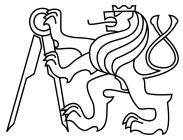
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Automatické testy pro vyhodnocení vlivu antiviru Avast na výkon počítače
Student:	Bc. Petr Kolář
Vedoucí:	Ing. Michal Vaněk
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2016/17

Pokyny pro vypracování

Cílem této práce je navrhnout a naimplementovat performance testy antiviru Avast a následně je začlenit do stávající testovací infrastruktury a provést potřebná měření.

Postupujte v následujících krocích:
1. Nastudujte techniky testování výkonu aplikací (performance testing).
2. Proveďte analýzu komponent antiviru Avast a současného testovacího prostředí (infrastruktury) ve společnosti AVAST Software.
3. Po dohodě s vedoucím práce vyberte ucelenou skupinu těchto komponent a navrhněte podobu automatických testů měřící vliv antiviru Avast na výkon počítače.
4. Na základě návrhu vhodně implementujte dané testy.
5. Implementované řešení začleňte do existující testovací infrastruktury.
6. Po nasazení vyhodnoťte naměřené výsledky.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 30. listopadu 2015

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra Softwarového inženýrství

Diplomová práce

Automatické testy pro vyhodnocení vlivu antiviru Avast na výkon počítače

Bc. Petr Kolář

Vedoucí práce: Ing. Michal Vaněk

8. května 2016

Poděkování

Na tomto místě bych v první řadě chtěl poděkovat všem, kteří měli cokoliv společného s touto prací, jmenovitě Jaromíru Cvrčkovi, Tomáši Daňkovi a Lukáši Kučerovi za vývoj a údržbu infrastruktury, Ondrovi Dobiášovi za údržbu databáze a REST API, Luboši Hnaníčkovi za zodpovídání všech možných i nemožných dotazů a v neposlední řadě Vláďovi Kašparovi za mentorování. Chtěl bych také poděkovat Michalu Vaňkovi, který se ukázal býti stejně dobrým vedoucím mé práce jako mým nadřízeným a neúnavně po mně vyžadoval demo. Na závěr bych chtěl poděkovat své rodině, přátelům a blízkým za respektování mých „volných“ víkendů strávených s touto kráskou.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 8. května 2016 .....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2016 Petr Kolář. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Kolář, Petr. *Automatické testy pro vyhodnocení vlivu antiviru Avast na výkon počítače.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Tato diplomová práce se zabývá problematikou performance testování a vlivem jednotlivých komponent antiviru Avast na výkon počítače. Popisuje analýzu, implementaci, nasazení a vyhodnocení výsledků automatických testů, které vychází ze zmíněné problematiky.

Klíčová slova performance testování, automatické testování, antivirus Avast, programování ve Windows

Abstract

This diploma thesis deals with performance testing and impact of Avast antivirus components on computer performance. It describes analysis, implementation, deployment and results evaluation of automation tests which are related to the mentioned issue.

Keywords performance testing, automation testing, Avast antivirus, Windows programming

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V roce 1988 založili spolu Eduard Kučera a Pavel Baudiš družstvo, pod jehož hlavičkou začali distribuovat první Avast antivirus. Po sametové revoluci převedli družstvo na firmu ALWIL Software a v roce 1995 napsal Ondřej Vlček, současný provozní ředitel, první antivirový program pro Windows 95. V roce 2001 vydali bezplatnou verzi antiviru a do roku 2004 získali první milión uživatelů. Uživatelská základna od té doby raketově rostla až přes hranici 200 miliónů. Avast se tak stal nejrozšířenějším antivirem na světě.

Na uživatelském fóru a ve zpětné vazbě se začaly objevovat stížnosti na pomalé skenování souborů a webových stránek, zhoršený čas startu operačního systému a instalace programů a pomalu odpovídající uživatelské rozhraní. Dále stanovil Microsoft všem antivirovým firmám podmínky, za jakých mohou být jejich programy instalované na nově vydaný operační systém Windows 10. Avast ovšem některé z těchto podmínek stejně jako další antiviry nesplňoval.

Napříč odděleními tak byli vyhrazeni lidé, kteří mají za úkol analyzovat hlavní problémy a vypořádat se s nimi. Jednou z částí pro analýzu je i tato práce, která se ve své praktické části věnuje automatizaci nejčastějších uživatelských scénářů a měřením jejich výsledků, které jsou následně vyhodnoceny.

Po dobu celé práce je používán pojem performance, který se za normálních okolností váže k výkonu testované aplikace (například webové). V kontextu této práce se ovšem jedná o performance počítače. Klademe si tedy otázku, jakým způsobem ovlivňuje Avast výkon počítače a uživatelovy běžné aktivity při jeho používání?

Cíle

Hlavním cílem této diplomové práce je navrhnout a naimplementovat performance testy antiviru Avast a následně je začlenit do stávající testovací infrastruktury a provést potřebná měření. Tento cíl se skládá z následujících částí:

1. Poskytnout teoretický základ pro performance testování a představit jednotlivé komponenty antiviru Avast.
2. Na základě získaných informací provést návrh a implementaci automatických performance testů.
3. Implementované řešení začlenit do existující testovací infrastruktury a vyhodnotit naměřené výsledky.

Struktura

Práce se skládá ze dvou psaných částí – teoretické a praktické – a části implementační.

Teoretická část je dělí na čtyři kapitoly. První rozebírá techniky performance testování a jednotlivé typy performance testů. Druhá kapitola prochází základní aspekty programování ve Windows a některé API funkce důležité pro tuto práci. Třetí kapitola představuje komponenty Avastu, které mohou mít zásadní vliv na performance počítače. Čtvrtá kapitola pak popisuje nástroje, které byly použity pro dosažení výše zmíněného cíle.

Praktická část sestává ze do tří kapitol. První z nich je analýzou současné testovací infrastruktury a dále zahrnuje její potřebnou konfiguraci pro spuštění testů. Druhá kapitola popisuje návrh a implementaci samotných testů a problémy s tím spjaté. Závěrečná kapitola pak vyhodnocuje naměřené výsledky a poskytuje vodítka pro vývojáře na zlepšení performance.

V implementační části byly vytvořeny automatické testy, které byly navrženy tak, aby je bylo možné začlenit do infrastruktury a opakovatelně je pouštět. Nad rámec zadání byl také vytvořen nástroj pro zobrazování a interpretaci výsledků.

Část I

Teoretická část

Kapitola 1

Performance testování

První kapitola je zasvěcena samotnému performance testování. Úvodní část shrnuje důležité základní pojmy, principy a druhy performance testů, druhá část představuje ideální postup, podle kterého by měl postupovat každý, kdo by chtěl začít s performance testováním své aplikace. Zdroje citované v rámci této kapitoly se zaměřují na webové aplikace, nicméně použité pasáže jsou obecně platné.

1.1 Úvod do performance testování

„Performance testování je definováno jako technický výzkum, který má za úkol zjistit nebo porovnat rychlost, škálovatelnost anebo stabilitu produktu, nad kterým je puštěn test.“ [1]

Ian Molyneaux ve své knize [2] definuje performance z pohledu uživatele: „Dobře fungující[1] aplikace je ta, která dovolí koncovému uživateli provést zadaný úkol bez zaznamenání nepatřičné prodlevy nebo rozčílení.“

1.1.1 Performance ukazatele

Pro objektivnost měření je potřeba vzít v potaz některé indikátory, které jsou součástí nefunkčních požadavků. Tyto identifikátory se dělí na orientované na služby (jak dobře poskytuje aplikace své služby uživateli), což jsou dostupnost a doba odezvy, a na orientované na efektivitu (jak dobře aplikace využívá infrastrukturu, na které běží), kam patří propustnost a zátěž. [2] Nyní trochu podrobněji k zmíněným indikátorům:

- Dostupnost – Doba, po kterou je aplikace dostupná uživateli. Po překročení neúnosně dlouhého časové úseku nebo pravidelných výpadků je uživatel odrazen od používání takovéto aplikace.

[1] well-performing

- Doba odezvy – Časové kvantum, které uběhne od uživatelova požadavku až do odpovězení aplikace. Takováto odpověď může být buď synchronní (blokující, uživatel musí čekat před provedením další akce), nebo asynchronní (uživatel nemusí čekat na odpověď před zadáním dalších požadavků).

- Propustnost – Počet uživatelských požadavků na aplikaci za určitý časový úsek.

- Zátěž – Procentuální vyjádření využití dostupných prostředků aplikací.

1.1.2 Performance standardy

Kniha [2] shrnuje výzkum z osmdesátých let 19. století, který se pokoušel sledovat závislost uživatelovy produktivity na době odezvy. Dá se říci, že závěry tohoto poněkud staršího výzkumu však platí dodnes:

- Více než 15 vteřin – Takto dlouhá doba vylučuje jakoukoliv interakci, která by měla probíhat ve smyslu „konverzace“ s aplikací. Pokud se tato prodleva objevuje, měl by nabídnout uživateli jiné aktivity do té doby, než se mu dostane odpovědi.

- Více než 4 vteřiny – Odezva je stále velká na to, aby uživatel udržel ve své krátkodobé paměti všechny informace potřebné ke „konverzaci“. Tento čas je nicméně tolerovatelný, pokud čekání je vyžadováno až po dokončení ucelené transakce.

- 2–4 vteřiny – Prodleva delší než dvě vteřiny může zamezovat uživateli vykonávání úkolů, které vyžadují vyšší míru koncentrace. Na druhé straně je dostatečně krátká, aby uživatele tolik neobtěžovala po uzavření menšího celku.

- Méně než dvě vteřiny – Tato hodnota odezvy je mezní hranicí pro zapamatování více informací, které byly každá odděleny tímto časovým úsekem.

- Méně než vteřina – Existují tvůrčí aktivity, u kterých uživatel potřebuje vidět výsledek svého umění co nejdříve, aby mohl nerušen pokračovat ve svém díle (psaní knihy, přidávání obrázků).

- Desetina vteřiny – Téměř okamžitá odezva je typicky vyžadována pro stisk kláves a tlačítek myši, obzvláště pro aktivity vyžadující dobrý postřeh (počítačové hry).

1.1.3 Kdy začít testovat

Kniha [2] dále představuje data shromážděná společností Forrester Research z roku 2006, která identifikují tři druhy performance testování na základě včasnosti.

První je metoda „hašení otevřeného ohně“[2], jež identifikuje a napravuje vzniklé problémy až v produkčním prostředí. Společnosti řídící se tímto přístupem, kterých ovšem není málo, se vystavují vysokému riziku.

Druhou metodou je kontrola/ověření performance těsně před vydáním nové verze s velmi malou časovou dotací, což zapříčiňuje nalezení až 30 % problémů v produkci. Tento přístup může způsobit zpoždění vydání kvůli řešení nalezených problémů, nebo hůře zrušení nasazení do té doby, než budou všechny problémy opraveny. Tímto směrem se ubírá většina dnešních organizací.

Poslední metoda bere v potaz performance testování v průběhu celého životního cyklu aplikace. Díky tomu je jenom minimum problémů nalezených po nasazení (5 %). Toto je cesta, kterou by se měly vydat ideálně všechny společnosti.

1.1.4 Nevýhody manuálního testování

Pokud bychom nepoužili automatické testy, museli bychom mít dostatečný počet lidí, který by 24 hodin denně, 7 dní v týdnu simuloval chování takového testu, a stát nad nimi se stopkami. Nicméně při takovémto testování by nebylo možné navodit stejný scénář dvakrát za sebou, natož vyhodnocovat chování uvnitř aplikace a nasbíraná data. V neposlední řadě si nemůžeme dovolit nechat pracovat zaměstnance týden v kuse bez přestání. [2]

1.2 Vyčlenění časového úseku

Je velmi důležité si v projektu naplánovat dostatečné množství času pro performance testování, proto je třeba zvážit následující [2]:

- Čas na přípravu testovacího prostředí – Finanční a časové náklady jsou minimální za předpokladu, že v rámci společnosti již existuje nějaké funkční řešení. Na straně druhé se musí začít s testovacím prostředím na „zelené louce“, což zahrnuje získání a konfiguraci patřičného hardwaru a následně instalaci a konfiguraci aplikace, kterou chceme testovat.
- Čas na identifikaci a implementaci scénářů – Tento časový úsek je životně důležitý pro celé performance testování. Identifikace scénářů může trvat

[2] firefighting

dny až týdny, kdežto samotná implementace jednoho scénáře zabere den. Následné ladění se může opět vyšplhat do hodnot podobných identifikaci.

- Čas na identifikaci a vytvoření dostatečného množství testovacích dat – Správná data mohou být klíčovým prvkem úspěšného performance testování. Najít je nebo vytvořit může ovšem trvat dny až týdny. Nutností je také zvážit, jak dlouho trvá tato data znovu uvést do původního stavu předtím, než začne nový test.
- Čas na přípravu prostředí pro monitorování – Tato část zahrnuje instalaci a konfiguraci jakýchkoliv nástrojů, které budou v průběhu testu monitorovat chování aplikace a jejího prostředí, v němž se nachází.
- Čas na vyřešení identifikovaných problémů – Nedá se časově vyčíslit, jakou dobu může řešení trvat, protože se potýkáme s něčím novým, neznámým. Do tohoto času se mimo jiné započítává čas vývojářů, po který se budou snažit danou chybu opravit. Může se stát, že v případě využívání nástrojů třetích stran bude potřeba řešit problémy vzniklé v důsledku používání jejich aplikací.

1.3 Získání code freeze

Je naprosto nesmyslné testovat aplikaci, která se mění „pod rukama“, proto je zapotřebí mít během testování konzistentní kód aplikace. Pokud je během code freeze[3] nalezena chyba, která musí být opravena, neznamená to, že tam zůstane až do příští verze, ale v případě jakékoliv změny aplikace musí být uvědoměni lidé, kteří provádějí testování. Automatické testy jsou totiž vytvořené na míru dané verze aplikace, takže zásadní změny v aplikaci by mohly mít za následek neschopnosti spuštění nebo vyhodnocení testů. S každou větší změnou je třeba testy upravit tak, aby stále odpovídaly současnému stavu testované aplikace. Týmy vývojářů a testerů tedy spolu musí ohledně jakýchkoliv změn komunikovat. [2]

1.4 Typy performance testů

Tato sekce vychází z [1] a [2] a shrnuje základní typy performance testů. Vzhledem k tomu, že se běžně používají původní názvy těchto typů a de facto tak nemají český ekvivalent, jsou uvedeny v anglickém jazyce.

1.4.1 Pipe-clean test

Pipe-clean test se používá ze dvou důvodů. Prvním z nich je kontrola testovacího skriptu, že se chová takovým způsobem, jak byl navržen. Druhým

[3] Období, ve kterém jakákoliv změna musí projít kontrolou před tím, než je zakomponována do stávajícího kódu.

důvodem je zjištění, jak se testovaný systém chová při jednoduchém opakovaném průchodu bez přidaných podmínek, omezení a zátěže. Naměřenou hodnotu lze pak použít jako základ, ze kterého lze pro ostatní typy spočítat relativní zhoršení oproti této hodnotě.

1.4.2 Volume/Load test

Tento druh testu zjišťuje chování aplikace za normálních okolností a nepřekračuje limit, za kterým je známo, že aplikace nepodává dobrý výkon, tj. bod do něhož aplikace splňuje naše očekávání na dostupnost, propustnost, dobu odezvy a zátěž. Volume test je tedy co nejpřesnějším napodobením reálného používání aplikace. Podmnožinou je endurance test, který se chová naprosto stejně, ale je puštěn značně delší dobu v kuse.

1.4.3 Stress test

Stress test má za úkol vytížit aplikaci mnohem více, než na co je navržená nebo používaná, aby byly odhaleny chyby, které mohou vzniknout jenom za takto extrémních podmínek. Kromě nalézání chyb lze díky tomuto typu testů zjistit, jaké jsou horní hranice aplikace, za kterými již nefunguje nebo ji nelze zčásti používat. Dále také zjišťuje, zdali nedochází k poškození dat nebo vzniku bezpečnostních trhlin. Podmnožinou je spike test, který opakovaně vytěžuje aplikaci v krátkých intervalech.

1.4.4 Soak/stability test

Soak test odhaluje chyby, které vznikají až po delší době chodu aplikace. Typicky se jedná o chyby jako memory leak[4] nebo takové chyby, které se objeví po několikanásobném zopakování stejného postupu v různých časových odstupech. Dalšími příznaky může být celkové zpomalení aplikace nebo její nedostupnost.

1.4.5 Smoke

Smoke test se zaměřuje pouze na poslední provedené změny v kódu. Proto pro měření performance jsou použity pouze ty scénáře, kterých se tyto změny dotkly.

1.5 Postup pro performance testování

Oba zdroje [1] a [2] použité pro tuto sekci se až na mírné odchylky shodují na „univerzálním“ postupu. Následující kroky jsou proto sjednocením informací obsažených ve zmíněných materiálech.

[4]Pozůstatek alokované paměti, který nebyl uvolněn a zabírá systémovou paměť až do ukončení aplikace.

1.5.1 Vytvoření testovacího prostředí

V prvním kroku je potřeba vytvořit takové testovací prostředí (zahrnující hardware, software a nastavení sítě), které odpovídá tomu reálnému produkčnímu. To ovšem není možné vzhledem k rozmanitosti hardwaru uživatelů a také některým speciálním nastavením, které zajistí co nejmenší ovlivnitelnost testů, ale zároveň neodpovídají nastavením uživatelů. Proto je dobré vytvořit takové prostředí, které má průměrný uživatel dané aplikace. Vstupní data a konfigurace aplikace by také měla co nejvěrohodněji kopírovat reálné hodnoty. Není na škodu provést jednou za čas revizi, která vezme v potaz měnící se uživatelskou základnu.

1.5.2 Identifikace požadavků

Druhým úkolem je zjistit, jaké jsou požadavky na ideální hodnoty pro již zmíněnou dostupnost, dobu odezvy, propustnost a zátěž. Navíc lze identifikovat cíle, které nejsou zachyceny těmito hodnotami, ale mohou být měřeny pomocí performance testů, jako je například ideální nastavení konfigurace, při němž aplikace pracuje nejlépe. Mezi předpoklady, které musí být splněny, pro přesunutí do další fáze patří mimo jiné tyto:

- Dostupná data předpokládaného dokončení a nasazení aplikace.
- Domluvený code freeze v rámci každého vývojového cyklu.
- Testovací prostředí není ovlivnitelné v průběhu testování. V opačném případě by došlo k znehodnocení výsledků.
- Existuje postup, jakým budou nahlašovány chyby a performance nedostatky nalezené v průběhu testů.

