Implementarea eficientă a unui server Web. Studii de caz și analiza performanței.

Eduard Budaca

Universitatea Alexandru Ioan Cuza, Facultatea de Informatică, Iași

Rezumat Prezentăm în această lucrare un server HTTP eficient, implementat în cod nativ C++, în care paginile dinamice sunt scrise în limbajul de programare Python. Acest server suportă funcționalități moderne, ca protocolul WebSocket, securitate prin Transport Layer Security (*TLS*) sau Secure Sockets Layer (*SSL*) și suport integrat pentru paradigma MVC. Unul din scopurile principale ale acestui proiect este expunerea unei interfețe de programare cu un bun echilibru între simplitatea utilizării și disponibilitatea facilităților utile sau necesare. Pentru îndeplinirea acestui scop, interfața de programare este intuitivă, iar limbajul de "machete de vizualizare" (*template*-uri) este ușor de folosit și facil de citit. Module importante care susțin execuția acestui server sunt un automat determinist puternic, reutilizabil și foarte util în elaborarea *parser*-elor, și un compilator de la fișiere de *template* în module din limbajul Python. La final, prezentăm studii de caz și comparații cu alte servere Web pentru același limbaj de programare.

Cuprins

1 Introducere

Krait este un server HTTP pentru sistemele Linux care furnizează capabilități de *scripting* la nivel de server în limbajul Python. Acest server nu este limitat la aplicații Web și contexte specifice, ci poate fi folosit pentru o largă varietate de aplicații.

1.1 Motivaţie

Această lucrare a început ca proiect final pentru disciplina Rețele de Calculatoare, fiind ulterior îmbunătățit considerabil, inițial pentru a fi folosit ca serverul HTTP care a furnizat aplicația Web prezentată ca proiect final pentru disciplina Tehnologii Web, iar apoi ca lucrare de licență. Alegerea inițială a fost motivată de interfețele de programare greoaie și excesiv de complexe folosite pentru a crea o aplicație Web în majoritatea serverelor HTTP existente, în special dacă se dorește folosirea limbajului de programare Python pentru scripturile la nivel de server. În plus, s-a dorit dezvoltarea unui server HTTP care să ofere mediul de programare bazat pe limbajul Python, menționat mai sus, pe dispozitive Internet of Things cu resurse foarte limitate.

1.2 Grad de noutate

Krait expune o interfață de programare simplă pentru paginile Web dinamice. Limbajul fișierelor de tip "machetă de vizualizare"(*template*), care pot fi view-urile din paradigma Model-View-Controller, dar și în orice alt fișier HTML, este ne-intruziv în codul paginii și, în același timp, este foarte similar codului Python, inclus în aceste pagini pentru a fi executat în procesul de generare a răspusului HTTP. În plus, cea mai mare parte a serverului este implementată în cod C++, compilat în limbaj mașină. Această trăsătură separă semnificativ Krait de alte servere HTTP existente scrise pentru limbajul Python, ca Flask și Django, care sunt implementate exclusiv în cod Python. Din cauza faptului că Python este un limbaj interpretat, aceste proiecte existente suferă o penalizare semnificativă de performanță, lucru evitat de această implementare. Un alt avantaj al acestui server este o implementare a protocolului WebSocket, în cod C++. Această implementare este integrată și coerentă cu restul interfeței de programare, spre deosebire de serverele menționate mai sus, în care, pentru a folosi acest protocol, este necesară instalarea de extensii și modificări. Motivul acestei separări este dependența acestora de paradigma WSGI[1], care nu este compatibilă cu modelul bidirecțional al protocolului WebSocket. O altă optimizare semnificativă a acestui proiect este compilarea paginilor dinamice în cod Python, executat de către server la primirea unei cereri HTTP, evitându-se o reprocesare a codului sursă Python pentru generarea fiecărui răspuns.

[1] `Web Server Gateway Interface` (*WSGI*) este o evoluţie a protocolului CGI, concepută pentru limbajul Python şi specificată iniţial în PEP 333, disponibil la `https://www.python.org/dev/peps/pep-0333/`.

1.3 Concepte utilizate

Protocolul HTTP. Acest server implementează protocolul HTTP, versiunea 1.1, RFC-ul curent fiind RFC 7230 al Internet Engineering Task Force (IETF)[1].

HTTP este un protocol de tip server-client. Fiecare cerere HTTP, trimisă de către un client către un server, are următoarele componente:

- O metodă HTTP, de exemplu *GET*, *POST* sau *DELETE*, care specifică tipul de acțiune pe care ar trebui să-l efectueze serverul.
- un URL, care specifică resursa asupra căreia se efectuează acțiunea.
- o versiune HTTP, de exemplu *HTTP/1.1*
- o listă de câmpuri antet, specificând metadate despre conținutul cererii sau despre conexiunea dintre client și server.
- opțional, un conținut, de lungime egală în octeți cu cea specificată în lista de câmpuri antet.

Un răspuns HTTP, trimis de un server către un client în urma unei cereri trimisă de acesta, are următoarele componente:

- versiunea protocolului HTTP, în același format ca în cazul cererii
- un cod de stare, format din trei cifre, din care prima specifică tipul general de stare (de exemplu, succes sau eroare a clientului), iar ultimele două dau informații detaliate.
- un scurt mesaj care detaliază codul de stare.
- o listă de câmpuri antet, cu aceeași semnificație ca în cazul cererii.
- opțional, un conținut, de lungime egală în octeți cu cea specificată în lista de câmpuri antet.

Protocolul WebSocket. Acest server implementează protocolul WebSocket, RFC-ul curent fiind RFC 6455 al IETF[3].

WebSocket este un procotol separat de HTTP în care modelul client-server este înlocuit cu un model bidirecțional de comunicare, care simplifică mecanismele disponibile unui server care dorește să trimită mesaje unui client în mod asincron. În plus, un mesaj WebSocket transmis folosește între 2 și 14 octeți de antet, spre deosebire de antetul semnificativ mai mare al unui mesaj HTTP, scăzând congestia rețelei și latența comunicării. De asemenea, analiza semantică a unei cereri HTTP este mai complexă decât citirea unui mesaj WebSocket.

Față de comunicarea prin socket-uri TCP brute, acest protocol prezintă următoarele avantaje:

- Mesajele sunt procesate unitar și integral, acestea fiind delimitate, spre deosebire de fluxul de octeți al socket-urilor TCP. În cazul acestora, un mesaj poate fi, la un anume moment, recepționat doar parțial, sau în care două mesaje consecutive pot fi concatenate fără posibilitatea de a le distinge.
- Procesul de stabilire a conexiunii WebSocket se efectuează prin HTTP, iar comunicarea WebSocket continuă peste aceeași conexiune TCP, ceea ce face ca acest protocol să fie compatibil cu HTTP.

- Protocolul WebSocket face diferența între mesajele de text și cele de conținut binar.
- Acest protocol suportă un mecanism de verificare a validității unei conexiuni, prin trimiterea unor mesaje speciale de *ping-pong*, cărora li se răspunde automat, imediat ce sunt recepționate.

Stabilirea conexiunii WebSocket se realizează prin protocolul HTTP, printr-o cerere care conține câmpul de antet "Connection" cu valoarea "Upgrade", câmpul "Upgrade" cu valoarea "websocket" și, opțional, un *challenge* în câmpul "Sec-WebSocket-Key". După ce serverul rezolvă acest challenge, trimite un răspuns HTTP de tip "101 Switching Protocols" cu rezultatul acestui challenge în câmpul de antet "Sec-WebSocket-Accept". După trimiterea acestui răspuns, comunicarea pe această conexiune TCP se efectuează prin protocolul WebSocket.

Protocolul WebSocket suportă negocierea de subprotocoale, care reprezintă convențiile de comunicare peste această conexiune. Astfel, în procesul de stabilire a conexiunii WebSocket, clientul specifică, în câmpul de antet "Sec-Websocket-Protocol", o listă de subprotocoale prin care este dispus să comunice, iar serverul alege unul din acestea, specificându-l într-un câmp de antet cu același nume în răspunsul său.

Protocolul TLS. Comunicarea dintre server și client se realizează folosind socket-uri TCP, fie in clar, fie folosind TLS sau predecesorul acestuia, SSL.

TLS și SSL, așa cum sunt folosite în contextul Web-ului, sunt protocoale criptografice care urmăresc să asigure confidențialitatea comunicării dintre client și server. În plus, acestea asigură autentificarea serverului față de client, asigurându-l pe cel din urmă că nu este victima unui atac de interceptare sau modificare a datelor transmise prin conexiunea folosită.

Versiunea în vigoare a protocolului TLS este 1.2, iar următoarea va fi 1.3. Specificația TLS 1.3 este disponibilă la [7], iar implementării OpenSSL a protocoalelor TLS și SSL este disponibilă la [8].

Limbajul de programare Python. Această lucrare pune la dispoziție pentru scripturile la nivel de server un mediu de execuție pentru limbajul de programare Python[13].

Acesta este un limbaj de programare interpretat, cu un pas adițional de compilare în *bytecode*, care suportă paradigmele de programare imperativă, funcțională, procedurală sau orientată-obiect[9]. Python este general apreciat pentru viteza mare de dezvoltare a aplicațiilor, sintaxa și comportamentul intuitiv și un ecosistem bogat de biblioteci ("pachete") ușor de integrat în proiectele existente. Python este general folosit ca limbaj de scripting, dar și în dezvoltarea aplicațiilor Web la nivel de server, analiza datelor și analiză numerică.

Cea mai distinctivă trăsătură a sintaxei acestui limbaj este folosirea indentării pentru a delimita blocurile de instrucțiuni. Variabilele nu sunt declarate și sunt

de tip dinamic[2]. Cu toate acestea, limbajul nu efectueaza conversii implicite între tipurile de variabile, evitând aceste clase de erori prezente în limbaje *loosely-typed* ca JavaScript.

Implementarea limbajului Python este furnizată de către biblioteca cu același nume, scrisă pentru limbajul C și compatibilă cu limbajul C++, reprezentând implementarea CPython a Python Software Foundation.

1.4 Contribuţiile lucrării

Contribuția principală a acestui proiect este posibilitatea de a crea o aplicație Web în limbajul Python, folosind o interfață de programare intuitivă și ușor de folosit. Această simplitate a interfeței se extinde și la limbajul fișierelor de *template*, limbaj elegant și clar. O altă contribuție adusă de această lucrare este implementarea protocoalelor HTTP și WebSocket în cod nativ, singurele locuri unde se rulează cod interpretat fiind implementarea aplicației Web, scrisă de utilizatorul acestei lucrări, și interfața cu această aplicație. Analiza textuală a cererilor HTTP, a limbajului fișierelor de *template* și a unei porțiuni restrânse din limbajul Python este realizată cu un automat generic, eficient și puternic, iar fișierele *template* sunt compilate în limbajul de programare Python, pentru a fi ușor executate de acest mediu de execuție. În plus, eficiența implementării lucrării face ca aceasta să funcționeze pe dispozitive Internet of Things cu resurse reduse de procesare și de memorie.

[2] În limbajele de programare cu *dynamic typing*, instrucţiunile de atribuire pot schimba la rulare tipul unei variabile.

2 Arhitectura lucrării

Lucrarea este o aplicație implementată în limbajul de programare C++, care incorporează un interpretor al limbajului de programare Python, versiunea 2.7, furnizat de către biblioteca libpython.

Arhitectura serverului este de tip forking la cerere, semnificând faptul că fiecare conexiune TCP nouă (un socket TCP) pornește un nou proces prin mecanismul de *fork* disponibil în sistemele Unix. S-a ales o arhitectură multiproces, spre deosebire de o arhitectură cu mai multe fire de execuție, cu scopul de a oferi un grad de separare a memoriei RAM dintre procesele care servesc clienți diferiți, iar arhitectura cu *fork* la cerere a fost aleasă din cauza simplității relative a implementării. Odată pornit, un astfel de proces va servi oricâte cereri sunt recepționate pe acea conexiune, folosind mecanismul de *Keep-Alive* disponibil în standardul HTTP/1.1[3].

2.1 Module

Implementarea acestei lucrări este împărțită în diferite module:

Lansarea serverului. Fișierul executabil *krait* primește ca argumente la linia de comandă un director în care se află fișierele sursă pentru paginile Web care vor fi servite (*director rădăcină*), două porturi TCP, la care se vor primi cererile HTTP, respectiv HTTPS și două fișiere unde să fie scrise mesajele de output, respectiv mesajele de eroare. În plus, o opțiune la linia de comandă activează modul "un singur process", care modifică arhitectura serverului cu scopul de a rula serverul fără a crea procese descendent. Acest mod de execuție este util atunci când serverul rulează sub un debugger, din cauza dificultăților apărute la depanarea aplicațiilor care rulează în mai mult decât un proces.

Configurarea acestui server se face printr-un fișier sursă Python prezent la `.py/init.py` (un exemplu de astfel de fișier este redat în secțiunea 3.1) relativ la directorul rădăcină a site-ului. Față de formatele tradiționale de configurare prin fișiere text, fișierele executabile au următoarele avantaje:

- Câmpurile care pot fi modificate în fișierul de configurare sunt declarate, alături, de obicei, de tipurile de date ale acestora, într-un fișier sursă, unde pot fi găsite în mod ușor. În plus, aceste fișiere pot fi procesate automat de către editoarele de cod sursă pentru a oferi completare a codului și documentație în procesul de elaborare a fișierului de configurare.
- Configurarea se efectuează în același limbaj cu cel în care se scrie aplicația.
- Valorile configurate pot fi obținute din orice surse, evitând adăugarea de complexitate limbajului de configurare prin instrucțiuni de citire din fișiere, variabile de mediu, sau alte surse.

[3] Potrivit secţiunii 6.3 al RFC 7230 al IETF[1], o conexiune TCP nu este închisă după ce răspunsul este trimis, cu excepţia situaţiilor de *timeout* sau dacă câmpul de antet HTTP `Connection` are valoarea `close`.

- Informațiile sensibile (de exemplu chei, credențiale, sau certificate) care sunt configurate pot fi obținute din exterior, fără a crea riscul ca acestea să fie incluse în mod neintenționat la o publicare a proiectului.
- Instrucțiunile condiționale ale limbajului de programare folosit permit modificarea configurării în funcție de parametri din mediul de execuție, fără necesitatea de a menține mai multe fișiere de configurare.
- Instrucțiunile repetitive ale limbajului de programare permit configurarea valorilor multiple, de exemplu nume de fișiere, în mod programatic, fără riscul ca fișierul de configurare să nu fie la curent cu modificările survenite ulterior scrierii acestuia, sau adăugarea de complexitate limbajului de configurare.
- Procesarea fișierului de configurare presupune doar executarea acestuia.

Inițializarea mediului de execuție Python este compusă din inițializarea bibliotecii Python și a diverselor module expuse de către server în interiorul acestui mediu de execuție. În continuare, se execută fișierul de configurare (rezultatele fiind disponibile în memoria mediului de execuție) și se înregistrează funcția de compilare în mecanismul de import al modulelor în mediul de execuție Python.

Administrarea serverului. După inițializarea mediului de execuție al paginilor dinamice, se creează conexiunile TCP in mod *LISTENING*, după care se așteaptă conexiuni noi pe acestea. La acceptarea unei noi conexiuni, se execută un fork, creându-se procesul care va servi acel client. În continuare, într-o buclă, până clientul se deconectează sau până conexiunea trebuie închisă (în caz de eroare), serverul citește o cerere, construiește un răspuns și-l trimite clientului. La finalul acestei bucle, procesul descendent se închide. Din cauza faptului că toată inițializarea se face înainte de fork, serverul citește și răspunde cererilor intr-un interval mic de timp.

Mecanismul de închidere ordonată (*graceful shutdown*) presupune așteptarea ca toate conexunile să fie închise corect, după ce toate cererile HTTP în așteptare au fost executate. În această lucrare, mecanismul este implementat cu ajutorul semnalelor Unix, trimise procesului părinte și propagate în procesele descendente. Astfel, un semnal `SIGUSR2` setează un câmp de tip boolean care previne acceptarea conexiunilor noi în procesul părinte, iar, în procesele descendente, previne citirea unor cereri noi. Astfel, procesele descendent se vor termina normal, fie imediat, fie la finalul cererii HTTP în curs de execuție. După închiderea tuturor proceselor descendente, procesul părinte se inchide. În cazul în care se dorește o închidere mai rapidă, semnalele `SIGUSR1` și `SIGINT` închid imediat conexiunile TCP deschise în procesul curent, procesul părinte așteptând apoi ca procesele descendente să-și închidă conexiunile. Nu se finalizează citirea unei cereri sau scrierea unui răspuns. Pentru o oprire forțată a serverului, semnalul `SIGTERM` închide imediat și necondiționat toate procesele, după ce conexiunile TCP au fost închise corect. Pe lângă trimiterea directă de semnale către procesul părinte, există un proces separat care citește comenzi de la un *fifo* Unix, disponibil la `/run/krait-cmdr` și trimite semnale procesului părinte. Comanda `^X` trimite un semnal `SIGINT` iar comanda `^K` trimite un semnal `SIGTERM`.

