MATEMATIČKI FAKULTET
UNIVERZITET U BEOGRADU

Alati i pristup rešavanju problema curenja memorije u JavaScript aplikacijama

Master rad

Miloš Kralj

Beograd 2014 godine

Univerzitet u Beogradu – Matematički fakultet

Master rad

Autor:	Miloš Kralj 1042/2009
Naslov:	Alati i pristup rešavanju problema curenja memorije u JavaScript aplikacijama
Mentor:	dr Vladimir J. Filipović
Članovi komisije:	dr Saša Malkov
	dr Dušan D. Tošić
Datum:	

Predgovor

Ovaj rad objašnjava značaj problema curenja memorije u *Windows 8* aplikacijama, teškoće prilikom rešavanja tih problema bez specijalizovanih alata i nove alate kojima se problemi efikasnije rešavaju.

Autor rada je imao priliku da radi u *Windows Performance* timu u toku razvijanja *Windows 8* operativnog sistema. Jedan od zadataka na kojima je autor radio je analiza potrošnje memorije u *Windows 8* aplikacija pisanih u JavaScript programskom jeziku. Prilikom inicijalnih analiza došlo se do zaključka da postojeći alati ne pružaju dovoljno informacija za analizu, te je sa njima jako teško rešavati probleme. Iz tog razloga su razvijeni novi alati, specijalizovani za rešavanje problema.

Proizvod autorovog rada u *Windows Performance* sistemu su izveštaji koji opisuju pronađene probleme u aplikacijama i korišćene alate, dizajn i razvijanje novog tipa alata, kao i rešavanje problema u aplikacijama koje su isporučene uz *Windows 8* operativni sistem. Izveštaji nisu dostupni javnosti (u pitanju su interna dokumenta *Microsoft-a*). Razvijeni alat je takođe dostupan samo interno, ali je veliki broj ideja preuzet prilikom razvijanja javno dostupnog alata ugrađenog u *Visual Studio.*

Sadržaj

1 Uvod 4
1.1 Važnost problema curenja memorije u operativnom sistemu Windows 8 5
1.1.1 Windows RunTime 6
1.1.2 Stanje povezane pripravnosti 9
1.1.3 *Windows web* aplikacije 10
1.1.4 Ciljani sistemi 12
1.1.5 Zaključak 12
2 Različiti pristupi rešavanju problema curenja memorije 12
2.1 Detekcija problema 13
2.1.1 *Event tracing for Windows* 14
2.2 Izolacija scenarija 16
2.3 Analiza problema 16
2.3.1 Neupravljani kôd 17
2.3.2 Brojanje referenci 17
2.3.3 Graf referenci 18
2.4 Rešavanje problema 19
3 Analiziranje problema curenja memorije u JavaScript aplikacijama 20
3.1 Iskustva stečena na razvijanju internog alata 20
3.1.1 Inicijalne opservacije stečene korišćenjem nespecijalizovanih alata 21
3.1.2 Pristup sa ispisivanjem grafa referenci 21
4 Primer analiziranja problema 27
4.1 Kretanje kroz *ListView* 28
4.2 Brisanje *ListView* kontrole 33
5 Zaključak 35
6 Dodatak 37
7 Literatura 42

1 Uvod

Postoji mnogo različitih definicija problema curenja memorije. Istorijski, u jezicima koji ne podržavaju automatsko upravljanje memorijom, taj problem se definiše kao gubitak pokazivača ka alociranoj memoriji. Na primer, ukoliko program alocira memoriju (na primer pozivom funkcije *malloc* u programskom jeziku C) i sačuva pokazivač ka alociranoj memoriji u lokalnoj promenljivoj funkcije koja vrši alokaciju, nakon povratka iz te funkcije više nije moguće vratiti memoriju hipu jer je za poziv funkcije *free* neophodan pokazivač ka bloku memorije koji je potrebno "osloboditi". U ovom primeru korišćen je hip, ali curenje memorije se može desiti i bez istog. Na isti način u *Windows* operativnom sistemu može procureti memorija uz pomoć funkcija *VirtualAlloc* [1] i *VirtualFree* [2].

Kada programski nedostatak dovodi do curenja dodeljene memorije (eng. *commited*) problem u jednom programu može negativno uticati na performanse celog sistema. Kada se memorija dodeli nekom programu upravljač memorijom obezbeđuje pristup toj memoriji. Drugim rečima - tek kada se memorija dodeli programu taj program može pristupiti (čitati i pisati) alociranoj memoriji. Da bi se obezbedio pristup memoriji, upravljač memorijom u operativnom sistemu mora rezervisati stvarne resurse, to može biti prostor u fizičkoj memoriji ili prostor u datoteci sa stranicama.

Problemi koje će korisnik iskusiti ukoliko u nekom od programa curi memorija su identični problemima koje bi iskusio ukoliko računar ima premalo memorije za izvršavanje svih programa koje je korisnik pokrenuo. Ekstremni slučaj može dovesti do tzv. rastresanja (eng. *thrashing*) [3]. Ova situacija se dešava kada sistemu ponestane fizičke memorije. Da bi oslobodio fizičku memoriju sistem će stranice iz radnog skupa nekog programa ispisati na disk (u datoteku koja sadrži stranice). Sledeći put kada program sa potkresanim radnim skupom pokuša da pristupi potkresanoj memoriji, umesto čitanja podataka iz memorije, on će morati da ih čita sa sporijeg medija koji sadrži datoteku sa stranicama.

U tom slučaju performanse celog sistema mogu da budu degradirane zbog jednog problematičnog programa. S obzirom na to da se sva memorija, dodeljena jednom programu, oslobađa kada program završi izvršavanje, problem curenja memorije često može da prođe nedetektovan ukoliko je program kratkog života.

Dakle, da bi curenje memorije preraslo u ozbiljan problem koji negativno utiče na performanse celog sistema, moraju se ispuniti sledeći uslovi:

- Mora da curi dodeljena memorija. U slučaju curenja virtualnog adresnog prostora jedina žrtva je program koji je izazvao curenje.
    - S obzirom na to da je neophodno da se memorija dodeli programu kako bi isti mogao da joj pristupi, ovaj uslov je skoro uvek ispunjen.
- Količina iscurele memorije mora biti dovoljno velika da bi razne polise operativnog sistema bile izvršene.
    - Operativni sistem, kada programima dodeli više memorije nego što je očekivano, mora konzervativnije trošiti resurse za druge zadatke.
        - Na primer, može se smanjiti količina video memorije, iz memorije se mogu izbaciti keširani podaci sa diska, itd.
    - Kako sve ove polise uglavnom uzimaju u obzir količinu dostupne memorije, ovaj uslov zavisi od ukupne količine memorije računara na kome se program izvršava i od brzine kojom memorija curi.
- Program koji ima problem curenja memorije mora da bude dugog životnog veka.

1.1 Važnost problema curenja memorije u operativnom sistemu Windows 8

Kako bi se *Windows 8* operativni sistem bolje takmičio sa modernim operativnim sistemima (*iOS* i *Android*), razvijenim prevashodno za uređaje opremljenim ekranom osetljivim na dodir, napravljene su značajne prepravke u odnosu na *Windows 7*. One uključuju unapređivanje korisničkog interfejsa kako bi se poboljšalo iskustvo na uređajima opremljenim ekranom osetljivim na dodir, ugrađenom prodavnicom za preuzimanje aplikacija, podršku za sisteme koji koriste čipove ARM arhitekture, itd.

Među najbitnijim promenama je dodatak novog interfejsa za programiranje [4]. Glavni način programiranja za *Windows* operativne sisteme, do pojave *Windows 8* verzije, je bio uz pomoć *Win32* interfejsa za programiranje aplikacija. Postoji dosta dodatnih biblioteka napravljenih da bi programiranje postalo lakše (jedan primer su *Windows Forms* isporučen uz .NET), ali su sve one generalno patile od istih manjkavosti:

- Uz pomoć njih je teško napraviti korisnički interfejs koji je pogodan za ekrane osetljive na dodir.
    - Jedan od bitnih ciljeva *Windows 8* operativnog sistema je da bude odličan na tom tipu uređaja.
- Dozvoljavaju previše mogućnosti programima.
    - Jednostavan primer viška privilegija je da program može da se izvršava (drugim rečima da koristi sistemske resurse poput procesora, internet konekcije, diska) i dok korisnik nema želju za korišćenjem tog programa.
- Deljenje podataka između programa zahteva posebne interfejse koji nisu standardizovani.

Kako bi se te boljke rešile, uz *Windows 8* operativni sistem je isporučen novi interfejs za programiranje - WinRT (eng. *Windows RunTime*).

1.1.1 Windows RunTime

WinRT aplikacije je moguće pisati u nekoliko programskih jezika – C++, C# i JavaScript. Ovaj tip aplikacija se može izvršavati samo na računarima sa *Windows 8* ili novijim operativnim sistemom. Mogu se praviti u *Visual Studio* integrisanom razvojnom okruženju.

Sve funkcionalnosti WinRT interfejsa za programiranje aplikacija su dostupne u svakom od podržanih jezika. Neke od njih su:

- Standardizovane kontrole za korisnički interfejs.
- Funkcije za pristup raznim vrstama uređaja poput štampača, uređajima za određivanje lokacije, itd
- Interfejs za pisanje i čitanje datoteka.
- Funkcije za pristup mreži.

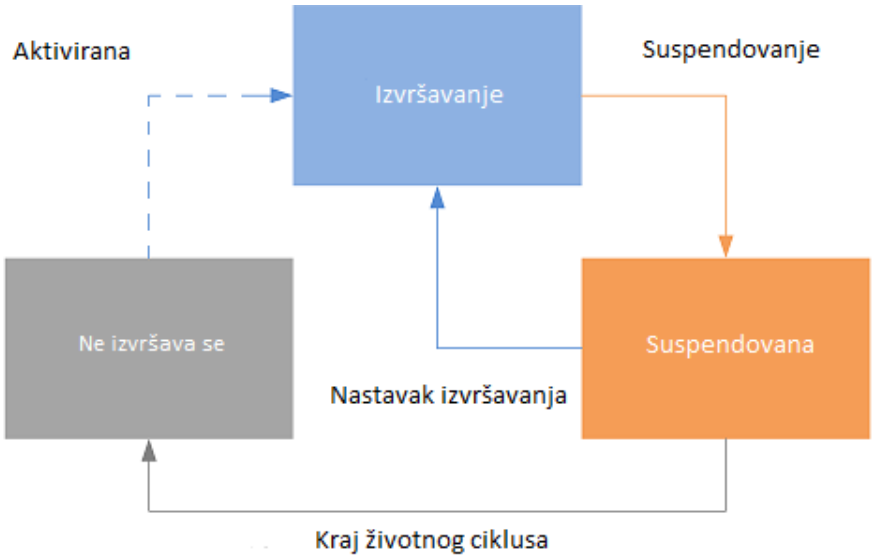
Jedna od glavnih odlika ovog tipa aplikacija je to što je njihova funkcionalnost ograničena radi obezbeđivanja veće sigurnosti korisnika. Tako, na primer, nije moguće alociranje memorije sa koje se može izvršavati kôd. Ta činjenica omogućava da se eliminiše velika klasa bezbednosnih problema koji omogućavaju da se, u okviru neke aplikacije, izvršava zlonamerni kôd.

Kako bi korisnički ulaz bio što brže procesiran WinRT interfejs za programiranje aplikacija prati asinhroni model programiranja [5]. Ovaj model programiranja omogućava da operacije koje zahtevaju mnogo vremena za izvršavanje (na primer čitanje podataka sa diska ili preuzimanje podataka sa mreže) ne blokiraju izvršavanje programa. Time se omogućava da korisnički interfejs ne bude blokiran prilikom dugih operacija.

1.1.1.1 Životni ciklus WinRT aplikacija

Kao što je već rečeno, problem curenja memorije teško može uticati na performanse sistema ukoliko je program koji omogućava curenje memorije kratkog veka - kad se program završi sva memorija koju je alocirao se vraća upravljaču memorijom. Performanse sistema se mogu ugroziti jedino ako se problematični program ne završava brzo.

Jedna od bitnih noviteta u WinRT modelu je životni ciklus aplikacija - regularno korišćenje aplikacija, pa i njihovo zatvaranje, ne prouzrokuje završetak programa. Slika 1 opisuje životni ciklus WinRT aplikacija.



Slika 1 - životni ciklus WinRT aplikacija[1]

[1] Slika preuzeta sa http://msdn.microsoft.com/en-us/library/windows/apps/hh464925.aspx

Na početku korisničke sesije nijedna aplikacija se ne izvršava (sve su u „Ne izvršava se“ stanju). Kada korisnik aktivira aplikaciju, na primer pritiskom na odgovarajuću pločicu u startnom ekranu, aplikacija prelazi u stanje izvršavanja. U tom stanju ona ima pristup sistemskim resursima i, između ostalog, može alocirati memoriju.

Kada korisnik prestane da koristi aplikaciju, ona postaje „Suspendovana“. U tom stanju se kôd aplikacije ne izvršava, ali aplikacija i dalje može biti u memoriji. Ostavljanje aplikacije u memoriji omogućava brzi povratak u stanje izvršavanja kada korisnik ponovo želi da je koristi.