1.5.3 Návrh testů

Návrh performance testů zahrnuje identifikaci klíčových uživatelských scénářů a zajištění jejich rozmanitosti napříč uživateli, identifikaci a vytvoření testovacích dat a specifikace všech typů dat, která se budou dále vyhodnocovat. Uživatelské scénáře by měly odpovídat klasickým a nejčastějším úkonům uživatelů, aby výsledky vytvářely jasný obraz, jak si aplikace stojí v reálném světě. Některé z těchto scénářů samy „vyplavou na povrch“ v průběhu identifikace požadavků. Na straně druhé jsou syntetické testy, které sice neodpovídají typickému uživatelskému scénáři, nicméně díky nim se dají odhalit slabiny, které by jinak nebyly detekovány.

1.5.4 Implementace testů

Konkrétní implementace je závislá na prostředcích a nástrojích, které byly pro její uskutečnění použity, a také na specifických vlastnostech testované

aplikace. Nezávisle na předchozím je třeba vždy vytvořit nejdříve samostatné scénáře, které po odladění jsou zkombinovány tak, aby celek představoval uživatelovo chování. Vždy je lepší věnovat více času tomu, aby implementace odpovídala návrhu, než ohýbat návrh samotný – z dlouhodobého hlediska se ušetří více času a nervů. V neposlední řadě je třeba zajistit, aby opakované spuštění testu se chovalo, jako kdyby byl test puštěn poprvé.

1.5.5 Spuštění testů

Může se zdát, že tento krok je pouhé spuštění a kontrola, jestli vše funguje v pořádku. Ve skutečnosti je vhodné projít všemi následujícími dílčími úkoly:

1. Koordinovat a kontrolovat spuštění v rámci týmu.
2. Zkontrolovat testy, konfigurace a stav testovacího prostředí a dat.
3. Spustit testy.
4. V průběhu testu kontrolovat skript, systém a data.
5. Po skončení testu v rychlosti projít výsledky a najít případné nesrovnalosti.
6. Archivovat testy, data, výsledky a jakékoliv další informace potřebné pro případné zopakování testu.
7. Zaznamenat čas spuštění a ukončení pro následné zařazení výsledků na časovou osu.

1.5.6 Analýza výsledků

V posledním kroku cyklu je potřeba sesbírat a vyhodnotit všechny získané výsledky. Na základě těchto výsledků mohou být kladně či záporně ohodnoceny změny v aplikaci v oblasti performance, které byly zakomponovány od posledního testování. Dále je také vhodné zvážit priority pro příští otestování jednotlivých scénářů vzhledem k příslušným výsledkům.

Kapitola 2

Programování ve Windows

Druhá kapitola se věnuje základním principům a konkrétním funkcím, které se běžně využívají při programování ve Windows, zároveň však je jejich rozsah omezen podle potřeb této práce. Celá tato kapitola vychází z knihy *Windows System Programming* [3], pokud není uvedeno jinak.

2.1 Unicode

Každý znak má v počítači svoji číselnou hodnotu. Tato hodnota se liší podle použité znakové sady, kterých před zavedením Unicode byly stovky. Žádná z těchto sad ovšem nebyla schopná pojmout všechny znaky, proto pod stejnými číselnými hodnotami v různých kódováních jsou různé znaky. Při přesunu dat na jinou platformu proto hrozilo, že data budou špatným kódováním znehodnocena. Unicode poskytuje každému znaku unikátní hodnotu bez ohledu na platformu, program nebo jazyk. [4]

Operační systémy Windows podporují jak 8bitové znaky (datový typ CHAR, ANSI) tak 16bitové (datový typ WCHAR, Unicode), které nazývá „široké znaky“. Kódování UTF-16 dokáže pojmout znaky a symboly většiny jazyků na světě.

2.1.1 Generické znaky

Při dodržení následujících bodů při vytváření aplikace lze takovýto program přeložit pro 8bitové i 16bitové znaky:

- Všechny znaky definovat pomocí generického typu TCHAR a řetězce LPTSTR, popřípadě LPCTSTR pro konstantní řetězce.
- Použít direktivy `#define UNICODE` a `#define _UNICODE` tam, kde chceme využívat Unicode znaky. V opačném případě bude TCHAR přeložen jako CHAR.

- Velikost znaku pro načítání řetězce z bufferu[5] zjistit pomocí `sizeof(TCHAR)`.
- Pro operace se znaky a řetězci použít generické funkce jako `_tprintf`, `_tcscpy` a další.
- Konstantní řetězce, které mají být vždy 8bitové, nijak neoznačovat, pro čistě Unicode řetězce napsat před něj „L“. Generický řetězec vznikne obalením makrem `_T()`.

Windows API funkce, které přijímají nebo vracejí řetězce, jsou v generické podobě makrem. Například funkce `CreateFile` je přeložena buď jako `CreateFileW`, nebo `CreateFileA` podle toho, jestli řetězce jsou Unicode, nebo nejsou.

2.1.2 Unicode podpora v Python

Řetězce od Python 3.0 jsou nativně v Unicode. K vytvoření ANSI řetězce je zapotřebí explicitně před něj dát „b“, stejně tak jako pro vytvoření Unicode řetězce ve starších verzích bylo nutné umístit před něj „u“. [5]

Windows API funkce jsou v Python přístupné skrze knihovnu c_types, která obsahuje většinu dostupných volání a datových typů. Když funkce vyžaduje parametr typu LPSTR (ANSI), předáme řetězec vytvořený skrze funkci `create_string_buffer()`. Naopak pokud je parametr typu LPWSTR (Unicode), použijeme funkci `create_unicode_buffer()`, kde každý znak bude mít dva bajty. [6]

2.2 Operace se soubory

Tato sekce popisuje použití čtyř základních API funkcí pro práci se soubory: `CreateFile`, `CloseHandle`, `ReadFile` a `WriteFile`.

2.2.1 Vytváření a otevírání souborů

Pomocí funkce `CreateFile` lze vytvářet a otevírat soubory. Chování této funkce se liší na základě přijatých parametrů, kterých je celkem sedm. První přijímá cestu k novému nebo již existujícímu souboru. Druhý určuje pomocí konstant GENERIC_READ a GENERIC_WRITE, jestli bude soubor otevřený pro čtení anebo pro zápis. Třetím parametrem lze nastavit přístup ostatních procesů ve chvíli, kdy je konkrétní soubor otevřený, a může nabývat hodnot nula (není povolen žádný přístup) nebo FILE_SHARE_READ (může být čten více procesy) anebo FILE_SHARE_WRITE (více procesů může zároveň zapisovat). Pátý parametr určuje, zdali má být soubor vytvořen,

[5] Dočasná paměť pro operace, u kterých není předem známá délka výstupu

pokud neexistuje (CREATE_NEW), vždy vytvořen (CREATE_ALWAYS), otevřen, pokud existuje (OPEN_EXISTING), otevřen, pokud existuje, jinak vytvořen (OPEN_ALWAYS), nebo zmenšen na velikost nul (TRUNCATE_EXISTING). Čtvrtý, šestý a sedmý parametr nejsou pro tuto práci podstatné. Návratovou hodnotou je HANDLE na otevřený soubor nebo hodnota INVALID_HANDLE_VALUE v případě neúspěchu.

2.2.2 Zavírání souborů

Funkce `CloseHandle` neslouží pouze k zavírání souborů, ale také jakéhokoliv objektu jádra[6], na který získal uživatel handle skrze nějaké API volání. Při pokusu zavřít neplatný handle nebo zavřít dvakrát stejný handle vrátí funkce FALSE.

2.2.3 Čtení ze souboru

Po otevření souboru a získání jeho handle lze funkcí `ReadFile`, které se jako první parametr předá daný handle, číst z otevřeného souboru. Druhým parametrem se předá ukazatel na buffer, kam se uloží získaná data. Třetí parametr říká, kolik bajtů chce uživatel načíst, a čtvrtý dá po skončení vědět, kolik ve skutečnosti bylo načteno. Pátý parametr není pro tuto práci důležitý. Návratová hodnota při úspěchu nabývá hodnoty TRUE a v opačném případě FALSE.

2.2.4 Zápis do souboru

Pokud chceme zapisovat do souboru, vystačíme si se znalostí čtení ze souboru. Funkce `WriteFile` přijímá totiž ve výsledku stejné parametry jako `ReadFile` s tím rozdílem, že třetí a čtvrtý parametr jsou počty bajtů k zapsání a zapsaných.

2.3 Soubory mapované do paměti

Díky souborům mapovaných do paměti může program pracovat s dynamickými daty výhodněji (než při klasické práci se soubory) a transparentně a zároveň data ze souboru lze zpracovávat algoritmy, které jsou určené pro práci s pamětí (řazení, vyhledávání, zpracování řetězců atd.). Mapování do paměti také přináší výrazné zrychlení zpracovávání souborů a poskytuje mechanismus pro sdílení paměti mezi různými procesy. Mezi výhody tohoto zpracování souborů patří:

- Není potřeba provádět přímé operace pro čtení a zápis.

[6] kernel

- Paměťové operace nad souborem lze provádět, i když je daný soubor větší než dostupná paměť.
- Zpracování souboru je mnohem rychlejší než při použití funkcí zmíněných v minulé sekci.
- Není potřeba se starat o buffery a data, kterými se plní. Tuto funkcionalitu efektivně a spolehlivě plní za uživatele operační systém.
- Procesy mohou sdílet svoji paměť namapováním svého virtuálního adresového prostoru do stejného souboru nebo stránkovacího souboru.

Při čtení nebo zápisu mapovaného souboru může být vyvolána SEH výjimka EXCEPTION_IN_PAGE_ERROR, pokud například dojde k odebrání cílového úložiště nebo je daný soubor poškozený. Získané handle lze zneplatnit pomocí dřív zmíněného `CloseHandle`.

2.3.1 Vytvoření mapování

Pro vytvoření nového mapování použijeme funkci `CreateFileMapping`, která přijímá šest parametrům. Prvním parametrem je handle (získaný z `CreateFile`) na souboru, který chceme mapovat. Druhým parametrem se nastaví zabezpečení, třetím jeden z následujících přístupů:

- PAGE_READONLY – program může pouze číst data v mapovaném regionu, nemůže do nich zapisovat ani je spouštět. Asociovaný soubor musí mít nastavený přístup pro čtení.
- PAGE_READWRITE – program má plný přístup k souboru, pokud původní soubor měl povolení pro zápis i čtení.
- PAGE_WRITECOPY – pokud proces změní obsah mapovaného souboru, vytvoří si vlastní kopii a změny promítne do ní.

Čtvrtý a pátý parametr[7] udávají velikost namapovaného objektu. Pokud oba nabývají hodnoty nula, je použita velikost původního souboru. Poslední parametr je název mapování, jenž slouží pro identifikaci mezi procesy, které by ho chtěly také používat. Pokud není sdílení potřeba, hodnota jména je NULL.

2.3.2 Otevření existujícího mapování

Pokud mapování již existuje a má svůj název, lze získat na něj handle skrze funkci `OpenFileMapping`. Pokud mapování s daným jménem neexistuje, návratová hodnota je NULL. První parametr udává požadovaný přístup, který je zkontrolován oproti tomu, který byl zadán při `CreateFileMapping`. Druhý parametr není podstatný pro tuto práci a poslední je název mapování.

[7] 64bitovou velikost nelze zadat jedním číslem, ale je potřeba ji rozdělit na dvě 32bitové

2.3.3 Mapování do adresového prostoru

Po vytvoření mapování je ještě potřeba namapovat získaný objekt do virtuálního adresového prostoru procesu a získat tak počáteční adresu pohledu[8]. To lze zařídit pomocí funkce `MapViewOfFile`, jež jako první parametr bere handle na mapovací objekt, který byl vytvořen jednou z množností v předchozích dvou podsekcích. Druhý parametr je opět požadovaný přístup, který musí souhlasit. Třetí a čtvrtý parametr dohromady udávají místo, odkud bude mapovaný soubor načítán (nula znamená od začátku). Posledním parametrem můžeme říct, jak velké mapování chceme vytvořit (nula značí celý soubor).

Pokud bychom potřebovali okamžitě zapsat současné změny do souboru, využijeme funkce `FlushViewOfFile`.

2.3.4 Uzavření mapování

Jakmile skončíme s prací nad souborem, musíme uvolnit alokovanou paměť (`UnmapViewOfFile`) a zavřít všechny handle (`CloseHandle`).

2.4 Dynamické knihovny

Dynamické knihovny jsou připojeny k programu až za běhu, takže musí být v kódu explicitně řečeno, kdy mají být nahrané do paměti a kdy uvolněné. Dále je potřeba získat adresu požadované funkce, kterou chceme z knihovny používat, a tu uložíme do námi vytvořeného ukazatele na funkci, který budeme k volání dané funkce využívat.

Mezi hlavní výhody dynamických knihoven oproti statickým patří:

- Díky připojení knihovny až za běhu programu je výsledný program menší, protože neobsahuje funkce knihovny.
- Dynamické knihovny mohou být využité jako sdílené, to znamená, že více programů může využívat jednu knihovnu, přitom pouze jedna kopie je nahraná v paměti. Každý proces má ovšem vlastní kopii globálních proměnných této knihovny.
- Upravením implementace knihovny není potřeba měnit programy, které ji využívají, pokud nebylo změněno rozhraní.
- Knihovna poběží ve stejném procesu jako program, který ji využívá.

[8] view

2.4.1 Nahrání do programu

Dynamická knihovna se dá nahrát pomocí jedné ze dvou funkcí `LoadLibrary` a `LoadLibraryEx`. Obě dvě přijímají jako první parametr cestu k dané knihovně, jež chceme nahrát. Druhá zmíněná funkce má ještě druhý parametr, který je rezervovaný pro operační systém, a třetí parametr, kterým se dají nastavit některé z těchto hodnot:

- LOAD_WITH_ALTERED_SEARCH_PATH – První místo, kde se knihovna hledá, není složka s programem, ale cesta specifikovaná prvním parametrem.
- LOAD_LIBRARY_AS_DATAFILE – Načtená knihovna bude sloužit pouze pro data, tudíž nebudeme moci z ní nic spouštět.
- DONT_RESOLVE_DLL_REFERENCES – Další knihovny, které má daná knihovna načíst (je na nich závislá), nejsou načteny.

Pomocí získaného handle a funkce `GetModuleFileName` lze získat název knihovny. Pokud naopak máme název a knihovna je již nahrána, lze získat její handle pomocí `GetModuleHandle`.

2.4.2 Získání adresy funkce

Funkcí `GetProcessAdress` lze získat adresu funkce, kterou chceme používat. Prvním parametrem je handle na knihovnu, v níž se nachází, a druhý je samotný název funkce. V případě neúspěchu vrací NULL.

2.4.3 Uvolnění knihovny

Ve chvíli, kdy již nebudeme pracovat s nahranou knihovnou, uvolníme ji pomocí `FreeLibrary`, což zaručí, že budou uvolněny všechny prostředky, které tato knihovna alokovala. Pokud knihovnu využívá více procesů, zůstane stále nahraná v paměti.

2.5 Práce s procesy

„Proces obsahuje svůj vlastní nezávislý prostor virtuálních adres jak pro kód, tak data, který je chráněný před ostatními procesy. Dále také každý proces obsahuje jedno nebo více nezávislých vláken. Vlákno běžící v rámci procesu může provádět kód aplikace, vytvářet nová vlákna a nezávislé procesy nebo spravovat komunikaci a synchronizaci mezi vlákny.“ [3]

2.5.1 Vytváření procesů

Funkce, která zajišťuje vytvoření procesu a jeho prvního vlákna, se nazývá `CreateProcess` a přijímá deset parametrů. První parametr je název programu, který má být spuštěn, a je doplňován druhým parametrem, jenž určuje argumenty programu. Třetí a čtvrtý parametr udávají zabezpečení procesu a vlákna, kde hodnota NULL zajistí implicitní nastavení. Pátý parametr říká, zdali nový proces zdědí otevřené handle objektů. Hodnota šestého parametru určuje, jak má být proces spuštěn:

- CREATE_SUSPENDED – hlavní vlákno bude vytvořeno jako uspané a začne pracovat až po zavolání `ResumeThread`.
- DETACHED_PROCESS a CREATE_NEW_CONSOLE – První hodnota zajistí vytvoření procesu bez konzole a druhá vytvoří proces s vlastním oknem pro výstup. Nastavení těchto hodnot je výlučné, pokud není ani jedna, proces zdědí konzoli svého předka.
- CREATE_UNICODE_ENVIRONMENT – nastavujeme, pokud pracujeme s Unicode řetězci.
- CREATE_NEW_PROCESS_GROUP – nový proces se stane předkem nové skupiny procesů. Není podstatné pro tuto práci.

Sedmý parametr je ukazatel na prostředí nového procesu, které díky hodnotě NULL může být zděděné od předka. Osmým parametrem se nastavuje výchozí složka. Pokud je hodnota NULL, použije se složka předka. Devátý parametr nastavuje vzhled hlavního okna a handle na standardní zařízení. Pro potřeby práce stačí naplnit pomocí `GetStartupInfo`, což poskytne hodnoty předka. Desátý a zároveň poslední parametr je struktura, která se naplní po vytvoření procesu. Obsahuje handle na nový proces a jeho vlákno a jejich ID.

2.5.2 Otevření existujícího procesu

Pokud proces již existuje, lze na něj získat handle pomocí `OpenProcess`. První parametr určuje přístup k procesu. Nejčastější jsou tyto hodnoty:

- SYNCHRONIZE – povolí čekání současného procesu do té doby, než bude otevřený proces ukončen.
- PROCESS_ALL_ACCESS – nastaví úplný přístup.
- PROCESS_TERMINATE – povolí ukončit proces pomocí `TerminateProcess`.
- PROCESS_QUERY_INFORMATION – handle procesu lze předat funkci `GetExitCodeProcess` pro získání návratového kódu procesu.

Druhý parametr určuje, zdali se může z handle procesu dědit. Poslední parametr je ID procesu, který chceme otevřít a získat tak jeho handle.

