



Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Elosztott feladatmenedzsment keretrendszer felhőben futó mérések automatizálásához

Készítette

Vincze Márton

Konzulens

Molnár Vince, doktorandusz

2016

TARTALOMJEGYZÉK

Összefoglaló	7
Abstract	8
1. Bevezetés.....	9
1.1. Motivációk	9
1.1.1. Feladat monitorozása.....	9
1.1.2. Több feladat monitorozása	10
1.1.3. Több feladat monitorozása több számítógépen	10
1.2. A keretrendszer fő célja.....	11
2. Hasonló eszközök.....	12
2.1. Szkript nyelvek.....	12
2.2. Continuous Integration szoftverek	13
2.2.1. Képességek.....	13
2.2.2. Hátrányok	13
3. Követelmények.....	15
3.1. Követelmények felderítése felhasználói igények alapján	15
3.2. Követelmények tisztázása	20
4. Specifikáció	23
4.1. Fogalomtár	23
Sikeres lefutás	23
Sikertelen lefutás	23
Erős függőség.....	23
Gyenge függőség.....	23
Feladat visszavonása	24
Feladatkörnyezet, Környezet azonosító, Környezeti mappa.....	24
4.2. Használati esetek	24
4.3. Architektúra.....	26
4.3.1. Master.....	26
4.3.2. Slave	32
4.4. Tipikus szekvenciák	33
5. Felhasznált technológiák	38

5.1.	C# nyelv, .NET platform	38
5.2.	WCF.....	38
5.2.1.	ServiceModel Metadata Utility Tool	38
5.3.	Task-based Asynchronous Pattern (Task alapú aszinkron tervezési minta)...	39
5.3.1.	Alternatívák	39
5.3.2.	Előnyök.....	39
5.3.3.	Hátrányok.....	40
5.3.4.	TAP minta megismerése	40
5.4.	NLog	40
5.5.	JSON.....	41
5.5.1.	Formátum rövid ismertetése	42
5.5.2.	Bemeneti formátum	42
6.	Részletes tervek	44
6.1.	Programozási nyelvek.....	44
6.2.	Projektstruktúra.....	44
6.3.	Bemenet	45
6.3.1.	A slaves szekció.....	45
6.3.2.	A taskGroups szekció	46
6.3.3.	InputHandler	47
6.4.	EnvironmentDescriptor.....	47
6.5.	ITask, TaskBase, CommandTask	48
6.5.1.	ITask	48
6.5.2.	TaskBase	48
6.5.3.	CommandTask.....	49
6.6.	SlaveManager	49
6.6.1.	WCF kliensek konfigurációja	50
6.6.2.	SlaveState.....	50
6.7.	Pool.....	51
6.7.1.	A Pool által biztosított garanciák.....	51
6.7.2.	Kényelmi szolgáltatások	51
6.8.	RecursiveTaskCanceller	52
6.8.1.	Példa.....	52

6.9.	AsyncProcessor, ütemező.....	53
6.9.1.	Explicit szálat használó verzió	53
6.9.2.	Aszinkron változat.....	54
6.10.	AsyncProgram	55
6.11.	UniqueStringGenerator	55
6.12.	ApplicationStateLogger	56
6.13.	SlaveServiceCmd	56
6.13.1.	Szolgáltatás konfigurációja.....	56
6.13.2.	ISlaveService, SlaveService	56
6.13.3.	Streaming engedélyezése.....	57
6.13.4.	ProcessHelper	57
6.14.	Kimenetek	58
6.14.1.	Results mappa.....	58
6.14.2.	Info és Error naplók.....	58
6.14.3.	Státusz fájl	59
6.15.	Tesztprogram.....	60
7.	Felhasználói útmutató	61
7.1.	Fordítás forráskódból	61
7.2.	Master telepítése.....	61
7.3.	Slave telepítése, konfigurálása	61
	Automatikus	62
	Manuális	62
7.4.	Slave elindítása.....	62
7.5.	Kampány, bemenet összeállítása.....	62
7.5.1.	Példák.....	64
7.6.	Master futtatása	67
7.6.1.	Master monitorozása	67
7.7.	Eredmények elemzése	67
8.	Potenciális kiterjesztések.....	69
8.1.	Folyamatosan futó Master szolgáltatás	69
8.2.	Biztonsági megfontolások	69
8.3.	Más OS-ek.....	69

8.4.	Hibákról értesítés küldése a felhasználóknak	69
8.5.	Új feladatok felvétele futás közben	69
8.6.	Feladatok visszavonása, megszakítása	70
8.7.	Új Slave felvétele futás közben	70
8.8.	Hibás Slave újrainicializálása	70
8.9.	Kétirányú, vagy fordított kapcsolat a Slave és Master között	70
8.10.	GUI	70
8.11.	Adatbázisba loggolás	71
8.12.	Fa szerkezetű függőségi gráf	71
8.13.	„Vagy” típusú függőségek	71
9.	Összefoglalás	72
	Köszönetnyilvánítás	73
	Ábrák jegyzéke	74
	Irodalomjegyzék	75

HALLGATÓI NYILATKOZAT

Alulírott Vincze Márton, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2016. 05. 30.

.....
Vincze Márton

Összefoglaló

A szakdolgozat keretében egy olyan keretrendszer került kidolgozásra, amely képes parancssori feladatokat több célszámítógép között ütemezni, a célszámítógépet a feladat futtatására felkészíteni, a feladatot a célszámítógépen lefuttatni, majd az eredményeket a célszámítógépről összegyűjteni. A felhasználó a feladatok között sorrendezést és függőségeket fogalmazhat meg, a keretrendszer ennek megfelelően, a célszámítógépek minél jobb kihasználtságára törekedve automatikusan osztja szét a feladatokat.

Az elosztott rendszer megfigyelésének egyszerűsítése érdekében a keretrendszer a komponensei aktuális állapotáról egy közös kimenetet készít, ezen keresztül a rendszer futása megfigyelhető. A futtatott feladatok eredményei szintén egy helyen kerülnek összegyűjtésre.

A szakdolgozat végigköveti a keretrendszer megvalósítását. Bemutatásra kerülnek a rendszer kifejlesztésének motivációi, a felhasználói igények alapján megállapított követelmények, majd az ez alapján elkészített specifikáció és rendszerterv. A megvalósítás során használt technológiák és az architektúra főbb aspektusai szintén bemutatásra kerülnek.

Az olvasó képet kaphat a keretrendszer működéséről, felhasználási területeiről, illetve használatának módjáról. A részletes tervek és az implementáció ismeretében szükség esetén a keretrendszer könnyen bővíthető.

Abstract

This thesis presents the design and implementation of a framework that is capable of scheduling, preparing and running command-line tasks distributed across multiple machines, as well as collecting the results of the tasks from them. The user is able to define ordering through dependencies between the tasks. While distributing the tasks automatically, the framework strives to utilize the machines as efficiently as possible.

To simplify the monitoring of the distributed framework, the system produces a unified output describing the state of all of its components. This makes observing the framework as-a-whole possible. The results of the tasks are also collected to a single place.

The thesis follows the design and implementation process. It presents the motivations inspiring the framework, the analysis of the user requirements, the specification of a system that meets these requirements, as well as the detailed plans and the implementation of the specified framework. The different technologies used and the main aspects of the architecture is also presented.

The reader will learn how the system works, will recognize the possible areas of application, and will also be able to use the software. Learning the details of the plans and implementation should also facilitate the easy extension of the framework.

1. Bevezetés

A következő fejezetben összefoglalásra kerül a szakdolgozat témája, a megvalósított keretrendszer, a keretrendszer mögötti motivációk, a keretrendszer fő céljai, illetve a szakdolgozat keretében a keretrendszer fejlesztésével kapcsolatos elvégzendő feladatok.

1.1. Motivációk

Számításigényes alkalmazások fejlesztésekor gyakori probléma, hogy a program tesztelése, illetve teljesítményének kiértékelése rendkívül hosszú időt vehet igénybe. Ezeket a feladatokat ráadásul a fejlesztés során többször is meg kell ismételni (például egy-egy hiba javítása után).

Különösen teljesítménymérésnél gyakori, hogy az alkalmazást egyre nagyobb bemeneteken futtatják, vizsgálva az eszköz korlátait. Ekkor egy bizonyos bemenetméret fölött már nem várható az eszköztől eredmény, a cél ennek a határnak a megtalálása. Jellemzően idő és erőforráslimiteket alkalmaznak a korlát megállapítására, ezek túllépése esetén a nagyobb bemeneteket már nem is érdemes lefuttatni.

Fontos feladat ezen műveletek automatizálása annak érdekében, hogy az egyébként is magas erőforráshasználatot minimalizáljuk, illetve jól kezelhetővé tegyük a feladatot, illetve a menet közben felmerülő hibákat.

A problémát nehezíti, hogy ezeket a tesztek és mérések az erőforrásigényesség miatt jellemzően több gépen, párhuzamosan futtatják, akár felhőben bérelt erőforrásokat használva. Ebben az esetben további kihívás a gépek közötti ütemezés, illetve ezek központosított monitorozása.

1.1.1. Feladat monitorozása

Egy hosszan tartó feladat monitorozása egyetlen számítógépen is munka. Minél aktívabban végzi ezt a felhasználó, annál több időt vesz igénybe, minél kevésbé aktívan, annál később értesül a hibákról és a feladat befejezéséről.

A munkát megkönnyíti, ha a futtatott eszköz az aktuális állapotát megfigyelhetővé teszi, például becslést ad a már elvégzett részfeladatok és az összes elvégzendő részfeladat arányára (hány százaléknál tart), becslést ad a hátralévő időre, vagy az összes részfeladat listázásával feltünteti, hogy melyikkel végzett, melyiken dolgozik, illetve melyik van még hátra. Segít továbbá, ha az eszköz képes kiemelt eseményekről értesítést küldeni (például kritikus hiba történt, vagy a feladat sikeresen befejeződött), ezzel biztosítva azt, hogy a felhasználó akkor is gyorsan reagálhasson, ha éppen nem monitorozza aktívan a feladatot.

Cél lehet, hogy a feladatot a felhasználónak aktívan ne is legyen szükséges monitorozni, elég legyen csak a feladat befejeződéséről értesülni, felszabadítva ezzel a felhasználót egy monoton munka alól.

Ha a futtatandó eszköz a fentebbi képességekkel nem rendelkezik, akkor a felhasználó a futtatókörnyezettől várhatja, hogy ezeket a hiányosságokat kompenzálni próbálja.

1.1.2. Több feladat monitorozása

A helyzet bonyolódik, ha a felhasználó nem egy, hanem több feladat futtatását szeretné. A feladatok egyszerre futtatása nem mindig kívánatos, gyakran elvárás, hogy egy feladat csak egy másik feladat után fusson. Ennek az okai lehetnek teljesítménybéli (például az operációs rendszer egy bizonyos terhelés fölött egyre romló teljesítményt mutat), vagy adódhat abból, hogy a feladat futásánál fontos az erőforrások kizárólagos használata, de lehet amiatt is, hogy az egyik feladat eredményétől függ, hogy egy másik feladatot futtatni kell-e egyáltalán. Ilyenkor a hibákra és a feladatok befejezésére mindig reagálni kell (el kell indítani az új feladatokat). Nagymértékű időkiesést jelenthet, ha a felhasználó ezeket az eseményeket későn veszi észre, vagy a bekövetkezésük idején nem áll módjában vezérelni a rendszert.

A hibákra reagálás legtöbbször hasonló. Amennyiben erőforráshiány okozta, valószínűleg az ehhez hasonló, de komplexebb feladatok sem fognak lefutni, ezeknek a futását érdemes megelőzni. Ilyenkor segít, ha a futtatókörnyezet képes a feladatokat szekvenciálisan futtatni, adott feladatok sikertelensége esetén a futtatási láncot megszakítani. Erre az operációs rendszerek nagy része magától is képes, például bash szkripteken, vagy batch állományokon keresztül.

Cél lehet, hogy a felhasználónak lehetősége legyen egyszerre elindítani az összes feladatot, és ne legyen szükséges aktívan monitoroznia az állapotot, az összes feladat befejezéséig ne is kelljen foglalkoznia a folyamattal.

1.1.3. Több feladat monitorozása több számítógépen

További összetettséget jelenthet, ha a felhasználó több, független számítógépen szeretné a feladatokat futtatni, például azért, hogy több számítógép használatával növelje az összes számítási kapacitást. Ilyenkor az aktív monitorozás bonyolódik, az állapot megtekintése a gépek számával egyre több kontextusváltást igényelhet a felhasználótól, emiatt pedig az aktuális állapotot már nem biztos, hogy egyetlen pillantással át lehet tekinteni.

A feladatok szigorúan szekvenciális ütemezése ekkor már nem mindig kívánatos. Ha még vannak elvégzetlen feladatok, a rendelkezésre álló számítógépeken minél hamarabb érdemes lehet elkezdni azokat akkor is, ha esetleg az egy számítógép esetében megfogalmazott szekvencia szerinti előző feladat még nem fejeződött be.

Nehéz lehet minden alkalommal eldönteni, hogy melyik gépen éppen melyik feladatot érdemes elkezdni futtatni. Ha erre a felhasználó előre meg tud fogalmazni egy szabályrendszert, ami alapján a futtatókörnyezet dönteni tud, akkor az ütemezési feladat automatizálható.

További kihívás több gép esetén a környezet felkészítése a feladatok futtatása. A feladat eredményeinek elérése, összegyűjtése is monoton, ismétlődő tevékenység lehet. Kényelmes lenne ezt egy központi helyen megtenni, vagy előre konfigurálni, majd ez alapján a feladatok futtatása előtt/után automatizálni.

Reagálni kell a számítógépek közötti kapcsolatok hibáira is. Ha egy célszámítógép elérhetetlenné válik, a felhasználónak fel kell derítenie a hiba fajtását és okát, majd a célszámítógép kapcsolatát, működését helyre kell állítani, vagy a számítógépet ki kell hagyni a feladatok futtatásából.

1.2. A keretrendszer fő célja

A cél egy olyan keretrendszer létrehozása, amely segít a felhasználónak nagy mennyiségű feladat futtatásában több számítógépen. A fentiek alapján a fő célok a következők.

- A feladatokat a felhasználó egyszerre, egy helyen indíthassa, az ütemezésért a keretrendszer felel.
- A feladatokat a gépek között ne a felhasználónak kelljen elosztania, ez is az ütemezés része legyen.
- A feladatok futtatására a számítógépeket a felhasználónak ne kelljen egyenként felkészítenie, minden gép egy előre megadott konfigurációval fusson.
- A feladatok futtatása egy helyről legyen monitorozható.
- Könnyen áttekinthető legyen, hogy melyik gépen éppen melyik feladat fut, melyek nem futottak még, illetve melyek futottak le sikeresen vagy hibával.
- Világos legyen, ha bármilyen hiba történt, a felhasználó utána tudjon nézni a hiba okának.
- Ha egy feladat hibázik, azok a feladatok ne fussanak, amik ebből következőleg előre tudhatóan szintén hibázni fognak.
- A feladatok eredményeit egy központi helyen nézhesse meg a felhasználó.

A továbbiakban ezen célok megvalósítását tárgyaljuk.

2. Hasonló eszközök

A következő fejezetben ismertetésre kerül több olyan, már létező eszköz, ami az előző fejezetben tárgyalt problémák részleges, vagy teljes megoldására alkalmas.

Az eszközöknél felsorolásra kerülnek azok a problémák és hiányosságok, amik indokolttá teszik a saját keretrendszer fejlesztését, ugyanakkor kiemeljük azokat a tulajdonságokat is, amik miatt az adott eszköz a fejlesztett keretrendszer valós alternatívája lehet.

2.1. Szkript nyelvek

Felmerül a kérdés, hogy érdemes-e egyáltalán keretrendszert fejleszteni, nem lehet-e az adott operációs rendszer rendszerhéjában kiadott parancsokkal, néhány szkripttel megoldani a feladatokat.

A Unix-szerű rendszereknél elterjedt shell szkriptek (például Bash [1]) elég erősek a feladatok nagy részének elvégzéséhez. Valószínűleg minden funkcionalitás, amit a keretrendszertől várunk megvalósítható bash szkriptekben, Windows operációs rendszereken azonban problémát jelent a batch szkriptek gyengesége. A követelmények elemzésénél kiderül, hogy a feladatok egy része biztosan Windows alatt kell fusson, így a batch nyelv helyett biztosan más nyelvet kell választani.

Hátrányt jelent a shell és batch szkriptek esetében az objektumorientált szemléletmód hiánya. A nyelveket nem komplex szoftverek megírására, hanem egyszerű automatizációkra tervezték. Bár a feladat valószínűleg nem lehetetlen, mégis, a tervezést, megvalósítást és karbantartást jócskán nehezítené ezen nyelvek választása.

Egyéb, általánosabb célú szkript nyelvek, mint például a Python [2], vagy JavaScript [3] (Node.js környezetben) is felmerülhetnek, ezek magasabb szintű nyelvi elemeket tartalmaznak, mindkettőben lehetséges objektumorientált és funkcionális paradigmák szerint programot tervezni.

Az ütemezési feladat komplexitása, a feladatok párhuzamos futtatásának aszinkron jellege (feladatokra nem lehet aktívan várakozni) komolyabb tervezést igényel. Ezzel szemben fontos szempont a fejlesztés gyorsasága, a hibákra akár órákon belül reagálás. Praktikus lenne egy olyan nyelvet választani a keretrendszer implementációjához, amiben mérnök informatikus képzés során megtanítanak bonyolultabb rendszereket megtervezni és implementálni, tehát amiben a jelenlegi, illetve későbbi fejlesztők is otthonosan programoznak.

Ezzel együtt a feladat egyes részeit (például kapcsolatfelvétel távoli géppel, utasítások kiadása a távoli gépen) végezhetjük például bash-ben, ez a keretrendszer részét képezheti, amennyiben ez bizonyul az adott esetben legkézenfekvőbbnek. Unix-szerű rendszerek távoli vezérléséhez például az SSH a legkézenfekvőbb megoldás.

Később a megvalósításnál látni fogjuk, hogy ugyan az első implementációban a hálózati kommunikáció a WCF [7] szolgáltatásain keresztül lett megvalósítva, a feladatok futtatása pedig egy .NET keretrendszerbeli gazdafolyamatban, viszont a keretrendszer lehetőséget nyújt ezek leváltására.

2.2. Continuous Integration szoftverek

Egyes Continuous Integration szoftverek képesek az összes követelményünket teljesíteni. Feladatok széles választékát képesek futtatni, a command-line parancsok is megtalálhatóak a palettán.

Részletesebben a TeamCity szoftver [4] került felmérésre, remélve, hogy jól reprezentálja a konkurens szoftverek hozzávetőleges képességeit is. A szoftver jóval több feladatot képes ellátni, mint amire szükségünk van.

2.2.1. Képességek

Azok a képességek, amelyek teljesítik a követelményeinket, vagy hosszú távon esetleg érdekesek, szükségesegek lehetnek, a következők.

- Feladatok automatikus elosztása több kliensre.
- Feladatok eredményeinek begyűjtése.
- Feladatok közötti függőségek definiálhatósága.
- A feladatot futtató gép – a TeamCity terminológiája szerint Agent – konzol kimenete már a futás közben is látható az összefoglaló felületen.
- Részletes felületen megtekinthető, hogy melyik Agent gépen mi történik éppen.
- A feladatok eredményei historikusan megtekinthetőek.
- A feladatok megismételhetőek.
- A webes felület a legtöbb eszközön használható.
- Beépített felhasználókezelés, jogosultságkezelés többfelhasználós helyzetekhez.

2.2.2. Hátrányok

A következő problémák nehezítik vagy akadályozzák a szoftver alkalmazását.

- Az ingyenes licenzhez korlátozott számú feladatkonfiguráció és számítógép engedélyezett. Valószínűleg van ingyenes alternatívája.
- A telepítő nagy, a telepítés hosszadalmas folyamat. A friss verziójának szerver telepítője 646 MB Windows alatt, telepítve több mint 1 GB-ot foglal, és a többi operációs rendszerre is hasonló méretű. Az Agent-ek telepítése ugyan kis beavatkozást igényel, de indokolatlanul hosszú időt vesz igénybe. Ezt minden új virtuális gépnél el kell végezni.
- Egy feladat bekonfigurálása hosszú folyamat, a felhasználói felületen nincs lehetőség több feladat köteget bekonfigurálására és a függőségek megadására, ezeket egyenként kell megtenni. Mivel alapvetően megismételhető feladatok többszöri futtatására tervezték ez érthető is, viszont esetünkben mérések egyszeri futtatásához ez túl hosszadalmas.

- Feladatok köteget bekonzurálásához lehetőségünk van bővítmőnyeket írni, viszont ez a bővítmőny a TeamCity mély megismerését igényelné és hasonlóan komplex feladat, mint egy egyszerű keretrendszer kidolgozása.

A TeamCity és egyéb Continuous Integration szoftverek használata elvetésre került a hátrányoknál felsorolt okok miatt.

3. Követelmények

A következő fejezetben összegyűjtésre kerül, hogy a keretrendszerrel szemben milyen elvárásokat támasztanak a leendő felhasználók. A követelményanalízis célja annak vizsgálata, hogy a felhasználói elvárások mennyire valósak, mi az, amit a felhasználó valóban szeretne, mik a rendszerrel szemben támasztott tényleges követelmények.

Gyakran előfordul, hogy a felhasználónak nem pontosan arra van szüksége, amit megfogalmaz, helyette egy olyan megvalósítást ír körül meg elvárásként, ami eleget tesz egy – akár nem megfogalmazott – tényleges követelménynek. A követelmény kiszűrése után a felhasználóval meg kell beszélni, hogy ténylegesen erre van-e szüksége.

A követelmények felmérésénél hamar definiálásra kerültek kifejezések, ezeket a követelmények leírása során használok.

- **Master:** A keretrendszer azon része, amit a felhasználó az alkalmazás indítási pontjaként lát, aminek a felhasználó a feladatokat megadja. A Master kifejezés használható a Mastert futtató gépre is.
- **Slave:** A keretrendszer azon része, ami a feladatokat futtatja. A Slave kifejezés használható a Slave-et futtató gépre is.
- **Kampány:** Több hasonló, összefüggő feladat, amit a felhasználó egyszerre, egy csomagban szeretne lefuttatni.