Ovde je bitno primetiti da korisnik nema lako dostupnu opciju da prekine izvršavanje aplikacije. To je jako bitno jer korisnik, u većini slučajeva, nema razloga da ugasi aplikaciju (to jest da joj završi životni ciklus) osim da bi povratio resurse koje je aplikacija alocirala. Obzirom na to da u suspendovanom stanju aplikacija ne može alocirati nove resurse, u svakom trenutku je poznato koliko ona „košta“ sistem. Tu činjenicu koristi PLM (eng. *Process Lifetime Manager*) kako bi na optimalan način upravljao životnim ciklusom aplikacija [6]. Ta komponenta operativnog sistema „zna“ koliko memorije svaka aplikacija košta i može da završi životni ciklus aplikacija kako bi se ta memorija vratila sistemu.

1.1.1.1.1 Process Lifetime Manager

PLM komponenta, kao što samo ime kaže, upravlja životnim ciklusom aplikacija. Pored upravljanja životnim ciklusom, neke od osnovnih odgovornosti ove komponente su da prati koliko resursa svaka aplikacija koristi i da osigura da aplikacije koje se trenutno ne koriste nemaju pristup resursima računara.

Ograničavanje pristupa resursima računara (poput procesorskog vremena ili mreže), iako ograničava sposobnosti aplikacija, omogućava efikasnije korišćenje baterije. Kontrolisanje potrošnje memorije, kroz završavanje životnog ciklusa aplikacija ili pražnjenje njihovog privatnog radnog skupa, je bitno kako bi aplikacije koje korisnik trenutno koristi imale dovoljno memorije.

Postupak pražnjenja privatnog radnog skupa se dešava u nekoliko koraka [7]:

- PLM primeti da sistemu ponestaje memorije i traži od upravljača memorijom da isprazni privatni radni skup neke od aplikacija koje su u „Suspendovana“ stanju.

- Upravljač memorijom prebacuje stranice iz aplikacijinog radnog skupa na listu modifikovanih stranica.
- Stranice sa modifikovane liste se asinhrono ispisuju na disk, u datoteku sa stranicama.
- Čak i kada se stranice uspešno ispišu na disk one ostaju u memoriji, ali se pomeraju na listu rezervnih stranica koje se u svakom trenutku mogu iskoristiti u neku drugu svrhu.

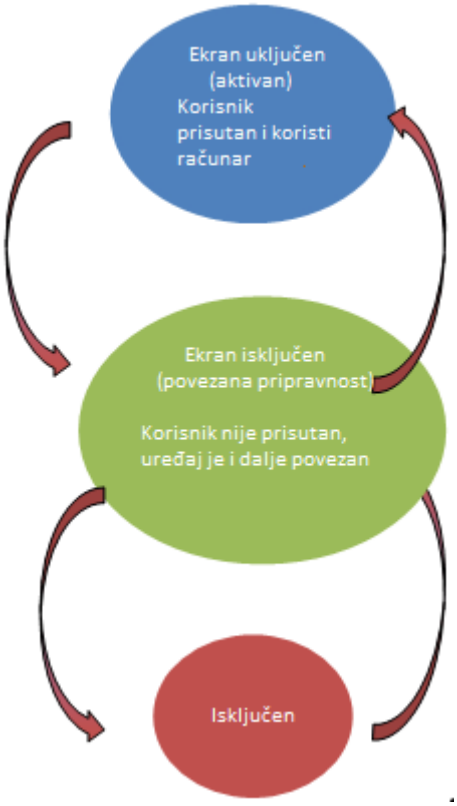
1.1.2 Stanje povezane pripravnosti

Jedan od bitnih faktora koje utiču na dužinu životnog ciklusa aplikacija je korisnička sesija. Kada se korisnik prijavi na računar stvara se nova korisnička sesija koja traje dokle god je korisnik prijavljen. Završetkom korisničke sesije svi programi koji su bili izvršavani u toj sesiji završavaju svoj životni ciklus. Dakle, kratke korisničke sesije bi mogle da učine da problem curenja memorije u aplikacijama bude zanemarljiv u odnosu na opšte performanse sistema jer bi životni ciklus aplikacija u tom slučaju takođe bio kratak.

Obzirom na to da se korisnici retko eksplicitno odjavljuju sa računara, glavni razlozi završetka korisničke sesije su restartovanje i gašenje računara.

Da bi se poboljšalo korisničko iskustvo, poželjno je da korisničke sesije traju što duže jer svako ponovno pokretanje računara zahteva da se svi neophodni programi ponovo pokrenu. U tom slučaju korisnik mora da sačeka određeno vreme pre nego što može koristiti računar, odnosno aplikaciju koju želi da koristi.

Windows 8 operativni sistem je objavljen sa podrškom za stanje povezane pripravnosti (eng. *connected standby*). Stanje povezane pripravnosti omogućava sistemu da, kada se izvršava na hardveru sa odgovarajućom podrškom, spusti potrošnju energije (drugim rečima - produži život baterije), a da se korisnička sesija ne završi. Takav sistem omogućava izvesne scenarije koji inače ne bi bili mogući. Jedan od primera je mogućnost primanja poziva dok je računar u tzv. stanju niske potrošnje (eng. *low power state*).



Slika 2 - dijagram stanja napajanja[2]

Iz ugla upravljanja memorijom, stanje povezane pripravnosti produžava korisničke sesije i samim tim postavlja sistem u položaj u kome je podložniji problemu curenja memorije.

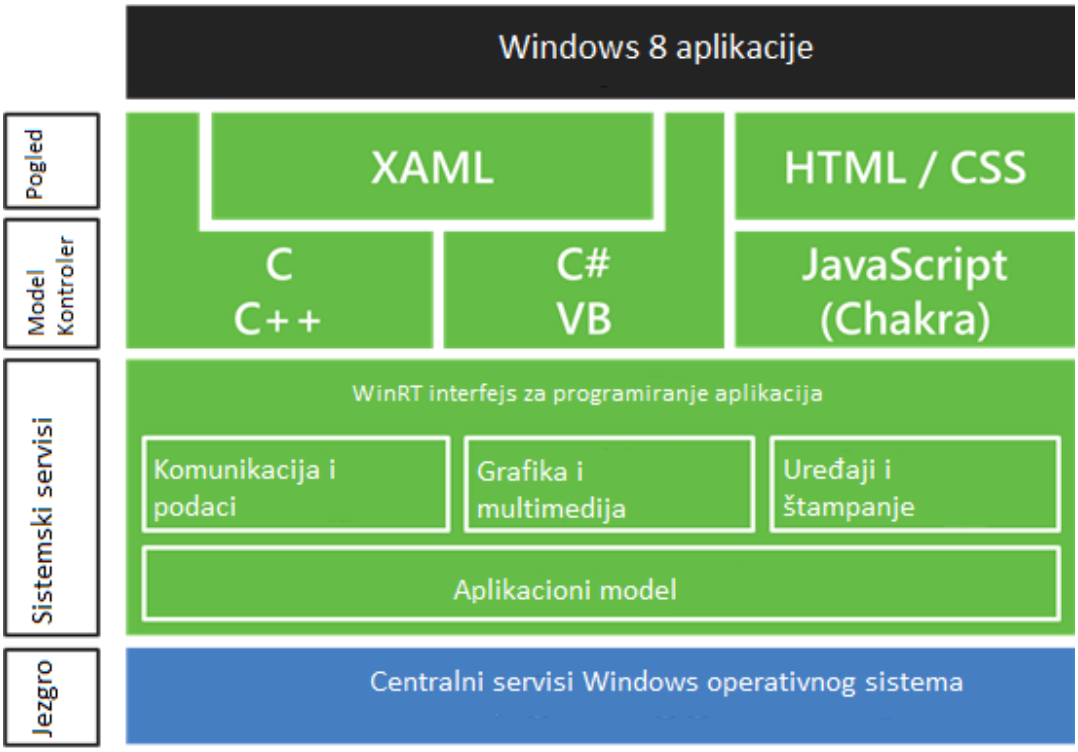
1.1.3 *Windows web* aplikacije

Postoji nekoliko jezika u kojima je moguće pisati WinRT aplikacije. Jedan od njih je JavaScript. Karakteristike ovog programskog jezika su opisane u knjizi „*Javascript: The Definitive Guide*“ [8]. Obzirom na to da je jedan od najrasprostranjenijih jezika [9] koji se koriste za programiranje klijentskih aplikacija, sasvim je logičan izbor dozvoliti programerima da ga koriste za pisanje aplikacija prvog reda.

Slika 3 prikazuje programerski model za WinRT aplikacije. *Windows web* aplikacije, ili skraćeno WWA, se prave korišćenjem obrasca dizajna Model-Pogled-Kontroler (MVC) [10],

[2] Slika preuzeta sa https://software.intel.com/en-us/articles/windows-8-store-app-connected-standby-whitepaper

pri čemu HTML/CSS predstavlja pogled, a JavaScript se koristi za implementaciju modela i kontrolera.



Slika 3 - programerski model za WinRT applikacije[3]

Jedna od ključnih odlika JavaScript aplikacija na mreži, iz ugla upravljanja memorijom, je činjenica da se navigacijom cela stranica osvežava. To dovodi do toga da se memorija koja je bila alocirana od strane JavaScipt aplikacije vraća sistemu.

Na žalost, navigacija dovodi do neoptimalnog korisničkog iskustva. Ona zahteva da se nova stranica ponovo iscrtava što se, iz ugla korisnika, manifestuje nestankom jedne stranice sa ekrana, pojavom potpuno belog ekrana i postepenim učitavanjem nove stranice. Da bi aplikacija imala optimalno korisničko iskustvo, ona nikad ne bi trebala da vrši navigaciju između stranica. Bolje rešenje je da se sadržaj modifikuje na jednoj stranici dinamičkim učitavanjem novog sadržaja uz pomoć JavaScript-a.

[3] Slika preuzeta sa http://blogs.msdn.com/b/b8/archive/2012/02/09/building-windows-for-the-arm-processor-architecture.aspx

1.1.4 Ciljani sistemi

Iz ugla proizvođača operativnog sistema postoji želja da operativni sistem može da se izvršava na što jeftinijem hardveru. Ta sposobnost omogućava spuštanje cene uređaja ili povećanje cene operativnog sistema.

Jedan od načina da se spusti cena hardvera je ugradnja manje količine radne memorije. Dakle potrebno je da se operativni sistem, i sve aplikacije koje korisnik pokreće, imaju dobre performanse i u sistemima sa malo radne memorije.

1.1.5 Zaključak

Nabrojano je nekoliko razloga zbog kojih je bitno rešiti problem curenja memorije u JavaScript aplikacijama:

- JavaScript aplikacije sada imaju povećani značaj.
    - Dosta aplikacija isporučenih uz *Windows 8* su napisane u JavaScript-u.
- Životni ciklus aplikacija je dugačak.
    - PLM se trudi da život aplikacije u toku jedne korisničke sesije bude što duži.
    - Stanje povezane pripravnosti značajno produžava prosečno trajanje korisničke sesije.
- Postoji želja da se isporučuju uređaji sa što manje radne memorije.

Ti razlozi su naveli *MicroSoft* da uloži u alate koji pomažu programerima da reše probleme curenja memorije u svojim aplikacijama.

2 Različiti pristupi rešavanju problema curenja memorije

Rešavanje problema curenja memorije se može podeliti u nekoliko koraka:

1. Detekcija problema
2. Izolacija scenarija
3. Analiza problema
4. Popravljanje problema

2.1 Detekcija problema

Prvi korak, prilikom rešavanja problema curenja memorije. je zapravo uočavanje postojanja problema. Na žalost, načini za detekciju problema nisu dobro definisani - najčešći način primećivanja problema je kroz posmatranje potrošnje memorije programa. Ukoliko program kroz svoj životni vek troši sve više i više memorije, verovatno je da negde u programu memorija curi.

Problem detekcije može se posmatrati na dva načina - problem se može detektovati interno, prilikom razvoja, odnosno testiranja programa, ili interno/eksterno prilikom korišćenja programa.

Prilikom testiranja programa, bilo ručnog ili automatizovanog, može se posmatrati njegova potrošnja memorije. Ukoliko se primeti da je potrošnja memorije, kroz životni ciklus programa, u porastu, sa značajnom sigurnošću se može tvrditi da program pati od problema curenja memorije. Prilikom inicijalne detekcije teško je utvrditi da li program pati od curenja memorije ili nekog drugog problema vezanog za potrošnju memorije poput fragmentacije.

Ukoliko problem detekcije posmatramo iz ugla korisnika sistema (program se često koristi i interno u razvojnom ciklusu, što se u žargonu naziva *dogfooding* [11]), problem će se primetiti kada sistem uspori zbog nedostatka memorije. U tom trenutku korisnik može upotrebiti nekakav alat (na primer upravljač zadacima) kako bi pronašao koji program, ili grupa programa, koristi previše memorije. Ukoliko korisnik tada primeti da izvesni program koristi znatno više memorije nego što je očekivano, budi se sumnja da taj program možda pati od problema curenja memorije.

Alati koji se koriste za praćenje potrošnje memorije obično mogu prikazivati različite metrike vezane za potrošnju memorije:

- Radni skup informacija *W(t, r)* programa u trenutku *t* je kolekcija informacija referenciranih od strane programa u vremenskom intervalu *(t - r, t)* [12].
    - Postoji nekoliko problema sa upotrebom ove metrike prilikom utvrđivanja da li program pati od problema curenja memorije.

        - Radni skup sadrži i one stranice koje nisu svojstvene samo posmatranom procesu (deljene stranice) što može dovesti do pogrešnih zaključaka.
        - Upravljač memorijom može potkresati radni skup kada sistemu ponestane memorije i time sakriti problem.
- Dodeljena memorija ne može biti potkresana od strane upravljača memorijom i ne postoji konfuzija oko toga ko je zapravo vlasnik memorije.
    - Da bi curenje memorije prouzrokovalo pad u performansama sistema potrebno je da se curi dodeljena memorija.
        - Ukoliko se programu dodeljuje mnogo memorije, ali program dodeljenoj memoriji nikada ne pristupa, može se desiti da performanse sistema ne budu kompromitovane.

