и други файлове, които играят ролята на входни параметри, неназовани в сигнатурата на методите.

За мониторинг на хода на изпълнение обаче са необходими още няколко допълнителни елемента:

- Поредица от добре дефинирани състояния на заданието по време на изпълнение. Те очертават жизнения цикъл на заданието от момента на неговото създаване до момента на неговото приключване. На езика на платформения подмодул, тези състояния се дефинират като константи на изброения тип **NSTaskActionState** и носят наименованията: ***UNSCHEDULED*** – заданието е в това състояние в момента на неговото създаване с помощта на NS API-то, преди да е станало ясно на кой възел ще се прехвърля за изпълнение; ***SCHEDULED*** – означава че вече е определено на кой възел ще се изпълнява заданието; ***WAITING*** – състоянието на заданието непосредствено след като е прехвърлено на възела за изпълнение; ***STARTED*** – когато заданието е стартирано за изпълнение; ***FINISHED*** – означава че заданието е приключило успешно и може да се вземе резултата му; ***STOPPED*** – когато заданието бъде спряно по желанието на клиента, след като е било стартирано[1]; ***ABORTED*** – означава, че нормалното изпълнение на заданието е било прекратено поради възникнала грешка. (*фигура 3*)



Фигура 3. Диаграма на преходите между състоянията на заданията.

- Мониторингови обекти, които съществуват в потребителската виртуална машина, но се достъпва с отдалечени извиквания чрез JavaRMI за нотификация по събитията, за които са регистрирани, от виртуалната машина на изпълнителния модул. Node Service Task Framework дефинира такъв мониторинг за следните събития в хода на изпълнение на заданието – 1) промяна на състоянието, дефинирано според етапите посочени по-горе; 2) приключване на заданието по нормален път или след възникнала грешка.

Моделът, по който клиентите на NS API-то се уведомяват за събитията, е следният – в кода на клиента се имплементват определени интерфейси, декларирани в подсистемата на Node Service Task Framework. Интерфейсите съдържат методи, които се извикват по RMI от модула за изпълнение, като при

[1] На този етап услугата за изпълнение на задания не поддържа възобновяване или повторно стартиране на произволно приложение след неговото спиране или приключване. За тази цел платформената подсистема трябва да се допълни със специализирани интерфейси за типовете задания, които поддържат подобни операции.

извикването им се подава като входен аргумент обект, който съдържа необходимата информация за настъпилото събитие. Клиентският код в имплементацията на мониторинговите интерфейси просто трябва да обработи подадения аргумент по желания от него начин.

В платформения подмодул се дефинират следните мониторингови интерфейси:

NSTaskListener – това е основният клас, от който се типизират всички останали видове мониторингови обекти; наследява се от:

NSActionStateListener – за мониторинг на промени в състоянието на заданието; методът за уведомяване за промени има за аргумент обект от типа **ActionStateChangedEvent**, от който може да се разбере какви са новото и старото състояние на заданието (по смисъла на константите в **NSTaskActionState**) и каква е причината за промяната.

NSTaskLifecycleListener – на този етап, с този интерфейс клиентът се уведомява за две важни събития от жизнения цикъл на заданието (и респективно, има два метода за всяко едно от тях) – 1) прекратяването му заради грешка, като входният аргумент на метода дава информация за събраните грешки при спирането, с което да се разбере причината за извънредната ситуация; 2) успешното приключване на заданието, като при това положение входният аргумент на метода е от типа **NSTaskResult**, който беше разгледан подробно по-горе.

Платформената подсистема съдържа два класа – **NSActionStateAbstractListener** и **NSTaskLifecycleAbstractListener**, които имплементват двата мониторингова интерфейса с празни тела на методите за уведомяване за настъпили събития. Клиентските програми трябва да разширяват именно тези два класа.

5) Специфициране на заданията за еднократно изпълнение.

До този момент очертахме модела на необходимия инструментариум, с който можем да дефинираме заявката за изпълнение на заданието от даден компилиран клас и механизмите за неговия мониторинг. Интерфейсите, които описват структурата на заданието в Node Service Task Framework са следните:

NSTaskRemote – основният клас, от който се типизират всички видове задания за изпълнение. Той е абстрактен и дефинира минималната полезна функционалност на заданието в контекста на услугата, която е най-вече свързана със следене на промените в състоянието му по време на изпълнение. Декларира методи за: 1) извличане на индентификатор на заданието – чрез него той се разпознава на хоста на изпълнителната подсистема; 2) извличане и промяна на състоянията от жизнения цикъл на заданието; 3) за добавяне и извличане на мониторинговите обекти, които следят за важните събития от изпълнението на заданието.

Наследява се от **NSJavaTask,** който е по-конкретизиращият интерфейс за заданията за еднократно изпълнение във възлите на грида. Чрез декларираните в него методи: 1) се извличат обекти от типа **NSTaskTargetClass** и **NSTaskMethod,** с които респективно се узнават класа, метода и аргументите на изпълнение; 2) списък на компилираните файлове, които участват в изпълнението и могат да бъдат отделни .class файлове или

.jar архиви; класът за изпълнение се търси именно сред тях; тези файлове се капсулират в аргументи от тип **FileArgument** и се прехвърлят на хоста на изпълнителния модул; 3) списък от допълнителни файлове, необходими за изпълнението (например конфигурационни файлове и т.н.), които също се капсулират като **FileArgument** обекти, за да се копират на хоста на изпълнителния модул; 4) се взима или добавя обект от типа **NSFileResultLocator** за получаване на резултата от изходни файлове.

3.2.1.2. Структурен модел на заявките за изпълнение на персистентните услуги.

За изграждането на структурния модел на персистентните услуги, на практика, можем да използваме като база целия инструментариум, изграден при заявките за еднократно изпълнение. Особеността при персистентните услуги е, че за разлика от заданията за еднократно изпълнение, те имат по-усложнен жизнен цикъл, свързан с елементи на по-устойчиво поведение. На първо време, те трябва да са способни да обработват повече от една заявка в рамките на жизнения си цикъл, при това с промени във входните параметри. Едно допълнително изискване към тях е да се зареждат на възела за изпълнение заедно със стартирането на изпълнителния модул, веднъж след като са били качени на него – с други думи, допълнителна характеристика на устойчивост при тях е да „преживяват” спирането на модула.

Повтаряемостта при изпълнението на този тип услуги може да се отнася до различни елементи от заданието (тук взаимстваме от терминологията от заданията за еднократно изпълнение). Несъмнено, фиксиран трябва да бъде наборът от компилирани файлове, на базата на който работи услугата. Към неизменните елементи можем да причислим и класа и метода за изпълнение, които практически съставляват ядрото на услугата. Един необходим атрибут е някакъв ключов идентификатор, уникален в рамките на изпълнителния модул на възела от грид-системата, чрез който да може да се адресира услугата през външното API за зареждането му с входни данни и т.н.

Node Service Task Framework възприема доста либерален подход към формулирането на заявките за изпълнение, като смята за фиксирани единствено ключовия идентификатор на персистентната услуга и компилираните файлове, които използва. Като допълнение, при качването на услугата на възела, се дефинират клас и метод за изпълнение по подразбиране, които се задействат при подаването на данни при последващите заявки. Но тези заявки обаче могат да дефинират и различни методи и класове за конкретното им изпълнение за конкретните данни. Избраният подход е особено подходящ за сценарий на използването на потребителски библиотеки от функции, които се зареждат на даден възел и могат да се ползват различни предоставени от тях услуги при различните заявки.

Реализацията на концепцията дали услугата ще бъде устойчива на спирането на изпълнителната подсистема или не се оставя на имплементацията на самата подсистема.

Тъй като голяма част от елементите от структурния модел вече бяха подробно разгледани в предишната точка, тука ще посочим направо двата допълнителни

интерфейса от платформения модул, с които се поддържат персистентните услуги в Node Service:

NSJavaPersistentTaskUpload – с него се дефинира задание за зареждането на персистентната услуга; с методите от интерфейса се извлича информация за ключовия идентификатор на услугата, за списъка от компилирани файлове, от който се зареждат класовете й (методът е напълно аналогичен на заданията за еднократно изпълнение), за класа и сигнатурата на метода за изпълнение по подразбиране.

NSJavaPersistentTaskInvocation – за дефинирането на заявки за изпълнение на персистенти услуги, които вече са заредени на възела; декларира методи за: 1) промяна на класа и метода, които са обект на изпълнение, но само в рамките на тази заявка; 2) за указване на ключовия идентификатор на персистентната услуга – по този начин изпълнителният модул може да разбере за коя услуга е адресирана заявката; 3) за извличане на списъка с инициализационни параметри на класа за изпълнение – по този начин се позволява да се конфигурира класа за изпълнение по различен начин при всяка заявка; 4) за извличане на списъка с допълнителни файлови параметри на заданието; 5) за добавяне и извличане на инстанция на **NSFileResultLocator,** чрез който да се вземат файловете, получени в резултат на изпълнението. Този интерфейс предоставя голяма гъвкавост на заявките, позволявайки всеки път да се променят критериите не само за подаваните входни параметри, но и за получаването на крайния резултат във вид на файлове, ако има такива.

3.2.2. Интерфейс на предлаганите услуги от изпълнителната подсистема на даден възел в грида.

Услугите са дефинирани в класа **bg.unisofia.fmi.grosd.ns.remote.NodeService** и в съответствие с дефинираните по-горе изисквания предоставените услуги са:

- Зареждането на задание за изпълнение на съответния възел – уточнява се типа на завката, самото задание и се подава обект за обратна връзка с клиента на сервизната част, в нашия модел това е API-то. Типовете заявки са изброени в отделен клас (**NodeServiceTaskType**) и в момента, в който се дефинира нуждата от нов тип, той се добавя на това място - сервизният подмодул има грижата да знае как да го обслужи. Самото задание е някакъв подтип на основния клас **NSTaskRemote** - с допълнитеното уточняване на типа на заявката става ясно как да се третира заданието. С други думи, параметърът за тип на заявката уточнява какво трябва да се направи (напр., да се изпълни някакъв конкретен тип услуга), докато самото задание е обект най-вече с информационна стойност, който съдържа в себе си входните данни за изпълнение. Накрая, обектът за обратна връзка с API-то е необходим за нуждите на комуникационния слой, а именно за обмена на файлове.
- Изпълнението на заявки за управление на заданията, които са стартирани на възела, най-вече за тяхното прекратяване при възникнала нужда. Управлението на *персистентните услуги* е обогатено с повече команди, свързани с тяхното стартиране, спиране и отстраняване от

възела. Посочването на заданията на възела става с подаването на техния идентификатор, който е уточнен още при създаването им.

- Команда за зачистване на излишните файлове, които са останали от изпълнението на приключило/спряно задание. Тези файлове не се изтриват автоматично с презумпцията, че могат да съдържат изходни данни, които ще се извлекат по-късно от клиента на API-то. Отново заданието се уточнява с негов идентификатор.

3.2.3. Допълнителни модули на платформената подсистема.

В Node Service Task Framework са дефинирани и няколко допълнителни интерфейса, които са най-вече от спомагателно естество за изпълнението на основните типове задания. Ще разгледаме тези от тях, които засягат директно функционалността на услугата за управление на възли.

1) NodeServiceTaskType – както се спомена по-горе, с този изброен клас се посочват типовете задания, които изпълнителният модул поддържа. На този етап, дефинираните видове приложения са три: еднократно изпълнение на задание, дефинирано според очертания от Node Service Task Framework структурен модел; зареждане на персистентна услуга на възел от грида; подаване на входни данни за персистентната услуга и нейното извикване за тяхната обработка.

2) NSOutputStreamWrapper – този интерфейс капсулира изходен поток за прехвърлянето на данни в двоичен вид; имитира се работата на *java.io.OutputStream,* но в контекста на *JavaRMI*; декларират се методи за запис на байтове в потока, за затварянето му и за получаването на URL към файла за запис на локалната файлова система. Този интерфейс е един от малкото с дефинирана директна имплементация в рамките на платформения модул, като имплементацията се използва от другите две големи подсистеми на Node Service, а именно API-то и изпълнителния модул (сервизната част).

3) NodeServiceClientCallback – интерфейсът за обратна връзка с NS API-то, като на този етап неговата функционалност е свързана с трансфера на файлове от клиентската машина до възела в грида. Декларира метод за получаването на **NSOutputStreamWrapper** обект към файла на клиентската машина и друг метод за копирането на файла до определена дестинация.

4) Беше посочено, че за дефинирането на по-специфични задачи платформената подсистема може да бъде разширявана с интерфейси, които дават допълнителна функционалност над приложенията-обекти на заданието. Пример за подобно разширение е пакетът ***bg.unisofia.fmi.grosd.ns.application,*** където е добавен интерфейса **StoppableApplication**. Клиентски приложения, които наследяват този интерфейс, трябва да дефинират метод за тяхното спиране по време на изпълнение, който се използва от изпълнителния модул при клиентска заявка за тяхното преждевременно прекратяване. По този начин клиентското приложение може сама да дефинира своето поведение при заявка за спиране; в противен случай, за стандартни приложения, изпълнителният модул ще изпозлва собствен механизъм за спиране, който обикновено е свързан с външна намеса, не е толкова ‘елегантен’ и може да има странични ефекти – това обаче е цената за универсалността на модула

3.3. Приложен програмен интерфейс на модула за управление на възли – Node Service API (NS API)

Характерната особеност на Node Service API –то е, че то работи изцяло откъм клиентската страна при дефинирането и изпращането на заданията за Node Service услугата. Негови клиенти в конктекста на GrOSD представляват определени централни сервизи от клъстерното ниво (RMS, MS), но принципно могат да бъдат приложения и от по-общ характер.

Модулът на API-то изпълнява две основни функции в схемата на изграждане на тази услуга:**1)** то представлява конкретно изпълнение на структурният модел на заданията за обработка, дефиниран в NS Task Framework; **2)** осигурява абстракцията за връзка с изпълнителния модул на услугата на конкретен възел от грида. Особеностите на API-то ще бъдат разгледани в съответствие с това разделение, като ще обърнем по-малко внимание върху техническите детайли от съдържанието на конкретните класове, а повече върху особеностите на техния дизайн и заложените идеи, които са характерни най-вече за този модул.

3.3.1. Конкретизация на структурния модел на заданията, дефиниран в NS Task Framework.

Очертаният в NS Task Framework модел за структуриране на заявките за изпълнение намира своята имплементация на представително ниво в модула на Node Service API. Интерфейсите от платформения компонент тук се запълват с конкретно съдържание, което подлежи на обработка в третия модул на Node Service услугата – изпълнителния. В общи линии, ресурсните методи, декларирани в интерфейсите за извличането на специфични данни, подсказват какво трябва да бъде съдържанието на конкретизиращите ги класове в API-то, което де факто стъпва върху вече изградените от платформата основи. Поради тази причина, няма да се спираме в детайли върху тяхното разглеждане.

Отново ще отбележим, че настоящето API не представлява единственият възможен вариант за реализиране на NS Task Framework, което се гарантира от абстракното съдържание на самия framework. Разделението между интерфейси и техни конкретни имплементации в API-то, от една страна, цели да предложи множество възможни варианти за взаимовръзка между двата модула на Node Service услугата, дефинирани от проектанти на това API, а от друга страна, осигурява фасада за клиентите на API-то, която скрива техническите детайли по реализацията на известните елементи от заявката и оставя на преден план тяхната функционална същност.

Конкретната реализация на настоящето Node Service API обаче осигурява още един слой на изолация на неговите клиенти от вътрешното представяне на заданията и заявките за тяхното изпълнение. Въвеждат се т.нар. *фабрични класове (factory classes)*, които осигуряват единствения интерфейс за инстанцииране и изграждането на отделните елементи от заявката до нейната пълна дефиниция. Конкретните класове, имплементиращи тези елементи, са капсулирани с конструктори с ограничен режим на достъп, които са ползваеми единствено от споменатите фабрични интерефейси.