3.1. Követelmények felderítése felhasználói igények alapján

Az első megbeszélések során rendszertelenül megbeszéltük, hogy ki mit vár el a keretrendszerrel, annak milyen feladatokat kell ellátnia. A közvetlen felvetések formája meglehetősen informális volt, sok igény többször is elhangzott különböző formában. Az ebből származó követelmények rendszerezésre kerültek.

Az első táblázat a felhasználók elhangzott igényeit rögzíti, egy azonosítóval, rövid névvel, illetve egy leírással, ami tartalmazza az elhangzott igény részleteit egy bekezdésben, az igény esetleges értelmezését, tisztázását, illetve az ebből származó követelményeket.

ID	Elnevezés
	Leírás, következtetés
A1	Parancssori feladatok futtatása A felhasználók parancssori feladatokat akarnak futtatni. Követelmények: • Parancssori feladatok futtatása
A2	Futtatás virtuális gépekre szétosztva A feladatokat a felhasználók egyszerre több (virtuális) gépen szeretnék futtatni. Követelmények: • Elosztott rendszer, amely egyszerre több (virtuális) gép kezelését teszi lehetővé.
A3	Tipikus működés A következő lista került egy táblán felvázolásra. 1. VM – Csatlakozunk a virtuális géphez. 2. Config – Konfiguráljuk fel. 3. Data – Küldjük át a méréshez szükséges fájlokat. 4. Initialize – Inicializáljuk a mérést. 5. Measure – Mérjük. 6. Collect – Gyűjtjük össze a mérés eredményét. A megfogalmazott igény valójában főleg egy félkész terv vázolása, több, egymástól független követelmény szűrhető le belőle. Követelmények: • A virtuális gépet a feladatok futtatása előtt fel kell konfigurálni. • Egy feladat futtatásához adatfájlokra van szükség, ezt a feladatnak futás közben el kell tudni érni, lehetőleg lokálisan. • A feladat futtatása előtt szükség van valamilyen inicializációra. • A feladat eredményét ne a feladatot futtató virtuális gépen lehessen csak elérni, hanem valamilyen központi helyen ahova az eredmények összegyűjtésre kerülnek.
A4	Feladat után következő induljon Ha egy feladat befejeződött, induljon a következő. Követelmények: • A feladatok ne egyszerre fussanak. • Egy feladat indítása után ne manuálisan kelljen a következő feladatot indítani, meg lehessen az összes feladatot előre adni és egy ütemezőre bízni. • A feladatok futtatása között kevés legyen az üresjárat.
A5	Maximális kihasználtság A rendelkezésre álló számítógépek bérlése gyakran drága, így lehetőleg a legkevesebb időt töltsék kihasználatlanul. A kampányt lehetőleg gyorsan be akarjuk fejezni. Követelmények: • A feladatok futtatása között kevés legyen az üresjárat. • Fölösleges műveletekre ne várakozzunk.

A6	Egy gépen egy dolog fusson
	Ne fusson egy gépen két feladat egyszerre, mert torzíthatja a mérést. Ez azt jelenti, hogy fontos, hogy a virtuális gépen a mérés futtatása közben ne történjen más teljesítményigényes művelet. Követelmények: • Egyszerre egy gépen ne fusson két feladat. • Egy feladat futtatása közben a gépen ne történjen semmilyen más művelet.
A7	Indítás a Masteren
	Legyen egy Master gép ahol a feladatokat meg kell adni, innen kerüljenek szétosztásra. Követelmények: • Az elosztott rendszernek legyen központja. • A feladatokat egy csomagban lehessen megadni, elindítani.
A8	Bemenet egy kampány
	A bemenet egy kampány legyen. Követelmények: • A feladatokat a futtatás előtt meg lehessen egy csomagban adni.
A9	Eredmények a Masteren
	A feladatok eredményeit ott lehessen megnézni egy helyen, ahol megadtuk őket, ne kelljen a gépek között keresni. A bemenet és az eredmények egy központi helyen kell, hogy megjelenjenek. Követelmények: • Az elosztott rendszernek legyen központja. • A központi helyen minden eredmény megtekinthető legyen. • A központi helyen kelljen a feladatokat megadni.
A10	Feladatok közötti sorrend
	Több hasonló feladat futtatása úgy, hogy először a kisebb modellméretekkel próbálkozzunk, aztán egyre nagyobbakkal. Követelmények: • A feladatokra meg lehessen adni valamilyen sorrendezést.
A11	Feladat visszavonása hibára
	Ha egy modellre a mérés nem fut le, akkor a nagyobb modellekkel már ne próbálkozzunk. Követelmények: • Egy feladatnak lehessen sikeres és sikertelen kimenete. • Feladatokra meg lehessen adni, hogy sikertelenségük esetén melyik más feladatok ne fussanak le, vagy melyik feladat sikertelensége esetén ne fusson le.
A12	Feladatok előre megadása
	A szoftver a feladatokat beavatkozás nélkül el tudja végezni, az elvégzendő feladatokat fel lehessen sorakoztatni és elkezdni, otthagyni és csak akkor ránézni, amikor minden végzett. Követelmények: • A feladatokat a futtatás előtt meg lehessen egy csomagban adni. • A kampány futtatása közben ne igényeljen beavatkozást.

A13	Hibakezelés
	Ha valami hiba történik, akkor ne omoljon össze a rendszer, hanem működjön tovább, ha még tud hasznosat csinálni, viszont értesüljön a felhasználó, hogy hiba történt, hogy minél hamarabb meg lehessen vizsgálni a fajtáját és okát. Követelmények: • A keretrendszer készüljön fel a lehetséges hibákra. • Amit nem gátol meg egy hiba, az kerüljön elvégzésre. • A hiba érhetően jelenjen meg a kimeneten, akkor is észrevehető legyen, ha nem közvetlenül a kimenet olvasásakor történt, hanem régebben. • A hibákról küldjön értesítést a rendszer.
A14	ThreadPool Slave-ekhez
	Legyen a vezérlőgépen egy ThreadPool, Slave-enként foglaljunk egy szálát, amik aktívan keresnek maguknak feladatot és futtatják. Ez egy megvalósítási javaslat, viszont abból, hogy egy ilyen megvalósítás milyen problémákra nyújt megoldást, lehet következtetni a követelményekre. A több szálra tett javaslat párhuzamosításra vonatkozik, Slave-ek ne várokozzanak egymásra, a feladatok befejezése után ne várják be egymást, minél hamarabb kezdjenek új feladaton dolgozni. Követelmények: • A feladatok a Slave-enként párhuzamosan futhassanak. • A feladatok futtatása között kevés legyen az üresjárat.
A15	Listák, veremk, bejárások
	Legyen több lista, vagy verem, amikből a feladatokat lehet kivenni. Ez egy megvalósítási javaslat arra, hogy lehessen a feladatokat sorrendezni. Követelmények: • A feladatok között lehessen sorrendet megadni. • Egyes feladatok között ne feltétlenül legyen sorrend definiálva,
A16	Ütemezések
	Figyeljünk arra, hogy minden verem körülbelül egyszerre fogyjon ki, mondjuk úgy, hogy mindig abból a veremből válasszunk feladatot, amelyik a leghosszabb, de lehet sima szélességi bejárás is, később lehessen akár megadni többféle ütemezést és választani közülük. Ez a feladatok sorrendezésére és ütemezésére több megoldási javaslat. A javaslatok a Slave éhezésének elkerülésére vonatkoznak, ezen probléma megoldása a valódi követelmény. A javaslat felveti a választható ütemezést is, ami azért problémás, mert valamilyen metodikát kell kitalálni arra, hogy a bemenet függvényében a megvalósított ütemezők közül mi alapján érdemes választani. Probléma, ha egy Slave nem kap feladatot, mert minden feladat valamelyik másakra várokozik. Probléma, ha a felhasználónak kell ütemezést választania, de nem tudja mi alapján (ugyanakkor ha rosszul választ, akkor is kap eredményt, csak lassabban). Ha nem tud választani, akkor nincs is értelme a választást felvetni. Követelmények: • Ne legyen olyan állapot, hogy a szabad Slave-ek elől minden feladat blokkolva van be nem fejezett függőségek miatt. • Világos legyen, hogy egy adott bemenetre milyen ütemezőt érdemes választani.

A17	Egy verem egy Slave-en
	Egy verem feladatai egy Slave-en fussanak, egymás után sorrendben, de ha kevesebb verem van, mint Slave akkor a megmaradt Slave-eknek is jusson a veremből. Ez az igény visszakérdezésre megszűnt, szükségtelennek lett ítélve, ha az alábbi követelmények teljesülnek. Követelmények: • A feladatok között lehessen sorrendet megadni. • Futhasson olyan feladat, aminek még nem teljesült minden függősége, de csak ha nem várakozik olyan, aminek teljesült.
A18	Hamarabb kezdés, ha nincs más
	Ha már nem tudunk mást választani, akkor kezdjük el a nagyobb modellre a mérést akkor is, ha a kisebb változata még fut. Követelmények: • Futhasson olyan feladat, aminek még nem teljesült minden függősége, de csak ha nem várakozik olyan, aminek teljesült.
A19	StdOut, StdErr gyűjtése
	A program Standard Output és Standard Error kimeneteit vissza kell küldeni a Masterre, de az sem baj, ha a Slave-en fájlba van irányítva és az kerül átküldésre, a készített fájlokat amúgy is át kell küldeni. Követelmények: • A feladatok kimeneteként készült fájlokat a Slave-ről össze kell gyűjteni a Masterre. • A feladatok Standard Output és Standard Error kimeneteit vissza kell küldeni a Masterre.
A20	Fájlvitel egyben
	A kampány előtt tömörítsük össze egyetlen fájlba az összes szükséges fájlt (futtatandók, bemenetek), majd egyben küldjük át a Slave-re és ott csomagoljuk ki. Ugyanezt a módszert használjuk az eredmények gyűjtésekor is. Így az átvitelt egyszerűbb megvalósítani és gyorsabban megtörténik, illetve egy kampány során sok mérés használhatja ugyanazokat a fájlokat. A feladatok közbeni fájlvitel torzíthatná a mérést. Érthetőbb is, ha a keretrendszer előbb másol, aztán mér, a másolás jellemzően úgy is gyors lesz, mivel a Slave gépek feltételezhetően egy hálózatban vannak. Ez az igény egy javaslat arra, hogy hogyan lesz egyszerű a fájlvitelt implementálni, illetve, hogy hogyan gyors a keretrendszer. Követelmények: • Gyors fájlvitel. • Több feladat használhassa ugyanazokat a bemeneti fájlokat. • Miközben egy feladat fut, ne történjen fájlvitel.
A21	Eredmények egyben
	A mérések eredményét csak az egész kampány lefutása után töltjük vissza. Lehet, hogy például két mérés ugyanahhoz a fájlhoz fűz hozzá, de ez ritka. Ez az igény részben egyezik az előzővel, de más az indoklás, így van új levont követelmény. Követelmények: • Egyes feladatok használhassák előző feladatok fájljait

A22	Dashboard
	Legyen egy felület ahol látható, hogy épp mi történik, mi sikerült, mi hibázott, illetve mi van hátra. Ha hiba történt legyen egy látványos hibajelzés, majd lehessen látni, hogy mi volt a hiba. Követelmények: • Legyen egy áttekintő felület. amit elég monitorozni, tartalmazza, hogy mi történik, mi sikerült, mi hibázott, mi van hátra. • Azonnal látható legyen, ha hiba történt. • A hibák rögzítésre kerüljenek, utólagosan is meg lehessen tekinteni a részleteiket.
A23	Windows
	A Master és a Slave-ek működjenek Windows alatt, mert egyelőre ezen a platformon tesztelnek algoritmusokat. Praktikus lenne, ha később lennének Linux alatti Slave-ek is, de egyelőre ne ez legyen a fókusz. Követelmények: • Master működjön Windows operációs rendszer alatt. • Slave működjön Windows operációs rendszer alatt. • Keretrendszer olyan kialakítása, hogy Linux alatt futó Slave is integrálható legyen később.
A24	Bővíthetőség, továbbfejleszthetőség
	A forráskód legyen érthető, a rendszer úgy készüljön, hogy később más fejlesztők is bővíteni tudják, vagy át tudják alakítani. Követelmények: • Kódkonvenciók betartása. • Érthetően feltagolt kód, • Tiszta szintaktikára törekedni. • Kód tervezésénél a potenciális bővítési pontokat figyelembe kell venni. • Egy olyan nyelvet kell választani, amiben a potenciális fejlesztők is otthonosan mozognak.

3.2. Követelmények tisztázása

Ebben az alfejezetben az előzőekben felderített követelmények kerülnek tisztázásra, a követelmények egységesítve és pontokba rendezve jelennek meg.

Az igények alapján körvonalazódik egy Master-Slave architektúra, egyetlen Master folyamattal és több Slave folyamattal. A követelmények ennek mentén csoportosíthatóak általános követelményekre, Masterrel szemben támasztott követelményekre és Slave-ekkel szemben támasztott követelményekre, erre az azonosítójuk utal (B-vel kezdődnek az általános, M-mel a Masterre vonatkozó, S-sel pedig a Slave-ekre vonatkozó követelményazonosítók).

B1	Beavatkozás nélküli futtatás
	Egy kampány futtatásának elindítása után ne legyen szükség a rendszer működésébe beleavatkozni addig, amíg a kampány nem végez.
B2	Feladatok közötti függőségek
	A feladatok között megadhatók legyenek függőségek. Egy feladatot lehetőleg akkor futtassunk, ha minden függősége sikerült. Ha nincs más, futtassunk olyan feladatot is, ahol még függőség van folyamatban. Egy feladat végződhesen sikeresen, vagy hibával. Ha egy feladat egy függősége hibával végzett, a feladatot vissza kell vonni. Egy feladathoz a függőségei mindig előbb kezdjenek el futni.
B3	Bővíthetőség, továbbfejleszthetőség
	Egy olyan nyelvet kell választani, amiben a potenciális fejlesztők is otthonosan mozognak, a kódot érthetően kell feltagolni és törekedni a tiszta szintaktikára. Kód tervezésénél a potenciális bővítési pontokat figyelembe kell venni.
B4	Master-Slave architektúra
	A kertetrendszer egy elosztott rendszer legyen, ami több (virtuális) gépen működik. Legyen egy központi Master folyamat, ahol a kampány megadható és az eredmények megtekinthetők. Legyen több Slave folyamat további (virtuális) gépeken, ahol a parancssori feladatokat lehet futtatni.
M1	Kampány megadása a Master gépen
	A kampány feladatait a Master gépen lehessen megadni egyben, innen kerüljenek át a Slave-ekre.
M2	Kampány eredményei a Master gépen
	A kampány feladatainak eredményeit (kimeneti fájlok és a feladat konzol kimenete) a kampány lefutása után a Master gépen el lehessen érni.
M3	Hatékony ütemező
	Az ütemező előre megadott szabályok szerint sorrendezze a feladatokat, ne hagyja, hogy egy Slave éhezzen. Slave ne várjon más Slave feladatára, ha egy Slave végez a feladatával, azonnal kapjon újat. A feladatok között ne legyen mindig sorrend, előfordulhasson, hogy két feladat bármilyen sorrendben futhat.
M4	Hibakezelés
	Ha egy feladat hibázik, a tőle függő, más feladatok ne fussanak. Ha egy Slave elérhetetlenné válik, a többi Slave még futtathassa a feladatokat. Egy Slave-ről az ütemező föl tudja mérni, hogy elérhető-e vagy nem. A hibák világosan jelenjenek meg, a kimenetről könnyen leolvashatók legyenek, érthető formában el legyenek tárolva. A hibákról küldjön a rendszer értesítést.
M5	Master fájlküldés, fájlfogadás
	A Master képes legyen a Slave-nek nagyméretű fájlokat küldeni. Ez a fájlátvitel történjen a kampány futtatása előtt. A Master képes legyen a Slave-ről a feladatok eredményeit letölteni. Ez a fájlátvitel történjen a kampány futtatása után. A fájlátvitel legyen gyors.

M6	Kimeneti fájlok, konzolkimenetek
	Egy feladat eredményének kiértékeléséhez szükség van a konzol kimenetére (Standard Output, Standard Error), illetve a feladat által létrehozott fájlokra, ezek mind a Master gépen legyenek elérhetőek a kampány lefutása után.
M7	Dashboard
	Legyen egy áttekintő felület, amit elég monitorozni. Tartalmazza, hogy melyik Slave éppen melyik feladaton dolgozik, melyik Slave működik egyáltalán, tartalmazza, hogy melyik feladatok futottak már le, hogy melyik feladatok álltak le hibával és melyik feladat várákozik még a futtatására.
M8	Master Windows-on
	A Master fusson Windows operációs rendszer alatt.
M9	Csatlakozás Linux-os Slave-ekhez
	A Master képes legyen Linux alatt futó Slave-ekhez is csatlakozni.
S1	Parancssori feladatok futtatása a Slave gépeken
	Lehetőség legyen konzol parancsokat futtatni feladatokként. A parancsok bemenete az argumentumok és a használt fájlok. Egy parancs futtatása előtt legyen lehetőség a Slave-et inicializálni.
S2	Slave gépek konfigurációja
	Feladatok futása előtt a virtuális gépet a futtatásra fel kell készíteni. Képes legyen a bemeneteket és a feladatot fogadni, a feladat bemeneteit a feladatok számára elérhetővé tudja tenni.
S3	Slave fájlkezelés
	A Slave képes legyen fájlokat fogadni és küldeni, a fogadott fájlokat a feladatok számára elérhetővé kell tudni tenni. Ezek a bemeneti fájlok a feladatok között megoszthatóak legyenek.
S4	Zavartalan mérések
	Miközben egy Slave egy feladatot futtat, a Slave gépen ne fusson más feladat, ne másolódjanak fájlok, illetve ne történjen semmi más erőforrásigényes művelet. Ez azzal is együtt jár, hogy egy virtuális gépen csak egy Slave folyamat fusson, semmi más (se Master, se más Slave, se más).
S5	Slave Windows-on
	A Slave szoftver fusson Windows operációs rendszer alatt.

4. Specifikáció

A következő fejezetben specifikálásra kerül a megvalósítandó keretrendszer és komponensei, a komponensek feladatai, illetve a komponensek felhasználásának módja. A Specifikált rendszernek ki kell elégítenie a 3. fejezetben megfogalmazottakat.

4.1. Fogalomtár

Elsőként a specifikáció során használt kifejezések kerülnek tisztázásra.

Sikeres lefutás

Egy feladat sikeresen lefutott, ha a Slave-en futás közben nem dobott kivételt, visszatérési értéke nem jelzett hibát és futás közben nem lépte túl a feladatra megadott időkorlátot.

Szinonimák: sikeres befejeződés, sikeres leállás.

Sikertelen lefutás

Egy feladat sikertelenül futott le, ha a Slave-en futás közben kivételt dobott, vagy visszatérési értékben hibát jelzett, illetve ha túllépte a feladatra megadott időkorlátot.

Szinonimák: hibával leállás, hibával befejeződés.

Erős függőség

Egy feladat csak akkor kerül futtatásra, ha minden erős függősége sikeresen lefutott. Ha egy feladat erős függősége hibával fejeződött be, vagy visszavonásra került, akkor a feladat visszavonásra kerül.

Gyenge függőség

Egy feladat csak akkor kerül futtatásra, ha minden gyenge függősége sikeresen lefutott, vagy futtatás alatt van. Olyan feladatok, amelyeknek már minden gyenge függősége lefutott egyértelmű prioritást kapnak olyan feladatokkal szemben, amelyeknek még fut gyenge függősége. Más szóval egy olyan feladat, amelynek gyenge függősége még fut, nem kerül futtatásra, ha futtatható olyan feladat, amelynek minden gyenge függősége sikeresen lefutott. Ha egy feladat gyenge függősége hibával fejeződött be, vagy visszavonásra került, akkor a feladat visszavonásra kerül.

A gyenge függőség előnyös tulajdonsága, hogyha a felépített függőségi gráf nem tartalmaz irányított kört, és még nem került az összes feladat kiosztásra, akkor mindig van olyan feladat, amit ki is lehet osztani, hiszen egy feladat akkor nem kiosztható, ha van függősége, ami még nem fut, így a feladat helyett vizsgálhatnánk az adott függőséget. A függőségek mentén, a gráf aciklikus volta miatt vagy eljutunk egy olyan feladathoz, aminek minden függősége fut, illetve már futott, vagy egy levélig, aminek nincs függősége, így az előző kitétel szélsőséges esete igaz rá. Természetesen ez nem egy rögzített algorit-

mus feladatok kiosztásának menetére, csak az előző kitétel belátásának módja. Ezzel garantáltuk, hogy amíg van kiosztható feladat, addig Slave gyenge függőség nem fog éhezni.

A gyenge függőség fogalma a megvalósítás során sokkal gyakrabban jelenik meg, mint az erős függőség fogalma, így ahol egyszerűen a függőség kifejezés kerül használatra, az a gyenge függőség fogalmára utal.

Feladat visszavonása

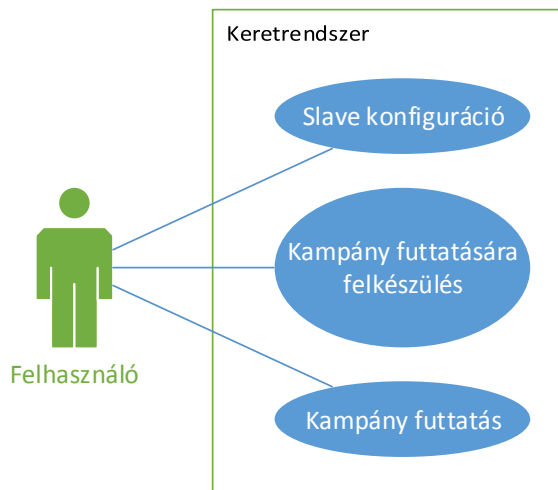
Ha egy feladat visszavonásra kerül, a feladatot az ütemező nem indíthatja el. A már futó feladatokat visszavonáskor az ütemezőnek nem kötelező leállítani, de a feladat futtatását megszakíthatja, a Slave-nek új feladatot oszthat.