2.5.3 Ukončení procesu

Když proces dokončí práci, může zavolat funkci `ExitProcess`, které předá návratovou hodnotu. Tímto voláním se ukončí i všechna vlákna vytvořená tímto procesem. Proces lze ukončit i z vnějšku pomocí `TerminateProcess`, kde se předá handle na daný proces a požadovaná návratová hodnota.

Pro zjištění návratové hodnoty ukončeného procesu slouží funkce `GetExitCode Process`, které předáme handle na proces a druhý parametr naplní návratovým kódem.

Před ukončením procesu je důležité dbát na řádné uvolnění všech sdílených prostředků.

Kapitola 3

Komponenty Avastu

Třetí kapitola shrnuje důležité komponenty antiviru Avast, které mohou mít viditelný vliv na výkon počítače. Informace v této kapitole jsou založeny na konzultacích s osobami zodpovědnými za jednotlivé komponenty jak v oblasti vývoje, tak testování.

3.1 FileRep

FileRep je součást engine, jež pro podezřelé spustitelné vzorky spočítá hash (UID), kterou pošle na server s databází virů, který na základě této databáze rozhodne, zdali se jedná o malware. FilRep je použit teprve až při pokusu o spuštění souboru, pokud není nalezen záznam ve VPS (lokální virová databáze), stringová kontrola nenašla problém, soubor není ve white listu nebo USN journalu.

3.2 Štít souborového systému

Štít souborového systému je komponenta, která má na starosti diskové operace. Ovladače této komponenty se zaregistrují ve Filter Manager, který jim zprostředkovává notifikace o různých operacích, jako jsou otevření, čtení, zápis, spuštění nebo zavření souboru. Při každém takovém volání se o něm pošlou informace do engine Avastu, který vyhodnotí, jestli je dané volání v pořádku, nebo není. Nejdříve se zkontroluje, jestli není vzorek v lokální virové databázi (VPS) nebo v USN cache, a provede se stringová detekce. Pokud během doby, kdy byl soubor otevřen, nedošlo k jeho modifikaci, vzorek se považuje stále za čistý. Je-li vše do tohoto okamžiku v pořádku, spočítá se hash souboru a pošle se dotaz na FileRep. Pokud se jedná o neznámý vzorek anebo má nízkou prevalenci[9], spustí se v DeepScreen.

[9]Pojem převzatý z biologie – podíl počtu jedinců trpících danou nemocí a počtu všech jedinců ve sledované populaci

3.2.1 Filter Manager

Filter Manager je kernel-mode ovladač, který je přizpůsoben původnímu filtrovacímu modelu souborového systému a vystavuje rozhraní pro ostatní ovladače. Vývojáři těchto ovladačů pak mohou jednoduše implementovat dané rozhraní a vytvořit takzvané minifilter ovladače, které se ve Filter Manager zaregistrují. [7]

3.2.2 USN journal

Kdykoliv je provedena změna jakéhokoliv souboru nebo složky na disku, do USN journalu se zapíše popis této změny a název daného souboru či složky. [8] Avast si při skenování souborů vytváří cache, do které ukládá pro každý soubor identifikátor, pomocí kterého přistupuje do USN journalu, aby zjistil, zdali se již skenovaný soubor nezměnil. Pokud ano, je znova otestován.

3.3 DeepScreen

Komponenta DeepScreen je rozdělena na dvě technologie: původní Sandbox a novější NG. Pokud počítač splňuje určité požadavky, je využíváno NG. Ve chvíli, kdy není tato technologie dostupná, je použit Sandbox.

3.3.1 Sandbox

Podezřelý soubor je nejdříve spuštěn na disku v chráněném a omezeném prostředí, kde po určitou dobu je zkoumána jeho aktivita. Pokud je vyhodnocena jako škodlivá, testovaný soubor je přesunut do truhly. V opačném případě je aplikaci umožněno pokračovat ve standardním spuštění.

3.3.2 NG

Tato technologie je vylepšením Sandboxu. Vzorek je spuštěn ve virtuálním stroji, který simuluje reálné prostředí operačního systému. Pro potenciální malware je pak mnohem obtížnější poznat, že se nenachází v reálném prostředí, a může se naplno projevit jeho jinak schované chování.

3.4 Webový štít

Webový štít je komplexní komponenta Avastu pro skenování webového provozu, jež zahrnuje několik dalších komponent, které spolu komunikují a vytváří tak jeden velký celek. Pokud je hlavní funkcionalita webového štítu vypnuta, ostatní komponenty mohou bez něj stále fungovat, dokud nejsou explicitně také vypnuty. Tyto komponenty jsou popsány dále v této sekci.

3.4.1 Streamfilter

Ústřední komponentou webového štítu je streamfilter, který je rozdělený na kernel-mode (ovladač) a user-mode část (engine). Při instalaci se ovladač zaregistruje ve WFP, skrze který dokáže snímat síťový provoz na síťové kartě. Snímané packety spojuje do streamu a ten pošle engine. Pokud se jedná o vyhodnocení DNS záznamu, předá zodpovědnost SecureDNS, jinak pokračuje v testu na škodlivý kód. Pokud je stream infikovaný, řekne WFP, aby okamžitě ukončil spojení.

3.4.1.1 WFP

„Windows Filtering Platform (WFP) je sada API volání a systémových služeb, které poskytují možnosti pro vytváření aplikací pro filtrování sítě. WFP povoluje aplikacím zpracování packetů na různých úrovních systémových volání. Data posílaná po síti proto mohou být sledována, filtrována nebo modifikována ještě předtím, než dorazí k cíli.“ [9]

3.4.2 HTTPS skenování

HTTPS je zabezpečená verze protokolu HTTP, která je šifrovaná pomocí SSL nebo TLS, čímž je zabráněno odposlouchávání komunikace nebo podvržení dat. Šifrování funguje na principu asymetrického šifrování – odesílatel zprávy ji zašifruje veřejným klíčem příjemce, který následně dešifruje zprávu svým privátním klíčem. K ověření identity se využívají certifikáty, které jsou spravovány certifikačními autoritami. Bez ověření by tak docházelo k útokům zvaným „man in the middle“, kdy útočník je schopen zprávu dešifrovat a zároveň ji poslat šifrovanou dál tak, aby si komunikující strany myslely, že je vše v pořádku.

Komponenta HTTPS skenování při otevření spojení naváže nejdříve šifrovanou komunikaci s klientem a poté s cílovým serverem. Stane se tedy jakýmsi prostředníkem, který dokáže testovat dešifrovaná data na případný škodlivý kód a dále je poslat opět šifrovaně. Tato komponenta má tak vlastní důvěryhodný certifikát, který používá pro oboustranné spojení.

3.4.3 SecureDNS

Jedna z největších hrozeb, kterým musí uživatelé zranitelných routerů čelit, je DNS hijaking. Tato technika spočívá v přenastavení adresy používaného DNS serveru na útočníkovu vlastní. Tímto lze dosáhnout přesměrovávání na falešné stránky (například internetové bankovnictví) za účelem vylákání uživatelových přihlašovacích údajů. Komponenta SecureDNS zajišťuje bezpečné spojení s DNS servery Avastu.

Po nainstalování probíhají prvních pár minut takzvané volby. V těchto volbách se z přibližně stovky DNS serverů vyberou dva (primární a sekundární), které se budou pro daný počítač používat, tedy ty, které mají nejrychlejší odezvu a nejmenší vytíženost. V případě, že je primární server nedostupný, je použit sekundární. Když je SecureDNS aktivní, streamfilter do něj posílá veškerý DNS provoz, což znamená, že tímto způsobem je obejit DNS server nastavený na routeru, který může být infikovaný. Nicméně SecureDNS nezíská jenom DNS záznam ze serveru Avastu, ale také záznam podle původního nastavení, aby je mohl porovnat. Pokud je načtení záznamu rychlejší přes SecureDNS, použije se tato verze. V opačném případě se čeká (nejdéle) do vypršení časového limitu pro načtení záznamu přes zabezpečený DNS server. Když k načtení nedojde, použije se záznam z původního DNS serveru, jinak ten ze za zabezpečeného.

3.4.4 Štít pro testování skriptů

Tento štít je napojen přímo na engine webového prohlížeče, takže je schopný analyzovat chování skriptu přímo při jeho spuštění, a tím zabránit vykonání škodlivého kódu.

3.5 Firewall

Firewall, který je součástí Avastu, funguje stejně jako každý jiný softwarový firewall. Jeho úkolem je monitorovat příchozí i odchozí síťový provoz, který filtruje na základě přednastavených pravidel pro konkrétní IP adresy a porty.

Kapitola 4

Použité nástroje

Poslední kapitola teoretické části popisuje jednotlivé nástroje, které byly použity buď přímo a staly se tak součástí testů, nebo slouží jako podpora testů v infrastruktuře. Každý z těchto nástrojů je nejdříve ve stručnosti představen a následně je demonstrováno jeho použití. Podle potřeby práce nejsou některé funkční prvky nástrojů zmíněny, jiné jsou naopak vyzdviženy.

4.1 Git

Git je open source distribuovaný systém správy verzí navržený tak, aby zvládl projekty různých velikostí, a to vysokou efektivitou a rychlostí. [10]

4.1.1 Použití

Nový repositář lze založit příkazem `git init` a následně jej připojit ke vzdálenému serveru pomocí `git remote add <nazev> <url>`, nebo lze získat kopii ze serveru: `git clone <url>`. Následně po provedení změn v souborech, které se nacházejí v repositáři, je potřeba tyto změny zařadit mezi ty, které budou navržené pro přidání do lokální kopie (`git add <nazevSouboru>`), a následně je přidat pomocí `git commit -m <zprava>` a odeslat na server: `git push`. Získání aktuální verze se provede příkazem `git pull`. [11]

Při vytváření nových funkcionalit a jejich ladění je někdy výhodné si vytvořit v rámci repositáře vlastní izolovanou větev[10] (`git checkout -b <nazevVetve>`) a po dokončení vývoje ji následně sloučit s větví hlavní (`git merge <nazevVetve>`), která musí být v tuto chvíli aktivní. Větev může být nahrána na server (`git push origin <nazevVetve>`) nebo smazána (`git branch -d <nazevVetve>`). [11]

[10] branch

4.2 Wpkg

Wpkg je open source nástroj pro automatizaci nasazení, aktualizování a vymazání programů na operačních systémech Windows. Může být použit na hromadnou distribuci programů z jednoho serveru na více počítačů koncových uživatelů, kteří se nacházejí ve stejné síti. Wpkg podporuje tyto typy instalátorů: MSI, InstallShield, PackagefortheWeb, Inno Setup, Nullsoft. Dále také dokáže distribuovat různé .exe balíčky, .bat a .cmd skripty a jim podobné. [12]

4.3 Jenkins

Jenkins je open source nástroj pro kontinuální integraci a nasazení[11], díky němuž lze zlepšit proces vývoje skrze automatizaci. Jenkins dokáže spravovat a ovládat životní cyklus vývoje software ve všech jeho fázích včetně sestavení, testování, dokumentace, balíčkování, nasazení a statické analýzy. [13]

4.3.1 Kontinuální integrace

Kontinuální integrace je metoda softwarového inženýrství, při které všichni členové týmu pravidelně začleňují svoji práci do společného repositáře tak, aby mohlo být spuštěno automatické sestavení následované automatickými testy. Tímto postupem lze detekovat integrační chyby v co nejkratším čase od jejich vzniku. Kontinuální integrace zajišťuje, že nové funkcionality jsou dostupné ve spustitelné aplikaci co nejdříve od jejich zanesení to repositáře. [14]

4.3.2 Správa projektu pro testování

Při vytváření nového projektu je potřeba si uvědomit, jaký test bude tímto projektem pouštěn, a tedy jaká nastavení tento test potřebuje. Po zadání názvu a popisu projektu, které pomáhají k identifikaci, je možné zaškrtnout, že je tento projekt parametrizovaný, a začít přidávat jednotlivé parametry. Ty mohou být různého typu, například řetězec, výběr z více hodnot nebo booleanovská hodnota. Pro spuštění testu s různými konfiguracemi lze projekt nastavit jako maticový[12] a na jeho osy nanášet například verze operačních systémů nebo verze a edice testovaného programu.

Dalším krokem je nastavení zdrojového repositáře. Lze použít Git, SVN nebo kombinace v neomezeném počtu v případě, že test to tak vyžaduje. Nakonec nezbývá nic jiného než napsání bash skriptu, který Jenkins řekne, jakým způsobem testy spouštět, které soubory vykopírovat, a kde najít výsledky běhu testů.

[11] continuos integration and delivery
[12] matrix job

Configuration Matrix	free2free	is2is	free2is	free2isNoUpdate
Win_7_Ult_32b_EN				
Win_7_Ult_64b_EN				
Win_81_Ent_64b_EN				
Win_XP_Pro_32b_EN				
Win_10_Pro_32b_EN				
Win_10_Pro_64b_EN				
Win_10_Pro_10586_32b_EN				
Win_10_Pro_10586_64b_EN				

Obrázek 4.1: Ukázkový výsledek běhu testů

4.3.3 Vyhodnocení výsledku běhu testů

Po dokončení běhu testů lze na hlavní stránce projektu přehledně v matici vidět, jak testy dopadly. Příklad výsledku lze vidět na obrázku 4.1. Zelené značky označují konfigurace, ve kterých testy skončily úspěchem, oranžové značí, že testovaná aplikace neprošla některými požadavky testů a červené říkají, že běh skončil fatální chybou (například nesplnění prerekvizit testu nebo problém v infrastruktuře). Pokud je test přerušen (ručně uživatelem nebo automaticky po vypršení časového limitu), je takováto konfigurace označená šedou značkou.

Pokud některé konfigurace neskončily úspěchem, lze projít jejich výsledky a zobrazit konkrétní zachycené chyby nebo zkoumat záznam testu a veškeré soubory pro logování, které byly během testování vygenerovány, ať už testem nebo programem. Pokud selhala jenom jedna konfigurace nebo menší podmnožina konfigurací, lze po opravení chyb tyto testy pustit znova samostatně, aniž by musely být pouštěny úspěšné konfigurace.

4.4 ApacheBench

ApacheBench (zkráceně ab) je nástroj pro měření výkonu Apache HTTP serverů. Slouží k zjištění, jak velkou zátěž daný webový server zvládne, například kolik požadavků za sekundu je schopen obsloužit. [15] Mezi základní přepínače pro konfiguraci testu patří tyto:

- -c – Počet paralelní požadavků.

- -b – Velikost vyrovnávací paměti TCP[13] v bajtech.
- -n – Celkový počet požadavků zaslaných v průběhu jednoho testu.
- -k – Povolení funkcionality HTTP KeepAlive, která zajistí, že během jednoho TCP spojení je možné poslat více požadavků (probíhá konverzace).

4.5 Chrome HAR Capturer

Chrome HAR Capturer je nastroj pro zachycení HAR souborů ze samostatně běžící (vzdálené) instance internetového prohlížeče Google Chrome. [16]

4.5.1 HAR soubor

Zkratka HAR znamená HTTP Archive, což je dnes běžně používaný formát pro zachycení informací vzniklých používáním protokolu HTTP. Tento soubor obsahuje záznam o každém objektu, který byl prohlížečem nahráván. Informace jsou uloženy ve formátu JSON. Mezi zachycené informace patří např. tato: [17]

- Doba získání informací z DNS.
- Doba požadavku na každý objekt.
- Doba pro připojení k serveru.
- Doba přenosu každého objektu ze serveru do prohlížeče.

Pro potřeby této práce je nejdůležitější objekt `pageTimings` a jeho atribut `onLoad`, který udává čas v milisekundách, jenž uběhl od začátku načítání dané stránky až po její celé načtení. [18]

4.5.2 Použití

Před prvním puštěním je samozřejmě nutné Chrome HAR Capturer nainstalovat, konkrétně příkazem `sudo npm install -g chrome-har-capturer`, a to pomocí manažera balíčků npm, který je součástí instalace Node.js. [19]

Před každým měřením je potřeba spustit Google Chrome s určitými parametry. Prvním z nich je `-remote-debugging-port=<port>`, který povolí „Protokol vzdáleného ladění“[14] na zadaném portu. Další dva parametry `-enable-benchmarking` a `-enable-net-benchmarking` povolí javascriptové rozhraní, které umožní, aby Chrome HAR Capturer mohl vyčistit vyrovnávací paměť pro DNS záznamy[15] a DNS socket pool. [16]

[13] buffer, chunk
[14] Remote Debugging Protocol
[15] DNS flush

Po spuštění prohlížeče již stačí zahájit měření příkazem `chrome-har-capturer <url>`, kde za `<url>` lze dosadit jednu nebo i více adres najednou oddělených čárkou. Důležité ještě jsou tyto přepínače: [16]

- -o – Cesta k souboru, do kterého se uloží výstup programu.
- -d – Časový limit v milisekundách pro načtení dané stránky.
- -v – Podrobný výpis programu.

Chrome HAR Capturer poskytuje i API [16], pro případ, kdy by základní chování programu uživateli nevyhovovalo a chtěl by si vytvořit vlastní upravenou aplikaci. Detaily používání tohoto API nejsou ovšem pro potřeby této práce podstatné.

4.6 Xperf

Xperf je nástroj, který je součástí Windows Performance Toolkit, sloužící k zaznamenávání událostí ve Windows (Event Tracing for Windows). Za pomocí vestavěných profilů lze nastavit, které události budou zaznamenávány, nebo lze si vytvořit profily vlastní. Varianta s GUI se nazývá Windows Performance Recorder. Pro zobrazení výsledného záznamu slouží Windows Performance Analyzer. [20]

Pro zapnutí zaznamenávání základních událostí a stromu volání funkcí stačí spustit příkaz `xperf -on latency -stackwalk Profile`. Po skončení sledovaného úseku je potřeba xperf ukončit příkazem `xperf -d <cesta>.etl`, kde musí být specifikována cesta k výstupnímu souboru. [21]

4.7 OxyPlot

OxyPlot je multiplatformní open source knihovna pro .NET Framework určená k vytváření grafů. OxyPlot obsahuje v základu různé druhy os a řad, jejichž funkcionalitu může uživatel libovolně modifikovat. Vytvořené grafy lze exportovat do formátů png, pdf a svg. [22]

OxyPlot lze získat pomocí balíčkového manažera NuGet a následně jej přidat do projektu. Do uživatelského rozhraní stačí umístit prvek `PlotView` a následně jej propojit s datovým modelem reprezentovaným třídou `PlotModel`. Takovýto model lze plnit řadami (`LineSeries`), které obsahují konkrétní body (`DataPoint`) grafu. [22]

Část II

Praktická část

Kapitola 5

Analýza a konfigurace infrastruktury

První kapitola praktické části ve svých sekcích nejdříve představuje současnou podobu testovací infrastruktury ve firmě Avast Software, kterou jsem měl možnost poznat při své standardní práci a hlavně díky kolegům Jaromíru Cvrčkovi, Tomáši Daňkovi a Lukáši Kučerovi. Dále je popsáno, jak správně nakonfigurovat projekt v Jenkins pro performance testování, a nakonec je zařazena společná konfigurace počítačů pro tento účel vyhrazených.