2.1.1 *Event tracing for Windows*

U *Windows* operativnom sistemu jedan od načina za analiziranje rada aplikacije je uz pomoć ETW (eng. *Event Tracing for Windows* [13]). To je sistem koji povezuje snabdevače događajima sa njihovim potrošačima. Događaji, koji mogu biti signalizirani i iz aplikacija trećih lica, se mogu smestiti u *log* ili konzumirati u realnom vremenu. Svaki događaj može zapisati proizvoljne informacije i uz svaki događaj ETW komponenta će uvek pružiti određene informacije. Tu spadaju i proces i nit na kojoj je događaj signaliziran, vreme signalizacije i stek na kome je događaj signaliziran.

2.1.1.1 Ugrađeni snabdevači

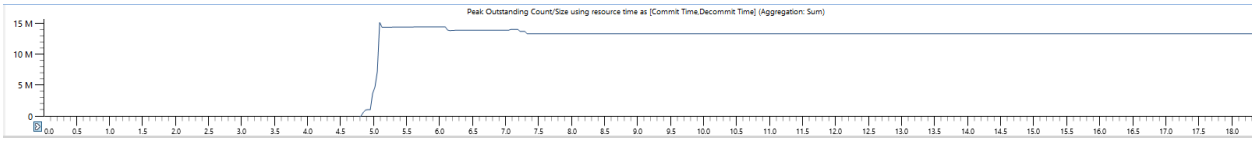
Jedan od načina za manipulisanje snabdevačima i logovanje događaja je uz pomoć „*xperf*" alata. Ovaj alat je moguće koristiti kako bi se pokrenulo prikupljanje događaja i kako bi se *log* snimio na disk.

Snabdevači događajima se mogu podeliti u nekoliko grupa. Nama zanimljivi su snabdevači koji rade u kernel modu (puna lista se može dobiti izvršavanjem komande "xperf -providers K") i ostali instalirani snabdevači (puna lista se može dobiti sa "xperf -providers I").

Postoji nekoliko snabdevača ugrađenih u *Windows 8* operativni sistem koji se mogu koristiti za analiziranje potrošnje memorije u aplikaciji. Mi ćemo se, za sada, fokusirati na dva: VIRT_ALLOC i Heap.

2.1.1.1.1 VIRT_ALLOC

VIRT_ALLOC snabdevač događajima je ugrađen u upravljač memorijom. Na svaki poziv funkcija *VirtualAlloc* i *VirtualFree* ovaj snabdevač može da ispiše informacije o pozivu (adresu alocirane memorije, tipa alokacije, veličinu alokacije, itd.). Pozivi ovih funkcija su uglavnom skriveni iza alokatora koji su nešto zgodniji za korišćenje (kao na primer hip).



Slika 4 - količina dodeljene virtualne memorije prilikom pokretanja test aplikacije napravljene za demonstraciju alata i detaljnije opisane u poglavlju 4

Process	Commit Stack	Size Sum	Count Sum
WWAHost.exe...	[Root]	17,092,608	758
	|- ntdll.dll!RtlUserThreadStart	16,912,384	731
	| kernel32.dll!BaseThreadInitThunk	16,912,384	731
	| |- SHCore.dll!Microsoft::WRL::FtmBase::MarshalInterface	8,794,112	420
	| | |- twinapi.appcore.dll!Microsoft::WRL::Details::RuntimeClass<Microsoft::WRL::Details::InterfaceLis...	8,757,248	415
	| | | |- twinapi.appcore.dll!Windows::ApplicationModel::Core::CoreApplicationView::Run	8,151,040	363
	| | | | WWAHost.exe!WebInstance::Run	8,151,040	363
	| | | | WWAHost.exe!CoreWindowDispatcher::RunMessageLoop	8,151,040	363
	| | | | Windows.UI.dll!Windows::UI::Core::CDispatcher::ProcessEvents	8,151,040	363
	| | | | Windows.UI.dll!Windows::UI::Core::CDispatcher::WaitAndProcessMessages	8,151,040	363
	| | | | |- user32.dll!DispatchMessageWorker	8,126,464	362
	| | | | | |- user32.dll!UserCallWinProcCheckWow	8,101,888	360
	| | | | | | |- mshtml.dll!GlobalWndProc	8,028,160	354
	| | | | | | | mshtml.dll!GlobalWndOnMethodCall	8,028,160	354
	| | | | | | | |- mshtml.dll!CPostManager::PostManOnTimer	5,455,872	240
	| | | | | | | | mshtml.dll!PostManExecute	5,455,872	240
	| | | | | | | | mshtml.dll!CHtmPost::Run	5,455,872	240
	| | | | | | | | mshtml.dll!CHtmPost::Exec	5,455,872	240
	| | | | | | | | |- mshtml.dll!CHtmParseBase::Execute	5,402,624	234
	| | | | | | | | | mshtml.dll!CHtmScriptParseCtx::Execute	5,402,624	234
	| | | | | | | | | mshtml.dll!CScriptData::Execute	5,402,624	234
	| | | | | | | | | mshtml.dll!CScriptData::CommitCode	5,402,624	234
	| | | | | | | | | mshtml.dll!CScriptCollection::ParseScriptText	5,402,624	234
	| | | | | | | | | mshtml.dll!CJScript9Holder::ParseScriptText	5,402,624	234
	| | | | | | | | | |- mshtml.dll!CJScript9Holder::ExecuteIntermediateCode	5,160,960	225
	| | | | | | | | | | jscript9.dll!ScriptEngine::ExecuteByteCodeBuffer	5,160,960	225
	| | | | | | | | | | jscript9.dll!ScriptEngine::ParseScriptTextCore	5,160,960	225
	| | | | | | | | | | |- jscript9.dll!ScriptEngine::ExecutePendingScripts	4,304,896	213
	| | | | | | | | | | | jscript9.dll!ScriptSite::Execute	4,304,896	213
	| | | | | | | | | | | jscript9.dll!ScriptSite::CallRootFunction	4,304,896	213
	| | | | | | | | | | | jscript9.dll!Js::JavascriptFunction::CallRootFunction	4,304,896	213
	| | | | | | | | | | | jscript9.dll!Js::JavascriptFunction::CallFunction<1>	4,304,896	213
	| | | | | | | | | | | jscript9.dll!amd64_CallFunction	4,304,896	213

Slika 5 - isečak steka koji alocira memoriju

2.1.1.1.2 Heap

Pozivi funkcija *malloc* i *free* u programskom jeziku C su, u *Windows* operativnom sistemu preusmereni na funkcije *HeapAlloc* [14] i *HeapFree* [15] implementirane u kernel32.dll biblioteci. Te funkcije su takođe instrumentovane kako bi moglo da se prati koliko je određena aplikacija alocirala memorije i sa kojim razlogom. Instrumentacija uključuje veličinu alokacije, adresu na kojoj je memorija alocirana, tip hipa na kojem je memorija alocirana [16].

2.2 Izolacija scenarija

Nakon uspešne detekcije problema potrebno je izolovati scenario u kojem dolazi do curenja memorije. Izolacija scenarija se najčešće izvodi sistemom pokušaja i pogrešaka. Osoba koja testira program isprobava razne, što jednostavnije, scenarije tokom kojih se očekuje da potrošnja memorije ostane na konstantnom nivou. Scenariji su uglavnom definisani nekom akcijom poput navigacije, korišćenjem izvesne funkcionalnosti i sl.

Najbolji pristup izolaciji scenarija je kroz automatizaciju. Nakon definisanja potencijalno sumnjivih scenarija, mogu se koristiti alati za automatizaciju (na primer kroz simulaciju korisničkog ulaza) koji mnogo puta ponavljaju isti scenario.

U ovom koraku se, sa znatno većom sigurnošću, može utvrditi da li program zaista pati od problema curenja memorije ili ne. U slučaju da prilikom izvršavanja programa zaista curi memorija očekuje se da, prilikom ponavljanja istog scenarija, potrošnja memorije raste linearno. Ukoliko rast potrošnje memorije posle izvesnog broja ponavljanja scenarija uspori, moguće je da je problem zapravo u fragmentaciji memorije ili, u slučaju upravljanog kôda, u neoptimalnom algoritmu koji upravlja sakupljačem otpadaka (eng. *garbage collector*).

2.3 Analiza problema

Kada se uspešno izoluje problematični scenario, može se preći na analizu istog. Analizi problema curenja memorije se može pristupiti na nekoliko različitih načina. Odabrani način zavisi od tipa programa i načina upravljanjem memorijom u istom.

2.3.1 Neupravljani kôd

Kod neupravljanog kôda za alokaciju memorije se koriste interfejsi koji uglavnom sadrže samo funkcije tipa *alloc* i *free*. Funkcija *alloc* vraća informaciju (pokazivač) koja se može koristiti za pristup memoriji i koja je neophodna za poziv funkcije *free*. Klasičan primer te vrste interfejsa su funkcije *malloc* i *free* u standardnoj biblioteci programskog jezika C.

Za ovaj tip programa je svojstveno da, ukoliko se koriste dobri programerski obrasci, uvek postoji tačno jedan vlasnik memorijske lokacije. Na taj način se obezbeđuje da je, u svakom trenutku u izvršavanju programa, jasno ko je odgovoran da alociranu memoriju vrati sistemu.

Prilikom analize problema curenja memorije kod ovog tipa programa jedina informacija, pored scenarija u kome dolazi do curenja memorije, koja je potrebna da bi se problem mogao rešiti je to koja je alokacija procurela. U slučaju da se uvek zna ko je vlasnik, rešavanje problema je trivijalno - vlasnik treba da vrati memoriju sistemu u odgovarajućem trenutku. Ukoliko vlasnik nije jednoznačno određen, mora se ustanoviti ko je odgovoran da oslobodi memoriju.

Jedan od načina pronalaženja lokacija koje nisu vraćene sistemu u *Windows* operativnim sistemima je uz pomoć ETW. Prikupljanjem logova odgovarajućeg alokatora (na primer *Heap*), kao i stekova prilikom poziva funkcija za alokaciju, se može utvrditi koja alokacija nije oslobođena.

2.3.2 Brojanje referenci

U slučaju da se u programu memorijom upravlja metodom brojanja referenci (svojstveno za C++ programski jezik, ali takođe lako izvodljivo i u C programskom jeziku), ne postoji jednoznačno definisan vlasnik memorijske alokacije - u svakom trenutku može postojati mnogo različitih vlasnika jedne alokacije. Objekti koji pristupaju memorijskoj lokaciji pozivaju funkciju *addref* koja inkrementuje trenutni broj vlasnika. Kada objekat završi sa korišćenjem lokacije, očekuje se da pozove funkciju *release* koja će dekrementovati broj vlasnika. Ukoliko broj vlasnika dođe do 0, memorijska lokacija se vraća sistemu.

Kod ovakvog pristupa upravljanju memorijom postoji par čestih obrazaca koji dovode do curenja memorije:

1. Nedostatak poziva funkcije *release* kako bi se vratila referenca.
2. Cirkularne reference između nekoliko objekata.

U oba slučaja je potrebno ustanoviti ko se nije odrekao svoje reference ka objektu za koji se očekuje da bude oslobođen. Da bi se procurela referenca pronašla, prvo je potrebno utvrditi koji objekat je potrebno osloboditi. To se može uraditi na isti način na koji se prate alokacije u programu u kome ne postoji automatsko upravljanje memorijom.

Nakon što se ustanovi koja alokacija je procurela, program se može pokrenuti sa zakačenim alatom za istrebljivanje problema. Prilikom alokacije objekta potrebno je postaviti *breakpoint* koji će prekinuti izvršavanje programa prilikom svake modifikacije promenljive koja sadrži broj referenci ka alokaciji. Praćenjem ko je uzeo i ko je vratio referencu može se zaključiti ko svoju referencu nije vratio i samim tim pronaći krivac.

2.3.3 Graf referenci

Noviji način automatskog upravljanja memorijom u okviru jednog programa je uz pomoć grafa referenci. Ovakav način upravljanja memorijom je svojstven C#, JavaScript i drugim programskim jezicima. U tom sistemu svaki alocirani objekat se može predstaviti čvorom u grafu. Kada objekat A uzme referencu ka objektu B, u graf se dodaje usmerena grana (A, B). U grafu postoji makar jedan "koreni" čvor (čvor koji je uvek prisutan u grafu i ka kome niko nema granu).

Kada podgraf X grafa referenci G postane nedostižan iz svih korenih čvorova, podgraf X se može eleminisati i objekti koji su predstavljeni čvorovima u podgrafu X se mogu osloboditi. Ovakvo potkresivanje grafa se dešava prilikom rada sakupljača otpadaka.

Kod ovakvog sistema upravljanja memorijom i dalje je teoretski moguće koristiti alocirajući stek (ukoliko alokator može da pruži tu informaciju) kako bi se identifikovao objekat, odnosno alokacija na kojoj se objekat nalazi. Za razliku od prethodna dva sistema, kada se identifikuje alokacija koja je procurela, nije jednostavno zaključiti zašto alokacija nije oslobođena, odnosno zašto je objekat još uvek u memoriji.

