

Programimi me soketa

16.04.2018

Albos Hajdari

160714100020

Rrjetet Kompjuterike

Prof. Asoc. Dr Blerim Rexha

Ass. Haxhi Lajqi

Microsoft Visual Studio(2017)

Python programming language

Windows 7

Universieti i Prishtinës

Fakulteti i Inxhinierisë Elektrike dhe Kompjuterike

* **IPADDR**
* **PORTNR**
* **ZANORE**
* **PRINTO**
* **HOST**
* **TIME**
* **LOJA**
* **FIBONACCI**
* **KONVERTO**
* **GUESS**
* **NXITIMI**

ABSTRAKTI

Duhet bërë krijimi i katër programeve në gjuhën programuese Python[[1]](#footnote-2)  nëpërmjet soketave[[2]](#footnote-3): klienti dhe serveri në protokollet TCP[[3]](#footnote-4) dhe UDP[[4]](#footnote-5).

Klienti dërgon disa metoda me ose pa parametra ku serveri më pas i kthen përgjigje.

Pasi që të vendoset lidhja ndërmjet serverit dhe klientit, problemi kryesor është që të konvertohen dhe përpunohen të dhënat si stringjë apo numra dhe të bëhet validimi i tyre nga të dy anët e komunikimit (në qoftë se njëra anë dështon ta bëjë këtë).

Ky problem kryesisht zgjidhet duke përdorur metodat built in në Python për manipulimin me stringje.

Gjithashtu duhet bërë që serveri të punojë me dy e më shumë klienta njëkohësisht. Kjo është e mundshme përmes multithreading[[5]](#footnote-6). Vërtetimi se a po i'u shërben një server dy e më shumë klientave apo vetëm njërit mund të bëhet duke krijuar unaza që ekzekutohen pambarimisht (psh. **while True: do something**) tek serveri dhe klientët. Kur nuk e kemi serverin me multithreading, atëherë tërë "hapësirën procesorike" do ta zëjë vetëm klienti i parë i cili kyçet në server, e klientit të dytë do të i'u duhet që të pres derisa ajo hapësirë të lirohet, kësisoji do mund të shohim vetëm dy unaza në veprim (atë të serverit dhe klientit të parë). Ndërsa në qoftë se serverin e kemi bërë me multithreading, atëherë ajo "hapësirë procesorike" do të i'u ndahet të gjithë klientëve njësoj, andaj dhe do mund të shohim tri e më shumë unaza në të njejtën kohë të ekzekutohen (të serverit, të klientit të parë, të klientit të dytë etj.)

Si rezultat kemi këto 4 programe që do mund të komunikojnë me cfarëdo klienti apo serveri tjetër të ngjashëm që punon në protokollin përkatës dhe i ka të njejtat metoda.

HYRJA

Programimi me soketa nuk është aq i vështirë e as aq i lehtë sa paragjykohet në fillim. Mjafton të mendohen klienti dhe serveri si dy programe të ndara por shumë të ngjashme që dallojnë kryesisht nga qëllimi i tyre. Ajo çka dërgohet nga klienti quhet kërkesë ndërsa çka dërgohet nga serveri tek klienti do quhet përgjigje.

Në protokollin TCP tek klienti dhe serveri krijohen soketat përkatëse, ku klienti konektohet në port ndërsa serveri "ngjitet" (bind). Dallimi mes *connect* dhe *bind* është se me *connect* lirohet porti për përdorim nga ana e klientit posa të dërgohet kërkesa, ndërsa me *bind* nga ana e serverit ai port është i zënë gjersa serveri të jetë aktiv. Hapi i ardhshëm i serverit është që të ndëgjojë *listen*(*nr*) ku *nr* paraqet se sa klienta qëndrojnë në pritje tek ai port nëse ai port është tashmë i zënë me ndonjë klient tjetër. E më pas serveri duhet të bëhet gati per pranimin e kërkesave me *socket.accept()* . Tanimë klienti e dërgon kërkesen *me socket.sendall*(*kerkesa*) e cila kërkesë duhet të shëndrrohet në bajta paraprakisht. Me connectionSocket.recv(*length*) (parametri i parë i socket.accept() ) serveri e pranon kërkesën në formë të bajtave ku *length* paraqet numrin e bajtave. Pas përpunimit të kërkesës nga ana e serverit, ai e kthen përgjigjen me anë të komandës *connectionSocket.send*(*përgjigjeja*) ku *përgjigjeja* duhet të konvertohet në bajta. Më pas lidhja mbyllet me *connectionSocket.close()* nëse klienti veçse nuk ka ndonjë kërkesë të re, dhe gjithashtu nëse dëshirojmë ta ndalojmë punën e serverit, atë e bëjmë me socket.close(). Klienti më pas e pranon përgjigjen nga serveri me komanden *socket.recv*(*length*) e cila përgjigje është e gatshmë të printohet. Pra thënë më shkurtë, fillimisht bëhet një lidhjë përshëndetjeje (handshake) në portin e caktuar me komandat *socket.connect* nga ana e klientit dhe *socket.bind* nga ana e serverit, e më pas përdoren portet tjera të lira të gjeneruara me radhë të cilat nuk janë nën kontrollën e asnjërit përdorues dhe gjenden brenda intervalit [1024,65535].