Feladatkörnyezet, Környezet azonosító, Környezeti mappa

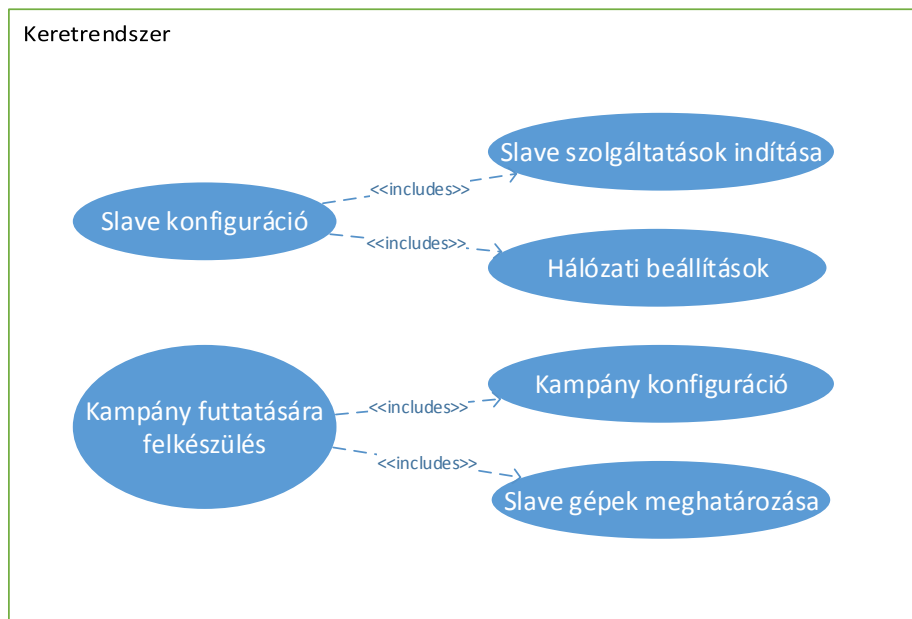
A feladatok a Slave-eken azonosítóval ellátott környezetekben futnak, a környezet azonosító szolgál a környezet egyértelmű meghatározására. Egy környezethez egyértelműen tartozik egy munkakönyvtár, ezt nevezzük környezeti mappának. A környezetben elérhetőnek kell lennie a futtatott feladatok bemenetének, illetve a feladatok a környezetben hozzák létre a kimenetüket.

4.2. Használati esetek

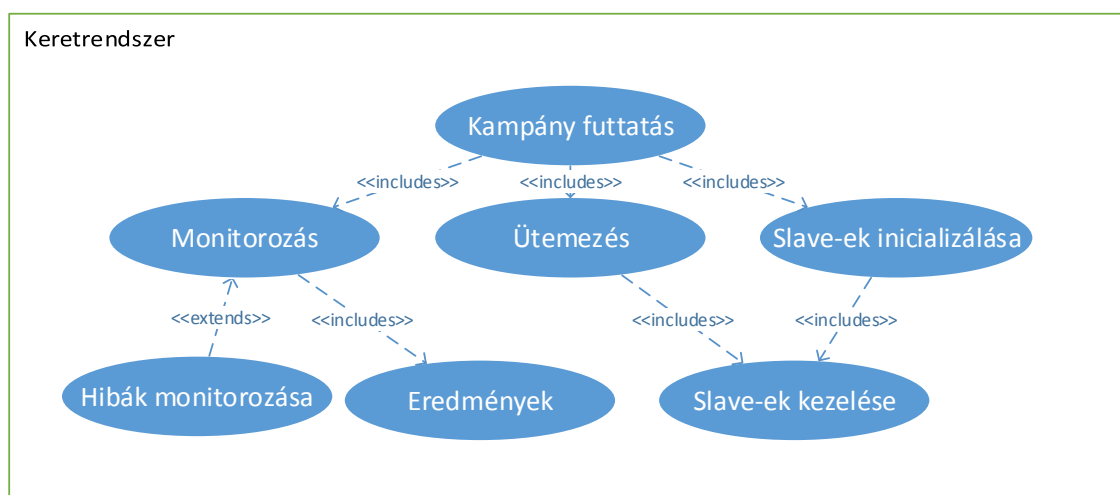
A felhasználó esetében alapvetően három használati esetet különítünk el (1. ábra). Ezek a Slave konfigurációja, a kampány futtatására felkészülés és a kampány futtatása. A használati esetek kifejtése funkciók felbontásával történt meg. A 2. és 3. ábrán ez a felbontás látható.



1. ábra. A három fő használati eset.

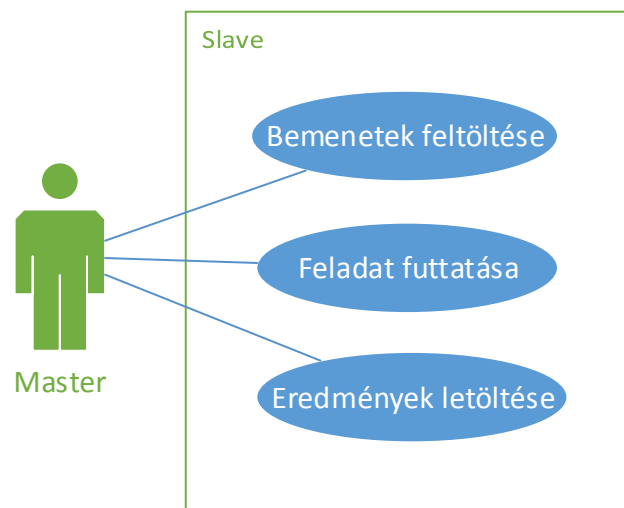


2. ábra. A használati esetek felbontása



3. ábra. A kampány futtatás használati eset felbontása

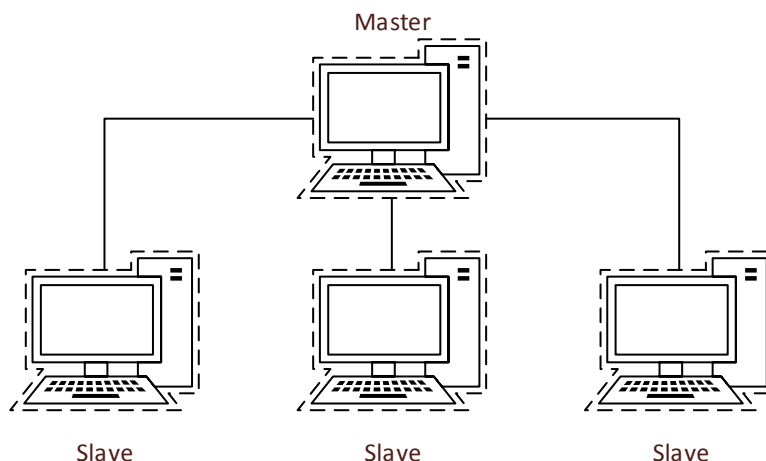
A Slave szolgáltatást nem a felhasználó, hanem a Masterben megvalósított kliens használja föl. Ennek használati eseteit egy külön use-case diagramon szemléltethetjük (4. ábra).



4. ábra. A Master felhasználási esetei a Slave szolgáltatáson

4.3. Architektúra

A rendszer fizikai felépítése szerint Master és Slave szoftverekre tagolható. Ezek külön (virtuális) gépen helyezkednek el. A Master kommunikál minden Slave-vel, a Slave-ek egymás között viszont nem. Ezt az architektúrát mutatja be az 5. ábra.



5. ábra. A Mastert és Slave-et futtató gépek hálózati elrendezése

4.3.1. Master

A Master a központi vezérlőszoftver, fő feladatai a következők:

- Slave-ek kezelése
- Feladatok ütemezése, Slave-ek közötti elosztása
- Eredmények, kimenetek létrehozása, rendszerezése

Indítás, üzemeltetés

A szoftver egy parancssori program, a bemeneti paramétereket indításkor kapja meg, futás közben nem vezérelhető, csak kimenetet ad, a feladatai befejeztével leáll.

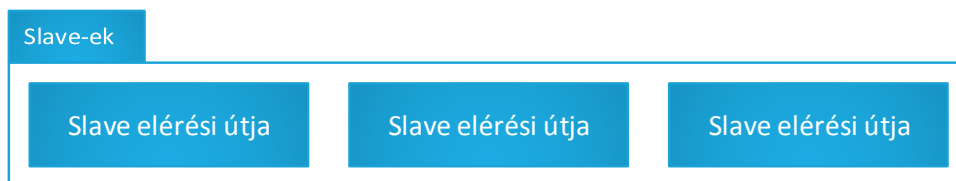
Bemenet

A master bemenete egy parancssori paraméterként megadott JSON fájl. A fájl tartalmazza a Slave-ek listáját, illetve a kampányt (6. ábra)



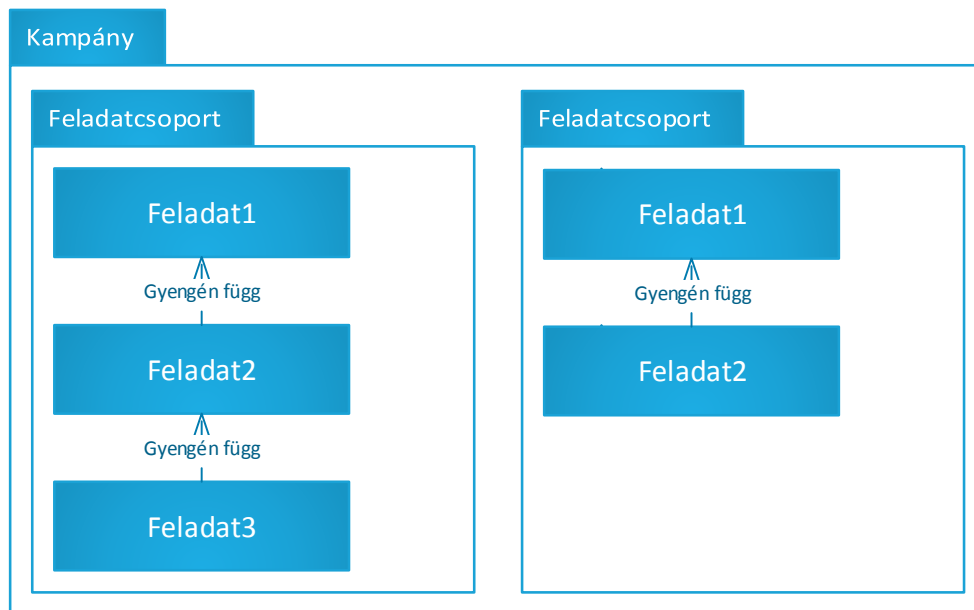
6. ábra. A bemeneti fájl két fő szekciója

Egy Slave-et az elérési útjával kell megadni, a Slave-ek megadása egy elérési út lista (7. ábra).



7. ábra. A slave szekció részletei

A kampányban feladatai közötti függőségeket nem explicit módon kell megadni, a gyenge függőség fogalma egy absztrakcióval a felhasználó elől elrejtésre kerül. Helyette a feladatokat feladatcsoportokba rendezve lehet megadni, ahol egy feladatcsoport egymástól gyengén függő feladatok láncá (8. ábra). Egy adott feladat mindig gyengén függ a csoportban közvetlenül előtte felsorolt feladattól, emiatt a feladatok sorrendben kerülnek elindításra, illetve egy feladat hibás befejezése visszavonja a csoportban utána következő feladatokat. A feladatcsoport fogalma csak a Master bemenetében jelenik meg.



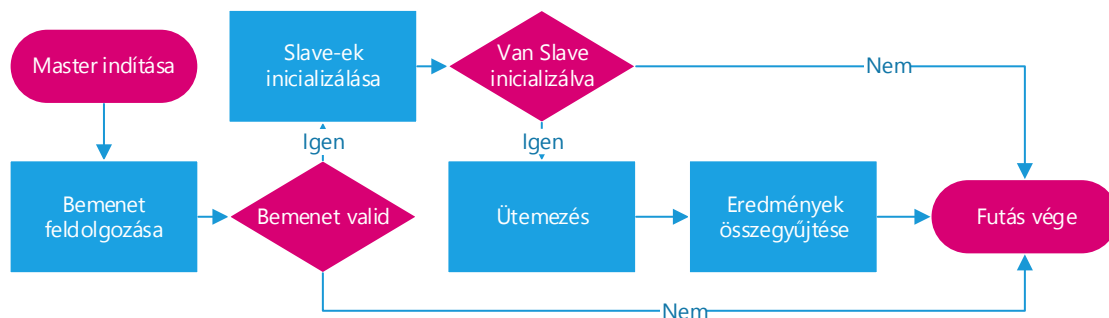
8. ábra. Kampány szekció részletei

Gyenge függőséget más módon (például explicit), nem lehet megadni. Ez meggátolja a helytelen függőségek megadását (például körkörös függőség). Erős függőséget egyáltalán nincs mód megadni a jelenlegi verzióban.

Egy feladatcsoport feladatai egy környezetben futnak, a feladatcsoportra közösen lehet megadni, hogy milyen bemenetet használhatnak a feladatai. A pontos bemenet részletezésre kerül a 6. és 7. fejezetekben.

Futási folyamat

A Master futási folyamata jól elkülöníthető lépésekre bontható (9. ábra).

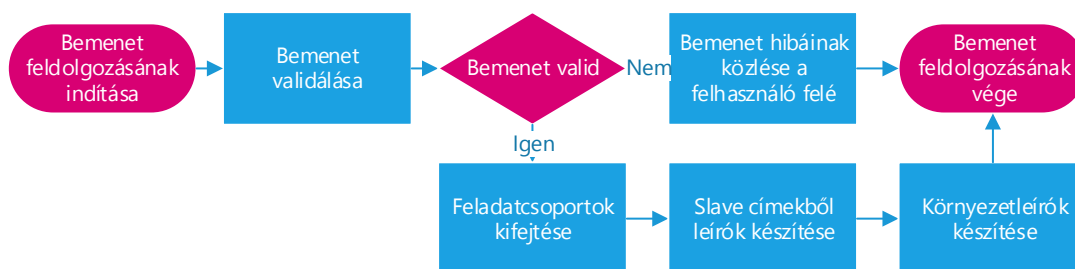


9. ábra. A Master működésének magas szintű folyamatábrája

Bemenet feldolgozása

A bemenet feldolgozása a következő feladatokból áll (10. ábra).

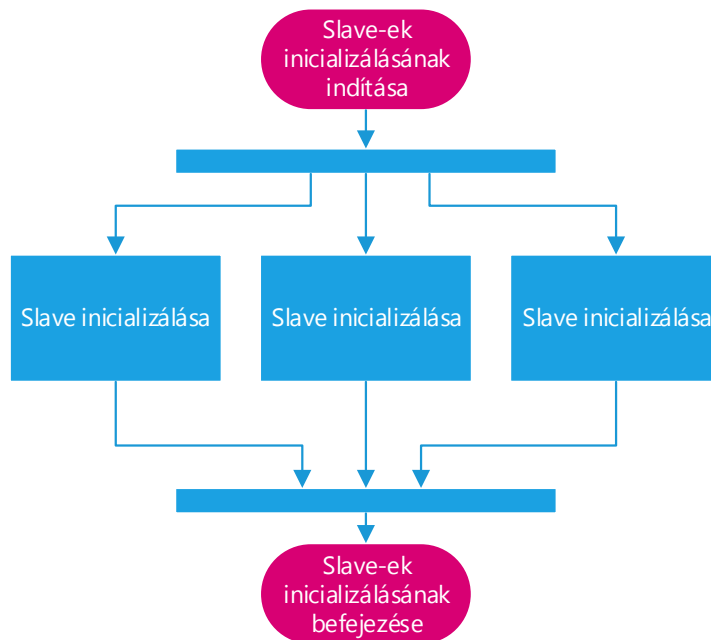
1. Validálni kell formailag és tartalmilag a bemenetet.
2. Ki kell fejteni a feladatcsoportokat. A feladatcsoportok feladatai között fel kell venni a függőségeket és a feladatcsoport tulajdonságait egyenként értelmezni kell a feladatokra.
3. A Slave címekből a Slave-et reprezentáló objektumokat kell készíteni.
4. Össze kell állítani a környezetek leíróit, tehát, hogy milyen környezetben milyen fájl-bemeneteknek kell megtalálhatónak lenni.



10. ábra. A bemenet feldolgozásának folyamatábrája

Slave-ek inicializálása

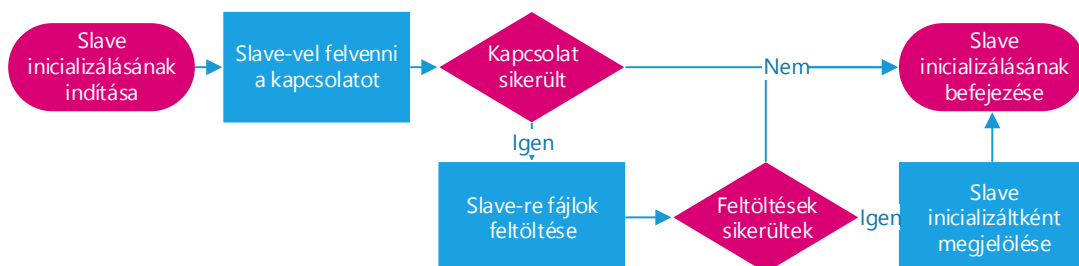
A Slave-ek inicializálása párhuzamosan történik (11. ábra). Az inicializálások végén a program az összes Slave-et bevárva szinkronizál, és csak ezután kezd neki az ütemezésnek.



11. ábra. Több Slave párhuzamos inicializálásának folyamatábrája

Egy Slave inicializálása a következő két főbb feladatra bontható fel (12. ábra).

1. A Slave-ekkel fel kell venni a kapcsolatot, biztosítva, hogy elérhetőek.
2. A Slave-ekre az összes környezet összes fájlját át kell másolni. A Slave képes legyen eldönteni egy fájlról, hogy megtalálható-e a környezetben, és ha igen, a másolást hamarabb megszakítani.

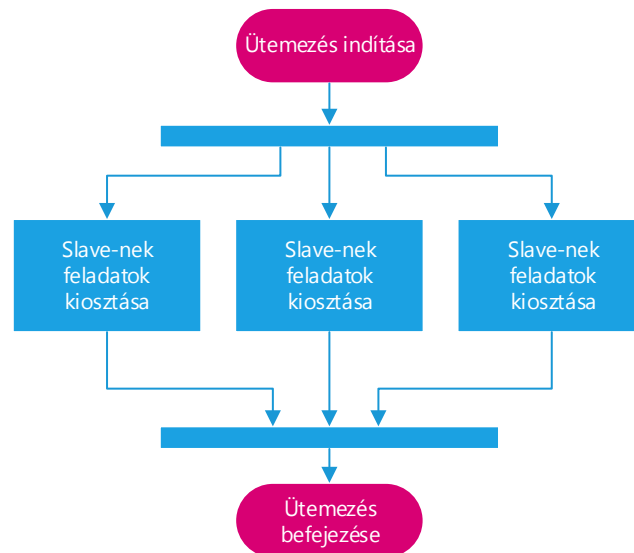


12. ábra. Slave inicializálásának folyamatábrája

Az inicializáció végén, ha egy Slave-et sem sikerült rendben inicializálni, az ütemezést a program nem kezdi el.

Ütemezés

A Slave-ek számára a feladatok kiosztása párhuzamosan történik (13. ábra). Minden Slave addig lesz igénybe véve, amíg van futtatásra várakozó feladat. Ha a feladatok elfogytak, az ütemező megvárja, amíg minden Slave végezt a feladatával, majd befejezi működését.

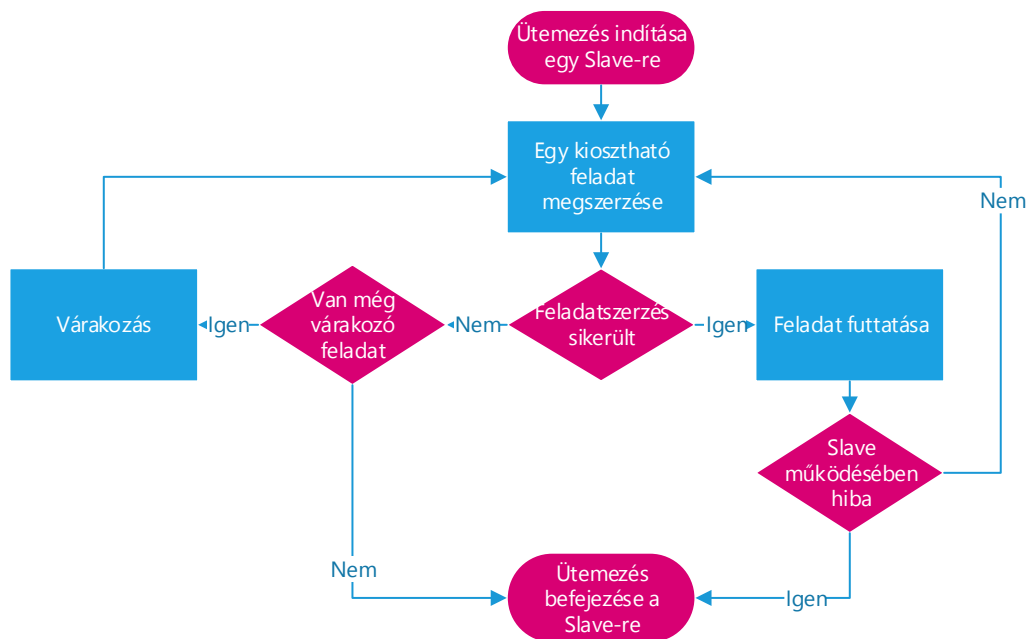


13. ábra. Az ütemezés magas szintű folyamatábrája

Egy Slave-nek feladat kiosztásakor a következő szempontokat figyelembe kell venni.

- Amint egy Slave felszabadul és van futtatására kész feladat, azt ki kell neki osztani.
- A függőségek adta megkötéseket befolyásolhatják egy feladat kioszthatóságát. Egy feladat nem kiosztható, ha van még nem indított függősége. Ekkor olyan feladatot kell kiosztani, amelynek már nincs futás alatt álló gyenge függősége.
- Visszavont feladatot nem szabad kiosztani, ki kell venni az ütemezésből.
- Ha egy Slave számára nincs kiosztható feladat, a Slave várakozzon, amíg új feladat nem válik kioszthatóvá.
- Ha egy Slave-vel megszakad a kapcsolat vagy működésében hiba lép fel, az ütemezés további részéből ki kell hagyni. Ha a futtatott feladat hibázik, az nem a Slave működésében hiba, ilyenkor a Slave továbbra is használható.
- Ha minden feladat befejezésre, vagy visszavonásra került, az ütemezést be kell fejezni.
- Ha több Slave áll készen feladat futtatására, akkor nem definiált, hogy melyik kapja a feladatot.