5.1 Současná podoba infrastruktury

Při globálním pohledu na celou dostupnou testovací infrastrukturu se jedná o master/slave zapojení, což je model komunikačního protokolu, ve kterém jedno zařízení (master) ovládá více jiných zařízení (slaves). Komunikace (co se rozdělování úkolů týče) probíhá pouze jednosměrně od master ke slaves. [23] Jenkins (Testux) zastává roli master a servery (Slavux), jejichž konfigurace je automaticky udržovaná ve stejném stavu, plní roli slaves.

Když je v Jenkins spuštěn test, který vyžaduje určitý operační systém (v případě maticového projektu několik najednou), Jenkins zajistí rovnoměrnou distribuci virtuálních strojů na servery, protože zná jejich aktuální a maximální zatížení. Před samotným spuštěním virtuálního stroje na serveru je vytvořen snímek[16] základního obrazu operačního systému, což není nic jiného než pevný disk pro virtuální stroj. Tento snímek je na konci testu smazán, nicméně v blízké době je v plánu zálohování těchto snímků pro případnou potřebu manuální analýzy stavu operačního systému po skončení testů.

[16] snapshot

Správa obrazů[17] operačních systémů je zajišťována vlastním systémem, skrze který lze například instalovat aktualizace nebo ovladače. Při změně obrazu tento systém automaticky distribuuje aktualizovanou verzi na všechny servery.

5.1.1 Fyzux

Fyzux je souhrný název pro systém, který spouští testy na fyzických strojích (zapojených do tohoto systému), zabezpečuje distribuci obrazů operačních systémů a spravuje testovací prostředí. Při každém startu takového počítače proběhne v miniaturním operačním systému, který se nachází na síti, kontrola, zdali se má nahrát nový obraz. Pokud předtím uživatel vybral obraz operačního systému (více v příští podsekci), je nahrán ze sítě na disk počítače. V opačném případě nastane klasické spuštění systému z disku.

Výhodou tohoto systému je, že při testování ve virtuálních strojích se nemusí projevit některé chyby, které lze zreprodukovat pouze na strojích fyzických. Dále také například technologii NG, která byla zmíněna v podsekci 3.3.2, lze nainstalovat a používat pouze na fyzických počítačích, takže její testování musí probíhat za pomocí systému Fyzux. V neposlední řadě se ve virtuálním prostředí nedají provádět performance testy, protože čas plynoucí uvnitř těchto strojů je relativní a tím pádem by naměřené výsledky neměly naprosto žádnou vypovídací hodnotu. Bez existence systému Fyzux by nemohla vzniknout tato práce a aktivity s ní spojené by nemohly být uskutečněny.

5.1.2 GLPI

GLPI je systém pro správu software a zařízení, jako jsou například počítače, monitory nebo tiskárny. [24] V infrastruktuře Avastu je tento systém rozšířen o webové rozhraní pro Fyzux. Lze tedy z něj počítač na dálku zapnout, vypnout nebo restartovat, a to korektním způsobem nebo „natvrdo“. Dále je rozšířen o správu obrazů systémů na daném počítači, je tak možné nahrát vybraný obraz na konkrétní počítat, uložit současný stav systému do nového obrazu nebo uložený obraz smazat. Funkcionalitu nahrávání obrazů pak využívá Jenkins při puštění automatického testu, který má běžet na fyzickém počítači.

5.1.3 Test runner

Test runner je framework zajišťující testovací workflow. Postup, který test runner dodržuje, je následující:

1. Vytvoření snímku operačního systému a spuštění virtuálního nebo fyzického stroje.

[17] image

2. Příprava prostředí pomocí wpkg balíčků. Pokud se tento krok nepodaří, opakuje se od prvního kroku (maximálně celkem třikrát).
3. Testovací fáze – pouštění Python skriptů nebo libovolných spustitelných souborů. V této fázi se při nastavení dá kombinovat pořadí spuštění jednotlivých skriptů, mezi něž se dají vkládat příkazy pro čekání nebo restartování počítače.
4. Sbírání logovacích souborů a informací o průběhu testu.
5. Úklid a ukončení, smazání snímku operačního systému.

Aby test runner věděl, co má dělat, musí uživatel vyplnit následující parametry:

- fyzux_pc_id – unikátní identifikátor počítače zapojeného do systému Fyzux.
- fyzux_user_token – token uživatele používaný pro autorizaci.
- fyzux_image – cesta k obrazu operačního systému.
- packages – seznam balíčků pro nainstalování před spuštěním testů.
- tests – seznam testů, které se mají spustit (na pořadí záleží).

Mezi volitelné parametry patří například seznam cest ke složkám/souborům, které se mají po skončení testů vykopírovat, nebo příznak, zdali se mají vykopírovat logovací soubory Avastu a případné záznamy o pádech[18].

5.1.4 Cotel2

Cotel2 je rozšíření standardních Python unit testů, aby se daly při spuštění parametrizovat. Dále je vylepšené logování, aby se výpis odsazoval podle hloubky zanoření v kódu, je přidána podpora výstupu v HTML formátu a čas jednotlivých záznamů. Nedílnou součástí je i vytvoření XML souboru s výsledky, který je ve formátu kompatibilním s rozhraním pro zobrazování výsledků v Jenkins.

5.1.5 Cotel2.utils

Cotel2.utils je pomocná knihovna obsahující funkce, které jsou často používány v automatických testech. Skládá se z těchto čtyř částí:

- Získávání informací o klientu Avastu (verze, instalační složka, hodnoty v registrech) a spuštění aktualizací programu a virové databáze.

[18] crash dump

- „Obaly“ Windows API volání pro jednodušší používání, například získání oken procesů a informací o uživatelích, práce s Event Log, ovládání služeb, síťové operace, práce s .ini soubory.
- Používání mailových protokolů POP3 a IMAP.
- Konverze různých vstupních formátů na Python struktury.

5.1.6 AswDataPy

AswDataPy je knihovna, která umožňuje v Python skriptu volání C++ funkcí, práci s třídami a některými vestavěnými datovými strukturami. Konkrétně poskytuje veškerou funkcionalitu, pomocí níž lze ovládat GUI Avastu.

5.1.7 Proxy cache

„Proxy cache slouží jako prostředník mezi uživatelem a poskytovatelem webového obsahu. Proxy cache funguje jako brána, která si uchovává obsahy serverů. Když uživatel se pokusí přistoupit k obsahu, proxy cache zkontroluje, jestli ho má po ruce. Pokud ano, okamžitě ho doručí uživateli. V opačném případě ho nejdříve získá ze zdroje a poté zároveň si ho uloží a doručí uživateli.“ [25]

V této práci využívám proxy cache, o jejíž zprovoznění jsem si zažádal, abych odstínil výsledky od zpoždění mezi uživatelem a server, a tím získal pouze časy pro získávání samotného obsahu.

5.2 Konfigurace projektu v Jenkins

V Jenkins jsem vytvořil dva maticové projekty, které jsou asociovány s dvěma komplexními celky: testy webového štítu a testy štítu souborového systému. Oba projekty mají většinu nastavení společnou. Liší se pouze v kombinacích zapnutých komponent (více v podsekci 5.2.3) a specifických pouštěních těchto testů.

5.2.1 Edice Avastu

Antivirus Avast je v současné době k dispozici ve čtyrech edicích (Free, Pro, Internet Security, Premier), které se liší počtem komponent, které obsahují. Pro performance testy jsem vybral edici Internet Security, protože narozdíl od Free a Pro obsahuje firewall a zároveň neobsahuje komponenty z Premier, které nemají vliv na performance a jenom by zbytečně prodlužovaly dobu instalace před každým testem. Nicméně jsem pro edici vytvořil parametr, kdyby se edice, jejich obsah a počet komponent v budoucnu změnil.

5.2.2 Verze Avastu

Tento parametr je důležitý pro otestování starších verzí a sledování regrese v průběhu času. Na základě výsledků a seznamu změn v jednotlivých verzích pak půjde určit, které konkrétní změny mají dopad na performance. Řetězec vyplněný v tomto parametru je pak použit pro instalování správného balíčku Avastu, proto je možné místo konkrétní verze (například 10.2.2218_A10R2SP2) zadat zástupný řetězec (například trunk, branch), který typicky směřuje na poslední verzi z dané vývojové větve.

5.2.3 Konfigurace Avastu

První osou matice jsou kombinace zapnutých komponent Avastu. Každé kombinaci je přiděleno číslo, které je předáno testu při spuštění jako parametr. Pro webový štít jsou to tyto komponenty:

- Webový štít
- FileRep
- HTTPS skenování
- SecureDNS
- Firewall

Celkový počet kombinací (včetně konfigurace bez Avastu) je 20, což odpovídá tomu, že nemá smysl mít zapnutý FileRep, zatímco webový štít jako takový je vypnutý, protože ho žádná jiná komponenta nevyužívá a vznikala by tak duplikátní data. Stejně tak ve všech konfiguracích, kde je zapnuté HTTPS skenování jsou přeskočeny testy na ApacheBench, který pracuje na HTTP.

Komponenty, které jsou konfigurovány v testech štítu souborového systému jsou pouze tyto tři:

- Štít souborového systému
- FileRep
- Sandbox

I zde ze stejných důvodů nemá smysl zapínat FileRep ve chvíli, kdy je štít souborového systému vypnutý.

5.2.4 Operační systém

Na druhou osu matice jsou nanášeny verze operačního systému Windows. V tuto chvíli jsou testy pouštěny pouze na Windows 7 64-bit, a to ze dvou důvodů. V první řadě se jedná o systém, který je podle interních statistik nejrozšířenějším mezi uživateli Avastu. Dále se osvědčil jako nejspolehlivější mezi ostatními systémy, na kterých se pouštějí standardní automatické testy pro kontrolu funkčnosti antiviru. V neposlední řadě není do systému Fyzux zapojeno dostatečné množství stejných počítačů (pro paralelizaci testů na klíčových konfiguracích) a také množství různých počítačů, které by sloužily pro simulaci rozmanitosti hardware napříč uživatelskou základnou. Díky faktu, že je testováno pouze na jednom systému, má tak matice v tuto chvíli pouze jeden řádek. V nejbližších měsících je v plánu pořízení i několika druhů notebooků a tabletů s Windows 10.

5.2.5 Spouštění testů

V tomto kroku je potřeba správně nastavit test runner, jak je popsáno v podsekci 5.1.3. Parametry ID počítače, token a operační systém jsou v tuto chvíli napevno dané. Z balíčků se vždy instalují následující: cotel2, Google Chrome, knihovna do Python pro vytváření snímků obrazovky (pro případné ladění) a pokud si to vyžaduje konfigurace, tak podle parametru odpovídající verze Avastu a knihovny aswDataPy.

Po nainstalování balíčků přichází fáze spouštění testů. Zatímco test webového štítu nepotřebuje žádnou speciální nastavení kromě čísla konfigurace komponent Avastu, test štítu souborové systému vyžaduje parametr, kterým je řečeno, jestli se jedná o první nebo druhý start počítače nebo další testy, které nejsou zaměřeny na délku startu počítače (více o konkrétních testech v sekci 6.2). Proto je test spuštěn celkem třikrát, pokaždé s jiným parametrem.

5.2.5.1 Nalezený problém

S instalací Avastu souvisí problém, který jsem objevil, když jsem pouštěl testy s čím dál tím starší verzí. Každý instalátor má totiž v sobě zabudovanou kontrolu na své stáří. Pokud tato doba přesáhne rok, nabídne instalaci nejnovější verze. Ovšem tichý instalátor, který je používán pro automatizaci ve wpkg balíčku, tuto aktualizaci provede automaticky bez možnosti nastavení jiného chování pomocí konfiguračního souboru, což zapříčiní, že testy jsou puštěny na jiné verzi, než bylo předpokládáno. Jediným dočasným řešením tak zůstává před instalací posunout čas dozadu a po ní opět zpátky. Na základě mého upozornění na tuto skutečnost bude od další vydané verze výchozí chování instalovat danou verzi podle instalátoru s možností nastavit původní chování pomocí konfiguračního souboru.

5.3 Konfigurace počítače

Po nainstalování jakékoliv verze operačního systému Windows jsou v základním nastavení zapnuty některé funkce, které by mohly ovlivňovat výsledky performance testů, nebo chybí nástroje pro případné ladění. Proto jsem vytvořil následující seznam úkonů, jež je potřeba provést před uložením finálního obrazu systému, který bude pro testy používán:

- Aktivovat systém platným produktovým klíčem.
- Vytvořit lokální účet s administrátorskými právy.
- Povolit automatické přihlášení do účtu.
- Přepnout nastavení UAC na „Nikdy neupozorňovat“.
- Nainstalovat všechny aktualizace skrze službu Windows Update.
- Vypnout automatické aktualizace systému.
- Vypnout automatické aktualizace internetového prohlížeče Internet Explorer.
- Zakázat funkcionality pro šetření energie.
- Zakázat vytváření bodu obnovení.
- Zakázat defragmentaci pevného disku.
- Vypnout Windows Defender.
- Vypnout Windows Firewall.
- Povolit kompletní výpis paměti (pro případ pádu systému).
- Nainstalovat nástroje z ADK a SysInternals.

Počítač vybraný pro tuto práci se skládá z následujícího hardware:

- Procesor: Intel Core i3-2120 @ 3.30 GHz
- Operační paměť: 8 GB
- Disk: Samsung SSD 850 EVO 250 GB
- Síťová karta: Intel Corporation 82579LM Gigabit Network Connection

Kapitola 6

Návrh a implementace

Tato kapitola představuje jednotlivé uživatelské scénáře, jejich implementaci a vypořádání se s případnými problémy. Součástí je také návrh databázového modelu a implementace jednoduchého nástroje pro zobrazování výsledků. V závěru kapitoly jsou ve zkratce popsány některé z plánů do budoucna.

6.1 Úvod

Po dohodě s vedoucím práce a na základě diskuse s kolegy z vývojového týmu jsem vybral dvě základní a zároveň nejdůležitější komponenty (včetně jejich podkomponent), které byly představeny v kapitole 3 a jež je potřeba podrobit performance testům: štít souborového systému a webový štít.

6.2 Performance testy

Uživatelské scénáře, které jsem implementoval a popsal v podsekcích 6.2.2 a 6.2.3, vycházejí z dokumentu [26] vytvořeného společnosti PassMark, která se mimo jiné zabývá měřením performance antivirů.

6.2.1 Společná funkcionalita

Funkce, které využívám napříč různými testy a mohly by být využívány i v testech budoucích, jsem oddělil do samostatného souboru, aby byly všem přístupné a nemusely se opakovat. Tyto funkce by se v budoucnu mohly stát součástí knihovny cotel2.utils, pokud by o ně projevili zájem i další vývojáři automatických testů.

6.2.1.1 Uvolnění paměťových stránek

Operační systém Windows rozděluje fyzickou paměť na stránky. Každá z těchto stránek se nachází v některém z následujících stavů, jak je popsáno

v kapitole 10 knihy *Windows Internals*: [27]

- Active – Stránka je okamžitě k dispozici, protože proces k ní aktivně přistupuje, a není ji v tomto stavu možné nahradit jinou.
- Standby – Ke stránce již delší dobu nebylo přistupováno a nebyla modifikována od posledního zápisu na disk, proto může být v případě nedostatku volné paměti nahrazena stránkou jinou, která se v paměti nenachází. Nicméně dokud je stránka ve stavu standby, je stále okamžitě přístupná.
- Modified – Stránka přestala být aktivní, ale během doby, kdy byla používána, byla také modifikována a její obsah nebyl ještě zapsán na disk. Než dojde k uvolnění této stránky, musí být uložena na disk.
- Free – Stránka je uvolněná, nicméně obsahuje nespecifikovaná data.

Při opakovaném spuštění stejného testu (v případě neúspěchu) mohou být v paměti stále nahrané pozůstatky aplikací a souborů z předchozího spuštění. To může zapříčinit, že by tyto pozůstatky nebyly nahrány do paměti, ale rovnou použity, což by vedlo k ovlivnění výsledku. Proto jsem napsal malou aplikaci inspirovanou [28], která uvolní veškeré stránky nahrané do paměti a tím tak zajistí čistý běh opakovaného testu.

6.2.1.2 Ovládání štítů

Skrze aswDataPy lze jednotlivým štítům posílat příkazy (Start, Stop, ...). Pro snazší používání jsem vytvořil funkci `manipulateProvider`, která na základě názvu štítu a jména příkazu vytvoří parametry kompatibilní s rozhraním aswDataPy a následně čeká do té doby, než se štít přepne do požadovaného stavu nebo vyprší časový limit (neúspěch).

6.2.1.3 Ovládání firewall

Funkce `manipulateFirewall` funguje na stejném principu jako předchozí zmíněná, nicméně její implementace se lehce liší, protože rozhraní poskytuje pro štíty a firewall dva rozdílné objekty, se kterými se zachází odlišně. Účel ovšem zůstává stejný – zapnout nebo vypnout danou komponentu, v tomto případě firewall.

6.2.1.4 Změna nastavení

Každému zaškrtávacímu políčku[19], které se nachází v GUI v nastavení Avastu, lze pomocí aswDataPy změnit jeho hodnotu (zaškrtnuté/nezaškrtnuté). Funkce `changePropetry` na základě identifikátoru políčka a cílové hodnoty dá

[19] checkbox

příkaz ke změně a pak vyčká, až se nastavení projeví (nejdéle však do vypršení časového limitu).