Në protokollin UDP është shumë më thjeshtë, nga ana e klientit vetëm krijohet soketa dhe dërgohet kërkesa me komandën *socket.sendto*(*kërkesa,* (*IPadresa* , *porti*)) ku kërkesa duhet të jetë e tipit bytes, pra duhet konvertuar paraprakisht. Nga ana e serverit krijohet soketa dhe lidhet serveri për portin dhe hostin përmes komandës *socket.bind*((*serverIP*, *serverPort*)) . Më pas serveri e pranon kërkesën me komandën *socket.recvfrom*(*length*) ku *length* është gjatësia e bajtave të kërkesës. Ne faktë *socket.recfrom* është një listë me dy anëtarë, ku anëtari kërkesa është ruajtur në anëtarin e parë ndërsa në anëtarin e dytë është ruajtur një listë tjetër me dy anëtarë tjerë: anëtari i parë është IP-adresa e klientit ndërsa anëtari i dytë është numri i portit. Pas përpunimit të kërkesës, dërgohet përgjigjeja (e konvertuar paraprakisht ne bajta) me komandën *socket.sendto*(*përgjigjeja*, *addr*) ku *addr* është ajo lista në të cilën ruhet IP-adresa e klientit dhe numri i portit. Klienti e pranon më pas përgjigjen përmes komandës *socket.recvfrom*(*length*) e cila është një listë ku anëtari i parë është përgjigjeja në bajta ndërsa anëtari i dytë është një listë e përbërë nga IP-adresa dhe numri i portit të caktuar në fillim.

Këta serverë duhet që gjithashtu të mund të iu përgjigjen shumë klientave njëkohësisht. Kjo arrihet përmes multithreading. Fillimisht importohen libraritë *\_thread* dhe *threading*. Pastaj krijohet nje funksion main ku krijohet soketa dhe brenda ketij funksioni krijojme nje unazë infinit ku krijohet threadi me komanden *start\_new\_thread(threaded, (parametratTjerë))* ku *threaded* është një funksion tjetër në të cilin gjendet i gjithë kodi për pranimin e kërkesave dhe përpunimit të tyre. Ndërsa *parametratTjerë* janë parametrat e nevojshëm të soketës që duhet për të pranuar kërkesa. Pas çdo "infinit" unaze është mirë (në UDP) apo edhe e nevojshme (në TCP) që të mbyllen lidhjet përkatësisht soketat pasi që lirohet hapësira procesorike dhe memorike që u ishte ndarë/rezervuar klientëve paraprak. TCP me multithreading është shumë ide e mirë ngase ky protokoll është më i ngadalshëm/kujdesshëm, kështu që nuk duam që i gjithë serveri të bllokohet vetëm nga një klient. Gjersa UDP me multithreading është një ide jashtëzakonisht e keqe për arsye se UDP ashtu-kështu dërgon të dhëna të vogla, pra përdoret për të dërguar "thërrmija", dhe kërkesat zakonisht janë të shkurta dhe klientët nuk e zënë/bllokojnë serverin për kohë të gjatë, andaj në momentin që klienti largohet, pra e liron portin, menjëherë hyn klienti tjetër (për dallim nga TCP ku duhet gjithsesi që të mbyllet një lidhjë në mënyrë që të hapet lidhja tjetër, gjersa nuk është multithreading). Dmth. UDP shumë mirë i'u shërben shumë klientave "njëkohësisht" për kërkësa të shpeshta nëse ato kërkesa veçse janë të vogla. Dhe krijimi i një serveri UDP me multithreading vetëm se do zinte shumë hapësirë procesorike dhe memorike (të RAM-it) pa nevojë. Megjithatë në këtë detyrë e kam shkruar një UDP server me multithreading por i cili është shumë jo praktik. Mendimi i parë i zgjidhjes së këtij problemi ka qenë që të mbyllet soketa pas çdo kërkese mirëpo kjo nuk mund të bëhet nëse ka klienta aktivë ngase serveri më nuk do mund të iu shërbejë të tjerëve. Prap se prap në qoftë se për cfarëdo arsye qoftë serveri apo klienti kanë punë tjera për të kryer pas komunikimit që kanë më njërin tjetrin, është mirë që të mbyllen soketat në mënyrë që të lirohet RAM-i dhe procesori nga variablat e panevojshme.

ZHVILLIMI I METODAVE

Kërkesën e çdo klienti të pranuar nga serveri fillimisht e kam ndarë duke e shëndrruar çdo fjalë në anëtarë të një liste të vetme. Kësisoji kam mundur t'i marrë shumë lehtë "funksionet" dhe "parametrat" e tyre ndaras, dhe shumë më lehtë është mundësuar identifikimi i saktë i parametrave. Kjo ka mundur edhe të bëhet me analizë të stringut por largimi i hapësirave të tepërta aty ku duhet dhe ndarja e parametrave "falso" nga ata "të vërtetit" do ishte një projekt tjetër ne vete.

Metodat të cilat nuk kanë asnjë parametër si IPADDR, PORTNR, HOST, TIME dhe LOJA i kam ndarë në një kategori me një kusht *if*, metodat që e kanë nga një parametër sikurse ZANORE, PRINTO, FIBONACCI, GUESS në një kategori tjetër me kushtin pasues *elif*, e kështu me radhë metodat tjera në kategoritë e tyre përkatëse me dy , tre e katër parametra dhe në fund me *else* i kam validuar çdo kërkesë tjetër nga ana e klientit, kërkesa të cilat nuk kanë arritur fare tek serveri që të zënë hapësirë pa nevojë.

**1.** **IPADDR**

Nëse kërkesa e klientit është vetëm një fjalë e gjatë dhe ajo fjalë është IPADDR, atëherë thirret metoda IPADDR e cila si parametër ka *addr* që është anëtari i dytë i *serverSocket.recv* përkatësisht *socket.recvfrom*. Nga *addr* mirret anëtari i parë i cili paraqet tamam IP-adresën e klientit.

**2. PORTNR**

Ngjashëm sikurse metoda IPADDR vetëm se këtu mirret anëtari i dytë i *addr*.

**3. HOST**

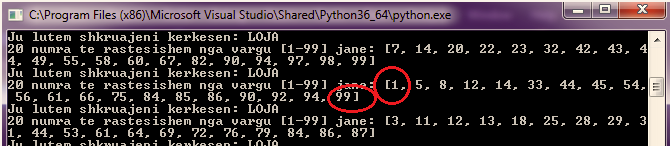
Thirret metoda HOST() tek serveri e cila kthen si rezultat përmbajtjen e funksionit built in *gethostname()* nga libraria *socket*.