Sakupljač otpadaka dodaje još jednu dimenziju problemu. U principu, nije moguće direktno upravljati sakupljačem otpadaka i dati mu instrukciju da počne da sakuplja. Ukoliko nije poznato da li je sakupljač otpadaka oslobodio svu moguću memoriju, nije jasno da li je

objekat i dalje u memoriji zato što neko i dalje nije poništio svoju referencu ka datom objektu ili zato što sakupljač otpadaka još uvek nije obišao ceo graf.

2.3.3.1 Implementacija sakupljača otpadaka

Sakupljač otpadaka u *Windows Web* aplikacijama je implementiran uz pomoć obeležavanja i brisanja [17]. Algoritam radi u nekoliko koraka:

1. Prvo se svi objekti obeleže.
2. Nakon toga se vrši obilazak grafa referenci i sa svakog objekta se skida obeležje.
3. Svi objekti koji su i dalje obeleženi se mogu osloboditi. To je bezbedno uraditi jer, do datih objekata, ne postoji putanja ni iz jednog korenog čvora.

Samo pokretanje rada sakupljača objekata se kontroliše raznim polisama koje u obzir uzimaju broj alociranih objekata, aktivnost aplikacijinog JavaScript kôda, i druge faktore.

Performanse sakupljača otpadaka su dalje unapređene posebnim načinom za procesiranje listova u grafu [18]. Određeni tipovi JavaScript objekata (na primer brojevi i niske karaktera) ne mogu imati reference ka drugim objektima. Ta činjenica omogućava efikasniju implementaciju sakupljača otpadaka jer se ti objekti ne moraju obeležavati. Izvršavanje sakupljača otpadaka paralelno sa glavnom niti JavaScript aplikacije je takođe nešto jednostavnije jer alokacije listova ne zahtevaju ponovno obilaženje grafa referenci.

2.4 Rešavanje problema

Kada se završi analiza problema i identifikuje koja alokacija je procurela i zbog čega, može se preći na rešavanje istog. Ovaj korak, isto kao analiza problema, je zavistan od načina upravljanja memorijom u programu koji ispoljava problem.

1. U slučaju neupravljanog kôda, potrebno je da se na odgovarajućem mestu u kôdu pozove funkcija koja oslobađa memoriju.
2. U slučaju upravljanog kôda, bilo da se koristi sistem brojanja referenci ili graf referenci, problem se popravlja tako što svi objekti koji drže referencu ka procureloj alokaciji poništavaju svoje reference.

3 Analiziranje problema curenja memorije u JavaScript aplikacijama

JavaScript aplikacije i HTML za prikazivanje korisničkog interfejsa su visoko apstraktni jezici. Oni se uglavnom ne kompajliraju unapred nego se izvršavaju u okviru izvesne vrste virtualnih mašina. Postoje razne implementacije JavaScript i HTML *engine*-a. *MicroSoft*-ova implementacija JavaScript-a se zasniva na *Chakra*, a HTML na *Trident* bibliotekama. U ovom radu biće razmotreni problemi u samim aplikacijama, a ne u bibliotekama korišćenim za interpretaciju aplikacija.

Bitno je primetiti da aplikacija koja pati od problema curenja memorije ima problem nezavisno od virtualne mašine na kojoj se izvršava. Kada postoje reference ka objektu, nijedna virtualna mašina ne može osloboditi memoriju jer se ne može zaključiti da li će aplikacija u budućnosti pristupiti datom objektu. Kako problemi postoje nezavisno od platforme na kojoj se aplikacija izvršava, moguće je koristiti alate dostupne na jednoj platformi i time rešiti problem na svim platformama.

U *Chakra* JavaScript *engine-u* memorijom aplikacije se upravlja uz pomoć sistema sa grafom referenci. Kada se sakupljač otpadaka pokrene, obilaze se čvorovi grafa i oni koji su nedostižni se oslobađaju. Iako taj pristup upravljanju memorije nije nov, on je za *Windows 8* operativni sistem postao jako značajan zbog novog tipa aplikacija koje su podržane u sistemu. Problem je manje ozbiljan kada se aplikacija izvršava u internet pretraživaču jer navigacija sa stranice (koja koristi mnogo memorije) oslobađa svu memoriju. Neke od internet aplikacija, koje inače ne vrše navigaciju, koriste ovu činjenicu kako bi zaobišle problem. Na svakih nekoliko interakcija, umesto dinamičkog učitavanja sadržaja, aplikacija može da odluči da izvrši navigaciju koja će za uzvrat zagarantovano osloboditi svu memoriju.

3.1 Iskustva stečena na razvijanju internog alata

U toku razvijanja operativnog sistema, u okviru *Windows Performance* tima, primećeno je da *Windows Web* aplikacije koje se isporučuju uz sistem koriste znatno više memorije nego što je bilo očekivano [19]. Ova opservacija je dovela do značajnih ulaganja u razvijanje internih alata koji se mogu koristiti za analiziranje problema. Iskustva stečena prilikom razvijanja i korišćenja internog alata su kasnije korišćena prilikom razvijanja javno dostupnog alata ugrađenog u *Visual Studio* integrisano razvojno okruženje.

3.1.1 Inicijalne opservacije stečene korišćenjem nespecijalizovanih alata

Jako je teško analizirati problem curenja memorije u JavaScript aplikacijama bez alata specijalizovanih za taj problem. Uz pomoć VIRT_ALLOC ETW snabdevača, koji je sastavni deo *Windows* operativnog sistema, je ipak moguća neka vrsta analize. Razlog tome je činjenica da je JavaScript hip implementiran na pozivima funkcija *VirtualAlloc* i *VirtualFree* (moguće je implementirati hip bez korišćenja ovih funkcija, ali bi to bila izuzetno čudna implementacija - na primer neka datoteka bi se mogla mapirati u memoriju i onda koristiti za alokacije).

Uz pomoć VIRT_ALLOC snabdevača u *log* događaja se mogu zapisati informacije o svakom pozivu za alociranje virtualne memorije i samim tim se može videti i stek svake alokacije. Na žalost, alokacije JavaScript objekata su mnogo manje granularnosti od alokacija virtualne memorije - svaki alocirani blok virtualne memorije se isparča na sitnije komade i onda se koristi za alociranje pojedinih objekata. To znači da sa VIRT_ALLOC snabdevačem možemo da vidimo samo one alokacije JavaScript objekata koje su prouzrokovale rast JavaScript hipa. Zbog toga je jako teško ustanoviti koji objekti su procureli.

Čak i ako kojim slučajem uspemo da zaključimo koji objekti su procureli (to se na primer može izvesti veštačkim povećavanjem veličine objekata tako da što više alokacija prouzrokuje rast JavaScript hipa) jako je teško ustanoviti zašto su ti objekti i dalje u memoriji. Drugim rečima, pronalaženje ko i dalje drži referencu ka objektu na procureloj alokaciji nije nimalo lako. U nekim drugim jezicima bi bilo moguće korišćenje alata za istrebljivanje problema kako bi se videlo svako uzimanje/vraćanje reference ka objektu, ali u JavaScriptu to jednostavno nije moguće.

3.1.2 Pristup sa ispisivanjem grafa referenci

Već prilikom inicijalnih pokušaja analiziranja problema curenja memorije u JavaScript aplikacijama, zaključeno je da neće biti lako pronaći razlog zašto je neki skup objekata i dalje u memoriji. U početku se, nakon velikog truda uloženog u to kako bi se identifikovale procurele alokacije na jednom primeru, manuelno prolazilo kroz kôd aplikacije i pokušavalo da se pronađe ko se nije odrekao reference eliminacijom pojedinih putanja u kôdu. Iako je taj specifičan problem bio uspešno rešen, uloženo vreme u rešavanje jednog problema je

bilo mnogo veće nego što je moguće priuštiti (drugim rečima – nikako ne bi bilo dovoljno vremena kako bi se rešili svi problemi za koje se sumnjalo da postoje).

Pošto je zaključeno da korišćenje postojećih alata neće moći da pruži zahtevane rezultate, odlučeno je da je potrebno da se ispiše ceo graf referenci kako bi kasnije mogao da se analizira i odredi koji objekti su još uvek u memoriji i, još bitnije, zašto su još uvek u memoriji. U tu svrhu napravljen je novi ugrađeni ETW snabdevač – *Microsoft-IE-JSDumpHeap*.

Provider Name	Task Name	Opc...	Index (Field 1)	Count (Field 2)	Values (Field 3)
▼ Microsoft-IE-JSDumpHeap					
	▼ JSDumpHeapEnvelope				
		win:Start	{1; 1024; False; Tru...		
		win:Start	{1; 1024; False; Tru...		
		win:Stop	{0x00000000; 7144;...		
		win:Stop	{0x00000000; 8893;...		
	▼ JSDumpHeapBulkNode				
		win:Info	0	1922	[{0x000000619E630200; 512; 0x000000619E630200; 0; 3; 1; 404} : {0x00000061A1384...
		win:Info	1	1922	[{0x00000061A2D00E40; 96; 0x00000061A2D00E40; 411; 1; 1; 0} : {0x00000061A2D0...
		win:Info	2	1922	[{0x00000061A1216080; 64; 0x00000061A1216080; 411; 1; 3; 0} : {0x00000061A1216...
		win:Info	3	1378	[{0x00000061A136D060; 16; 0x00000061A136D060; 29; 65; 1; 1} : {0x00000061A136...
	▼ JSDumpHeapStringTable				
		win:Info	0	2276	[{GlobalObject} : {} : {NaN} : {Infinity} : {undefined} : {eval} : {parseInt} : {parseFlo...
		win:Info	1	1968	[{upload} : {emoji} : {twopage} : {leavechat} : {mailforward} : {clock} : {send} : {cr...
		win:Info	2	1360	[{GroupsContainer_setDomElements} : {GroupsContainer_removeElements} : {Gr...
	▼ JSDumpHeapBulkEdge				
		win:Info	0	4669	[{2; 3; 1; 0x000000619E63F0E0} : {2; 3; 1; 0x00000061A0A60840} : {3; 0; 2; 0x00619E1...
		win:Info	1	4669	[{3; 0; 2495; 0x00619E100000000D} : {3; 0; 2496; 0x00619E100000000E} : {3; 0; 2497;...
		win:Info	2	2787	[{3; 3; 4965; 0x00000061A11E28C0} : {3; 3; 4966; 0x00000061A11E2900} : {3; 3; 4967;...
	▼ JSDumpHeapBulkAttribute				
		win:Info	0	6537	[{6; 0x000000619E63E000} : {6; 0x000000619E635B80} : {6; 0x00000061A0A6CC00} :...
		win:Info	1	6537	[{6; 0x000000619E635B80} : {7; 0x0000000000001124} : {5; 0x00000061A1234640} : {...
		win:Info	2	167	[{6; 0x000000619E635B80} : {7; 0x00000000000015CD} : {5; 0x00000061A11AB700} :...

Slika 6 - podaci dobijeni uz pomoć Microsoft-IE-JSDumpHeap snabdevača

3.1.2.1 Vizuelizacija grafa

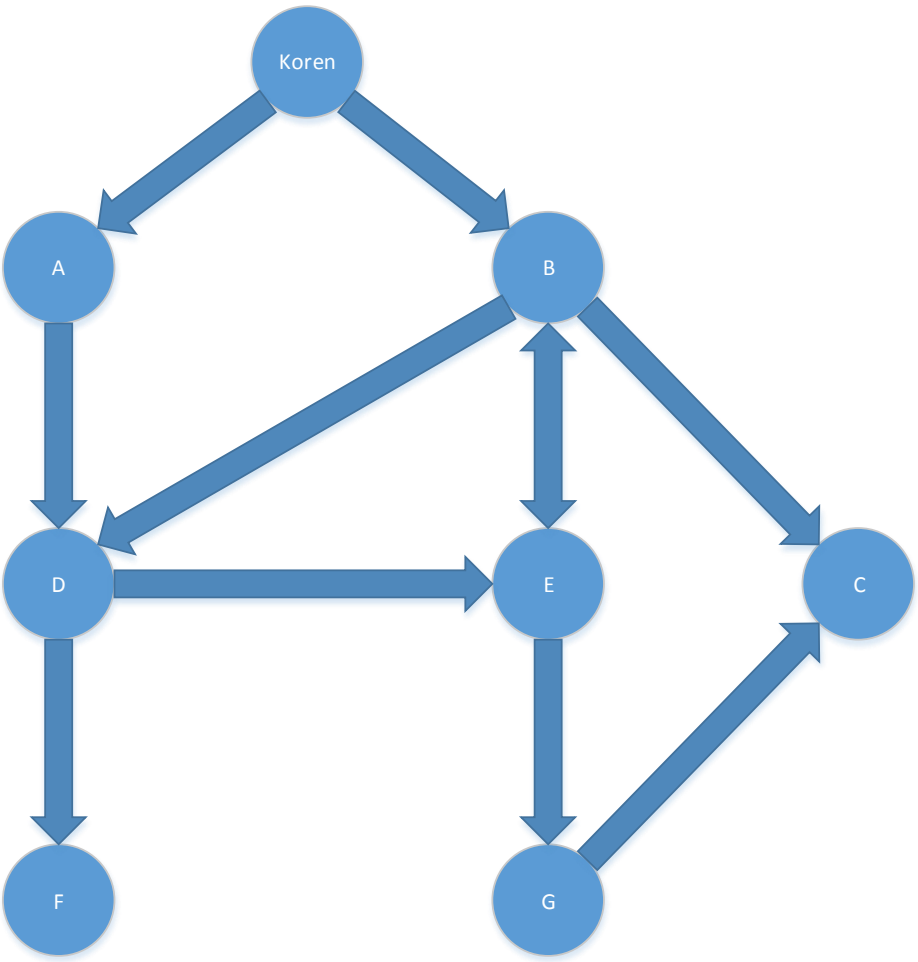
Nakon što se prikupe sve informacije iz grafa potrebno ih je prikazati tako da je jednostavno pretraživati i analizirati dobijeni graf. Kako su grafovi dosta nezgodni za vizuelizaciju, inicijalna ideja je utopiti graf u drvo. Algoritam za utapanje grafa u drvo se može opisati na sledeći način:

1. Dodati novi čvor i postaviti ga za koren drveta.
2. Sve čvorove ka kojima ne postoje grane postaviti kao direktne potomke korenog čvora.
3. Za svaki čvor D u drvetu kome odgovara čvor G u grafu:
    a. ukoliko je ovaj čvor G već obrađen, ne raditi ništa;
    b. obići sve grane koje izlaze iz čvora G kao potomke čvora D, ubaciti u drvo svaki čvor ka kome postoji direktna grana iz čvora G;

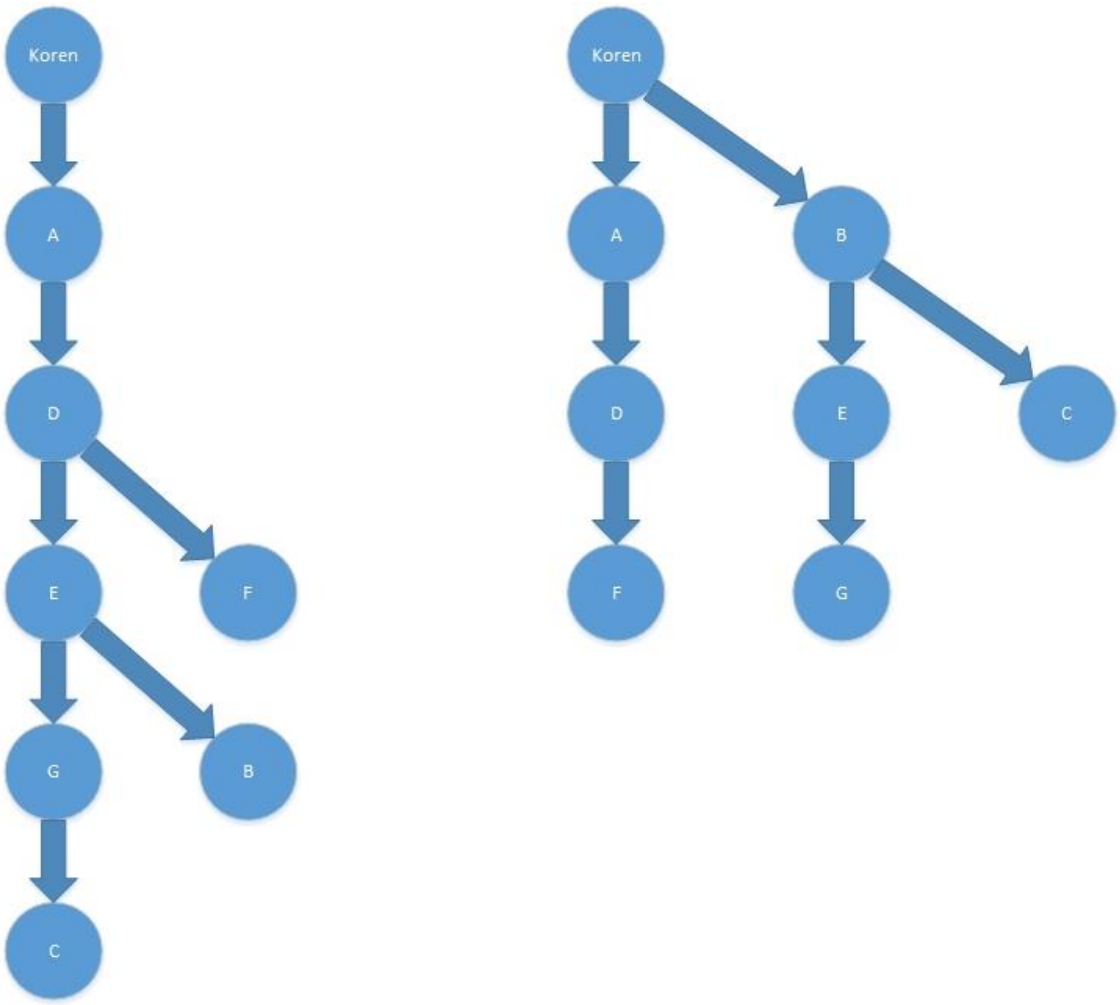
c. obeležiti čvor G kao obrađeni čvor.

Ovako dobijeno drvo se može analizirati na intuitivan način, jer je obilazak drveta jako sličan strukturi direktorijuma u sistemu datoteka.

Može se uočiti da različiti algoritmi za obilaženje grafa u koraku 3. proizvode različita drveta. Razmotrimo primer grafa (Slika 7).



Slika 7 - primer grafa



Slika 8 - drveta dobijena obilaskom u dubinu (levo) i u širinu (desno)

Slika 8 prikazuje drveta dobijena obilascima u dubinu i u širinu. Drvo napravljeno obilaženjem grafa u širinu ima manju visinu, nego drvo napravljeno obilaskom u dubinu, jer taj algoritam pronalazi najkraće puteve od korenog čvora ka svim ostalim čvorovima u usmerenom netežinskom grafu [20]. Iako to ne deluje kao značajna razlika na malom grafu koji je korišćen kao primer, na grafovima koji sadrže hiljade čvorova pravi izbor obilaska grafa je ključan.

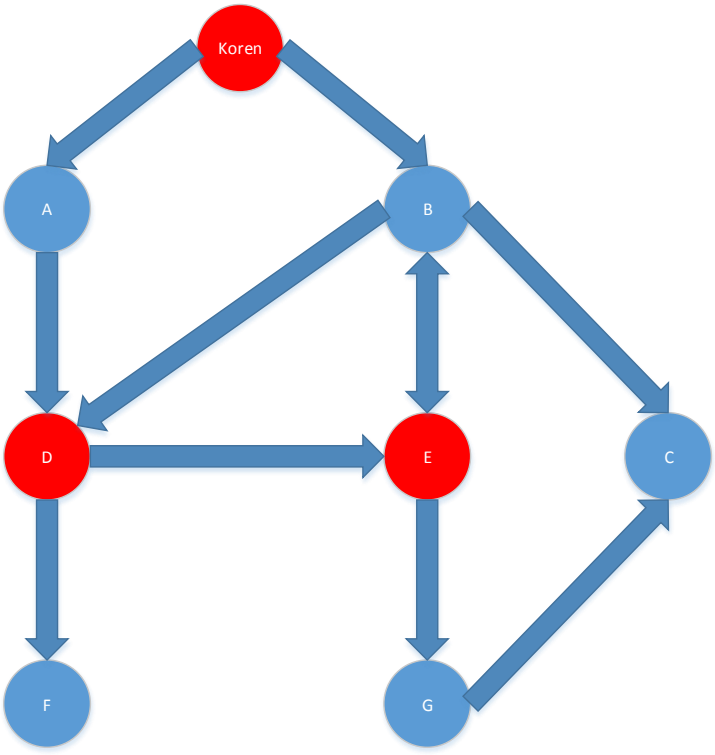
Kako bi se dalje uprostilo drvo (to jest kako bi dobijeno drvo bilo lakše za analizu), obilazak se može dalje dirigovati. Na primer, može da se bira redosled u kojem se obilaze grane čvora. Dobar izbor (recimo na osnovu tipa čvora) će napraviti drvo koje je lakše analizirati.

3.1.2.1.1 Primena dominatorskog drveta

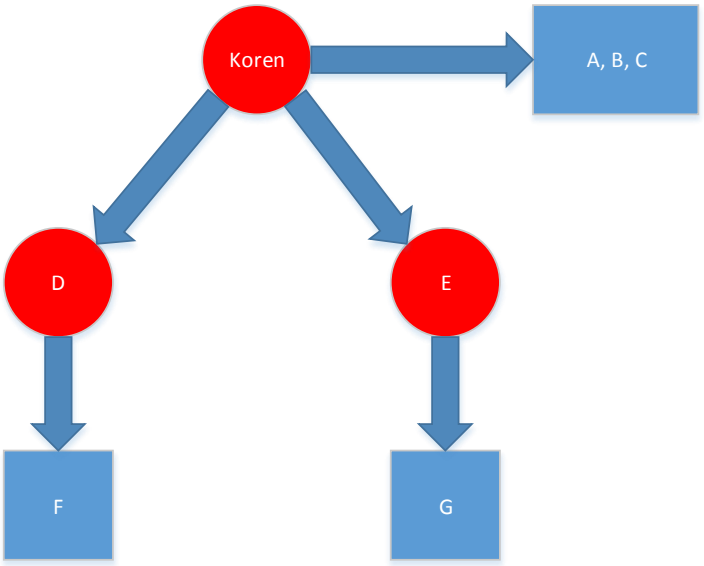
Iako obilazak grafa u širinu generiše drvo koje je relativno lako razumeti, možemo ga dalje uprostiti korišćenjem dominatora – specijalnih čvorova drveta koje ćemo sada definisati.

- Čvor D dominira čvorom Č (i naziva se dominatorom čvora Č) ukoliko svi putevi iz korenog čvora do čvora Č moraju da prođu kroz čvor D.
    - U našem primeru, koreni čvor je veštački dodati čvor čiji su direktni potomci čvorovi grafa ka kojima ne postoje grane u grafu referenci.
- Čvor D strogo dominira čvorom Č ukoliko D dominira Č i D nije jednako Č.
- Neposredni dominator čvora Č je jedinstveni čvor koji strogo dominira nad Č, ali ne dominira strogo ni nad jednim drugim čvorom koji strogo dominira nad čvorom Č.
- Dominatorsko drvo je drvo gde su potomci svakog čvora D oni čvorovi kojim čvor D neposredno dominira.

Slika 9 prikazuje primer grafa u kome su dominatori obeleženi crvenom bojom. Slika 10 prikazuje dominatorsko drvo dobijeno obradom grafa. Može se primetiti da dominatorsko drvo dodatno uprošćava prikaz grafa.



Slika 9 - primer grafa sa dominatorima



Slika 10 - dominatorsko drvo

3.1.2.2 Identifikacija objekata

Kada se graf referenci analizira, potrebno je da se identifikuju objekti iz grafa. Bez adekvatnih metoda za identifikaciju objekata sam graf referenci (odnosno drvo generisano obilaskom grafa) ne mogu pomoći pri analiziranju problema jer, iako je moguće napraviti opservacije o tome da li problem curenja memorije postoji, ne bi bilo moguće reći koji tačno objekti nisu oslobođeni te ne bi bilo moguće preći na korak u kojem se problem zapravo rešava. Dakle, da bi alat bio potpun neophodno je da pruži dovoljno informacija o objektima kako bi oni mogli da se pronađu u kôdu.

Alat koji je razvijen u svrhu rešavanja ovog zadatka pruža dve ključne informacije koje se mogu koristiti za identifikaciju objekta:

1. Imena objekata.
    a. Bitno je primetiti da imena objekata ne odgovaraju čvorovima u grafu već granama – jedan objekat može imati više referenci ka sebi i svaka od njih može imati različito ime.
2. Tip objekata.
    a. Iako je JavaScript jezik koji ne koristi strogi sistem tipova, tipovi su i dalje korisni jer će prikazati i HTML tip (na primer *Div*, *Image*, itd.) što može u mnogome pomoći pri identifikaciji objekta.

3.1.2.3 Veličina objekata

Kada se razmatra veličina objekta u grafu referenci, postoji nekoliko interesantnih opservacija:

1. Količina memorije koja je zadržana samim objektom (to jest čvorom u grafu referenci) je definisana samom veličinom čvora.
2. Veličina podgrafa grafa referenci jednaka je sumi potrošnje memorije čvorova u podgrafu.
    a. Podgraf se može definisati kao graf koji je dostupan iz nekog čvora.
3. Eliminacija čvora iz grafa će platformi vratiti onoliko memorije koliko je dostupno isključivo kroz eliminisani čvor. Ukoliko se eliminiše čvor, sistemu se vraća memorija koja je zauzeta svim čvorovima kojima eliminisani čvor dominira. Ovu meru nazivamo zadržanom veličinom (eng. r*etained size*)

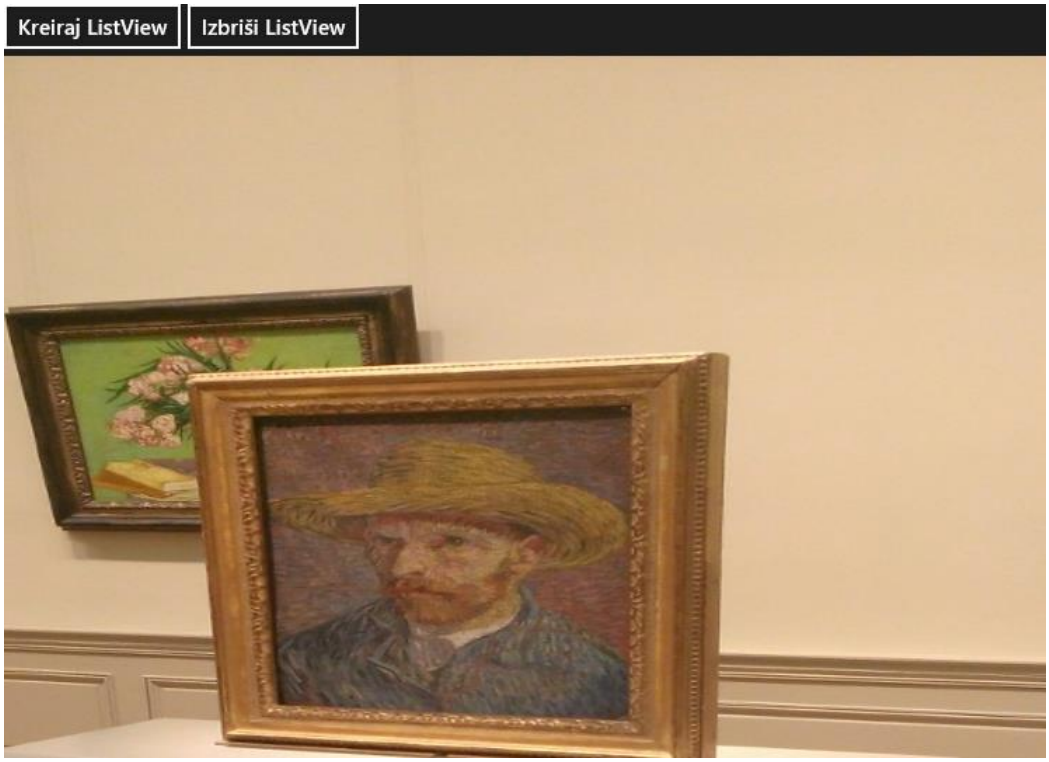
Prilikom analize potrošnje memorije u JavaScript aplikaciji veličina čvora i zadržana veličina su daleko korisnije mere nego što je veličina podgrafa.