Принципът на изграждане на фабричните класове е следният – всеки един от тях, в зависимост от неговото предназначение, предлага за клиента набор от *фабрични методи (factory methods)*, които вземат входни данни за

изграждането на елемент от заявката и връщат като резултат интерфейса, съответстващ на този елемент от NS Task Framework, зад който стои конкретен обект, дефиниран в API-то. Един прост пример за такъв метод е **NSTaskMethodFactory.createSerializableArgument(Serializable),** който капсулира *java.io.Serializible* аргумент в имплементация на **SerializableArgument.**

Този подход предлага две предимства:

- фабричните класове позволяват да се дефинират различни комбинации от входни параметри за изграждането на елемент от заявката;
- клиентът е напълно изолиран от начина на работа на фабричните методи, които, в зависимост от входната комбинация от параметри и техните стойности, могат да се обръщат към различни имплементиращи класове от NS API-то, скрити зад фасадата на един определен интерфейс от платформения модул.

Долната таблица описва съответствията между интерфейсите от платформата и техните имплементиращи и фабрични класове в NS API-то:

Интерфейс	Директен имплементиращ клас	Фабричен клас
NSTaskRemote	NSAbstractTask	NSTaskFactory
NSJavaTask	NSJavaTaskImpl	
NSJavaPersistentTaskUpload	NSJavaPersistentTaskUploadImpl	
NSJavaPersistentTaskInvocation	NSJavaPersistentTaskInvocationImpl	
NSTaskTargetClass	NSTaskTargetClassImpl	NSTaskTargetClassFactory
NSTaskMethodPrototype	NSTaskMethodPrototypeImpl	NSTaskMethodFactory
NSTaskMethod	NSTaskMethodImpl	
NSTaskArgument	-	NSTaskArgumentFactory
SerializableArgument	SerializableArgumentImpl	
FileArgument	FileArgumentImpl	
NSFileResultLocator	NSFileResultLocatorImpl	NSFileResultLocatorFactory

Таблица 2. Съответствие между интерфейси, имплементиращи класове и фабрични класове в структурния модел на заданията на Node Service услугата.

Ще отбележим, че по отношение на имената на трите типа класа от колоните на таблицата, с някои изключения, се спазва следната конвенция:

Име на интерфейс	Име на импл. клас	Име на фабричен клас
<име на интерфейс>	*<име на интерфейс>*Impl	*<име на интерфейс>*Factory

Видно е, че за повечето фабрични класове, от значение е името на най-общия интерфейс в неговата функционална област.

Предназначението на интерфейсите беше детайлно разгледано в предната глава.

Дизайнът и техническата реализация на имплементиращите ги класове не се отличава с някакви съществени особености, освен споменатите дотук. Ще посочим само два интересни момента сред набора от API класове:

- **NSAbstractTask** е абстрактен клас, който не може директно да се инстанциира, но от него се наследяват всички конкретни имплементации за обекти на задание. Ролята му е аналогична на тази на **NSTaskRemote** за интерфейсите на различните разновидности на задания, дефинирани в NS Task Framework. Класът конкретизира логиката на работа на общите методи на заданията. Един особено важен метод има задачата да уведомява клиентите на заданието за промените в етапите на неговата обработка в смисъла на константите дефинирани в типа **NSTaskActionState**. Причината имплементацията на този вид мониторинг да е в API-то, а не в изпълнителната част, е че жизненият цикъл на заданието да не протича изцяло на възела на изпълнение (обърнете внимание на дефиницията на състоянията, дадени в описанието на платформената подсистема) и някои етапи трябва да се отчитат още преди заданието да бъде заредено там. Затова API-то е избрано като най-удачното място за тази цел.
- За разлика от **NSTaskRemote,** друг общ интерфейс – **NSTaskArgument** няма конкретна имплементация в NS API-то, тъй като на този етап не декларира някаква обща функционалност за типовете входни аргументи на методите за изпълнение.

Ще завършим прегледа на тази част от подсистемата на NS API-то с фабричните класове. Особеност на техния набор от методи е да олекотят клиентския код за съставянето на елементите на заявките за изпълнение. Това се постига чрез дефинирането на голям брой комбинации от входни аргументи, които методите приемат – от такива, чрез които заявката може да се уточни и до най-дребния детайл, който съществува в изискванията на NS Task Framework за създаването на задание, до такива, които определят само най-важните моменти, а останалите детайли се дефинират от API-то със стойности по подразбиране. Спестяването на голям брой входни аргументи спомага за опростяването на клиентския код.

Типичен пример за използването на различни комбинации са фабричните методи в **NSTaskArgumentFactory** за дефинирането на подтиповете на **NSTaskArgument**, някои от които изискват както стойността на входния аргумент, така и името на класа, под който да се обработи от изпълнителния модул, а за други е достатъчна само стойността, тъй като по обекта, който служи за създаването на входния аргумент за метода на изпълнение, може да се прецени кой е неговият реален клас. Като допълнителен пример за ползата от избрания подход, най-мощният и удобен за ползване фабричен метод е **createArgsList**, който е способен да преобразува списък от обикновени Java обекти в списък от **NSTaskArgument** инстанции, като вътрешната логика се грижи за разделението им между подтиповете **SerializableArgument** и **FileArgument** в зависимост от това дали оригиналният входен обект наследява съответно типовете *java.io.Serializable* или *java.io.File.*

3.3.2. Клиент на заданията за изпълнение от Node Service услугата.

Вече беше посочено, че тази част от NS API-то има грижата да осигури за своите клиент обект, който представлява логическата връзка за комуникация с

изпълнителната подсистема на Node Service услугата на конкретен възел. Този обект играе ролята на посредник между клиента на API-то и сервизният модул, работещ на възела от грида, като ретранслира заявките до възела, изолирайки клиента от техническите подробности по предварителната обработка на заявката (ако има такава) и използвания вътрешен комуникационен протокол за връзка. В този смисъл, този обект е най-прекият клиент на ядрото на услугата Node Service, поради което неговият интерфейс носи името **NodeServiceClient.** Ще обърнем внимание на следните негови особености:

- Начинът му на получаване от клиентите на API-то.
- Вътрешният комуникационен механизъм за връзка с възлите на Node Service.
- Функционалността, която той предлага – типове заявки за изпълнение и други услуги.

Клиентите на API-то получават инстанция на NodeServiceClient-а по същия начин, по който това става при конструирането на елементите на заявката, поради вече изложените предимства на този подход. Отново това става с участието на три елемента – интерфейс за работа, имплементиращ клас на интерфейса и фабричен клас за създаването на инстанция. Различното в случая е, че дефинициите на интерфейса и имплементацията се намират в пакета на NS API-то (за елементите на заданието интерфейсите са в модула на NS Task Framework). Това решение се мотивира от съображението, че проектантът на всяко едно API за Node Service-а трябва да има възможността да дефинира неговата функционалност изцяло в него, зависейки единствено от възможностите на самия сервиз, дефинирани в интерфейса му от платформения модул. Също така, дизайнерът на API-то взима решението как избраният вътрешен протокол за връзка може да повлияе върху цялостната визия на интерфейса. Не на последно място, проектирането на NodeServiceClient-а оказва влияние и върху формата на фабричните методи за неговото създаване.

Дизайнът на интерфейсите на NS Task Framework предполага използването на технологията *JavaRMI* [21] [22] [23] (виж глава 6 за повече детайли по темата) за осъществяването на връзка с изпълнителния модул. Особеностите на тази технология спомагат за създаването на един сравнително прост механизъм за общуване, като участието на имплементиращия клас на NodeServiceClient в тази схема се изчерпва с това, че той капсулира отдалеченият обект на сервизния модул и обръщенията към клиента на сервиза се преобразуват вътрешно в *RMI* извиквания към капсулирания обект. Начинът за получаване на отдалечения интерфейс се извършва според стандартните процедури на *RMI* технологията и изискват като входни параметри за осъществяването на връзка Интернет адреса и порта на възела от грида, както и името, под което сервизът на изпълнителната подсистема на Node Service се е регистрирал на този възел. Ще отбележим, че стойностите по подразбиране на порта и името на регистрация са уточнени в един от платформените класове.

Респективно, тези входни данни определят и сигнатурата на фабричните методи за инстанциране на NodeServiceClient. В случая, от гледна точка на известните по подразбиране параметри, предлаганите входни комбинации са

две – Интернет адрес и порт или само адрес. Името на регистрация е фиксирано и не може да се променя през тези методи. По отношение на работата на фабричния клас, ще отбележим една негова особеност – той кешира вече създадени NS клиенти към дадени възли, като по този начин връзката с даден възел се осъществява само през един клиент. По този начин, от една страна, се преизползва вече създаденият комуникационен ресурс, а от друга страна, се синхронизира реда на подаване на заявките от един клиентски хост към даден възел на грида.

Ще завършим прегледа на NS API-то, като разгледаме какви са предлаганите функции от интерфейса на NodeServiceClient. Всъщност, от огледа на този интерфейс става ясно, че настоящето API използва в пълна степен възможностите на изпълнителния модул, декларирани в съответния интерфейс от платформена подсистема. В долната таблица можете да намерите съответствията между декларираните методи в двата интерфейса и кратко обяснение за предмета на тяхното действие:

NodeService	NodeServiceClient	Предназначение
uploadTask	uploadJavaTask	зареждане и еднократно изпълнение на задание
	uploadJavaPersistentTask	зареждане на персистентна услуга
	invokeJavaPersistentTask	извикване на персистентна услуга за обработка на подадени данни
gcTaskResources	gcTask	изтриване на създадени файлови ресурси при изпълнението на задание
stopJavaTask	stopJavaTask	принудително прекратяване на стартирано задание
startPersistentTask	startPersistentTask	стартиране на персистентна услуга на възела
stopPersistentTask	stopPersistentTask	спиране на персистентна услуга на възела
removePersistentTask	removePersistentTask	отстраняване на персистентна услуга от възела
-	close	затваряне на клиентската връзка към възела; след затварянето, не може да се подават повече заявки към възела

Таблица 3. Съответствие между функциите на изпълнителния модул и на неговия клиент според дефиницията му в Node Service API-то.

Отново напомняме, че интерфейсът на NodeServiceClient е предназначен за ползване от клиентите на NS API-то и за адресирането на заявки за изпълнение върху възлите на грида, докато NodeService е интерфейс за вътрешна употреба и неговото ползване трябва да става само от „вътрешността” на API-то, така че е от интерес именно за проектанти на ново Node Service API.

3.4. Изпълнителна подсистема на модула за управление на възли.

Можем да кажем, че тази подсистема представлява **ядрото** на услугата за управлението на възлите. Причина за това са следните нейни основни характеристики:

- Подсистемата е изцяло локализирана във възлите на грид системата и оползотворява предоставените хардуерни ресурси за изпълнението на заявки.
- В нея е събрана цялата изпълнителна логика за осигуряването на изчислителните услуги на грида.
- Подсистемата, бидейки във възлите на грида, заема най-ниското звено в йерархията на сервизите, които работят не само в рамките на Node Service услугата (например Node Service API-то функционира при централните сервизи като RMS за разпределяне на заданията), но и в цялата грид система въобще.

3.4.1. Участие на подсистемата в комуникационния протокол на Node Service.

Като ядро на услугата, достъпът до подсистемата е със специализиран режим, който се осъществява през модула на API-то. Трябва обаче да се подчертае, че в съответствие с концепцията за функционална автономност на изпълнителната и на клиентската части на Node Service услугата, достъпът от API-то до ядрото се осъществява през съвкупността от общи интерфейси, дефинирани в NS Task Framework, и чрез използването на *JavaRMI* технологията за осъществяването на връзка.

За реализацията на RMI комуникация, в изпълнителната част е реализиран сървърен модул, който свързва имплементацията на основния интерфейс на услугата Node Service с отдалечен обект, през който достъпът до услугите на ядрото, декларирани в интерфейса, става възможен за API-то. Поради тази причина, изпълнителната подсистема може да бъде наречена също така и сървърна, на принципа, че именно през сървър се осигуряват съответните услуги за изпълнение в рамките на вътрешния протокол с API-то. Моделът клиент-сървър е широко използван в моделите за изпълнение на услуги, като особеността му тука е употребата му в съчетание с RMI протокол за комуникация.

Чрез комбинацията от използването на *JavaRMI* и от интерфейсите на платформения модул се създава и изолационен слой, скриващ взаимно подробностите от техническите реализации на сървърната и на клиентските части от Node Service услугата. Беше наблягано неведнъж на факта, че локализирането на тези интерфейси в общия платформен модул прави съществуването на останалите два модула независимо един от друг. Нещо повече – докато за NS API-то е необходима информация за отркиването на възела, към който да адресира заявките, то за настоящата версия на ядрото аналогична информация (данни за местоположението на хоста на API-то) не е нужна, понеже не се предполага то да инициира връзка в обратната посока,

докато тя не бъде открита от другата страна. С други думи, изпълнителният модул не трябва да знае нищо и за хоста на API-то, което е негов клиент.

Все пак, механизъм за обратна връзка по време на изпълнение на задание е необходим и той е осигурен в две направления, които имат различно предназначение:

1. За осъществяването на файловия обмен между възела от грида и клиентския хост, NS Task Framework осигурява интерфейси за ползване от ядрото като отдалечен обект, резидиращ при клиентската машина, плюс един имплентиращ клас за изходен поток за запис, който се ползва и от двете страни. Участието на API-то в случая е минимално и се изчерпва с имплементация на отдалечения интерфейс за обратна връзка и с това, че при обръщенията си към ядрото подава този отдалечен обект като аргумент за обратна връзка, ако такава е нужна.
2. Изпълнителният модул също така има директен канал за връзка с клиентите на NS API-то с помощта на мониторинговите обекти, уведомяващи за хода и резултата от изпълнение на заданието. Особеностите на мониторинговият механизъм бяха вече разгледани в главата за подсистемата на NS Task Framework. Тука ще припомним, че аргументите, които се връщат при този процес на клиентите на API-то, се конструират на изпълнителните възли, докато начинът на обработка на предадената информация се определя от самите клиенти. API-то играе само посредническа роля при предаването на тези мониторингови обекти при създаването на заявката. Тъй като тези обекти служат за отдалечен достъп в смисъла на *JavaRMI,* предаването на информацията в обратната посока се осъществява директно, без намесата на API-то.

3.4.2. Модел на обслужване на пристигащите заявки.

По отношението на обслужването на заявките, пред изпълнителното ядро стоят няколко изисквания, освен тези споменати в увода:

1. То трябва да обслужва няколко типове заявки, по начина, по който те са декларирани в **NodeService** интерфейса от платформения модул.
2. Позволява се пускането на няколко заявки едновременно на един възел, като изпълнителният модул трябва да осигури условия за тяхната независима работа. Последната понятие се използва в условен смисъл, тъй като в общия случай ядрото няма вътрешна информация как са реализирани заявките.
3. Модулът трябва да бъде лесно разширяем за облсужването и на други типове заявки, които ще се определят в бъдеще.
4. Допуска се работата не само на няколко заявки, но и на няколко клиента с изпълнителния модул.
5. Логиката за предварителна и последваща обработка на заданията преди тяхното ‘физическо ‘ пускане за изпълнение трябва да бъде минимална и ефективна с цел да се осигури по-голяма производителност на възела. Разбира се, в крайна сметка времето за изпълнението на заданието в общия случай зависи предимно от самото задание.

6. Навременно уведомяване на клиентите на заданието според параметрите на подадените мониторингови обекти.

Ще обърнем внимание, че първата характеристика на практика е единствената задължителна за всички имплементации на изпълнителния модул, тъй като само тя се базира на деклариран интерфейс от платформената част. Други варианти могат например да позволяват едновременното изпълнение само на една заявка на възела, като така се гарантира че предоставените хардуерни ресурси ще бъдат разпределени изцяло за нуждите на тази заявка. Все пак, сегашният вариант е по-либерален и оставя логиката как да се ползват тези ресурси на съответните сервизи от по-високо ниво в грида.

Целта на изложението в настоящата глава е да видим как са постигнати тези изисквания.

NodeServiceImpl е главният клас в логиката на изпълнение на задания. Той предоставя имплементация на интерфейса **NodeService,** през който API-то изпраща своите заявки. Чрез сървъра, реализиран в класа **NSServer,** този клас става достъпен за отдалечени извиквания от няколко клиента (*изискване 4)* според спецификите на технологията *JavaRMI*, както стана ясно в предишната подточка.