**4. TIME**

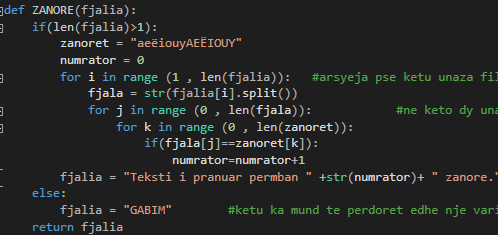
Për këtë duhet importuar librarinë *datetime* dhe përdoret funksioni i gatshëm *datetime.datetime.noë()* i cili kthen vitin, muajin, diten, orën, minutat, sekondat dhe mikrosekondat, andaj mikrosekondat si të panevojshëm largohen me komanden *koha = koha[:-7]* . Stringjet dhe listat në përgjithësi në Python janë të numrueshme nga të dyja anët, ku [-1] është anëtari i fundit, operatori : fshin nga cilado anë e kundërt që gjindet, pra dmth *koha[:-7]* i fshin 7 karakteret e fundit , në këtë rast 6 për shifrat e mikrosekondave dhe 1 për pikën ndarëse në mes të sekondave dhe mikrosekondave.

**5. LOJA**

Për këtë duhet importuar librarinë *random* dhe përdoret funksioni i gatshëm *random.sample(range(a,b), c)* ku *a* dhe *b* paraqesin intervalin e numrave [a,b-1] ndërsa *c* paraqet sa numra duhet të zgjedhen nga ky interval, në rastin tone këta parametra duhet te jene *a*=1, *b*=100, *c*=20 .



**6. ZANORE**

Tani e tutje kërkesat kanë më shumë se 1 fjalë. Andaj si parametra gjithmonë përpunohen vetëm fjalët që vijnë pas emrit të metodës. Andaj unaza e jashtme fillon nga anëtari i dytë (anashkalon fjalen ZANORE).

Më pas e gjithë kërkesa ndahet në fjalë në këtë unazë, ndërsa unaza e dytë shërben për t'u qasur në çdo karaktër të fjalës aktuale, gjersa unaza e tretë shërben për tu qasur në çdo karakter të variables *zanore*.

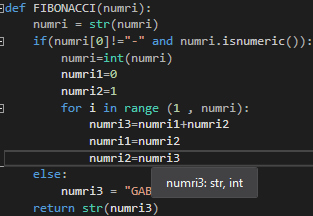
Duhet cekur që unaza fillon nga parametri i parë e përfundon deri tek një numër më pak se parametri i dytë.

**7. PRINTO**

Për metodën PRINTO është përdorur variabla *var0* ngase variabla *var* nuk ka hapësira të krijuara tasti TAB, hapësira të cilat për shkak të origjinalitetit që metoda PRINTO kërkon duhet të ruhen. Nga *var0* janë hequr të gjitha hapësirat para fjales "PRINTO" me funksionin e gatshem në Python, që është *lstrip(" ")*, pastaj janë hequr 6 karaktere nga ana e majtë në të djathtë, pra vetë fjala "PRINTO" . E më pas arrijmë tek kushtëzimet e metodës e që janë se teksti duhet të printohet pa hapësirat në të majtë dhe djathtë të tij, pra i heqim këto hapësira me *lstrip(" ")* dhe *rstrip(" ")* dhe më pas dërgohet përgjigjeja për t'u printuar.

**8. FIBONACCI**

Validimi i parametrit të dytë tani bëhet pak më i vështirë ngase nëse parametri i dytë është numër negativ, ai nuk njihet nga metoda *isnumeric()* , andaj shenjën negative duhet ta heqim me lstrip("-") e më pas ta pyesim se a është numër me *isnumeric()* e më vonë ta pyesim me një kusht *if* nëse numri është më i vogël se zero apo jo (për ta ditur a është negativ apo jo).