Veličina podgrafa nije zanimljiva kao mera jer ne govori ništa o tome koliko memorije se može osloboditi eliminacijom čvora kojim je definisan podgraf. Takođe, lako je doći u situaciju u kojoj objekat A, uz pomoć reference ka objektu blizu nekom od korenih čvorova, ima jako veliki podgraf. U tom slučaju, prilikom analize, veličina podgrafa može dovesti do konfuzije o tome koliko je objekat A značajan za memorijsku potrošnju u aplikaciji.

4 Primer analiziranja problema

U svrhu prikazivanja alata koji dalje konzumira podatke iz *Microsoft-IE-JSDumpHeap* snabdevača napisana je aplikacija koja pati od problema curenja memorije. Koristiće se alat ugrađen u *Visual Studio* kako bi se identifikovao razlog za curenje memorije i popravio problem.

Aplikacija je vrlo jednostavna – može da kreira *ListView* kontrolu koja prikazuje sve slike iz korisnikove biblioteke sa slikama. Slika 11 prikazuje izgled test aplikacije. Aplikacija sadrži dva dugmeta za kreiranje i brisanje *ListView* kontrole i samu *ListView* kontrolu koja prikazuje slike iz korisnikove biblioteke.



Slika 11 - test aplikacija

4.1 Kretanje kroz *ListView*

Kako bi se *ListView* kontrola popunila slikama, razvijen je izvor podataka (eng. *data source*) koji omogućava pristup slikama. Ta kontrola je dovoljno pametna da u memoriji drži podatke samo za artikle (odnosno slike u test aplikaciji) koji su neophodni za trenutno prikazivanje i za performantno korisničko iskustvo prilikom kretanja kroz kontrolu.

Kada *ListView* kontroli zatrebaju nove slike za prikazivanje (prilikom inicijalizacije ili dok se korisnik kreće kroz listu) ona će od izvora podataka zatražiti neophodne podatke kako bi mogla da prikaže nove slike.

Za sada ćemo se fokusirati na dve, ključne, metode – *initialize* i *itemsFromIndex.*

```
initialize: function () {
    var queryOptions = Windows.Storage.Search.QueryOptions(
        Windows.Storage.Search.CommonFileQuery.orderByDate,
        [".jpg", ".jpeg", ".png"]);
    var pictures = Windows.Storage.KnownFolders.picturesLibrary;
    var query = pictures.createFileQueryWithOptions(queryOptions);
    var that = this;
    var queryResult = query.getFilesAsync();

    queryResult.then(function (items) {
```

```
        that.queryResults = items;
        that.picturesArray = new Array(items.length);
    });

    return queryResult;
}
```

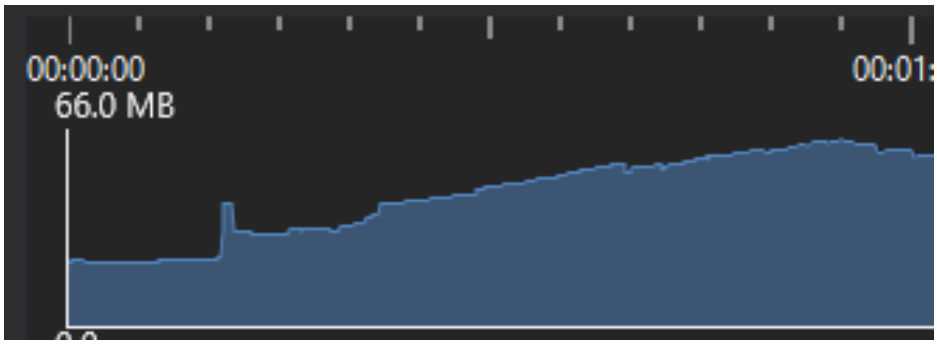
Funkcija *Initialize* vrši upit ka sistemu datoteka da bi dobila podatke o slikama koje se nalaze u korisnikovoj biblioteci sa slikama. Vratiće „obećanje“ (eng. *promise*) [21] objekat koji je deo WinJS biblioteke za pisanje *Windows Web* aplikacija. Kad se obećanje izvrši omogućava se korišćenje inicijalizovanog izvora podataka.

```
itemsFromIndex: function (index, countBefore, countAfter) {
    var first = (index - countBefore),
        count = (countBefore + 1 + countAfter);
    var that = this;
    var items = new Array();

    for (var i = first; i < first + count; i++) {
        if (this.queryResults.length > i) {
            if ((this.picturesArray[i] == null)) {
                var imgURL =
                    URL.createObjectURL(this.queryResults[i]);
                var img = document.createElement("img");
                img.src = imgURL;
                img.style.height = "1024px";
                img.style.width = "1024px";

                that.picturesArray[i] = { key: imgURL, data: img };
            }

            items.push(this.picturesArray[i]);
        }
    }

    return WinJS.Promise.wrap({
        items: items,
        offset: countBefore,
        absoluteIndex: index,
    });
}
```

Funkcija *itemsFromIndex* je deo interfejsa *IListDataAdapter* [22] i biće pozvana svaki put kada je *ListView* kontroli potrebno još elemenata za prikazivanje. U ovoj funkciji se, na

osnovu ulaznih parametara, izračunava koje elemente treba da vrati i obezbeđuje pristup istima kroz povratnu vrednost.

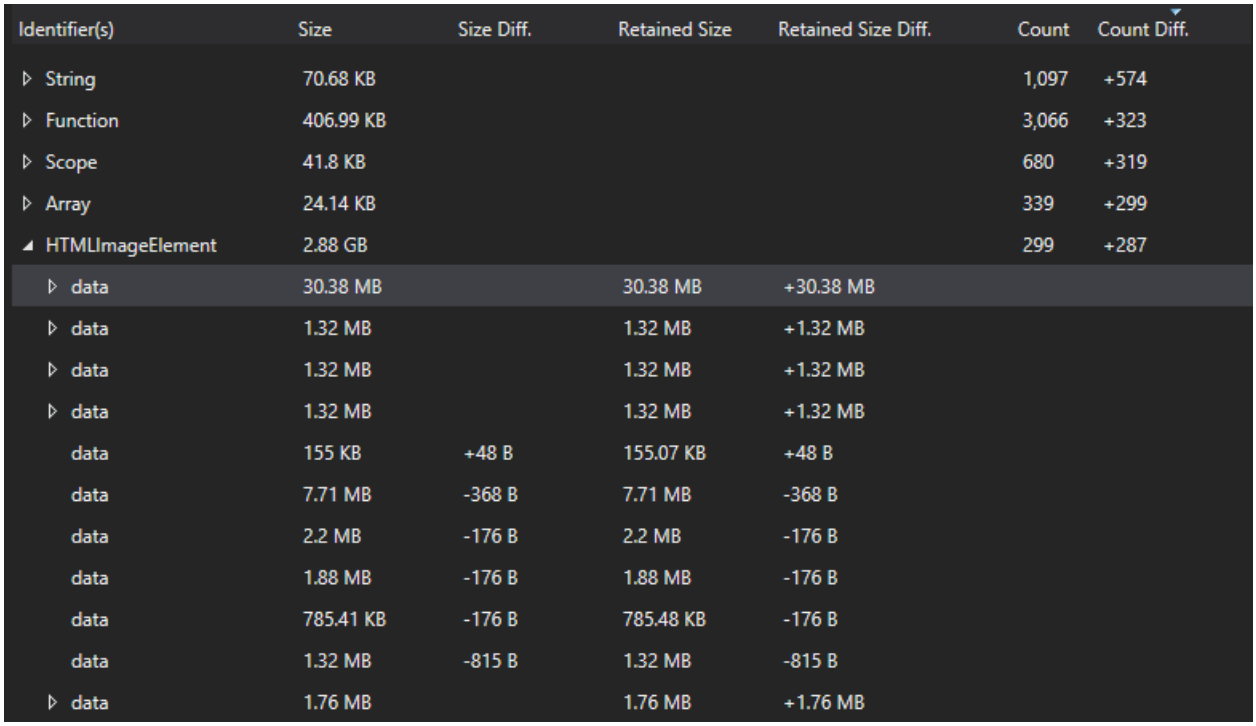
Posmatrajmo upotrebu memorije u test aplikaciji kroz alat ugrađen u *Visual Studio*. Slika 12 prikazuje količinu memorije korišćenu od strane JavaScript hipa tokom vremena. Inicijalno aplikacija ne radi ništa zanimljivo te se upotreba memorije ne menja značajno. Kada se *ListView* kreira možemo da se uoči nagli skok u potrošnji memorije, nakon koga je verovatno sakupljač otpadaka pokupio mnogo privremenih alokacija i omogućio da se izvesna količina memorije vrati sistemu.



Slika 12 - veličina JavaScript hipa prilikom kretanja kroz ListView

Nakon kreiranja *ListView* kontrole korišćenje memorije je, očekivano, stabilno. Očekivano je, takođe, da korišćenje memorije bude stabilno i dok se korisnik kreće kroz *ListView*. Ovo se omogućava tako što *ListView* u svakom trenutku sadrži samo nekoliko stranica u memoriji. Ostali elementi se oslobađaju i memorija se vraća sistemu. Slika 12 prikazuje da to nije slučaj. Dok se korisnik kreće kroz listu možemo primetiti da potrošnja memorije raste. Na par mesta može se videti blago opadanje korišćenja memorije koje je prouzrokovano sakupljačem otpadaka, ali se memorija nikad ne vraća na inicijalni nivo.

Očigledno je da postoji problem prilikom kretanja kroz *ListView* kontrolu. Može se pristupiti rešavanju ovog problema ispisivanjem grafa referenci pre nego što korisnik započne kretanje i još jednim ispisivanjem nakon što se korisnik pomerio kroz listu.

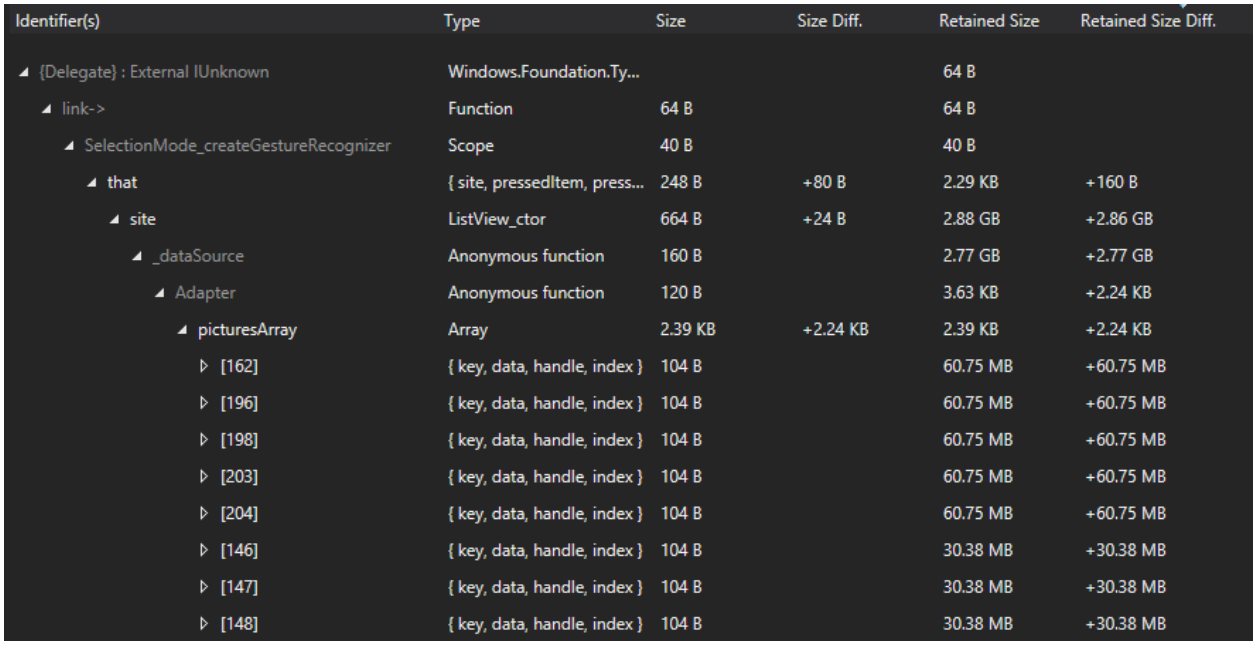


Identifier(s)	Size	Size Diff.	Retained Size	Retained Size Diff.	Count	Count Diff.
String	70.68 KB				1,097	+574
Function	406.99 KB				3,066	+323
Scope	41.8 KB				680	+319
Array	24.14 KB				339	+299
HTMLImageElement	2.88 GB				299	+287
data	30.38 MB		30.38 MB	+30.38 MB		
data	1.32 MB		1.32 MB	+1.32 MB		
data	1.32 MB		1.32 MB	+1.32 MB		
data	1.32 MB		1.32 MB	+1.32 MB		
data	155 KB	+48 B	155.07 KB	+48 B		
data	7.71 MB	-368 B	7.71 MB	-368 B		
data	2.2 MB	-176 B	2.2 MB	-176 B		
data	1.88 MB	-176 B	1.88 MB	-176 B		
data	785.41 KB	-176 B	785.48 KB	-176 B		
data	1.32 MB	-815 B	1.32 MB	-815 B		
data	1.76 MB		1.76 MB	+1.76 MB		