Az ütemezés egy Slave-re vetített folyamatábráját mutatja a 14. ábra. A diagram egy egyszerűsített folyamatot ábrázol. A hibakezelés és a párhuzamosítás optimális megvalósítása miatt a megvalósított folyamat ettől kissé eltérhet.

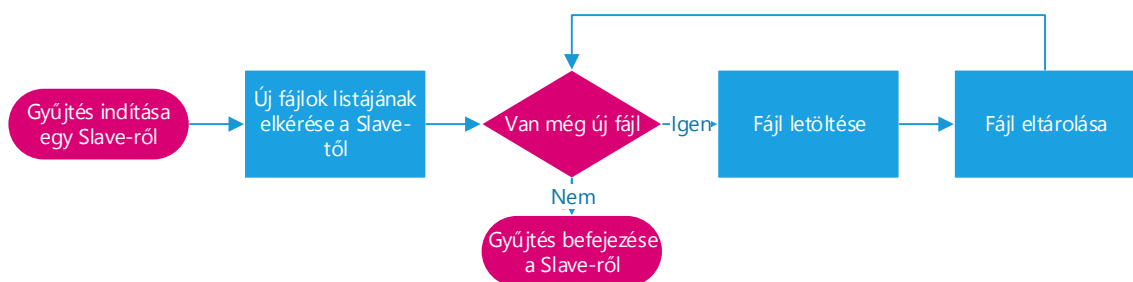


14. ábra. Ütemezés egy Slave-re vetített folyamatábrája

Eredmények összegyűjtése

Minden Slave minden lefuttatott feladatának kimeneteit a Mastert futtató számítógép fájlrendszerében kell eltárolni. Ez Slave-enként párhuzamosan történik. A párhuzamosítás folyamatábrája az előző két lépésével megegyezik.

Slave-enként az eredmények összegyűjtésének folyamata egyedül a Slave-ek környezetének újonnan létrejött fájljai lemásolásából és eltárolásából áll (15. ábra).



15. ábra. Eredmények gyűjtése egy Slave-ről

Kapcsolat a Slave-ekkel

A Master a Slave-eket explicit parancsokkal vezérli (pl. fájlküldés, fájlok letöltése, megadott parancssori feladat futtatása). A kapcsolat jellege a Slave implementációtól függ, de a vezérlés interfésze általános.

A Slave vezérlése a Master forráskódján belül is csak egy általános interfészen keresztül történik. Ez az interfész elfedi a hálózati kapcsolatnak, a kommunikáció módjának részleteit, így a kapcsolat implementációja lecserélhető.

Az inicializálás, az ütemezés, illetve az eredmények összegyűjtése is ezt az interfészt használja, az interfészen minden művelet megtalálható.

Kimenet

Az alkalmazásnak többféle kimenetet kell létrehoznia, a monitorozás, az eredmények megtekintése, illetve a hibakeresés segítésének érdekében. Az alábbi kimeneteket különböztetjük meg.

Eredmények

A feladatok összegyűjtött eredményei egy mappában, érthető struktúrában. Ezen struktúra leírása a felhasználói útmutatóban kerül kifejtésre.

Futási napló

A Master szolgáltatás futásainak részletes naplója. A futások a naplóban elkülönülnek. A futási napló tartalmazza a Master műveleteit, például: melyik Slave kerül inicializálásra, melyik Slave-nek melyik feladat kerül kiosztásra, melyik feladat sikerül, melyik nem, melyik kerül visszavonásra, melyik Slave-vel szakadt meg a kapcsolat, vagy melyik Slave-ről kerülnek az eredmények összegyűjtésre. A napló historikusan megőrzésre kerül, hogy utólag elemezhető legyen.

Hibanapló

A futási naplóhoz hasonló, de csak a hibákat tartalmazó napló. Hiba lehet a keretrendszer működésében fellépő hiba (hibás állapot, kapcsolati hiba a Slave-vel), vagy egy feladat hibája (feladat kivételt dob, hibás értékkel tér vissza, túllépi a kiszabott időt). A napló historikusan megőrzésre kerül, hogy utólag elemezhető legyen.

Státusz fájl

A fájl célja, hogy a rendszer monitorozását megkönnyítse. Tartalmazza a Master aktuális állapotát a futási folyamat szerint, a Slave-ek listáját, ahol feltüntetésre kerül a Slave aktuális tevékenysége, valamint a feladatok listáját, ahol feltüntetésre kerül a feladat aktuális feldolgozási állapota. A fájlban észrevehetően jelzésre kerül, ha hiba van.

A fájl egy egyszerű felhasználói felületet hivatott kiváltani, ha a keretrendszerhez felhasználói felület készül, ez a fájl megszüntethető.

4.3.2. Slave

A Slave egy hálózati szolgáltatás, amit a Master vezérel távoli parancsokkal. A Slave konfigurálásának és indításának módja nem megkötött, megvalósítástól függ.

Slave interfész műveletei

A Slave szolgáltatás interfészén az alábbi műveletek szerepelnek.

Elérhető fájlok listázása egy adott környezetben

A kliens ezen a műveleten keresztül el tudja kérni az adott környezetben elérhető összes fájlt, ez alapján ellenőrizheti, hogy milyen fájlokat kell még elküldenie az inicializálás során, vagy milyen fájlokat kell letöltenie az eredmények összegyűjtése során.

Fájl feltöltése egy adott környezetbe

Fájl feltöltésénél megadható a relatív útvonal, amivel a környezeten belüli mappastruktúra felépíthető. A mappákat nem szükséges létrehozni, ezek automatikusan jönnek létre a fájlok alapján. Ha a fájl már megtalálható a környezetben, a küldés megszakad. A szolgáltatás képes kell legyen a Slave-et futtató gép elérhető memóriájánál nagyobb méretű fájlok fogadására is.

Fájl letöltése egy adott környezetből

A feltöltésnél hasonlóan itt is a környezeti mappához képesti relatív útvonalat szükséges megadni. A szolgáltatás képes kell legyen a Slave-et futtató gép elérhető memóriájánál nagyobb méretű fájlok küldésére is.

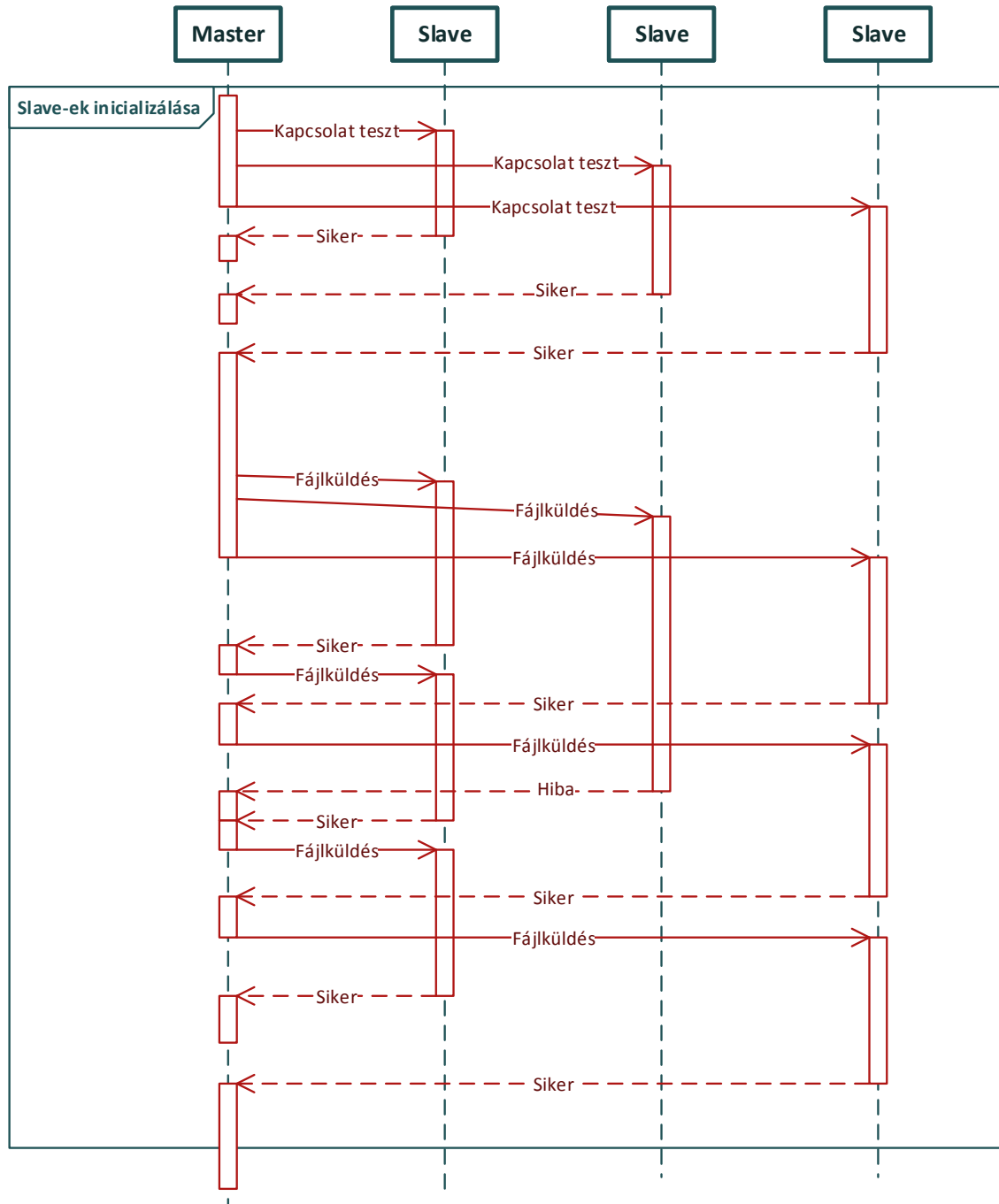
Feladat futtatása egy adott környezetben

A feladat futtatásánál meg kell adni a feladat paramétereit és az időkorlátot. A feladat futtatása aszinkron módon megtehető, az indítás nem kell, hogy blokkolja a kliensben a kódvégrehajtást.

4.4. Tipikus szekvenciák

A felhasználó az alkalmazást nem szeretné aktívan monitorozni. Helyette, ha az alkalmazás futása közben elolvassa a státusz fájl aktuális állapotát, abból látnia kell, hogy az alkalmazás éppen milyen műveletet végez. A következő szekvencia diagramon (16. ábra) az kerül szemléltetésre, hogy a felhasználó elindítja az alkalmazást, otthagyja, és a státusz fájlra néz csak rá ritkán, hogy leellenőrizze, hogy az alkalmazás min dolgozik, a feladatok véget értek-e már. Az ábrán a státusz fájlban végzett műveletek csak az alkalmazás egyetlen állapotváltozójának státusz fájlba írására és a fájlból elolvasására vonatkoznak.

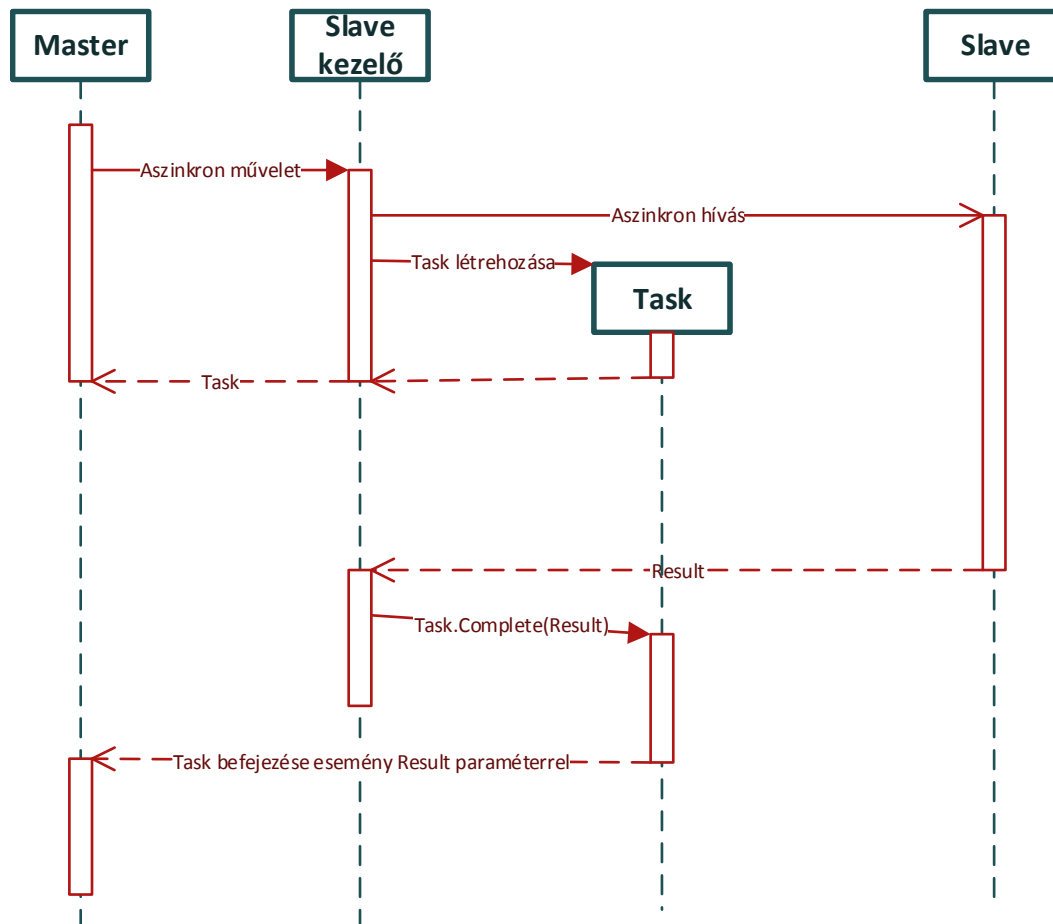
egy parancs kiadása után az eredményre várakozik, de közben egy másik befejezett parancs aktiválja. Például három Slave inicializálásának folyamata alakulhat a 17. ábra szerint, ha Slave-enként összesen három fájl szükséges elküldeni.



17. ábra. A Slave-ek inicializálásának tipikus szekvenciája

Az ábrán látszik, hogy az egyik Slave-re a fájlküldés megghiúsul, ennek a Slave-nek az inicializációja megszakad. Látszik továbbá, hogy a Mastert az aszinkron műveletek visszahívása aktiválja, ha elfogy a feladata, inaktív állapotban várakozik.

Az ábrán elhagyásra került az az implementációs részlet, hogy a Master inicializációért felelős komponense célszerűen különálló résztvevőként szerepel. Ha az ábrán ezek a Slave-eket kezelő szereplők is megjelentek volna, az ábra komplexitása jelentősen megnő. Nézzünk egy példát, hogy néz ki egyetlen aszinkron hívás a Slave felé a Slave-kezelő szereplőn keresztül, feltételezve, hogy az aszinkronitás a Technológiák fejezetben ismertetett TAP mintában használt Task objektumokkal kerül megvalósításra (18. ábra).

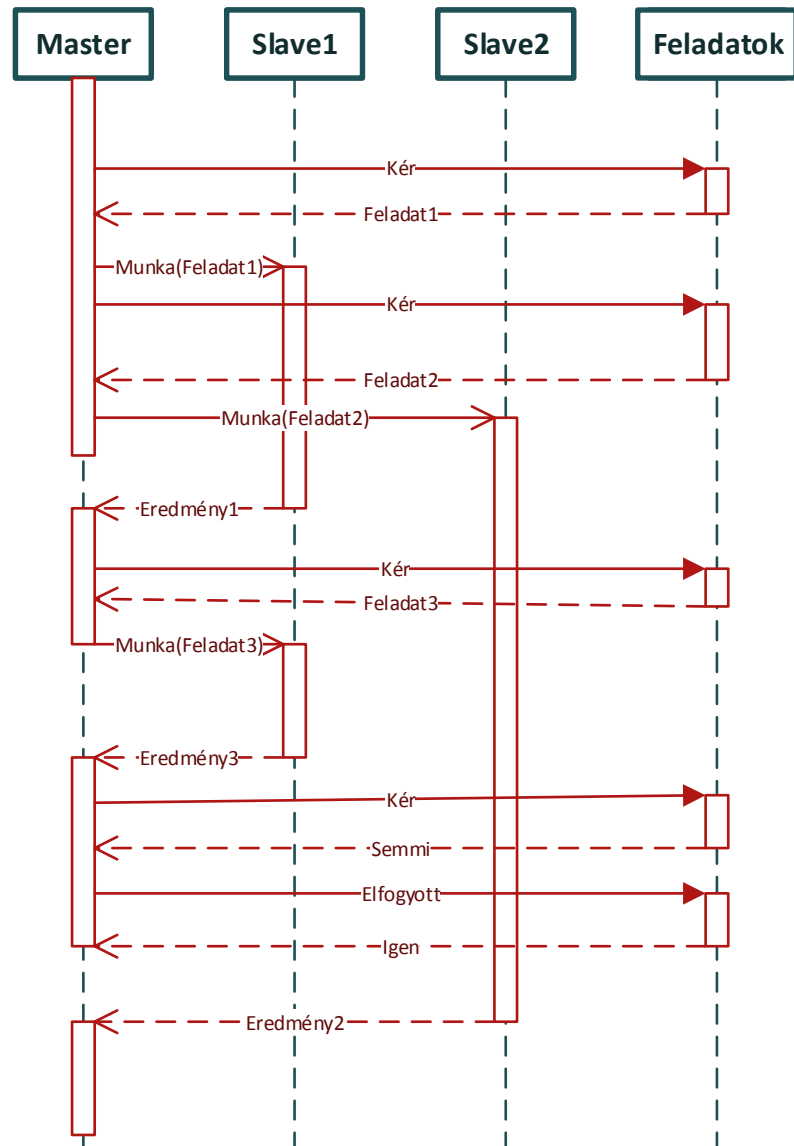


18. ábra. Aszinkron művelet kifejtése Slave kezelő komponenssel

Az ábra alapján érthető, hogy ha a 17. ábra esetében minden aszinkron művelet külön kifejtésre kerül, a szekvencia kezelhetetlenül hosszú lenne és túl sok szereplőből állna. Innentől a többi ábrán is a fenti minta helyett egyszerű Slave felé intézett aszinkron művelet kerül feltüntetésre.

A következő példában tekintsünk meg egy ütemezést a két sikeresen inicializált Slave-ünkre. Tegyük föl, hogy 3 egymástól gyengén függő feladatunk van és ezeket egy olyan adatstruktúrán keresztül kapjuk meg, ami kérésre képes egyetlen futtatásra kész feladatot adni, vagy üres eredménnyel tér vissza, ha egy feladat sem futtatásra kész. Egy másik kéréssel megtudható, hogy maradt-e egyáltalán feladat. Ha a feladatok már elfogytak, az ütemező szinkronizál, a Slave-ek eredményeit megvárja. Ha a feladatok még nem fogytak el, úgy a Slave várakozik (adott ideig, vagy olyan eseményre, ami megváltoztathatja a

feladatok futtathatóságát). Mivel a gyenge függőség a második esetet nem teszi lehetővé, így az ábrán is csak az első eset kerül feltüntetésre (19. ábra)



19. ábra Az ütemező egy tipikus szekvenciája

5. Felhasznált technológiák

A következő fejezetben azokat a megvalósításhoz felhasznált technológiákat ismertetem, amik vagy a projekt szempontjából jelentősek, vagy a megvalósításhoz fontos volt a részletes megismerésük

5.1. C# nyelv, .NET platform

A C# [5] (ejtsd: szí-sárp) a Visual Basic mellett a .NET fő programozási nyelve. 1999-ben Anders Hejlsberg vezetésével kezdték meg a fejlesztését.

A C# tisztán objektumorientált, típusbiztos, általános felhasználású nyelv. A tervezésénél a lehető legnagyobb produktivitás elérését tartották szem előtt. A nyelv elméletileg platform független (létezik Linux és Mac fordító is), de napjainkban a legnagyobb hatékonyságot a Microsoft implementációja biztosítja.

Maga a .NET platform a Microsoft, a Hewlett Packard, az Intel és mások közreműködésével megfogalmazott CLI (Common Language Infrastructure) egy implementációja. A .NET nem programozási nyelv, hanem környezet. Gyakorlatilag bármelyik programozási nyelvnek lehet .NET implementációja. Jelenleg kb. 50 nyelvnek létezik hivatalosan .NET megfelelője, nem beszélve a számtalan hobbifejlesztésről. [6]

5.2. WCF

A Windows Communication Foundation [7] a Microsoft összevont programozási modellje szolgáltatásorientált alkalmazások építéséhez. A fejlesztők számára lehetővé tesz biztonságos, megbízható, tranzakciókat betartó megoldások létrehozását, amik különböző platformokon keresztül integrálódnak, és együttműködnek létező szolgáltatásokkal. Egységes programozási modellt, általános absztrakciót biztosít szolgáltatások és a hozzájuk tartozó klienseinek építéséhez.

Egy WCF szolgáltatás protokollok széles skálájára építhet, nincs rögzítve a kommunikáció módja. Egyes protokollok támogatása a modellel együtt a .NET keretrendszer részeként jelen van, de mások támogatása is implementálható. A keretrendszer jelenleg például HTTP protokoll fölött SOAP üzenetekkel kommunikál, de ez az igények megváltozása esetén konfigurációval változtatható.

Fontos, hogy a szolgáltatás kliens és szerver oldala építhet teljesen más technológiákra, amennyiben a kommunikáció formájában megegyeznek. Egy WCF szolgáltatással nem csak WCF kliens kommunikálhat, és egy WCF kliens nem csak WCF szolgáltatáshoz lehet képes csatlakozni. Ez a keretrendszerünkben is szabadságot jelenthet, ha különböző szoftverkörnyezetek támogatására van szükség.

5.2.1. ServiceModel Metadata Utility Tool

A ServiceModel Metadata Utility eszköz [8] képes WCF szolgáltatásokhoz klienst generálni a szolgáltatás által publikált metaadatok alapján. Ez nagyban megkönnyíti egy WCF kliens fejlesztését, illetve a szolgáltatás interfészének karbantartását.

A keretrendszerben használt WCF kliens teljes mértékben a szolgáltatás metaadatai alapján került generálásra. A generált kliens a szolgáltatás aszinkron használatára is képes, így külön fejlesztés nélkül felhasználható a később ismertetett TAP mintával implementált kódban.