Fișierele de jurnalizare pot fi scrise fie la ieșirile implicite *stdout* și *stderr*, fie în fișiere specificate la linia de comandă. În cel de-al doilea caz, scrierea acestor fișiere este efectuată de către un proces dedicat de jurnalizare, conectat la fiecare din procesele serverului printr-un pipe Unix pentru fiecare ieșire (stdout și stderr), duplicate automat de către sistemul de operare la fiecare fork.

Informațiile scrise în situații normale în fișierele de jurnalizare conțin mesaje în momentul când un client se conectează sau deconectează, atunci când este decodată o cerere HTTP (conținând metoda HTTP și URL-ul cererii), și orice mesaj afișat din scripturile la nivel de server. În plus, în fișierul de jurnalizare în caz de eroare vor apărea eventualele excepții rezultate în urma execuției (fie erori în server, fie în aplicația Web) și alte situații excepționale. În acest moment nu se menține un jurnal complet al tuturor cererilor HTTP, dar, datorită implementării modularizate a serverului, această funcționalitate nu este greu de adăugat.

Procedeul de servire a cererilor este redat în figura 1.

Comunicarea cu clienţii HTTP. Pentru a asigura autentificarea și confidențialitatea comunicării dintre server și client, această lucrare poate răspunde și la cereri transmise folosind protocoalele SSL sau TLS. Pentru aceasta, serverul folosește biblioteca OpenSSL, dar este implementat pentru a putea fi adăugat ușor suport pentru alte biblioteci similare, ca mbed TLS[4] (anterior PolarSSL) sau GUARD TLS Toolkit (anterior MatrixSSL)[5]. Certificatul SSL, cheia privată și parola necesară pentru decriptarea acesteia sunt configurate în procesul de pornire a serverului, prin fișierul `.py/init.py` menționat mai sus. Administrarea conexiunii SSL/TLS, criptarea și decriptarea mesajelor se realizează în acest modul, restul serverului funcționând identic indiferent dacă această conexiune este sau nu protejată criptografic.

Citirea unei cereri HTTP se face impunând o limită atât pe durata de timp până se primește primul octet al acesteia cât și pe duratele ulterioare când se așteaptă primirea restului cereii.

Analiza (*parsarea*) unei cereri HTTP se face cu ajutorul unei instanțe a unui automat complex, determinist, cu un număr finit de stări tradiționale, dar în care tranzițiile pot avea condiții sau acțiuni arbitrare. Astfel, automatul folosește un spațiu de stare infinit. Atât specificația automatului, cât și instanțele acestuia folosite pentru parsările necesare acestui server sunt detaliate în secțiunea 5.

Construirea răspunsului. După citirea și analiza unei cereri HTTP, este necesară *rutarea* acesteia către un fișier sau către un controller (din Model-View-Controller). Acest proces se realizează cu ajutorul unei liste de rute, obținută în procesul de inițializare, configurată în fișierul `.py/init.py`.

Rutele sunt compuse dintr-o metodă (valori posibile sunt o metodă HTTP, “ANY” semnificând orice metodă HTTP, sau “WEBSOCKET” pentru o conexiune de tip *Upgrade: websocket*), un URL prestabilit, o expresie regulată pentru

[4] `https://tls.mbed.org/`

[5] `https://www.insidesecure.com/Products/Data-Communication/Secure-Communication-Toolkits/GUARD-TLS-TK`

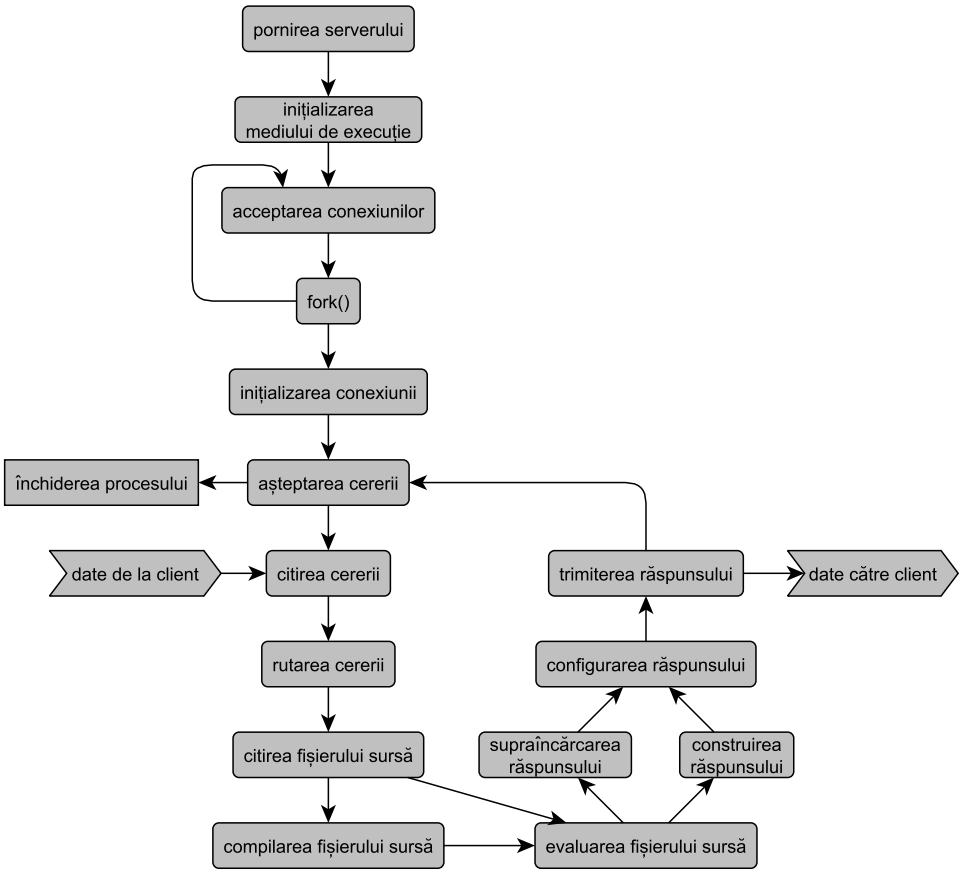

Figura 1. Diagramă generală a arhitecturii serverului HTTP şi a servirii unei cereri

URL, un fișier sursă pentru răspuns și o clasă de controller MVC sursă pentru răspuns. Oricare din aceste câmpuri poate lipsi. Dacă lipsește metoda HTTP, se presupune "GET"; dacă metoda HTTP din cerere nu este egală cu cea din rută, regula nu poate fi potrivită. Dacă lipsesc ambele reguli pentru filtrarea URL-urilor, regula va fi potrivită indiferent de acesta. Dacă lipsesc ambele surse pentru răspuns, numele fișierului sursă va fi obținut considerând URL-ul o cale relativă în directorul rădăcină al site-ului.

Prima rută care îndeplinește toate condițiile din lista de rute va fi selectată pentru a stabili sursa din care se va obține răspunsul. Dacă nu se va găsi nici o rută potrivită, răspunsul va fi "404 Not Found".

Această implementare oferă posibilitatea de a stoca fișiere inaccesibile pentru clienți, de exemplu certificate TLS, chei private, baze de date, sau scripturi la nivel de server în directoare descendente directorului rădăcină. Astfel, dacă în urma procesului de rutare a unei cereri, calea fișierul sursă pentru răspuns, relativă la directorul rădăcină, conține o componentă (numele fișierului sau un director ascendent acestuia) a cărui nume începe cu caracterul ".", răspunsul la această cerere va fi "404 Not Found". De exmeplu, dacă o bază de date este stocată într-un director numit ".db", un client nu ar putea să o acceseze.

În continuare, se salvează cererea HTTP în mediul de execuție Python pentru ca paginile dinamice să aibă acces la aceasta. Apoi, in cazul în care sursa răspunsului este un fișier, acest fișier este citit, parsat după regulile limbajului fișierelor de tip "machetă de vizualizare" (*template*), compilat într-un fișier sursă Python, care este apoi importat și rulat. In cazul în care sursa răspunsului este o clasă de controller MVC, se creează o nouă instanță a acestei clase și i se cere calea către fișierul în care este definit view-ul acesteia. Acest fișier va fi apoi procesat și executat la fel ca în cazul în care sursa răspunsului este un fișier.

La final, dacă pagina Web dinamică a suprascris răspunsul HTTP (de exemplu, pantru a trimite un răspuns de tip "400 Bad Request"), este trimis acel răspuns, altfel, este trimis răspunsul generat în urma executării fișierelor de *template*.

Pentru construirea eficientă a răspunsurilor s-au implementat următoarele optimizări:

- Dacă sursa paginii este un fișier static (doar fișierele cu extensia ".html", ".py" sau extensia proprie ".pyml" pot avea cod Python executabil; chiar pentru acestea, se verifică dacă există cu elemente dinamice, altfel fișierul este considerat static), nu se configurează mediul Python și fișierul nu este compilat, ci răspunsul este generat direct din conținutul fișierului. Astfel, aceste cereri primesc răspuns fără să fie executat cod Python.
- Krait suportă mecanismele de caching din HTTP prin o durată de valabilitate a unui răspuns HTTP (durată în care acesta nu va mai fi cerut de către client), și *entity tags*[6]. Atât durata de valabilitate cât și fișierele cărora se va aplica aceasta sunt configurabile. În plus, fiecare răspuns care poate fi

[6] *Entity tags* reprezintă un mecanism de etichetare a versiunii unei entităţi HTTP (conţinutul unui răspuns), acestea prevenind descărcarea repetată a unui răspuns complet o anumită perioadă de timp, şi după expirarea acesteia, dacă nu au survenit

salvat de clienți într-un cache este etichetat cu un entity tag. Atunci când clientul include un entity tag într-o cerere egal cu cel curent, se economisește retrimiterea acestui fișier. Implicit, paginile dinamice nu primesc un entity tag. Această valoare se calculează din *inode*-ul fișierului, dimensiunea sa și momentul ultimei modificări.
- După ce fișierele sursă sunt citite și parsate potrivit cu limbajul de *template*, rezultatul acestei analize (un arbore de parsare, cu informații sintactice și semantice), alături de entity tag, este salvat într-un cache, în memoria RAM. La cererile viitoare pentru același fișiere sursă, acest tag va fi recalculat (fără a citi fișierul) și va fi găsit în acest cache, economisindu-se timpul de citire și parsare. Acest cache este întreținut și în procesul părinte, acesta fiind notificat printr-un pipe atunci când se modifică cache-ul unui proces descendent.
- Fișierele sursă dinamice nu sunt executate direct, din cauza faptului că această operațiune s-ar realiza printr-un mare număr de apeluri similare cu instrucțiunea `eval` din limbalul Python. Această operațiune este costisitoare din punct de vedere al performanței, deoarece codul Python trebuie parsat, compilat in bytecode, și apoi executat. Astfel, aborarea noastră creează un fișier modul Python din fiecare fișier *template*, care este apoi importat (această instrucțiune menține codul compilat în memoria RAM), și rulat prin apelarea funcției `run`, creată de compilator.
- Se evită pe cât posibil copierea șirurilor de caractere în program. Astfel, oriunde a fost posibil, s-au trimis ca argumente referințe către datele unui șir de caractere, fără ca acestea să fie copiate.

Procesarea şi execuţia fişierelor sursă. Pentru a fi trimise către un client, fișierele sursă pentru un răspuns (fie fișiere obținute prin aplicarea unei rute la URL-ul unei cereri HTTP, fie view-uri din paradigma Model-View-Controller), acestea trebuie parsate conform sintaxei limbajului de *template*. Această sarcină se realizează cu ajutorul unei instanțe al parserului implementat pentru acest proiect (detaliat în secțiunea 5, utilizat și în analiza textuală a a cererii HTTP.

Fișierul este citit și consumat, caracter cu caracter, de către parser. Pe măsură ce se efectuează tranzițiile, acestea creează un arbore cu informații sintactice și semantice. Acest arbore poate fi ulterior evaluat pentru a produce un răspuns HTTP (adăugând nodurile statice ale arborelui la răspuns și rulând nodurile dinamice ale acestuia) sau compilat într-un fișier sursă Python (convertind nodurile statice ale arborelui în apeluri de adăugare de date la răspuns și transformând nodurile dinamice în instrucțiuni Python). Din cauza superiorității compilării față de evaluare, cea din urmă este folosită doar pentru pagini statice.

Arborele este apoi salvat în cache-ul serverului pentru a fi imediat disponibil la o cerere ulterioară pentru același fișier. Înainte de a fi folosit un arbore salvat în cache, se verifică (cu același algorithm pentru obținerea entity tag-ului) dacă

---

modificări de la ultima accesare. Acest mecanism este explicat în secţiunile 4.3.1 şi 4.3.2 a RFC 7234 al IETF[2].

fișierul a fost modificat de la ultima citire a acestuia. În caz contrar, se omite citirea și parsarea a acestuia, răspunsul fiind generat din arborele salvat.

Compilarea fişierelor sursă în cod Python. Acest modul transformă un fișier *template* într-un fișier sursă Python, care poate fi executat pentru a produce un răspuns într-un timp mai scurt.

Pentru a permite menținerea fișierului rezultat, în memoria RAM, compilat în bytecode Python, acesta este folosit ca un modul Python. Din această cauză, s-a optat folosirea mecanismului de importare din acest limbaj de programare, adăugându-se un *import hook*[7] care verifică existența unui modul Python deja compilat, iar în caz contrar găsește fișierul care trebuie compilat și declanșează compilarea acestuia.

Implementarea acestui modul este descrisă în secțiunea 6.

Serverul WebSocket. În momentul în care serverul HTTP primește o nouă cerere de tip "`Upgrade: websocket`" (după regulile menționate în secțiunea 1.3) și în procesul de rutare se potrivește o rută de tip "WEBSOCKET", execuția intră într-un modul separat, dedicat servirii de mesaje prin protocolul WebSocket.

Întâi, se stabilește conexiunea WebSocket în acord cu secțiunea 4 din specificația protocolului WebSocket, RFC 6455 al IETF[3]. Dacă acesta se finalizează cu succes, conexiunea este in mod WebSocket.

Controlul conexiunilor WebSocket se face prin intermediul unei interfețe de controller WebSocket, care este detaliată în secțiunea 3.4. Într-o buclă, se citesc și se trimit mesajele WebSocket.

Citirea se face cu un timp maxim de așteptare (*timeout*) de 8 milisecunde (aplicat la timpul până se primește primul octet și timpii când se așteaptă octeți ulteriori). Dacă este primit un mesaj de tip text sau binar, acesta este trimis către controller. La primirea unui mesaj Ping din protocolul WebSocket, se trimite un mesaj Pong, iar dacă se primește un mesaj de închidere, se confirmă închiderea conexiunii.

După ce se încearcă citirea unui mesaj WebSocket, se încearcă preluarea unui mesaj de trimis de la controller-ul conexiunii curente. Dacă există un mesaj, acesta va fi trimis.

Atunci când conexiunea este închisă de către client, controller-ul este notificat și este așteptat să se închidă cel mult un minut, după care procesul care servește această conexiune se inchide forțat.

[7] Un *import hook* este o metodă de a modifica procesul de găsire al unui modul Python la momentul unui import, şi poate fi folosită pentru a genera fişiere sursă Python din alte fişiere incompatibile.

3 Interfaţa de programare Python

Una din contribuțiile principale ale acestei lucrări este interfața de programare intuitivă și ușor de folosit. Întreaga interfață este extensiv documentată, și partea interfeței implementată în limbajul Python este facil de înțeles.

Această lucrare expune un pachet Python, numit "krait", în care un număr de funcții și variabile sunt expuse direct pe obiectul pachetului, iar alte funcționalități mai specifice sunt disponibile in modulele care fac parte din acest pachet. În procesul de inițializare al serverului HTTP, directorul ".py" din rădăcina siteului, dacă există, este adăugat la calea de rezolvare a importurilor de module Python. De exemplu, un fișier Python numit "database.py" care se află în acest director poate fi accesat prin instrucțiunea "import database".