С оглед на разнообразието от типове заявки, за което дори се очаква да се увеличава с развитието на услугата, е възприет следният подход – за всеки вид задание е предвиден ***специализиран процесорен модул*** за неговата обработка. С всеки нов вид задание в кода на ядрото се добавя нов процесор (*изискване 3*). Изискване за тези обработващи модули е да наследяват интерфейса **NSTaskProcessor**, който дефинира общите изисквания към тях. В случая това са:

- Модулите да поддържат изпълнение като нишки; по този начин се осигурява работата по заданията за изпълнение да се извършва като в отделен процес.
- Да позволяват спиране на заданието, ако това действие е релевантно.
- Да връщат информация за индентификационния низ на заданието – по него заданията се откриват, ако клиентът ги потърси на възела за последващо действие (например спиране на изпълнението им). Вътрешно, новата заявка се адресира към обработващия модул, който има директния контрол над заданието, докато то не е свършило.

По този начин в ядрото са осигурени отделни елементи на специализация по отделните видове задания. Главният клас **NodeServiceImpl,** през своите публични методи, играе ролята на ***диспечер*** на подадените задания, като тяхното разпознаване става по един от следните два начина:

- Вида на заданието се разпознава според самия метод, през който е дошъл (конкретно, това са всички методи за спиране на задание и за управлението на жизнения цикъл на персистентните услуги).
- Ако това не е достатъчно, разпознаването става с помощта на допълнителен идентификационен аргумент от типа **NodeServiceTaskType** (разгледан в главата за платформения модул) –

в случая, това важи за единствения метод за зареждане на задания за изпълнение.

Фактически, разпределянето на заданията по *специализираните процесори* става в последния метод. Механизмът на разпределение се осигурява: 1) от факта, че това е единствената входна точка в интерфейса за зареждането на задания за изпълнение; 2) от изброеният тип за всички видове задания, който се използва за тяхното разпознаване; 3) от вече споменатите специализирани модули.

Самият процес на разпределение се извършва, като се създава инстанция от съответния процесорен тип, на която се подават необходимите данни по заданието (обикновено това е един-единствен обект, капсулиращ цялата необходима входна информация, плюс отдалечен обект за обратна връзка с клиента на ядрото – API-то) и се пуска да работи в своя нишка.

С последното се осигурява многонишковост на изпълнителното ядро, което влече две предимства – 1) ядрото е готово да приеме следваща заявка за облсужване (*изискване 2*); 2) срив в нишката на изпълнение не може да засегне работата на диспечерския модул. Недостатък обаче е, че нишките стават независими от централния модул и в този вид директен контрол над тях не е възможен. Този недостатък все пак е преодолян с един допълнителен градивен елемент на ядрото – ***регистър на заданията за изпълнение.*** В началото на своята работа, всеки процесор регистрира определена информация, зависеща от типа на заданието, в предназначена за това таблица от регистъра.

Към момента, в съответствие с двата основни типа задания за обслужване - за еднократно изпълнение и персистентните услуги, таблиците също са две. За ключ в тях служи идентификационния низ на заданието. С всяко задание се регистрира и процесора, който го обслужва. Така, от този регистър, с помощта на идентификатора на заданието, диспечерът във всеки момент може да се свърже със съотвествящия му процесор. При заданията за еднократно изпълнение, допълнителна информация е фазата, до която е стигнало изпълнението при обработка.

В края на общия прегледа на централния модул, реализиран в класа **NodeServiceImpl,** става ясно, че той имплементира заявките за изпълнение, декларирани в интерфейса на изпълнителната подсистема на услугата NodeService (*изискване 1*). Удовлетворяването на изисквания 5 и 6 е предмет на вътрешна реализация в логиката на работа на специализираните обработващи процесори.

3.4.3. Други градивни елементи на изпълнителното ядро.

Дотук се запознахме с най-важните части от изпълнителната подсистема:

- Основният клас **NodeServiceImpl,** който има задачата да разпределя заявките по процесори и нишки и в по-редки случаи да изпълнява по-леките от тях сам.
- Сървърният клас **NSServer**, който се грижи функционалността на ядрото да стане достъпна за отдалечени извиквания.

- Специализираните процесорни модули за обслужването на отделните типове задания – тяхното разглеждане в детайли е предмет на следващата глава.
- Регистърът на заданията (клас **NSTaskRepository**), в който се описват кои модули в момента кои задания обслужват (разпознават се по идентифкатор). По този начин се осъществява контролът над модулите, макар и да са пуснати да работят в отделна нишка от тази на главния диспечерски модул. Поддържат се два вида записа – за заданията за еднократно изпълнение (клас **NSRepositoryRecord**) и за персистентните услуги (клас **NSTaskRepositoryRecord**).

Преди да пристъпим към детайлното разглеждане на изпълнителната логика на процесорите за обслужване на заданията за еднократно изпълнение и на персистентните услуги, ще се спрем на някои от допълнителните модули, които са помощни за тези процесори и осигуряват тяхната работа.

Механизъм за зареждане на клиентски класове в паметта

При разглеждането на интерфейсите от платформения модул, беше уточнено, че специфицирането на заданието за изпълнение става в термини, аналогични на използваните в *Java Reflection API* - клас, метод и аргументи за изпълнение. Да уточним, че със споменатото API е възможен детайлният разбор на един компилиран клас на неговите съставни елементи (методи, член-променливи и т.н.), разбор на самите елементи, динамично зареждане в паметта, конструиране на обект от класа и изпълняване на неговите методи с публичен режим на достъп. Върху тези средства се базира и работата на специализираните процесори, за които един от входните им параметри е файл или архив с компилирания код на заданието. Респективно, в настоящата подсистема е реализиран модул (клас **NodeServiceClassLoader**), който по тези файлове прави съдържащите се в тях класовете достъпни за ползване в паметта. Имплементиращият клас наследява *java.net.URLClassLoader* и използва негов метод, който зарежда класове, посочени чрез URL адресите на техните компилирани файлове. Настоящият вариант на classloader-а прави този метод наличен за ползване от всички процесори. В рамките на подсистемата е позволено да има само една инстанция на обект от този клас, за да може процесорите да имат единен и общ достъп до всички заредени до момента класове.

Конфигурационен клас на изпълнителната подсистема – NodeServiceInternal

Този клас има задачата да осигурява конфигурационна информация, която е общовалидна в рамките на цялото изпълнително ядро. Поради тази причина, инициализацията на **NodeServiceImpl** започва с инициализацията на този клас, който е длъжен да настрои определени параметри и да бъде наличен през цялото време на работа на модула, за да връща изискваната информация. На този етап, при сегашния опростен вид на изпълнителното ядро методите на класа имат задачата да:

- Да дават достъп до единствената инстанция на класа **NodeServiceClassLoader.**

- Да връщат информация за работните директории на процесорите и на персистентните услуги.

Помощни класове за мониторинг на заданията за изпълнение.

Можем да ги разделим на три вида:

А) Класове, които нотифицират отдалечените обекти, които са се регистрирали за едно от следните две събития – успешно или безуспешно приключване на изпълнението на заданието. Единственият по-интересен момент от техническата имплементация на тези класове, е че те са конструирани според принципите на visitor-шаблона в Java. Повече подробности за този шаблон са дадени в глава 6.

Б) Класове, които капсулират информация за настъпилото събитие и се връщат като аргумент на методите за отдалечено извикване, дефинирани на мониторинговите обекти. Имплементацията на тези методи е реализирана изцяло в клиентския код, който обработва получените чрез тези методи аргументи. Конкретните класове са:

- **NSTaskAbortedEventImpl –** връща информация за възникналите изключения, поради които е пропаднало изпълнението на дадено задание.
- **NSTaskResultImpl –** връща информация за резултата в желания от клиента формат. Една особеност на тази имплементация е, че е позволено тя да съдържа и да връща едновременно два типа изходни данни – обектни и под формата на файлове. Друга особеност е, че методът, който връща резултата като масив от файлови URL адреси, има грижата при своето извикване да ги копира на клиентския хост. Прието е това копиране да не става веднага след завършването на заданието, а едва при поискването на изходните файлове, за да се избегне излишният мрежови трафик, докато не стане ясно че тези данни са нужни на клиента, защото те могат да се пренебрегнат за сметка на обектните данни.
- **NullNSTaskResultImpl –** фиктивен обект, който не каспулира никакъв резултат, а просто служи като заместител за аргумента на мониторинговия метод за уведомяване за приключило задание, което не връща резултат.

В) Класът **NSTaskLifecycleNotifier** използва посоченият по-горе инструментариум за обработването на регистрираните с всяка заявка мониторингови обекти, тоест за фактическото уведомяване на клиентите за посочените два типа събития.

Освен за тези събития, за които информацията се събира изцяло от изпълнителния модул, ще припомним, че клиентите се уведомяват и за преходите в състоянията на изпълнение на заданията. Припомняме, че имплементацията на този мониторинг е реализиран в API-то, а отговорност на процесорите е навременното обръщение към тази имплементация през съответния метод на интерфейса **NSTaskRemote** (виж описанието му в точка 3.2.1.1.) при преминаването към следващ етап на изпълнение.

Помощни класове за обработка на аргументи от типа NSTaskArgument.

Функцията на тези класове е при обработката на този тип аргументи да извлекат следните данни от тях:

- За типа **SerializableArgument:**
  - Стойността на капсулирания в аргумента обект;
  - Класа на капсулирания обект;
- За типа **FileArgument:**
  - Входният файл се копира на машината на възела според подаден URL, който сочи към източника на файла. Така той става наличен за локален и директен достъп
  - Добавя се информация, че класът на реалния входен параметър е *java.io.File.*

В случая, помощните класове са два, като се различават по това, че единият от тях по указание зарежда реалния клас на входния аргумент в паметта, докато другият се интересува само от името на класа. И двата класа са създадени според visitor-шаблона с цел полиморфна обработка на инстанции от подкласовете на типа **NSTaskArgument.**

Пакет на класовете на изключения в изпълнителното ядро.

Изключенията в Java са признак за възникнали грешки по време на изпълнение в кода и подлежат на специална обработка. За улеснение на тяхната обработка, изключенията при различни ситуации се квалифицират в отделни класове, с което да се подскаже какъв е източника на регистрираната грешка.

В модула са налични следните класове на изключения:

- **NodeServiceException –** общият клас за изключения, който регистрира грешка от общ характер или просто такава, която не може да се причисли към останалите подвидове.
- **NSClassLoaderException –** за отчитането на грешка при добавянето на URL към класовете за зареждане.
- **NSFileTransferException –** при грешки по време на прехвърлянето на файлове между хостовете на възела и на клиента.
- **NSInitializationException –** за грешки, възникнали при инициализацията на определени обекти, използвани при конфигурирането и работата на персистентните услуги.
- **NSJavaPersistentTaskException –** при възникнали грешки в работата на процесора за изпълнение на персистентни услуги.
- **NSJavaTaskProcessorException –** при възникнали грешки в работата на процесора за еднократно изпълнение на задания.
- **TaskLifecycleNotificationException –** при грешка, възникнала в процеса на нотифициране на клиентите за настъпило събитие.

Други помощни класове:

- **NodeServiceUtilities –** общ клас, който обединява помощни методи с разнообразно предзначение; най-важните сред тях са: метод за откриването на изходните файлове, които са резултат от изпълнението на приключило задание, с помощта на **NSFileResultLocator** обекта, дефиниран в заявката; метод за изтриването на файл с дефинирано време на изчакване за извършване на файловата операция.
- **NSClassChecker –** този клас има специалната цел да разпознае посоченият в заявката метод за изпълнение в случаите, когато подадените списък от класове на аргументите не съответства изцяло на сигнатурата на нито един метод в посочения клас. Причините за това могат да бъде две: 1) класът на входния аргумент може да не съвпада с изискания тип, но да бъде негов подтип; 2) примитивните данни в Java във версията за JDK5.0 се преобразуват автоматично в съответстващ им обектен тип при определени обстоятелства, и обратното. На базата на тези предположения се определя кои разлики между сигнатурите са допустими и се търси методът с 'възможно най-близкото' (тоест с най-малък брой несъответствия) описание до заявената от клиента.
- **NSFileTransferUtilities –** както се подсказва от името, съдържа помощни методи за прехвърлянето на файлове от и към хоста на възела. Каналът за трансфер се осъществява с помощта на класа **NodeServiceClientCallback.**
- **NSTaskCleanerUtil –** помощен клас за изтриване на файловете, останали от работата на приключило задание – входни (само за заданията за еднократна употреба) или изходни. Забележителното за този клас е, че процесът на триене е реализиран в отделна нишка, тъй като се предполага, че файловите операции отнемат време и не трябва да се забавя работата на основния модул от изпълнителното ядро. Файловете за триене са отбелязани в записа, съответстващ на заданието в регистъра на ядрото. Триенето на файловете става само по указание от страна на клиента, за да се потвърди, че информацията за приключилото задание е била анализирана от него и вече не му е необходима. Друго предимство на отложеното премахване на входните файлове е, че заявката за изпълнението на задание не е необходимо да съдържа препратки към тях, тъй като те вече са на възела, освен ако не се налага промяна в тяхното съдържание.

3.4.4. Специализирани процесорни модули за изпълнение на задания във възлите на грид-системата.

Стигаме, вероятно, да най-същественият елемент на изпълнителното ядро, който се занимава с най-важната му задача – изпълнението на различните типове задания, подадени на възела. Инструментариумът и концепциите, на които е подчинена логиката на процесорите, беше започнато да се очертават още с разглеждането на платформената подсистема на услугата за управление на възли и обзорът им приключи с прегледа на помощните класове в изпълнителното ядро. В настоящата точка ще видим как посочените понятия и спомагателни средства влизат в действие и се вписват в логиката на работа на

процесорите. Ще избегнем да се впускаме в големи технически детайли по реализацията на класовете на процесорите, а вместо това акцент на нашето изложение ще бъде да маркираме етапите в тяхната работа и по-интересните концепции, въз основа на които те са създадени и не са били конкретизирани досега.

3.4.4.1. Процесор за еднократно изпълнение на потребителски задания – NSJavaTaskProcessor.

А) Сценарий: Изпълнение на подадено потребителски задание.

Входни параметри за процесора:

- Входни метаданни, събрани в инстанция от типа **NSJavaTask**, описващи параметрите на заданието.
- Обект от типа **NodeServiceClientCallback**, служещ за обратна връзка с клиента на изпълнителния модул.

Етапи на работа:

1. При пристигането си в процесора, клиентите на заданието са уведомени, че заданието е поставено в статус ***WAITING*** и чака своето изпълнение. От обекта на заданието се извлича идентификатора му, създава се работна директория за него и се създава запис за него, който включва текущия процесор.
2. Копират се файловете, указани в заданието. За копирането се използват услугите на **NSFileTransferUtilities.** Копираните файлове се регистрират в записа на заданието, за да са известни, когато дойде моментът да се изтрият. За част от файловете, които съдържат входни данни за програмата на заданието, няма последващи действия. Файловете с компилиран код се подават на **NodeServiceClassLoader** за зареждането на потребителските класове в паметта. Обръщаме внимание, че подадените класове трябва да покриват всички реферирани класове на главната програма, подадена за изпълнение, а не само класа на програмата. Един полезен ефект от това е например, че става възможно аргументите на метода за изпълнение да не са от стандартните за Java типове.
3. Извлича се името на класа на заданието и се инстанциира неговия тип с помощта на **NodeServiceClassLoader.**
4. Извлича се името на метода за изпълнение и списъка с имената на класовете на входните му параметри. Типовете им също се инстанциират. Позволено е този списък да е празен – в такъв случай класовете ще се открият по подадените стойности на входните аргументи.
5. Извличат се самите входни аргументи за метода. С помощта на един от помощните класове за тяхната обработка (**NSMethodArgumentVisitor**), се събират техните стойности. Ако на предишната стъпка не са определени класовете им, това става сега.
6. След като името на метода и входните му аргументи са изяснени, се проверява дали класа на заданието за изпълнение съдържа такъв клас.