5.3. Task-based Asynchronous Pattern (Task alapú aszinkron tervezési minta)

A keretrendszerrel szemben támasztott egyik legfontosabb elvárás az, hogy képes legyen a megadott feladatokat párhuzamosan, több Slave-en elvégezni. A feladatok kiosztása, a feladat elindítása még a Masteren történik, a feladat eredményének feldolgozása szintén. Miközben egy feladatot futtatunk egy Slave-en, az eredményére várakozás nem blokkolhatja a többi feladat futtatását, így nem történhet a Master központi ütemezőjében szinkron módon.

Egy kódrész vagy metódus aszinkron futtatásához a .NET keretrendszerben változatos lehetőségeink vannak, például inicializálhatunk egy új szálat, és a metódust futtathatjuk az adott szálon. Nyilván a lefutás végéről értesülnünk kell valahogyan, hiszen a metódus eredményeit csak a sikeres futás után használhatjuk föl. Ez történhet valamilyen várakozással (`Thread.Join` [9] függvénnyel például egy szálon megvárhatjuk egy másik szál futásának befejezését), vagy a befejezésről értesülhetünk egy callback függvényben vagy eseményben is (`Action.BeginInvoke(AsyncCallback callback, object object)` függvénynek például a callback paramétere akkor kerül meghívásra, ha a művelet befejeződött).

Általánosan elmondható, hogy bármilyen módszert választunk műveletek aszinkron futtatására, az aszinkron kód komplexitása nagyobb lesz a szinkron megfelelőjéhez képest. Elég néhány egyszerűbb vezérlési szerkezetet kombinálnunk (műveletek aszinkron meghívása ciklusban, aszinkron műveletek egymásba ágyazása) és a kód könnyen áttekinthetlenné, érthetlenné válik, de ha mégsem, az aszinkronitás akkor is a kód meghatározó része lesz.

5.3.1. Alternatívák

Az aszinkron tervezési minták között több olyan is van, amiket a Microsoft ajánl, illetve amelyeknek a beépített osztálykönyvtárak is megfelelnek. Ilyen tervezési minták például az Asynchronous Programming Model [10] és az Event-based Asynchronous Pattern [11], azonban a Task-based Asynchronous Pattern (TAP) [12] az „ajánlott tervezési minta az új fejlesztésekhez”.

5.3.2. Előnyök

Ennek a mintának számos előnye van.

- Az aszinkron kód struktúrája nagyon hasonló a megfelelő szinkron kódhoz. Nincs szükség a kódot például callback függvényekre tagolni, vagy minden függvényhívást egy `*Begin` és `*End` függvénypárra cserélni. Az ezzel a mintával megvalósított a kód olvasható.

- Absztrakt. Az aszinkronitás tényét nem mossa össze az aszinkronitás megvalósításának módjával. Első megközelítésben teljesen mindegy, hogy a háttérben milyen szálak készülnek, például ThreadPoolt használ-e a kódunk, vagy új szálakat példányosít, vagy egyáltalán készülnek-e szálak, nem ez alapján szeretnénk az alkalmazást tervezni. Egy konzolalkalmazásnak, UI alkalmazásnak, vagy Webalkalmazásnak például teljesen más szálakezelési igényeik vannak, nem árt, ha a tervezési minta ezeket a különbségeket egy alacsonyabb absztrakciós szinten valósítja meg (esetünkben ez a .NET keretrendszer, a WPF és az ASP.NET része, ezekben keretrendszer szinten megvalósításra került).
- Egyszerű hibakezelés. Lehetőségünk van egyszerűen try-catch blokkokkal az aszinkron programrész kivételeit kívülről elkapni. Ez callback-ek és begin-end függvénypárok esetében ritkán megvalósítható.
- .NET 4.5 óta nyelvi elemként megjelent az async/await szintaktikai édesítőszer [13]. Ez nagyban megkönnyíti mind a kódstrukturálást, mind a hibakezelést, mind a kód absztrakciót.

5.3.3. Hátrányok

Az async/await új nyelvi elemként jelenik meg. Amíg az egyéb tervezési minták a nyelv egy régebbi verziójának ismeretével pusztán a függvények szignatúrája alapján megérthetőek, addig itt az új nyelvi elemeknek utána kell olvasni, meg kell őket érteni, hiszen a kód futtatásának sorrendje teljesen új koncepciók szerint változik.

Természetesen lehetőség van az új szintaktikai édesítőszert kihagyva implementálni a függvényeinket, viszont az így kapott függvények szignatúrája és tartalma, az implementálandó állapotgépek, jelentősen bonyolultabbak ahhoz képest, mintha az új kulcsszavak megértése és használata. [14]

Az async „ragályos betegség”. Ahhoz hogy a kód hatékony legyen, az aszinkron függvényeket felhasználó függvényeket is érdemes aszinkronként implementálni, teljesen áthatva ezzel az alkalmazás vezérlési logikáját. Nem érdemes a kódot „csak egy kicsit aszinkronra” tervezni. [15]

5.3.4. TAP minta megismerése

Az MSDN cikkeken kívül a hivatalos Task-based Asynchronous Pattern dokumentum [16] részletes és érthető segítséget nyújt a témában, ennek elolvasása bőven elégséges a keretrendszerben használt aszinkronitás működésének megértéséhez.

5.4. NLog

Az NLog egy naplózást segítő platform a .NET keretrendszerhez [17]. Bővíthető felépítésének és elterjedtségének köszönhetően gyakorlatilag a fejlesztett keretrendszer minden naplózási igényét lefedi. A következő pontokban a platform egyes igénybe vett funkcióit írom le.

- *Naplózás fájlokba.* Az alkalmazás kiemelt eseményekről naplóbejegyzéseket készít. Ezek az események lehetnek hibák, vagy a futás várt részei (például egy feladat kezdete). Az így létrehozott napló fájlokban kerül eltárolásra.
- *Hibatűrő naplózás.* Naplózás esetében problémát jelenthet, ha a naplózás sikertelensége megakadályozza az alkalmazás működését, például fájlba naplózáskor, ha valamilyen más alkalmazás a fájlt zárolja. Esetleg kívánatos lehet, hogy a naplóbejegyzés kihagyásával ugyan, de az alkalmazás tovább működjön. Ez az NLog esetén beállítás kérdése.
- *Külön hibanaplók.* Naplózási szint szerint különböző naplófájlok kerülhetnek elválasztásra, a keretrendszer esetében például csak a hibákról külön napló készül.
- *Naplózás konzolra.* Bár konzolkimenet készítése nem egy összetett feladat, a kimenet általános formázásában, a kimenet színezésében az NLog segítséget nyújt.
- *Naplóbejegyzés formázása.* Napló típustól függően a bejegyzések formátuma fix lehet. Esetleg feltüntetendő a pontos idő, a naplóbejegyzés eredetének helye, a naplóbejegyzés naplózási szintje. Praktikus, ha kódolás közben a naplóbejegyzés leírásakor ezekkel a konvenciókkal nem kell törődnünk, helyette ez egy központi helyen definiálható. Az NLog lehetőséget ad a formázásra, a formázásban felhasználható adatok bővíthetők. A keretrendszer esetében például egyes konzolkimeneteken az implementációs részleteknél ismertetett `TimingLoggerLayoutRenderer` osztály segítségével feltűnik a keretrendszer indítása óta eltelt idő.
- *Egy naplóbejegyzés több naplóba.* Segít, hogy ugyanazt a naplóbejegyzést konfigurációtól függően több naplóban is eltárolhatjuk úgy, hogy a különböző naplóokban elvárt, különböző formázását is betartjuk.
- *Naplók archiválása.* Egyes esetekben a régi naplókat szeretnénk kategorizálni és elkülöníteni az aktuálisaktól. A keretrendszer esetében például egyes naplók csak az aktuális, vagy legutóbbi futás bejegyzéseit tartalmazzák. Az előző futások naplói átnevezésre és átnevezésre kerülnek.
- *Fájl egyetlen, folyamatosan frissülő bejegyzéssel.* A keretrendszerben a monitorozás megkönnyítésére készül egy státusz fájl. A fájlt az OpenVPN szoftver routing táblája ihlette, lényege, hogy a fájl tartalma egy pillanatkép a keretrendszer aktuális állapotáról. Az állapot megváltozásakor a fájl egész tartalma fölülírásra kerül. A platform erre is lehetőséget ad, bár a funkcióval kapcsolatban a keretrendszer fejlesztése közben kiderült egy hiba, ez jelentésre került [18] és a következő NLog verzióban ki lesz javítva. Addig egy workaround kerül alkalmazásra.

5.5. JSON

A JavaScript Object Notation [19] egy „pehelysúlyú” adatátviteli formátum. Emberek által egyszerűen írható és olvasható, gépek által egyszerűen feldolgozható és generálható. Ugyan a JavaScript programozási nyelv egy részén alapszik, egy olyan szövegformátum, ami nyelvfüggetlen, de a C nyelvcsaládra jellemző konvenciókat használ, így ismerős lehet a C, C++, JavaScript, Perl, Python vagy egyéb C jellegű nyelvvel ismerő programozóknak.

5.5.1. Formátum rövid ismertetése

Referenciának lássuk az alábbi példát. A példa leírása alább olvasható.

```
{
  "gyumolcs": "alma",
  "darabszam": 25,
  "csakFriss": false,
  "szin": {
    "R": 225, "G": 30, "B": 30
  },
  "telefon": [
    {
      "nev": "mentok",
      "tipus": null,
      "szam": "01189998819991197253"
    },
    {
      "nev": "otthon",
      "tipus": "vonalas",
      "szam": "112 - 233 - 4455"
    }
  ]
}
```

A gyökérelemünk egy névtelen objektum. A JavaScript objektumokhoz hasonlóan a { és } határolókarakterek között definiáljuk az objektum tulajdonságait.

A tulajdonságok kulcs-érték párok. A kulcsot dupla idézőjel között kell megadni. A kulcsot és az értéket : karakter választja el. A kulcs speciális karaktereket is tartalmazhat, de a maximális kompatibilitás miatt érdemes az angol ABC kis- és nagybetűit használni. A tulajdonságokat vessző választja el.

A határolókarakterek közötti térköz karakterek száma, az elrendezés szabadon változtatható (de nyilván emberi szerkesztés esetében praktikus egy konvenciót betartani).

A tulajdonságok típusa lehet:

- Szám: a pontos támogatott formátum feldolgozófüggő, de az egész számok mindenhol támogatottak.
- String: idézőjelek között kerül megadásra.
- Boolean: a két megengedett érték **true** és **false**.
- Objektum: az érték szintaktikailag helyes JSON.
- Tömb: [és] határolókarakterek között, vesszővel elválasztott értékek. Az értékek típusa az itt felsoroltak közül bármelyik lehet.
- null: Üres érték.

Az objektum és tömb típusokkal hierarchiát definiálhatunk.

5.5.2. Bemeneti formátum

A JSON formátum került kiválasztásra az alkalmazás bemeneti formátumának. Ennek oka többek között az XML-lel szemben az, hogy a formátum meglehetősen kompakt, egy

bemeneti fájl kevés karakterből áll. Cserébe viszont hasonlóan (bár talán kevésbé) elterjedt és ismert, mint az XML formátum.

Az egyszerűbb formátumokkal szemben (például INI formátum) az XML-hez hasonlóan támogat összetett objektumokat, listákat, ami a keretrendszer esetében szükséges. Támogat még adattípusokat, bár ez a keretrendszerben nem kerül kihasználásra.

A bemeneti fájlformátum bármikor lecserélhető, a rendszer fejlesztése során fontos szempont volt, hogy a JSON formátum csak a keretrendszer elhanyagolható részének (csak a bemenetet kezelő osztálynak) legyen függősége.

A bemenet deszerializálását a Json.NET [20] könyvtár segítségével oldottam meg.

6. Részletes tervek

A következő fejezetben pontosan ismertetésre kerül a megvalósított rendszer, annak komponensei, valamint a komponensek belső osztálystruktúrája. Részletezésre kerül az osztályok felelősségi köre, feladatai, és megvalósításuk részletei. Ismertetésre kerülnek továbbá a bemeneti és kimeneti interfészek és formátumok, amiken keresztül a felhasználó a rendszert vezérelheti és megfigyelheti.

Bár az alkalmazás alapvetően feltagolható Master és Slave komponensekre, következő alfejezetek nem követik ezt a felosztást. Helyette a Slave komponens egyetlen alfejezetben szerepel, az összes többi alfejezet pedig vagy általános, vagy a Master komponensre vonatkozik.

6.1. Programozási nyelvek

Mind a Master mind a Slave szoftver C# programozási nyelven készült. A nyelv választásánál a két fő szempont a következő volt:

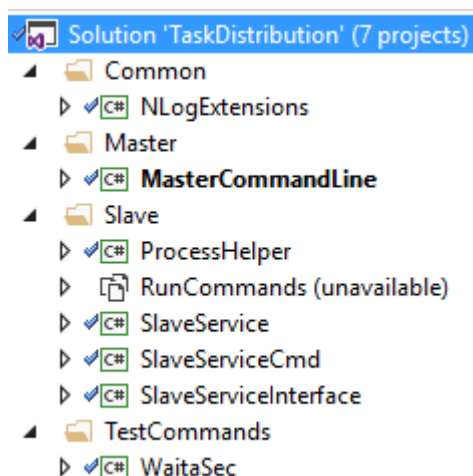
- A projekt jelenlegi és potenciális fejlesztői számára jól ismert legyen. A mérnök informatikus képzés során a C# nyelvvel sokszor találkoznak a hallgatók.
- A nyelv elég erős legyen komplex rendszerek tervezéséhez.

A Master bemenetének leírónyelve JSON formátumú. A formátum választásánál a főbb szempontok:

- Emberek által is olvasható és szerkeszthető egyszerű szintaxis.
- Támogatja a hierarchikus adatszerkezeteket.

6.2. Projektstruktúra

Az alkalmazás forráskódja a következő projektstruktúra szerint lett felbontva (20. ábra).



20. ábra. A forráskód projektstruktúrája

Értelemszerűen a Master mappában a Master komponens projektje, a Slave mappában a Slave komponens projektjei találhatóak. A Common mappa az általános projekteket tartalmazza, a TestCommands mappa pedig egy konzol alkalmazást tartalmaz a keretrendszer teszteléséhez.

Bár a Slave mappa több projektet tartalmaz ennek pusztán annyi az oka, hogy fejlesztés közben a Slave szolgáltatásnak, illetve a Slave szolgáltatás gazdafolyamatának több különböző prototípusa került tesztelésre és a több prototípusban is közösen felhasznált kód külön projektbe került kiemelésre. Ezzel szemben a Master alkalmazás egyetlen projekt-ként fejlődött így a föltagolást nem volt szükséges megtenni.

6.3. Bemenet

A program bemenete egyetlen JSON fájl. Egy rövid bemeneti fájl tartalma lehet például a következő:

```
{
  "slaves": [
    "http://127.0.0.1:5555/slave"
  ],
  "taskGroups": [
    {
      "environmentIdentifier": "environmentName", //optional
      "uploadFolder": "../SomeFolder/Upload", //optional
      "tasks": [
        {
          "command": "runthis.exe",
          "args": "input.txt -t 0", //optional
          "inUploadFolder": true, //optional
          "count": 10, //optional
          "timeout": 120
        }
      ]
    }
  ]
}
```

A bemenet két fő szekcióra osztható.

- **slaves** (kötelező): A Slave-ek listája, amiket a kampány futtatása során a Master felhasználhat.
- **taskGroups** (kötelező): A kampány feladatcsoportjai.

6.3.1. A slaves szekció

Egy string lista, elemei a Slave szolgáltatások elérési útvjai. A jelenleg implementált Slave-eknél ez HTTP URL-eket jelent.

6.3.2. A taskGroups szekció

A kampány feladatcsoportjai. A feladatcsoportok egymás után lefuttatandó feladatok láncát jelentik. Egy kampány akárhány feladatcsoportot tartalmazhat. A feladatcsoportok között futási sorrendet, függőséget nem lehet megadni.

- `environmentIdentifier` (opcionális): Egy szöveges azonosító, ami meghatározza, hogy egy adott Slave-en a feladatcsoport feladatai melyik környezetben fussanak. A környezet jelenleg egy mappát jelöl. A mappa útvonala nem közvetlenül ez az azonosító, viszont egy Slave-en egyértelműen származtatható az azonosítóból. Amennyiben nincs megadva, a Master a feladatcsoport számára véletlenszerű környezet azonosítót generál.
 - A feladatok kimeneteit jelenleg eszerint az azonosító szerint osztja mappákra a Master.
 - A környezet mappa a Slave-eken a Master futtatásai között is megmaradhat, amennyiben a mappa már létezik, a futások eredményei keveredhetnek.

A környezetek később részletesebben bemutatásra kerülnek.

- `uploadFolder` (opcionális): A Master gépen található mappa. A Slave-ek inicializálásakor ennek a mappának a tartalma átmásolásra kerül a feladatcsoportot futtató környezet mappába. A mappán belüli hierarchia megőrzésre kerül.
- `tasks` (kötelező): A feladatcsoporton belüli feladatok.

A tasks lista

A tasks lista tartalmazza a feladatcsoporton belüli feladatokat. A feladatok közötti sorrend számít, minden feladat gyengén függ a közvetlenül előtte felsorolt feladattól.

- `command` (kötelező): A futtatandó parancssori feladat. Nem tartalmazza a paramétereit.
- `args` (opcionális): A futtatandó feladat parancssori paraméterei. Összefűzésre kerül a `command` paraméterrel egy szóköz elválasztó karakterrel.
- `inUploadFolder` (opcionális): Amennyiben az értéke `true`, azt jelöli, hogy a `command`-ot a Slave-en a feladatcsoport környezet mappájához relatívan kell értelmezni. Potenciálisan akkor van erre szükség, amikor a `command` egy olyan futtatható fájlra mutat, ami a feladatcsoport `uploadFolder` tulajdonságában megadott mappával együtt kerül feltöltésre. Amennyiben az értéke `false`, úgy a `command` a Slave munkakönyvtárához képest relatív, vagy abszolút útvonalat jelöl.
- `count` (opcionális): Pozitív egész szám. Amennyiben megadásra kerül úgy a feladat a tulajdonságnak megfelelő számú azonos feladatokként értelmezendő. Pontosan azonos a jelentése azzal, mintha ennyiszer felsoroltuk volna a bemenetben ugyanazt a feladatot.
- `timeout` (kötelező): Pozitív egész szám. Az az idő másodpercben, amennyi ideig a feladatot maximálisan futtatni szeretnénk. A feladat ennél hosszabb futásnál megszakításra kell, hogy kerüljön. Ekkor a futása hibásként kerül megjelölésre, a függőségei pedig visszavonásra kerülnek.

6.3.3. InputHandler

Az osztály feladata a jelenlegi bemeneti formátumot elrejtteni az alkalmazás többi része előtt. A bemenetet deszerializálja, validálja, kiegészíti, az alkalmazás kezdeti állapotát felépíti a bemenet alapján, a bemeneti formátumot pedig az alkalmazás belső működéséhez szükséges osztályokra alakítja.

Bemenet deszerializálása

A bemeneti fájl tartalma egy string-ként kerül beolvasásra, a string közvetlenül bemeneti osztályokba kerül deszerializálásra. Ezek az osztályok (InputWrapper, CommandTaskInput, TaskGroupInput) a bemenetet reprezentálják, az InputHandler osztályon kívül máshol nem kerülhetnek felhasználásra.

Bemenet validálása

Ellenőrizni kell, hogy a bemenet összes kötelező tulajdonsága kitöltésre került-e, a kötelező listák legalább egy elemet tartalmaznak-e, a bemenetre vonatkozó megkötések (például a pozitív egész számok pozitívak) teljesülnek-e.

Amennyiben a bemenet nem érvényes, a futtatás hibaüzenettel leáll.

Bemenet kiegészítése

A bemeneten a környezet azonosító nem kötelező adat. Amennyiben nincs kitöltve, egy egyedi azonosító kerül generálásra a UniqueStringGenerator osztály segítségével.

Kezdeti állapot felépítése

A bemenetből készülő állapot három osztály szerint tagolható:

- EnvironmentDescriptor-ok
- SlaveManager-ek
- CommandTask-ok

Ezen osztályokat az InputHandler példányosítja, a bemeneti osztályok alapján felépíti és az alkalmazás többi része számára rendelkezésre bocsátja.

Amíg a SlaveManager és CommandTask osztályok példányosítása, felépítése egyértelmű és nem igényel egyéb adatokat, az EnvironmentDescriptor-ok minden eleménél az UploadFolder tulajdonság alapján az összes mappában található fájl kifejtésre kerül.

6.4. EnvironmentDescriptor

A környezetek egy azonosítóból és a környezetben futtatandó feladatok futtatásához szükséges fájlokból állnak.

Egy EnvironmentDescriptor egy környezetet reprezentál. Az osztályt csak a Master ismeri, az ő feladata, hogy a Slave-en biztosítsa a fájlok elérhetőségét a környezetben egy feladat futtatása előtt.

6.5. ITask, TaskBase, CommandTask

Ezek az osztályok egy feladatot reprezentálnak. Neveiknél fontos volt a .NET keretrendszer Task osztályával való névütközés elkerülése. A nevek hasonlósága továbbra is kellemtelen, főleg, hogy a TAP minta alkalmazása miatt a Task osztály gyakran fordul elő a kódban.

6.5.1. ITask

Az ITask interfész egy általános feladatot reprezentál. Tulajdonságai:

- **Key:** Az alkalmazás futása során a feladatok egyedi azonosítója. Jelenleg pozitív egész szám, amely növekvő sorrendben kerül kiosztásra.
- **State:** A feladat állapota.
- **CancellationState:** A feladat visszavonásra került-e, és ha igen, hogyan.
- **SoftDependencies:** A feladat gyenge függőségei.
- **HardDependencies:** A feladat erős függőségei.

TaskState

Egy felsorolás típus (enum). A feladat végrehajtással kapcsolatos állapota.

- **NotYetStarted:** A feladat végrehajtása még nem kezdődött el.
- **BeingProcessed:** A feladat éppen végrehajtás alatt áll.
- **Done:** A feladatot egy Slave már végrehajtotta, sikerrel.
- **Failed:** A feladatot egy Slave végrehajtotta, de a feladat végrehajtása hibát okozott, így a feladat sikertelen.