Din cauza faptului că programarea scripturilor la nivel de server al unei aplicații Web se face într-un limbaj de programare cu scop general, aceste scripturi au acces complet la sistemul în care rulează această aplicație. Din acest motiv, este extrem de importantă securitatea interfeței de programare. În această lucrare, totuși, nu există nici o metodă prin care un client malițios să poată modifica codul Python care este executat la nivelul serverului. Singura sursă de unde se poate obține cod Python care să fie executat este din fișiere de pe disc, deci orice modalitate de a aduce schimbări codului Python al aplicației de pe server necesită drepturi de a scrie fișiere pe disc. Cu excepția situației când un script la nivel de server scrie fișiere care pot conține cod Python (cu extensiile ".html", ".py" sau ".pyml") în directoare accesibile pentru clienți, sau execută cod care provine din surse care nu sunt de încredere, este imposibil pentru un atacator să cauzeze execuție de cod Python pe server folosind această implementare.

3.1 Configurarea serverului

Configurarea serverului se face prin scrierea de către dezvoltatorul aplicației Web al unui fișier la calea `.py/init.py` relativă la directorul rădăcină al siteului. Acest fișier va fi executat o singură dată, în procesul părinte, ca parte a procedeului de inițializare a serverului.

Fișierul de configurație nu poate fi accesat de către clienții aplicației, din cauza faptului că se află în directoul `.py`, al cărui nume începe cu caracterul ".". Astfel, configurația serverului poate rămâne un secret din punctul de vedere al clienților aplicației Web.

În interiorul acestui fișier de configurație dezvoltatorul are posibilitatea de a modifica variabilele disponibile pe modulul `krait.config`.

La variabila "routes" al acestui modul trebuie să existe o listă de rute, fiecare rută fiind un obiect pe care se găsesc atributele specificate în secțiunea 2.1. O serie de variabile aleg fișierele care pot fi păstrate în cache-urile clienților, în cache-urile publice, sau în nici un cache, și durata de expirare a acestor fișiere în cache-uri. În plus, există posibilitatea de cere ca o submulțime din aceste fișiere să fie păstrate în cache-uri pe termen lung, și a specifica durata de expirare în acest caz. În final, dezvoltatorul poate configura certificatul TLS/SSL și cheia

privată a acestuia, alături de parola necesară pentru decriptarea acestei chei private.

Configurarea rutelor pentru controller-ele MVC va fi explicată în secțiunea 3.3.

Exemplu de configurare a serverului HTTP

```
# Se adauga rutele MVC.
# Aceasta configurare este explicata mai jos.
mvc.import_ctrls_from("ctrl")

# Se adauga rutele non-MVC.
# Ultima ruta va fi folosita in principal de fisierele
# statice, dar se aplica doar cererilor GET.
config.routes.extend([
    config.Route("POST", url="/comment"),
    config.Route("WEBSOCKET", url="/ws_socket"),
    config.Route()
])

# Permite fisierele cu extensia .js, .css si .map sa fie
# salvate in cache-urile clientilor si proxy-urilor.
config.cache_public = [".*\.js", ".*\.css", ".*\.map"]

# Fisierele cu extensia .js si .css pot fi mentinute in
# cache-uri o durata mai lunga de timp.
config.cache_long_term = [".*\.js", ".*\.css"]

# Se aleg duratele de expirare a raspunsurilor in cache-uri.
# Duratele sunt exprimate in secunde.
config.cache_max_age_default = 6 * 60  # 6 minute
config.cache_max_age_long_term = 24 * 60 * 60  # 24 de ore

# Se seteaza certificatul TLS/SSL si cheia privata
config.ssl_certificate_path = os.path.join(
    krait.site_root, ".private", "ssl", "cert.pem")
config.ssl_private_key_path = os.path.join(
    krait.site_root, ".private", "ssl", "key.pem")
# Parola cheii private se citeste dintr-un fisier.
with open(os.path.join(
        krait.site_root, ".private",
        "ssl", "pk_passphrase")) as f:
    config.ssl_key_passphrase = f.read()
```

Fragmente din fișierul de configurare al aplicației folosită la testarea lucrării

3.2 Accesarea cererii şi modificarea răspunsului

Majoritatea variabilelor și funcțiilor folosite de scripturile la nivel de server sunt configurabile prin variabilele expuse direct pe obiectul pachetului `krait`.

Crerea HTTP care este servită în momentul curent de către scriptul care rulează este accesibilă în variabila `krait.request`. Această variabilă este un obiect care conține metoda HTTP, URL-ul cererii, *query string*-ul[8], versiunea protocolului HTTP (doar "HTTP/1.1"), câmpurile de antet ale cererii, și conținutul acesteia. O serie de metode prezente pe acest obiect analizează conținutul cererii ca un formular tip POST (un formular cu tipul MIME "x-www-form-urlencoded"[9]), sau tip *multipart*[10].

Răspunsul HTTP este de obicei obținut prin evaluarea fișierelor *template*, din care să rezulte o pagină Web. În cazul în care se dorește suprascrierea acestui răspuns, dezvoltatorul aplicației Web poate atribui variabilei `krait.response` un obiect de tipul `krait.Response` sau unul din tipurile care moștenesc din acesta. Câmpurile configurabile pe aceste obiecte sunt versiunea HTTP (obligatoriu "HTTP/1.1"), codul de stare, câmpurile de antet, și conținutul răspunsului. Dacă nu se dorește suprascrierea întregului răspuns, ci doar adăugarea de câmpuri de antet la răspunsul implicit, se poate modifica variabila
`krait.extra_headers`, iar tipul MIME al răspunsului se configurează cu ajutorul funcției
`krait.set_content_type`.

O altă variabilă utilă disponibilă pe pachetul `krait` este "site_root", setată de serverul HTTP în procesul de inițializare și conținând calea către directorul rădăcină al aplicației Web, iar funcția `krait.get_full_path` transformă o cale relativă față de acest director în calea absolută corespunzătoare. Aceste funcții ale interfeței de programare sunt utile dacă se dorește accesul la fișiere aflate în aceste directoare din scripturile la nivel de server. La final, funcția `krait.set_content_type` este utilă atunci când această lucrare nu poate deduce tipul MIME corect al răspunsului HTTP din extensia fișierului sursă, fie deoarece extensia acestuia nu reflectă tipul MIME dorit, fie deoarece sursa răspunsului este o clasă de controller MVC, și nu un fișier.

Această lucrare expune ca parte a interfeței de programare modulul
`krait.cookie`, care permite citirea și modificarea *cookie*-urilor din protocolul HTTP. Funcția `krait.cookie.get_cookies` întoarce o listă de obiecte care reprezintă câte un cookie, pe care se pot citi numele și valoarile acestora. Pentru a trimite cookie-uri noi către clienții HTTP se poate folosi funcția `krait.cookie.set_cookie`, dezvoltatorii aplicației Web putând alege numele, valoarea și o listă de atribute ale acestora. Semnificația diferitelor atribute care pot fi adăugate cu ajutorul

[8] În reprezentarea internă a unei cerei HTTP, URL-ul din cerere este separat în calea din cerere, care este partea din URL de înainte de caracterul "?", şi *query string*-ul, care este partea de după acest caracter.

[9] Tipul MIME "x-www-form-urlencoded" specifică un formular Web codat după secţiunea 5 a standardului URL[6].

[10] Codarea unui formular cu tipul MIME "multipart/form-data" este specificată în RFC 7578 al IETF[4].

funcțiilor disponibile în acest modul este definită în secțiunea 4.1.2 din RFC 6265 al IETF[5].

Exemplu de accesare a cererii, modificare a răspunsului, și utilizare a modulului `krait.cookie`

```
# Aceasta pagina este destinatia unui formular POST
# (tip MIME x-www-form-urlencoded)
# Obtinem formularul din cererea HTTP
post_form = krait.request.get_post_form()
name = post_form["name"]
message = post_form["text"]

# Suprascriem raspunsul HTTP cu un raspuns de tip 302 Found
krait.response = krait.ResponseRedirect("/db")

# Obtinem valoarea cookie-ului "comment_count"
# (daca nu exista, valoarea implicita este 0)
comment_count = int(krait.cookie.get_cookie(
    "comment_count", "0"))
# Cream noul cookie
new_cookie = krait.cookie.Cookie("comment_count",
    str(comment_count + 1))
# Setam atributele Expires si HttpOnly.
new_cookie.set_expires(datetime.datetime.utcnow()
    + datetime.timedelta(minutes=1))
new_cookie.set_http_only(True)

# Salvam cookie-ul ca sa fie trimis catre client
krait.cookie.set_cookie(new_cookie)
```

Fragment din fișierul "comment.py", parte a aplicației folosită pentru testarea lucrării

3.3 Interfaţa pentru Model-View-Controller

Paginile Web care folosesc paradigma MVC (definită original în 1979 de către Trygve Reenskaug, o copie a notei sale fiind redată la [10]) se creează prin definirea unei clase controller, rutarea unui URL la acea clasă și scrierea unui view, care este un fișier *template*. Avantajul principal al acestui mod de a scrie pagini Web este decuplarea procesării datelor de prezentarea acestora, ușurând modificarea acestora, facilitând implementarea mai multor moduri de prezentare pentru aceeași sursă de date (de exemplu, un singur URL poate fi folosit pentru servirea rezultatelor în formatul HTML, JSON sau XML în funcție de valoarea câmpului de antet "Accept", dacă se creează mai multe view-uri pentru același controller), sau a mai multor surse de date pentru același mod de prezentare (de

exemplu, în procesul de testare se poate folosi un controller care nu afectează baza de date, dar view-ul poate rămâne același).

Un controller MVC este un obiect Python cu o metodă numită "get_view". Această metodă este apelată de această lucrare pentru a obține calea (relativă la directorul rădăcină) unde se află fișierul *template*, care va fi folosit ca view pentru această cerere HTTP. Clasa `krait.mvc.CtrlBase` este un exemplu de controller MVC și poate fi folosit ca superclasă abstractă pentru controller-e MVC.

Pentru a ruta un URL la un controller, dezvoltatorul aplicației Web poate importa modulul în care se află clasa controller și apoi adăuga o rută cu URL-ul și clasa de controller dorite în lista de rute în fisierul de configurare. O soluție alternativă este decoratorul `krait.mvc.route_ctrl_decorator` care, aplicat unei clase, adaugă o rută cu parametrii specificați. Dezavantajul acestei metode este că nu se evită importarea manuală a modulelor în care sunt implementate controllerele. Pentru a rezolva această problemă, modulul `krait.mvc` expune funcția "import_ctrls_from" care importă recursiv toate modulele Python dintr-un pachet. La momentul importului, decoratorul va fi executat, lucru care va adăuga controller-ul specificat la lista de rute.

Rutele adăugate de metodele descrise mai sus specifică o clasă de controllere. La momentul rutării, dacă ruta potrivită este o rută MVC, se va instanția un nou obiect aparținând acestei clase și se va evalua view-ul returnat de metoda "get_view" al acestuia.

View-urile sunt fișiere normale scrise în limbajul de *template*. Pentru a nu fi accesibile direct de către clienți, acestea trebuie să se afle într-un director care începe cu caracterul "." (de exemplu ".view") sau un descendent al acestuia. În mediul de execuție al unui view, variabilei "ctrl" îi este atribuit obiectul controller curent, iar view-ul are acces la atributele acestuia. Astfel, modalitatea de a transmite imformații de la controller către view, pentru a fi prezentate clientului, este prin expunerea de atribute pe obiectul controller. În final, în view-uri este necesar apelul funcției `krait.set\_content\_type` pentru a seta tipul MIME al răspunsului, deoarece serverul HTTP poate deduce tipul MIME doar din extensia unui fișier. Această funcție poate accepta și o extensie de fișier, care va fi transformată într-un tip MIME de către server.

Exemplu de controller MVC

```
# Acest decorator ruteaza URL-ul "/db" la clasa DbController
@krait.mvc.route_ctrl_decorator(url="/db")
class DbController(krait.mvc.CtrlBase):
    def __init__(self):
        super(DbController, self).__init__()

        # Setam atribute accesibile din view
        self.page_url = "/db"
        self.messages = None
        self.load_db(global_config.sqlite_db)
        self.nr_comments = int(krait.cookie.get_cookie(
```

```
            "comment_count", 0))

    def load_db(self, db_filename):
        # Atributele se pot seta si din exteriorul
        # functiei __init__.
        conn = sqlite3.connect(db_filename)
        c = conn.cursor()

        c.execute("select␣*␣from␣messages")
        self.messages = c.fetchall()

        conn.close()

    def get_view(self):
        # Metoda obligatorie, specifica view-ul
        # care va fi folosit pentru generarea raspunsului
        return ".view/db.html"
```

Fragment din fișierul ".py/ctrl/db.py", parte a aplicației folosită pentru testarea lucrării

Acest controller MVC se află într-un fișier Python situat în directorul "ctrl". Astfel, în exemplul de fișier de configurare prezentat în secțiunea 3.1, apelul `mvc.import_ctrls_from("ctrl")` va importa și acest fișier, care se va adăuga singur, prin decorator, la lista de rute.

Exemplu de view MVC

```
<!DOCTYPE html>
<html lang="en">

<!-- Parte din fisier omisa -->

<!--
    Se observa sintaxa limbajului de template,
    secventa @for marcand inceputul unui bloc repetat
    de cod HTML.
    In plus, se observa colectia enumerata de instructiunea
    @for, ctrl.messages, accesand proprietatea setata
    in functia DbController.load_db.
    Sintaxa acestui limbaj va fi explicata
    mai jos.
-->
@for name, message in ctrl.messages:
    <div class="panel␣panel-default␣comment-item">
        <div class="panel-heading␣comment-name">
            <!--
```

```
                Se observa sintaxa de afisare a unei
                variabile in cod de template, @variabila.
            -->
            <p>@name says:</p>
        </div>
        <div class="panel-body␣comment-text">
            <!--
                Aici se foloseste varianta a doua a sintaxei
                de mai sus, @variabila@.
            -->
            <p>@message@</p>
        </div>
    </div>
@/for

<!-- Parte din fisier omisa -->
</html>
```

Fragment din fișierul ".view/db.html", cerut ca view de controller-ul de mai sus

3.4 Interfaţa pentru WebSocket

Controlul conexiunilor WebSocket prezintă o problemă semnificativă, și anume că modelul conexiunii nemaifiind de tip client-server, ci bidirecțională, este nevoie de un mecanism prin care aplicația Web să anunțe serverul de dorința de a trimite un mesaj WebSocket, cerință care necesită rularea continuă a codului aplicației Web. Astfel, aplicația este întrebată periodic, cu o frecvență mare, dacă și ce mesaj să trimită, iar aceasta poate rula componenta de control pe un fir separat de execuție, comunicând cu firul principal de execuție prin două cozi de mesaje, una pentru mesajele WebSocket recepționate de la client, și una pentru mesajele care se dorește a fi trimise către acesta. Totuși, nu este necesară folosirea acesturi fir de execuție, deoarece aplicațiile cu necesități mari de performanță și care nu necesită controlul complet al mesajelor trimise pot alege să trateze mesajele într-o manieră bazată pe evenimente, rulând procesarea necesară în interiorul funcțiilor apelate de către server la primirea unui mesaj sau la interogarea pentru mesaje care să fie trimise, după explicația de mai sus.

Serverul expune modulul `krait.websockets` pentru implementarea aplicațiilor Web care folosesc conexiuni WebSocket. Cel mai important membru al acestui modul este `krait.websockets.WebsocketsCtrlBase`, o clasă de bază care este moștenită de către controller-ele WebSocket.