Ако той не бъде открит, влизат в действие алгоритмите на **NSClassChecker.** Целта е да се открие точно един подходящ метод за изпълнение, иначе заданието се прекратява принудително.

7. Ако методът не е статичен, изпълнението му изисква обект от класа на заданието. Той се създава с помощта на конструктор на класа, който е указан чрез списък с имената и стойностите на входните му аргументи. Ако този списък е празен, се използва конструктора на класа по подразбиране; в противен случай обработката на списъка протича както е описано в точки 4 и 5. Откриването на конструктора става както в точка 6.
8. На този етап вече са ясни класа и метода на изпълнение, входните аргументи на метода и е създаден обект от класа, ако е било необходимо. Това е достатъчно за да се извика метода и да започне същинското му изпълнение. Самото изпълнение се пуска в отделна нишка от вътрешен тип **TaskExecuitonThread,** което цели процесорът да има минимален външен контрол над изпълнението, когато е нужно. Тази нишка при своето стартиране сменя статуса на заданието на ***STARTED.***
9. Посочената нишка има задачата да вземе резултата, върнат от изпълнението на метода, и да уведоми мониторинговите обекти за неговото приключване. Първо, върнатият обект се капсулира в инстанция на **NSTaskResultImpl.** След това се проверява за наличието на подаден **NSFileResultLocator.** С негова помощ и с помощта на **NodeServiceUtilities** се събират изходните файлове, които са от интерес за клиента. Тези файлове също се подават на обекта от типа **NSTaskResultImpl** и се регистрират в записа на заданието, за да са налични за последващо изтриване. Накрая статусът на заданието се сменя на ***FINISHED*** и клиентът се уведомява за неговото приключване, като достъпът до събраните резултати е възможен през отдалечения интерфейс **NSTaskResult,** който **NSTaskResultImpl** имплементира.

Като общ коментар, ще посочим две неща:

- На всеки етап, при възникнала грешка, която е фатална за изпълнението, заданието се прекратява и клиентите на изпълнителния модул се уведомяват със съобщението на изключението, което обозначава грешката, или с друго, което е дефинирано в кода.
- При дизайна на процесора е търсен баланса между минималистичност на обработващата логика (*изискване 5*) и по-широка функционалност за клиентите на изпълнителното ядро. Забелязва се например, че стойностите на някои параметри (класовете на входните аргументи, метод за изпълнение) се извличат индиректно, ако не са били указани точно или въобще за простота при конструиране на заявката.

Б) Сценарий: Преждевременно прекратяване на хода на изпълнението на задание.

Входни параметри за системата:

- Идентификатор на заданието.

- Булева променлива за синхронно или асинхронно изпълнение на спирането (тоест дали методът за спиране завършва със самото спиране или не).

Етапи на работа:

1. По посочения в заявката идентификатор на заданието, **NodeServiceImpl** намира неговия запис в регистъра и уведомява процесора му да го прекрати. Ако такъв запис не бъде открит, се връща грешка.
2. Вътре в процесора се проверява статуса на заданието:
   a. Ако то вече е приключило, нищо друго не се прави.
   b. Ако вече е в процес на изпълнение (***STARTED***), заявката за спиране се адресира към нишката за изпълнение (тип **TaskExecuitonThread**). От своя страна, нишката проверява дали класа на заданието наследява типа **StoppableApplication** (виж описанието му в платформения модул), тоест дали поддържа вътрешно операцията за преждевременно спиране. Ако го поддържа, заданието се прекратява със собствения му метод; в противен случай нишката се спира със стандартния ***Thread.stop()*** метод.
   c. Ако е в процес на изчакване (***WAITING***), се вдига специален флаг на процесора, който указва, че заданието трябва да се спре. Този флаг се проверява непосредствено преди да се смени статуса на заданието на стартиран и респективно спира изпълнението му след обработката на входните му параметри. Ако е указано синхронно изпълнение на метода за прекратяване, се влиза в цикъл на изчакване, който завършва с уведомяването за спряло задание.
3. Случаи **b** и **c** завършват с уведомяването на клиента за спряло задание. В случая **a** нищо не се прави, защото се предполага, че той вече е бил уведомен за неговото приключване.

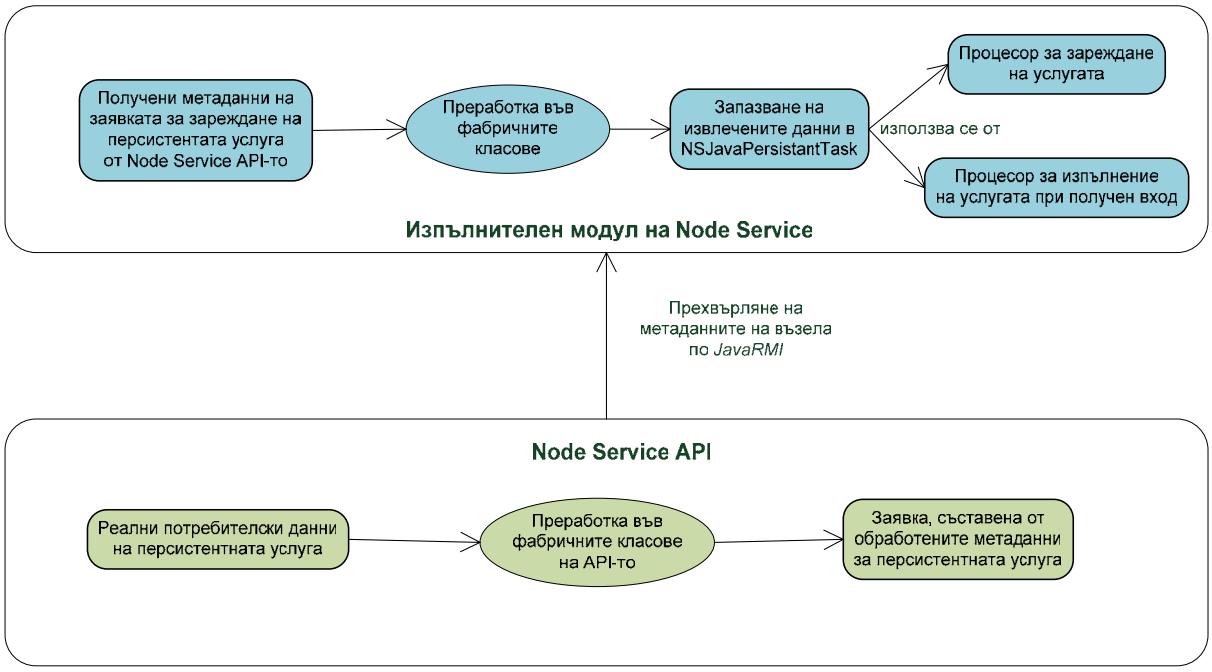
3.4.4.2. Поддръжка на персистенти услуги (сервизи) в изпълнителното ядро на Node Service.

В общи линии, за логиката на изпълнение на персистентните услуги се използват концепции и алгоритми, аналогични на посочените в предишната точка. Съществената разлика при този тип услуги произтича от тяхната „персистентост” – с други думи, от тях се изисква многократно стартиране с различни входни данни и следователно те трябва да имат постоянно присъствие на възела, докато изпълнителната подсистема е включена на него. Нещо повече, понятието за устойчивост на услугите в настоящия вариант на Node Service е подсилено с изискването тези услуги да се зареждат на възела заедно с изпълнителното ядро, докато не бъдат отстранени по заявка. Това, заедно с характеристиката им да могат да се спират и пускат по желание, говори за много по-богат жизнен цикъл от този на заявките за еднократно изпълнение, обосновано от по-дългосрочното им присъствие на възела.

В избрания вариант за настоящата версия на изпълнителното ядро, поддръжката на персистентните услуги се осъществява чрез вътрешни обекти,

които съдържат метаданните, необходими за извикването на услугите при постъпването на нови входни данни (*фигура 4*). За възстановяването на тези данни след спиране и стартиране на Node Service услугата на възела, тези обекти наследяват интерфейса *java.io.Serializable*, което означава, че могат **и** се записват във файлове, откъдето могат да бъдат възстановени при нужда. Засега вариантът на съхранение във файлове е най-удачен, тъй като се предполага, че изпълнителната подсистема на Node Service трябва да работи на възли без осигуровката на база данни (виж глава 5 за повече подробности относно системните изисквания). В допълнение на това, записването на текущото състояние на услугите се поддържа чрез предвидена за тази цел таблица в *регистъра* на изпълнителния модул.

Споменатите вътрешни обекти в известен смисъл са огледален образ на отдалечените обекти, които са елементи на заявките за изпълнение, подавани през API-то (*фигура 4*). Разликата спрямо обработката им при преходните услуги е, че в единия случай информацията от тях се извлича, за да се използва веднага, а тук се прехвърлят в друг модул от подсистема за по-късна употреба с помощта на специфични фабрични класове, които конвертират данните от единия формат в друг. На по-горно ниво в модела на поддръжка на персистентните услуги стоят процесорите за тяхното извикване, които стават „потребители" на съхранените в локалните обекти данни.



Фигура 4. Преобразувания на метаданните за зареждането на персистентна услуга в двете страни на Node Service.

Посочените нови концепции пораждат изискването от създаването на допълнителни инструменти за тяхната реализация.

А) Локални обекти за запис и устойчиво съхранение на метаданните на персистентните услуги:
NSTaskMethodPrototypeSerializable – този клас е *устойчив вариант* на **NSTaskMethodPrototype** и съхранява данни за името на метода за изпълнение и списъка с имена на класовете на входните му аргументи.

NSTaskTargetClassSerializable – аналог на **NSTaskTargetClass;** съдържа информация за името на класа за изпълнение, списък с имената на класовете на входните аргументи за конструктора, който да се ползва при създаване на обект от класа, и списък с техните стойности. Списъкът със стойности е задължителен, докато списъкът с имена на класовете не е – те ще се извлекат от стойностите. Интерес представлява методът **buildNewInstance(),** който инстанциира обект на класа по събраните входни данни.

NSJavaPersistentTaskImpl – този клас играе главната роля в модела на персистентните услуги, с помощта на останалите. От една страна, той съдържа цялата информация за работата на дадена услуга, и от друга страна, в него е вградена логиката за изпълнението и за *жизнения й цикъл* като цяло. От гледна точка на това какви данни съхранява обект от този клас за услугата, това са: ключов идентофикатор за името на услугата; списък от файлови обекти с компилиран код; данни за класа на изпълнение от типа **NSTaskTargetClassSerializable;** данни за метода на изпълнение, посочен по подразбиране, от типа **NSTaskMethodPrototypeSerializable.** При създаването на обект от този клас, необходимото описание за поддържаната персистента услуга е налице и то веднага се записва като отделен файл в специално предназначена за това директория от възела в грид-системата. Този файл съществува до отстраняването на услугата от възела. Логиката на работа на този клас ще бъде разгледана заедно със съответните процесори.

Б) Фабрични класове за създаването на локалните обекти.

Въвеждането им в кода на изпълнителната подсистема цели, от една страна, да осигури гъвкавост в изграждането на модела на персистентните услуги, а от друга страна, да се облекчи обработващата логика в конструкторите на локалните обекти.

NSJavaPersistentTaskSerializablesFactory – фабрични методи:

- **buildNSTaskMethodPrototypeSerializable –** създава обект от класа **NSTaskMethodPrototypeSerializable** по името на класа за изпълнение, където се търси метода, и отдалечен обект от класа **NSTaskMethodPrototype.** Данните от единия обект просто се извличат и се прехвърлят в другия. Особеното е, че се прави проверка за съществуването на указания метод в класа с помощта на **NSClassChecker.** Този метод обаче може да се подмени с друг от класа в рамките на една потребителска заявка за изпълнение.
- **buildNSTaskTargetClassSerializable –** за създаването на обект от типа **NSTaskTargetClassSerializable** от отдалечен обект от класа **NSTaskTargetClass;** тук се прави оценяване на подадените входни аргументи за инициализирането на обект от класа за изпълнение. Посочените параметри за инициализация също могат да се подменят в рамките на една заявка за изпълнение.

NSJavaPersistentTaskFactory – съдържа един-единствен фабричен метод за конвертирането на данните от обект от типа **NSJavaPersistentTaskUpload** в обект от типа **NSJavaPersistentTaskImpl.** Този клас има грижата за извличането на ключовия иденитификатор на услугата, за прехвърлянето на файловете с компилиран код (използва **NSFileTranferUtilities**) и за

създаването на други елементи от изходния обект с помощта на двата фабрични метода, посочени по-горе.

В) Процесор за зареждането на персистентната услуга – NSJavaPersistentTaskUploadProcessor.

Основната работа на процесора е да създаде персистентен обект с данните за услугата и да го запише във файл.

Входни параметри за зареждането:

- Входни метаданни, събрани в инстанция от типа **NSJavaPersistentTaskUpload**, описващи параметрите на заданието.
- Обект от типа **NodeServiceClientCallback**, служещ за обратна връзка с клиента на изпълнителния модул.

Етапи на работа:

1. При пристигането си в процесора, клиентите на заданието са уведомени, че заданието е поставено в статус ***WAITING*** и чака своето изпълнение. Процесорът извлича низ с ключовия идентификатор на услугата и проверява в регистъра дали сервиз с такова име вече не е регистриран. Не се допуска продължение на изпълнението при положителен резултат от проверката.
2. Статусът на процеса се сменя на ***STARTED***.
3. Създава се обект от типа **NSJavaPersistentTask** с помощта на фабричния клас **NSJavaPersistentTaskFactory.**
4. Създаденият обект се записва във файл в специално предназначена за персистентните услуги директория от възела.
5. Създаденият обект се регистрира в **NSTaskRepository** заедно с ключовия си идентификатор и съответната персистента услуга е налична за ползване при последващи заявки.
6. Потребителят на заявката е уведомен за успешното приключване на операцията по зареждане на услугата.

Г) Процесор за изпълнение на заявки към персистентна услуга – NSJavaPersistentTaskInvocationProcessor.

Основната функция на процесора е да адресира заявката към съответния обект-носител на персистентната услуга, създаден по описание по-горе алгоритъм. Съответно, по-голямата част от логиката за изпълнение лежи в този обект. Тъй като процедурата по изпълнение в много моменти се доближава до аналогичната процедура при преходните услуги, тук няма да повтаряме същите детайли, ами ще наблегнем на съществуващите разлики.

Входни параметри на заявката за изпълнение:

- Входни метаданни, събрани в инстанция от типа **NSJavaPersistentTaskInvocation**, описващи параметрите на заданието.
- Обект от типа **NodeServiceClientCallback**, служещ за обратна връзка с клиента на изпълнителния модул.

Етапи на работа:

1. При пристигането си в процесора, клиентите на заданието са уведомени, че заданието е поставено в статус ***WAITING*** и чака своето изпълнение. Процесорът извлича низ с ключовия идентификатор на услугата и намира в регистъра сервизният обект от тип **NSJavaPersistentTask**, който отговаря на това име. Изпълнението може да продължи само при наличието на такъв обект.
2. Проверява се дали сервизният обект в момента изпълнява друго задание. Не се позволява да се създават няколко инстанции от един и същи сервизен обект, които да обслужват няколко заявки. Причината е да се избегне конфликт между тях, ако ползват общи ресурси, което е много вероятно. Следователно, ако сервизът в момента е зает, се изчаква да приключи и му се подава новото задание за обработка, референция към процесора и клиента за обратна връзка.
3. Изпълнението преминава изцяло в инстанцията на **NSJavaPersistentTask.** Инстанцията вдига флаг в знак, че е в процес на работа и се извличат имената на класа и метода за изпълнение. Проверява се по ключов идентификатор дали заданието е предназначено за този сервиз; ако не – се отхвърля със съобщение за грешка.
4. Извлича се идентификатора на заявката (да не се бърка с идентификатора на сервиз, който е нещо като негов адрес, към който заявката е изпратена) и се създава запис за нея в регистъра.
5. Извличат се входните аргументи на заявката. Ако са посочени файлове за прехвърляне, те се трансферират до възела. Аргументите са обработва по познатия начин, като се извличат техните класове и стойности.
6. Проверява се дали заявката реферира метод, различен от този по подразбиране, дефиниран при зареждането на персистентната услуга. Ако се подава нов метод, той се обработва по подобен начин като при зареждането; ако ли не, се ползва вече дефинираният.
7. Ако методът не е статичен, е необходима инстанция от посочения клас за неговото изпълнение. Проверява се дали заявката задава различни параметри за конструктора, който да се използва, от вече дефинираните при зареждането. Процедира се по същия начин, както в предишната стъпка.
8. След като са подготвени метода, аргументите и инстанцията (ако е било нужно създаването ѝ) за изпълнение, както в случая при **NSJavaTaskProcessor,** тези параметри се подават на създадена нишка, която да стартира физическото изпълнение на заданието. Отново съображението е създадената нишка да служи като обект за известен контрол над хода на заданието. При започване на своята работа, нишката сменя статуса на заданието на ***STARTED*** и уведомява съответните мониторингови обекти за промяната. След успешното изпълнение на заданието, статусът му се сменя на ***FINISHED*** и резултати се събират по описания вече начин. Процесът по нищо не се различава

от описанието при **NSJavaTaskProcessor,** затова няма да се спираме на него.