Slika 13 – razlika u elementima između grafova kreiranih pre i posle kretanja kroz ListView

Slika 13 prikazuje razliku između dva grafa referenci (prvog uzetog nakon kreiranja *ListView* kontrole i drugog uzetog nakon kretanja kroz istu). Možemo primetiti da ima 287 novih HTMLImageElement objekata. Svaki od njih zahteva da se zadrži i dekodirana slika što prouzrokuje novih 2.88 GB upotrebljene memorije (približno 10 MB po slici, dakle slike prosečno imaju oko 2.6 miliona piksela što deluje razumno jer *FullHD* slike imaju približno 2 miliona piksela).

Da bi se shvatilo zašto ove slike nisu izbrisane iz memorije treba pogledati zašto su zadržane u memoriji. Slika 14 prikazuje putanju do iscurelih slika u grafu referenci. Prikazana putanja je najkraća putanja od jednog od korenih čvorova do *picturesArray* čvora. Iako to možda i nije najintuitivnija putanja (intuitivnije putanje bi sadržale manje objekata iz *frameworka*, a više objekata definisanih od strane samog programa), može se zaključiti u čemu je problem. Implementirani izvor podataka zadržava reference ka slikama iako one više nisu zahtevane od strane *ListView* kontrole.



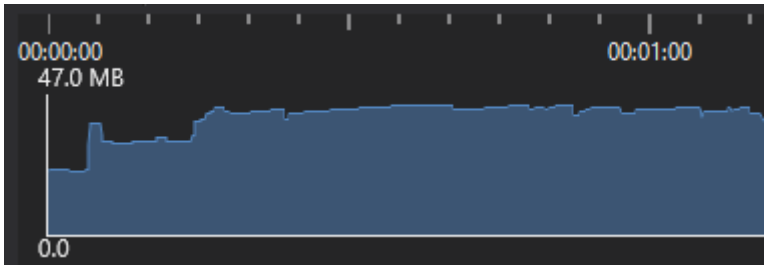
Identifier(s)	Type	Size	Size Diff.	Retained Size	Retained Size Diff.
{Delegate} : External IUnknown	Windows.Foundation.Ty...			64 B	
link->	Function	64 B		64 B	
SelectionMode_createGestureRecognizer	Scope	40 B		40 B	
that	{ site, pressedItem, press...	248 B	+80 B	2.29 KB	+160 B
site	ListView_ctor	664 B	+24 B	2.88 GB	+2.86 GB
_dataSource	Anonymous function	160 B		2.77 GB	+2.77 GB
Adapter	Anonymous function	120 B		3.63 KB	+2.24 KB
picturesArray	Array	2.39 KB	+2.24 KB	2.39 KB	+2.24 KB
[162]	{ key, data, handle, index }	104 B		60.75 MB	+60.75 MB
[196]	{ key, data, handle, index }	104 B		60.75 MB	+60.75 MB
[198]	{ key, data, handle, index }	104 B		60.75 MB	+60.75 MB
[203]	{ key, data, handle, index }	104 B		60.75 MB	+60.75 MB
[204]	{ key, data, handle, index }	104 B		60.75 MB	+60.75 MB
[146]	{ key, data, handle, index }	104 B		30.38 MB	+30.38 MB
[147]	{ key, data, handle, index }	104 B		30.38 MB	+30.38 MB
[148]	{ key, data, handle, index }	104 B		30.38 MB	+30.38 MB

Slika 14 - putanja do iscurelih slika

Dakle, rešenje problema je da se slike ne zadržavaju u izvoru podataka. Preporučeno rešenje je da izvor podataka čuva podatke o svim datotekama, a da se slike po potrebi učitavaju u funkciji *render* koja je definisana u kontroli *ListView*. Kako bi se problem curenja memorije eliminisao, funkcija *itemsFromIndex* je prepravljena na sledeći način:

```
itemsFromIndex: function (index, countBefore, countAfter) {
    var first = (index - countBefore),
        count = (countBefore + 1 + countAfter);
    var that = this;
    var items = new Array();

    for (var i = first; i < first + count; i++) {

        if (this.queryResults.length > i) {

            items.push({ data: this.queryResults[i],
                         key: this.queryResults[i].path });
        }
    }

    return WinJS.Promise.wrap({
        items: items,
        offset: countBefore,
        absoluteIndex: index,
    });
}
```

Nakon promena, da se slike ne prave i ne čuvaju u izvoru podataka već da o njima brine isključivo *ListView* kontrola, možemo videti da količina upotrebljene memorije više ne raste dok se korisnik kreće kroz listu (Slika 15).



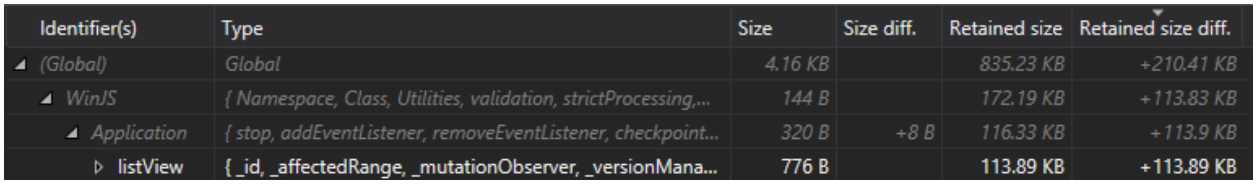
Slika 15 – veličina JavaScript hipa prilikom kretanja kroz Listview nakon popravljanja problema curenja memorije

4.2 Brisanje *ListView* kontrole

Još jedna od bitnih korisničkih interakcija je brisanje *ListView* kontrole. Kada se kontrola obriše, očekujemo da se sva memorija korišćena za kontrolu vrati sistemu. Kreiranje i brisanje liste ostvareno je sledećim kôdom:

```
app.ButtonListViewCreateOnClick = function () {

    var dataSource = new Data.MyDataSource();
    dataSource.initialize().done(function () {

        var listViewContainer = document.createElement("div");
        listViewContainer.style.height = "1024px";

        var options = {
            itemDataSource: dataSource,
            itemTemplate: app.StorageRenderer,
            layout: new WinJS.UI.GridLayout(),
            selectionMode: "none"
        };

        app.listView = new WinJS.UI.ListView(listViewContainer,
                                             options);

        document.getElementById("DivListViewContainer").
                 appendChild(listViewContainer);
    });
}
app.ButtonListViewDeleteOnClick = function () {
    var container;
    container = document.getElementById("DivListViewContainer");
    container.innerHTML = "";
}
```

Prilikom analiziranja problema moglo se uvideti da aplikacija zadržava referencu na listView promenljivu (Slika 16). Zajedno sa tom promenljivom zadržani su i svi objekti koje je listView referencirao, između ostalog i DOM elementi koje listView koristi za iscrtavanje.



Identifier(s)	Type	Size	Size diff.	Retained size	Retained size diff.
(Global)	Global	4.16 KB		835.23 KB	+210.41 KB
WinJS	{ Namespace, Class, Utilities, validation, strictProcessing,...	144 B		172.19 KB	+113.83 KB
Application	{ stop, addEventListener, removeEventListener, checkpoint...	320 B	+8 B	116.33 KB	+113.9 KB
listView	{ _id, _affectedRange, _mutationObserver, _versionMana...	776 B		113.89 KB	+113.89 KB

Slika 16 - putanja do ListView objekta

Prepravljena funkcija za brisanje liste izgleda ovako:

```
app.ButtonListViewDeleteOnClick = function () {
    var container;
    container = document.getElementById("DivListViewContainer");
    container.innerHTML = "";
    app.listView.dispose();
    app.listView = null;
}
```

Na ovaj način se postiže da, prilikom brisanja, bude obrisana referenca na kontrolu.

5 Zaključak

Iz ugla operativnog sistema problemi curenja memorije mogu biti pogubni po performanse. Jedna aplikacija koja pati od problema curenja memorije može znatno uticati na korisničko iskustvo.

- Upravljač memorijom može odlučiti da potkreše radne skupove drugih programa kako bi oslobodio fizičku memoriju i alocirao je aplikaciji koja pati od problema curenja memorije.
  Upravljač memorijom nema dovoljno informacija o kontekstu u kome se izvršava aplikacija koja zahteva memoriju, pa ne može doneti odluku da li aplikacija ima problem ili realan zahtev za još memorije.
- PLM može odlučiti da završi životni ciklus WinRT aplikacija kako bi memorija dodeljene njima bila vraćena upravljaču memorijom.
  Isto kao i upravljač memorijom, ni PLM nema dovoljno informacija o tome zbog čega potencijalno problematična aplikacija zahteva memoriju.
  Završavanje životnog ciklusa ostalih aplikacija od njih pravi žrtve jer će korisnik iskusiti duže učitavanje pri povratku na žrtvovane aplikacije.

Nažalost, iz ugla vlasnika problematične aplikacije, najveći motiv za popravljanje problema je kako bi se aplikacija dobro ponašala u sistemu. Većina korisnika ne ume, niti ima želju, da koristi alate kako bi primetili i isključili problematičnu aplikaciju. Jedini negativni efekat na aplikaciju koja ispoljava problem, iz ugla korisnika aplikacije, je to što je aplikacija potencijalno sporija jer sakupljač otpadaka koristi više procesorskog vremena (što je uzrokovano činjenicom da graf referenci raste usled curenja memorije).

Najčešći i najozbiljniji problemi koji su primećeni prilikom analiziranja aplikacija su oni koji procure memoriju korišćenu za multimedijalni sadržaj. Ta vrsta objekata, iako podjednako verovatna za curenje kao i bilo koja druga vrsta objekata, zauzima daleko više memorije od ostalih vrsta. Takođe je primećeno da su problemi jako često prouzrokovani ne očekivanim zadržavanjem referenci zatvaranjima (eng. *closure*).

Pristup analizi problema sa ispisivanjem grafa referenci se može primeniti kod svih jezika koji koriste graf referenci za upravljanje memorijom. Ovo je ujedno najmoćniji način

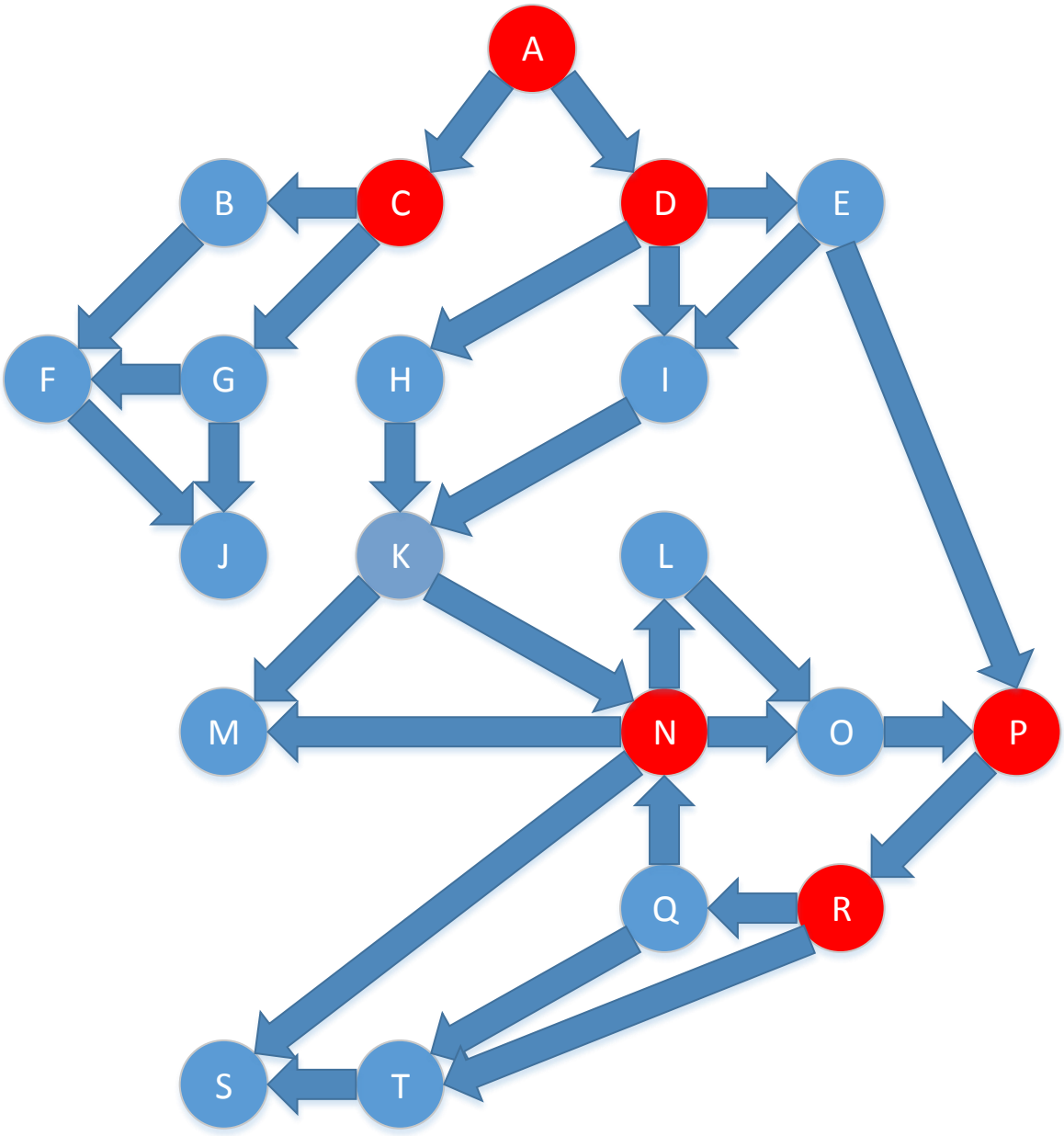
za analizu problema. Ispisivanje grafa referenci je mnogo bitnije ukoliko jezik, odnosno propratni alati, ne omogućavaju da se svako dodavanje/oduzimanje reference ka objektu loguje. Neki od popularnih primere takvih jezika su C# i Java.