Există două moduri de a moșteni această clasă, determinate de valoarea argumentului "use_in_message_queue" al apelului către constructorul clasei părinte (WebsocketsCtrlBase):

– Dacă acest argument are valoarea `True`, atunci este necesară supraîncărcarea metodei "on_thread_start", conținutul acesteia fiind codul responsabil de

implementarea WebSocket al aplicației, care va rula într-un fir de execuție separat. Acest fir de execuție poate folosi metodele "pop_in_message" și "push_out_message" pentru a citi, respectiv scrie mesaje WebSocket. Acest fir de execuție trebuie să urmărească valoarea întoarsă de metoda "should_stop" și să se închidă dacă aceasta întoarce valoarea `True`.
– În cazul în care valoarea argumentului este `False`, atunci este necesară supraîncărcarea metodei "on_in_message", metodă care va fi apelată la primirea unui mesaj WebSocket de la client. Opțional, se poate supraîncărca și metoda "on_thread_start", care va rula într-un fir separat de execuție, dar, de multe ori, modelul bazat pe evenimente implemecntatu ajutorul funcției "on_in_message" face ca firul adițional de execuție să nu fie necesar și să se închidă automat. Scrierea mesajelor se face fie prin apelarea "push_out_message", fie prin suprascrierea metodei "pop_in_message", care este apelată regulat de către server pentru a obține mesajul care trebuie trimis, dacă acesta există. Ca în primul caz, firul de execuție, dacă este folosit, trebuie să urmărească valoarea întoarsă de metoda "should_stop".

Pentru a ruta un URL la un controller WebSocket este nevoie de un script la nivel de server care să pregătească procesul de stabilire a conexiunii WebSocket. Acest script citește valoarea `krait.websockets.request`, verifică să nu fie `None` (în caz contrar, acesta nu este un request WebSocket), și alege un subprotocol dintr-o listă disponibilă pe acest obiect. Apoi, acest script atribuie variabilei `krait.websocket.response` un obiect care specifică subprotocolul ales și o instanță a clasei de controller WebSocket ales. În final, în fișierul de configurare a serverului trebuie adăugată o rută pentru metoda "WEBSOCKET", de la URL-ul dorit la acest fișier.

Exemplu de controller WebSocket cu fir de execuție separat

```
# Aceasta clasa trimite si primeste mesaje PING urmate de
# un sir de caractere aleator. Cand fie aceasta fie clientul
# primesc un mesaj PING, raspund cu un mesaj PONG
# cu acelasi sir de caractere primit.
class WsPingpongController(websockets.WebsocketsCtrlBase):
def __init__(self):
    # Dorim folosirea cozilor de mesaje, deci argumentul
    # de mai jos este True
    super(WsPingpongController, self).__init__(True)

def on_thread_start(self):
    rand_chars = string.ascii_letters + string.digits

    # Firul de executie trebuie sa se inchida cand metoda
    # should_stop intoarce valoarea True
    while not self.should_stop():
        # Citim mesajele de la client
```

```
        in_msg = self.pop_in_message()

        # Cand pop_in_message intoarce None, nu exista
        # nici un mesaj
        if in_msg is not None and\
                not in_msg.startswith("PONG"):
            # Trimitem mesajul PONG
            self.push_out_message("PONG: " + in_msg)

        if random.randrange(0, 50) == 0:
            ping_str = "".join(random.choice(rand_chars)
                for _ in range(random.randrange(5, 10)))

            # Trimitem mesajul PING
            self.push_out_message("PING: " + ping_str)

        # Asteptam timp de 100 de milisecunde pentru
        # a preveni utilizarea inutila a resurselor
        time.sleep(0.1)

    print "Closing..."
```

Fragment din fișierul ".py/ctrl/pingpong_ws.py", parte a aplicației de testare a lucrării

Un exemplu de controller WebSocket fără fir de execuție adițional se poate găsi în cel de-al doilea studiu de caz din secțiunea 8.2.

Exemplu de fișier de configurare al conexiunii WebSocket

```
# Verificam ca acesta sa fie un request WebSocket
if krait.websockets.request is None:
    # Daca nu, raspunsul HTTP va fi 400 Bad Request
    krait.response = krait.ResponseBadRequest()
else:
    protocols = krait.websockets.request.protocols
    # Singurul protocol suportat este "pingpong"
    if "pingpong" in protocols:
        # Alegem controllerul si protocolul
        krait.websockets.response =\
            krait.websockets.WebsocketsResponse(
                pingpong_ws.WsPingpongController(),
                "pingpong")
    else:
        # Daca clientul nu suporta protocolul "pingpong",
        # raspundem cu 400 Bad Request
        krait.response = krait.ResponseBadRequest()
```

Framgent din fișierul "ws_socket.py", parte a aplicației de testare a lucrării

În final, fișierul este adăugat la lista de rute din exemplul de fișier de configurare al serverului din secțiunea 3.1, prin adăugarea obiectului `config.Route ("WEBSOCKET", url="/wssocket")` la lista `config.routes`.

3.5 Documentaţia interfeţei de programare

Documentația interfeței de Python pentru dezvoltarea aplicațiilor Web cu ajutorul acestui server este scrisă în *docstrings*[11]. Acestea sunt segmente de documentație incluse în codul sursă (la începutul claselor și funcțiilor și imediat după variabilele globale), care specifică tipurile de date și o descriere, și informații despre utilizare pentru clase, funcții, variabile, câmpuri ale claselor, argumente ale funcțiilor și valorile întoarse de către acestea. Formatul docstring-urilor nu este specificat și depinde de instrumentul aleasă pentru generarea documentației.

Documentația pentru această lucrare este scrisă în stilul Google, care este definit la `https://github.com/google/styleguide/blob/gh-pages/pyguide.md#38-comments-and-docstrings` și, folosind o sintaxă ușor de citit, permite transmiterea de informații detaliate, inclusiv despre tipurile de date, pentru elementele documentate. Documentația este generată cu instrumentul Sphinx[12], și publicată pe site-ul Read The Docs[13], la adresa `https://krait.readthedocs.io`.

Documentația proiectului include:

- O pagină de introducere cu o prezentare sumară a proiectului
- Funcțiile, clasele și variabilele prezente pe obiectul pachetului Krait și pe modulele acestuia (`krait.config`, `krait.cookie`, `krait.mvc`, `krait.websockets`)
- Câte o descriere succintă pentru fiecare modul
- Câte un scurt ghid de utilizare pentru configurarea serverului, accesul și trimiterea de cookie-uri, utilizarea paradigmei Model-View-Controller, și folosirea protocolului WebSocket.

Cu excepția paginii de introducere, întreaga documentație este integrată în codul Python, și astfel modificările interfeței de programare (de exemplu adăugarea sau ștergerea câmpurilor) se actualizează automat în documentație.

---

[11] *Docstring*-urile sunt specificate în PEP 257 (`https://www.python.org/dev/peps/pep-0257/`)

[12] Sphinx este un instrument de generare a documentaţiei din fişiere sursă Python. Site-ul proiectului se găseşte la adresa `https://www.sphinx-doc.org`.

[13] Read The Docs este un proiect care găzduieşte documentaţie pentru proiectele open-source. Site-ul proiectului se găseşte la adresa `https://readthedocs.org`.

4 Limbajul fişierelor de tip "machetă de vizualizare" (*template*)

Această lucrare folosește pentru paginile dinamice și pentru view-urile din paradigma Model-View-Controller un limbaj de *template* original numit *PyML*, întreaga specificație și implementare fiind proprii.

4.1 Specificaţia limbajului

În limbajul PyML, caracterul marcator care denotă elementele dinamice ale paginilor Web este caracterul "@". În funcție de caracterele care urmează acestui marcator, codul *template*-urilor este interpretat în diferite moduri:

- "`@{ … }`"[14] semnifică una sau mai multe instrucțiuni Python care, la evaluarea acestui element, vor fi executate fără să modifice conținutul răspunsului. Adâncimea perechilor "{" - "}" este contorizată, astfel încât acest bloc să nu se finalizeze până la echilibrarea numărului de caractere "{" și "}".
- "`@…@`" semnifică o expresie Python care va fi evaluată la momentul evaluării acestui element, iar rezultatul acestei expresii, transformată în șir de caractere, va apărea în conținutul răspunsului HTTP, după ce este sanitizat conform regulilor limbajului HTML (astfel încât, de exemplu, șirul de caractere "`<script>`" devine "`&lt;script&gt`", protejând aplicația Web de atacuri Cross Site Scripting sau alte atacuri similare[15]). Ultimul caracter poate fi înlocuit cu un caracter alb[16]. Caracterul de final ("@" sau caracter alb) este ignorat atunci când apare în șiruri de caractere sau între paranteze.
- "`@!…@`" se comportă similar elementului "`@…@`", dar sanitizarea conținutului este dezactivată. Acest comportament este util în cazul în care conținutul produs nu se află într-un fișier HTML (de exemplu, într-un răspuns în formatul JSON), dar presupune riscuri majore de securitate care trebuie rezolvate de către dezvoltatorii aplicațiilor Web.
- "`@if …:`" introduce un element dinamic condițional, terminat de o instrucțiune "`@/if`". La evaluarea acestui element se evaluează expresia Python specificată. Dacă rezultatul evaluării, transformat în tipul boolean, are valoarea `True`, blocul de elemente de pagină dintre aceste instrucțiuni va fi inclus în răspuns, altfel va fi omis.
- "`@else:`" introduce o alternativă pentru instrucțiunea condițională de mai sus. Această instrucțiune trebuie plasată între o pereche de instrucțiuni "`@if`" și "`@/if`" și delimitează secvența de elemente de pagină incluse în răspuns în cazul evaluării condiției la o valoare "`True`" de elementele incluse în caz contrar.

[14] În această secţiune, caracterul "…" ţine locul unei secvenţe de cod Python, iar interpretarea acesteia este explicată sau dedusă din context.

[15] Atacurile Cross Site Scripting(XSS) sunt descrise de Open Web Application Security Project (OWASP) la `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`.

[16] Se consideră caractere albe cele cu codul ASCII 9, 10, 13 sau 32.

- "`@for … in …:`" introduce un element dinamic repetitiv, terminat de o instrucțiune "`@/for`". În momentul evaluării acestei instrucțiuni, se evaluează a doua expresie, se începe o iterație, iar variabila sau variabilele din prima secvență de cod va primi pe rând fiecare valoare din această iterație. Acest comportament imită evaluarea instrucțiunii "`for`" din Python.
- "`@import …@`" permite includerea altui fișier *template* în răspunsul HTTP. La evaluarea elementului, se evaluează expresia Python specificată ca o cale relativă la directorul rădăcină al site-ului. Acest fișier este parsat ca fișier *template*, evaluat și inclus în răspunsul HTTP. Astfel, se pot scrie fragmente de pagină reutilizabile sau se poate implementa moștenirea view-urilor. (Fiindcă expresia Python este evaluată, nu este nevoie ca aceasta să fie statică, iar fragmentele de pagină pot fi configurate de controller-e sau view-uri descendente.) La fel ca în cazul instrucțiunilor de evaluare, caracterul de terminare "@" poate fi înlocuit cu un caracter alb.
- "`@import-ctrl …@`" permite includerea view-ului unui controller în răspunsul HTTP. Comportamentul este identic elementului "`@import …@`", cu excepția faptului că expresia Python trebuie să se evalueze la un controller MVC, a cărui metodă "get_view" va fi apelată pentru obținerea căii către fișierul *template*, care va fi evaluat cu variabila "ctrl" setată la controller-ul specificat. Astfel, view-urile pot include alte view-uri MVC cu controller-ele sale, sau își pot delega controller-ele să aleagă fragmentele de pagină incluse.
- "`@@`" introduce un caracter "@", lucru necesar din cauza modificării semnificației caracterului "@" în limbajul PyML.

Cu excepția secvenței "`@@`", toate elementele de mai sus sunt considerate elemente de pagină dinamice și necesită configurarea mediului de execuție și compilarea în cod Python.

O versiune anterioară a limbajului PyML a folosit ca secvențe de marcare a elementelor dinamice "`<! …!>`", "`<@ …@>`" și alte secvențe similare. Acest limbaj a fost înlocuit din cauza dificultăților întâmpinate de editoarele de text configurate pentru analiza sintactică a limbajului HTML. Astfel, unele editoare raportau erori cauzate de similaritatea dintre sintaxa PyML și tag-urile HTML. Prin comparație, limbajul curent, din punctul de vedere al sintaxei limbajului HTML, apare ca text simplu și nu este interpretat în mod greșit.

4.2 Surse de inspiraţie

Cea mai mare inspirație pentru limbajul fișierelor de *template* din această aplicație o reprezintă sintaxa Razor folosită în framework-ul de pagini Web ASP.NET. Avantajul acesteia este capabilitatea de a transmite vizual comportamentul paginii Web în momentul în care aceasta va fi evaluată într-un mod neintruziv. Astfel, limbajul PyML a preluat caracterul "@" ca marcator al părților dinamice, blocurile de cod executabil delimitate prin "`@…`" și secvența "`@@`" pentru introducerea unui caracter real "@".

O inspirație secundară pentru PyML este sintaxa limbajului de programare Python. Fiind unul din limbajele considerate cel mai ușor de citit și înțeles,

și fiind cel în care sunt evaluate componentele dinamice, inspirația din acest limbaj este o alegere evidentă. Elementele de sintaxă Python preluate în PyML sunt sintaxa instrucțiunilor “for”, “if”, “else” (finalul acestora fiind marcat prin caracterul “:”) și cuvântul cheie “import” folosit pentru includerea unui fișier separat în contextul curent. În plus, alegerea cuvintelor cheie și implementarea parserului au depins de specificația sintaxei Python, anumite conflicte aparente în specificație fiind rezolvate de regulile codului valid Python. De exemplu, “`@if`” nu poate fi considerat element de evaluare a codului Python pentru a fi inclus în răspuns, deoarece “`if`” nu este o expresie Python validă. Un alt exemplu este secvența “`@!…@`”, în care nu se poate considera expresie “`!…`”, din cauza faptului că expresiile Python nu pot începe cu caracterul “!”.

La final, limbajul HTML (și, prin extensie, XML și/sau SGML) a furnizat convenția finalizării blocurilor marcată de caracterul “/” urmat de numele blocului finalizat, convenție folosită în cuvintele cheie “`/if`” și “`/for`”.

4.3 Exemple de fişiere în limbajul de *template*

Exemplu de sintaxă al acestui limbaj

```
<!DOCTYPE html>
<html lang="en">
    <!-- Parte din fisier omisa -->
    <!-- Executie de cod Python. Se seteaza
         tipul de continut pentru a fi trimis spre client
    -->
    @{ krait.set_content_type(ext="html") }

    <!-- Parte din fisier omisa -->
    <!-- Antetul paginii Web este un fragment
         reutilizabil, importat in fiecare pagina
    -->
    @import ".view/header.html"

    <!-- Parte din fisier omisa -->
    <!-- Exemplu de afisare al unui sir de caractere
         in raspunsul HTTP
    -->
    <p>It's @ctrl.time_str already! </p>
    </div>

    <!-- Parte din fisier omisa -->
    <!-- Exemple de instructiune alternativa
         si repetitiva
    -->
    @if ctrl.alerts_exist:
        @for idx in range(1, 3):
```

```
            <div class="alert␣alert-info␣demo-alert-div">
            <p>This is the @(
                str(idx) + ctrl.suffixes.get(idx, "th")
                ) alert.</p>
            </div>
        @/for
    @else:
        <p>There are no alerts.</p>
    @/if

    <!-- Parte din fisier omisa. -->

</html>
```

Fragment din view-ul paginii principale al aplicației de test

Se observă că afișările de conținut dinamic s-a realizat cu ajutorul instrucțiunii "`@…@`" (cu un spațiu în loc de caracterul de final), instrucțiune care efectuează automat sanitizarea rezultatului, în cazul în care valoarea variabilelor afișate ar proveni din surse care nu sunt de încredere în privința siguranței (ca exemplu, comentarii scrise de către utilizatori pe o pagină Web). Cu toate acestea, există situații când rezultatul evaluării codului Python trebuie să fie interpretat ca HTML:

Exemplu de folosire a afișării nesanitizate a unei valori Python

```
<!-- Aceasta este o lista HTML generata dinamic -->
<ul class="nav␣nav-pills␣pull-right">
    @for name, url in g.header_items:
        <!-- Variabila "cl" retine atributul "class"
             al elementului <li>, sau un sir vid
             in cazul in care acesta nu primeste nici
             o clasa HTML.
        -->
        @{
            cl = 'class="active"' if url == ctrl.page_url\
                else ""
            a_id = "header-a-{}".format(name.lower())
            li_id = "header-li-{}".format(name.lower())
        }
        <!-- Variabila cl este afisata fara sanitizare
             deoarece trebuie interpretata ca HTML
             iar valoarea acesteia nu poate fi controlata
             de catre client.
        -->
        <li role="presentation"
            @!cl@
            id="@li_id@">
```

```
            <a href="@url@" id="@a_id@">@name@</a>
        </li>
    @/for
</ul>
```

Fragment din fișierul ".view/header.html" din aplicația de test, importat în view-ul de mai sus

5 Automatul de analizare text

5.1 Funcţionalitate

Această lucrare implementează un automat generic, performant, determinist, de analiză de text. Similar unui automat finit determinist[11], acesta are o mulțime finită de stări clasice, reprezentate prin numere, iar tranzițiile au ca intrare o stare fixată. O instanță al unui automat este în exact o stare la un moment dat, și acesta trebuie să efectueze tranziții atât timp cât există simboluri (concret, caractere) la intrare.