Забележка: Един страничен начин на използване на **NSJavaPersistentTaskInvocationProcessor**-ите е следният - макар в общия случай заявките за извикване на персистентните услуги да пристигат от външни клиенти, няма пречка същите тези процесори да се използват и за *вътрешна употреба*. Например, възможно е в дадена конфигурация на изпълнителния модул да има указания зареждането на даден сервиз да е последвано от неговото стартиране с определени входни данни. Уточняването на детайлите по изпълнението на такъв сценарий е обект на бъдещото развитие на Node Service.

Е) Други моменти от жизнения цикъл на персистентните услуги (сервизи).

По-горе проследихме най-важните моменти от жизнения цикъл на персистентните услуги, които изпълняват тяхното основно предназначение в системата на Node Service – да се качват на даден възел и да изпълняват пристигащи заявки. Наред с тях, сервизите преминават и през други фази на своето съществуване:

Стартиране на персистентните услуги заедно с изпълнителното ядро

Както отбелязахме, персистентността на услугите трябва да се отнася и до тяхното присъствие на възела и след спиране на изпълнителната подсистема на Node Service, така че да са на разположение за повторно автоматично зареждане и обработката на заявки след ново стартиране. Една от първите задачи на изпълнителното ядро при зареждането си на възела е да прочете от предназначената за това директория файловете с метаданните на персистентните услуги, които изброихме при описанието на класа **NSJavaPersistentTaskImpl** - ключов идентофикатор за името на услугата; списък от файлови обекти с компилиран код; данни за класа и метода на изпълнение. Следващата стъпка е регистрацията на сервизите по техните ключови идентификатори, така че клиентите да могат да адресират заявки към тях.

Спиране на заявка, обслужвана от персистентна услуга

Параметрите за спирането на заявката са същите като при преходните услуги – идентификатор на заданието и булев флаг за синхронно спиране (тоест дали клиентът е блокиран до края на самото спиране или не). Алгоритъмът също е подобен с някои особености в началните етапи:

- Заявката за спиране се адресира към процесора от тип **NSJavaPersistentTaskInvocationProcessor**, който е бил регистриран за обслужването на заданието, указано с идентификатор.
- Процесорът, от своя страна, адресира заявката към инстанцията на сервиза, която е поела заданието. Тъй като обаче инстанцията за дадена персистентна услуга е една-единствена, процесорът проверява дали той обслужва все още същото задание, за да се избегне спирането на погрешно задание.
- Оттам насетне процесът по спирането на заданието, изпълнявано от сервиза, се прехвърля в класа на **NSJavaPersistentTaskImpl.**

Реализираният тук алгоритъм не се различава съществено от варианта за преходните услуги, тъй като използваните структури са подобни.

Ще отбележим, че не се поддържа спиране на процеса на зареждане на персистентна услуга на възела.

Спиране на персистентната услуга

Спирането може да стане по искане на клиента след посочването на ключа на услугата или при спирането на самото изпълнително ядро. Аргументите са:

- Ключов идентификатор на услугата.
- Булев аргумент за синхронно спиране.
- Булев аргумент за форсиране – ако е вдигнат, сервизът се спира заедно с изпълняваното текущо от него задание, ако има такова; ако е свален, спирането става след изчакване заданието да приключи.

Процедурата започва в главния модул на изпълнителната подистема (**NodeServiceImpl**), където се открива сервиза по ключа му в регистъра. След това се прави обръщение към него за неговото спиране.В зависимост от това дали сервизът е зает с изпълнението на заданието и в зависимост от флага за форсиране, се изчаква или не персистентната услуга да се освободи и след това му се настройва флаг за спряло състояние. Докато този флаг е включен, инстанцията на услугата не приема нови заявки за обслужване. Тъй като инстанцията за дадена услуга е една-единствена, от съображения за производителност, тя остава заредена в паметта и записана в регистъра на изпълнителното ядро, за да може после да бъде стартирана по-бързо, вместо да се зарежда отново от файл.

Стартиране на персистента услуга по време на работа на изпълнителното ядро

Заявката се адресира към главния модул на ядрото с указване на ключовия индентификатор на сервиза за спиране. Първо се проверява дали този сервиз присъства в регистъра и ако да и е спрян, се сваля флагът му за спряло състояние, т.е. той вече може да приема нови заявки. Ако не бъде открит, се търси неговият файл в специална директория (името на файла се образува от уникалния му идентификатор и определено разширение) и се зарежда от него, ако бъде открит. В противен случай се хвърля съобщение за грешка.

Отстраняване на персистента услуга от възела

Входните параметри напълно съвпадат по значение с тези, които се подават за спиране на услугата. Започва се с едно обикновено спиране на услугата (процедурата е разгледана по-горе), след което записът за наличието на тази услуга се отстранява от регистъра на изпълнителното ядро заедно с файла на сервиза.

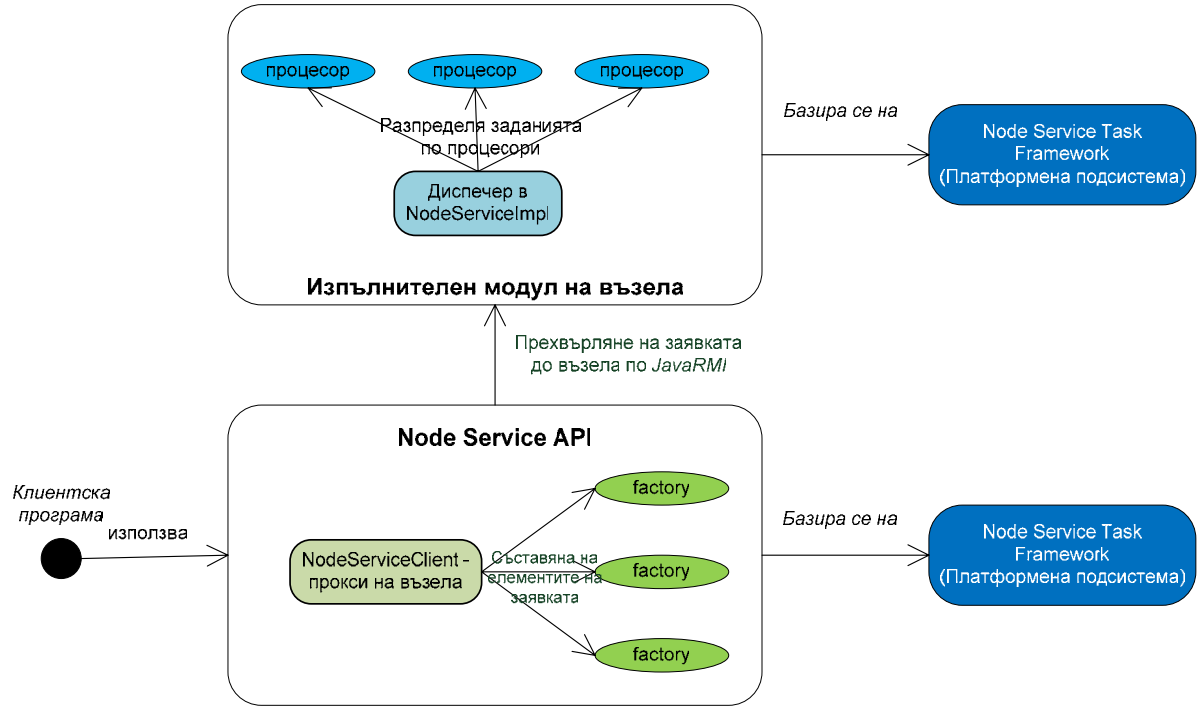
4. Типични сценарии за употреба на услугата за управление на възли (Node Service)

Още в увода очертахме кръга от задачи, които услугата е предназначена да изпълнява. В глава 3 описахме детайлно какви са концепциите, довели да

настоящата имплементация на услугата, и какви са вътрешните механизми, въз основа на които тя функционира. Цел на настоящата глава ще бъде не да изредим всички възможни сценарии за нейната употреба (както отбелязахме, възможностите й са вече описани в увода), ами да разгледаме Node Service в един по-широк мащаб – какви са етапите на работа с него и как се обработват заявките към него при двата основни сценария за неговата употреба – като самостоятелен продукт и като модул от грид системата GrOSD.

4.1. Самостоятелна форма на употреба.

Както беше отбелязано още в увода, услугата е написана по начин, по който тя да може да се използва за самостоятелни цели и да се интегрира по възможно най-лесен начин с други системи, които се интересуват от нейните възможности, благодарение на осигурените API и платформен модул. Тези елементи спестяват написването на специален интерфейс за комуникация, създаването и адресирането на заявки към ядрото.



Фигура 5. Опростена схема на действие на Node Service в самостоятелен режим

Във възможно най-простия си вариант на употреба (*фигура 5*), единствен допълнителен компонент в сценария освен модулите на услугата се явява клиентът на API-то, който най-често се намира на хост различен от този на изпълнителната подсистема. Клиентът в общия случай представлява програма на Java (за пример виж Приложение А), която с помощта на осигурените от API-то инструменти (фабрични класове) създава клиентска връзка към отдалечения хост на изпълнителното ядро, съставя своята заявка за изпълнението на еднократно задание или за зареждането или изпълнението на персистентна услуга, предоставени под формата на предварително компилиран код, и я адресира през отворения Node Service клиент. Важна част от клиента представляват дефинираните от него мониторингови обекти, които задължително трябва да наследяват адаптерните класове

NSTaskLifecycleAbstractListener и **NSTaskLifecycleAbstractListener.** След пускането на заявката за изпълнение, това са единствените елементи от клиентския код, които са отговорни за уведомяването за статуса на заданието и за обработката на крайния резултат. Подчертахме, че изпълнението на основните типове задания е предвидено да става асинхронно с работата на клиентския код, за да не бъде блокиран клиентът, докато чака изхода от изпълнението, което може да отнеме значително време в зависимост от задачата.

Основните стъпки от процеса са :

1. Клиентската програма в кода си създава с помощта на **NSAPIFactory** Node Service клиент към определена машина, на която работи изпълнителния модул и която се специфицира по IP адрес. Създаденият клиент се явява логическата връзка с изпълнителното ядро и негово прокси, през което се зареждат задачите за изпълнение.
2. С помощта на фабричните класове от Node Service API-то се конструират елементите от заявката и се сглобява самото задание. При генерирането си заданието получава статус ***UNSCHEDULED.***
3. Важен момент в края е да добавят мониторинговите обекти, защото това е единственият начин да се разбере как върви хода на заданието и да се получи резултатът от него.
4. Заданието се изпраща през отворената клиентска конекция към машината на изпълнителния модул. Непосредствено преди самото изпращане, API-то променя статуса на заданието на ***SCHEDULED*** и определя идентификатор на типа му, за да се обработи по съответния начин***.***
5. При пристигането на заявката в изпълнителния модул от Node Service, по идетификатора на типа му се разпознава за какъв процесор заданието е предназначено (или го отхвърля, ако идентификаторът му е непознат), създава инстанция на този процесор, променя статуса на заданието на ***WAITING*** и го подава на процесора.
6. В зависимост от логиката на работа на процесора, заданието може да се постави в състояние на изчакване (например ако сервизът на една персистентна услуга е зает с друга задача и трябва да се изчака да се освободи) или да се премине към нейната предварителна обработка веднага. Тази обработка (виж за повече детайли главата с описанието на процесорите в изпълнителния модул) най-често се явява извличане и оценка на метаданните на заявката (клас, метод и входни аргументи на изпълнението) и съставянето на извикващата инструкция към метода за изпълнение.
7. След обработката на елементите от заявката следва самото извикване на метода за изпълнение (статусът на заданието се сменя на ***STARTED***) и последващо събиране на резултатите от него (статусът на заданието се сменя на ***FINISHED***). Резултатите се изпращат с отдалечено извикване към регистрираните мониторингови обекти на заданието. *Този етап не е валиден за заявките за зареждането на персистентни услуги, където няма метод с входни параметри за извикване.*

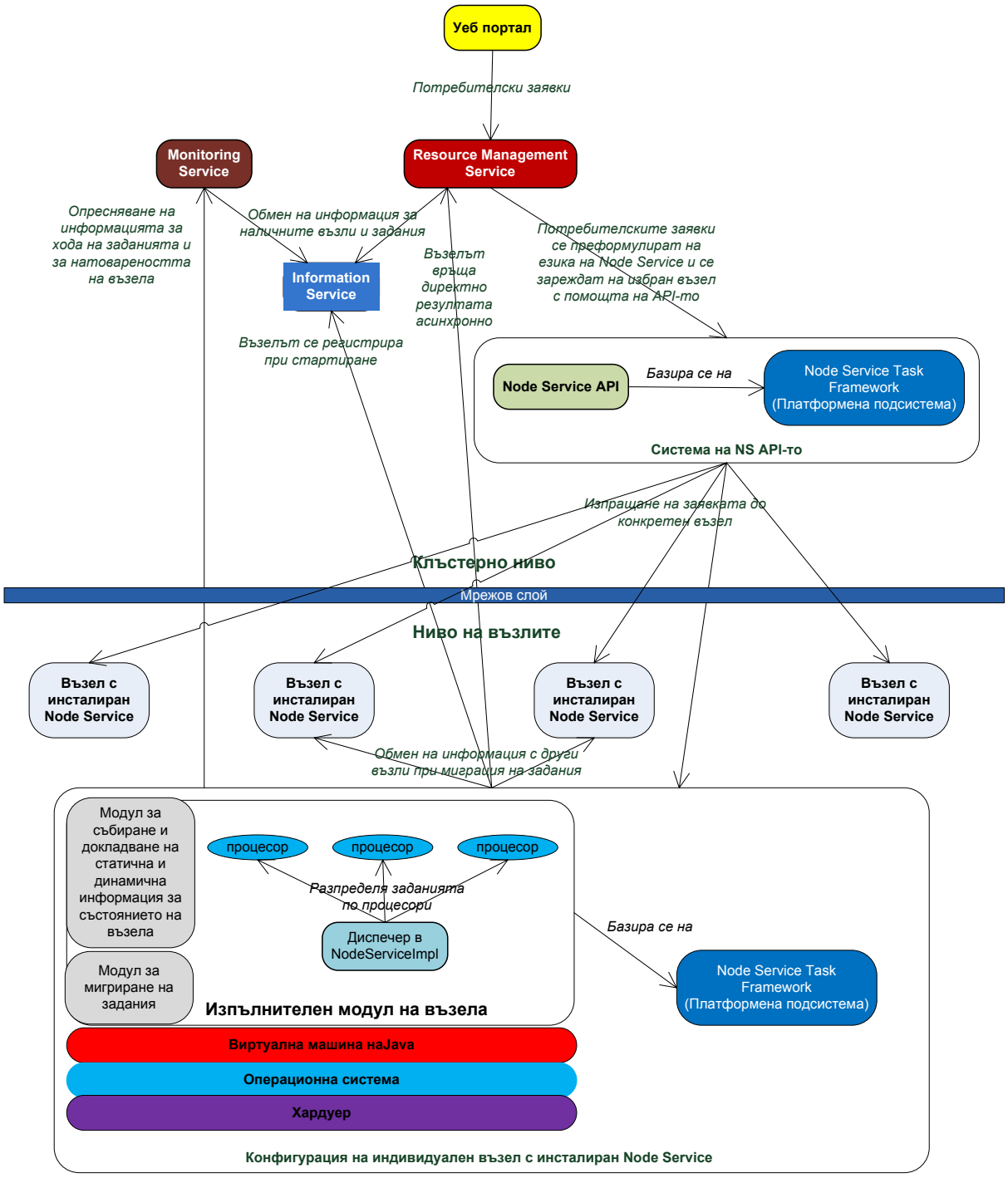
8. Клиентската програма обработва по дефиниран от него начин получените данни (дефинцията е в имплементацията на мониторинговите обекти). Нормалният завършек на процеса е програмата да изпрати на изпълнителния модул заявка да се зачистят работните файлове на приключилото задание, а отворената клиентска конекция да се използва за други заявки (тя може да се използва веднага след пращането на първата заявка за изпълнение, докато се чака за резултата) или да се затвори.