6 Dodatak

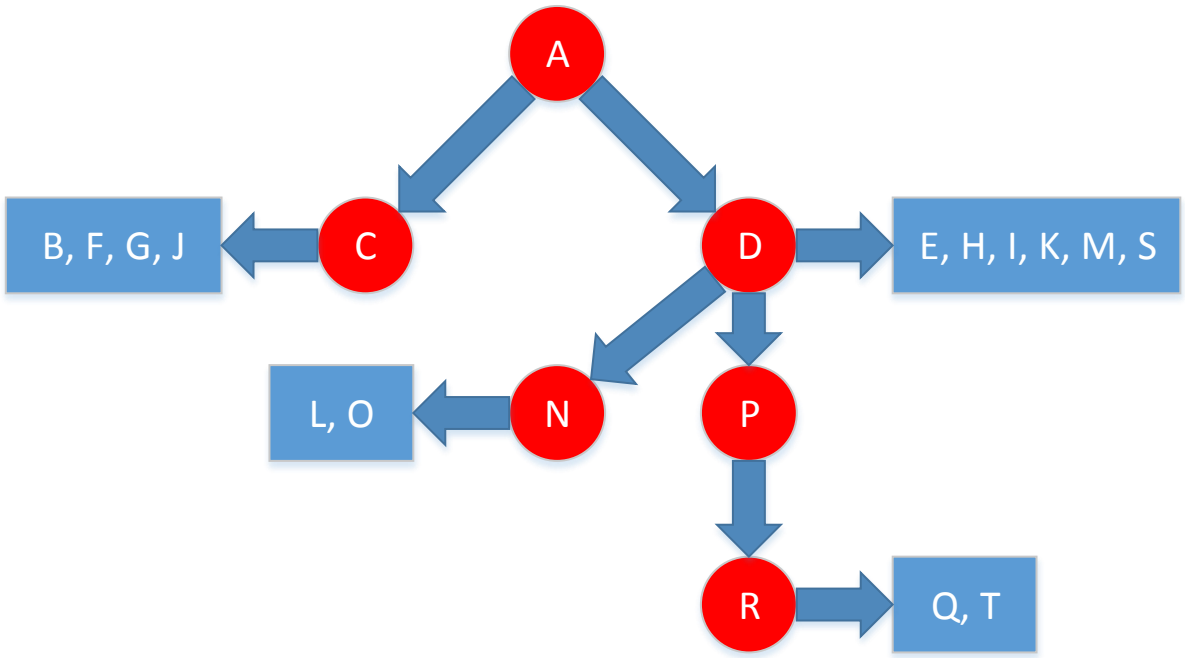
Alate za prikupljanje i analiziranje ETW logova (xperf i WPA) je moguće preuzeti sa http://www.microsoft.com/en-US/download/details.aspx?id=39982.

Visual Studio integrisano razvojno okruženje moguće je preuzeti sa http://www.visualstudio.com/en-us/downloads#d-express-windows-8.

Dodatni primer grafa i dominatorskog drveta dobijenog obradom grafa:



Slika 17 - dodatni primer grafa sa dominatorima



Slika 18 - dominatorsko drvo

Kôd test aplikacije sa *ListView* kontrolom:

```
(function () {
  "use strict";

  WinJS.Binding.optimizeBindingReferences = true;

  var app = WinJS.Application;
  var activation = Windows.ApplicationModel.Activation;

  app.onactivated = function (args) {
    if (args.detail.kind === activation.ActivationKind.launch) {
      if (args.detail.previousExecutionState !==
          activation.ApplicationExecutionState.terminated) {
        var create =
document.getElementById("ButtonListViewCreate");
        create.onclick = app.ButtonListViewCreateOnClick;
        var remove =
document.getElementById("ButtonListViewDelete");
        remove.onclick = app.ButtonListViewDeleteOnClick;
      } else {
        // TODO: This application has been reactivated from
        // suspension. Restore application state here.
      }
      args.setPromise(WinJS.UI.processAll());
    }
  };
```

```
app.oncheckpoint = function (args) {
  // TODO: This application is about to be suspended. Save any
  // state that needs to persist across suspensions here. You
  // might use the WinJS.Application.sessionState object, which
  // is automatically saved and restored across suspension. If you
  // need to complete an asynchronous operation before your
  // application is suspended, call args.setPromise().
};

app.ButtonListViewCreateOnClick = function () {

  var dataSource = new Data.MyDataSource();
  dataSource.initialize().done(function () {

    var listViewDiv = document.createElement("div");
    listViewDiv.style.height = "1024px";

    var listViewOptions = {
      itemDataSource: dataSource,
      itemTemplate: app.StorageRenderer,
      layout: new WinJS.UI.GridLayout(),
      selectionMode: "none"
    };

    app.listView = new WinJS.UI.ListView(listViewDiv,
                                         listViewOptions);

    document.getElementById("Container").appendChild(listViewDiv);
  });
};

app.ButtonListViewDeleteOnClick = function () {
  var container;
  container = document.getElementById("Container");
  container.innerHTML = "";
  app.listView.dispose();
  app.listView = null;
};

app.StorageRenderer = function (itemPromise, element) {
  var itemStatus;
  if (!element) {
    element = document.createElement("div");
    element.style.height = "1024px";
    element.style.width = "1024px";
  } else {
    element.innerHTML = "";
  }
```

```
    return {
      element: element,
      renderComplete: itemPromise.then(function (item) {
        return item.ready;
      }).then(function (item) {
        var imgURL = URL.createObjectURL(item.data,
                                         { oneTimeOnly: true });
        var img = document.createElement("img");
        img.src = imgURL;
        img.style.height = "1024px";
        img.style.width = "1024px";

        element.appendChild(img);

        return WinJS.Promise.wrap();
      })
    };
  }

  app.start();
})();

(function myDataSourceInit(global) {
  var MyDataAdapter = WinJS.Class.define(function () {
    this.compareByIdentity = false;
    this.picturesArray = new Array();
    this.queryResults = new Array();
    this.picturesCount;
  }, {

    itemsFromIndex: function (index, countBefore, countAfter) {
      var first = (index - countBefore),
          count = (countBefore + 1 + countAfter);
      var that = this;
      var items = new Array();

      for (var i = first; i < first + count; i++) {

        if (this.queryResults.length > i) {

          items.push({
            data: this.queryResults[i],
            key: this.queryResults[i].path
          });
        }
      }

      return WinJS.Promise.wrap({
        items: items,
```

```
        offset: countBefore,
        absoluteIndex: index,
      });
    },
    getCount: function () {
      return WinJS.Promise.wrap(this.queryResults.length - 1);
    },
    initialize: function () {
      var search = Windows.Storage.Search;
      var queryOptions = search.QueryOptions(
          search.CommonFileQuery.orderByDate,
          [".jpg", ".jpeg", ".png"]);
      var pictures = Windows.Storage.KnownFolders.picturesLibrary;
      var query = pictures.createFileQueryWithOptions(queryOptions);

      var that = this;
      var queryResult = query.getFilesAsync();

      queryResult.then(function (items) {
        that.queryResults = items;
        that.picturesArray = new Array(items.length);
      });

      return queryResult;
    }
  });

  WinJS.Namespace.define("Data", {
    MyDataSource: WinJS.Class.derive(
        WinJS.UI.VirtualizedDataSource,
        function () {

          this.Adapter = new MyDataAdapter();
          this._baseDataSourceConstructor(this.Adapter);
        },
    {
      initialize: function () {
        return this.Adapter.initialize();
      }
    })
  });
})();
```

7 Literatura

[1] Microsoft, „VirtualAlloc function,“ 24 8 2014. [Na mreži]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa366887%28v=vs.85%29.aspx.

[2] Microsoft, „VirtualFree function,“ 24 8 2014. [Na mreži]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa366892%28v=vs.85%29.aspx.

[3] P. J. Denning, „Thrashing: its causes and prevention,“ New York, 1968.

[4] B. Dewez, Getting Started with Windows 8 Apps, O'Reilly Media, 2012.

[5] J. Olson, „Keeping apps fast and fluid with asynchrony in the Windows Runtime,“ Microsoft, 20 3 2012. [Na mreži]. Available: http://blogs.msdn.com/b/windowsappdev/archive/2012/03/20/keeping-apps-fast-and-fluid-with-asynchrony-in-the-windows-runtime.aspx. [Poslednji pristup 31 8 2014].

[6] Microsoft, „Application lifecycle (Windows Runtime apps),“ Microsoft, [Na mreži]. Available: http://msdn.microsoft.com/en-us/library/windows/apps/hh464925.aspx. [Poslednji pristup 31 8 2014].

[7] B. Karagounis, „Reclaiming memory from Metro style apps,“ Microsoft, 17 4 2012. [Na mreži]. Available: http://blogs.msdn.com/b/b8/archive/2012/04/17/reclaiming-memory-from-metro-style-apps.aspx. [Poslednji pristup 24 8 2014].

[8] D. Flanagan, JavaScript: The Definitive Guide, O'Reilly Media, 2011.

[9] P. Carbonnelle, „PYPL PopularitY of Programming Language index,“ 19 8 2014. [Na mreži]. Available: https://sites.google.com/site/pydatalog/pypl/PyPL-PopularitY-of-Programming-Language. [Poslednji pristup 24 8 2014].

[10] G. E. Krasner / S. T. Pope, „A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80,“ *Journal of Object Oriented Programming,* t. 1, br. 3, pp. 26-49, 1988.

[11] Wikipedia, „Eating your own dog food,“ 17 7 2014. [Na mreži]. Available: http://en.wikipedia.org/wiki/Dogfooding. [Poslednji pristup 24 8 2014].

[12] P. J. Denning, „The working set model for program behavior,“ t. 11, br. 5, 1968.

[13] Microsoft, „About Event Tracing,“ Microsoft, [Na mreži]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa363668%28v=vs.85%29.aspx. [Poslednji pristup 24 8 2014].

[14] Microsoft, „HeapAlloc function,“ Microsoft, [Na mreži]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa366597%28v=vs.85%29.aspx. [Poslednji pristup 24 8 2014].

[15] Microsoft, „HeapFree function,“ Microsoft, [Na mreži]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa366701%28v=vs.85%29.aspx. [Poslednji pristup 24 8 2014].

[16] Microsoft, „Memory management tracing events,“ Microsoft, [Na mreži]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/dn569310%28v=vs.85%29.aspx. [Poslednji pristup 30 8 2014].

[17] E. Lippert, „How Do The Script Garbage Collectors Work?,“ Microsoft, 17 9 2003. [Na mreži]. Available: http://blogs.msdn.com/b/ericlippert/archive/2003/09/17/53038.aspx. [Poslednji pristup 31 8 2014].

[18] Microsoft, „Advance in JavaScript performance in IE10 and Windows 8,“ Microsoft, 13 6 2012. [Na mreži]. Available: http://blogs.msdn.com/b/ie/archive/2012/06/13/advances-in-javascript-performance-in-ie10-and-windows-8.aspx. [Poslednji pristup 31 8 2014].

[19] M. Kralj, „Memory leaks in WWA apps,“ Internal Microsoft documentation, Redmond, 2012.

[20] M. Živković, „Algoritmi,“ Beograd, pp. 130-142.

[21] Microsoft, „WinJS.Promise object,“ Microsoft, [Na mreži]. Available: http://msdn.microsoft.com/en-us/library/windows/apps/xaml/br211867.aspx. [Poslednji pristup 24 8 2014].

[22] Microsoft, „WinJS.UI.IListDataAdapter interface,“ Microsoft, [Na mreži]. Available: http://msdn.microsoft.com/en-us/library/windows/apps/br212603.aspx. [Poslednji pristup 24 8 2014].