Spre deosebire de un automat finit determinist, totuși, acest automat permite ca tranzițiile accesibile dintr-o stare să poată fi simultan satisfăcute, și sunt permise ϵ-tranzițiile (tranziții care nu consumă un simbol de la intrare). Determinismul acestui automat, totuși, este dat de aplicarea unei ordini asupra tranzițiilor, și este garantat că prima tranziție posibilă va fi efectuată. Totuși, prezența ϵ-tranzițiior nu garantează imposibilitatea ca automatul să ruleze la infinit, cu toate că aceasta poate fi considerată o eroare a proiectării instanței de automat, și nu a implementării sistemului.

În plus, tranzițiile pot efectua operații mai complexe decât testarea prezenței unui simbol, consumarea acestuia și schimbarea stării automatului. Fiecare tranziție este un obiect cu patru metode principale configurabile. Metoda de gardă verifică dacă simbolul de la intrare (un caracter) poate fi consumat de această tranziție. Metoda de obținere a stării de ieșire întoarce numărul corespunzător stării în care se va afla automatul după executarea tranziției. O a treia metodă specifică dacă această tranziție consumă simbolul de la intrare (în caz contrar, această tranziție se comportă ca o ϵ-tranziție din automatul finit nedeterminist), iar a patra metodă este apelată la execuția unei tranziții. Deoarece oricare din aceste metode poate executa cod arbitrar, acest automat poate executa orice program, cu toate că utilitatea acestuia este concentrată pe analiza textuală.

Spre deosebire de automatele clasice, și similar cu parserele existente, utilitatea principală a automatului de analizare a textului al acestei lucrări nu este acceptarea sau refuzul unei secvențe de simboluri, ci extragerea de informații din șirul de caractere de la intrare. Aceste infomații pot fi un arbore cu informații sintactice sau semantice (arbore de parsare), o structură, dar și verificarea unei proprietăți pentru caracterele din textul de la intrare, ca într-un automat clasic. Datorită libertății date de posibilitatea de a include cod arbitrar în tranziții, acest parser poate consuma caracterele de la intrare, le poate separa în unități lexicale (*token-uri*), le poate interpreta sintactic și semantic, într-o singură parcurgere și într-un singur automat. În plus, din cauza faptului că o tranziție poate fi executată în funcție de simbolul curent, dar fără a-l consuma, pot fi create perechi de tranziții care pot lua decizii în funcție de simbolul următor[17]. Astfel, acest automat este cel puțin la fel de puternic ca un parser *LR(1)*[12].

[17] Dacă se doreşte ca o executarea unei tranziţii să depindă de simbolul următor, aceasta se poate face prin consumarea simbolului curent într-o tranziţie către o stare adiţională, de unde mai multe tranziţii pot verifica simbolul care urmează

Pentru a facilita analiza textului, automatul stochează în mod implicit caracterele consumate. Această stocare este împărțită pe două niveluri: cel imediat este reprezentat de *buffer* unde sunt depuse caracterele consumate, iar nivelul secundar conține o coadă de șiruri de caractere, în care sunt depozitate șirurile de caractere din buffer, atunci când analiza unui element semantic necesită preluarea mai multor șiruri separate de la intrare. De exemplu, un câmp antet dintr-o cerere HTTP este format din două șiruri separate, numele și valoarea câmpului. La apariția simbolului ":", numele câmpului este extras din buffer în coada de pe nivelul secundar. La finalul valorii câmpului, aceasta este extrasă la rândul ei și depozitată în coadă, de unde ambele componente sunt extrase imediat și adăugate în structura cererii HTTP.

Tranzițiile au posibilitatea de a preveni ca simbolul (caracterul) consumat la momentul curent să fie stocat în buffer, posibilitate utilă pentru omiterea caracterelor de control. De exemplu, caracterul ":" din exemplul anterior nu face parte nici din numele, nici din valoarea câmpului antet, deci poate fi omis. În cazul în care o secvență mai lungă de caractere trebuie omise, automatul permite tranzițiilor să renunțe la întregul șir de caractere salvat în buffer.

O altă funcție a acestui parser este funcția de puncte de salvare (*savepoint-uri*), care sunt poziții în buffer-ul automatului și sunt manipulate de către tranziții. Există două operații posibile cu aceste poziții: "salvarea" stochează ca *savepoint* lungimea curentă a șirului din buffer, iar "restaurarea" trunchiază acest șir la lungimea reținută anterior. O situație în care această funcționalitate este utilă este analiza instrucțiunii "`@for … in …:`", în care este cele două secvențe de cod Python sunt separate de o secvență "␣in␣". Dificultatea parsării acestei structuri este dată de faptul că este imposibil ca automatul să prezică dacă un caracter spațiu este sau nu parte al secvenței separator de mai sus, deci, dacă acest spațiu, alături de caracterele care urmează, fac sau nu parte din codul Python care trebuie salvat în buffer. Soluția acestei probleme este salvarea unui *savepoint* la întâlnirea unui caracter spațiu, urmărirea printr-un număr de stări intermediare progresul în separatorul de mai sus, iar, la finalul acestei secvențe, restaurarea acestui *savepoint*, urmată de adăugarea buffer-ului (care va conține prima secvență de cod Python, înainte de secvența "␣in␣") la coada menționată mai sus, pentrua fi adăugată ulterior la arborele sintactic și semantic.

Unul din mecanismele prin care acest parser poate analiza unele limbaje formale de tipul 2 este dicționarul de proprietăți configurabile al acestuia, care este un vector asociativ cu chei șiruri de caractere și valori numere întregi. Asfel, tranzițiile pot citi și modifica valorile din acest dicționar, și pot permite sau nu efectuarea tranziției în funcție de valorile acestea. De exemplu, o proprietate din acest dicționar poate reprezenta un contor care reține adâncimea curentă a unui șir de paranteze imbricate. Tranzițiile care consumă caracterul "(" vor incrementa acest contor, o tranziție pentru caracterul ")", posibilă doar atunci când acest contor are o valoare nenulă îl va decrementa, iar o altă tranziție care

(acum simbol curent), executându-se cea care se potriveşte, dar fără ca aceasta să consume acest simbol.

consumă caracterul ")" va trece la altă stare doar atunci când valoarea contorului este zero.

O problemă impusă de parsarea limbajelor este comportamentul automatului la finalul șirului de caractere de la intrare. Din moment ce salvarea rezultatelor este efectuată cu ajutorul tranzițiilor, iar tranzițiile sunt determinate de consumul unui caracter, ar fi imposibil ca automatul să reacționeze la finalul simbolurilor de la intrare. Soluția acestei probleme este parcurgerea finală, un pas adițional de consum, cu un simbol care nu provine de la intrare, care poate fi cerută de către codul care folosește parserul. În momentul acestui pas final de execuție, există un câmp boolean setat în structura automatului, lucru care permite recunoașterea acestei stări speciale în automat și efectuarea de tranziții speciale. Un exemplu de utilitate al acestei funcții este in parsarea limbajului fișierelor de *template*, în care elementele de conținut static (de exemplu, într-o pagină Web, părțile scrise în limbajul HTML, și nu elementele de cod Python) sunt delimitate de elementele dinamice. Pentru eficiență, acestea nu sunt salvate din buffer în arborele rezultat decât la finalul acestora, adică fie la apariția unui element dinamic, fie la finalul fișierului. Din acest motiv, este necesară detectarea finalului, pentru a adăuga elementele de conținut în lucru.

O soluție adițională la problema impusă în paragraful anterior este funcționalitatea de "acțiuni finale", acestea fiind funcții asociate fiecărei stări, și executate după parcurgerea finală menționată mai sus. Astfel, dacă o stare a automatului nu este validă la finalul unui șir de caractere analizat, indicând, de exemplu, un fișier trunchiat, o acțiune finală specificată pentru această stare poate arunca o excepție sau seta o variabilă, specificând o stare de eroare. Această funcționalitate este extensiv folosită în parsarea cererilor HTTP, care au o stare finală foarte bine definită, orice altă stare (de exemplu, câmp antet incomplet) fiind o stare de eroare.

Automatul suportă și executarea de cod la începutul consumării unui simbol, în funcție de starea curentă a automatului, dar această funcționalitate de "acțiuni de stare" nu este momentan folosită.

Configurarea automatului se face prin alegerea unui număr maxim de stări, prin adăugarea de tranziții între acestea, și prin configurarea acțiunilor de stare și finale. Un dezavantaj al acestui parser este că, din cauza complexității acestuia, nu există la momentul curent nici un algoritm care să transforme o gramatică într-un astfel de parser. Pentru a compensa acest dezavantaj, sistemul oferă un set de funcționalități concentrate pe analiza limbajelor de programare. Astfel, fiecare automat are două tipuri de stări: stări normale și stări adiționale. Stările normale sunt accesibile utilizatorului parserului, acesta având posibilitatea de a adăuga tranziții între acestea și de a configura acțiunile de stare și finale pentru aceste stări. În contrast, stările adiționale sunt alocate de către automat într-o serie de metode care crează subautomate, simplificând elaborarea unui astfel de parser în cazuri comune:

– Una din aceste metode primește trei stări normale și un șir de caractere și folosește stările adiționale pentru a crea un subautomat care, pornind din prima stare normală, consumă șirul de caractere specificat, terminând în cea

de-a doua stare. În momentul în care simbolurile de la intrare nu se potrivesc șirului de caractere specificat, automatul ajunge imediat în starea a treia. Astfel se facilitează analiza unui *token* format din mai multe caractere, lăsând automatul să aloce stările intermediare și să adauge tranzițiile acestora.
- O altă metodă din această serie urmărește să parseze reprezentarea unui șir de caractere într-un limbaj de programare, începând și terminându-se cu un caracter separator, cu posibilitatea ca un caracter stabilit (de obicei "\") să acționeze ca *escape character*. Acestă metodă primește două stări, caracterul delimitator și caracterul de *escape*, și crează un subautomat care pornește din prima stare, analizează un astfel de șir de caractere, și se termină în starea a doua. Această funcționalitate este utilă la parsarea secvențelor de cod, asigurându-se că anumite caractere speciale (de exemplu, caractere de delimitare) vor fi ignorate în interiorul șirurilor de caractere.
- La final, o metodă parsează un bloc demarcat de o pereche de caractere (de obicei complementare, ca "(" și ")" sau "{" și "}"), dar care poate la rândul său conține alte blocuri imbricate, delimitate de aceleași caractere. Această metodă primește două stări și perechea de caractere, iar rezultatul este adăugarea unui subautomat care pornește de la prima dintre cele două stări, consumă caracterul de început, urmărește adâncimea de imbricare în funcție de perechea de caractere de demarcare, iar, doar atunci când adâncimea este nulă, consumă caracterul de final și termină în cea de-a doua stare. Astfel, se pot analiza blocuri din secvențe de cod, asigurându-se, la fel ca pentru metoda anterioară, că simbolurile de delimitare sunt ignorate în interiorul acestor blocuri.

O problemă a stărilor adiționale este comportamentul automatului la finalizarea șirului de caractere de la intrare, atunci când starea curentă este una din cele adiționale. Din această cauză, aceste funcții de mai sus configurează, după context, câte o stare normală pentru fiecare stare adițională, în care automatul să treacă dacă, în momentul în care simbolurile de la intrare se termină, acesta este în starea adițională specificată. De exemplu, pentru parsarea unui token, această stare este cea de-a treia, aceeași ca în cazul apariției unui caracter diferit de următorul caracter din token.

O altă funcționalitate a acestui automat este implementarea existentă a unor tranziții des folosite. În plus, unele tranziții primesc o altă tranziție ca parametru, producând un lanț de tranziții, fiecare adăugând comportament. De exemplu, prin înlănțuirea unei tranziții de adăugare a buffer-ului în coada menționată mai sus cu o tranziție care consumă un caracter, se obține o singură tranziție între două stări care efectuează ambele operațiuni. Spunem că o tranziție înlănțuie o alta dacă prima o primește pe cea de-a doua ca argument și-i modifică comportamentul.

Tranzițiile implementate de către automat sunt:

- "Simplu" acceptă un caracter specificat și trece automatul în altă stare. Opțional, poate lăsa caracterul neconsumat (similar unei ϵ-tranziții).

- "Întotdeauna" acceptă orice caracter și trece automatul în altă stare. Ca în cazul "Simplu", poate lăsa caracterul neconsumat. (În diagrame, după caz, mai este numit "Altceva")
- "Spațiu alb" acceptă orice caracter alb[18] și trece automatul în altă stare. Poate lăsa caracterul neconsumat. (În diagrame, pentru simplitate, mai este numit "WS", de la "White space")
- "Final" acceptă orice caracter doar în timpul parcurgerii finale, și trece automatul în altă stare.
- "Acțiune" înlănțuie o tranziție și, în plus, execută o funcție specificată ca argument.
- "Și condiție" înlănțuie o tranziție, dar permite execuția acesteia doar atunci când o funcție specificată ca argument întoarce valoarea `true`.
- "Sau final" înlănțuie o tranziție, dar permite execuția ei indiferent de condiția acesteia, dacă automatul efectuează tranziția finală.
- "Omitere" înlănțuie o tranziție, dar previne salvarea caracterului curent în buffer.
- "Adăugare în coadă" (*push*) înlănțuie o tranziție și adaugă conținutul bufferului în coada de șiruri de caractere salvate.
- "Golire buffer" înlănțuie o tranziție și golește buffer-ul, ingorând întreg conținutul acestuia.
- "Salvare savepoint" înlănțuie o tranziție și adaugă un savepoint.
- "Restaurare savepoint" înlănțuie o tranziție și restaurează un savepoint.
- "Eroare" aruncă o excepție în momentul în care se încearcă potrivirea acesteia. Folosit pentru situații când automatul nu este într-o stare validă.

Utilizatorii automatului pot defini tranziții noi, care să construiască rezultatul analizei, fie că acesta este un arbore de parsare sau o altă structură de date.

În diagramele automatelor, tranzițiile sunt numerotate în ordinea priorității acestora, și tranzițiile înlănțuite sunt scrise câte o componentă pe linie. Numele tranzițiilor sunt cele din enumerarea anterioară, iar rolurile tranzițiilor implementate pentru o utilizare anume (de exemplu, tranziția care adaugă un câmp antet HTTP) sunt evidente din numele acestora, sau sunt explicate ulterior.

5.2 Automatului pentru analiza cererilor HTTP

Acest parser urmărește popularea câmpurilor unei structuri care memorează o cerere HTTP. Pentru acest scop, s-au implementat o serie de tranziții, fiecare înlănțuind o tranziție dată ca parametru, care pot consuma conținutul bufferului sau cozii de șiruri de caractere salvate, fiecare adăugând date la un câmp al acestei structuri.

Diagrama parserului HTTP este prezentată în figura 2. Pentru simplificarea diagramei, au fost omise anumite cazuri speciale ale protocolului HTTP, dar acestea sunt implementate în lucrare.

[18] Se consideră caractere albe cele cu codul ASCII 9, 10, 13 sau 32.

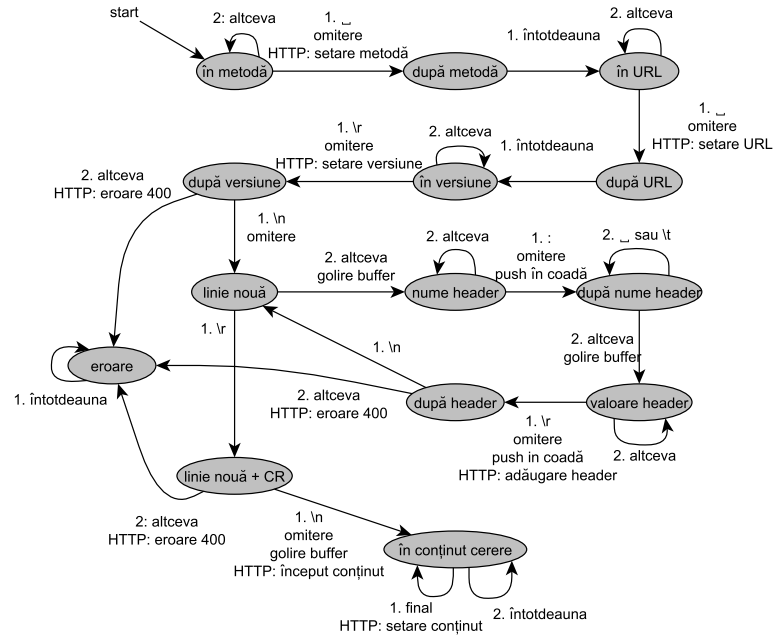

Figura 2. Reprezentare vizuală a parserului pentru cereri HTTP. S-au omis din diagramă stările şi tranziţiile adăugate pentru tratarea anumitor cazuri speciale.