6.2.1.5 Spuštění příkazu

V testech často potřebuji pustit příkaz v příkazové řádce Windows takovým způsobem, abych zaznamenal standardní i chybový výstup a zároveň aby bylo ošetřeno vypršení případného limitu pro vykonání. Protože bych zmíněnou funkcionalitu musel pořád opakovat, vytvořil jsem funkci `callCmd`, která ji vykonává a dá se parametrizovat podle konkrétních potřeb.

6.2.1.6 Uložení snímku obrazovky

Díky wpkg balíčku `screenshot-cmd` lze z příkazové řádky pořizovat snímky obrazovky. Funkce `saveScreenshot`, která přijímá jako volitelné argumenty název cílového obrázku a handle konkrétního okna (chceme snímek konkrétního programu), obaluje funkcionalitu příkazu `screenshot-cmd` pro jeho snazší používání.

6.2.1.7 Vytvoření objektu výsledku

Po skončení každého testu se vždy vytváří JSON objekt reprezentující výsledek testu, který bude odeslán do databáze. Aby byl tvůrce testu odstíněn od struktury tohoto objektu a zároveň neduplikoval kód, vytvořil jsem funkci `getTestResultJSON`, která po předání správných parametrů objekt vytvoří za něj.

6.2.1.8 Vytvoření objektu pro spuštění testů

Těsně před před odesláním výsledků do databáze je funkcí `getTestRunJSON` vytvořen JSON objekt TestRun (více o entitách v sekci 6.3). Stejně jako v předchozím případě se jedná o odstínění uživatele od implementace. Při zavolání tak stačí předat parametrem identifikátor konfigurace, celkový uplynutý čas všech testů, výsledky těchto testů, verze Avastu a případný komentář.

6.2.1.9 Odeslání výsledků

Na konci všech testů jsou výsledky odeslány do databáze za pomocí funkce `sendResults`, která přijímá parametrem číslo konfigurace, začátek spuštění prvního testu, verzi Avastu a komentář. Poté, co je vytvořen výsledný JSON, jsou data odeslána.

6.2.1.10 Zapnutí a vypnutí xperf

V případě potřeby (například velký časový rozdíl ve výsledcích jednotlivých konfigurací) lze z projektu v Jenkins nastavit parametr, kterým se v testech

povolí zapnutí xperf pro hlubší analýzu. Funkce `startXperf` se volá ve chvíli, kdy začne měření času, a `stopXperf` zase při ukončení měření času. Výstupní soubor je pro dohledatelnost pojmenovaný podle konkrétního testu a je uložen na síťový disk, protože při dalším spuštění testů se disk přehraje čistým obrazem.

6.2.2 Testy webového štítu

Před spuštěním všech testů webového štítu jsou nastaveny komponenty podle vybrané konfigurace. Pokud je zapnuté SecureDNS, čeká se do té doby, než proběhnou volby serverů, jak je zmíněno v podsekci 3.4.3. Protože webové stránky mohou sdílet externí zdroje (reklamy, Google Analytics), rozhodl jsem se před každým testem konkrétní stránky nebo souboru mazat tyto vyrovnávací paměti, aby při každém běhu testu byly stejné podmínky:

- ARP cache – ARP se používá pro spojení síťové a datové vrstvy OSI modelu, což z pohledu uživatele znamená, že za pomocí IP adresy je zjištěna MAC adresa cílového zařízení, s nímž má proběhnout komunikace. Toto spojení adres je uloženo pro příští použití v cache zařízení do vypršení časového limitu spjatého s daným ARP záznamem. [29]

- DNS cache – Tato cache uchovává záznamy, které překládají doménová jména serverů na IP adresy, a tím zajišťuje efektivní synchronizaci s DNS servery v případě změny IP adresy nebo vzniku nových serverů. [30]

- Certificate cache – CRL a OSCP jsou dvě metody (starší a novější) pro přístup na server za pomocí PKI. [31] Operační systém Windows si výsledky těchto metod uchovává až do té doby, než vyprší jejich platnost. [32]

6.2.2.1 ApacheBench

Celý test spočívá ve spuštění nástroje ApacheBench oproti internímu webovému serveru, na kterém jsou nahrané různé soubory a webové stránky, a následném „vytažení“ výsledku z výstupu tohoto nástroje. Na tom není nic až tak zajímavého. Klíčovým prvkem je totiž nastavení jednotlivých parametrů. Pro základní zmapování chování jsem použil tyto tři konfigurace:

- ab -c 4 -b 8192

- ab -c 4 -b 32000

- ab -c 4 -b 32000 -k

Jak lze vidět, počet paralelních spojení jsem zanechal konstantní. Je to z toho důvodu, že nejdříve bych chtěl zjistit rozdíly (pokud vůbec nějaké) při manipulaci s vyrovnávací pamětí a teprve poté pracovat s kombinacemi s variabilním počtem spojení. Další kombinace by také výrazně prodloužily dobu provádění testu. Po naměření a zjištění základního chování mám v plánu přidat do projektu možnost si tento test pustit samostatně a jako parametr předat konfiguraci pro ApacheBench, což by mohli využívat vývojáři při ladění konkrétního problému.

Poslední parametr, který jsem nezmínil, je celkový počet stažení jednoho souboru. Ten se odvíjí od velikosti tohoto souboru a nastavil jsem ho pro každý soubor tak, aby celkový objem stažených dat na jeden soubor se pohyboval okolo 800 MB. Seznam souborů stahovaných ze serveru je následující:

- HTML stránka o velikosti 10 kB
- HTML stránka o velikosti 100 kB
- HTML stránka o velikosti 500 kB
- HTML stránka o velikosti 1 MB
- HTML stránka o velikosti 15 MB
- Firefox_Setup_26.0.exe
- AdbeRdr11006_en_US.exe
- gimp-2.8.10-setup.exe
- LibreOffice_4.2.0_Win_x86.msi
- filezilla_3.10.0.1_win32-setup.exe
- 7z920.exe
- npp.6.7.4.Installer.exe
- install_flash_player_18_active_x.exe
- vlc-2.1.5-win32.exe
- Thunderbird_Setup_31.4.0.exe
- jre-8u40-windows-i586.exe
- SkypeSetupFull.exe

Tento test může při určitém nastavení simulovat kromě vícenásobného paralelního spojení (více otevřených panelů v prohlížeči, stahování několika souborů najednou) také zátěžový test. Zmíněný fakt se projevil při testování jedné z verzí Avastu, kdy zátěž programu ApacheBench způsobila uváznutí[20], kvůli kterému nedocházelo k překladu jmen serverů na IP adresy, tudíž byl počítač z hlediska přístupu na internet nepoužitelný.

6.2.2.2 Internet Explorer

Testy pro Internet Explorer využívají programu, který vytvořil kolega Vladimír Kašpar, a získávají výsledky z jeho výstupu. Tento program pracuje s COM rozhraním, díky němuž může komunikovat s prohlížečem Internet Explorer. Tento program bych chtěl v budoucnu z testů odstranit a nahradit ho Python skriptem, který spustí xperf nastavený speciálně pro Internet Explorer podle [33] a z výstupního souboru získá jak dobu načtení stránky, tak dobu zapnutí prohlížeče. Z tohoto výstupu lze následně získat informace o tom, které knihovny se načítaly jak dlouho, a tedy zpozdily start prohlížeče.

Stránky, které byly použity pro tento typ testu, jsou následující:

- http://www.amazon.com/
- http://www.aol.com/
- http://www.apple.com/
- http://www.ebay.com/
- http://www.facebook.com/
- http://www.flickr.com/
- http://www.google.de/
- http://www.guardian.co.uk/
- http://www.heise.de/
- http://www.imdb.com/
- http://www.linkedin.com/
- http://www.live.com/
- http://www.microsoft.com/
- http://www.nytimes.com/

[20] deadlock

- http://www.paypal.com/
- http://www.spiegel.de/
- http://www.twitter.com/
- http://www.wikipedia.org/
- http://www.yahoo.com/
- http://www.youtube.com/

6.2.2.3 Google Chrome

Prohlížeč Google Chrome umožňuje uložit informace o síťovém provozu při načítání stránky jako HAR soubor. [34] Tato funkcionalita je bohužel dostupná pouze z GUI prohlížeče, proto jsem použil nástroj Chrome HAR Capturer [16], z jehož výstupu mohu získat potřebné informace. To se na první pohled může zdát jako jednoduchý úkol, bohužel tomu tak není.

První problém se váže k nedeterministickému času spuštění prohlížeče. Neexistuje totiž způsob, jak by Google Chrome o sobě řekl, že je již připravený. Zkoušel jsem čekat do té doby, než se objeví handle hlavního okna. Ten ale bohužel vzniká mnohem dříve, než je celý prohlížeč připravený. Proto nezbývá nic jiného, než určit časový limit, po němž si mohu být jistý, že prohlížeč je již plně funkční. Tento limit ovšem nesmí být na straně druhé moc dlouhý, protože při velkém počtu opakování testu se zdánlivě málo vteřin nasčítá do velkého časového rozsahu. Uchýlil jsem se tedy k empirické metodě pozorování monitoru se stopkami v ruce a na jejím základě jsem určil limit 10 vteřin. Do budoucna není tato metoda ideální, protože každá hodnota by musela být „šitá na míru“ každému stroji. Bude proto potřeba použít pravděpodobně nástroje založené na rozpoznávání změn na obrazovce, jako je například Sikuli [35].

Druhý problém souvisí se zavíráním prohlížeče. Tato aktivita byla v testech původně zajišťována vestavěným Windows příkazem `taskkill`. Ve výstupu testu jsem ale často nacházel takové chování, že od určité chvíle se negenerovaly výsledky. Protože se tento jev objevoval náhodně, nemohl jsem z časových důvodů čekat u monitoru, až daná situace nastane. Proto jsem nechal pokaždé při výskytu tohoto problému vytvořit snímek obrazovky. Ten odhalil, že okna prohlížeče zůstávají otevřená, a proto Chrome HAR Capturer nevěděl, na který prohlížeč se má připojit. Příkaz `taskkill` jsem tak nahradil enumerací všech oken procesu chrome.exe, kterým zasílám zprávu WM_CLOSE. To sice stále nestačilo, ale nyní jsem měl k dispozici handle všech oken, takže jsem si mohl vytvořit snímek každého z nich. Ukázalo se, že na vině je dialogové okno prohlížeče oznamující, že Google Chrome přestal odpovídat, a zároveň se dotazující, jestli ho chci restartovat. Tento dialog nešel zavřít, protože

„zavírací křížek“ byl ve stavu disabled, a tak jedinými ovládacími prvky jsou tlačítka „Ano“ a „Ne“. Protože prohlížeč nechci restartovat, ale zavřít, pošle skript každému oknu, které se nezavřelo po přijetí WM_CLOSE, zprávu WM_COMMAND s parametrem IDNO, čímž se zajistí „stisknutí“ tlačítka „Ne“.

Poslední dosud objevený problém je na straně štítu pro testování skriptů. Stává se totiž, že určitá podmnožina testovaných stránek vždy v konfiguracích s Avastem nemá žádné výsledky. Při zkoumání výstupu Chrome HAR Capturer jsem zjistil, že dochází k jeho pádu. Snažil jsem se proto postupným vypínáním štítů a funkcionalit „usvědčit“ konkrétního viníka. Bohužel i při plně vypnutém Avastu k chybě docházelo. Pomocí Process Explorer [36] jsem narazil na to, že do procesu prohlížeče jsou stále nahrávány knihovny zmíněného štítu. Po smazání knihoven vše začalo fungovat správně. Problém je reprodukovatelný bohužel jen při automatickém testu, takže v době měření pro potřeby této práce musel být štít v průběhu testů vypnutý a chyba bude analyzována a opravena až po dokončení měření. Po provedení opravy musí následná měření zahrnout zapnutí štítu pro testování skriptů, protože je hlavním podezřelým pro performance problémy s prohlížeči.

6.2.3 Testy štítu souborového systému

Stejně jako u testů webového štítu jsou na začátku nejdříve nastaveny komponenty. Dále jsou ze síťového disku staženy veškeré soubory potřebné pro nadcházející testy. Tyto soubory původně prošly každým testem pouze jedenkrát, jenže časy byly tak krátké, že neměly vypovídací hodnotu a nedaly se porovnávat. Proto soubory prochází každým testem stokrát. To ovšem vyžaduje některá opatření.

Aby se nemohla USN cache, která je nahraná v paměti, uložit na disk, potlačím toto chování funkcí `supressUSNCacheSave` a následně smažu již uloženou databázi (`deleteSavedUSNCache`), aby se nemohla načíst do paměti. Před každým opakováním pak smažu nově vytvořenou USN cache v paměti (`cleanUSNCache`). Původně jsem chtěl mazat i dočasnou interní cache Avastu (`clearPathHashCache`), ale po diskusi s odpovědným vývojářem jsem zjistil, že stačí soubor pokaždé přejmenovat a engine ho bude vždy testovat.

Operační systém Windows při zápisu na disk nejdříve data uloží do vyrovnávací paměti a teprve až se naplní, začne fyzicky zapisovat. Toto chování může způsobit, že funkce jako `FileWrite` se mohou vrátit dříve, než jsou data ve skutečnosti zapsaná na disk. Vyprázdnění této paměti lze zajistit API voláním `FlushFileBuffers`. [37] Abych tedy měřil správné časy, na konci každé série zápisu na disk zavolám funkci `flushVolumeBuffers`, která obaluje zmíněnou

funkcionalitu, a čas vyprazdňování vyrovnávací paměti započítám do celkového času.

Pro všechny testy kromě posledních dvou je použito pět náhodně vybraných čistých binárních souborů velikosti od 5 MB do 87 MB, které se nachází v interní databázi vzorků.

6.2.3.1 Načtení knihoven

Prvním testem je načítání knihoven (i obyčejných binárních souborů) do paměti pomocí API funkce `LoadLibraryEx` s příznakem DONT_RESOLVE_DLL_REFERENCES, abych měl jistotu, že je vždy do paměti načtený jen obsah dané knihovny a nikoliv i další závislosti.

6.2.3.2 Kopírování souborů na lokální disk

Připravené soubory v adresáři na lokálním disku jsou kopírovány do jiného adresáře. Jak bylo výše zmíněno, každý soubor je v cíli přejmenován, aby vždy došlo ke čtení zdrojového souboru a testování Avastem.

6.2.3.3 Kopírování souborů na síťový disk

Tento test využívá společnou funkcionalitu (`copyFiles`) s předchozím jenom s tím rozdílem, že soubory jsou kopírovány do adresáře na síťovém disku.

6.2.3.4 Zabalení souborů do archívu

Soubory jsou jeden po druhém zabaleny do vlastního archívu bez jakékoliv komprese. Výsledný archív má opět pokaždé jiný název ze stejných důvodů. Při zápisu souboru do archívu je potřeba dbát na to, že dokumentace [38] sice tvrdí, že název souboru v archívu, pokud není parametrem řečeno jinak, odpovídá cestě k původnímu souboru bez písmene disku, nicméně není z této definice patrné, že archív zachová celou adresářovou strukturu vedoucí až k souboru (což na druhou stranu dává smysl). Pro archivaci souboru bez adresářů je tedy potřeba specifikovat parametr „arcname“.

6.2.3.5 Rozbalení souborů z archívu

Archívy vytvořené předešlým testem jsou zde postupně rozbalovány.

6.2.3.6 Počítání hash souborů

Pro každý soubor je pomocí knihovny `hashlib` spočítána MD5 hash.

6.2.3.7 Kopírování souborů pomocí vlastní implementace

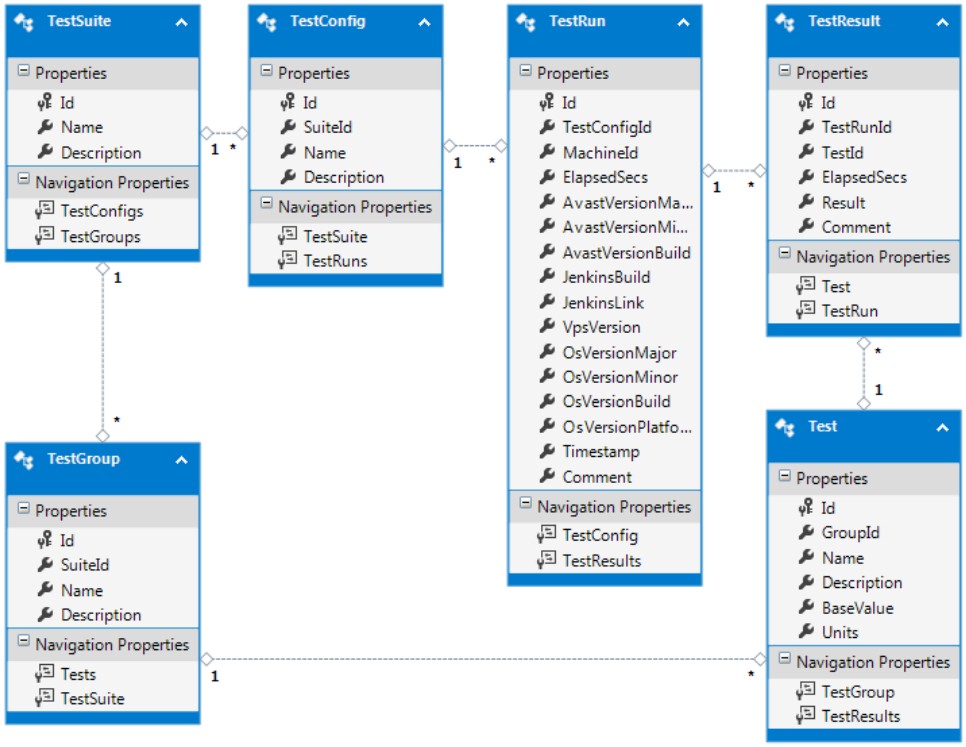
Avast se chová jinak k souborům, které jsou kopírovány funkcí `CopyFile` než k těm, na něž byly použity funkce `ReadFile` a `WriteFile`, protože u druhé metody nelze zjistit, zda byl soubor mezi těmito dvěma API voláními změněn či nikoliv. Proto jsem přidal do sady i tento test, který využívá vyrovnávací paměť stejné velikosti jako `CopyFile`, tedy 64 kB.