CancellationState

Egy felsorolás típus (enum). Azt reprezentálja, hogy a feladat visszavonásra került-e. Egy feladatot visszavonhat például egy sikertelen függősége. A sikertelen feladatoktól függő feladatok visszavonását a RecursiveTaskCanceller osztály végzi.

- **NotCancelled:** A feladat nincs visszavonva.
- **CancelledManually:** A feladatot egy felhasználó visszavonta.
- **CancelledByFailedDependency:** A feladatot egy függőségének sikertelensége visszavonta.

6.5.2. TaskBase

Absztrakt feladat ősosztály, a benne megvalósított funkcionalitás csak egy, a megadott függőségek kiértékelésével kapcsolatos problémát hivatott megoldani. A probléma abból adódott, hogy a megadott függőségek listájának lusta kiértékelése nagyban lassította a függőségeket felhasználó algoritmusokat (mind például a feladatok visszavonása). Az ősosztály funkciója, hogy a megadott függőségek listáját előre kiértékelje.

6.5.3. CommandTask

Egy parancssori feladatot reprezentáló osztály. Jelenleg ez az egyetlen feladattípus, amit a Slave-ek kezelni tudnak, így a keretrendszer csak ezt a feladattípust támogatja. A TaskBase funkcionalitásán kívül a parancssori feladatok egyedi tulajdonságait tartalmazza.

- **Command:** A futtatandó parancssori feladat. Nem tartalmazza a paramétereket.
- **Args:** A futtatandó feladat parancssori paraméterei. Összefűzésre kerül a Command paraméterrel egy szóköz elválasztó karakterrel.
- **Timeout:** A feladat maximálisan ennyi ideig futhat, másodpercben.
- **EnvironmentIdentifier:** A Slave-en a környezet azonosítója.
- **CommandIsInEnvironmentFolder:** A Command-ot a Slave-en a feladatesoport környezet mappájához relatívan kell értelmezni. Amennyiben az értéke `false`, úgy a Command a Slave munkakönyvtárához képest relatív, vagy abszolút útvonalat jelöl

Az osztályhoz tartozik egy állandó szöveges reprezentáció is, ami a kimeneteken felhasználásra kerülhet, és azokon könnyen azonosíthatóvá teszi a feladatot.

6.6. SlaveManager

Az osztály felelőssége, hogy az alkalmazás többi részétől elfedje a Slave szolgáltatások megvalósítási részleteit, a Slave-vel történő kommunikáció módját. Amennyiben a keretrendszerben használható Slave-ek interfésze, implementációja megváltozik, a SlaveManager osztály felelős meggátolni a változások végiggyűrűzését a Master alkalmazásban.

Amennyiben többféle Slave kerül implementációra (például különböző operációs rendszerek, vagy különböző jellegű feladatok miatt), a SlaveManager osztályt interfésszé kell átalakítani, és abból leszármaztatni a megvalósításokat, az ütemezőben pedig az interfészt kell használni. Ezt megvalósítani várhatóan nem lenne ennyire egyszerű, mivel ez magával vonhatná azt, hogy egy feladat csak adott típusú Slave-eken képes futni, így a feladat és Slave típus összerendelését meg kellene oldani.

Egy SlaveManager egy-egy kapcsolatban áll az általa elfedett Slave-vel, saját maga számára példányosít WCF klienseket.

Az osztály egyik erőssége a hosszú műveleteinek aszinkron jellege. A hosszú műveletek WCF hívásokat tartalmaznak, szerencsére az `svcutil.exe` eszköz (ismertetése az 5. fejezetben található) a szolgáltatáshívások aszinkron párjait is legenerálja, így az aszinkron jelleg továbbterjesztése egyszerűen megoldható.

Az osztály egy korábbi verziója képes volt szinkron módon is hívni a Slave műveleteit, de ez a funkcionalitás megszűnt, mert fölösleges volt.

Nyilvános tulajdonságai a következők.

- **key:** Az alkalmazás futása során a SlaveManager-ek egyedi azonosítója. Jelenleg pozitív egész szám, amely növekvő sorrendben kerül kiosztásra.

- **State:** A Slave állapota. Részben redundáns a Slave-ek lentebb ismertetett Poolokba rendezésével, viszont egyszerűsíti az általános, statikus műveletek (például alkalmazás állapotának naplózása) működését.
- **CurrentTaskKey:** Jelenleg a Slave melyik feladaton dolgozik. Publikusan csak azért jelenik meg, hogy az alkalmazás állapotát könnyebben naplózhassuk.

Fontosabb publikus műveletei a következők:

- **InitializeAsync:** Inicializálja a Slave-et, az összes bemenetként kapott környezet-hez létrehoz a Slave-en egy mappát, a mappába feltölti az összes szükséges fájlt.
- **UploadFileAsync:** Feltölti a paraméterként megadott fájlt a Slave paraméterként kapott környezet mappájába. Jelenleg csak az **InitializeAsync** függvényben kerül felhasználásra, így nem szükséges publikusnak lennie.
- **RunTaskAsync:** Lefuttatja a paraméterül kapott feladatot a Slave-en. Felel a feladat állapotát reprezentáló tulajdonságok karbantartásáért, a feladat közben fellépő hibák kezeléséért. A feladat eredményével nem tér vissza, a függvény befejezése után az a **ProcessResult** függvény segítségével kérhető el.
- **ProcessResult:** A futtatott feladat eredményét adja vissza, illetve a Slave állapotváltozóit feladatra várakozó állapotba állítja.
- **PullResultsAsync:** A paraméterként kapott környezetekben található összes, a környezetben kezdetben még nem megtalálható (tehát a futások során keletkezett) fájlt lemásolja a Master gépre, a paraméterként kapott mappába. Ez a függvény használható, ha a futtatott parancssori feladat nem csak a standard kimenetre és a standard hiba kimenetre írja az eredményét, hanem egyéb fájlokba is. Jelenleg minden kampány futása után egyszer minden Slave minden környezetében található összes fájlt letölti a keretrendszer. A kampány futása után egy mappastruktúrában az eredmények megtekinthetők.

6.6.1. WCF kliensek konfigurációja

A szolgáltatásokhoz csatlakozás konfigurációja az App.config fájlban történik, a szolgáltatás végpontok címe viszont a bemeneti fájlban megadottak szerint, kódban, csatlakozás előtt felülírásra kerül.

6.6.2. SlaveState

Ez az enum egy Slave lehetséges állapotait reprezentálja. A feladatok állapotához hasonlóan ez a tulajdonság is részben redundáns azzal, hogy a Slave jelenleg melyik poolban található.

- **NotAvailable:** A Slave nem elérhető, az állapota nem felmérhető.
- **Uninitialized:** A Slave nem került még inicializálásra.
- **Ready:** A Slave készen áll feladatok fogadására.
- **Busy:** A Slave éppen egy feladaton dolgozik.
- **Done:** A Slave befejezte az adott feladaton való munkát, az eredmények feldolgozására vár.

6.7. Pool

A keretrendszerben egyes erőforrásoknál fontos, hogy egyszerre csak egyszer kerüljenek kiosztásra. Fontos például, hogy egy feladaton csak egy Slave dolgozhat, vagy egy Slave-nek egyszerre csak egy feladat kerüljön kiosztásra.

Ezek problémaként főleg akkor jelennek meg, ha párhuzamosan futó folyamatok kezelik az erőforrásokat, vagy versengenek azokért.

6.7.1. A Pool által biztosított garanciák

A Pool osztály szerepe erőforrások halmaza elé egy olyan interfész megfogalmazása, ami garantálja a következőket.

- Minden erőforrás egyedi a poolban. Az erőforrások kulcs szerint kerülnek azonosításra, speciális esetben a kulcs lehet az erőforrásra mutató referencia is.
- Egy erőforrást a poolból csak egyszer lehet kivenni, az erőforrás megszerzése garantálja, hogy egyedül az adott folyamat szerezte meg az erőforrást.

6.7.2. Kényelmi szolgáltatások

A kényelmes felhasználás miatt az osztály a következőket is szolgáltatja.

Erőforrás ne csak kulcs, vagy referencia szerint legyen elérhető. A legtöbb esetben teljesen mindegy, hogy az alkalmas erőforrások közül melyik kerül kiválasztásra.

Erőforrásra aszinkron várakozás. Amennyiben egy erőforrás nem azonnal elérhető (például nincs szabad Slave), valamilyen jellegű várakozást kell megvalósítani. Az osztály ezt a TAP minta szerint, szálbiztos módon valósítja meg. Több aszinkron módon erőforrásra várakozó folyamat közül garantáltan egy kaphatja meg az erőforrást.

Erőforrások kérésénél szűrés és rendezés megadása. Egyes erőforrásoknál előfordul, hogy kiosztásuknál prioritásszempontok fordulnak elő. Például inkább egy olyan feladat kerüljön kiosztásra, aminek minden gyenge függősége sikeres volt, mint egy olyan feladat aminek még gyenge függősége éppen futtatás alatt áll. Más erőforrásoknál előfordulhat, hogy egy adott célra csak adott feltételeknek eleget tevő erőforrások alkalmasak. Például olyan feladat nem osztható ki Slave számára, aminek még van nem sikeresen befejezett erős függősége. A szűrést és rendezést az erőforrás elkérésénél paraméterként megadhatjuk.

Az összes aktuálisan elérhető erőforrás elérése egy listában úgy, hogy emiatt az erőforrások továbbra is a poolban maradnak. Ez egy kiskapu, a használatakor fokozott óvatossággal kell eljárni. Azért van rá szükség, mert egyes folyamatok úgy használják az erőforrásokat, hogy nem szeretnék az erőforrás kizárólagos kezelőjévé válni, vagy körülményesebb lenne a standard interfészen keresztül az erőforrásokat kötegetlen elérni:

- Ilyen folyamat például a naplózás, ahol az erőforrásokat csak olvassuk, és külön felkészülünk arra, hogy az erőforrást közben egy másik folyamat megváltoztathatja.
- Ilyen folyamat még a feladatok rekurzív visszavonása. Az algoritmus az összes elérhető feladaton dolgozik, azok közül választja ki a visszavonandókat. Fontos, hogy a

futása nem akadályozza meg a feladatok közüli választást, illetve mivel ez adatmódosító folyamat, így az feladatok felhasználása közben nem okozhat hibát, ha a visszavonási állapot megváltozik. Az ütemezés abból a szempontból hibátűrő, hogy egy feladat visszavonási állapota csak kiválasztáskor kerül ellenőrzésre, ha közben megváltozik, attól még a feladat elvégzésre kerül. Megeshet emiatt például, hogy egy visszavont feladat sikeresen lefut, de ez általában nem probléma.

6.8. RecursiveTaskCanceller

Az osztály célja egy feladattól függő más feladatok visszavonása. Egyetlen statikus függvényének bemenete egy feladat kulcsa, feladatok listája, illetve hogy manuális, vagy automatikus visszavonásról van-e szó.

Azok a feladatok kerülnek visszavonásra, ahol a megadott kulcsú feladat szerepel a függőségek között, illetve a függőségek függőségei között, rekurzívan. A felderítés csak a megadott feladatok listája mentén történik, visszavonásra csak ezek, és rekurzív függőségeik kerülhetnek.

A megadott feladatlisaként elég a még futtatásra várakozó feladatokat megadni, hiszen a már kész, vagy épp futó feladatoknál a visszavonásnak nincs semmilyen következménye. A visszavonást ugyanis csak az ütemező veszi figyelembe annyiban, hogy egy Slave számára nem választ visszavont feladatot.

6.8.1. Példa

Tekintsünk egy gyenge függőségi láncot. (A)-tól gyengén függ (B), (B)-tól gyengén függ (C), (C)-tól gyengén függ (D). Két Slave áll rendelkezésre.

1. Az (A) feladat elkezd futni, hiszen egyedül ennek nincs függősége.
2. A (B) feladat elkezd futni, hiszen a második Slave-re ez az egyetlen olyan feladat, aminek nincs várakozó függősége.
3. A (B) feladat előbb fejeződik be, mint a tőle gyengén függő (A), sikeresen.
4. A (B) feladattól függő (C) feladat feldolgozás alá kerül.
5. Az (A) hibával áll le.
6. Elkezdődik a hibával leállt feladattól (A) rekurzívan függő más feladatok visszavonása ((B), (C) és (D) mind rekurzívan függenek (A) feladattól).
7. A várakozó feladatok között egyedül (D) található, ennek mentén történik a függőségek felderítése. (D) függ (C)-től, (C)-függ (B)-től, (B)-függ (A)-tól. (B) így visszavonásra kerül, ettől (C)-is és ettől (D)-is.
8. Az ütemező (D)-t a visszavonás hatására eltávolítja a várakozó feladatok közül.

Ekkor az alábbi lesz a feladatok állapota:

- (A) feladat hibával fejeződött be, nem került visszavonásra.
- (B) feladat sikeresen befejeződött, visszavonásra került.
- (C) feladat éppen fut, visszavonásra került.
- (D) feladat nem fog lefutni, visszavonásra került.

Vegyük figyelembe, hogy bár az (A) feladaton kívül mindegyik visszavonásra került, de egyedül a (D) feladat nem fog lefutni, mivel a már befejeződött és az éppen futó feladatokra a visszavonás nincs hatással.

6.9. AsyncProcessor, ütemező

Az ütemező feladata, hogy a szabad Slave-eknek válasszon a várakozó feladatok közül, a feladat futtatását a Slave-en elindítsa, illetve a futtatás eredményét feldolgozza, majd a Slave-nek újra feladatot válasszon. Ha egy Slave és egy feladat is rendelkezésre áll, a feladatot minél gyorsabban ki kell osztani a Slave számára. A visszavont feladatokat el kell távolítani a futtatandó feladatok közül. Ügyelni kell arra is, hogy a működését befejezze, ha már minden feladat lefutott, vagy visszavonásra került.

Az ütemezőnek három különböző verziója került megtervezésre. Az első a specifikációban megadottakhoz hasonlóan Slave-enként teljesen párhuzamosan végezte a feladatát úgy, hogy minden Slave teljes kezelése külön szálon történt. Ebben a megvalósításban bonyolult volt egyes erőforrások szálbiztos használatát biztosítani. A második verzióban az ütemező az elérendő erőforrások mentén került szálakra felosztásra, ez a 6.9.1.-es alfejezetben ismertetésre kerül. A végső verzió azon az ötleten alapszik, hogy a feladatok futtatásához képest az ütemező minden más művelete elhanyagolható idejű, így akár egy szálon sorosítható, ehhez az C# nyelv új async/await kulcsszavait használja. Ez a verzió a 6.9.2-es alfejezetben kerül ismertetésre.

Az ütemező a Slave-ekkel a `SlaveManager` osztályon keresztül dolgozik. Az osztály leírásában a Slave-ekre hivatkozás valójában `SlaveManager` példányokra hivatkozás.

6.9.1. Explicit szálakat használó verzió

Bemutatásra kerül az ütemező egy előző verziója is annak érdekében, hogy ennek tükrében lehessen mérlegelni az új ütemező képességeit.

Az ütemező egy előző iterációja két külön szálon periodikusan futó cikluson alapult. Az egyik szál felelt a Slave-ekhez feladatok választásáért és a feladatok indításáért, a másik szál a kész feladatok eredményeinek feldolgozásáért és a Slave-ek újrahazsnosításáért. Mindkét szál egy másodpercig várakozott, ha a feladata elfogyott.

Ez a verzió a feladatok indításához további új szálakat használt, amiket a Slave-en futó feladat befejezéséig várakoztatott. Az ütemezőn belül így potenciálisan Slave-enként egy, illetve további két szál futhatott egyszerre.

Előnyök

Ez a megoldás viszonylag egyszerű volt, rövid idő alatt elkészült, a működését gyorsan be lehetett mutatni. Jól elkülönült, hogy melyik szál melyik feladatokat és Slave-eket kezel:

- Egy szál kezelte a futásra várakozó feladatokat és feladatra várakozó Slave-eket.
- Slave-enként egy szál kezelte magát a Slave-et és az éppen rajta futó feladatot.
- Egy szál kezelte a kész feladatokat és a feladatát befejezett Slave-eket.

Ennek az elkülönítésnek az érdekében jöttek létre a Poolok. A két ciklikusan futó szál mindig a számára fenntartott Poolból egy másik Poolba helyezte át az erőforrásokat. A Pool garantálta, hogy az adott erőforrást egyedül ez a szál kezeli az ütemezőben.

Problémák

Túl sok várakozás. A szálak periodikus altatása valójában általában nem adott időre kellett volna szóljon, hanem egy olyan eseményre várakoztak, amit nem tudtak megfigyelni, csak utólag leellenőrizni, hogy megtörtént-e. Az aktív várakozás még a szál rövid időkre történő elaltatásával is fölöslegesen erőforrásigényes, valamint a következő problémákat okozza:

- Az altatás ideje egy kompromisszum a lehető leggyorsabb lefutás, és a processzor terhelésének csökkentése között.
- A feladat futásának befejezése egy megfigyelhető esemény. A `SlaveManager`-ben a szinkron WCF hívás visszatér, ha feladat befejeződött. Meghívható itt egy callback függvény, elsüthető egy esemény. Nyilván nehézséget okoz, hogy nem ugyanabban a számban tér vissza a feladat, mint amelyikben a visszatérésről értesülni szeretnénk.
- Túl sok szál fölösleges használata. Akkor van értelme egy feladathoz külön szálat rendelni, amikor az processzor-intenzív. A mi esetünkben a szálak a felhasználásuk legnagyobb százalékában várakoznak. Olyan környezetben ahol a szálak korlátozottan állnak rendelkezésre (például egyes webszervereken) ez pazarlást jelent.

6.9.2. Aszinkron változat

Az ütemező jelenlegi verziója a Task-based Asynchronous Pattern (TAP) szerint került fölépítésre, és így aszinkron módon működik. A specifikációban megadott párhuzamosítási mód helyett az aszinkronitást a TAP minta természetes vágópontjai adják, így explicit szálakat egyáltalán nem használ

Megvalósítás során fő cél volt az előző ütemező problémáinak kijavítása.

Előnyök

Az ütemezőben nincs párhuzamos kódvégrehajtás, egyszerre egy dolog történik, csak a végrehajtás sorrendje szokatlan a TAP minta miatt. Éppen ezért az erőforrások kezelésében nem alakulhat ki versenyhelyzet, mert sosem dolgozik egyszerre két külön kezelő.

Az aktív várakozás minimális, helyette minden közvetlenül a logikailag őt kiváltó esemény hatására történhet. Az eseményekre így a lehető leggyorsabban reagálunk, de mégsem használunk el processzoridőt aktív várakozásra.

Nincs szálpazarlás sem. Bár konzolalkalmazások esetében a futtatókörnyezet az aszinkron vágópontoknál a kontextust az aktuális számban tárolja el, a futtatást egy másik számban folytatja, ez pusztán a futtatókörnyezet beállítása. Egyes környezetekben (például ASP.NET vagy WPF alkalmazások), ez nincs így.

Hátrányok

A megvalósítás megértéséhez a TAP minta ismerete az `async/await` kulcsszavak jelentésének megértése szükséges. Akeretrendszer későbbi fejlesztői számára ez kellemetlenséget jelenthet.

6.10.AsyncProgram

Az osztály a Master alkalmazás aszinkron részének belépési pontja. Az ezt meghívó Program osztály a konzolalkalmazás bemeneteinek ellenőrzésén és az AsyncProgram osztály egyetlen nyilvános függvényének meghívásán kívül semmit nem tesz. Erre azért van külön szükség, mert a C# nyelv egyelőre még nem támogatja a konzol alkalmazásokban az aszinkron belépési pontot (például `static async Task Main(string[] args)`).

Az osztály feladatai:

- Logger osztályok inicializálása.
- Poolok példányosítása a feladatok és Slave-ek számára.
- A bemeneti paraméterek feldolgozásának (InputHandler) elindítása.
- Feladatok és Slave-ek Poolokba rendezése.
- Slave-ek inicializálása a bemenetből összeállított környezetek alapján.
- Ütemező indítása.
- Ütemező leállása után az összes környezetben létrejött új fájl összegyűjtése a Slave-ekről.
- Összegzés kiírása a konzolra.

6.11.UniqueStringGenerator

Az osztály feladata megadott sablon alapján különböző, egyedi stringek generálása. A sablon különböző kihelyettesíthető részeket tartalmaz:

- {guid}: véletlenszerű, kötőjelekkel feltagolt hash, 00000000-0000-0000-0000-000000000000 alakban.
- {guid8}: 8 karakter hosszú véletlenszerű hash. (00000000)
- {date}: Mai dátum yyyyMMdd alakban.
- {datetime}: Mai dátum yyyyMMddHHmmss alakban.

Amennyiben a sablon nem tartalmaz {guid} vagy {guid8} paramétert, úgy ütközés esetén a generált string egy egyre növekvő számmal kerül kiegészítésre.

Az osztály az egyediséget a konstruktorában kapott teszt függvénnyel ellenőrzi. Ha a teszt függvény nem tudja garantálni az egyediséget (például konstans `true` értékkel tér vissza), úgy a string generálás eredménye sem lesz garantáltan egyedi. Ugyanakkor a hash paraméterek ütközése van annyira ritka, hogy ha kritikus hibát nem okoz, akkor elfogadható kompromisszum lehet.

A keretrendszerben jelenleg az osztály csupán egy kampány futtatás gyökér eredmény-mappája nevének generálására használt, a környezeti azonosítók generálásában nem vesz részt (ami alapján szintén mappák kerülnek elnevezésre).

6.12.ApplicationStateLogger

Az osztály a keretrendszer futásának monitorozását hivatott megkönnyíteni egy státusz fájl segítségével. A státusz fájl felépítése a kimenetek alpontjaként kerül részletezésre.

Az osztály közvetlen referenciával rendelkezik az összes `ITask` és `SlaveManager` példányra és nem csak a Poolokon keresztül éri el azokat. Emiatt gyorsan és biztosan el tudja érni mindegyiket, cserébe az erőforrásokról nem látja, hogy melyik Poolban vannak. Ez a fő oka annak, hogy mind a feladatokon, mind a Slave-eken az állapotváltozók értékei redundánsak az erőforrást tartalmazó Poolal.