****

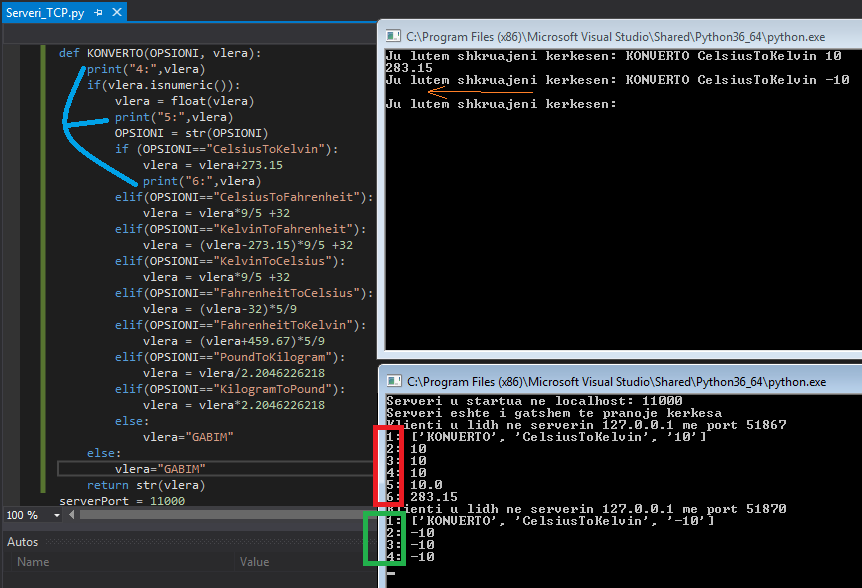
**9. GUESS**

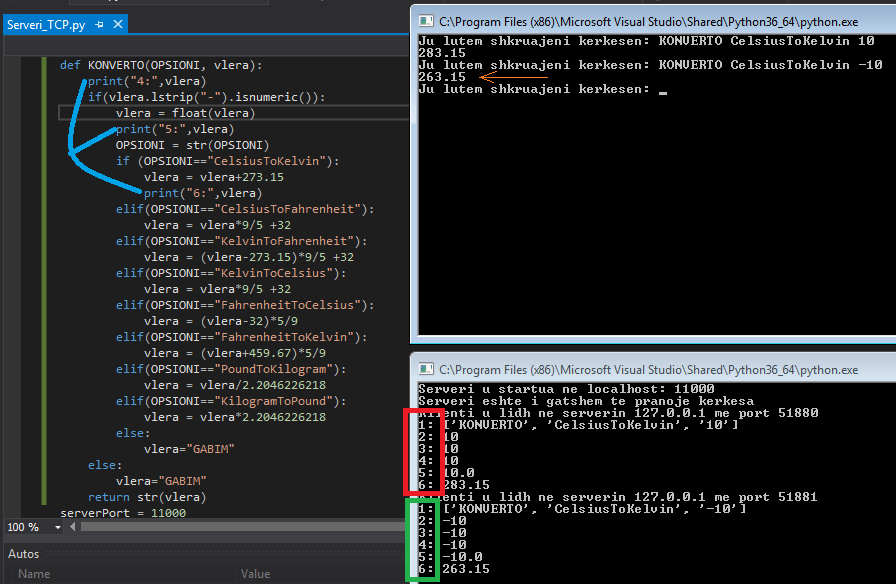
Validimi është i njejtë mu sikurse tek metoda FIBONACCI. Numri random gjenerohet jashtë çdo funksioni për arsye se nëse inicializohet brenda atëherë ai ndryshon për çdo kërkesë të re. Dhe qëllimi i kësaj metode është që klienti të mund t'a qëlloj numrin e saktë i cili është brenda intervalit [1,100].

**10. KONVERTO**

Validimi i saj ka qenë shumë i vështirë ngase është dashur që të tregohet se a mungon një apo dy parametra. Pastaj nëse janë të dy parametrat, a është parametri i tretë negativ apo jo, ngase nëse është negativ ai nuk guxon të përdoret për konvertim të masave. Pastaj Pythoni nuk ka ndonjë metodë të gatshme që tregon se a është numër qoftë edhe decimal, andaj me kombinimin e funksioneve *replace("." , "").isnumeric()* është arritur kjo gjë. Pjesa tjetër ka qenë vetëm formula matematikore shumë të thjeshta të marrë nga sistemi i njësive SI.

Vërtetimi se si funksioni *isnumeric()* nuk i njeh numrat negativ si numra të vërtetë:





**11. NXITIMI**

Validimi i numrave është bërë ngjashëm sikurse tek KONVERTO vetëm se parametri i tretë, pra koha, nuk guxon të jetë negative e as zero. Nxitimi është llogaritur sipas formulës (v2-v1)/t .

VALIDIMI I IP-ADRESËS DHE PORTIT

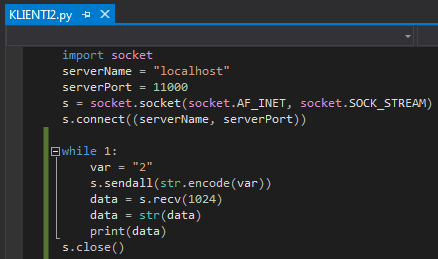
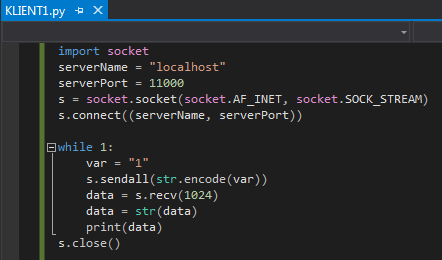
Ky validim është bërë asisoji që nëse nuk është shtypur asnjë tekst, si vlera default janë marrë "localhost" dhe 11000 . Në rastet tjera, porti është kufizuar brenda intervalit [1024,65535], e nësë ai nuk është numër me një unazë "infinit" kthehet prap që të shtypet përsëri. Ngjashëm dhe IP-adresa, ajo ose duhet të jetë "localhost" ose t'i takojë formatit *nr1.nr2.nr3.nr4* ku numrat duhet të jenë brenda intervalit [0,255].

SFIDAT E MULTITHREADING DHE VËRTETIMI QË FUNKSIONON

Multithreading është thjeshtë një lloj procesimi paralel. P.sh. e zëmë që klienti i parë kërkon një fajll të madhësisë 10GB dhe supozojmë se ky fajll në single-thread merr 1h kohë që të transferohet, pra nëse serveri nuk e ka aftësine të procesojë paralel, atëherë klientët tjerë nuk do kenë qasje në këtë server për 1h. Andaj serveri programohet që të punojë paralel, ku çdo klienti i ndahet nga një thread, kjo kuptohet që edhe fajlli prej 10GB tanimë nuk do transferohet për 1h por pak më shumë, psh për 1h10min. Mirëpo (hipotetikisht) këta 10 minuta jemi të gatshëm t'i "sakrifikojmë" për klientin e parë në mënyrë që edhe klientët e tjerë të kenë qasje për kohë të shkurtë (qoftë edhe për kohë të gjatë).