4.2. Интеграция на Node Service в системата на GrOSD.

В този сценарий, по отношение на стъпките на изпълнение особени разлики няма, като API-то и изпълнителното ядро запазват своите функции. Разликите се състоят в:

- Конкретизиране на специфичните клиенти на API-то (няма директни обръщения към ядрото) – например, услугата на RMS за управления на ресурси е отговорна за генерирането на заявките към Node Service (NS); услугата за наблюдение и контрол (MS) пък научава за промените в състоянието на заданията.
- Въвеждането на допълнителни модули на NS, специфични *само* за GrOSD:
    - с информационни функции – дава сведения на информационната услуга (IS) за статичните хардуерни характеристики на възела, с което да се следи натовареността му; междувременно периодично уведомява мониторинговата услуга (MS) за наличността на услугата на възела и промените в натовареността на възела (свободна памет и т.н.).
    - с цел мигрирането на задания при необходимост – в този случай NS по заявка от RMS има грижата да препрати задание с даден идентификатор на посочения в заявката възел.

Допълнителните модули са все още в етап на проектиране и са част от бъдещото разширяване на функционалността на Node Service за нуждите на GrOSD.



Фигура 6. Основни взаимодействия и взаимовръзки на услугата за управление на възли (Node Service) с други сервизи на GrOSD

Основни взаимодействия и взаимовръзки на услугата за управление на възли (Node Service) с други сервизи на GrOSD (*фигура 6*)

- При своето стартиране, информационният модул на Node Service, работещ на съответния възел, изпраща данни с хардуерните параметри на възела до информационната услуга (IS); по този начин се узнава, че този възел е наличен в клъстера и може да се направят някои преценки какви натоварвания може да поеме (респективно, за задачи с каква сложност е подходящ). Последваща реакция е да се отвори клиент през

модула за управление на ресурси (RMS) и да се регистрира мониторингов обект, подаден от MS, за следене на състоянието на възела.

- Периодично модулът на Node Service, работещ на възела, нотифицира регистрираните за това мониторингови обекти за наличността си в клъстера, като заедно с това може да изпраща информация за текущата си натовареност.
- Услугата на RMS е тази, която е отговорна за кеширането и съхранението на клиентска конекция към всеки възел, на който работи NS. Необходимо е в клъстера да се пази само една такава връзка към даден възел от гледна точка на това централните услуги да не са натоварени с поддръжката на много конекции към многото възли в клъстера. Регистрацията на другите модули в грида за получаването на информация от възлите става или със запитване към RMS или IS, ако те разполагат с такава информация, или чрез регистрирането на съответни мониторингови обекти, което отново се прави от RMS – предимството на този подход е, че обратната връзка от NS към заинтересованите услуга е директна и не минава през междинни обекти
- За изпълнението на задания RMS играе ролята на директен клиент, който съставя заявките и разпределя заданията по възли по свой алгоритъм. RMS има грижата да се регистрира за резултата и да го обработи според спецификацията си, а MS се регистрира за промените в хода на изпълнение на заданията. Детайлите по фазите на изпълнение не се различават съществено от описанието на стъпките в предишната точка и затова няма да се спираме повторно на тази логика. Специфичното в случая се крие в работата и функциите на клиентите на Node Service API-то – RMS и MS.
- Допълнителен сценарий на употребата на NS е миграцията на задания. Заявките отново идват от RMS, а за прехвърлянето на самото задание е необходимо да се отвори връзка между миграционните модули на Node Service на съответните възли, тоест имаме общуване на най-ниско ниво в грида. Освен това, за успешната миграция се изисква заданията да са от специфичен тип, който да позволява запазване на тяхното текущо състояние преди прехвърлянето им и възстановяването им от него след пристигането им на следващия възел. Концепцията е много близка до тази на интерфейса **Checkpointable** в системата **JGrid.**
- Всяка централна услуга в GrOSD, която желае да работи с възлите, трябва да има достъп до класовете на Node Service API-то и до интерфейсите, декларирани в платформената подсистема на NS. Както подчертахме обаче, в рамките на клъстера до един възел трябва да се пази само една клиентска връзка, създадена от RMS. Друга особеност е, че интерфейсите на Node Service платформата са налични както на хостовете на централните услуги, така и на самите възли, като единствените общи класове и за API-то, и за изпълнителното ядро. Това изисква използването на тяхна унифицирана версия навсякъде в клъстера, за да се избегнат несъответствия и грешки в комуникацията.

5. Инсталиране и конфигуриране на Node Service.

Няма съществени разлики в конфигурирането на услугата, независимо от това дали се използва самостоятелно или в интегриран с GrOSD режим. Особеното е, че и в двата случая класовете на услугата са разпределни между два типа хостове – хостовете-приемници, които са натоварени с изпълнението на заданията, и клиентските хостове, които подават заявките (по принцип, няма пречка един хост да е и от двата типа едновременно).

Node Service е разпределен в три пакета:

- *NodeService.jar* – съдържа компилирания код на изпълнителното ядро на услугата и се поставя заедно с придружаващите го скриптове в произволна директория на машината за изпълнение.
- *NodeServiceAPI.jar* – пакетът на API-то се поставя на място, избрано от клиента, така че да бъде достъпно в class-пътя на неговата програма.
- *NodeServiceIF.jar* – този пакет съдържа общата съвкупност от интерфейси и, така да се каже, дефинира *общият език*, на който си *говорят* двата модула. Поради тази причина, той трябва да присъства в същите директории, където се намират двата пакета, т.е. този архив е единственият от изброените три, който работи и на двата типа хоста.

Разбира се, за коректната работа на Node Service се изисква в една система да се използват пакети от една и съща версия на услугата.

Node Service е проектиран да работи, използвайки възможностите на платформата Java, версия 5.0, правейки го преносим между операционните системи, които тази платформа поддържа. За работата си на максимално широк кръг от възли, Node Service не разчита на помощта нито на приложен сървър, нито на база данни. Като единствено изискване остава инсталация на JDK, версия 5.0 или по-висока.

Стартирането на Node Service на възела на изпълнение става от скрипт, доставен с архивите, който може да е различен, заради особеностите при писането на скриптове за Windows и за Linux/UNIX. В общи линии, работата на скрипта е да засече инсталацията на JDK на машината, да стартира rmiregistry – стандартната програма за регистъра на RMI услуги и накрая да стартира изпълнимият клас **NSServer,** който зарежда ядрото на услугата. След този момент хостът е готов да приема и обслужва заявки. По подразбиране, услугата чака за заявки на порт *2007.*

На този етап, няма външни начини за конфигурирането на изпълнителния модул, който вместо това си има вградени настройки (например за претърсване на определни директории за файлове на персистентите услуги и зареждането на услугите от тях). За в бъдеще се планира възможностите за конфигурация да са по-големи, като механизмът ще е стандартен – всеки път при стартирането на изпълнителния модул настройките ще се четат като атрибути

от файлове, които потребителят може да променя с обикновен текстов редактор.

Настройките на клиентите на услугите на Node Service се изразяват само в това, че техните програми, които се възползват от тях, трябва да са написани на Java и class-пътищата им да съдържат местоположението на двата пакета *NodeServiceAPI.jar* и *NodeServiceIF.jar,* за да могат да ползват класовете, събрани в тях.

6. Технологии, използвани при проектирането и имплементацията на Node Service.

Езикът Java

Едно от най-важните изисквания към услугата за сигурност (и към цялата грид-система GrOSD) е платформената независимост. В грид трябва да могат да се включат най-разнообразни хостове, както по отношение на хардуерната архитектура, така и по отношение на софтуерната платформа, работеща на тях. Това изисква максимална преносимост на грид мидълуеъра – той трябва да работи на колкото е възможно повече хардуерни и софтуерни платформи. Поради тази причина за реализация на системата GrOSD, и съответно на системната услуга за сигурност, беше избран програмният език Java. По този начин системата може да работи на всяка платформа, за която има създадена Java виртуална машина, без да се правят промени в кода. Тъй като Java е широко разпространена и широкоподдържана, тя е логичният избор за такъв тип приложения.

Платформата Java

Освен преносимостта, Java разполага и с редица други предимства. На първо място Java не е само език за програмиране, а е цялостна платформа, представляваща набор от технологии, които предлагат разнообразни услуги, като се започне от средства за обработка на изображения и звук и се стигне до мрежови комуникации и сигурност. Голямата разпространеност на платформата е причина и за много налични библиотеки на трети страни, които не са включени в платформата, но могат да се използват. Не на последно място се нарежда и фактът, че Java платформата е безплатна. Тъй като GrOSD е академичен проект, едно от важните изисквания е той да бъде изграден с лицензно свободни езици и средства.

Технология на Java RMI (Remote Method Invocation) [21] [22] [23]

Технологията на JavaRMI лежи в основата на изграждане на комукационния модел, реализиран в настоящия проект. Тя дава възможност на програмиста да създава разпределени приложения на Java, при които методите на отдалечените обекти могат да се извикват от друга виртуална машина, която най-вероятно се намира на друг хост. RMI използва сериализация на обектите за пакетирането и разпакетирането (*marshalling and unmarshalling*) на

аргументите при прехвърлянето им през мрежата, с пълна поддръжка на обектно-ориентиран полиморфизъм на типовете в Java. [21]

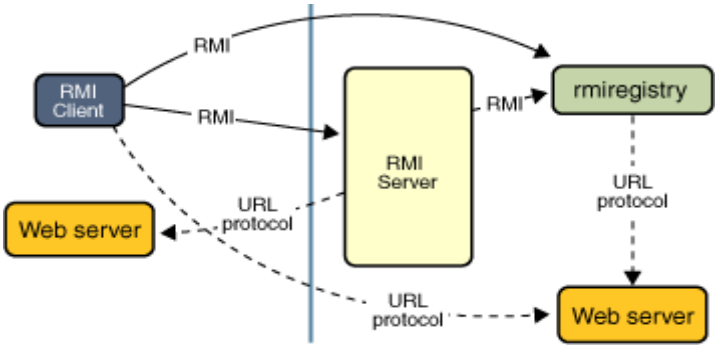
RMI предоставя прост и ясен модел за разпределени изчисления върху обекти на Java. На най-ниско ниво, технологията може да се разглежда като вариант на RPC (Remote Procedure Call – Отдалечено извикване на процедури), реализиран на Java платформа. RMI обаче притежава редица предимства пред традиционните RPC системи благодарение на обектно-ориентирания подход в Java. [22]

Основното предимство на технологията е възможността да се борави с произволни обекти и типове на входно-изходните аргументи на методите за отдалечено извикване. При откриването на клас, който е непознат за отдалечената виртуална машина, дефиницията му автоматично се сваля от отсрещната машина, която е подала обекта. RMI предава обектите по техните актуални класове, запазвайки оригиналното им поведение при прехвърлянето им при отдалечения хост. Последната характеристика обосновава друго предимство на технологията – *мобилност на кода.*

От гледна точка на производителност, RMI притежава следните черти:

- Разпределено освобождаване на паметта (distributed garbage collection) – отдалечените сървърни обекти се изчистват, щом се установи, че няма повече клиенти в мрежата, които да ги реферират.
- Паралелни изчисления – RMI е многонишкова технология и се възползва от тази характеристика за едновременното обслужване на няколко клиентски заявки едновременно.

Програмно приложение на RMI най-често се състои от две отделни части – сървърна и клиентска. [23] Сървърната програма създава няколко отдалечени обекта, създава референции за отдалечен достъп към тях и чака за клиенти, които да извикат техните методи от друг хост. Клиентът, от своя страна, се сдобива с отдалечена референция към един или повече обекта, създадени от сървъра и извиква техни методи. RMI осигурява механизъма, по който клиентът и сървърът си комуникират и обменят информация. Детайлите по вътрешната комуникация между отдалечените обекти остават скрити за програмиста – за него извикванията на методите не се различава от извикването им при всеки друг обичаен сценарий.



Фигура 7. Схема на комуникация в RMI [23]

На горната диаграма се вижда как разпределеното приложение използва регистъра на RMI от сървърния хост за получаването на референция към

отдалечения обект. За тази цел, преди това сървърът се обърнал към регистъра, за да свърже този обект с определено уникално за хоста име. Клиентът търси обекта именно под това име и след това може да извиква негови методи. На диаграмата е изобразен уеб сървър, от който се свалят дефинициите на класовете при необходимост и в двете посоки – от клиента към RMI сървъра и обратното.

Java Reflection API [24]

С помощта на това API се получава вътрешен достъп до класовете, заредени в Java виртуалната машина (JVM) и позволява да се пише код, който работи с класове, избрани по време на изпълнение, а не вградени в сорс-кода на приложението. Тази възможност превръща технологията в мощен инструмент за създаването на гъвкави приложения.

Използването на технологията Java reflection се различава от нормалното програмиране на Java по отношение на това, че се работи с *метаданни* – данни, които описват други данни. В конкретния случай, става въпрос за описанието на класове и обекти, заредени във виртуалната машина. Възможностите на API-то са огромни – то позволява изграждането на гъвкав код, който се асемблира по време на изпълнение, без да се изискват връзки между компонентите, вградени в текста на техните програми. Някои от възможностите са [25]:

- Получаване на детайлно описание и метаданни за даден клас под формата на обект от тип *java.lang.Class.*
- Разучаване на използваните модификатори и типове в класа.
- Достъп до член-данните на класа – конструктори, полета, методи и вградени класове и възможност за тяхната самостоятелна обработка:
    - За член-променливите на класа – получаване на детайлна информация за модификаторите, получаване и промяна на стойността им;
    - За методите - получаване на детайлна информация за модификаторите и тяхното извикване;
    - За конструкторите – откриването им и създаването с тяхна помощ на обекти от класа.
- Работа с масиви – идентифицирането на типа на масива, създаването на нови масиви по време на изпълнение и настройка на стойностите в тях.
- Работа с изброени типове (*enums*) – изучаване на константите, събрани в типа, както и на неговите член-променливи, методи и конструктори, ако има дефинирани такива; получаване и промяна на стойностите в полетата на типа.

За съжаление, мощта на тази технология си има и своята цена:

- Ниска производителност – тъй като работата на API-то е свързано с динамичното откриване на типовете, някои определени оптимизации във виртуалната машина не са приложими. Операциите с това API се

извършват по-бавно от случаите, когато има възможност да се извършат по стандартния начин.

- Съображения за сигурността – API-то се нуждае от определени права, които може да не му се предоставят в определен контекст.
- Разкритие на вътрешни детайли – с API-то в някои случаи е възможно да се извършват определени операции (но не и извикване на методи) върху данни с частен модификатор за достъп. Така се разбива концепцията за капсулация на кода.

В крайна сметка, трябва да се отбележи, че за целите на настоящия проект, използването на Java Reflection API беше избран като най-удачния вариант, тъй като в общия случай изпълнението на задания се извършва върху изцяло непознат потребителски код и са необходими средства за неговото разучаване и обработка. API-то е естественият избор за задачи, свързани с динамичното конструиране на методи и тяхното извикване за изпълнение.