Az osztály fenntart saját állapotváltozókat azzal kapcsolatban, hogy a Master alkalmazás éppen mit csinál, vagy a keretrendszer aktuális futása közben történt-e már hiba. Ezek az állapotváltozók az alkalmazásban sehol máshol nem kerülnek felhasználásra, pusztán a státusz fájl kimenetét befolyásolják.

A státusz fájl frissítése explicit, olyan eseményekkor van kiadva a frissítés parancs, amikor az alkalmazás állapota potenciálisan megváltozik, így a fájl tartalma is változhat. Az osztály saját állapotváltozóit is explicit kell frissíteni, viszont ez magával vonja a státusz fájl tartalmának frissítését is.

6.13.SlaveServiceCmd

Az osztály egy parancssori alkalmazás, amelynek egyedüli funkciója, hogy a `SlaveService` szolgáltatás gazdafolyamata legyen. A program felépítése a „How to: Host a WCF Service in a Managed Application” [21] cikk szerint történt.

Az igények bővülésekor a szolgáltatás egyéb módokon is kiszolgálható lehetne, például egy Windows szolgáltatás részeként, vagy egy IIS szerveren, de első körben egy parancssori alkalmazás elégséges.

6.13.1.Szolgáltatás konfigurációja

A szolgáltatás konfigurációja kódban történik.

- A Binding (protokoll, átviteli mód, időkorlátok, üzenetméret-korlátok) beállításra kerül.
- A szolgáltatás metaadatainak publikálása, a hibák részleteinek közzétevése engedélyezésre kerül.
- A szolgáltatás figyelni kezd a `http://127.0.0.1:{port}/slave` címen ahol a port az alkalmazás első és egyetlen bemeneti paramétere.

6.13.2.ISlaveService, SlaveService

A szolgáltatás feladatai:

- Fájlok átvitelének lehetővé tétele a környezeti mappába
- Feladat végrehajtása a környezeti mappában
- Fájlok lekérése a környezeti mappából

Az első és harmadik feladatot támogatandó a szolgáltatástól elkérhető az egy környezeti mappában elérhető fájlok listája, ezt felhasználva a Master csak a hiányzó fájlokat próbálja meg feltölteni, illetve képet kap arról, hogy milyen fájlokat töltsön le onnan.

6.13.3.Streaming engedélyezése

A szolgáltatáson keresztül lehetőség van nagyobb méretű fájlok küldésére és fogadására. Ehhez szükség volt a szolgáltatás egyes üzenetméret-korlátainak magasra állításához, viszont ez nem volt elég a probléma megoldásához.

Egy WCF szolgáltatás átviteli módja alapesetben bufferelt. Ez azzal jár, hogy a küldő a teljes üzenetet memóriában összeállítja, elküldi az üzenetet a fogadónak, ami az üzenet teljes tartalmát memóriába olvassa, mielőtt azt felhasználói kódban elérhetővé teszi. Sajnos ez a megközelítés egy bizonyos üzenetméret fölött járhatatlan, a fájlok méretei meghaladhatják a szolgáltatást, vagy a szolgáltatás kliensét futtató számítógépek teljes memóriatartományának méretét.

Szerencsére egyes protokollok fölött a WCF beépítve támogatja a nagyobb üzenetek streamelt átvitelét. Ez remekül együttműködik a `FileStream` [22] osztály használatával, így megtehető, hogy a fájl csak az üzenet küldése közben kerül folyamatosan beolvasásra és az üzenet fogadása közben folyamatosan kiírásra kerül, miközben mindkét oldalon csak egy minimális buffer van memóriában fenntartva. Az elérhető memória mérete így elméletileg egyáltalán nem korlátozza a küldhető és fogadható üzenetek lehetséges méretét.

A streaming beállításához a szolgáltatási szerződésre bizonyos megkötések [23] vonatkoznak, illetve nagy üzenetek átvitelére képes szolgáltatás tervezésénél több mindent meg kell fontolni [24].

6.13.4.ProcessHelper

A szolgáltatás legfontosabb feladata a megadott parancssori feladatok futtatása. Erre a `Process` osztály használható, ami képes egy új folyamatot külön, vagy alfolyamatként elindítani. Az osztály a paraméterként kapott parancsot alfolyamatként indítja, és feliratozik a kimeneteire (`StdOut`, `StdErr`).

Ugyan a folyamat futtatásánál megadható, hogy a munkakönyvtár a környezeti mappára mutasson. A `process` által futtatandó parancs ugyanakkor nem a `Process` osztályon megadott munkakönyvtárban kerül kiadásra, hanem a szolgáltatás gazdafolyamának munkakönyvtárához képest relatív módon lesz értelmezve.

Mivel felhasználástól függően az útvonal lehet abszolút, a szolgáltatáshoz képest relatív (például ha a program már előre föltöltésre került a Slave gépre, környezeti mappán kívül), vagy a környezeti mappához képest relatív (a Master bemenetében megadott a feladatsorozat környezetébe feltöltendő mappa tartalmaként kerül a Slave-re feltöltésre), az osztály egyetlen függvénye külön paraméterként fogadja ezt az információt. Ha a parancs a környezeti mappához képest relatív, akkor a környezeti mappa útvonalával a parancs összefűzésre kerül, egyébként módosítás nélkül kerül kiadásra.

6.14. Kimenetek

A keretrendszer számos kimenettel rendelkezik. Futás közben mind Master, mind Slave oldalon megjelenik a kimeneteken a keretrendszer aktuális állapota. A Master a futásának részleteiről pontos naplót állít össze, illetve a feladatok eredményei is megjelennek az alkalmazás kimenetén. Az alább felsoroltakon kívül a Master és Slave alkalmazás is szolgáltat fejlesztőknek szánt kimeneteket.

6.14.1. Results mappa

A mappa a Master alkalmazást futtató gépen jelenik meg. Egy kampány futtatásának végén tartalmazni fogja a feladatok kimeneteit futtatott kampányonként, ezen belül környezetenként, ezen belül Slave-enként tagolva. A standard output és standard error kimenetek `{feladatkulcs}__output.log` és `{feladatkulcs}__error.log` néven jönnek létre az adott mappában. A Slave környezeti mappáiban a kampány futtatása közben létrejött fájlok szintén megjelennek az adott mappában.

Ha például a kampányon belüli `SlowSimulation` környezetben futtatott egyik feladatunk eredményére vagyunk kíváncsiak, akkor a konzol kimenetből, vagy info logból eldönthetjük, hogy a feladat melyik Slave-en futott és milyen generált kulcsot kapott, ez alapján az eredményt megtalálhatjuk például az alábbi, Masterhez képest relatív útvonalon:

```
./Results/campaign-20160510180016/SlowSimulation/2/22__output.log
```

A kampány mappa neve tartalmazza a jelenlegi dátumot és pontos időt, így nem teljesen egyedi, de feltételezve, hogy a Mastert futtató gépen az idő nem kerül visszaállításra és a programot másodpercnél ritkábban futtatjuk, elkerülhető az ütközés.

A Slave kulcsa (esetünkben 2) és a feladat kulcsa (esetünkben 22) nem túl beszédes, éppen ezért az `*__output.log` és `*__error.log` fájlok fejléceként megjelenik a kiadott parancs és paraméterei.

6.14.2. Info és Error naplók

A Master futtatásakor egy log mappa jön létre, ez tartalmazza a Master futásának részletes naplóját (`info.log`) és a futás közbeni hibák naplóját (`error.log`). Futásonként új fájl készül, az előző fájl archiválásra kerül a `log/archive` mappában `info-{date}-{id}.log` vagy `error-{date}-{id}.log` néven ahol a `{date}` az archiválás dátuma, az `{id}` pedig egyre növekvő természetes szám.

A futás részletes naplója pontos időbélyeggel tartalmazza többek között azt, hogy az ütemező milyen Slave-eken mikor milyen feladatot indít, a feladatok milyen eredménnyel térnek vissza, illetve melyik feladat kerül visszavonásra.

A hibák naplója csak a hibákat tartalmazza, legyen az a Slave szolgáltatástól visszakapott hiba, vagy a Masteren történt hiba.

6.14.3. Státusz fájl

A státusz fájl szintén a log mappában található status.txt néven. A célja, hogy a keretrendszer futásának állapotáról a felhasználó egy összefoglaló képet kapjon. Az alábbiakban egy példakimenet látható.

```
==== STATUS: Processing Tasks ====

Slaves:
(1) http://127.0.0.1:5555/slave | Working | (30)
../../../../WaitaSec/bin/debug/waitasec.exe 5 tg4_otherenv
(2) http://127.0.0.1:5556/slave | Working | (13)
../../../../WaitaSec/bin/debug/waitasec.exe wait.json tg1_upload_6
(3) http://127.0.0.1:5557/slave | Working | (22)
../../../../WaitaSec/bin/debug/waitasec.exe 5 tg2_nonameenv

Tasks:
FAIL (1) waitasec.exe 8 3 tg5_exception > output.log 2> error.log
CNCL (2) waitasec.exe 8 3 tg5_exception > output.log 2> error.log
CNCL (3) waitasec.exe 8 3 tg5_exception > output.log 2> error.log
CNCL (4) waitasec.exe 8 3 tg5_exception > output.log 2> error.log
CNCL (5) waitasec.exe 8 3 tg5_exception > output.log 2> error.log
CNCL (6) waitasec.exe 8 3 tg5_exception > output.log 2> error.log
CNCL (7) waitasec.exe 8 3 tg5_exception > output.log 2> error.log
CNCL (8) waitasec.exe 8 3 tg5_exception > output.log 2> error.log
CNCL (9) waitasec.exe 8 3 tg5_exception > output.log 2> error.log
CNCL (10) waitasec.exe 8 3 tg5_exception > output.log 2> error.log
DONE (11) ../../../../WaitaSec/bin/debug/waitasec.exe wait.json tg1_upload_6
DONE (12) ../../../../WaitaSec/bin/debug/waitasec.exe wait.json tg1_upload_6
2>WORK (13) ../../../../WaitaSec/bin/debug/waitasec.exe wait.json tg1_upload_6
WAIT (14) ../../../../WaitaSec/bin/debug/waitasec.exe wait.json tg1_upload_6
WAIT (15) ../../../../WaitaSec/bin/debug/waitasec.exe wait.json tg1_upload_6
WAIT (16) ../../../../WaitaSec/bin/debug/waitasec.exe wait2.json tg1_upload_5
WAIT (17) ../../../../WaitaSec/bin/debug/waitasec.exe wait2.json tg1_upload_5
WAIT (18) ../../../../WaitaSec/bin/debug/waitasec.exe wait2.json tg1_upload_5
DONE (19) ../../../../WaitaSec/bin/debug/waitasec.exe 5 tg2_nonameenv
DONE (20) ../../../../WaitaSec/bin/debug/waitasec.exe 5 tg2_nonameenv
DONE (21) ../../../../WaitaSec/bin/debug/waitasec.exe 5 tg2_nonameenv
3>WORK (22) ../../../../WaitaSec/bin/debug/waitasec.exe 5 tg2_nonameenv
WAIT (23) ../../../../WaitaSec/bin/debug/waitasec.exe 5 tg2_nonameenv
FAIL (24) ../../../../WaitaSec/bin/debug/waitasec.exe 5 tg3_timeout
CNCL (25) ../../../../WaitaSec/bin/debug/waitasec.exe 5 tg3_timeout
CNCL (26) ../../../../WaitaSec/bin/debug/waitasec.exe 5 tg3_timeout
CNCL (27) ../../../../WaitaSec/bin/debug/waitasec.exe 5 tg3_timeout
CNCL (28) ../../../../WaitaSec/bin/debug/waitasec.exe 5 tg3_timeout
DONE (29) ../../../../WaitaSec/bin/debug/waitasec.exe 5 tg4_otherenv
1>WORK (30) ../../../../WaitaSec/bin/debug/waitasec.exe 5 tg4_otherenv

==== STATUS: Processing Tasks ====
```

A kimeneten fejlécében látszik, hogy jelenleg a keretrendszer feladatokon dolgozik. Alatta a Slave-ek felsorolásában látható, hogy összesen három Slave van, ezeknek mi az azonosítója, a címe, dolgozik-e éppen, és ha igen, melyik feladaton.

Ez alatt a feladatok felsorolása következik, jelenleg 30 elvégzendő feladat van. Minden feladtnál látható, hogy mi az azonosítója és pontosan milyen parancsot takar, valamint

követhető, hogy a feladaton melyik Slave dolgozik éppen és a feladatnak mi az állapota. A lehetséges állapotok a következők.

- WAIT: Futtatásra várakozik
- WORK: Futtatás alatt áll
- DONE: Sikeresen lefutott
- FAIL: Hibával futott le
- CNCL: Visszavonva
- WORK|CNCL: Futtatás alatt áll, visszavonva
- DONE|CNCL: Sikeresen lefutott, visszavonva
- FAIL|CNCL: Hibával futott le, visszavonva

6.15. Tesztprogram

A keretrendszer forráskódjával együtt egy tesztprogram is készült, ami képes fájlban vagy parancssori bemenetkén adott paraméterek alapján konzolkimenetet és hibát produkálni. Ugyan semmilyen hasznos feladatot nem végez, viszont alkalmas tesztelésre, vagy használható a keretrendszer működésére felhozott példákban. Ennek használata nem kerül ismertetésre, de a program forráskódja alapján könnyen megérthető.

7. Felhasználói útmutató

A fejezet során a keretrendszer használata kerül leírásra, beleértve a keretrendszer forráskódjának lefordítását, a keretrendszer komponenseinek telepítését és használat előtti felkonfigurálását.

A fejezetben nem szerepelnek a keretrendszer megvalósítási részletei, viszont elolvasása eléréséhez szükséges a keretrendszer használatához.

7.1. Fordítás forráskódból

Szükséges szoftver:

- .NET 4.5.2 [25] (Minimum)
- Visual Studio 2013 [26] (Minimum)
- NuGet kliens [27]
- SVN kliens (Opcionális)

A következő lépéseket tegyük meg.

1. A forráskódot töltsük le az SVN szerverről, vagy egyéb forrásból¹.
2. Amennyiben a forráskódot sikeresen beszereztük nyissuk meg a Solution-t Visual Studio-ban.
3. A harmadik féltől származó NuGet csomagokat (NLog) a projektekben töltsük le a Visual Studio NuGet bővítmény, vagy a parancssori `nuget.exe` segítségével.
4. A Master fordításához a MasterCommandLine projektet fordítsuk le.
5. A Slave fordításához a SlaveServiceCmd projektet fordítsuk le.

7.2. Master telepítése

A Master nem igényel külön telepítést, viszont a futtató felhasználónak írásjoggal kell rendelkeznie a mappára, amiben a programot futtatja.

7.3. Slave telepítése, konfigurálása

1. A Slave alkalmazást másoljuk a futtató virtuális gépre.
2. A Slave alkalmazást futtató felhasználónak írásjoggal kell rendelkeznie a mappára, amiben a programot futtatja.
3. Ki kell választanunk egy portot, ahol a Slave szolgáltatás kéréseket fogad. Ezt a futtató felhasználó számára engedélyezni kell, illetve a bejövő kommunikációt a tűzfalon át kell engedni. A megadott példákban az 5555 port kerül használatra. A beállításhoz használhatjuk a mellékelt parancsfájlt, vagy megtehetjük manuálisan.

¹ A szoftver beszerezhető a <http://petridotnet.inf.mit.bme.hu> címről. Itt a lefordított binárisok is megtalálhatóak, így a fordítás lépés szükségtelenné válik.

Automatikus

Futtassuk parancssorban (emelt szintű felhasználói jogosultság szükséges) a mellékelt `initialize.bat` parancsfájlt. Az egyetlen paramétere a kiválasztott port:

```
C:\TaskDistribution\SlaveService> initialize.bat 5555
```

Manuális

Engedjük át a bejövő TCP kapcsolatokat a megadott porton. Ezt megtehetjük a Windows Tűzfal konfigurációs felületén (`wf.msc`), vagy a parancssorban (emelt szintű felhasználói jogosultság szükséges):

```
> netsh firewall add portopening TCP 5555 "SlaveService on port 5555"
```

A vegyünk fel a futtató felhasználó számára URL-foglalást parancssorban (emelt szintű felhasználói jogosultság szükséges):

```
> netsh http add urlacl url=http://+:5555/slave user=%USERDOMAIN%\%USERNAME%
```

```
URL reservation successfully added
```

A `%USERDOMAIN%` és `%USERNAME%` változók a jelenleg bejelentkezett felhasználóra vonatkoznak, ha másik felhasználóval tervezzük a Slave-et futtatni, a paraméterek helyett használjuk a kívánt értékeket.

7.4. Slave elindítása

A Slave program futtatásához indítsuk el a `SlaveServiceCmd.exe` programot parancssorban. Bemeneti paraméterként meg kell adni a használni kívánt portot.

```
C:\TaskDistribution\SlaveService> SlaveServiceCmd.exe 5555
```

```
The service is ready at http://127.0.0.1:5555/slave  
Press <Enter> to stop the service.
```

7.5. Kampány, bemenet összeállítása

A Master futtatása előtt össze kell állítani a bemenetet. Ebben segítséget nyújthat a programhoz mellékelt `document.json` példabemenet.

A bemenet első blokkjában adjuk meg a használni kívánt Slave-eket.

```
"slaves": [  
  "http://192.168.1.101:5555/slave",  
  "http://192.168.1.102:5556/slave",  
  "http://192.168.1.103:5557/slave",  
],
```

A bemenet második blokkjában adjuk meg a kampányt. A kampány feladatcsoportok listájaként kerül megadásra.

```

"taskGroups": [
  {
    ...
    "tasks": [
      { ... },
      { ... }
    ]
  },
  { /* taskGroup2 */ ... },
  ...
]

```

Egy feladatcsoport feladatok listáját jelenti. A feladatcsoport feladatai egymás után sorban futnak.

- Egy feladatot a keretrendszer csak akkor kezd el futtatni, ha a feladatcsoport összes előző feladata sikeresen lefutott.
- Kivételt képez, ha van kihasználatlan Slave és a kampányban nincs olyan feladat, amire az előző kitétel igaz. Ilyenkor olyan feladat is futtatható, ahol a feladatcsoport összes előző feladata sikeresen lefutott, *vagy éppen fut*.

Ha egy feladat hibával fut le, a keretrendszer visszavonja a feladatcsoport összes további feladatát. A visszavont feladatok nem kerülnek elindításra, viszont ha már futnak, nem kerülnek leállításra sem.

Egy feladatcsoportnál megadhatóak a következők.

- `environmentIdentifier` (string): Egy szöveges azonosító, a Slave-eken ez alapján létrehozott környezeti mappában kerülnek a feladatcsoport feladatai lefuttatásra. Az azonosítóban felhasznált karakterek csak egy mappa nevében felhasználható karakterek lehetnek. Opcionális.
- `uploadFolder` (string): Relatív útvonal egy helyi mappára. A mappa tartalma a Slave-ekre felmásolásra kerül az `environmentIdentifier` által megadott mappába, tehát a feladatcsoport feladatai számára elérhető. Opcionális.
- `tasks` (array): A feladatcsoport feladatainak listája.

Egy feladatnál megadható:

- `command` (string): A lefuttatni kívánt parancs.
- `args` (string): A lefuttatni kívánt parancs paraméterei (opcionális).
- `count` (int): A feladatot ennyiszer kell lefuttatni (opcionális).
- `timeout` (int): A feladat ennyi másodpercig futhat, ezután megszakításra kerül.
- `inUploadFolder` (boolean): A feladat az `uploadFolder` paraméterben megadott mappával együtt kerül felmásolásra, a Slave gépekre. Ebben az esetben a lefuttatott parancs a környezeti mappához relatívan kerül értelmezésre (hiszen a Slave gépre az `uploadFolder` tartalma a környezeti mappába átmásolásra kerül). Egyébként a parancs abszolút, vagy a Slave szolgáltatás munkakönyvtárához relatívan kerül értelmezésre.

7.5.1. Példák

Egyszerű kampány programtelepítés nélkül

Szeretnénk a LongSimulation.exe programunkat 3 különböző bemeneti fájlal futtatni. A bemeneti fájlok egyre komplexebbek, a futtatásuk így egyre több időt vesz igénybe. A bemeneti fájlok a program mellett az Inputs mappában találhatóak.

A programunkat lokálisan így futtatnánk:

```
C:\Sims\LongSimulation> LongSimulation.exe ./Inputs/input1.xml
C:\Sims\LongSimulation> LongSimulation.exe ./Inputs/input2.xml --verbose
C:\Sims\LongSimulation> LongSimulation.exe ./Inputs/input3.xml
```

Két Slave gépünk áll rendelkezésre, ezeket a bemeneti fájlban cím alapján adjuk meg.

```
"slaves": [
  "http://192.168.1.101:5555/slave",
  "http://192.168.1.102:5555/slave",
],
```

Mivel a program a Slave gépeken nem érhető el így a kampányunkat úgy kell konfigurálnunk, hogy mind a program mind a bemenet a Slave gépekre felmásolásra kerüljön.

Mivel igaz, hogy egy szimulációt csak akkor szeretnénk elvégezni egy bemenetre, ha az kisebb bemenetre már sikerült, így egyetlen feladatcsoportba vegyük fel azokat.

A feladatok számára felvett feladatcsoportban adjuk meg a feladatokat, illetve az uploadFolder paraméterben a LongSimulation programot tartalmazó mappát. Ekkor a mappa az összes Slave-re felmásolódik a feladatok futtatása előtt.

A feladatoknál meg kell adnunk a futási idő felső korlátját. Azt tippeljük, hogy az input1.xml bemeneti fájlal a programunk 1 óráig, az input2.xml fájlal 1 napig, az input3.xml fájlal 2 napig fog futni. Az időkorlátunkat ennél nagyobb értéknek érdemes feladatonként megadni, ha mondjuk a feladat már kétszer ennyi ideje fut, akkor valószínűleg a feladat által végrehajtott algoritmus beakadt valamilyen módon. Legyenek így az időtúllépés értékei a következők: 2 óra, azaz 7200 másodperc, 2 nap, azaz 172800 másodperc, 4 nap, azaz 345600 másodperc.

Mivel a programunk az uploadFolder mappával együtt került felmásolásra így az inUploadFolder paramétert true értékkel adjuk meg.