5.3 Automatului pentru limbajul fişierelor de *template*

Limbajul de *template* al acestui server este transformat într-un arbore cu informații sintactice și semantice de către un parser complex care folosește automatul acestui proiect. În plus, acest parser folosește în multiple locuri subautomatele menționate mai sus.

Din cauza complexității diagramei, aceasta este împărțită în mai multe figuri. Astfel, schema generală a întregului parser este prezentată în figura 3, iar părțile acestuia sunt prezentate în figurile 4 și 5.

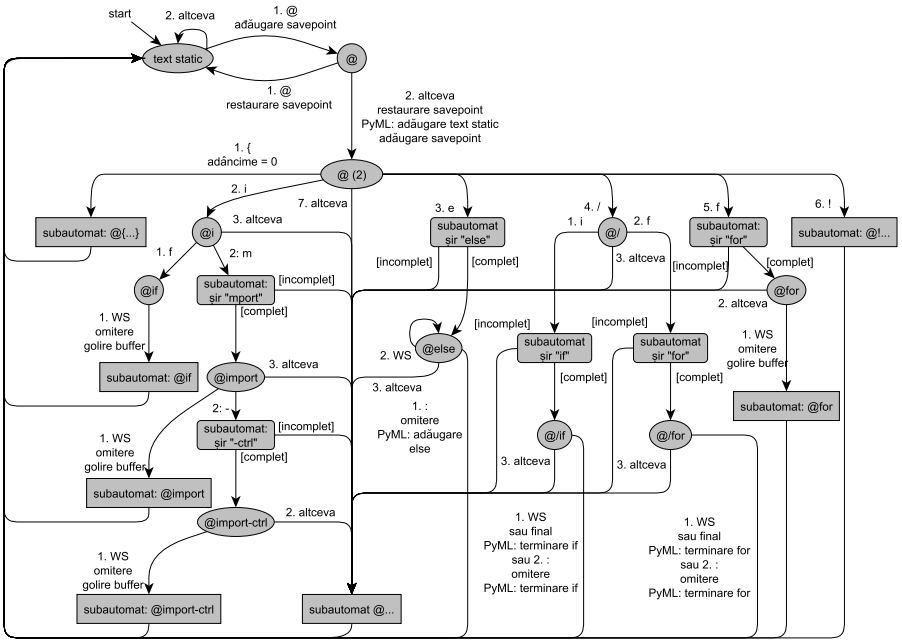

Figura 3. Schemă generală a parserului pentru limbajul PyML. Nodurile dreptunghiulare sunt subautomate detaliate mai jos. Nodurile dreptunghiulare cu colţuri rotunjite sunt subautomate generate de către acest parser, în stări adiţionale.

Tranzițiile “PyML: …” din figurile 3, 4 și 5 prelucrează un arbore de elemente ale limbajului de *template*. La finalul parsării unui fișier, acest arbore este transformat într-un arbore similar, dar imutabil. Aceste structuri de date sunt diferite deoarece în procesul parsării, acest arbore trebuie modificat, dar după aceasta, deoarece arborele este reținut în cache-ul serverului, este importantă garanția că acesta nu se va schimba.

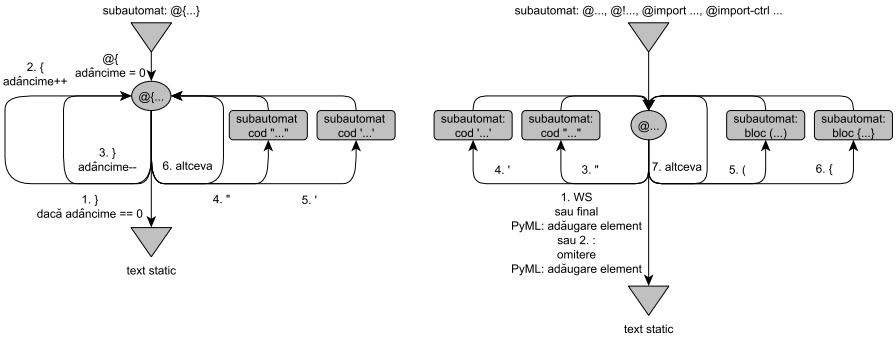

Figura 4. Diagrama subautomatelor pentru blocuri de cod şi expresii Python. Parserul din stânga este folosit pentru instrucţiunea "`{ ... }`", iar cel din dreapta pentru instrucţiunile care sunt compuse dintr-un cuvânt cheie, urmat de o expresie Python (instrucţiuneea de evaluare, evaluare fără sanitizare, şi cele de import al unui view sau controller.)

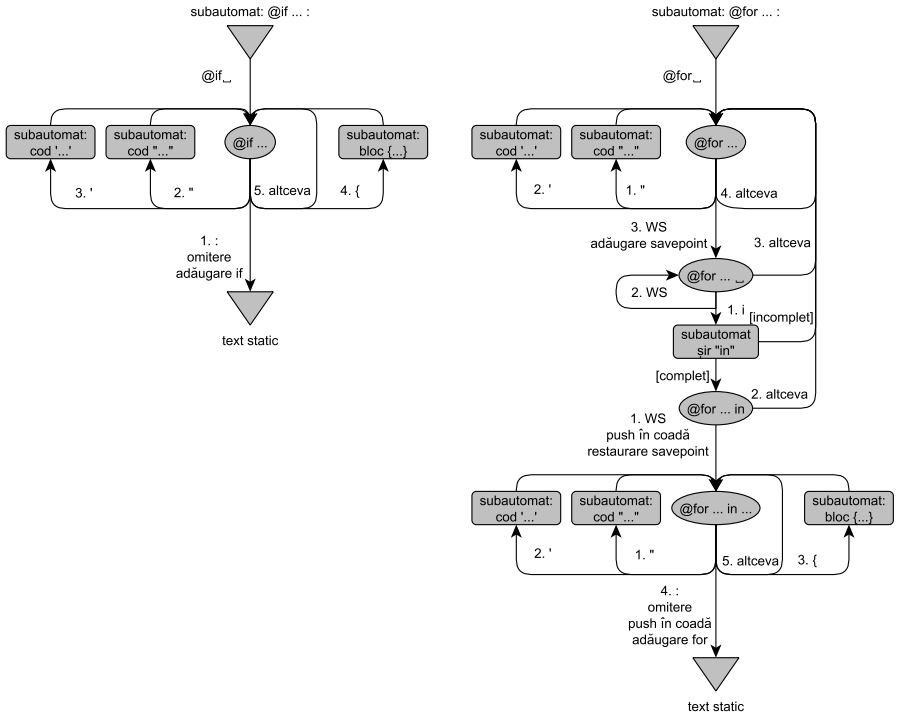

Figura 5. Diagrama subautomatelor pentru elementele condiţionale (stânga) şi repetitive (dreapta)

5.4 Alte automate

Această lucrare mai folosește încă o instanță a acestui sistem de implementare de automate, pentru consumarea fragmentelor de cod Python și determinarea locațiilor și metodelor de a introduce separatoare de linie în acest fragment de cod. Această funcționalitate este utilă în procesul de compilare a codului, după cum se va detalia în secțiunea 6.

Limbajul Python permite introducerea de linii noi între instrucțiuni, și în interiorul majorității acestora, prin terminarea unei linii cu caracterul "\". Singurele situații în care nu este posibilă introducerea unei linii noi în acest mod, fără a modifica corectitudinea și semantica unei instrucțiuni, sunt:

- În interiorul comentariilor pe o linie ("`# …`"). Pentru a introduce o linie nouă, linia introdusă trebuie să înceapă cu un caracter "#".
- În interiorul șirurilor de caractere. În acest caz, trebuie împărțit șirul în două, adăugând o pereche de caractere de delimitare ("`'`" sau "`"`", după caz), și introducând separatorul de linie între acestea. Interpretorul Python concatenează automat șirurile de caractere care nu sunt separate prin alte instrucțiuni.
- În interiorul șirurilor de caractere pe mai multe linii ("`"""…"""`" sau "`'''…'''`"). Separarea acestora se face ca în cazul anterior, cu excepția faptului că trebuie introduse 6 caractere de separare (o pereche de separatoare de câte 3 caractere fiecare).

În acest caz, parserul este folosit ca un automat finit determinist, fără folosirea funcționalităților avansate ale acestui parser. Din interese de limitare a detaliilor din această secțiune, diagrama acestui automat nu este inclusă în lucrare.

6 Compilatorul pentru fişiere *template*

Sistemul dezvoltat poate evalua un fișier *template* în două moduri, după ce acesta a fost citit și transformat într-un arbore de parsare. Prima modalitate este parcurgerea arborelui, luând elementele statice și evaluând codul Python al elementelor dinamice și adăugându-le la răspunsul HTTP. A doua modalitate ese generarea un fișier sursă Python care, în momentul când este rulat, să apeleze o funcție care adaugă date la acest răspuns HTTP, producând același rezultat ca în primul caz. Generarea acestui fișier Python reprezintă scopul compilatorului detaliat în această secțiune.

6.1 Transformarea din HTML în Python

Acest compilator primește la intrare arborele rezultat în urma parsării și produce un modul de Python care conține o funcție "run". Când este nevoie de generarea unui răspuns dintr-un fișier sursă compilat, serverul importă modulul acestuia și execută metoda "run" a acestuia. Avantajul acestui sistem este că, o dată importat un modul de Python, acesta rămâne în memoria RAM a procesului și nu mai trebuie importat o a doua oară. În plus, interpretorul de Python scrie pe disc, pentru fiecare fișier ".py" importat, un fișier ".pyc" care conține bytecode-ul acestuia, eliminând nevoie de a-l analiza la fiecare rulare.

Prima decizie a acestui compilator este locația și numele fișierelor rezultate. Aceste fișiere sunt scrise în directorul ".compiled/__krait_compiled" din directorul rădăcină al site-ului. Din cauza faptului că aceste fișiere sunt module Python, numele acestor fișiere trebuie sa fie identificatori valizi în acest limbaj de programare. Asfel, caracterele nevalide sunt transformate într-o secvență de două sau mai multe caractere, de exemplu "_s" pentru "/" sau "_p" pentru ".". În afară de o listă prestabilită de substituții, caracterele rămase sunt transformate într-o secvență de trei caractere, formată din caracterul "_" urmat de codul ASCII al caracterului sursă, în baza 16. Deoarece această transformare trebuie să fie reversibilă, caracterul "_", care și-a pierdut semnificația, este codat ca "___".

Serverul expune două module Python scrise în limbajul C++, care permit codului Python să adauge șiruri de caractere la răspunsul HTTP (operațiune numită în acest capitol *emitere*), să compileze fișiere *template*, și să ruleze aceste fișiere. La începutul funcției "run", compilatorul introduce o secvență de cod care asignează unor varibile locale aceste funcții, deoarece variabilele locale sunt găsite și accesate mult mai rapid decât cele globale sau cele aparținând altor module[19]. Presupunând că funcțiile de emitere (de adăugare al unui șir de caractere la răspunsul HTTP) sunt apelate de un număr semnificativ de ori, aceasta este o optimizare simplă și utilă.

O problemă a acestei variante de compilare este comportamentul acesteia la modificarea unui fișier de *template*. Din această cauză, la executarea unui fișier compilat, acesta trebuie să se recompileze dacă se modifică. Din această cauză,

[19] `https://wiki.python.org/moin/PythonSpeed/PerformanceTips#Local_Variables`

fișierul generat de compilator conține, într-un comentariu aflat pe prima linie a acestuia, *entity tag*-ul folosit în mecanismul de cache HTTP. La începutul metodei "run", compilatorul introduce o secvență de cod care verifică ca fișierul de *template* din care a fost generat acest fișier compilat să nu fie modificat. In caz contrar, fișierul este recompilat și modulul este importat din nou.

După această verificare din funcția "run" urmează codul generat din arborele sintactic și semantic al unui fișier de *template*. Compilatorul transformă fiecare element din acest arbore în codul său Python echivalent.

Secvențele de cod Python (din elementele dinamice) rămân în mare parte nemodificate, iar elementele statice (de exemplu, într-o pagină Web, secvențele de cod HTML) devin apeluri la funcția de emitere fără sanitizare al unui șir de caractere. Totuși, din cauza faptului că, în codul Python, elementele statice devin șiruri de caractere potențial foarte lungi, compilatorul folosește automatul descris în secțiunea 5.4 pentru a împărți aceste șiruri de caractere pe mai multe linii. Se încearcă împărțirea șirurilor astfel încât fiecare linie din codul Python rezultat să aibă între 80 și 90 de caractere, cu toate că în anumite cazuri (dacă ar rămâne mai puțin de 6 caractere pe linia următoare), se permit și linii puțin mai lungi. O altă problemă este că, în secvențele de cod Python, există problema separatoarelor de linie de tip Windows (*CRLF*) care trebuie transformate în separatoare de tip Unix (*LF*). O dificultate este faptul că această transformare nu trebuie făcută în șiruri de caractere pe mai multe linii (separate de "`"""`" sau "`'''`", șirurile fiind determinate de automatul menționat mai sus), în interiorul cărora aceste separatoare trebuie transformate în secvențele "`\r`" și "`\n`". În plus, din cauza faptului că această compilare modifică indentarea codului Python, aceste șiruri de caractere trebuie transformate din șiruri de caractere pe mai multe linii într-o serie de șiruri de caractere, unul pe linie. (Interpretorul limbajului Python concatenează șiruri de caractere consecutive, atât timp cât nu există alte elemente de sintaxă între acestea.)

Compilarea elementelor PyML condiționale și repetitive ("`@if …:`", "`@else:`" și "`@for … in …:`") este banală, semantica lor în acest limbaj fiind identică cu semantica din limbajul de origine, Python. Singura modificare necesară este actualizarea indentării, lucru efectuat de clasa care scrie fișierul compilat pe disc. Instrucțiunile "`@import …@`" și "`@import-ctrl …@`" funcționează prin generarea codului care evaluează expresia date ca argument (și în cazul "`@import-ctrl`", apelează metoda "get_ctrl" a rezultatului), transformarea căii de fișier rezultate într-un nume de modul, după regulile de mai sus, iar apoi rularea modulului acesta.

O dificultate întâlnită de către compilator este compilarea rutelor MVC. Din cauza faptului că aceste rute primesc o clasă Python și nu o secvență de cod, crearea unui fișier compilat care să găsească această clasă este dificilă. Pentru rezolvarea acestei probleme, serverul scrie după configurarea rutelor, pentru fiecare rută care duce la un controller MVC, câte un fișier similar cu cele compilate. În acest proces, clasele controller-elor MVC sunt reținute într-o variabilă globală și sunt accesate în momentul execuției. Aceste fișiere au numele generat în funcție

de indicele din lista de rute, pentru ca mecanismul de rutare al acestei lucrări să poată găsi acest modul compilat, pentru a-l executa.

6.2 Integrarea cu instrucţiunea `import`

Pentru ca aceste fișiere compilate să poată fi încărcate ca module, este nevoie ca să poată fi găsite de instrucțiunea Python "import". Mecanismul recomandat pentru modificarea comportamentului acestei instrucțiuni în limbajul de programare Python este de a implementa un *import hook*, care este o metodă standardizată de a genera fișiere Python, la cerere, pentru a fi importate. Acest mecanism este descris în PEP 302[20] și constă din a furniza un obiect cu o metodă care va fi apelată la fiecare import, care poate întoarce un alt obiect, acesta având o metodă care întoarce modulul importat.

Pentru ca directorul unde aceste fișiere sunt scrise, "__krait__compiled", să fie considerat un pachet, potrivit cu regulile mecanismului de import din limbajul Python, *import hook*-ul acestui proiect scrie la cerere fișierul "___init___.py", care semnifică rolul de pachet al unui director.