Шаблони на дизайн в Java (Java Design Patterns) [26]

Шаблоните на дизайн могат да се разгледат като възникнали в резултат за многократното използване на определени откъси от обектно-ориентиран код в различни проекти, които са се оказали особено ефективни за решаването на специфични проблеми. Идеята зад създаването на шаблоните е проста – написването и каталогизирането на добре познати *взаимодействия между обектите*, които програмистите са сметнали за полезни. Най-общо казано, шаблоните описват как обектите *комуникират* помежду си, без да навлизат в подробности относно имплементираните от тях модели за данни и методи. Поддръжката на такъв тип изолация между обектите винаги е била една от целите на доброто обектно-ориентирано програмиране.

Следват няколко по-формални дефинции за това какво представляват дизайнерските шаблони:

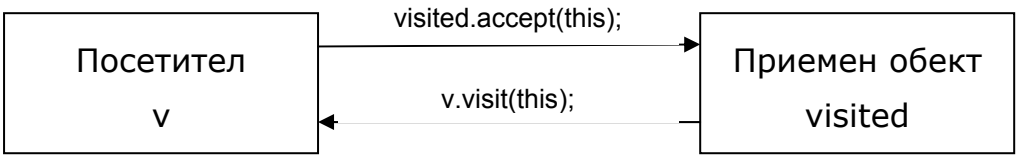
- *„Дизайнерските шаблони представляват набор от правила за постигането на определени цели в областт на софтуерните разработки”* [27]
- *„Шаблонът се отнася до често повтарящ се проблем на дизайна, който възниква в определени ситуации, и дава решение за него.”* [28]
- *„Шаблоните идентифицират и специфицират абстракции, които надхвърлят нивото на отделни класове и инстанции, или на компоненти.”* [29]

Трябва да подчертаем, че дизайнерските шаблони се отнасят не само до дизайна на обекти, а също и до *комуникацията* между обектите. Простите и елегантни методи за комуникация правят шаблоните ефективни. Дизайнерски шаблони съществуват на много нива – от специализирани решения на най-ниско ниво до такива, адресиращи обобщени системни проблеми. В настоящия проект са приложени няколко такива решения, по-съществените от които са:

- **Шаблонът на фабрични класове –** при този шаблон даден клас е натоварен с функцията да връща инстанция на един от няколко възможни подкласа на един абстрактен основен клас, в зависимост от подадените входни данни. В проекта на Node Service този шаблон се употребява особено много в дизайна на API-то и част от изпълнителния

модул в леко модифициран вариант, целящ най-вече да скрие използваната имплементация, която стои зад върнатия интерфейс.

- **Шаблонът на „наблюдателите" –** на този шаблон се базира асинхронното уведомяване на клиентите на Node Service за хода на изпълнение на заданията и за върнатия резултат. В случая, *мониторинговите обекти* се явяват в качеството си на *наблюдатели* на процеса. При него имаме два типа обекти, условно наречене *Субект (Subject)* и *Наблюдател (Observer).* Субектът държи определени данни, които са от интерес за Наблюдателя, и предоставя чрез интерфейса си метод, чрез който Наблюдателят регистрира своя интерес, като подава референция към себе си. Оттам насетне, всяко важно събитие или промяна (кое е важно се определя от имплементацията на Субекта) се съобщава на Наблюдателя чрез извикването на определен метод от подадения интерфейс. Субектът не се интересува как Наблюдателят ще обработи изпратеното събитие.
- **Шаблонът на проксито („представителя"/"заместителя") –** този шаблон влиза в употреба, когато един сложен обект е по-добре да се представи чрез по-прост. Нуждата от такова заместване може да се дължи на най-разнообразни причини – 1) ако създаването на сложния обект отнема много време и ресурси, този процес може да се отложи до момента на обръщение към заместителя; 2) ако истинският обект се намира на отдалечена машина и достъпването му се забавя, защото се прави през мрежа; 3) когато истинският обект има ограничени права, представителят може да ги провери за даден потребител. В общия случай, в интерфейса на представителя се срещат същите или подобни методи като тези при истинския обект, и извикването им обикновено води след известна обработка до обръщение към реалния адресат. В настоящия прокет, обектът на **NodeServiceClient** играе ролята на представител на **NodeService**, който работи на друг хост и скрива детайлите по оформянето и изпращането на заявките по мрежата.
- **Шаблонът на „посетителя" –** този шаблон е особен с това, че обръща някои от концепциите на обектно-ориентираното програмиране, като изнася обработката на вътрешни данни на даден клас извън тях. Този подход е полезен, ако имаме известен брой инстанции на няколко класа и искаме да извършим определена (еднотипна) операция с повечето от тях или всички. За да може един външен клас *(посетителят)* да достъпи вътрешните данни на друг клас, единственият начин е да се обърне към публичните му методи. *Посещаването* на всеки клас става с помощта на предварително създаден в тях метод – *accept,* който взима за аргумент референция към *посетителя.* В него се прави обръщение към метода от интерфейса на *посетителя,* подавайки референция към себе си, тоест към *приемния обект.* По този начин *посетителят* вече има достъп до публичните методи на *приемника* и може да ги използва за получаването на данни и извършването на изчисления. (За повече яснота, виж фигурата по-долу.) В изпълнителния модул на Node Service шаблонът се използва за последователната обработка на *мониторинговите обекти*, които са от различни подтипове на **NSTaskListener,** и за обработката на входните аргументи на метода за изпълнение, отново по същите причини.



Фигура 8. Взаимодействие между посетителя и приемника в шаблона Visitor [26]

Инструменти за работа

Основно изискване към инструментите, използвани за реализацията на GrOSD, е те да са с лиценз за свободно ползване. Една от най-добрите свободни развойни среди за Java е платформата на IBM Eclipse [30]. Тя е разширяема и поддържа много добавки (plug-ins) на трети страни. Поради тези причини тя беше избрана за реализация на услугата.

Използването на други иснтрументи не беше наложително поради минималните изисквания за реализацията на Node Service, която не разчита на използването нито на bean-технологии и Java EE (Enterprise Edition) платформа (ползва се Standard Edition), нито на база данни или на специален приложен сървър (application server), инсталирани на възлите в грида.

7. Разширяемост на проекта и перспективи за бъдещо развитие.

В проекта на Node Service стоят много предпоставки за неговото допълване и изменение.

На първо място, проектът дефинира един общ пакет от интерфейси, които определят общият език, на който общуват API-то и изпълнителното ядро, изолирайки ги взаимно от реалните имплементации. Ограниченията, при която всеки от останалите два пакета може изцяло да се подмени и услугата да запази функционалността си, са минимални и позволяват лесни промени в кода.

На ниво API, фабричните класове за съставянето на елементите от заявката позволяват прозрачна за клиента промяна на вътрешните класове в модула, без това да води до промени в клиентския код.

По отношение на вътрешния модел на изпълнение в ядрото на Node Service, въвеждането на специализирани процесори за обслужване на определени типове заявки и работата на централния клас в качеството му на *диспечер* на заявките улеснява въвеждането на нови типове задания за услугата, които ще се поемат от съответните нови видове процесори.

Не на последно място, за гъвкавостта на услугата допринася и факта, че клиентите на Node Service API-то допълват процеса на обработка на заявките със свои имплементации на някои интерфейси от платформата за дефиниране на задания, чрез които определят критериите за извличане на резултата от изпълнение и неговата последваща обработка.

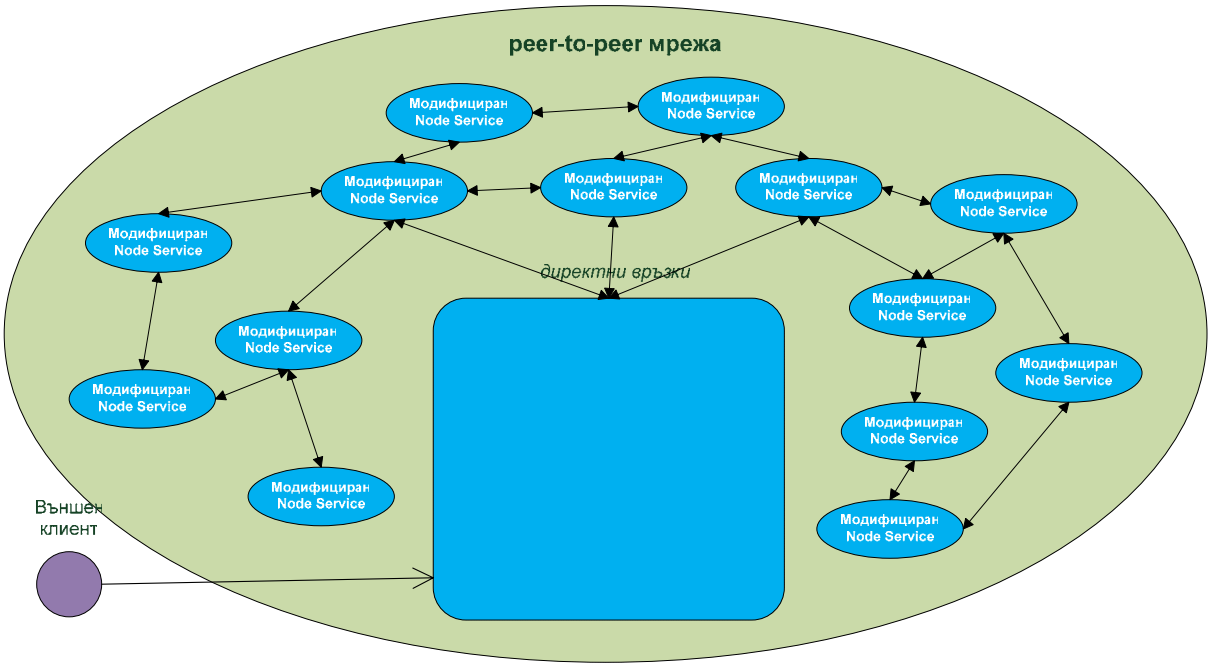
Всички тези предпоставки дават широко поле за развитие на услугата Node Service. Тук ще дадем примери в две направления:

А) най-належащите за нуждите на проекта GrOSD:

- Оптимизация на сегашните процеси в изпълнителното ядро;
- Доразвитие на сегашната имлементация:
  - въвеждането на систематизирано управление на нишките в ядрото, лимит на създаваните нишки;
  - логинг инфраструкура за отчитане на дейността и възникналите грешки;
  - подобрения в механизъма за зареждане на класове и отключването на компилираните файлове, когато не са нужни;
  - подобрено управление на създаваните от заданията файлове и тяхното изчистване от възела;
  - персистентен модел на регистъра със задания на изпълнителното ядро;
  - същинска процедура по „гладко” спиране на ядрото, евентуално с възможности за запазване на състоянието на изпълняваните задачи и тяхното възобновяване при стартиране;
- Въвеждане на допълнителни модули в ядрото, които поддържат изпълнението на програми, написани не на Java, а на C/C++.
- Въвеждане на допълнителни модули в ядрото за информационните нужди на системата GrOSD – събиране на статична и динамична информация за хардуера и процесите на възела и изпращането ѝ на заинтересованите сервизи от по-високо;
- Въвеждане на допълнителен модул в ядрото за осъществяването на миграция на процесите по заявка.

Б) модификация на Node Service за самостоятелна работа в peer-to-peer мрежи:

С известни промени в логиката, Node Service може да се използва напълно самостоятелно, извън рамките на грид системата, за извършването на разпределени изчисления във възлите на една peer-to-peer мрежа. В този вариант Node Service трябва да поеме и функциите по управлението на ресурсите в мрежата (аналогията на RMS в GrOSD). Сега модулът на всеки възел трябва периодично да обменя информация с останалите за състоянието на другите възли. В тази връзка, могат да се използват алгоритми, много близки като идея до маршрутизиращите протоколи в мрежовите комуникации (за подробна информация за тези протоколи виж например [31]), но използваната метрика е различна – тя се образува по критерии не за натовареност на връзката и разстояние до даден хост, а по отношение на предоставените от хоста хардуерни ресурси – процесор, памет, свободно дисково пространство, текуща заетост и т.н. (скоростта на мрежовия интерфейс може да се взима в предвид, ако се налага прехвърлянето на големи масиви от данни).



Фигура 9. Условна схема на работа на Node Service в модифициран вариант за peer-to-peer мрежа.

Предложеният по-долу вариант е скициран на абстрактно ниво като начален стадий на проект и е само една от възможностите за реализация на идеята. Освен това, обръщаме внимание, че идеята на проекта е разпределените изчисления да бъдат възможни само на база работата на услугата на Node Service по възлите, без помощта на сервизи от по-високо ниво. В това е и смисълът от равнопоставеността на възлите в peer-to-peer мрежата.

Всеки Node Service, който работи на определен възел, разполага с адресите на няколко сервиза, които работят на съседни или близки възли. Този списък от адреси може да се конфигурира ръчно при инсталацията на услугата или Node Service да се комбинира с друга технология за динамичното откриване на услугите от останалите възли. Периодично всеки Node Service обменя с няколко регистрирани от него пиъра информация за състоянието на възлите в мрежата. Част от информацията може да се разглежда като статична (напр. процесор, оперативна памет, мрежови интерфейс и т.н.) и може да се обновява само при определени събития, например рестартиране на възел, при това само за конкретния възел. Динамичната информация, сочеща общата натовареност на възела, трябва да се актуализира по-често, но трябва да се търси и някакъв баланс между честотата на опресняване на информация и оставащото процесорно време на възлите за самото обслужване на заданията. Това може да се постигне, от една страна, като всеки възел обменя информация с ограничен брой други възли едновременно (но в такъв случай опресняването на информацията за цялата мрежа ще става по-бавно), а от друга страна, изискваната динамична информация да е от такъв характер, че да не натоварва излишно възела с нейното изчисляване (например, лесно може да се каже какъв е броят на обработваните заявки в момента). Крайният резултат е, че всеки Node Service поддържа таблица от възлите в мрежата, в която са отразени техни статични и динамични параметри. Тази таблица подлежи на чести обновления и разширявания (особено в началото). По тази таблица по

определен алгоритъм услугата може да избере кои възли за какви типове задания са по-подходящи – за задания, които изискват повече процесорна мощ, повече памет или по-бърза връзка и дисково пространство.

Обслужването на заявка в такава мрежа протича, условно казано, в следните стъпки:

1. Клиент през инсталирано Node Service API отправя заявка за изпълнението на задание с *уникален за мрежата* идентификатор към произволно избран от него възел от мрежата. Ако клиентът е на възел от мрежата, заявката се адресира към *localhost,* т.е. Node Service от този възел. В това отношение, можем да предложим два варианта на мрежата, в зависимост от това дали се допуска да има клиентите, които са външни за нея, или всичките са част от възлите.
2. В заявката клиентът може да посочи с маски какви са приоритети за изпълнението – бързодействие, памет, мрежови трафик. По посочените критерии, адресираният възел изготвя някакви метрики на базата на поддържаната от него таблица и избира кандидат за изпълнение. В частност, това може да е той. Ако кандидатът е друг, възелът изпраща запитване към него дали може да поеме заявката. В запитването фигурира идентификационният номер на заданието. Отсрещната страна може да откаже, ако смята че вече е претоварена.
3. Ако отговорът е отрицателен, възелът може да сметне, че таблицата му е некоректна и да поиска извънредно обновяване от своите „съседи”. За да не се забави обслужването на клиента обаче, друг вариант е да отправи заявка към следващия най-добър кандидат, в случай, че се поддържа класацията на най-подходящите възли.
4. Ако се запитаният кандидат приеме заявката, той връща отговор, с който указва времето, през което ще чака пристигането на задание с посочения идентификационен номер. По този начин възелът се *резервира* за определено задание, което гарантира, че той няма да поема други анагажименти над определен лимит, за да е свободен за изпълнението. От друга страна, обявеното време на изчакване е необходимо, за да не бъде възелът блокиран вечно да чака за определена заявка.
5. Крайният резултат от допитването на клиента е, че той получава данните за избрания кандидат. Нещо повече, допитаният възел може направо да върне отворена клиентска връзка като отговор на запитването. По този начин процесът на „подбор” на кандидатите ще е максимално прозрачен за клиента. Тъй като процесът на подбор отнема известно време в голяма мрежа, вметсо клиентътда бъде блокиран до връщането на резултата от запитването, той може да го получи асинхронно, ако е регистриран по шаблона за *Наблюдател.*
6. От тази стъпка нататък обслужването на заявката протича по вече познат начин.