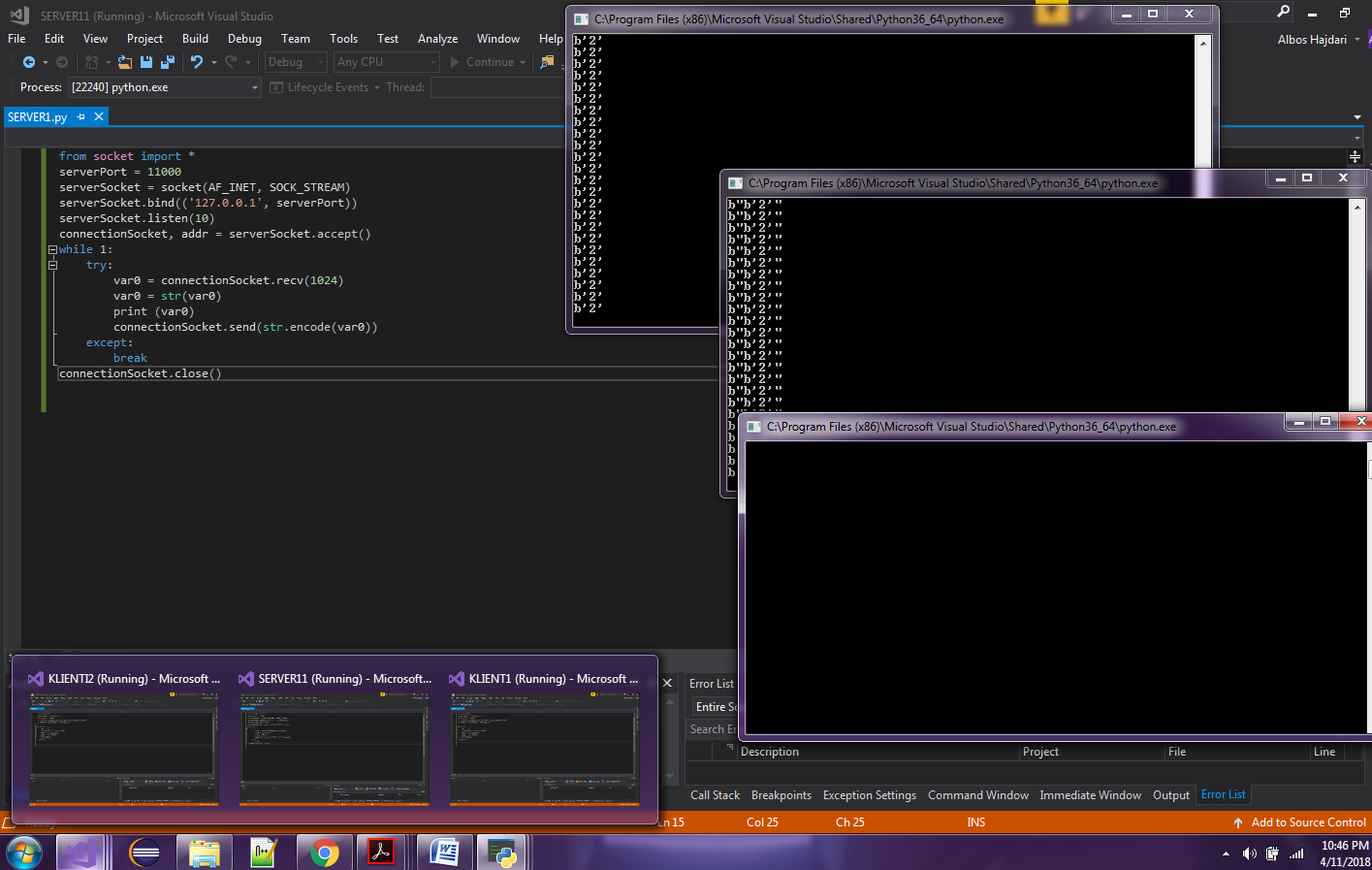
Siç është përmendur edhe më herët, në funksionin main të serverit krijohet soketa dhe ngjitet serveri për port, pastaj në unazën infinit krijohet lidhja (për TCP) dhe thirret funksioni i thread-it ku pranohen kërkesat dhe kthehet përgjigjeja.

Vërtetimi se a është duke i'u shërbyer një server dy e më shume klientave njëkohësisht është e vështirë të bëhet ngase një person i vetëm apo qoftë edhe dy persona nuk mund të sinkronizohen tamam që të i dërgojnë serverit dy kërkesa në të njejtën kohë. Andaj më lindi ideja që të krijoj dy klientë me unaza infinit ashtu që qëndrojnë duke dërguar nga një numër ("1" për klientin e parë dhe "2" për të dytin) dhe duke e pranuar përgjigjen e duke e printuar më pas tërë kohën. Ndërsa serveri nuk i bën asnjë lloj përpunimi kërkesës por vetëm printon dhe e kthen atë që ka pranuar tek klientët.



Rasti i parë, TCP single-threaded:

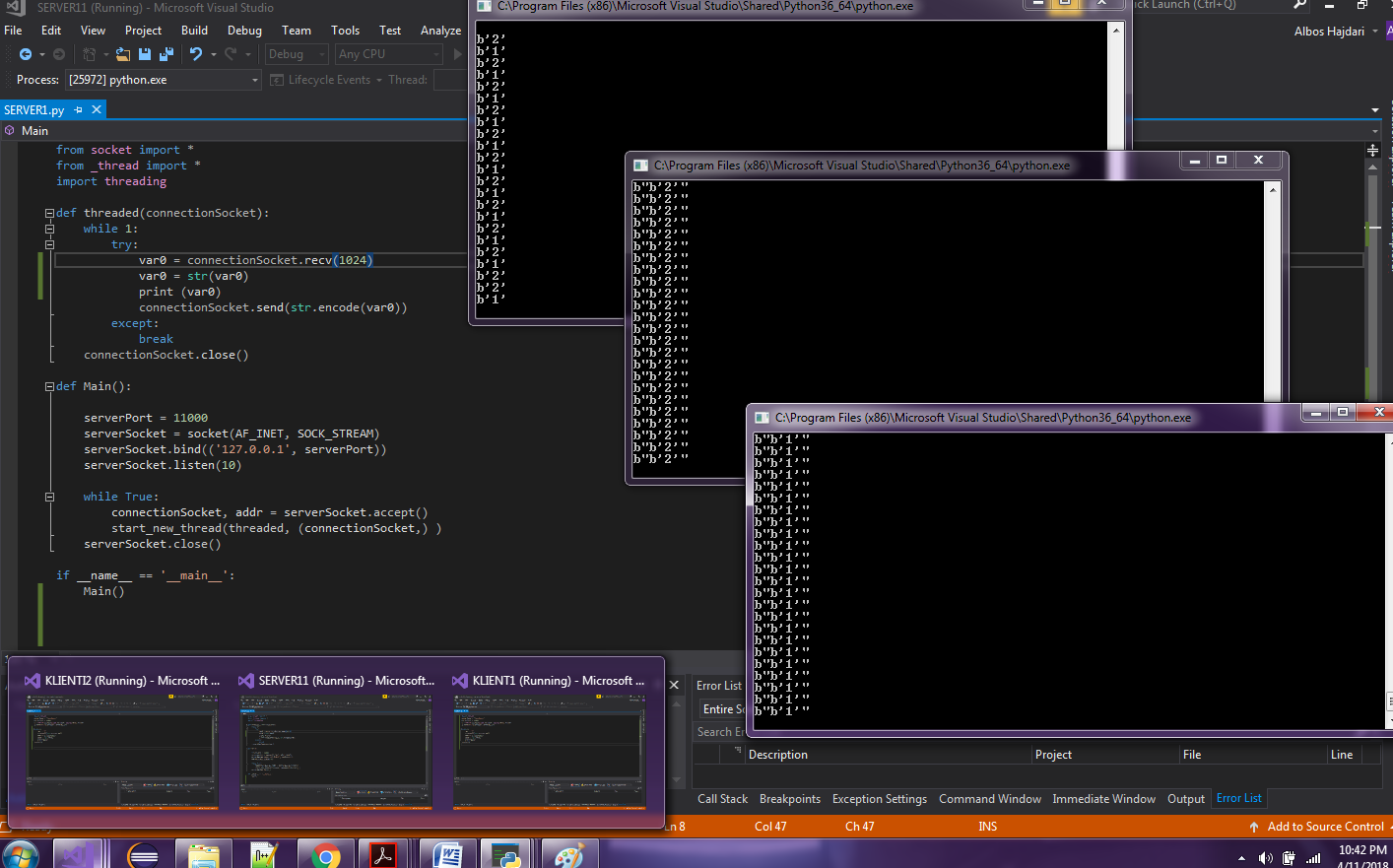
Nëse e ekzekutojmë serverin, dhe më pas klientin e dytë e më pas klientin e parë, do shohim se vetëm numri "2" do printohet në server, ndërsa klienti i parë do qëndrojë gjithmonë në pritje.



Pra është vërtetuar që vetëm njëri klient me një unaze infinit e ka bllokuar tërë serverin për klientët tjerë.

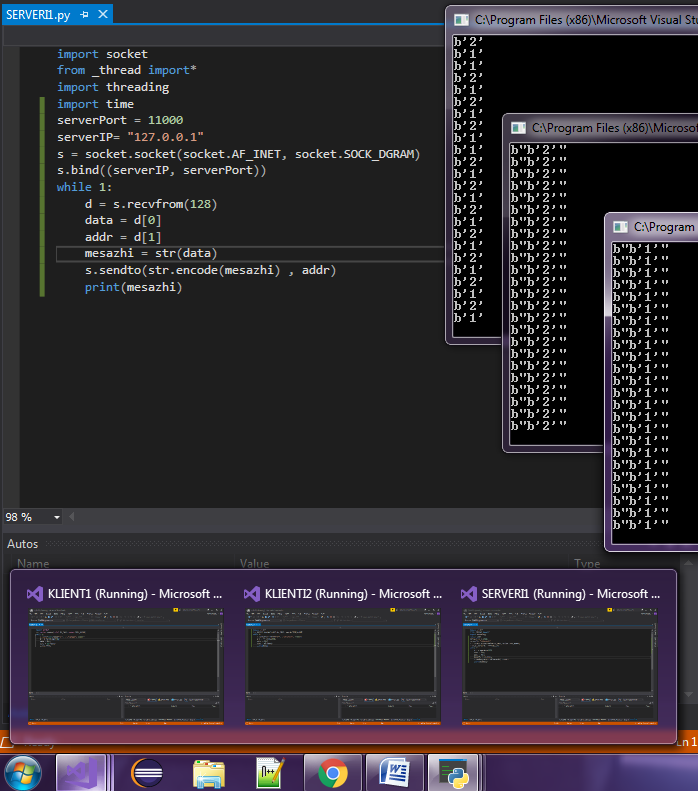
Rasti i dytë, TCP multi-threaded:

Ekzekutohet serveri, pastaj klienti i dytë e më pas i pari (rënditja nuk ka rëndësi) dhe shihet se serveri i'u kthen përgjigje të dy klientëve në të njejtën kohë ngase edhe numri "1" edhe "2" printohen tek serveri dhe asnjëri klient nuk qëndron në pritje por e printon atë çfarë i kthehet nga serveri.



Rasti i tretë, UDP single-threaded:

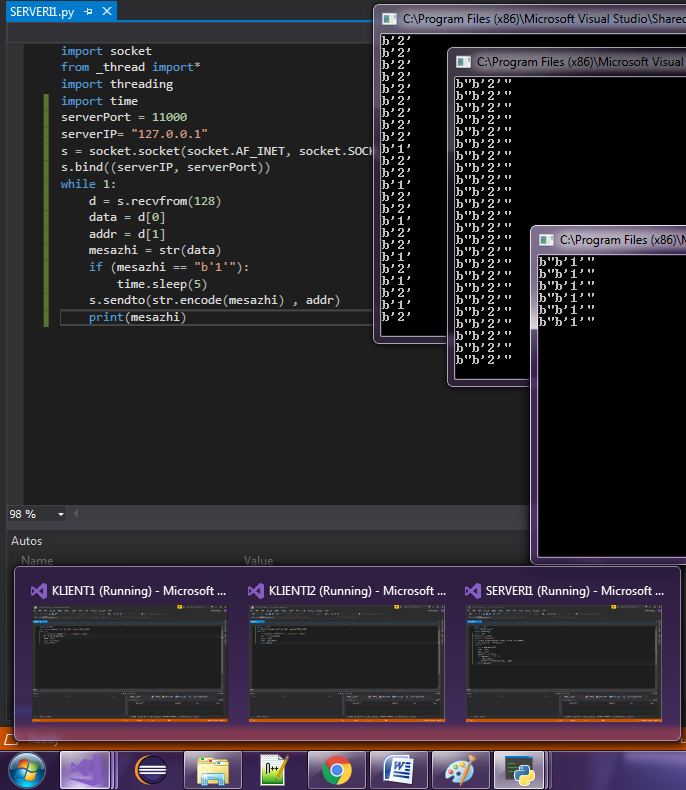
Ekzekutojme së pari serverin dhe më pas klientët, shihet se serveri mundet të iu kthej përgjigje të dy klientëve në "të njejtën kohë" . Mirëpo kjo është për arsye se protokolli UDP nuk krijon lidhje por vetëm e pranon kërkesën (e përpunon nëse ka nevojë por jo në këtë rast) dhe e dërgon përgjigjen tek klienti. Pra klienti këtu nuk mund ta bllokojë përgjithmonë portin ngase nuk formon lidhje, për dallim prej protokollit TCP ku porti bllokohet për shkak të lidhjes, dhe deri të shkëkuptet lidhja nuk mund të kyçet klienti tjetër.



Pra siç duket serveri i'u përgjigjet të dy klientëve në të njejtën kohë, mirëpo kjo nuk është e vërtet, thjeshtë i'u përgjigjet herë njërit e herë tjetrit, dhe kjo krijon iluzionin e "procesimit paralel"

Rasti i katërt, UDP single-threaded me vonesë kohore:

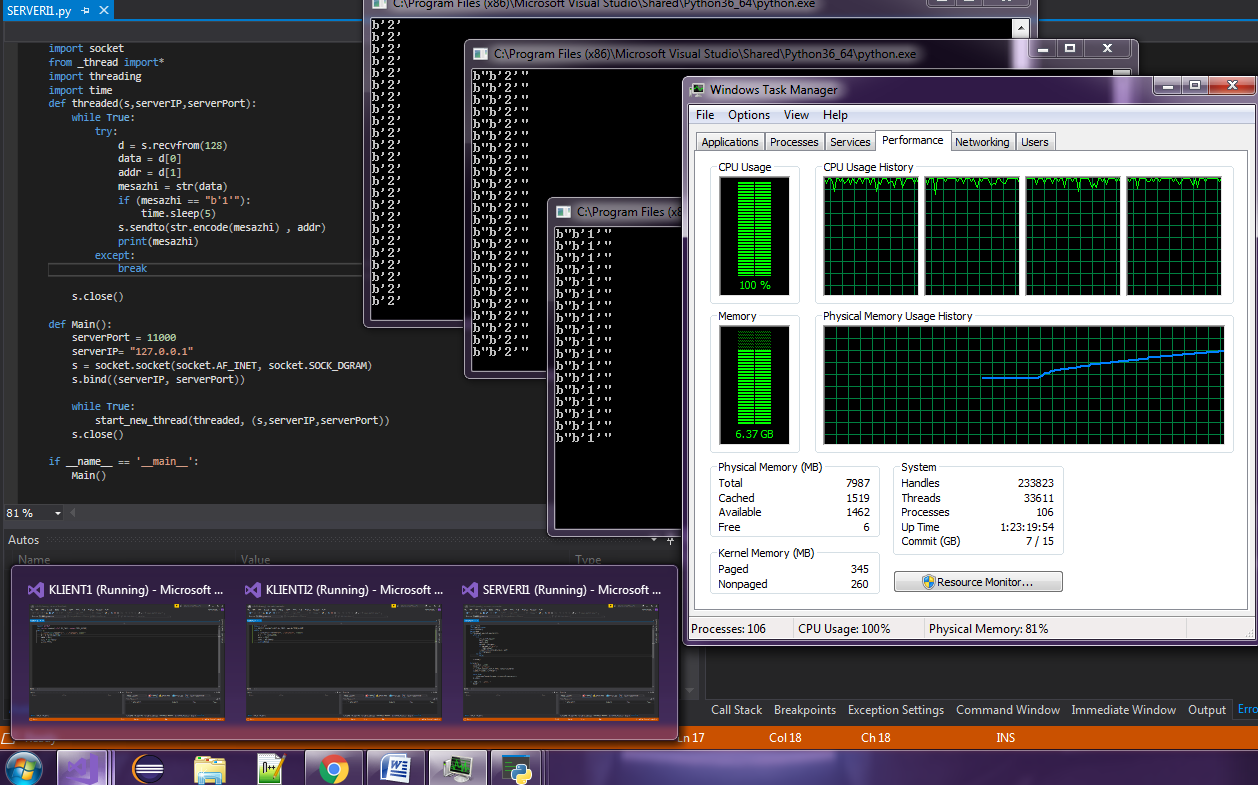
Për ta vërtetuar që serveri me protokollin UDP single-threaded nuk i'u përgjigjet të dy klientëve në të njejtën kohë, vendosim një vonesë kohore. P.sh. nëse vjen numri "1" (pra kërkesa nga klienti i parë) e bëjmë serverin të pushojë 5 sekonda përmes komandës time.sleep(5). Kështu nëse e ekzekutojmë serverin, më pas klientin e dytë, do printohet numri "2" tek të dy programet disa herë shumë shpejt (shkaku i unazes infinit) mirëpo nëse e ekzekutojmë edhe klientin e parë i cili e dërgon numrin "1" atëherë i gjithë serveri do vonohet për 5 sekonda, dhe brenda këtyre 5 sekondave nuk do printohet asgjë, pra as numri "1" e as "2" . Mirëpo pasi që të kalon kjo vonesë, do printohet "1" dhe "2" në "kohë reale" dhe më pas menjëherë futet serveri në vonesën kohore prej 5 sekondave. Dmth. siç shihet tek serveri në pjesën e parë të konsollës është printuar vetëm "2" , ndërsa në pjesën e dytë është printuar edhe "2" edhe "1" me kombinim.