Bemenetünk így a következő lesz:

```

{
  "slaves": [
    "http://192.168.1.101:5555/slave",
    "http://192.168.1.102:5555/slave",
  ],
  "taskGroups": [
    {
      "uploadFolder": "C:/Sims/LongSimulation",
      "tasks": [
        {
          "command": "LongSimulation.exe",
          "args": "input1.xml",
          "timeout": 7200,
          "inUploadFolder": true
        },
        {
          "command": "LongSimulation.exe",
          "args": "input2.xml --verbose",
          "timeout": 172800,
          "inUploadFolder": true
        },
        {
          "command": "LongSimulation.exe",
          "args": "input3.xml",
          "timeout": 345600,
          "inUploadFolder": true
        }
      ]
    }
  ]
}

```

Kampány telepített programokkal

Ebben a példában egy összetett vállalati szoftvercsomag képességeinek határait szeretnénk lemérni. A szoftver `measure1.exe` és `measure4.exe` parancssori programjait szeretnénk használni.

Mivel a programok a registry-t és egyéb rendszerszintű változókat használnak, amit a szoftvercsomag telepítője hoz létre, így az előző példában megismert módszerrel a feltöltési mappával együtt egyszerű átmásolásuk nem jöhet szóba, helyette a Slave-eken egyenként telepíteni kell a szoftvercsomagot. Ezt a `C:\Program Files\Measure Pro\` mappába meg is tesszük Slave-enként.

A `measure1.exe` és `measure4.exe` ugyanazt a feladatot végzi, de a `measure4.exe` egy javított algoritmussal. A két programot így ugyanazokkal a bemenetekkel szeretnénk kipróbálni.

Mivel az egyik algoritmus hibája esetén a másik méréseit nem szeretnénk visszavonni, így a két programnak két különböző feladatcsoportot szeretnénk fölvenni.

A bemeneti fájljainkat összekészítettük a Master `C:\Upload\` mappájába. Ezek meglehetősen nagyok, praktikus lenne, ha Slave-enként csak egyszer kerülnének felmásolásra.

Ha mind a két feladatcsoportnak ugyanazt a környezeti azonosítót adjuk meg, akkor az egyik feladatcsoport által feltöltött fájlokat a másik is elérheti, hiszen a feladatok ugyanabban a mappában futnak, adjunk meg hát az `environmentIdentifier` paraméternek mind a két feladatcsoportban egy fix értéket.

Mivel a programunk nem a feltöltött mappában található, így az `inUploadFolder` paramétert ne adjuk meg.

Az időkorlátokat nem tudjuk megbecsülni, de egy hétnél tovább nem szeretnénk a parancsokat futtatni, így egy hetet (604888 másodperc) adunk meg.

```
{
  "slaves": [
    "http://192.168.1.101:5555/slave",
    "http://192.168.1.102:5555/slave",
  ],
  "taskGroups": [
    {
      "environmentIdentifier": "fixvalue",
      "uploadFolder": "C:/Upload",
      "tasks": [
        {
          "command": " C:/Program Files/Measure Pro/measure1.exe",
          "args": "model1.model",
          "timeout": 604888,
        },
        {
          "command": " C:/Program Files/Measure Pro/measure1.exe",
          "args": "model2.model",
          "timeout": 604888,
        },
      ],
    },
    {
      "environmentIdentifier": "fixvalue",
      "uploadFolder": "C:/Upload",
      "tasks": [
        {
          "command": " C:/Program Files/Measure Pro/measure4.exe",
          "args": "model1.model",
          "timeout": 604888,
        },
        {
          "command": " C:/Program Files/Measure Pro/measure4.exe",
          "args": "model2.model",
          "timeout": 604888,
        },
      ],
    },
  ],
}
```

7.6. Master futtatása

Futtassuk a Master programot parancssorból (emelt szintű felhasználói jogosultság szükséges):

```
C:\TaskDistribution\Master> MasterCommandLine.exe input.json
```

7.6.1. Master monitorozása

A Master futása közben a konzol kimenetén kiírásra kerül az éppen végzett művelet, illetve a hibák, a program indítása óta eltelt idő mellett.

A Mastert monitorozni praktikusabb a `log/status.txt` fájl figyelésével. A fájl leírása egy példával együtt a 6.14.3. alfejezetben található.

A Státusz fájlban hibák két féle módon kerülnek feltüntetésre.

Ha az alkalmazás működésében hiba történt (például egy Slave nem elérhető), a fejlécben jelzésre kerül a hiba ténye:

```
==== AN ERROR OCCURED, CHECK THE LOG FOR DETAILS - STATUS: Starting ====
```

Ha egy feladat elvégzésében történt hiba, a feladat mellett a FAIL státusz kerül feltüntetésre:

```
FAIL (1) waitasec.exe 8 3 tg5_exception
```

A hiba részleteiről a `log/error.log` naplóban tájékozódhatunk, de a hiba megtalálható a részletes `log/info.log` naplóban is, ahol nem csak a hibás műveletek kerülnek feltüntetésre.

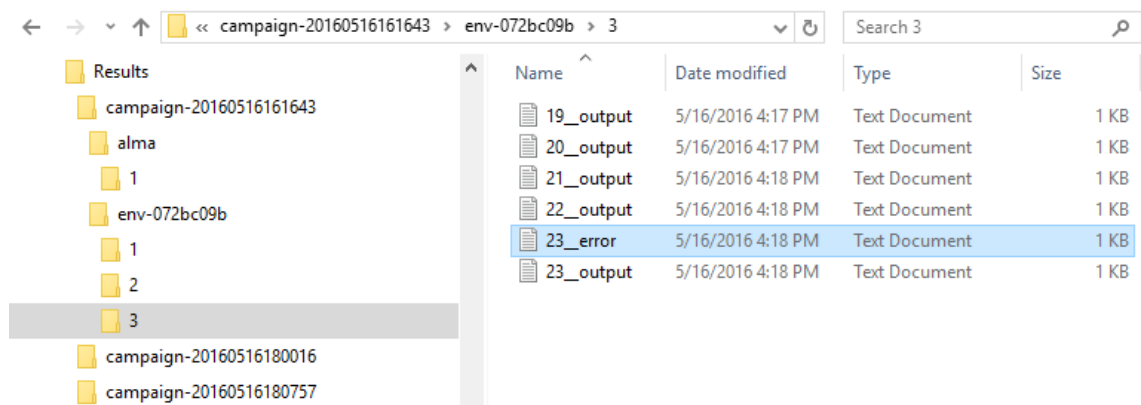
7.7. Eredmények elemzése

A feladatok eredménye a kampány futtatása után a Master gépen a Results mappában lesz megtalálható kampányonként, környezetenként és Slave-enként mappaszerkezetbe rendezve.

A Results mappában kampányonként egy új mappa jön létre, ennek a neve a futtatás időpontja alapján generált. A kampány mappában környezetenként egy mappa jön létre, ennek a neve a környezeti azonosító. A környezet mappájában Slave-enként egy új mappa jön létre, ennek a neve a Slave kulcsa. Ez növekvő természetes számokat jelent. A mappában az adott környezetben, az adott Slave-en futtatott feladatok eredményei jelennek meg.

A feladat standard kimenete és standard hibakimenete feladatonként egy fájlban kerül eltárolásra, aminek a neve a feladat kulcsa alapján jön létre. Például a 21-es azonosítóval rendelkező feladat kimenete `21__output.log` a hibakimenete `21__error.log` néven jön létre az adott mappában. Ezen fájlok első sorában feljegyzésre kerül, hogy pontosan milyen parancs futott.

Ha a feladat egyéb fájlokat hozott létre futás közben, akkor az adott mappában ezek a fájlok is megjelennek. A kimeneteket tartalmazó mappahierarchiára a 21. ábra ad példát.



21. ábra. Példa a kimeneti mappa hierarchikus szerkezetére

8. Potenciális kiterjesztések

Ebben a fejezetben olyan elvárások kerülnek leírásra a rendszerrel szemben, amik egyelőre nem képezik a feladat részét, de hosszú távon, a rendszer továbbfejlesztésénél megjelenhetnek követelményként.

8.1. Folyamatosan futó Master szolgáltatás

A Master command-line alkalmazás helyett lehetne egy gépen folyamatosan futó szolgáltatás. Ez praktikus lenne felhasználókezelés, feladatok, illetve Slave-ek menet közbeni hozzáadása és más kiterjesztések esetében.

8.2. Biztonsági megfontolások

A Slave szolgáltatás a futtató rendszernek egy támadási pontja. Kapott állományokat a rendszerben eltárol, ezeket futtatni képes. Ez támadási vektorként könnyen kiaknázzható, főleg, ha a Slave szolgáltatást adminisztrátori jogokkal futtatjuk.

A támadás mitigálásában segítene, ha a Mastert a Slave azonosítani tudná, csak a Mastertől fogadna el kéréseket, ami megoldható lenne például SSL tanúsítványok alkalmazásával.

A Slave túlterhelhető egy támadó által, ha túl sok kérést intéz felé és nyitva tartja azokat. Ez ellen segít, ha a túl sokáig nyitva tartott WCF kérések helyett több rövid kérést alkalmazunk. Például a feladat kezdete és befejezése két külön rövid kérésben történhet.

8.3. Más OS-ek

A .NET keretrendszer egyelőre elsősorban Windows operációs rendszeren működik. A Slave-ek lehetnek más nyelven implementálva, a WCF vagy egyéb Webservice szolgáltatások ugyanis nagyrészt platformfüggetlenek. Java nyelven megírt Slave-ek például a legtöbb platformon futnának.

8.4. Hibákról értesítés küldése a felhasználóknak

Jelenleg a rendszer működése közbeni hibákról a felhasználó nem kap tájékoztatást amíg a kimeneteket meg nem vizsgálja. Praktikus lenne, ha a hibákról a felhasználó valamilyen médiumon, például e-mail üzenetben értesítést kapna, így amikor éppen nem aktívan monitorozza a rendszert, akkor is azonnal reagálni tud a hibákra.

8.5. Új feladatok felvétele futás közben

Egy hosszabb kampány közben a futtatásban résztvevő (virtuális) gépek lefoglalásra kerülnek. Új feladatok futtatásához jelenleg meg kell várni a kampány befejezését.

Kényelmes lenne, ha kampány futtatása közben a rendszernek egy új kampányt meg lehetne adni amit vagy az előző után, vagy azzal párhuzamosan futtatna.

8.6. Feladatok visszavonása, megszakítása

A felhasználó kampány futtatása során dönthet úgy, hogy a kampány egyes feladatait mégsem szeretné lefuttatni. Jelenleg ebben az esetben az egyetlen amit tehet, hogy a keretrendszert leállítja, a kampányból kihagyja az érintett feladatokat és a keretrendszert a módosított kampánnyal elindítja.

Kényelmes lenne, ha a kampány futása közben egyes feladatokat vissza lehetne vonni a kampány megszakítása nélkül. Kényelmes lenne az is, ha az egyik Slave-en épp futó feladatot meg lehetne szakítani, a feladatot hibával befejezettnek megjelölni és a Slave-et az ütemező számára újra rendelkezésre bocsájtani. Az implementációban vannak ezt támogató elemek, azonban a funkció jelenleg még nem elérhető.

8.7. Új Slave felvétele futás közben

Jelenleg a Slave-ek száma a rendszer futása közben csak csökkenhet (Slave hibája esetén). Előfordulhat, hogy a kampány annyira erőforrásigényesnek bizonyul, hogy érdemes lenne új Slave-et bevonni az ütemezésbe. Ez jelenleg nem megtehető.

8.8. Hibás Slave újrainicializálása

Ideiglenes hálózati hibák és problémák miatt a Slave az ütemezésből jelenleg végleg kihagyásra kerül. Praktikus lenne, ha egy újra működőképes Slave-et a rendszer felismerne, újrainicializálna és az ütemezésbe ismét bevonna.

8.9. Kétirányú, vagy fordított kapcsolat a Slave és Master között

Jelenleg a Slave szolgáltatás működésében nem optimális, hogy egy feladat futtatása közben egy nyitott http kapcsolatot tart fent. Ha a szolgáltatás mindkét irányban kezdeményezőképes lenne, a kérést azonnal le lehetne zárni és a feladat befejezését egy vissza irányú hívással jelezni.

Meg lehetne azt is oldani, hogy a Slave helyett a Master ajánljon ki egy WCF szolgáltatást. A vezérlés így megfordulna, a Slave-ek a szolgáltatáson időközönként ellenőriznék, hogy van-e feladat számukra kiosztva, a feladatot elvégeznék és a feladat befejezésekor új hívást intéznének a Master felé.

Ebben az esetben a Slave-ek automatikusan, külön listázás nélkül is használhatóak lehetnének, ha a Masterben engedélyezzük, hogy bármely bejelentkező Slave inicializálásra kerüljön és részt vegyen az ütemezésben.

A jelenlegi SlaveManager osztály megvalósítását ez teljesen megváltoztatná, viszont a legtöbb műveletének interfésze maradhatna a jelenlegi, a Master többi komponense számára tűnhetne úgy, hogy még mindig a Master intéz hívásokat a Slave felé.

8.10. GUI

A menet közbeni vezérléshez praktikus lenne egy felhasználói felület, a lehetséges műveleteket kiemelve. A felületen megjelenhetnének az állapotadatok, a feladatok futásának eredményei, valamint a Slave-ek a naplói rendszerezve.

Első tervek szerint a keretrendszer részét képezte volna egy felhasználói felület, viszont egyéb funkciók megvalósítása nagyobb prioritást kaptak a felhasználói igények kielégítésében így a felhasználói felület elképzelt funkcionalitása egyéb, egyszerűbb funkciókkal lett helyettesítve.

8.11. Adatbázisba loggolás

Szöveges fájlok helyett az alkalmazás működése lehetne adatbázisba naplózva, lehetőség szerint egyedi adatszerkezettel, kereshető, szűrhető bejegyzéstípusokkal, és -forrásokkal.

8.12. Fa szerkezetű függőségi gráf

A függőségeket lánc helyett tetszőleges fa (esetleg körmentes irányított gráf) struktúrában is meg lehetne adni. Jelenleg ezt csak az akadályozza, hogy a bemenet formátuma láncokra képződik le. Egy új bemeneti formátummal a függőségeket egy irányított gráfként is meg lehetne adni. A keretrendszer jelenlegi állapotában ezt képes lenne kezelni.

8.13. „Vagy” típusú függőségek

Egyes mérések eredménye nem stabil, több ismételt futtatás során lehet, hogy alkalmanként hibáznak. Ha a keretrendszer képes egy feladatot többször lefuttatni, viszont nem képes arra, hogyha a futások közül bármelyik sikeres, akkor az ettől függő feladatok fusanak. Felmerülhet a feladat ismételt futtatásakor esetleg egy küszöbérték, aminél nagyobb arányú hiba gátolja csak meg a függőségek futtatását.

9. Összefoglalás

A keretrendszer jelenlegi állapotában egy már használható szoftver. Széleskörű felhasználását ugyan nyilvánvalóan gátolja egyes funkciók nehezebb működése, a felhasználói felület hiánya, a Windows platform kizárólagos támogatása, vagy a biztonsági funkciók hiánya, de a szoftver egy bizonyos környezetben (ideiglenes Windows operációs rendszert futtató virtuális gépek felhőben, a külvilággal csak virtuális magánhálózaton keresztül kommunikálva), bizonyos típusú feladatok futtatására (parancssori alkalmazások Windows operációs rendszerre fordítva) tökéletesen alkalmas, a motivációkban ismertetett problémákat megoldja.

A környezet és feladattípus támogatása természetesen egybevág az érdekeltek eredeti igényeivel, hiszen a legfontosabb szempont az volt, hogy az kerüljön minél hamarabb megvalósításra, amire tényleg szükség van, és most van szükség, de szem előtt tartva a lehetséges későbbi igényeket is.

Egyetemi képzésem során messze ez volt a legnagyobb, teljesen önállóan végzett projekt. A tervezés, a specifikáció és a megvalósítás az én munkám, természetes hát, ha az eredményt az erősségeivel és hibáival együtt a sajátomnak érzem. Jócskán van ötletem, hogy mit lehetne még a keretrendszeren bővíteni, hogy mit lehetne jobban megvalósítani. Öröömre szolgál, ha a rendszert használják, hibáit és hiányosságait, vagy esetleg sikeres működését jelzik felém. Remélem, hogy a kódbázis ezután is fejlődni fog és rajtam kívül is lesz valakinek igénye a később felmerülő hibáit kijavítani, vagy új funkciókkal bővíteni azt.

Köszönetnyilvánítás

Szeretnék köszönetet mondani Klenik Attilának, akinek a projekttel kapcsolatban semmilyen kötelezettsége nem volt, viszont a megvalósítása során jelen volt, időt és energiát szánt arra, hogy az ötleteimet, terveimet meghallgassa, az alkalmazást kipróbálja és véleményezze.

Ábrák jegyzéke

1. ábra. A három fő használati eset.	24
2. ábra. A használati esetek felbontása.....	25
3. ábra. A kampány futtatás használati eset felbontása	25
4. ábra. A Master felhasználási esetei a Slave szolgáltatáson.....	26
5. ábra. A Mastert és Slave-et futtató gépek hálózati elrendezése	26
6. ábra. A bemeneti fájl két fő szekciója	27
7. ábra. A slave szekció részletei.....	27
8. ábra. Kampány szekció részletei	27
9. ábra. A Master működésének magas szintű folyamatábrája	28
10. ábra. A bemenet feldolgozásának folyamatábrája.....	28
11. ábra. Több Slave párhuzamos inicializálásának folyamatábrája.....	29
12. ábra. Slave inicializálásának folyamatábrája	29
13. ábra. Az ütemezés magas szintű folyamatábrája.....	30
14. ábra. Ütemezés egy Slave-re vetített folyamatábrája	31
15. ábra. Eredmények gyűjtése egy Slave-ről	31
16. ábra. Szekvenciadiagram az alkalmazás monitorozásáról	34
17. ábra. A Slave-ek inicializálásának tipikus szekvenciája	35
18. ábra. Aszinkron művelet kifejtése Slave kezelő komponenssel	36
19. ábra Az ütemező egy tipikus szekvenciája.....	37
20. ábra. A forráskód projektstruktúrája	44
21. ábra. Példa a kimeneti mappa hierarchikus szerkezetére	68

Irodalomjegyzék

- [1] Free Software Foundation, Inc, *The GNU Bash Reference Manual, for Bash, Version 4.3.*, 2014. február 2., online: <https://www.gnu.org/software/bash/manual/bashref.html>
- [2] Python Software Foundation, *The Python Language Reference – Python 3.5.1 documentation*, 2016. május 18., online: <https://docs.python.org/3.5/reference/>
- [3] Ecma International, *ECMAScript® 2015 Language Specification*, 2015. június, online: <http://www.ecma-international.org/publications/standards/Ecma-262.htm>
- [4] JetBrains s.r.o, *TeamCity 9.x Documentation*, <https://confluence.jetbrains.com/display/TCD9/TeamCity+Documentation> (2016. május)
- [5] Microsoft Developer Network, *C#*, <https://msdn.microsoft.com/en-us/library/618ayhy6.aspx> (2016. május)
- [6] Reiter István, *C# programozás lépésről lépésre*, ISBN 978-615-5012-17-4, Jedlik Oktatási Stúdió, Budapest, 2012.
- [7] Microsoft Developer Network, *Windows Communication Foundation*, <https://msdn.microsoft.com/en-us/library/dd456779.aspx> (2016. május)
- [8] Microsoft Developer Network, *ServiceModel Metadata Utility Tool (Svcutil.exe)*, <https://msdn.microsoft.com/en-us/library/aa347733.aspx> (2016. május)
- [9] Microsoft Developer Network, *Thread.Join Method*, <https://msdn.microsoft.com/en-us/library/system.threading.thread.join.aspx> (2016. május)
- [10] Microsoft Developer Network, *Asynchronous Programming Model (APM)*, <https://msdn.microsoft.com/en-us/library/ms228963.aspx> (2016. május)
- [11] Microsoft Developer Network, *Event-based Asynchronous Pattern (EAP)*, <https://msdn.microsoft.com/en-us/library/ms228969.aspx> (2016. május)
- [12] Microsoft Developer Network, *Task-based Asynchronous Pattern (TAP)*, <https://msdn.microsoft.com/en-us/library/hh873175.aspx> (2016. május)
- [13] Microsoft Developer Network, *Asynchronous Programming with async and await (C#)*, <https://msdn.microsoft.com/en-us/library/mt674882.aspx> (2016. május)
- [14] Dixin Yan, *Understanding C# async / await*, <https://weblogs.asp.net/dixin/understanding-c-sharp-async-await-1-compilation> (2016. május)

- [15] Stephen Cleary, *Async/Await - Best Practices in Asynchronous Programming*, <https://msdn.microsoft.com/en-us/magazine/jj991977.aspx> (2013. március)
- [16] Stephen Toub, *The Task-based Asynchronous Pattern*, Microsoft, 2012 február 29., online: <https://www.microsoft.com/en-us/download/details.aspx?id=19957>
- [17] Jaroslaw Kowalski, Kim Christensen, Julian Verdurmen, *NLog*, <http://nlog-project.org/> (2016. május)
- [18] Vincze Márton, *replaceFileContentsOnEachWrite does not work with createDirs properly*, <https://github.com/NLog/NLog/issues/1445> (2016. május)
- [19] Ecma International, *The JSON Data Interchange Format*, 1st edition, Október 2013., online: <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>
- [20] James Newton-King, *Json.NET Popular high-performance JSON framework for .NET*, <http://www.newtonsoft.com/json> (2016. május)
- [21] Microsoft Developer Network, *How to: Host a WCF Service in a Managed Application*, <https://msdn.microsoft.com/en-us/library/ms731758.aspx> (2016. május)
- [22] Microsoft Developer Network, *FileStream Class*, <https://msdn.microsoft.com/en-us/library/system.io.filestream.aspx> (2016. május)
- [23] Microsoft Developer Network, *How to: Enable Streaming*, <https://msdn.microsoft.com/en-us/library/ms789010.aspx> (2016. május)
- [24] Microsoft Developer Network, *Large Data and Streaming*, <https://msdn.microsoft.com/en-us/library/ms733742.aspx> (2016. május)
- [25] Microsoft, *Download Microsoft .NET Framework 4.5.2 (Offline Installer)*, <https://www.microsoft.com/en-us/download/details.aspx?id=42642> (2016. május)
- [26] Microsoft, *Visual Studio Downloads*, <https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx> (2016. május)
- [27] .NET Foundation, *Installing and Updating NuGet Client*, <https://docs.nuget.org/consume/installing-nuget> (2016. május)