Ще отбележим, че реализирането на представения модел спрямо сегашния ще изисква:

- Малки промени в Node Service API-то и платформената съвкупност от интерфейси, тъй като в процеса се въвежда най-много още една стъпка

по запитването за подходящ кандидат и дефинирането на критерии за обработка. Съответно старият клиентски код може да претърпи само леки промени.

- Съществени промени в ядрото на Node Service, тъй като се налага въвеждането на повече стъпки във вътрешната обработка на заявките, плюс допълнителен модул за съставяне и поддръжка на *маршрутизираща таблица на заявките.*

Заключение

Основната задача на дипломната работа, а именно осигуряването на изпълнителни фунцкии на грид-системата GrOSD за обслужването на потребителски задания, беше постигната чрез разработката и реализацията на системна услуга за отдалечено изпълнение на приложения и услуги, които се установяват и работят във възлите на грида.

В началото на изложението направихме въведение в основната област на приложение на услугата Node Service, която мотивира нейното създаване, а именно архитектура и технология на грид-системите, като наблегнахме на спецификите при леките гридове и дадохме няколко примера за реализации. Във втора глава допълнително конкретизирахме модела на грид чрез обстоен анализ на структурата на GrOSD и детайлно разглеждане на съставящите го модули и обзор на техните функции и взаимодействия. В този контекст въведохме нуждата от реализация на услугата Node Service, посочихме общите изисквания към тип услуга и направихме сравнителен анализ на аналогични модули в други грид-системи.

В трета глава очертахме концептуалния модел, на който е базиран Node Service, като във фокус поставихме гъвкавостта на конфигуриране на услугата, нейната разширяемост и универсалност на употребата ѝ за изпълнението на произволни приложения, написани на Java. С разглеждането на платформенатата подсистема на услугата за описанието на задания беше демонстрирана възможността на системата за гъвкава дефиниция на широк кръг от задачи за изпълнение и формулирането на разнообразни критерии за събиране на резултата. Платформата за задания се допълва от представеното Node Service API, което, от една страна, дефинира конкретни реализации на платформените интерфейси и осигурява инструментиращи класове за съставянето на елементите на заявката и на самата заявка, а от друга страна, осигурява прозрачен достъп за клиента на услугата до функциите и ресурсите на Node Service, работещ на конкретен възел. В последната част на трета глава описахме логиката на изпълнение на заданията на възела и модела на разширяемост, която се постига чрез въвеждането на специализирани процесорни модули за обслужването на всеки един вид задание за изпълнение.

С четвърта глава илюстрирахме приложението на услугата в два сценария – 1) в опростен вариант, извън контекста на работа в грид – при него клиентската програма директно използва възможностите на Node Service API-то за създаването на заявки и тяхното зареждане за изпълнение; 2) детайлите по работата на Node Service в интегриран вид с услугите в GrOSD, с фокус върху взаимодействията с тях – в този случай Node Service получава заявките от

сервиза на RMS, а негово разширение обменя информация за капацитета и състоянието на възела с мониторинговата услуга на MS.

В шеста глава дадохме детайлите по използваните технологии, за които основно изискване беше да са от некомерсиално естество. Беше наблегнато върху възможностите на Java и JavaRMI, които осигуряват мобилност на кода и прозрачна комуникация, върху Java Reflection API, чрез който става възможно обработката на произволен компилиран клас, и върху шаблоните за проектиране, които дават ефективни решения за проблемите, възникнали по време на разработка на модулите на Node Service.

Последната глава от изложението беше специално посветена на разширяемостта на услугите, предлагани от Node Service, посредством гъвкавостта на нейната архитектура, капсулация на конкретните имплементации на елементите на заявката в API-то и специализация на модулите за обработка в изпълнителното ядро. Очертахме широките перспективи за развитие на Node Service за нуждите на GrOSD и направихме прогноза за възможностите му за модификация за самостоятелна работа в peer-to-peer мрежи, при което се поемат част от функциите на някои услуги в GrOSD. Планираната още при неговото създаване разширяемост и гъвкавост на конфигуриране на градивните елементи на проекта Node Service бяха основната причина тази глава да получи самостоятелно място в изложението.

Приложение А. Примерна програма за работа с Node Service API

Долният листинг цели да покаже как изглежда програмата на един типичен клиент на Node Service API-то. Демострира се:

- Създаване на логическа конекция към изпълнителния модул на Node Service от даден възел.
- Съставянето на заявка за изпълнението на Java приложение с main-метод, който записва резултата си във файлове.
- Добавяне на мониторингови обекти за следене хода на заданието.
- Добавяне на файлов локатор за маркиране на всички файлове, които съдържат изходния резултат.
- Зареждане на заявката за изпълнение.
- Клиентска имплементация на мониторингови обекти и файлов локатор.
- Получаване файловите при клиента.
- Зачистване на остатъчните файлове на възела и затваряне на клиентската връзка.

Самата програма създава задание за извикването на main-метода на класа SimpleProgram1, чийто компилиран код е пакетиран в архива test1.jar. Main-методът на SimpleProgram1 приема за вход два низа, съдържащи числа, изчислява тяхната сума и я записва в изходен файл, който впоследствие се прехвърля при клиента по негова заявка и резултатът се прочита от него.

```
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.net.URL;
import java.rmi.RemoteException;

import bg.unisofia.fmi.grosd.ns.api.NodeServiceClient;
import bg.unisofia.fmi.grosd.ns.api.exception.CreateNSClientException;
import bg.unisofia.fmi.grosd.ns.api.impl.NSAPIFactory;
import bg.unisofia.fmi.grosd.ns.api.task.args.impl.NSTaskArgumentFactory;
import bg.unisofia.fmi.grosd.ns.api.task.impl.NSTaskFactory;
import bg.unisofia.fmi.grosd.ns.api.task.result.locator.NSFileResultLocatorFactory;
import bg.unisofia.fmi.grosd.ns.api.task.targetclasses.impl.NSTaskTargetClassFactory;
import bg.unisofia.fmi.grosd.ns.remote.listener.adapter.NSActionStateAbstractListener;
import bg.unisofia.fmi.grosd.ns.remote.listener.adapter.NSTaskLifecycleAbstractListener;
import bg.unisofia.fmi.grosd.ns.remote.listener.event.ActionStateChangedEvent;
import bg.unisofia.fmi.grosd.ns.remote.listener.event.NSTaskAbortedEvent;
import bg.unisofia.fmi.grosd.ns.remote.task.NSJavaTask;
import bg.unisofia.fmi.grosd.ns.remote.task.arg.FileArgument;
import bg.unisofia.fmi.grosd.ns.remote.task.arg.NSTaskArgument;
import bg.unisofia.fmi.grosd.ns.remote.task.result.NSTaskResult;
import bg.unisofia.fmi.grosd.ns.remote.task.result.locator.NSFileFilterAbstractAdapter;
import bg.unisofia.fmi.grosd.ns.remote.task.result.locator.NSFileResultLocator;
import bg.unisofia.fmi.grosd.ns.remote.task.targetclass.NSTaskTargetClass;

public class NSClient {

  public static void main(String[] args) {

    // архив с компилирания код на заданието
    final String classFileName = "resource" + File.separator + "test1.jar";
```

```
    File classFile = new File(classFileName);

    try {
       // създаване на логическа връзка с услугата на локалния хост
       NodeServiceClient client = NSAPIFactory.createNodeServiceClient("localhost");
       System.out.println("Node Service Client created...");


       // дефиниране на класа на изпълнение
       NSTaskTargetClass targetClass = NSTaskTargetClassFactory.createNSTaskTargetClass("SimpleProgram1", new
NSTaskArgument[] {});

       // входни параметри за main-метода на изпълнение
       String[] strargs = new String[] {"3", "4"};

       // капсулация на jar-архива като файлов аргумент на заявката
       FileArgument fileArg = NSTaskArgumentFactory.createFileArg(classFile, true);

       // създаване на заявка с main-метод на изпълнение; указват се класа, от който да се вземе метода,
       // входните му аргументи и параметър, указващ компилирания код
       NSJavaTask javaTask = NSTaskFactory.createNSJavaMainTask(targetClass, strargs, new FileArgument[] {fileArg});

       // добавят се собствени мониторингови обекти за следене хода на заданието и получаване на резултата
        javaTask.addListener(new NSClient2().new ActionStateListener());
        javaTask.addListener(new NSClient2().new TaskLifecycleListener(client, javaTask.getTaskID()));

       // създава се и добавя собствен файлов локатор, чрез който Node Service
       // да се открият желаните от клиента файлове като изходен резултат
       NSFileResultLocator locator = NSFileResultLocatorFactory.createNSFileResultLocator(new NSClient2().new
MyFileFilter());
       javaTask.addFileResultLocator(locator);
       // зареждане на заявката през логическата връзка с изпълнителния модул
       client.uploadJavaTask(javaTask);

     } catch (RemoteException e) {
       e.printStackTrace();
     } catch (CreateNSClientException e) {
       e.printStackTrace();
     }

  }

  // дефиниция на мониторинга за следене промените в статуса на изпълнение на заданието
  class ActionStateListener extends NSActionStateAbstractListener {
    public void actionStateChanged(ActionStateChangedEvent event) throws RemoteException {
      try {
        System.out.println("Action state changed from " + event.getOldActionState() + " to " +
event.getNewActionState());
        String comment = event.getComment();
        if (comment != null && comment.length() > 0) {
          System.out.println("Reason: " + comment);
        }
      } catch (RemoteException e) {
        e.printStackTrace();
      }
    }
  }

  // дефиниция на мониторинга за получаване на резултата или за уведомяване за пропаднало задание
  class TaskLifecycleListener extends NSTaskLifecycleAbstractListener {

  private NodeServiceClient client;
  private String taskID;

  public TaskLifecycleListener(NodeServiceClient client, String taskID) {
    this.client = client;
    this.taskID = taskID;
  }

  public void taskAborted(NSTaskAbortedEvent event) throws RemoteException {
    System.out.println("Task failed !!!");
```

```
  }

    public void taskFinished(NSTaskResult result) throws RemoteException {
      try {
        String resultComment = result.getComment();
        if (resultComment != null && resultComment.length() > 0) {
          System.out.println("Result comment: " + resultComment);
        }

        // резултатът се получава като файлови URL-и
        URL[] myResult = result.getResultAsFileURLs();
        System.out.println("Transferred output files count: " + myResult.length);
        for (URL url : myResult) {
          File file = new File(url.toURI());
          DataInputStream in = new DataInputStream(new FileInputStream(file));
          int i = in.readInt();
          in.close();
          System.out.println("Output file " + file.getAbsolutePath() + " contains number " + i);
        }
      }  catch (Exception e) {
        e.printStackTrace();
      } finally {
        // заявка за изчистване на остатъчните файлове от изпълнението и затваряне на клиентската връзка
        client.gcTask(taskID);
        client.close();
      }
    }
  }

  // дефиниция на файловия локатор за откриване на изходните файлове
  class MyFileFilter extends NSFileFilterAbstractAdapter {

    public boolean accept(File pathname) {
      // маркират се всички файлове, записани в директория с име folder, да се третират като резултат
      if (pathname.isDirectory() && pathname.getName().equals("folder")) {
        return true;
      }

      return false;
    }
  }

}
```

Приложение Б. Списък на използваните таблици и диаграми

Библиография и използвани уеб ресурси

1. *I. Foster and C. Kesselman.* The Grid: Blueprint for a New Computing Infrastructure, *Morgan Kaufman Publishers 1999, ISBN 1558604758*
2. *I. Foster, C. Kesselman and S.Tuecke.* The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *IJSA, 15(3), Sage Publications, USA , 2001*
3. Globus Toolkit, www.globus.org/toolkit
4. Core Grid Technologies
5. The Open Grid Services Architecture, Version1.0, http://www.ggf.org/documents/GFD.30.pdf
6. *R. Badia, O. Beckmann, M. Buback, D. Caromel, V. Getov, S. Isaiadis, V. Lazarov, M. Malawski, S. Panagiotidi, J. Thiyagalingam.* Lightweight Grid Platform: Design Methodology. *In Proceedings of GRIDS@Work, Sophia Antipolis, France, October 10 – 14, 2005*
7. *Y.M. Teo and X. B. Wang.* AliCE: A Scalable Runtime Infrastructure for High Performance Grid Computing. *In Poceedings of IFIP International Conference on Network and Parallel Computing, Springer-Verlag Lecture Notes in Computer Science, Wouhan, China, October 2004*
8. AliCE Grid Computing Project, http://www.comp.nus.edu.sg/~teoym/atsuma.htm
9. *D. Kurzyniec, T. Wrzosek, D. Drzewiecki and Vaidy Sunderam.* Towards Self-organizing Distributed Computing Frameworks: the H2O Approach. *Parallel Processing Letters, 13(2):pp 273-290, 2003*
10. H2O Project, www.maths.emory.edu/dcl/h2o/
11. *L. B. Costa, L. Feitosa, E. Araujo, G. Mendes, R. Coelho, W. Cirne and D. Fireman.* MyGrid: a Complete Solution for Running Bag-of-tasks Applications. *In Proceedings of the SBRC 2004*
12. OurGrid Project, www.ourgrid.org
13. Jini, www.jini.org
14. JavaSpaces Specification, www.jini.org/nonav/standards/davis/doc/specs/html/js-title.html
15. GigaSpaces, www.gigaspaces.com
16. *D. Kurzyniec, T. Wrzosek, Vaidy Sunderam and A. Slominski.* RMIX: A Multiprotocol RMI Framework for Java. *In Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), pp.140-146, Nice, France, April 2003*
17. *M. Blyantov, L. Kirchev, V. Georgiev and K. Boyanov.* A Hierarchical Architecture Supporting Services in Grid. *In Proceedings of the Automatics and Informatics'05, Sofia, October 3-5, 2005, pp. 186 - 192*
18. *M. Blyantov, V. Georgiev and K. Boyanov.* A Model of Simplified Resource Management for Lightweight Multilevel Grid. *Information Technologies and Control, vol 2, 2005*
19. *L. Kirchev, M. Blyantov, V. Georgiev, K. Boyanov, I. Taylor, A. Harrison, S. Isaiadis, V. Getov and N. C. Linde.* Mapping "Heavy" Scientific Applications on a Lightweight Grid Infrastructure. *CoreGRID Integration Workshop CGIW'05, Piza, Italy, 28-30 November, 2005*
20. *L. Kirchev, M. Blyantov, V. Georgiev and K. Boyanov.* Communication Model Supporting Process Migration in Grid. *EXPGRID, Paris, 21 June 2006*

21. Java Remote Method Invocation Home Page - *http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp*

22. Java Remote Method Invocation – Distributed Computing for Java - *http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/index.jsp*

23. An Overview of RMI Application (The Java Tutorials) - *http://java.sun.com/docs/books/tutorial/rmi/overview.html*

24. Java programming dynamics, Part 2: Introducing reflection - *http://www.ibm.com/developerworks/library/j-dyn0603/*

25. Trail: The Reflection API (The Java Tutorials) - *http://java.sun.com/docs/books/tutorial/reflect/index.html*

26. *James W. Cooper*, The Design Patterns Java Companion, October 1998

27. *Pree, Wolfgang,* Design Patterns for Object Oriented Software Development, *Addison-Wesley,1994*

28. *Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M.,* A System of Patterns, *John Wiley and Sons, New York, 1996*

29. *Gamma, E., Helm, T., Johnson, R. and Vlissides, J.,* Design Patterns: Abstraction and Reuse of Object Oriented Design*. Proceedings of ECOOP '93, 405-431*

30. Eclipse, http://www.eclipse.org

31. *К. Боянов, Хр. Турлаков, Д. Тодоров, Л. Боянов, Вл. Димитров, В. Желязков,* Принципи на работа на компютърни мрежи. Интернет, *издателство Апиинфоцентър „Котларски", София, 2003*

32. *Z. Juhász, Kr. Kuntner, M. Magyaródi, G. Major, Sz. Póta,* JGrid Design Document, *Department of Information Systems, University of Veszprém, 2003*