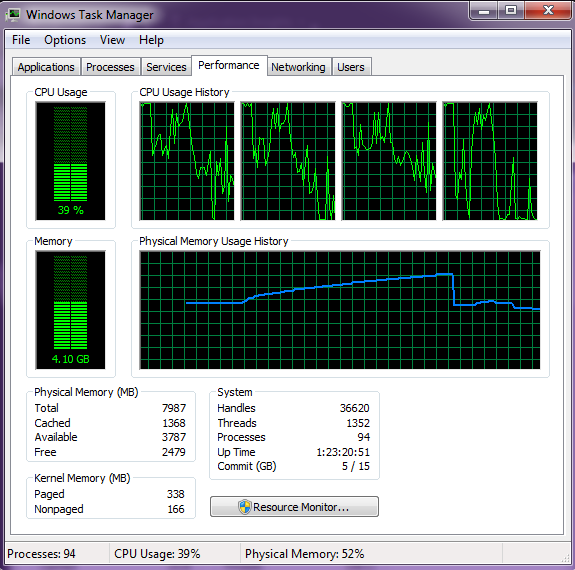
Pra është vërtetuar që nëse njëri klient e ka një kërkesë që merr shumë kohë nga serveri që të procesohet, klientët tjerë detyrohen të presin.

Rasti i pestë, UDP multi-threaded:

Tani shihet qartë se serveri i'u përgjigjet të dy klientëve në të njejtën kohë ngase gjatë vonesës kohore prëj 5 sekondave, e gjithë konsolla e serverit është e mbushur me numrin "2" . Por këtu qëndron një problem tjetër. UDP këtu krijon shumë thread-a dhe e zën tërë "hapësirën" procesorike. Siç shihet dhe në grafikon të përformancës së Task Manager, përdorimi i procesorit është në 100%, dhe përdorimi i RAM-it vetëm sa shkon e rritet, gjithashtu numri i thread-ave është jashtëzakonisht i madh. Pra me protokollin UDP nuk është praktike që të bëhet me shumë threada. I bie që UDP nuk përdoret aty ku ka transferim të fajllave të madhësive të mëdha por vetëm për fajllva/kërkesa të vogla (e edhe nëse janë të shpeshta nuk paraqet problem).



Gjithashtu në screenshot-in e radhës, pasi e kam mbyllur programin e serverit, vërehet se pas një kohe të shkurtë (57 sekonda në këtë rast, por screenshot-i ka mundur të bëhet edhe më herët, shiko UP Time) përqindja e CPU usage bie drastikisht (shiko edhe grafikun), numri i threadave bie për 24 herë (përafërsisht) dhe gjithashtu përdorimi i RAM-it bie me të madhe.



Duhet përmendur që ky përdorim i madh i CPU-s dhe RAM-it mund të jetë shkaku i përmbajtjes së kodit apo shkaku i vetë gjuhës programuese Python. Mirëpo problemin e njejtë siç duket e kane edhe të tjerët kur e shkruajnë një server multithreading me protokollin UDP në gjuhën programuese Java.

Shiko pyetjen në linkun më poshtë:

[https://ëëë.stackoverfloë.com/questions/49643046/high-cpu-usage-on-udp-datagramsocket-threads-in-java/49919958#49919958](https://www.stackoverflow.com/questions/49643046/high-cpu-usage-on-udp-datagramsocket-threads-in-java/49919958%2349919958)

PËRFUNDIMET

Duhet pasur shumë kujdes në tipet e të dhënave në python ngase shpesh ngatërrohen dhe p.sh. dëshiron që të zbresësh numra ndërkohë që variablat janë në tipin string, pra duhet bërë shprehi që nëse jemi duke e shkruar një program në nguti në python të konvertojmë të dhënat paraprakisht. Konvertimet më të shpeshta që i kam përdorur kanë qenë *var=str(var)* dhe *var = int(var)* .

Është shumë më lehtë që kërkesa të ndahet në listë fjalësh nëse kemi të bëjmë me shumë mëtoda dhe me parametrat e tyre.

Serveri me protokollin TCP ia vlen që të bëhet multithreaded mirëpo UDP asesi. TCP mund të përdoret edhe per transferime të mëdha të të dhënave. UDP nuk është farë praktike që të transferohen të dhëna të mëdha.

BIBLIOGRAFIA

* [https://stackoverfloë.com/questions](https://stackoverflow.com/questions)
* [https://ëëë.digitalocean.com/community/tutorials/](https://www.digitalocean.com/community/tutorials/)
* [https://ëiki.python.org/moin/UdpCommunication](https://wiki.python.org/moin/UdpCommunication)
* [https://ëiki.python.org/moin/TcpCommunication](https://wiki.python.org/moin/TcpCommunication)

1. Gjuhë e lartë programuese që zakonisht përdoret në GIS (Geographic Information Systems). [↑](#footnote-ref-2)
2. Socket (gjep) është pikë përfundimtare në një komunikim me dy kahje. [↑](#footnote-ref-3)
3. &4 Protokolle që përcaktojnë se si komunikojnë plaët në internet. [↑](#footnote-ref-4)
4. 5 Aftësia që një bërthamë e vetme procesorike të kryej disa procese "njëkohësisht" . [↑](#footnote-ref-5)
5. [↑](#footnote-ref-6)