Ulterior, când se cere un modul aparținând acestui pachet, *import hook*-ul implementat verifică dacă fișierul corespunzător modulului este compilat și nu au survenit modificări fișierului sursă, în caz contrar declanșând procesul de compilare. Apoi, acest fișier compilat este importat și poate fi executat.

6.3 Obţinerea răspunsului HTTP

Răspunsul HTTP este modificat de către fișierele compilate prin apeluri succesive ale funcțiilor de emitere. Aceste funcții de emitere trebuie să stocheze șirul de caractere primit pentru a fi trimis ulterior către client.

O cerință principală a acestui mecanism a fost evitarea copierii inutile de șiruri de caractere. Astfel, acest modul (implementat în limbajul C++) stochează trei vectori, unul de referințe către șiruri de caractere cu lungimile acestora, reprezentând fragmentele de răspuns HTTP, iar două cu obiectele unde sunt memorate aceste șiruri: unul pentru șiruri din limbajul C++ iar celălalt pentru șiruri din limbajul Python. Pentru alcătuirea răspunsului, se folosește doar primul vector, celelalte fiind păstrate doar pentru a evita dealocarea acestora. În general, se preferă folosirea șirurilor de caractere în limbajul în care au fost primite ca argumente, ceea ce, de obicei, înseamnă șiruri de caractere Python. Singura situație când se efectuează copierea acestora este atunci când aceste șiruri trebuie modificate în procesul de sanitizare. Această sanitizare se efectuează doar la executarea unei instrucțiuni "`@…@`", și doar când șirul de caractere rezultat în urma evaluării are caractere care necesită acest proces (caractere care au o semnificație specială în limbajul HTML).

[20] `https://www.python.org/dev/peps/pep-0302/`

7 Practici de dezvoltare

Încă de la începutul implementării acestui server, a fost folosit sistemul Git de control al versiunii, iar proiectul a fost găzduit pe GitHub, ca proiect public, sub licența MIT. Această licență a fost aleasă din cauza libertății pe care o oferă dezvoltatorilor care ar dori să extindă funcționalitatea acestui proiect. De-a lungul dezvoltării proiectului, au fost publicate următoarele versiuni:

- **0.1**: versiunea inițială, serverul este funcțional dar rudimentar.
- **0.2**: sistemul de rutare și interfața de programare pentru paginile dinamice au fost îmbunătățite.
- **0.3** (nepublicată): s-a adăugat suport pentru cookie-uri HTTP și protocolul WebSocket.
- **0.4**: stabilitatea serverului a fost îmbunătățită.
- **0.5**: configurația site-urilor a fost trecută la sistemul cu fișier executabil Python și a fost scrisă documentația pentru interfața de programare.
- **1.0**: au fost realizate modifcări interne majore, s-a adăugat suport pentru TLS/SSL și arhitectura MIPS, și a fost implementat compilatorul pentru fișiere Python.

Un instrument util în organizarea *repository*-ului de Git a fost proiectul Git Flow[21], care prezintă un model bine organizat de administrare a *branch*-urilor Git. Cu ajutorul acestuia, s-a menținut un *repository* clar structurat evitându-se problemele inconsecvenței unei organizări ad-hoc.

7.1 Mediul de lucru

Una din dificultățile întâlnite în dezvoltarea acestui proiect a fost faptul că serverul este scris pentru sistemul de operare Linux, dar, din diverse motive, a fost imposibilă instalarea acestui sistem pe calculatorul personal al autorului acestei lucrări. La început, s-a încercat folosirea unei mașini virtuale, dar acest mediu de dezvoltare era greu de folosit. Din acest motiv, implementarea proiectului a fost continuată în mediile de programare Jetbrains CLion, iar apoi Microsoft Visual Studio Community, cel din urmă suportând dezvoltarea aplicațiilor pentru sistemul Linux, de pe un sistem Windows. Compilarea și testarea serverului s-a efectuat inițial pe un server ARM găzduit pe platforma Scaleway, iar apoi în subsistemul Linux din Windows 10 (*Windows Subsystem for Linux*).

7.2 Testarea serverului

Serverul a fost testat cu ajutorul unui unei aplicații Web cu patru pagini, fiecare testând un aspect diferit al serverului:

- "index", pagina principală, testează, pe lângă funcționarea generală a serverului, afișarea de variabile din cod Python, instrucțiunile "@if" și "@else" și "@for".

[21] `https://github.com/nvie/gitflow`

– “db” testează accesul la baze de date SQLite[22], citirea cookie-urilor, și formularele Web trimise prin metoda POST. În scriptul care procesează formularul, se testează citirea acestuia din cererea HTTP, scrierea bazei de date SQLite, salvarea unui cookie, și mecanismul de suprascriere a răspunsului HTTP (în acest caz, cu o redirectare printr-un răspuns “302 Found”).
– “http” testează rutarea cu expresii regulate, parsarea corectă a cererilor HTTP și sanitizarea și desanitizarea caracterelor speciale din URL-uri.
– “ws” testează implementarea protocolului WebSocket prin implementarea unei mici aplicații care necesită modelul bidirecțional al acestui protocol.

În plus, fiecare pagină folosește o instrucțiune “@import” pentru a include aceeași bară de navigare, care testează, pe lângă mecanismul de import, instrucțiunea de afișare a unei variabile fără sanitizare (“`@!…@`”).

Pentru automatizarea procesului de testare, s-a optat pentru folosirea uneltei Selenium[23], un proiect care oferă un mecanism de control al unui browser Web, și permite atât observarea cât și interacțiunea cu pagina Web trimisă de către server. Testele menționate mai sus au fost implementate într-o serie de scripturi Python care compilează și pornesc serverul, iar apoi folosesc biblioteca Selenium pentru a deschide un browser Web, a naviga la pagina unde este găzduit serverul (de obicei pe același calculator), și a rula testele în mod automat.

În plus, în procesul de testare al serverului se generează și documentația, verificând că aceasta nu conține erori de sintaxă, nici nu lipsesc secțiuni sau unele fragmente de informație.

[22] `https://www.sqlite.org/index.html`
[23] `https://www.seleniumhq.org/`

8 Studii de caz

Pe lângă aplicația Web de test menționată în secțiunea 7.2, acest proiect a fost folosit și pentru implementarea altor aplicații Web, ambele evidențiind avantajele acestui server HTTP.

8.1 Two Stones: joc pentru mai mulţi jucători

Această aplicație a fost elaborată de către autorul lucrării împreună cu trei alți colegi în cadrul materiei Tehnologii Web din al doilea semestru al anului 2, în perioada mai-iunie 2017. Proiectul a folosit acest server, baze de date Oracle, și *framework*-ul pentru jocuri Web Phaser CE[24].

Această aplicație consta dintr-un joc de strategie, în care pot juca mai mulți jucători, prin Internet (spre deosebire de jocurile în care mai mulți jucători participă la un joc de pe același calculator). În acest joc, fiecare jucător își crează și controlează o echipă formată din cinci unități, și două echipe, fiecare aparținând unui jucător, se confruntă pe aceeași tablă de joc. Fiecare echipă protejează un steag, și încearcă să obțină steagul echipei inamice și să-l aducă înapoi la baza propriei echipe, fără ca steagul propriu să-i fie luat. O echipă obține un punct când ambele steaguri din joc, cel propriu și cel inamic, sunt în baza echipei. Tabla de joc este formată dintr-o grilă hexagonală bidimensională, iar unitățile din joc se pot mișca pe aceasta și pot ataca unități inamice. Unitățile interacționează cu steagurile prin deplasarea în hexagonul acestora: steagul este luat de către unitate, care trebuie să se întoarcă cu acesta, fără să fie scos din joc, la baza proprie. Un joc este format dintr-o succesiune de ture, jucătorii alternând controlul echipei proprii. Atunci când este rândul unui jucător, acesta poate mișca cel mult o dată și ataca cel mult o dată, pentru fiecare din unitățile din echipa sa, dar poate pune în orice ordine atacurile și mișcările.

Fiecare unitate dintr-o echipă are patru abilități, cuantificate numeric: rezistența sa la atacuri (câte puncte de atac poate absorbi înainte de a ieși din joc), puterea sa de atac (câte puncte de atac scade acesta din rezistența rămasă a unității inamice), raza sa maximă de atac (cât de aproape trebuie să fie o unitate inamică pentru a putea fi atacată, în hexagoane) și raza sa maximă de deplasare (cât de departe se poate mișca într-o singură tură, în hexagoane). Aceste abilități sunt calculate dintr-o bază, la care se adaugă maxim trei modificări, care adaugă un procent (20%, 30% sau 40%) la una din abilități și scade același procent din alta. Bazele acestor abilități se numesc clase. Există patru clase, fiecare din acestea prioritizând una din cele patru statistici, iar jucătorul trebuie să aleagă o clasă pentru fiecare unitate, și maxim trei modificări. Scopul acestor decizii este ca un jucător să-și creeze o echipă echilibrată, în care să existe unități cu avantaje variate.

Din nefericire, această aplicație nu poate rula momentan, parțial din cauza schimbărilor survenite interfeței de programare a serverului (cu toate că modificările necesare nu sunt majore), dar în principal din cauza dificultăților impuse de pregătirea infrastructurii de execuție, și mai ales de baza de date Oracle.

[24] `http://phaser.io/`

Modulele principale ale acestei aplicații sunt următoarele:

Baza de date. Baza de date reține tablele de joc, clasele și modificatorii disponibili, jucătorii cu echipele și unitățile acestora, jucătorii care așteaptă să intre într-un joc, jocurile în desfășurare și cele terminate.

Scripturile la nivel de server. Aceste scripturi trimit paginile către jucători, aleg perechi de jucători care așteaptă începerea unui joc și pornesc acest joc, și administrează jocurile în desfășurare. În timpul acestor jocuri, comunicarea dintre server și client, precum și modificările tablei de joc, se fac prin protocolul WebSocket.

Paginile Web. Acestea sunt scrise în HTML, cu aplicarea foilor de stiluri CSS și comportamentul în limbajul TypeScript[25] compilat în JavaScript. În dezvoltarea meniurilor s-au folosit bibliotecile Bootstrap 3 și JQuery, iar în implementarea jocului propriu-zis, biblioteca Phaser CE, care oferă acces ușor de folosit la grafica în paginile Web.

Codul acestui proiect este disponibil sub o licență MIT la adresa `https://github.com/TED-996/krait-twostones`.

8.2 Interfaţă de control pentru LED-uri RGB

Un studiu de caz care demonstrează capabilitatea acestui server pentru dispozitive Internet of Things este un proiect care a urmărit controlul unei benzi de LED-uri RGB cu ajutorul unei interfețe Web. Cerințele acestei interfețe au fost posibilitatea de a alege o culoare pentru aceste LED-uri, de a configura o tranziție între una sau mai multe culori, de a pulsa o culoare la un anume tempo (exprimat în bătăi pe minut), și de a produce o vizualizare de tip "orgă de lumini", modificând culoarea LED-urilor în timp real, în funcție de o melodie aleasă de utilizator.

Această aplicație rulează pe o plăcuță de dezvoltare Onion Omega 2 Plus[26], un sistem bazat pe arhitectura MIPS, care folosește sistemul de operare Linux și suportă conectivitate prin Wi-Fi. Procesorul acesteia rulează la frecvența de 580 MHz, memoria RAM este de 128MB, iar spațiul pe disc este de numai 32MB, înainte de instalarea sistemului de operare. Dat fiind faptul că întregul sistem (fără conexiunile USB și pinii de uz general) are dimensiunea de 42mm în lungime și 26mm în lățime, acesta este un sistem suficient de puternic în aceste circumstanțe, dar, totuși, este mai slab decât un sistem Raspberry Pi de prima generație.

[25] TypeScript (`https://www.typescriptlang.org/`) este o extensie a limbajului JavaScript care conţine informaţii de tipuri şi verifică, la momentul compilării, utilizarea corectă a acestor tipuri, ajutând la eliminarea multor erori comune care apar în acest limbaj.

[26] `https://onion.io/omega2/`

Banda RGB are patru borne: una din ele trebuie să fie conectată la borna "+" a unei surse de curent continuu cu tensiunea de 12 volți, iar restul de trei borne trebuie conectate la borna "−" a sursei, atunci când se dorește aprinderea LED-urilor roșii, verzi, respectiv albastre. Când una din aceste borne "−" este conectată, circuitul care trece prin LED-urile corespunzătoare este închis, lucru care face ca aceste LED-uri să se aprindă. Dacă se dorește formarea culorilor mai complexe, trebuie conectate mai multe borne "−", dar formarea culorilor în care componentele roșu, verde sau albastru nu au valoarea maximă este mai dificilă. Pentru acest lucru, este nevoie de deschiderea și închiderea acestor ciruite cu o frecvență suficient de mare, și într-o rație dintre durata de timp în care circuitul este închis (deci LED-urile sunt aprinse) și cea în care circuitul este deschis (deci LED-urile sunt stinse) corespunzătoare. Acest sistem este cunoscut și sub numele de "pulse width modulation", sau *PWM*.

Pentru această aplicație s-a optat pentru folosirea paginilor Web statice, comportamentul dinamic al serverului fiind disponibil prin trimiterea unor cereri POST (cu ajutorul unui formular Web) și printr-o interfață WebSocket. Dacă se dorește configurarea culorii o singură dată, consumul unei conexiuni WebSocket nu este justificat, dar în cazul în care se produce un efect în timp real, complexitatea de timp a protocolului HTTP și a procesării cererilor acestuia avantajează categoric protocoul WebSocket.

Această aplicație are trei pagini Web:

- "/index.html" conține două metode de a introduce o culoare: prin trei selectoare de interval (*slider*-e) pentru componentele roșu, verde și albastru dintr-o culoare RGB, sau trei slider-e pentru componentele "hue", "saturation" și "value" dintr-o culoare HSL.
- "/direct.html" schimbă culoarea LED-urilor printr-o conexiune WebSocket și permite setarea unei culori RGB, tranzițiile între una sau mai multe culori și pulsarea culorilor după un anumit tempo.
- "/music" conține o aplicație de redare de muzică, cu o vizualizare în timp real a frecvențelor acesteia, și colorarea benzii de LED-uri în funcție de aceste frecvențe.

Analiza muzicii se face în aplicația Web, în browser-ul clientului. Pentru a evita aglomerarea serverului cu fișiere de muzică preluate de la client, aplicația de redare de muzică prezintă utilizatorului un buton de alegere a unui fișier de muzică, după care citește direct în JavaScript aceste fișiere (folosind Web File API[14]) și le redă la ieșirea de sunet a clientului, folosind Web Audio API[15]. Această intefață pentru procesări audio din browser permite ca datele audio să treacă printr-un lanț configurabil de noduri (de generare, procesare, analiză și ieșire de sunet). Folosind această funcționalitate, această aplicație prezintă utilizatorului o vizualizare grafică a frecvențelor sunetului (redată prin mai multe bare verticale alăturate, înălțimea fiecărei bare reprezentând nivelul sonor într-un interval de frecvență corespunzător fiecărei bare). Aceeași informație este simultan folosită pentru a calcula trei valori între 0 și 255, reprezentând intensitățile componentelor roșu, verde și albastru din culoarea finală. Scopul general

al acestui algoritm este ca fiecare din aceste intensități să depindă de nivelul sunetului dintr-o regiune a spectrului auditiv: componenta roșie pentru sunete joase, componenta verde pentru sunete medii, iar cea albastră pentru sunete înalte.

Acest algoritm folosește trei funcții de gradul doi suprapuse peste această vizualizare a spectrului pentru a determina influența nivelului de sunet din fiecare interval de frecvențe în valoarea componentei, iar, pentru fiecare interval, această influență este înmulțită cu nivelul sunetului în această frecvență. Acest algoritm nu calculează aceste influențe în spațiul obișnuit sRGB, deoarece acesta nu este un spațiu liniar (operația de adăugare a două culori nu conduce la un rezultat corect). Pentru ca aceste calcule să poată fi efectuate corect într-un spațiu liniar, trebuie ca, la final, valoarii fiecărei componente să-i fie aplicată o funcție *gamma*. Pentru simplitatea implementării, s-a aproximat funcția gamma ca fiind $\gamma\colon [0,1] \to [0,1], \gamma(c) = c^{2.2}$. Spațiul sRGB, folosit implicit pentru imagini, și a metodelor de a transforma din acesta într-un spațiu liniar, sunt descrise într-unul din standardele W3C, [16].