6.2.3.8 Instalace a spuštění programů

V tomto testu jsou nainstalovány všechny programy, jejichž instalátory byly použity pro ApacheBench testy popsané v 6.2.2.1. Každý z těchto instalátorů má jiné přepínače pro automatickou a tichou instalaci a jsou následující:

- Firefox_Setup_26.0.exe -ms
- AdbeRdr11006_en_US.exe /sAll /msi /norestart ALLUSERS=1 EULA_ ACCEPT=YES
- gimp-2.8.10-setup.exe /verysilent /norestart /dir=%ProgramFiles% GIMP 2
- msiexec /quiet /i LibreOffice_4.2.0_Win_x86.msi
- filezilla_3.10.0.1_win32-setup.exe /S
- 7z920.exe /S
- npp.6.7.4.Installer.exe /S
- install_flash_player_18_active_x.exe -install
- vlc-2.1.5-win32.exe /S
- Thunderbird_Setup_31.4.0.exe -ms
- jre-8u40-windows-i586.exe /s
- SkypeSetupFull.exe /VERYSILENT /SP- /NOCANCEL /NORESTART /SUPPRESSMSGBOXES /NOLAUNCH

Motivace pro spuštění právě nainstalovaných programů je taková, že uživatel instaluje program, aby ho mohl okamžitě začít používat. Protože není možné stejně jako u Google Chrome určit, kdy je aplikace spuštěna a plně funkční (6.2.2.3), simuluji toto spuštění postupným mapováním všech binárních souborů do paměti, které se nachází ve složce, kam byl program nainstalován.



Obrázek 6.1: Databázový model

6.2.3.9 Start počítače

Pomocí Prohlížeče událostí[21] se dá zjistit mimo jiné čas startu počítače. „Prohlížeč událostí je nástroj zobrazující podrobné informace o významných událostech v počítači (například o tom, že programy nejsou spuštěny očekávaným způsobem nebo že jsou automaticky staženy aktualizace.)“ [39] Zmíněný záznam o startu počítače lze najít v cestě `Microsoft-Windows-Diagnostics-Performance/Operational` pod ID 100 a jeho čas pod položkou „BootTime“. Konečného získání času lze docílit sekvencí API volání `EvtQuery`, `EvtNext` a `EvtRender`.

6.3 Databázový model

Výsledky měření musí být nahrány do konzistentního a strukturovaného úložiště, aby byly jednoduchým způsobem dohledatelné, filtrovatelné a zobrazitelné. Proto nemá smysl výsledky uchovávat ve výstupních souborech, ale je záhodno je posílat do databáze. Databázový model, který lze vidět na obrázku 6.1, se skládá z následujících entit:

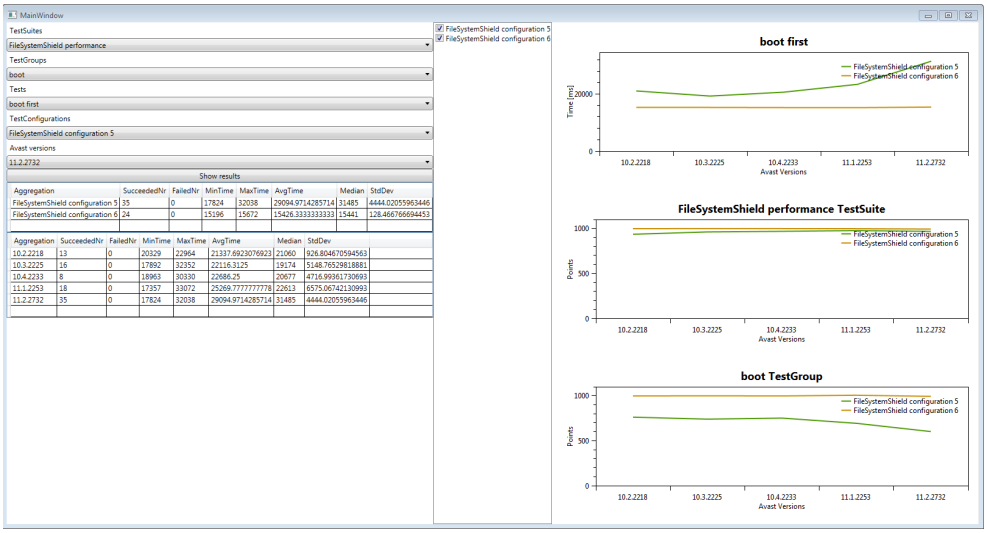
[21] Event Viewer

- Test – Stavební kámen celkové struktury, který popisuje jeden konkrétní typ testu. Kromě jména a bližšího popisu testu v sobě obsahuje základní hodnotu, jež byla naměřená s čistým prostředím bez Avastu a je použita pro výpočet skóre (více v podsekci 6.4.4), a také jednotky, v nichž se výsledky tohoto testu zaznamenávají (slouží pro následné zobrazování). Příklad takového testu je načtení stránky www.google.com v prohlížeči Internet Explorer.
- TestGroup – Každý test musí být součástí nějaké skupiny testů. TestGroup tak funguje pro přehlednější organizaci a zobrazování výsledků. Jako příklad skupiny testů uvedu načítání stránek v prohlížeči Internet Explorer.
- TestSuite – Nejvyšší entita pro organizaci testů. Na této úrovni rozeznáváme pouze komplexní celky, jako jsou performance testy webového štítu nebo performance testy štítu souborového systému.
- TestConfiguration – V kontextu performance testů je konfigurací testu myšlena kombinace zapnutých komponent Avastu. Tyto konfigurace mohou být pro každou TestSuite jiné (případ performance testů) nebo společné (obecný případ).
- TestRun – Ačkoliv název této entity napovídá, že se jedná o spuštění jednoho testu, ve skutečnosti je to spuštění všech testů z vybrané TestSuite. Tato entita by se proto mohla jmenovat spíše SuiteRun nebo TestsRun, aby název odpovídal „spuštění TestSuite“ nebo „spuštění testů“, nicméně stávající název jsem zvolil pro lepší používání v mluvené řeči. Informace, které uchovává, se váží ke stavu při spuštění testů: verze operačního systému, verze Avastu a jeho antivirové databáze, identifikátor použitého počítače, datum a čas spuštění/ukončení, celkový čas průběhu všech testů a případný komentář.
- TestResult – Výsledek jednoho konkrétního testu. Obsahuje délku trvání testu, číslo pro kód chybového hlášení a případný komentář.

6.4 Zobrazování výsledků

Podle původního plánu měl mít zobrazování výsledků na starosti kolega, který by vytvořil interaktivní webové rozhraní. Tomu byly ovšem zadány úkoly na novém projektu, a proto již neměl čas věnovat se grafické interpretaci výsledků uložených v databázi. Abych tedy mohl výsledky svého měření vyhodnotit, musel jsem nad rámec zadání této práce vytvořit vlastní nástroj pro zobrazování výsledků.

Uživatelské rozhraní aplikace (viz obrázek 6.2) je rozděleno na tři základní části, které budou popsány v následujících třech podsekcích.



Obrázek 6.2: Uživatelské rozhraní aplikace pro zobrazování výsledků

6.4.1 Rozbalovací seznamy

Každý rozbalovací seznam slouží k filtrování výsledků a vybrání skupin dat, jejichž výsledky mají být zobrazovány. Jednotlivé kategorie odpovídají databázovému modelu:

- TestSuites – Při startu programu jsou načteny všechny TestSuite, jejichž název obsahuje klíčové slovo „performance“. Při výběru konkrétní TestSuite jsou pak načteny všechny objekty, které jsou s ní asociované: TestGroup, které obsahují testy, TestConfig, které mají alespoň jeden TestRun a AvastVersion (složeno z jednotlivých složek verze Avastu). Načtením těchto objektů se tím pádem naplní seznamy TestGroups, TestConfigs a AvastVersions. Následuje vytvoření os všech grafů na základě všech získaných TestConfig a TestVersion. Poté je odpovídajícími daty naplněn graf související s TestSuite.

- TestGroups – Po vybrání konkrétní skupiny testů jsou načteny všechny testy, které mají výsledky, a je tak naplněn seznam Tests. Pak jsou do modelu grafu pro TestsGroups připravena správná data.

- Tests – Výběrem konkrétního testu se z databáze stáhnou všechny výsledky úspěšných testů a jsou použity pro odpovídající graf.

- TestConfigurations a AvastVersions – Hodnoty vybrané z těchto seznamů jsou použity pro data v tabulkách.

6.4.2 Tabulky s výsledky

Po stisknutí tlačítka „Show results“ jsou na základě vybraných hodnot z vyjmenovaných seznamů v minulé podsekci vyfiltrována data a načtena do příslušných tabulek. Protože obě tabulky neslouží jen pro přesné zobrazení výsledků, ale také pro porovnání mezi konfiguracemi/verzemi a dále také zkontrolování kvality dat, disponují těmito sloupci:

1. Název konkrétní agregace (číslo verze nebo konfigurace)
2. Počet úspěšných měření
3. Počet neúspěšných měření
4. Minimální dosažený čas
5. Maximální dosažený čas
6. Průměr časů
7. Medián časů
8. Směrodatná odchylka

6.4.3 Grafy

Na základě vybraných položek ze zmíněných rozbalovacích seznamů jsou zobrazovány tyto grafy:

- Graf skóre souboru testů (TestSuite)
- Graf skóre skupiny testů (TestGroup)
- Graf průměrného času konkrétního testu

Všechna data pro každý graf jsou načtena do vlastní vyrovnávací paměti ihned při první potřebě graf vykreslit. Při změně hodnot zaškrtávacích polí pro dostupné konfigurace již pak proto nedochází k dalším dotazům nad daty, ale jsou použita již předem načtená. Změna konfigurace pak ovlivňuje všechny grafy najednou (přidání/odebrání série dat).

6.4.4 Výpočet skóre

První dva grafy potřebují pro své zobrazení takzvané skóre, které se kaskádově počítá od nejvyšší abstrakce (TestSuite) po nejnižší (TestResult). Vzato „odspoda“ se nejdříve pro každý výsledek vypočítá poměr základní hodnoty (BaseValue) ku naměřené hodnotě, a to celé vynásobeno tisícem, tedy:

$$\frac{zakladni\_hodnota}{namerena\_hodnota} * 1000$$

S postupem na vyšší úrovně jsou pak počítány průměry hodnot nižší úrovně. Využití vypočítaného skóre je nasnadě: získání přehledu o stavu performance na úrovni větších celků, než je samostatný test.

6.5 Plány do budoucna

V průběhu vypracovávání této práce se objevily další nápady, možnosti a performance problémy, které by bylo dobré řešit v rámci performance testů. Bohužel rozsahově je realizace následujících myšlenek absolutně mimo rozsah této práce, proto je alespoň pro představu v krátkosti popíšu.

6.5.1 Automatické vyhodnocování xperf

Pomocí xperf lze kromě analýzy využití prostředků počítače také získat strom volání funkcí. [21] Výstup se dá zapsat do souboru ve formátu XML a z něj pak není problém jednoduše dostat žádaná data jako jméno funkce a čas v ní strávený. Tyto informace by pak poskytly vývojářům konkrétní představy o tom, v jakých částech programu při daných aktivitách se tráví nejvíce času.

6.5.2 Porovnávání výsledků různých testů

V současné podobě poskytuje aplikace pro zobrazování výsledku dostatečnou vypovídací hodnotu. Některé testy (jako například ApacheBench) se pouštějí vícekrát na stejných datech, ale jiných vstupních konfiguracích, které mohou být zásadní pro chování Avastu v průběhu testu. Proto by se hodil interaktivní graf (tabulka), který by dokázal přehledně porovnat výsledky více různých testů.

6.5.3 Přehrávání videí na YouTube

Nedeterministicky se začal objevovat problém, kdy se zapnutým Avastem dochází k zasekávání videí přehrávaných na serveru YouTube. Pomocí API IFrame přehrávače lze na jakoukoliv stránku umístit přehrávač videí YouTube, který je možné ovládat skrze JavaScript. Lze tak mimo jiné spustit/zastavit přehrávání, zjistit celkovou délku videa nebo dobu přehrávání. [40]

Kapitola 7

Vyhodnocení výsledků

Závěrečná kapitola poskytuje naměřené výsledky a jejich základní interpretaci. Veškeré závěry usouzené z naměřených výsledků vznikly na základě konzultací s vývojáři odpovědnými za danou komponentu. Tyto závěry jsou předběžné, proto budou muset být ještě detailně vyzkoušeny všechny vzniklé hypotézy a také musí být provedena hlubší analýza problémů ze strany vývojářů.

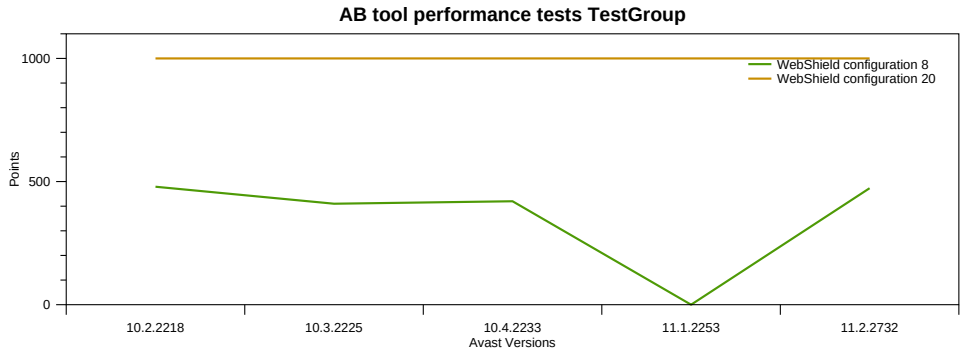
V grafech vygenerovaných mým nástrojem pro zobrazování výsledků jsou u popisků konfigurací uvedena čísla pro jejich jednoznačnou identifikaci. Zde publikované grafy sice obsahují vždy pouze dvě konfigurace (s nainstalovaným Avastem a bez), ale obecně existuje konfigurací více (5.2.3). Proto zde uvedu jejich interpretaci:

- S Avastem – FileSystemShield configuration 5 a WebShieldConfiguration 8
- Bez Avastu – FileSystemShield configuration 6 a WebShieldConfiguration 20

Grafy získané z nástroje xperf nemají popisky linií vůbec, protože jsou v programu vyčleněny jako ovládací prvky a nejsou součástí grafu. Proto jsem se rozhodl pro výmluvný slovní popis za použití jejich barev.

Tabulka 7.1: Výsledky ApacheBench s Avastem ve verzi 11.2.2732 pro instalátor LibreOffice

Konfigurace	# měření	Min	Max	Průměr	Medián	Odchylka
S Avastem	43	14,317	19,429	15,647	15,492	1,303
Bez Avastu	36	7,379	7,385	7,384	7,385	0,002



Obrázek 7.1: Skóre skupiny ApacheBench

7.1 ApacheBench

Všechny výsledky této skupiny ve většině případů odpovídají grafu 7.1, což znamená, že testy byly dvakrát pomalejší s nainstalovaným Avastem a také že v poslední verzi bylo zaznamenáno mírné zlepšení oproti předchozím verzím.[22] Tabulka 7.1 pak pro představu doplňuje konkrétní data pro stažení instalátoru LibreOffice. Díky výstupu xperf jsem pak dokázal identifikovat konkrétní funkce, které způsobují zpomalení, a jejich strom volání spolu s dalšími informacemi jsem předal odpovědným vývojářům.

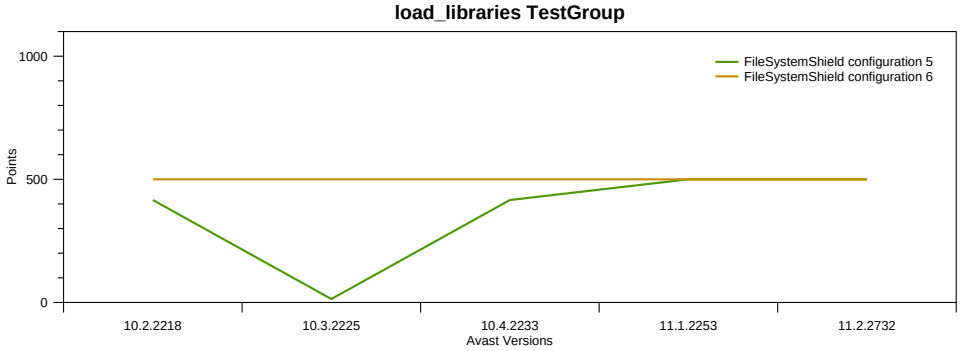
7.2 Internet Explorer a Google Chrome

Výsledky těchto testů jsou velkým zklamáním, protože vysoká odezva serverů způsobila velké odchylky, což v lepším případě znamenalo stejné časy ve všech konfiguracích a v horším případě lepší časy při nainstalovaném Avastu než s čistým systémem. Tomuto faktu měla zabránit proxycache (viz 5.1.7), která ovšem nefungovala tak, jak bylo předpokládáno, takže její použití nemělo žádný efekt na výsledky. Po tomto nezdaru jsem se pokusil ještě si vytvořit lokální kopii jednotlivých stránek, které bych pak nahrál na vlastní webový server. Bohužel na všech stránkách byl vždy minimálně jeden skript, který se nestáhl, a proto načítání skončilo až po vypršení časového limitu. Vzhledem k těmto faktům budu muset naměřit mnohem více dat (pravděpodobně tisíce), nad kterými bych aplikoval některé statistické metody.

[22] Propad ve verzi 11.1.2253 je způsoben tím, že se v této verzi nachází chyba blokující testy. Slova „mírné zlepšení“ se tak nepojí k tomuto skoku, ale k verzím ještě dřívějším.