Controlul culorii se face prin trimiterea unei cereri HTTP POST la URL-ul "/set_color" cu un cod de culoare, în format RGB hexazecimal, sau prin deschiderea unei conexiuni WebSocket, pe subprotocolul "rgb-direct", și trimițând oricâte coduri de culoare, fiecare fiind setat cât mai repede posibil.

Acesta este codul controller-ului WebSocket:

```
class WsRgbDirectController(websockets.WebsocketsCtrlBase):
    def __init__(self):
        # argumentul False de aici permite folosirea
        # controller-ului fara fire aditionale de executie
        super(WsRgbDirectController, self).__init__(False)

    def on_thread_start(self):
        # functia on_thread_start este goala, deci
        # firul de executie se va inchide
        pass

    def on_in_message(self, message):
        # aceasta functie este apelata la primirea
        # unei culori, care este trimisa imediat
        # la daemon-ul detaliat mai jos
        rgb_utils.raw_set_color(message)
```

Controller WebSocket pentru schimbarea rapidă a culorii benzii RGB

Funcția `rgb_utils.raw_set_color` de mai sus scrie culoarea în format RGB într-un *named pipe* Unix localizat la "/var/run/rgb", din care o va citi un *daemon* implementat tot în limbajul C++, special pentru acest proiect, care va seta culoarea.

Majoritatea plăcuțelor de dezvoltare au un număr de pini de uz general care suportă *pulse width modulation* (PWM), dar numărul acestora este de obicei limitat. Sistemul Onion Omega 2 Plus furnizează, din nefericire, doar doi astfel

de pini, dar, fiind trei componente de culoare, acest proiect are nevoie de trei pini cu PWM, deci este nevoie de implementarea acestui sistem în software, într-un daemon. Astfel, un program scris în limbajul C++ rulează în fundal, primind pe pipe-ul localizat la "/var/run/rgb" culori, codate în format RGB hexazecimal, după care le decodează și, într-o buclă, la o frecvență de 200 de cicluri pe secundă, setează valorile a trei pini de uz general ai sistemului, cu scopul de a da aparența culorii corecte. Datorită frecvenței înalte și implementării foarte performante a algoritmului, culorile sunt clare, fără să se observe că modularea acestor semnale se face în software, și nu în hardware. Acest program este evoluat din aplicația "fast-gpio" din distribuția de Linux furnizată cu acest sistem, aplicație care furnizează PWM în software pentru un singur pin. Codul sursă a programului "fast-gpio" este disponibil la `https://github.com/OnionIoT/fast-gpio`, iar, în concordanță cu licența GPLv3 a acestui proiect, modificările proprii sunt disponibile la `https://github.com/TED-996/fast-gpio`.

De la pinii de uz general al plăcuței de dezvoltare, patru conexiuni electrice duc cele trei linii de control (una pentru fiecare componentă RGB) și borna "−" a plăcuței, la un ciruit electronic cu trei tranzistori "TIP31". Necesitatea acestora vine din faptul că o plăcuță de dezvoltare nu are capabilitatea de a controla curent de 12 volți, sau de a rezista intensității mare a curentului electric din aceste circuite. Din această cauză, ieșirile plăcuței sunt conectate la comenzile tranzistorilor, care au rolul de a comuta conexiunile electrice ale benzii de LED-uri. Acest circuit a fost proiectat cu ajutorul aplicației Autodesk Eagle[27] și materializat cu ajutorul unei mașini CNC. În figurile 6 și 7 se pot vedea schema, respectiv schița plăcuței de circuit pentru controlul LED-urilor.

Această aplicație Web, alături de sursele circuitului electronic, sunt disponibile la `https://github.com/TED-996/krait-rgb`.

9 Comparaţie cu alte servere Web similare

Pentru elaborarea acestei secțiuni, s-a ales pagina principală din aplicația Web folosită la testarea serverului (detaliată în secțiunea 7.2) și a fost implementată în *framework*-urile Django[28] și Flask[29]. Această pagină a fost aleasă din cauza faptului că folosește paradigma MVC și o largă varietate de instrucțiuni în limbajul de template (condiționale, repetitive, și de includere al unui alt fișier template).

În timpul implementării, au fost observate câteva diferențe între experiența de utilizare a acestor framework-uri și a acestui server Web. Prima diferență este că nici unul dintre acestea nu poate funcționa ca un server HTTP simplu, fără comportament, fără a fi configurat, o funcționalitate utilă în momentul transformării unei aplicații Web statice într-una dinamică. Dacă amândouă *framework*-urile studiate suportă directoarele statice (spre deosebire, acest server recunoaște în mod automat fișierele statice din directorul rădăcină), în cazul

[27] `https://www.autodesk.com/products/eagle/overview`
[28] `https://www.djangoproject.com/`
[29] `http://flask.pocoo.org/`

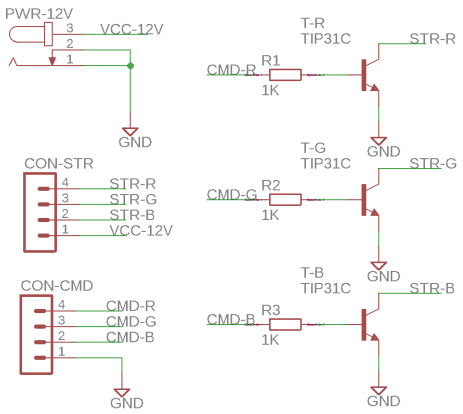

Figura 6. Schema plăcuţei de control. Liniile verzi cu acelaşi nume sunt considerate conectate între ele. Pe partea stângă sunt, de sus în jos, mufa de alimentare şi conexiunile cu banda LED, respectiv plăcuţa de dezvoltare, iar în partea dreaptă sunt trei ansambluri dintr-un tranzistor şi un rezistor de limitare a intensităţii curentului.

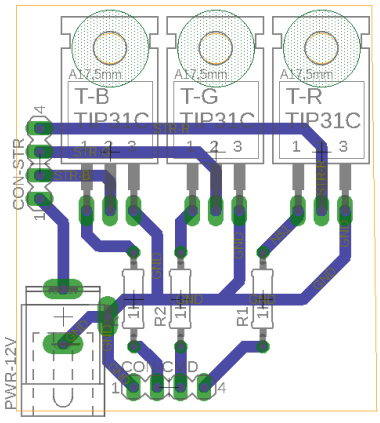

Figura 7. Schiţa organizării fizice a plăcuţei de control. în centru sunt cei trei tranzistori, mai jos cei trei rezistori şi conexiunea cu plăcuţa de control, iar în partea stângă, de sus în jos, conexiunea cu banda de LED-uri şi mufa de alimentare.

în care se folosește Django este imposibil ca aceste fișiere să fie în același loc cu scripturile la nivel de server. În aceste framework-uri, acest lucru este unul pozitiv, deoarece aceste script-uri trebuie să fie inaccesibile clienților HTTP. Abordarea serverului detaliat în această lucrare este de a permite fișierelor de template, fișierelor executabile și fișierelor statice să existe în același director, iar serverul să le distingă după câteva reguli foarte simple (în funcție de extensia fișierelor), iar fișierele ascunse față de clienții HTTP să fie marcate cu o convenție Unix clasică: prefixarea numelui unui fișier sau director cu caracterul ".".

Spre deosebire de acest server, care urmărește să aibă o interfață de programare cât mai simplă, ambele *framework*-uri studiate, dar, în special, Django, necesită învățarea unei cantități semnificative de concepte noi, configurarea serverului fiind deosebit de complexă. Capacitatea de reîncărcare a fișierelor serverului este inexistentă în proiectul Flask, dar cuprinzătoare în Django. În această implementare, template-urile și fișierele statice sunt reîncărcate, dar nu și modulele Python deja importate. Acest lucru oferă un echilibru dintre flexibilitatea evoluției server-ului și menținerea disponibilității acestuia (Django se închide și repornește la modificarea unui fișier.)

Flask nu are suport oficial, strucurat, pentru paradigma MVC, dar scripturile la nivel de server pot fi ușor scrise aderând la aceste principii. Echivalentul Django pentru această paradigmă, numit Model-Template-View (pe lângă faptul că redefinește unul din termenii cunoscuți, "view", la semnificația acceptată a controller-ului), este similară din punctul de vedere al conceptelor.

Fișierele template în *framework*-ul Django au o sintaxă relativ simplă, dar mai complexă decât limbajul PyML al acestui proiect, iar sintaxa template-urilor acestui *framework* este foarte diferită atât de limbajul HTML, cât și Python, limbajele cele mai folosite în elaborarea restului template-ului. Ca exemplu al complexității acestei sintaxe, instrucțiunea "`@variabila`" devine, în Django, "`{{ variabila }}`", fiind două puncte în care sintaxa pagini trebuie modificată (blocurile "{{" și "}}") și 6 caractere (inclusiv spațiile) în loc de unul. În sintaxa PyML, dacă acestă instrucțiune se termină cu un spațiu (datorită structurii textului), există un singur punct în care pagina trebuie modificată și un singur caracter. În caz contrar, modificările se vor face în două puncte, în total adăugând două caractere. În plus, limbajul de template al proiectului Django este foarte restrictiv și insuficient de compatibil cu limbajul Python. De exemplu, nu există nici o metodă de a rula secvențe mici de cod Python, care nu sunt relevante pentru a fi scrise în controller (de exemplu, calculul numelui unei clase HTML). Un alt exemplu este sintaxa instrucțiunii "`{% for … in … %}`", care nici nu suportă, nici nu dă un mesaj de eroare de sintaxă, pentru *destructurarea tuplurilor* din limbajul Python. (De exemplu, expresia "`{% for (t1_1, t1_2), (t2_1, t2_2) in lista_de_tupluri_de_tupluri %}`" va da un mesaj de eroare de genul "Este nevoie de 4 valori pentru despachetare, au fost primite 2".) Cu toate acestea, template-urile Django suportă mecanismul de moștenire al fișierelor de template, funcționalitate utilă în elaborarea interfețelor Web, dar care este relativ dificilă în acest moment în limbajul de template al acestei lucrări.

Fișierele template în Flask sunt rulate de motorul Jinja2[30], implementat pentru a emula template-urile Django, și îmbunătățind o mare parte din problemele de compatibilitate cu limbajul Python. Sintaxa a rămas nemodificată, fiind în continuare complexă, dar incompatibilitatățile menționate mai sus și alte restricții nenecesare asupra limbajului au fost rezolvate.

Un domeniu important al comparațiilor între implementările serverelor Web este performanța. Astfel, s-a luat implementarea aplicației de test în cele trei servere și s-au făcut 8000 de cereri HTTP din 10 procese, simulând concurența. Cinci procente dintre acestea au efectuat cereri HTTP adiționale pentru fișierele statice de care depinde această pagină HTML (foi de stiluri CSS, script-uri la nivel de client JavaScript și fonturi), simulându-se diferențele dintre o reîncărcare completă a unei pagini și luarea acestor resurse existente din cache-ul de pe disc al clientului HTTP. Acest program de test a fost implementat în limbajul Python, folosind biblioteca requests[31]. S-au făcut mai multe rulări, făcându-se media dintre acestea. Rezultatele au fost următoarele:

- **Krait**: 38 de secunde, în medie 4.75 milisecunde pentru fiecare cerere
- **Django**: 43.7 de secunde, în medie 5.45 milisecunde pentru fiecare cerere
- **Flask**: Pentru 8000 de cereri (pentru orice număr de clienți mai mare decât 1), conexiunile clientului au eșuat cu un mesaj de eroare. Fiindcă aceste erori nu au fost întâlnite la celelalte servere HTTP, aceasta ar indica prezența unei erori de implementare în acest server. Extrapolând de la o rulare cu 7200 de cereri cu 10 clienți, care a durat 41.8 secunde, timpul de execuție echivalent ar fi 46.4 secunde, în medie 5.8 milisecunde pentru fiecare cerere.

Atât Flask, cât și Django, suportă ca parte a interfeței acestora de programare funcționalități indisponibile în acest proiect. Cu toate acestea, scopul acestui proiect este o interfață simplă, utlizatorul având posibilitatea de a implementa sau de a folosi biblioteci independente de server pentru o parte din aceste sarcini. Astfel, se atinge un echilibru dintre funcționalitățile unui server și ușurința de a-l folosi.

10 Concluzii

Această implementare prezintă un server Web performant, cu o interfață de programare ușor de folosit, și cu suport pentru funcționalități extrem de utile în dezvoltarea aplicațiilor Web, ca suportul protocolului WebSocket și a paradigmei MVC.

O primă concluzie este că acest server demonstrează posibilitatea proiectării și implementării unei interfețe de programare pentru un server Web care să fie facil de utilizat în dezvoltarea unei aplicații Web, reducând la minim cantitatea de cunoștințe specifice serverului necesară pentru lucrul cu astfel de aplicații. Limbajul fișierelor *template* are o sintaxă ușor de înțeles, fiind în același timp

[30] `http://jinja.pocoo.org/docs/2.10/`
[31] `http://docs.python-requests.org/en/master/`

simplă, puternică și compatibilă cu limbajul HTML. în plus, suportul protocolului WebSocket permite cazuri de utilizare imposibile în protocolul HTTP, lucru demonstrat în ambele studii de caz din secțiunea 8. La final, automatul de analizare text are capabilitatea de a fi reutilizat cu succes pentru implementarea altor limbaje de programare, iar compilarea fișierelor template în fișiere sursă Python oferă un avantaj de performanță necesar în implementarea aplicațiilor Web.

10.1 Direcţii viitoare de dezvoltare

Acest server Web, chiar dacă poate fi utilizat cu succes în acest moment, poate fi îmbunătățit în continuare în câteva domenii:

- Implementarea anumitor părți ale protocolului HTTP rar folosite, în special de către clienți, ca suportul compresiei mesajelor HTTP, de exemplu cu algoritmul GZIP[32]
- Implementarea protocolului HTTP/2.0, cea mai nouă specificație a acestuia fiind RFC 7540 al IETF[33], acest protocol promițând eficiența semnificativ mai mare a conexiunilor HTTP, aceasta fiind făcută posibilă prin optimizarea protocolului pentru contextele moderne ale aplicațiilor Web
- Moștenirea fișierelor de template, permițând modularizarea mai ușoară a interfeței aplicațiilor (această moștenire este momentan posibilă, dar, din păcate, este dificilă)
- Modificarea serverului pentru a rula pe mașinile cu sistemul de operare Microsoft Windows
- Modificarea arhitecturii serverului la una de tip *preforked*

[32] `https://www.gnu.org/software/gzip/`
[33] `https://tools.ietf.org/html/rfc7540`

11 Referinţe

Bibliografie

1. Internet Engineering Task Force, RFC 7230 - Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing, `https://tools.ietf.org/html/rfc7230`
2. Internet Engineering Task Force, RFC 7234 - Hypertext Transfer Protocol (HTTP/1.1): Caching, `https://tools.ietf.org/html/rfc7234`
3. Internet Engineering Task Force, RFC 6455 - The WebSocket Protocol, `https://tools.ietf.org/html/rfc6455`
4. Internet Engineering Task Force, RFC 7578 - Returning Values from Forms: multipart/form-data, `https://tools.ietf.org/html/rfc7578`
5. Internet Engineering Task Force, RFC 6265 - HTTP State Management Mechanism, `https://tools.ietf.org/html/rfc6265`
6. Web Hypertext Application Technology Working Group, URL Living Standard, `https://url.spec.whatwg.org/`
7. Internet Engineering Task Force TLS Working Group, TLS 1.3 Draft Specifications `https://github.com/tlswg/tls13-spec`
8. Contribuitorii proiectului OpenSSL, Documentația OpenSSL, `https://www.openssl.org/docs/`
9. Python Software Foundation, Documentația Python, `https://docs.python.org/2/index.html`
10. Reenskaug, T., Models – Views – Controllers, `http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf`
11. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading, Massachusetts (1979)
12. Chapman, N.: LR Parsing: Theory and Practice. Cambridge University Press (1987)
13. Lutz, M., Ascher, D., Learning Python. O'Reilly & Associates, Inc., Sebastopol, CA, USA (1999)
14. Mozilla și contribuitorii individuali, Documentația Web File API, `https://developer.mozilla.org/en-US/docs/Web/API/File`
15. Mozilla și contribuitorii individuali, Documentația Web Audio API, `https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API`
16. Stokes, M., Anderson, M., Chandrasekar, S., Motta, R., A Standard Default Color Space for the Internet - sRGB, `https://www.w3.org/Graphics/Color/sRGB`, 1996