Tabulka 7.2: Načtení knihovny v sekundách

Verze	# měření	Min	Max	Průměr	Medián	Odchylka
10.2.2218	13	0,016	0,047	0,036	0,031	0,010
10.3.2225	16	0,905	0,967	0,937	0,936	0,020
10.4.2233	8	0,016	0,047	0,031	0,031	0,012
11.1.2253	51	0	0,920	0,049	0,031	0,127
11.2.2732	35	0	0,047	0,024	0,031	0,013



Obrázek 7.2: Skóre skupiny načtení knihoven

7.3 Načtení knihoven

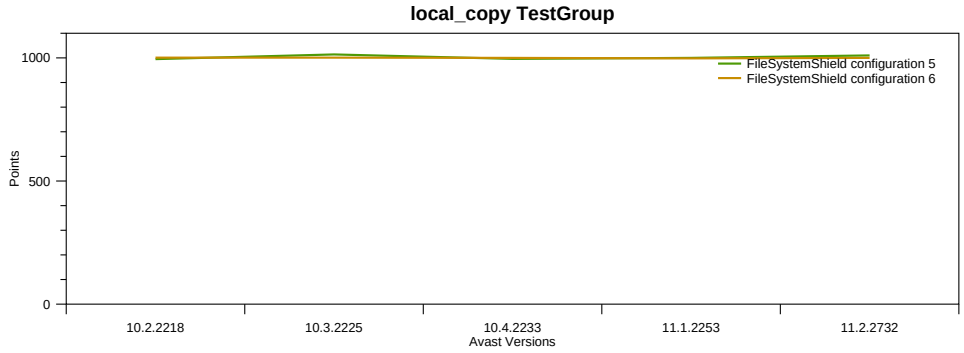
Realizace toho testu nedopadla podle očekávání. Z tabulky 7.2 lze vyčíst, že načítání knihoven (v tomto případě dokonce té největší ze sady) proběhne velmi rychle, a proto i minima v některých verzích jsou rovna nule, protože tak krátká doba nebyla měřitelná. Dobře to je vidět i na grafu 7.2, že skóre konfigurace bez Avastu dosahuje hodnoty 500, což znamená, že každý druhý výsledek nebyl měřitelný. S největší pravděpodobností jsem tak vybral binární soubory, jež mají krátkou sekci s kódem, která se nahrává do paměti, a dlouhou sekci se zdroji[23], které se nenahrávají. Pro budoucí měření tak budou pro tento test nejlepší známé knihovny s minimem zdrojů. Další vliv na výsledek může mít fakt, že knihovny zůstávají v paměti, a proto nedochází k opakovanému načítání z disku. Pro další měření bude tedy potřeba ověřit, že knihovna již není aktivní, a vyčistit paměť ve stavu Standby.

Na zmíněném grafu a tabulce lze pozorovat zajímavý fenomén ve verzi 10.3.2225, kde čas potřebný pro načtení knihovny byl 30krát vyšší než v jakékoliv jiné verzi. Nicméně vzhledem k naprosto zanedbatelnému počtu uživatelů vlastnících tuto verzi nemá smysl provádět hlubší analýzu objeveného problému.

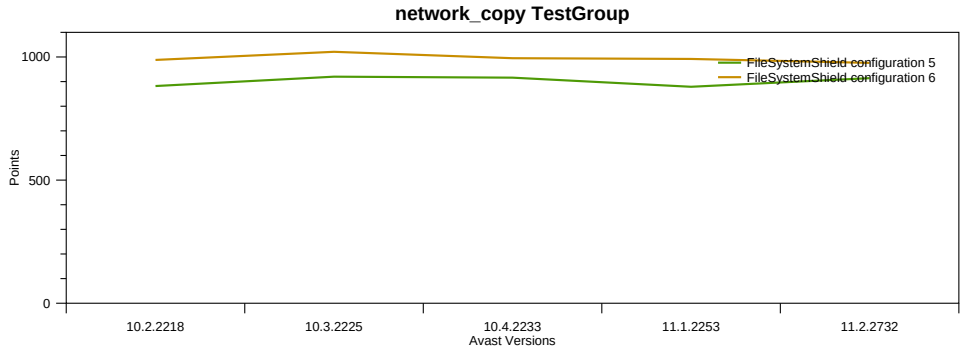
[23] resources

Tabulka 7.3: Kopírování souboru na pevný disk v sekundách

Konfigurace	# měření	Min	Max	Průměr	Medián	Odchylka
S Avastem	35	25,693	26,707	26,264	26,208	0,282
Bez Avastu	18	26,083	26,754	26,483	26,520	0,211



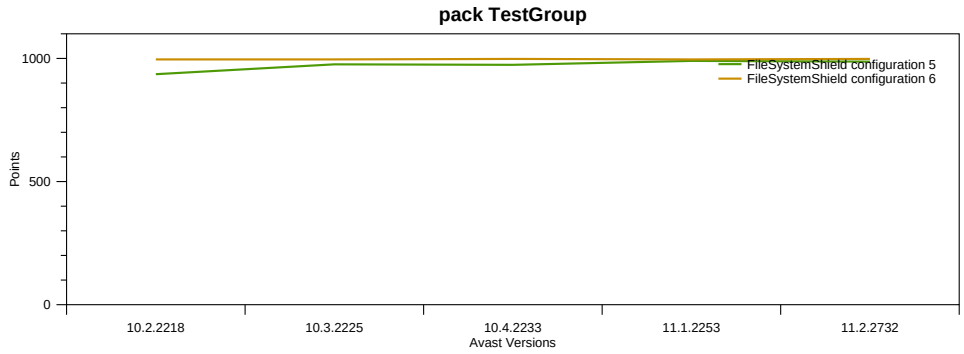
Obrázek 7.3: Skóre skupiny kopírování souborů na lokání disk



Obrázek 7.4: Skóre skupiny kopírování souborů na síťový disk

7.4 Kopírování souborů na lokální disk

Díky optimalizacím odpovědného vývojáře nejsou výsledky lokálního kopírování překvapivé. Na grafu 7.3 vidíme, že v průběhu verzí se performance nemění a časy s Avastem a bez něj jsou téměř totožné. Pro úplnost přikládám ještě tabulku 7.3, která demonstruje srovnání na poslední verzi s jedním konkrétním souborem.



Obrázek 7.5: Skóre skupiny zabalení souborů do archívu

Tabulka 7.4: Zabalení souboru do archívu v sekundách

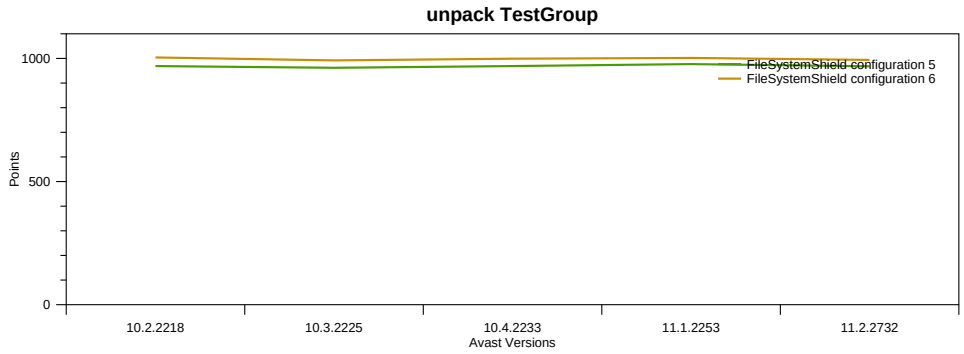
Konfigurace	# měření	Min	Max	Průměr	Medián	Odchylka
S Avastem	35	18,845	20,171	19,610	19,672	0,325
Bez Avastu	18	18,923	19,640	19,369	19,430	0,199

7.5 Kopírování souborů na síťový disk

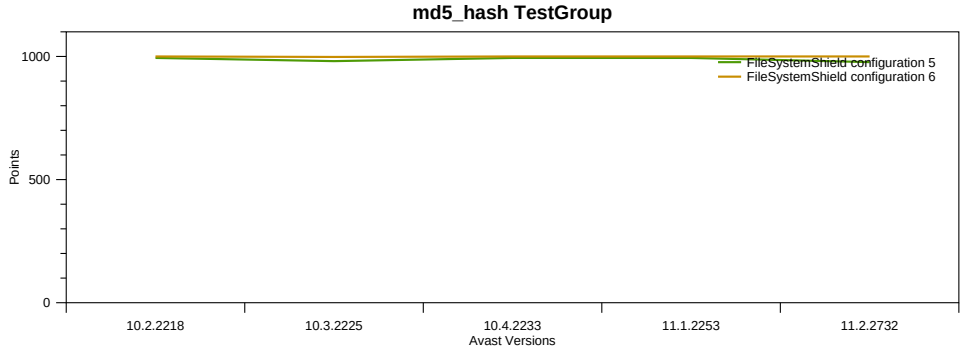
V nastavení štítu souborového systému je v základu zaškrtnuta možnost „Netestovat soubory na vzdálených složkách“, proto by srovnání konfigurací mělo odpovídat předchozímu případu. Nicméně se objevily rozdíly mezi jednotlivými soubory. Zatímco některé byly zkopírovány vždy za stejný čas nehledě na konfiguraci, jiné zaznamenaly s Avastem zhoršení (celkový pohled na tuto skupinu přináší graf 7.4). Při namátkové kontrole výstupu xperf jsem zjistil, že v případech, kde se časy v různých konfiguracích rozchází, se do testu „připletl“ ovladač zvukové karty (služba audiodg.exe) a indexování disku (SearchProtocolHost.exe). Je podezřelé, že by tyto aktivity nastaly vždy jen u konkrétních souborů, proto by bylo potřeba nashromáždit xperf záznamy pro všechna měření. Jejich vyhodnocování by zabralo ovšem hodně času, proto by se nejen pro tyto případy hodilo automatické vyhodnocování zmíněné v 6.5.1.

7.6 Zabalení souborů do archívu

Avast nemá v tomto testu důvod pracovat se zabalovanými soubory, a tomu také odpovídá graf 7.5. Pro úplnost přikládám opět srovnání konfigurací v poslední verzi v tabulce 7.4.



Obrázek 7.6: Skóre skupiny rozbalení souborů z archívu



Obrázek 7.7: Skóre skupiny počítání hash souborů

Tabulka 7.5: Kopírování souborů pomocí vlastní implementace v sekundách

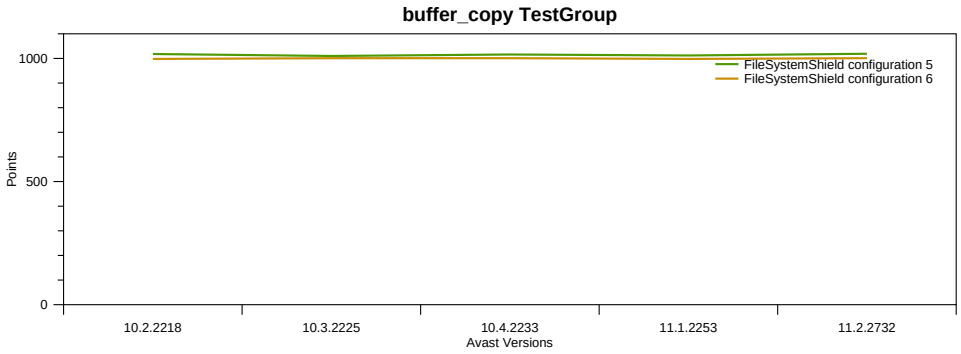
Konfigurace	# měření	Min	Max	Průměr	Medián	Odchylka
S Avastem	35	6,739	15,272	13,583	14,789	2,069
Bez Avastu	18	13,556	15,818	15,421	15,538	0,552

7.7 Rozbalení souborů z archívu

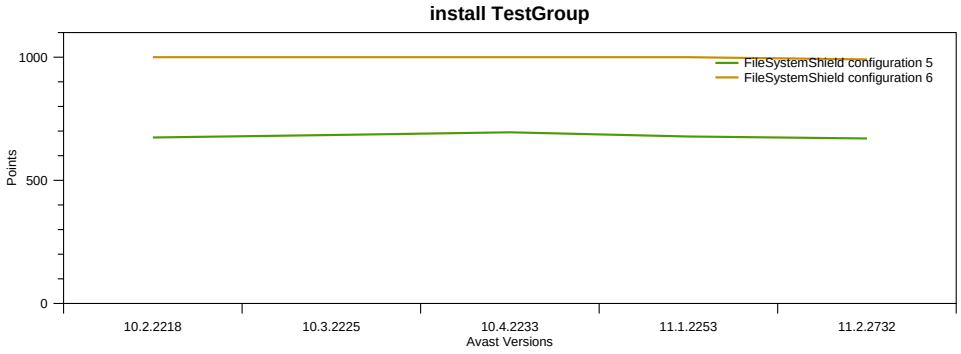
I zde stejně jako v testu 7.4 funguje optimalizace pro zapisování souborů na disk, proto graf 7.6 není žádným velkým překvapením.

7.8 Počítání hash souborů

Když se počítá hash, obsah souborů se samozřejmě nemění, proto odpovídající graf 7.7 je opět vcelku nezajímavý.



Obrázek 7.8: Skóre skupiny kopírování souborů pomocí vlastní implementace

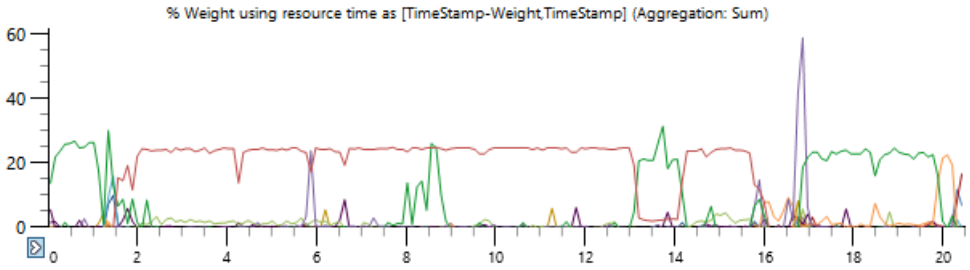


Obrázek 7.9: Skóre skupiny instalace a spuštění programů

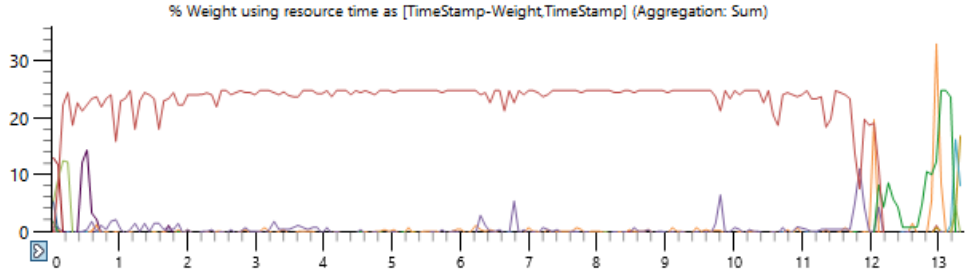
7.9 Kopírování souborů pomocí vlastní implementace

I přes jiné očekávané chování k vlastní implementaci kopírování zmíněné v podsekci 6.2.3.7 i zde zafungovala silná optimalizace, proto opět dopad Avastu na kopírování není měřitelný (graf 7.8). Z tohoto důvodu by bylo dobré vytvořit takový test pro kopírování, který by svým chováním dokázal optimalizace „obejít“. Zajímavé je srovnání tabulek 7.3 a 7.5, na nichž lze pozorovat, že vlastní implementace kopírování je skoro dvakrát rychlejší než vestavěná. To je způsobeno transakčním API použitém v `CopyFile`, které zpomaluje celé kopírování.[24]

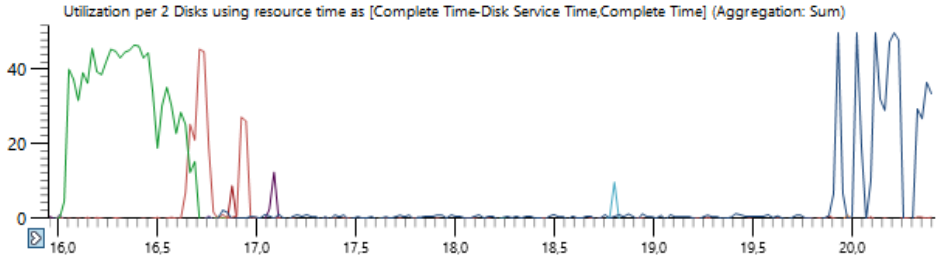
[24]Tuto informaci jsem získal od kolegy Luboše Hnaníčka, Senior Researcher, který analyzoval implementaci v systémové knihovně Kernel32.



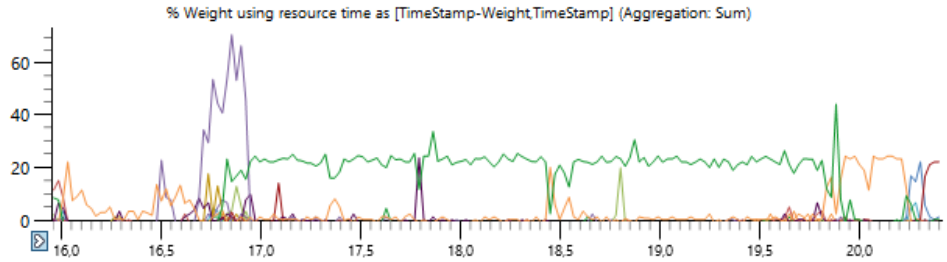
Obrázek 7.10: Vytížení procesoru při instalaci programu GIMP s Avastem



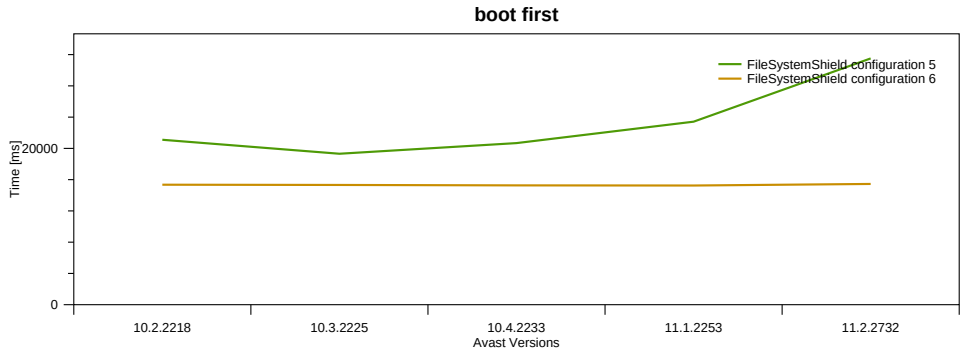
Obrázek 7.11: Vytížení procesoru při instalaci programu GIMP bez Avastu



Obrázek 7.12: Vytížení disku po instalaci programu GIMP bez Avastu



Obrázek 7.13: Vytížení procesoru po instalaci programu GIMP bez Avastu



Obrázek 7.14: Čas prvního startu počítače po instalaci Avastu

7.10 Instalace a spuštění programů

Již ze souhrného grafu 7.9 vidíme, že skóre této skupiny testů dosahuje přibližně 700, proto bude zajímavé se na problém podívat podrobněji pomocí xperf. Protože je průběh instalací velmi podobný, vybral jsem jako reprezentativního zástupce pro demonstraci program GIMP. Při porovnání grafů 7.10 a 7.11 si můžeme všimnout, jak je průběh samotné instalace (červená linie na obou grafech) ovlivněn Avastem (zelená linie na grafu 7.10). Před spuštěním instalačního souboru je totiž zkontrolován jeho obsah (protože se jedná o archív) a dále je provedeno skenování v 8. a 13. vteřině.

Zajímavé je také se podívat na aktivity po skončení instalace (od 16. vteřiny dále), tam totiž nastává moment mapování binárních souborů testovaného programu do paměti. Na grafu 7.12 je vidět vytížení disku při zápisu dat na disk po vynucení vyprázdnění vyrovnávací paměti (zelená linie). To je následováno na grafu 7.13 zmíněným mapováním. Ve chvíli, kdy se začne k souborům přistupovat, Avast je oskenuje a tím způsobí viditelné vytížení procesoru (zelená line na grafu 7.13) a další zdržení. Protože toto zdržení je stejně velké jako to při samotné instalaci a jedná se spíše o syntetický test, bylo by dobré pro další měření rozdělit výsledky tohoto testu do dvou samostatných: instalace a spuštění.

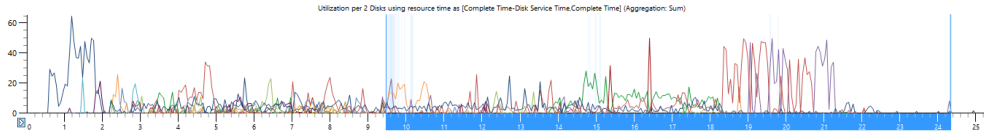
Tabulka 7.6: Vytížení disku konkrétními procesy bez úvodního skenování

Process	IO Time (μs)	Size (B)	Disk Service Time (μs)
System	5 073 930,545	944 345 600	2 469 058,865
AvastSvc.exe	14 272 178,718	212 333 056	2 350 152,302
afwServ.exe	1 171 414,256	71 101 952	918 164,825
backup.exe	12 267 148,176	12 918 272	883 196,032

Tabulka 7.7: Vytížení disku konkrétními procesy s úvodním skenováním

Process	IO Time (μs)	Size (B)	Disk Service Time (μs)
System	4 253 532,067	607 401 984	1 806 114,692
AvastSvc.exe	11 777 885,511	126 519 808	957 619,559
afwServ.exe	893 570,878	80 468 480	756 625,371
backup.exe	11 420 563,225	13 446 656	647 450,238

Obrázek 7.15: Přístup na disk procesu instup.exe ve verzi 10.4.2233

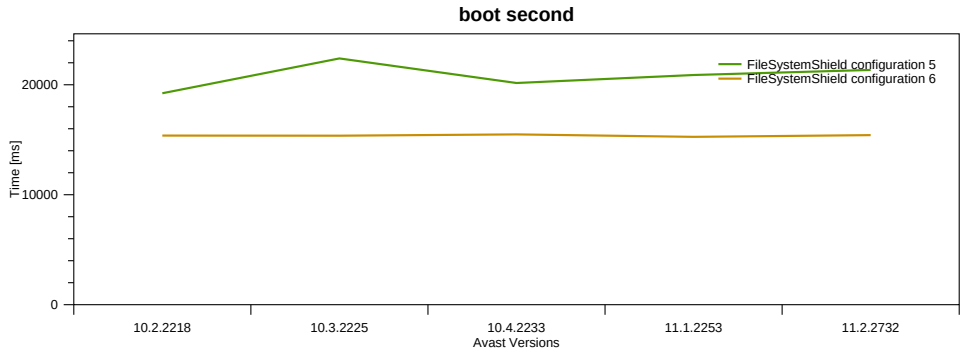


Obrázek 7.16: Přístup na disk procesu instup.exe ve verzi 11.2.2732

7.11 Start počítače

Pohled na graf 7.14 přináší otázku, jaké byly provedeny změny kódu od verze 10.3.2225 do 11.2.2732, že došlo k takovémuto razantnímu zhoršení při prvním startu počítače po instalaci Avastu? Jedna z odpovědí je zrušení úvodního skenování ihned po instalaci Avastu, které kontroluje podpisy systémových souborů a nainstalovaných aplikací. Tato kontrola se tak začne vykonávat ihned po restartu počítače a může zapříčinit zpoždění. Abych nasimuloval na nové verzi úvodní skenování, pustím ho z GUI manuálně a poté restartuji. V tabulce 7.6 lze vidět vytížení disků jednotlivými procesy bez skenování a v tabulce 7.7 s manuálním spuštěním. Při porovnání vidíme, že skutečná práce s diskem (Disk Service Time) v procesu AvastSvc.exe klesla přes polovinu.

Žádné další změny, které by měly ovlivnit start počítače jsem nebyl schopen dohledat, což vůbec neznamená, že neexistují. Nicméně při procházení xperf výstupu jsem našel nesrovnalost s procesem instup.exe, který po každém startu počítače kontroluje, zda není dostupná nová verze Avastu nebo její virové databáze. Na grafu 7.15 lze sledovat, že tento proces ve verzi 10.4.2233 běžel celkem kolem dvou vteřin (modrá výplň osy x), a na grafu 7.16 je vidět, že ten samý proces byl spuštěn po dobu 15 vteřin, přitom s diskem skoro nepracoval (čím tmavší odstín modré barvy uvnitř grafu, tím intenzivnější práce). V logovacích souborech pak lze nalézt, že proces několikrát čeká na



Obrázek 7.17: Čas dalších startů počítače po instalaci Avastu

odpověď serveru. Kromě vyřešení tohoto problému by se měly aktivity procesu instup.exe přesunout na pozdější čas, aby jakkoliv nezpomalovaly start počítače.

V tabulce 7.6 se na čtvrtém místě nachází program backup.exe, který slouží pro zálohu nastavení a licence Avastu pro případ aktualizace operačního systému na Windows 10, aby v případě problému byly tyto soubory zachovány a následně obnoveny. Tento program byl distribuován na všechny stávající verze, proto nemůže způsobovat rozdíl mezi verzemi (graf 7.14), nicméně může obecně způsobovat zpomalení vzhledem k tomu, že i na starších verzích je rozdíl oproti čistému systému bez Avastu 5 vteřin. Stejně jako instup.exe by mohlo být jeho spuštění odloženo.

Další starty počítače již mají napříč verzemi celkem konstantní průběh (graf 7.17), nicméně za startem čistého systému stále zaostávají na testovacím stroji o 5 vteřin. Při pohledu do výstupu xperf lze vidět, že AvastSvc.exe přistupuje celou tuto dobu na disk, konkrétně čte katalogy (C:\Windows\system32\catroot\), které obsahují digitální podpisy souborů. Chování je očekávané, ale nemělo by trvat takto dlouho, proto bude potřeba zjistit, které konkrétní soubory se při startu oproti těmto katalogům kontrolují.

Závěr

Hlavním cílem této diplomové práce bylo navrhnout a naimplementovat performance testy antiviru Avast a následně je začlenit do stávající testovací infrastruktury a provést potřebná měření. Tento cíl byl dosažen následujícím postupem:

Byl poskytnut teoretický základ pro performance testování včetně základních pojmů, principů a postupů. V práci je také věnováno místo některým technikám a používání API při programování ve Windows, které byly zapotřebí v této práci. Dále byly představeny jednotlivé komponenty antiviru Avast, které mohou mít viditelný vliv na výkon počítače, a to takovým způsobem, aby na základě těchto informací mohly být vyvozeny závěry z naměřených výsledků. Součástí této práce je i popis využitých externích nástrojů, bez nichž by tato práce nemohla existovat nebo by musela být mnohem obsáhlejší. Před samotným vytvářením testů byla provedena analýza a konfigurace testovací infrastruktury. V neposlední řadě byl proveden návrh a implementace performance testů, které byly začleněny do stávající infrastruktury, a jejich výsledky byly vyhodnoceny.

Přínosem této práce je převážně výsledný produkt, díky němuž lze změřit vliv antiviru Avast na výkon počítače, a prvotní naměřené výsledky. Tento vliv lze měřit zpětně a vyhodnotit tak regresi. Vzhledem k začlenění testů do infrastruktury je možné tyto testy pouštět pravidelně nejenom na oficiálně vydaných verzích, ale primárně na těch interních, aby zákazníkům byla doručena již optimalizovaná aplikace s minimálním dopadem na výkon jejich počítače. Vedlejším přínosem je vznik jakési platformy, na které mohou být postaveny další testy s mnohem menší časovou náročností a díky tomu i jednoduše nasazeny. V práci byly naznačeny některé plány do blízké budoucnosti.

Měření se odehrávala pouze na jednom fyzickém počítači, a to na dnešní poměry výkonném. Bude tak potřeba do infrastruktury zapojit i slabé počítače, notebooky a tablety (s operačním systémem Windows). Microsoft, který posílá

výsledky vlastních testů, zaslal konkrétní seznam svých počítačů, na nichž provádí svá měření. Dostupné počítače tak budou zakoupeny a zapojeny do infrastruktury.

Díky této diplomové práci jsem se seznámil s velmi zajímavým odvětvím v rámci testování, kterému bych se rád dál věnoval, a rozšířil tak stávající množinu automatických performance testů.

Literatura

[1] Meier, J.; Farre, C.; Bansode, P.; aj.: *Performance Testing Guidance for Web Applications*. 2007.

[2] Molyneaux, I.: *The Art of Application Performance Testing*. O'Reilly Media, Inc., 2014, ISBN 978-1-491-90054-3.

[3] Hart, J. M.: *Windows System Programming*. Addison-Wesley, 2010, ISBN 978-0-321-65774-9.

[4] Unicode, Inc.: What is Unicode? [online], 2015 [cit. 2016-3-2], dostupné z WWW: `http://unicode.org/standard/WhatIsUnicode.html`.

[5] Python Software Foundation: Unicode HOWTO. [online], 2016 [cit. 2016-3-3], dostupné z WWW: `https://docs.python.org/3/howto/unicode.html`.

[6] Python Software Foundation: ctypes — A foreign function library for Python. [online], 2016 [cit. 2016-3-3], dostupné z WWW: `https://docs.python.org/3.4/library/ctypes.html`.

[7] Microsoft Corporation: Filter Manager and Minifilter Driver Architecture. [online], 2016 [cit. 2016-3-20], dostupné z WWW: `https://msdn.microsoft.com/en-us/library/windows/hardware/ff541591(v=vs.85).aspx`.

[8] Microsoft Corporation: Change Journals. [online], 2016 [cit. 2016-3-20], dostupné z WWW: `https://msdn.microsoft.com/en-us/library/aa363798.aspx`.

[9] Russinovich, M.; Solomon, D.; Ionescu, A.: *Windows Internals, Part 1 (6th Edition)*. Microsoft Press, 2012, ISBN 978-0-7356-4873-9.

[10] Software Freedom Conservancy: About git. [online], 2016 [cit. 2016-2-24], dostupné z WWW: `https://git-scm.com`.

[11] Dudler, R.: git - the simple guide. [online], 2016 [cit. 2016-2-24], dostupné z WWW: http://rogerdudler.github.io/git-guide/.

[12] WPKGSysop: Main Page. [online], 2015 [cit. 2016-2-27], dostupné z WWW: http://wpkg.org.

[13] CloudBees, Inc.: About Jenkins CI. [online], 2010–2016 [cit. 2016-2-27], dostupné z WWW: https://www.cloudbees.com/jenkins/about.

[14] Fowler, M.: Continuous Integration. [online], 2006 [cit. 2016-2-27], dostupné z WWW: http://martinfowler.com/articles/continuousIntegration.html.

[15] The Apache Software Foundation: ab - Apache HTTP server benchmarking tool. [online], 2016 [cit. 2016-2-24], dostupné z WWW: https://httpd.apache.org/docs/2.2/programs/ab.html.

[16] Cardaci, A.: Capture HAR files from a remote Chrome instance. [online], 2012 [cit. 2016-2-9], dostupné z WWW: https://github.com/cyrus-and/chrome-har-capturer.

[17] Dyke, A.: HAR File. [online], 2015 [cit. 2016-2-9], dostupné z WWW: https://www.maxcdn.com/one/visual-glossary/har-file/.

[18] Odvarko, J.: HAR 1.2 Spec. [online], 2007 [cit. 2016-2-9], dostupné z WWW: http://www.softwareishard.com/blog/har-12-spec/.

[19] McFarland, D.: How to Install Node.js® and NPM on Windows. [online], 2015 [cit. 2016-2-13], dostupné z WWW: http://blog.teamtreehouse.com/install-node-js-npm-windows.

[20] Microsoft Corporation: Windows Performance Toolkit Overview. [online], 2013 [cit. 2016-4-22], dostupné z WWW: https://msdn.microsoft.com/en-us/library/windows/hardware/hh162981.aspx.

[21] Sträßner, C.: Windows Performance Toolkit – Xperf. [online], 2008 [cit. 2016-4-20], dostupné z WWW: https://blogs.msdn.microsoft.com/ntdebugging/2008/04/03/windows-performance-toolkit-xperf/.

[22] OxyPlot. [online], 2015 [cit. 2016-2-28], dostupné z WWW: http://oxyplot.org.

[23] Rouse, M.: master/slave. [online], 2008 [cit. 2016-4-2], dostupné z WWW: http://searchnetworking.techtarget.com/definition/master-slave.

[24] Association INDEPNET: The GLPI Project. [online], 2002 [cit. 2016-4-3], dostupné z WWW: http://www.glpi-project.org/spip.php?article43.

[25] Gibb, R.: Proxy Caching. [online], 2015 [cit. 2016-4-22], dostupné z WWW: https://www.maxcdn.com/one/visual-glossary/proxy-caching/.

[26] Baquiran, M.; Wren, D.: 2015 Consumer Security Products Performance Benchmarks (Edition 2).

[27] Russinovich, M.; Solomon, D.; Ionescu, A.: *Windows Internals, Part 2 (6th Edition)*. Microsoft Press, 2012, ISBN 978-0-7356-6587-3.

[28] Hart, C.: PurgeStandbyList. [online], 2014 [cit. 2016-4-14], dostupné z WWW: https://gist.github.com/bitshifter/c87aa396446bbebeab29.

[29] Hemphill, J.: ARP cache: What is it and how can it help you? [online], 2008 [cit. 2016-4-15], dostupné z WWW: https://www.petri.com/csc_arp_cache.

[30] Mitchell, B.: What Is a DNS Cache? [online], 2014 [cit. 2016-4-15], dostupné z WWW: http://compnetworking.about.com/od/dns_domainnamesystem/f/what-is-a-dns-cache.htm.

[31] Rouse, M.: Certificate Revocation List (CRL). [online], 2007 [cit. 2016-4-15], dostupné z WWW: http://searchsecurity.techtarget.com/definition/Certificate-Revocation-List.

[32] GlobalSign: View and/or Delete CRL, OCSP Cache. [online], 2014 [cit. 2016-4-15], dostupné z WWW: https://support.globalsign.com/customer/portal/articles/1353318-view-and-or-delete-crl-ocsp-cache.

[33] Microsoft Corporation: Measuring Browser Performance with the Windows Performance Tools. [online], 2010 [cit. 2016-4-15], dostupné z WWW: https://blogs.msdn.microsoft.com/ie/2010/06/21/measuring-browser-performance-with-the-windows-performance-tools/.

[34] Basques, K.; Kearney, M.: Evaluating network performance. [online], [cit. 2016-4-15], dostupné z WWW: https://developers.google.com/web/tools/chrome-devtools/profile/network-performance/resource-loading#copy-save-and-clear-network-information.

[35] Sikuli Lab: Sikuli Script. [online], [cit. 2016-4-16], dostupné z WWW: http://www.sikuli.org/.

[36] Russinovich, M.: Process Explorer v16.12. [online], 2016 [cit. 2016-4-16], dostupné z WWW: https://technet.microsoft.com/en-us/sysinternals/processexplorer.aspx.

[37] Microsoft Corporation: FlushFileBuffers function. [online], 2016 [cit. 2016-4-17], dostupné z WWW: https://msdn.microsoft.com/en-us/library/windows/desktop/aa364439(v=vs.85).aspx.

[38] Python Software Foundation: Work with ZIP archives. [online], 2016 [cit. 2016-4-17], dostupné z WWW: https://docs.python.org/3/library/zipfile.html.

[39] Microsoft Corporation: Open Event Viewer. [online], 2016 [cit. 2016-4-19], dostupné z WWW: http://windows.microsoft.com/en-us/windows/open-event-viewer#1TC=windows-7.

[40] Google, Inc.: YouTube Player API Reference for iframe Embeds. [online], 2016 [cit. 2016-4-20], dostupné z WWW: https://developers.google.com/youtube/iframe_api_reference.

Příloha **A**

Seznam použitých zkratek

HAR HTTP Archive

HTTP Hypertext Transport Protocol

DNS Domain Name System

API Application Program Interface

SEH Structured Exception Handling

WFP Windows Filtering Platform

HTML HyperText Markup Language

XML Extensible Markup Language

UAC User Account Control

ADK Assessment and Deployment Kit

JSON JavaScript Object Notation

GUI Graphical User Interface

ARP Address Resolution Protocol

MAC Media Access Control

PKI Public Key Infrastructure

COM Component Object Model

Příloha **B**

Obsah přiloženého DVD

- `readme.txt` ..... stručný popis obsahu DVD
- `data` ..... adresář obsahující soubory použité pro účely testů
- `exe` .... adresář se spustitelnou formou nástroje pro zobrazování výsledků
- `src`
  - `PageListCleaner` ... zdrojové kódy aplikace pro uvolnění paměťových stránek
  - `PerformancePlot` ... zdrojové kódy nástroje pro zobrazování výsledků
  - `PerformanceTests` ..... zdrojové kódy implementace automatických performance testů
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF