UNIVERSITATEA POLITEHNICA BUCUREŞTI
FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

LUCRARE DE LICENŢĂ

Smart Presentation Feedback
Comunicaţia client-server

Coordonatori ştiinţifici:
Prof. dr. ing. Adina Magda Florea
As. dr. ing. Andrei Olaru

Absolvent:
George-Cristian Stoica

BUCUREŞTI
2012

Rezumat

O prezentare ţinută în faţa unei audienţe numeroase reprezintă o activitate predominant unilaterală, în care cei prezenţi în audienţă nu pot interveni în niciun fel pe parcursul prezentării, astfel încât strângerea unui feedback relevant din partea acestora este dificilă. Acest lucru se întâmplă chiar şi în condiţiile în care majoritatea persoanelor deţin un dispozitiv smartphone sau chiar o tabletă, deci un suport electronic pe care ar putea urmări prezentarea şi prin care s-ar putea devolta o interacţiune între aceştia şi speaker.

Aplicaţia Smart Presentation oferă oportunitatea celor din audienţă să urmărească prezentarea făcută de speaker pe propriul dispozitiv Android, smartphone sau tableta, în mod sincronizat cu prezentarea speakerului sau nu. În plus, aplicaţia oferă posibilitatea acordării de feedback direct pe documentul prezentării, în timp real, astfel încât speakerul să aducă lămuriri sau să răspundă la întrebări chiar în timpul prezentării, fără intervenţia verbală a audientei. Speakerul are acces la forma agregată a feedbackului, extrem de util în cazul unei audiente mari.

Pentru realizarea acestei aplicaţii, am dezvoltat un sistem client-server, bazat pe cereri efectuate de clienţi, dispozitivele Android, către un server web, care pune la dispoziţie resurse ce pot fi accesate prin adresele lor URL. Resursele pot fi documentul prezentării sau feedback agregat adresat de cei din audienţă.

CUPRINS

1. Introducere

1.1 Contextul proiectului

Prezentările realizate în faţa unei audiente de mărime medie sau mare pot fi uneori greu de urmărit din diferite motive, cum ar fi incapacitatea persoanelor din audienţă de a înţelege anumite concepte din materialele prezentate, care duc la pierderea atenţiei, sau imposibilitatea acestora de a reveni asupra unor slide-uri anterioare.

Aceste prezentări ar putea deveni mai interactive, prin implicarea ascultătorilor încă din timpul audienței în procesul de apreciere pozitivă sau negativă a elementelor prezentate, precum şi prin posibilitatea urmăririi prezentării atât în timp real cât şi a revenirii asupra unor slideuri anterioare sau a avansării către slideuri următoare.

1.2 Ideea şi scopul proiectului

Ideea proiectului Smart Presentation este de a realiza o interacţiune strânsă între membrii audienței unei prezentări şi cel care ţine prezentarea (asupra căruia mă voi referi în continuare ca *speaker*). [1]

Ca precondiţii, se presupune că atât speakerul, cât şi cei din audienţă posedă câte un dispozitiv mobil, tableta sau telefon mobil, având instalat sistemul de operare Android. De asemenea este necesar un server conectat la un router wireless, pentru a permite conectarea cu dispozitivele Android.

Funcţionalitatea aplicaţiei porneşte de la posibilitatea membrilor audientei de a urmări pe propriul dispozitiv Android prezentarea ţinută de speaker. Speakerul este cel care porneşte prezentarea, care schimbă slide-urile şi care termină prezentarea. Cei din audienţă au posibilitatea de a urmări prezentarea în timp real şi pe dispozitivul lor într-un mod *live*, sau pot naviga liber prin restul prezentării. În plus, aceştia pot furniza feedback în timp real prezentării astfel: pot selecta porţiuni din prezentare, asupra cărora pot furniza feedback pozitiv, de apreciere, feedback de ambiguitate, prin care se remarcă necesitatea unor clarificări ale acelor elemente sau feedback de cerere a unor dovezi (proof). De asemenea, cei din audienţă pot pune întrebări legate de elementele selectate, sau pot alege să adere la o întrebare deja pusă, aceştia având posibilitatea de sincronizare a feedbackului cu cel acordat de toţi cei aflaţi în audienţă. În plus, aceştia vor avea şi posibilitatea retragerii propriului feedback, dacă ulterior nu îl mai consideră necesar. La finalul prezentării, speakerul poate vedea o formă agregată pentru toate tipurile de feedback. În cazul întrebărilor adresate speakerului referitor la o selecţie făcută pe prezentare, agregarea se face prin alegerea unor întrebări reprezentative din punct de vedere semantic şi gruparea (eng. c*lustering*) celorlalte întrebări în jurul acestora.

Comunicarea între dispozitive şi sincronizarea elementelor de feedback acordate prezentării se face prin intermediul unui server central care depozitează datele agregate primite din partea tuturor clienţilor cu care se comunică wireless. Serverul este şi cel care conţine prezentarea în format PDF, aceasta fiind descărcată pe fiecare dispozitiv mobil care accesează aplicaţia SmartPresentation.

1.3 Structura proiectului

Proiectul a fost structurat în patru module cu funcționalități diferite, fiind cinci persoane implicate în dezvoltarea aplicaţiei. Cele patru arii sunt:

[1] Pagina web cu ideea proiectului, 20.06.2012, <http://aimas.cs.pub.ro/androidEDU/>

- modulul de manipulare a prezentării PDF. Acesta presupune folosirea unei biblioteci open source de manipulare a formatului PDF, pentru a permite navigarea prin prezentare, selectarea unor elemente ale prezentării prin folosirea ecranului touchscreen al dispozitivului Android şi evidenţierea selecţiei făcute prin diverse culori de highlighting.
- interfaţa clientului pe Android, care include toate elementele vizuale prin care clientul interacţionează cu aplicaţia (ferestre, butoane, liste).
- modulul de comunicaţie client-sever, care asigură transmisia resurselor între dispozitive şi server şi sincronizarea acestora.
- modulul de clusterizare a întrebărilor puse de audință pe baza similarităţilor semantice.

Partea mea de proiect a constat în implementarea modulului de comunicaţie între clienţii Android şi server. Acest modul include partea de stocare a datelor atât pe Android, cât şi pe server, dezvoltarea serverului web şi a protocolului de comunicaţie bazat pe HTTP şi Google Protocol Buffers, obţinerea resurselor prin URLuri, tipul mesajelor şi modalitatea de serializare a acestora, precum şi interfaţarea pe server cu modulul de clusterizare a întrebărilor şi interfaţarea pe sistemul de operare Android cu modulul de manipulare a PDFurilor şi cu interfaţa de utilizare.

1.4 Structura lucrării

În continuare, capitolele acestei lucrări sunt structurate astfel: aspecte teoretice ale proiectului, urmate de tehnologiile folosite la implementarea proiectului, cu detalierea celor folosite la comunicaţia client-server şi la crearea protocolului de comunicaţie. Urmează arhitectura aplicaţiei, în care sunt prezentate diferitele module ale aplicaţiei şi interacţiunea dintre ele, din nou cu detalierea celor de backend direct implicate în schimbul de date între clienţi şi server. Capitolul detaliilor de implementare prezintă metodele de programare folosite, oferind spre exemplificare porţiuni de cod explicate. În cadrul acestui capitol este descris protocolul de comunicaţie între server şi clienţi, tipurile de URLuri folosite pentru accesul resurselor, tipurile mesajelor schimbate şi diferitele tipuri de serializare. Mai apare şi detalierea interfaţării dintre server şi modulul de clusterizare a întrebărilor de feedback.

În final, capitolul de utilizarea a aplicaţiei prezintă în detaliu aplicaţia şi modalitatea de utilizare interfeţei grafice, prin screenshoturi şi înfăţişarea unor diverse scenarii de utilizare.

2. Tehnologii folosite

2.1 Sistemul de operare Android

Android este un sistem de operare pentru dispozitive mobile, telefoane sau tablete. Android a devenit lider mondial în acest segment în 2010, în primul trimestru al anului 2012 raportând o cotă de 59% din piaţa mondială a smartphoneurilor, cu un total aproximat la 331 milioane de dispozitive cu acest sistem de operare instalat[1].

Android a început de la o companie mică, achiziţionată de Google în 2005. În prezent, dezvoltarea sistemului de operare este supervizată de Open Handset Alliance, o comunitate condusă de Google din care mai fac parte companii că ARM Holdings, HTC, Inel, LG, Motorola, Samsung Electronics, Nvidia, T-Mobile sau Texas Instruments.

Kernelul sistemului de operare Android se bazează pe kernelul de Linux. Acest kernel a suferit modificări de arhitectura realizate de inginerii Google în afara procesului tipic de devoltare a kernelului Linux. Android nu are un sistem nativ X Window şi nu suportă întregul set de biblioteci standard GNU, ceea ce face destul de dificilă portarea aplicaţiilor şi bibliotecilor existente de Linux pe Android. Principalele modificări pe care Google le-a adus într-o prima faza kernelului au fost legate de eficientizarea consumului bateriei. Aceste modificări au fost respinse în prima faza de dezvoltatorii kernelului Linux de bază, motivând lipsa de încredere în intenţia Google de a se ocupa în continuare de acest cod. În 2010 s-a făcut un pas major pentru integrarea modificărilor din Android în Linux, prin adăugarea unui patch care îmbunătăţea frameworkul de wakeup events existent şi care permitea driverelor Android să fie uşor integrate în Linux de bază. În August 2011 Linus Torvalds spunea că în patru sau cinci ani Linux şi Android se vor întoarce la un kernel comun [1].

Arhitectura Android presupune existenţa a patru layere, cel de la baza fiind kernelul de Linux. Al doilea layer, cel de middleware, conţine biblioteci de C. Acesta este cod nativ, rulând direct peste cel de Kernel. Următorul layer este cel de application framework, care cuprinde biblioteci compatibile Java, bazate pe versiunea de Java Apache Harmony.

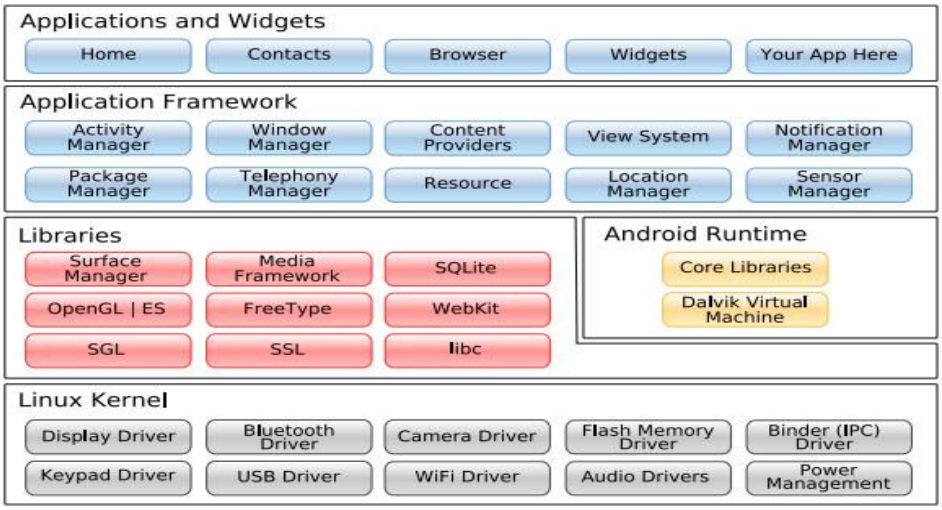


Fig. 1 Arhitectura Android [2]

[1] Wikipedia, The Free Enciclopedia, 20.06.2012,
< http://en.wikipedia.org/wiki/Android_(operating_system)>

[2] Kebomix blog, 20.06.2012, <http://kebomix.wordpress.com/2010/08/17/android-system-architecture/>

Maşina virtuală care face translaţia din codul Java în byte-code se numeşte Dalvik virtual machine, şi se deosebeşte de maşina virtuală clasică JVM prin faptul că nu este o maşină bazată pe stivă, ci una bazată pe regiştri. Un tool numit **dx** este folosit pentru a converti o parte a fişierelor .class Java într-un format .dex. Mai multe clase sunt incluse într-un singur fişier .dex. Stringurile duplicate şi alte constante folosite în mai multe fişiere class sunt incluse doar o dată în outputul .dex, pentru conservarea spaţiului. Java bytecode-ul este de asemenea convertit într-un set de instrucţiuni definit de Dalvik Virtual Machine. Un fişier necomprimat .dex este sensibil mai mic decât o arhivă .jar, derivată din aceleaşi fişiere .class. O altă diferenţă majoră faţă de clasicele JVM este introducerea compilatorului JUST-IN-TIME, care reprezintă o metodă hibridă faţă de cele două metode clasice de runtime (intrepretat sau static – cod compilat). Astfel, acest compilator translatează codul din bytecode în machine code la runtime, înainte de a-l rula nativ.

Fiecare aplicaţie Android rulează în propriul proces cu propria instanţă a maşinii virtuale. Dalvik a fost dezvoltat în aşa fel încât un dispozitiv poate rula mai multe maşini virtuale eficient. Maşina virtuală Dalvik se bazează pe kernel-ul Linux pentru funcţionalităţile de bază, cum ar fi gestionarea thread-urilor şi menţinerea nivelului de memoriei scăzut.

Layerul de application framework cuprinde acele servicii scrise în Java care fac legătura între aplicaţii şi sistemul de operare, fiind separate pe diverse funcţionalităţi: Activity Manager, Content Providers, Resource Manager, Notification Manager, View System, Telephony Manager, Location Manager şi altele. Layerul de top este cel al aplicaţiilor, unde dezvoltatorii pot adăuga noi aplicaţii utilizând API-ul existent sau pot modifica aplicaţiile deja existente.

În prezent, Android are o vastă comunitate de dezvoltatori care scriu aplicaţii („apps”) care extind funcţionalitatea dispozitivelor. Dezvoltatorii folosesc în principal codul versiunii custom de Java. Applicațiile pot fi descărcate de pe magazine online ca Google Play (fostul Android Market), magazinul condus de Google. În octombrie 2011 erau mai mult de 500 000 aplicaţii disponibile pentru Android.

În cadrul proiectului Smart Presentation, o mare parte din devoltare s-a făcut pe sistemul de operare Android, folosind versiunea 2.3 a sdk-ului Android. Dezvoltarea s-a făcut în Eclipse, acesta fiind mediul standard şi global folosit pentru crearea aplicaţiilor Android, datorită pluginului Android şi a emulatorului care poate fi lansat direct din interfaţa IDEului.

În cadrul dezvoltării, am folosit atât cod nativ C, cât şi cod standard Java. Codul nativ face parte din biblioteca mupdf, pe care am folosit-o la manipularea prezentării în format PDF, pentru partea de selecţie. Codul nativ a fost compilat folosind toolul ndk, biblioteca rezultată putând fi încărcată direct în codul de Java.

Pentru modulul client-server, am folosit design patternul Model-View-Controller, detaliat la capitolul Arhitectura sistemului. Pentru implementarea acestui pattern am folosit clasele existente în API-ul Android, care încurajează separarea ariei funcţionale, comunicarea pe retea sau persistenţa datelor, de interfaţa grafică a aplicaţiei, pentru a se asigura un flow continuu al interfeţei, fără întreruperi care ar dăuna calităţii utilizării. Principalele mecanisme folosite sunt cel de ContentProvider, care returnează activităţii de frontend conţinut pe baza unor interogări (echivalent conceptului de Model). În cadrul ContentProviderului, persistenţa datelor este asigurată printr-o bază de date SQLlite, iar comunicarea cu serverul wireless se face asincron, în cadrul unor threaduri separate. Pentru rularea acestor taskuri, API-ul Android pune la dispoziţie numeroase soluţii, cea mai populară fiind cea de a extinde clasa AsyncTask, aceasta oferind metode handler pentru execuţie şi pentru returnarea rezultatelor.

2.2 Tehnologiile specifice folosite pentru comunicarea client-server

Pentru crearea serverului central, responsabil cu centralizarea feedbackului de la clienţi, distribuţia acestuia către clienţi şi persistenţa datelor, am avut de ales între multiple tehnologii disponibile, cum ar fi realizarea unei comunicaţii asincrone pe socketi sau implementarea unui mecanism de Remote procedurce calling. Alegerea cea mai potrivită mi s-a părut în final implementarea unui web service, datorită protocolului de nivel superior, Hypertext transfer protocol, care asigură o bază pentru dezvoltarea unui mecanism facil de acces la date. De asemenea, acest protocol permite schimbul datelor în mai multe formate, atât de tip binar cât şi plain text, prin completarea headerului de mime-type.

Un alt punct forte al alegerii unui sistem client-server bazat pe protocolul HTTP este portabilitatea, fiecare limbaj de programare având biblioteci care facilitează crearea mesajelor HTTP şi transmiterea/recepţionarea lor pe baza URLului. Datorită folosirii unui limbaj compatibil Java pe Android, eu am ales tot Java pentru devoltarea serverului web. Tehnologia Java pentru crearea web serverelor se bazează pe o arhitectură de tip servlet-container, care permite definirea unor scripturi Java (servlets) şi înregistrarea acestora, de obicei într-un fişier de configurare xml, pentru a fi încărcate dinamic într-un container web. Un container web este acea componentă a unui serviciu web care interacţionează cu servleturile şi este responsabilă cu managementul ciclului de viaţă a acestora şi cu maparea lor la un URL. Un web container implementează contractul unei componente web Java EE, specificând un mediu de runtime care asigură securitatea, concurenţa, managementul ciclurilor de viata, completarea tranzacţiilor şi alte servicii.

În cazul acestei aplicaţii, pentru implementarea serverului web am ales să folosesc un web container Glassfish, bazat pe popularul Tomcat. Pentru interacţiunea cu acest container am folosit API-ul Grizzly, care va fi detaliat într-un subcapitol următor. Pentru maparea resurselor http unor metode http şi a unor URLuri, am folosit frameworkul Java Jersey, o soluţie care respectă arhitectura RESTful, cea mai folosită în momentul de faţă pentru dezvoltarea serviciilor web.

Protocolul HTTP

Hypertext Transfer Protocol, pe scurt HTTP, este un protocol la nivelul aplicaţie care stă la fundaţia comunicaţiei în World Wide Web. Hypertext reprezintă un set de obiecte care construiesc conexiuni între noduri prin legături logice, hyperlinks. HTTP este protocolul care permite transferul unor astfel de obiecte[1].

HTTP este un protocol de tip cerere-răspuns în modelul arhitectural client-server. Un client, cel mai des un web browser, trimite o cerere HTTP către un server web, care întoarce un răspuns. Acest răspuns conţine o stare al completării cererii şi, în caz de succes, informaţia cerută.

Resursele HTTP sunt identificate şi localizate pe rețea prin Uniform Resource Locators (URLs) folosind schema http URI definită în RFC 3986 astfel:

<nume_schema> : <porţiunea_ierarhică> [? <query>] [# <fragment>]

În cazul HTTP numele schemei este http. Partea ierarhică începe cu un dublu forward slash "//", urmat de un nume al autorităţii, care poate fi un nume de domeniu sau un ip, urmat apoi de un port opţional, precedat de ":". După autoritate urmează un path construit ca o succesiune de segmente, asemănător unei structuri de directoare, caracterul separator fiind "/". Porţiunea de query este opţională, începe cu "?" şi conţine informaţie adiţională care nu este ierarhică, ci organizată tipic prin perechi de tip <cheie>=<valoare>, cu perechile separate prin ";" sau "&". Partea fragmentului este de

[1] Wikipedia, The Free Enciclopedia, 20.06.2012, < http://en.wikipedia.org/wiki/Http>

asemenea opţională, începe cu "#" şi conţine o directivă către o resursă secundară, cel mai des un atribut id al resursei principale, aşa cum se întâmplă în cazul documentelor HTML.

HTTP defineşte nouă metode care indică acţiunea dorită asupra resursei accesate. Aceste nouă metode sunt: HEAD, GET, POST, PUŢ, DELETE, TRACE, OPTIONS, CONNECT şi PATCH. Cele mai utilizate sunt cele de GET, prin care se citeşte resursa, şi cele de POST, PUT şi DELETE, prin care se modifică resursa. Metodele safe (sigure) sunt considerate cele prin care se intenţionează doar citirea informaţiei, fără modificarea stării serverului. Acestea sunt HEAD, GET, OPTIONS şi TRACE. Metodele idempotente sunt cele asupra cărora mai multe cereri identice ar trebui să aibă acelaşi efect. Acestea sunt POST şi DELETE.

O cerere HTTP conţine următoarele :

- o linie de request, spre exemplu GET /diverse/car.jpg HTTP/1.1 care formulează o cerere pentru resursa /diverse/car.jpg de la server.
- Headere Http, cum ar fi Accept-Language: en-US, Accept-Charset: utf-8, etc.
- O linie goală
- Un corp al mesajului, opţional

Un răspuns HTTP conţine următoarele:

- linie de Status (spre exemplu HTTP/1.1 200 OK, care indică finalizarea cu succes a cererii clientului). Un status 3xx indică necesitatea unei redirectari, 4xx şi 5xx sunt statusuri de eroare (4xx – eroare la client, 5xx- eroare la server).
- Headere HTTP, cum ar fi Content-Type: text/html
- O linie goală
- Un corp al mesajului, opţional

Antetele HTTP sunt perechi "cheie: valoare", scrise în clear-text şi separate prin succesiunea de caractere carriage return (CR) şi line feed (FD). Aceste headere definesc parametrii de operare a unei tranzacţii HTTP.

Datorită necesităţii trasmiterii unor tipuri diferite de conţinut prin mesajele HTTP, am folosit în cadrul aplicaţiei Smart Presentation setarea headerului Content-Type prin care se specifică mime-typeul conţinutului. Un **mime-type** (Multipurpose Internet Mail Extension) este un standard Internet care a pornit de la descrierea seturilor de caractere folosite la formatul email, ajungând azi să fie utilizat ca standard de descriere al conţinutului unui mesaj web în general. Un astfel de antet descrie atât conţinuturi de tip text, cât şi conţinut binar. În cazul clienţilor şi al serverului, interpretarea acestui câmp este vital pentru alegerea modului în care resursa este citită, respectiv scrisă.

Pentru aplicaţia Smart Presentation, am folosit următoarele tipuri mime:

- text/plain – defineşte un conţinut în text clar, necodat.
- application/pdf – indică prezenţa unui fişier PDF în interiorul mesajului, în format binar
- application/x-protobuf – indică un conţinut binar, serializat cu Google Protocol Buffers. Este datoria clientului şi a serverului de a serializa, respectiv deserializa acest format binar.

Grizzly Web Container

Grizzly este un framework web născut din dorinţa de a ajuta dezvoltatorii să profite de avantajele API-ului Java NIO (Java new I/O). Este devoltat de comunitatea GlassFish, sub conducerea Oracle[1]. Până la apariţia acestui API, scrierea unor aplicaţii server scalabile în Java era foarte dificilă, problemele de thread management făcând imposibilă scalarea unui server la mii de utilizatori. Scopul Grizzly este dezvoltarea unor sisteme server robuste şi scalabile, oferind componente extinse ale frameworkului Java NIO ca:

- un framework HTTP Protocol pentru crearea unor aplicaţii HTTP costumizate;

- un framework HTTP Server care oferă abstractizări la nivel înalt ale protocolului HTTP (similare Servlets).

Grizzly foloseşte un sistem keep-alive bazat pe clasele Selector ale NIO, care suportă monitorizarea conexiunii şi previn atacurile de tip Denial-of-Service. Sistemele Denial-of-Service adăugă suport pentru validare IP, numărul tranzacţiilor completate per IP, detecţia conexiunilor inactive, cu scopul de a preveni epuizarea sau atacurile "flooding". Toate aceste servicii sunt folosite în conjuncţie cu sistemele Keep-alive. Conectorul Grizzly va înainta cererile pentru resursele statice şi dinamice către containerul servleturilor, care procesează cererile pentru resurse statice într-un servlet dedicat (org.apache.cătălina.servlets.DefaultServlet) [5].

Servicii Web RESTful şi Jersey JAX-RS

Representational State Transfer (REST) reprezintă un stil arhitectural bazat pe standardele web şi pe protocolul HTTP, pentru sisteme distribuite că World Wide Web. REST s-a evidenţiat în ultimii ani ca modelul de design predominant al web-ului, datorită simplităţii sale. REST a fost descris în 2000 de Roy Fielding, în lucrarea sa de doctorat[2].

Într-o arhitectură bazată pe REST, totul este o resursă. O resursă este accesată printr-o interfaţă comună bazată pe metodele standard HTTP. Într-o arhitectură REST, există tipic un server REST care asigură accesul la resurse şi clienţi REST care accesează sau modifică resursele. Fiecare resursă ar trebui să suporte operaţiile standard HTTP (GET, POST, PUŢ, DELETE). Resursele sunt identificate prin ID-uri globale – URIuri.

"Limbajul" REST este bazat pe substantive şi verbe şi pune accentul pe lizibilitate. Spre deosebire de SOAP, REST nu necesită parsare XML şi nu necesită un header al mesajului de la/către un service provider, ceea ce duce la reducerea bandwidth-ului consumat. REST are şi o altă metodologie de manipulare a erorilor: în timp ce SOAP poate avea mesaje definite de eroare, REST necesită folosirea mecanismelor HTTP de error handling .

Stilul arhitectural REST impune anumite direcţii de luat în considerare la realizarea designului, implementarea componentelor individuale fiind la discreţia dezvoltatorului:

Client-server – o interfaţă uniformă separă clienţii de servere. Separarea preocupărilor înseamnă că, spre exemplu, clienţii nu se ocupă de stocarea datelor, aşa cum serverul nu se ocupă cu interfaţa utilizatorilor sau cu starea clienţilor. Astfel, serverele şi clienţii pot fi dezvoltaţi independent, interfaţa rămânând constantă.

[1] Grizzly website, 20.06.2012, < http://grizzly.java.net/>

[2] Roy Fielding, "Architectural Styles and the Design of Network-based Software Architectures", 20.06.2012, <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

- Stateless – acest principiu asigură că niciun context al clientului nu este memorat pe server între cereri. Fiecare cerere efectuată conţine toate informaţiile necesare, şi orice informaţie de stare a sesiunii este stocată pe client. Aceasta este o caracteristică de rezistenţă la erori a serverelor.
- Cacheable – clienţii pot reţine în cache răspunsurile. O memorie cache bine întreţinută poate îmbunătăţi scalabilitatea şi performanţa sistemului, prin reducerea numărul de cereri de la clienţi la server.
- Layered – un client nu-şi poate da seama prin interfata comună dacă este conectat direct la server sau la un proxy intermediar. Servere intermediare pot îmbunătăţi scalabilitatea sistemului prin load-balancing sau prin partajarea unei memorii cache.
- Code on demand (opţional) – serverele pot să extindă temporar funcţionalitatea clientului prin transferul codului - exemple fiind Java applets sau limbajele client-side scripting ca JavaScript.

Un WebService RESTful se bazează pe metodele HTTP şi pe conceptele arhitecturii REST. Acesta defineşte tipic un URI de baza pentru resurse şi tipurile MIME suportate (XML, Text, JSON, Protocol Buffers, etc) şi setul de operaţii HTTP. Aceste metode standard sunt [6]:

- GET – defineşte un acces la citirea unei resurse, fără efecte secundare. Resursa nu este niciodată alterată în urma unei cereri GET.
- PUT – crează o nouă resursă, trebuie să fie idempotentă.
- DELETE – şterge o resursă; operaţia trebuie să fie idempotentă, o repetare a cererii nu trebuie să producă efecte suplimentare primei cereri.
- POST – actulizează resursa existentă sau crează o nouă resursă.

Jersey reprezintă implementarea Java open source, de referinţă, la calitate de producţie a standardului JAX-RS (JSR 311) pentru dezvoltarea web serviceurilor RESTful. Dar este mai mult decât implementarea referinţei, oferind şi un api prin care programatorii pot extinde Jersey pentru nevoile proprii [7]:

Standardul JAX-RS implementat de Jersey suportă următoarele anotatii (annotations) pentru crearea resurselor web:

Adnotare	Descriere
@PATH(my_path)	setează patul la URL de baza + "/" + my_path. URL-ul de baza este cel definit ca URL al containerului de bază, în web.xml sau în aplicaţie, în cazul folosirii unui container că Grizzly
@POST	indică faptul că metoda va răspunde unei cereri POST
@GET	indică faptul că metoda va răspunde unei cereri GET
@PUT	indică faptul că metoda va răspunde unei cereri PUT
@DELETE	indică faptul că metoda va răspunde unei cereri DELETE
@Produces(mime-type [, more types])	defineşte ce mime-type va avea conţinutul returnat l ao metoda adnotată cu @GET. Mime-tipurile pot fi de tip "plain/text" sau binare, că "application/pdf" sau "application/x-protobuf". Alte exemple: "application/xml", "application/json"

@Consumes(mime-type [, more types])	Defineşte ce mime-type este consumată de această metodă, PUŢ sau POST
@PathParam	Folosita pentr a injecta valori din URL că parametru al metodei. Se foloseşte spre exemplu la obţinerea ID-ului resursei din cadrul URLului, pentru returnarea resursei corecte
@QueryParam	Leagă parametrul de o valoarea unui parameru de interogare HTTP
@HeaderParam	Leagă parametrul de valoarea unui header HTTP

Google protocol-buffers

Protocolul HTTP este un protocol care permite definirea oricărui format de serializare a conţinutului propriu-zis al pachetului, atât timp cât şi emiţătorii şi receptorii definesc programatic metode de citire şi scriere a acelor mesaje. Limbajul *de facto* folosit încă pentru schimbul de date este XML, datorită simplităţii sale şi uşurinţei de citire. Totuşi, din dorinţa optimizării utilizării benzii de transfer , în dauna lizibilităţii, în cazul mesajelor mari sau a situaţilor în care există necesitatea schimbului unui număr foarte mare de mesaje, au câştigat teren şi formatele de serializare binare [9].

Pentru serializarea mesajelor de mărime potenţială mare, am folosit soluţia Google Protocol Buffers, un mecanism extensibil, neutru din punct de vedere al limbajului şi al platformei pentru serializarea binară a datelor, reprezentând o variantă mult mai eficientă faţă de XML şi JSON. Acest mecanism a fost dezvoltat de Google pentru uz intern, aceştia creând compilatoare pentru C++, Java şi Python disponibile publicului larg printr-o licenţă open source.

Designul Protocol Buffers pune accentul pe simplitate şi performanță. A fost devoltat special pentru a crea o alternativa mai compactă şi, deci, mai rapidă decât XML. Metoda de creare a mesajelor Protocol Buffers are la bază un limbaj de descriere a interfeţei (IDL) care descrie structura datelor într-un format foarte simplu. Aceste "mesaje" sunt definite într-un fişier de definire Proto (.proto) care este compilat cu un executabil protoc. Compilarea generează cod care poate fi invocat de destinatarul sau receptorul acestor structuri de date. Clasele generate pun la dispoziţia programatorului metode Get şi Set pentru câmpurile mesajelor, asigurând citirea şi manipularea facilă a acestora. De asemenea, se pot defini mesaje încapsulate în cadrul altor mesaje, care în mod programatic vor fi translatate în clase inner. O altă facilitate a claselor generate dinamic este posibilitatea obţinerii unei instanţe a mesajului direct din streamul de octeţi.

În mod canonic, Protocol Buffers sunt serializate într-un format binar care este compact, forward-compatible şi backwards-compatible. Datorită simplităţii şi a vitezei sale, acest format şi-a extins popularitatea dincolo de comunicarea pe retea la cea între procese sau sisteme scrise în limbaje de programare diferite, care suportă serializare/deserializare Google Protocol Buffers.

```
message MessageExample
{
  required int32 id = 1;
  requird string messageName = 2;
  enum Type
  {
       TYPE1 = 1;
       TYPE2 = 2;
  }
```

```
  required Type type = 3;
  message EmbeddedMessage
  {
       required int64 id = 1;
       required string label = 2 [default = "SAMPLE"];
  }
 repeated EmbeddedMessage internalLabels = 4;
}
```

Exemplu Protocol Buffers

După cum se poate observa, formatul .proto este foarte intuitiv, foarte facil de scris şi permite o mapare la tipuri de date şi structuri de date de bază în toate limbajele de programare populare: stringuri, tipuri numerice de date, liste, dar şi concepte de programare orientată obiect, cum este încapsularea.

Astfel, eficienţa acestui mecanism de serializare este dată atât de dimensiunea extreme de redusă a pachetului care este transferat pe retea, cât şi de de modul foarte succinct şi compact de definire a formatului mesajului. Aceste aspecte sunt net superioare limbajului XML, motiv pentru care foarte multe aplicaţii folosesc astăzi acest mecanism de serializare. Dezavantajul Google Protocol Buffers faţă de XML este acela că nu e human-readable, fiind un limbaj binar prea puţin important însă pentru aplicaţii care nu pun accentul pe urmărirea conţinutului mesajului între procesele de serializare şi deserializare.

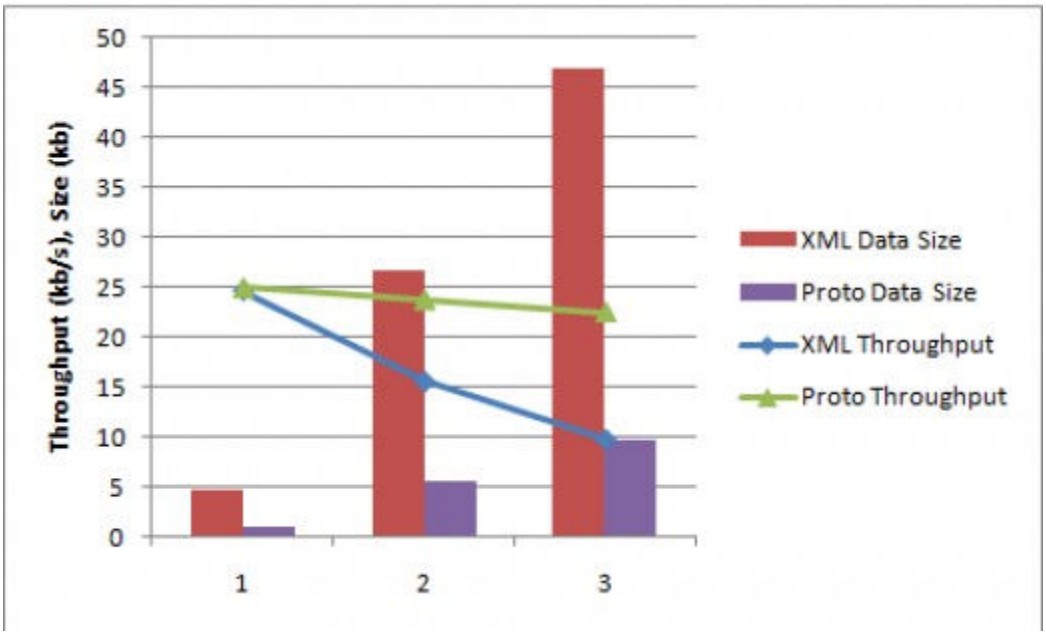


Fig. 2 Comparaţie Xml – Google Protocol Buffers[1]

În cadrul proiectului Smart Presentation, am folosit Protocol Buffers la transmiterea între client şi server a feedbackului acordat de ascultători sub forma de întrebări puse asupra unor elemente selectate din prezentare. Serverul este cel responsabil de serializarea clusterului de întrebări, mesajul proto fiind prezentat în detaliu la capitolul detaliilor de implementare.

[1] Alternative to XML – Protocl Buffers. 20.06.2012, <http://afrozahmad.hubpages.com/hub/protocolbuffers>

2.3 Alte tehnologii folosite în cadrul proiectului

Formatul PDF

Pentru reprezentarea documentelor am ales formatul PDF (Portable Document Format), motivul principal fiind reprezentat de popularitatea acestui şi de unele avantaje tehnologice clare.

PostScript care este limbajul de programare interpretat aflat la baza formatului. Scopul său principal este de a descrie modul de randare a textului, formelor şi imaginilor grafice. Acesta oferă, de asemenea, un cadru pentru controlul dispozitivelor de imprimare. Deşi PostScript şi PDF sunt înrudite, sunt formate diferite. PDF foloseşte capacitatea limbajului PostScript de randare de stiluri complexe de text şi grafică şi aduce această caracteristică atât pe ecran, cât şi la imprimare. Pentru PDF s-a ales o flexibilitate redusă în favoarea unei eficienţe şi unei predictibilități mai bune. Spre deosebire de PostScript, PDF poate conţine o mulţime de structuri de document, link-uri, precum şi alte informaţii conexe, dar nu poate schimba rezoluţia, sau să folosească orice alte caracteristici specifice hardware.

Sintaxa PDF poate fi împărţită în patru părţi [4]:

- Obiecte. Un document PDF este o structură de date compusă dintr-un set redus de tipuri de obiecte. PDF include opt tipuri de baza de obiecte: valori boolene, numere întregi şi reale, şiruri de caractere, nume, vectori, dicţionare, fluxuri de date şi obiectul null.
- Structura fişierului. Structura fişierului PDF determină cum sunt stocate în fişier obiectele, cum sunt accesate şi cum sunt modificate. Această structură este independentă de sintaxa obiectelor.

  Structura fişierului PDF este formată din următoarele patru elemente:

  - Un antet format dintr-o singura linie specificând versiunea PDF.
  - Un corp conţinând obiectele ce compun documentul.
  - Un tabel cross-reference conţinând informaţii despre obiectele indirecte din fişier
  - Un trailer oferind locaţia tabelului cross-reference.

  Structura inițială poate fi modificată ulterior, ceea ce adăugă elemente adiţionale la finalul fişierului.
- Structura documentului. Structura documentelor PDF specifică modul cum obiectele de bază sunt folosite pentru reprezentarea componentelor unui document: pagini, fonturi, adnotări etc.

  Catalogul documentului este un dicţionar care conţine referinţe la alte obiecte care definesc conţinutul, cuprinsul, threaduri de articole, nume de destinaţii şi formulare interactive

  Paginile documentului sunt acesate printr-o structură cunoscută ca arborele de pagini (eng. page tree), care defineşte ordonarea de pagini în document. Folosind această structură arborescentă aplicaţiile de citit PDFuri care folosesc o cantitate limitată de memorie, pot deschide imediat un document conţinând sute de pagini. Arborele conţine noduri de două feluri: noduri interne reprezentate de arborii de pagini şi frunze reprezentate de obiectele pagină.
- Fluxuri de date. Un flux de date PDF conţine o secvenţă de instrucţiuni care descriu modul de apariţie al paginii sau alte entităţi grafice. Aceste instrucţiuni, deşi sunt reprezentate că obiecte, sunt diferite din punct de vedere conceptual de obiectele folosite în descrierea structurii documentului.

Datele dintr-un flux de date de conţinut sunt interpretate ca o secvenţă de operatori şi operanzii acestora. Un flux de date de conţinut poate descrie modul de prezentare a unei poze sau poate fi tratat ca un element grafic în alte contexte.

Sistemul de coordonate defineşte spaţiul în care randarea are loc. Determină poziţia, orientarea şi dimensiunea textului, graficelor şi imaginilor ce apar pe pagină (Figura 3).

Conţinutul unei pagini apar în cele din urmă pe un dispozitiv de ieşire raster, cum ar fi un ecran sau o imprimantă. Asemenea dispozitve variază în sistemul de coordinate folosit pentru a adresa pixeli. Sistemul particular al unui dispozitiv se numeşte spaţiul dispozitv (eng. *device space*). Originea sistemului de coordinate al dispozitivelor poate fi în locuri diferite, de asemnea şi axele pot avea orientări diferite.

Pentru a evita dependenţa de dispozitiv, PDF defineşte un sistem de coordonate independent de dispozitiv. Acest sistem de coordonate se numeşte spaţiul utilizator (eng. *user space*).

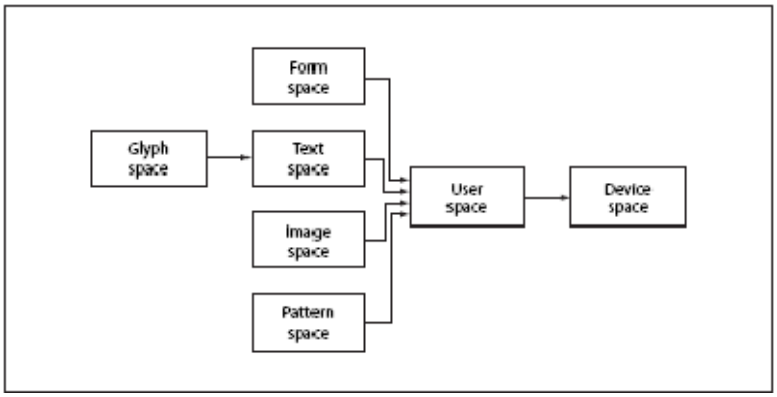


Fig. 3 Relaţiile dintre sistemele de coordonate (imagine preluată din ISO 32000-1)

Pe lângă aceste două sisteme de coordonate PDF mai foloseşte şi alte spaţii de coordonate [4]:

- Coordonatele pentru text va fi specificat în spaţiul text. Transformarea din spaţiu text în spaţiu utilizator va fi definit de o matrice în combiantie cu alţi parametrii specifici textului respectiv.
- Simbolul unui caracter dintr-un font este definit în spaţiul simbol (eng. glyph space). Transformarea din spaţiul simbol în spaţiul text va fi realizată de matricea de font.
- Toate imaginile vor fi definite în spaţiul imagine. Transformarea din spaţiul imagine în spaţiul utilizator este predefinit şi nu poate fi modificat.

Biblioteca MuPDF

MuPDF este o bibliotecă software gratuită şi open source dezvoltată în C care implementează un motor de parsare şi randare de PDF-uri şi XPS-uri. Acesta este utilizat în principal pentru a face pagini în bitmapuri, dar oferă de asemenea suport pentru alte operaţiuni, cum ar fi căutarea, listarea cuprins şi hyperlinkuri. Motorul de randare în MuPDF este adaptat pentru grafică de înaltă calitate. Aceasta randează textul cu o precizie de fracţiuni de pixel pentru o calitate cât mai bună în reproducerea aspectului unei pagini imprimate pe ecran.

Motivul pentru care a fost ales pentru acest proiect muPDF îl reprezintă caracteristicele de baza ale acestuia. MuPDF are o dimensiune redusă de cod, este rapid şi, cu toate acestea, complet. Aceasta susţine PDF 1.7, cu transparenţă, criptare, hyperlinkuri, adnotări, căutarea în text şi mai multe. Suportă, de asemenea, şi documente OpenXPS. Nu oferă suport pentru caracteristici interactive, cum ar fi

completare de formulare sau JavaScript. Un alt avantaj pentru care am ales MuPDF este acela că este foarte bine modularizat, astfel încât alte caracteristici pot fi adăugate dacă se doreşte acest lucru.

Există un număr de aplicaţii software gratuite care folosesc MuPDF pentru a randa documente PDF, cel mai important fiind Sumatra PDF. MuPDF este, de asemenea, disponibil ca pachet pentru Debian, Fedora, FreeBSD Ports și OpenBSD Ports.

MuPDF foloseşte o structură de date complexă pentru reprezentarea internă a documentului PDF, de care se foloseşte pentru a manipula fişierul pentru diferite funcţionalităţi, precum randare a paginilor, extragere de text, de imagini, căutare în text etc. Practic realizează o copie în memorie a fişierului PDF aflat pe hard. Folosirea acestei structuri care e completată cu date efective din fişierul PDF facilitează realizarea de noi funcţionalităţi, cum a fost şi cazul acestui proiect.

3. Arhitectura sistemului

3.1 Descrierea arhitecturii şi modulele funcţionale ale sistemului

Arhitectura sistemului aplicaţiei Smart Presentation este de tip client-server, clienţii fiind, din punct de vedere logic, de două tipuri: speakerul, cel care ţine prezentarea şi cei din audienţă. Clienţii au în general două posibilităţi: să trimită date către server, precum în cazul feedbackului, sau să interogheze serverul pentru resurse, spre exemplu documentul PDF, slideul curent al prezentării sau forma globală, agregată a feedbackului curent (Figura 4).

Serverul aplicaţiei are rolul de a stoca resursele comune tuturor. Aceste resurse pot fi statice, cum este cazul documentului PDF, sau pot fi dinamice, cum e cazul elementelor de feedback sau a slide-ului curent al speakerului. În ambele cazuri, serverul are rolul de a centraliza datele şi de a le pune la dispoziţia clienţilor, prin adrese URL.

În cazul resurselor dinamince, serverul poate avea doar rolul de stocare, astfel încât orice client să poată accesa resursa, sau poate avea şi un rol de prelucrare, cum ar fi în cazul elementelor de feedback. Aceste resurse sunt refăcute la primirea oricărui feedback, astfel încât răspunsul la o cerere să conţină o formă agregată a feedbackului.

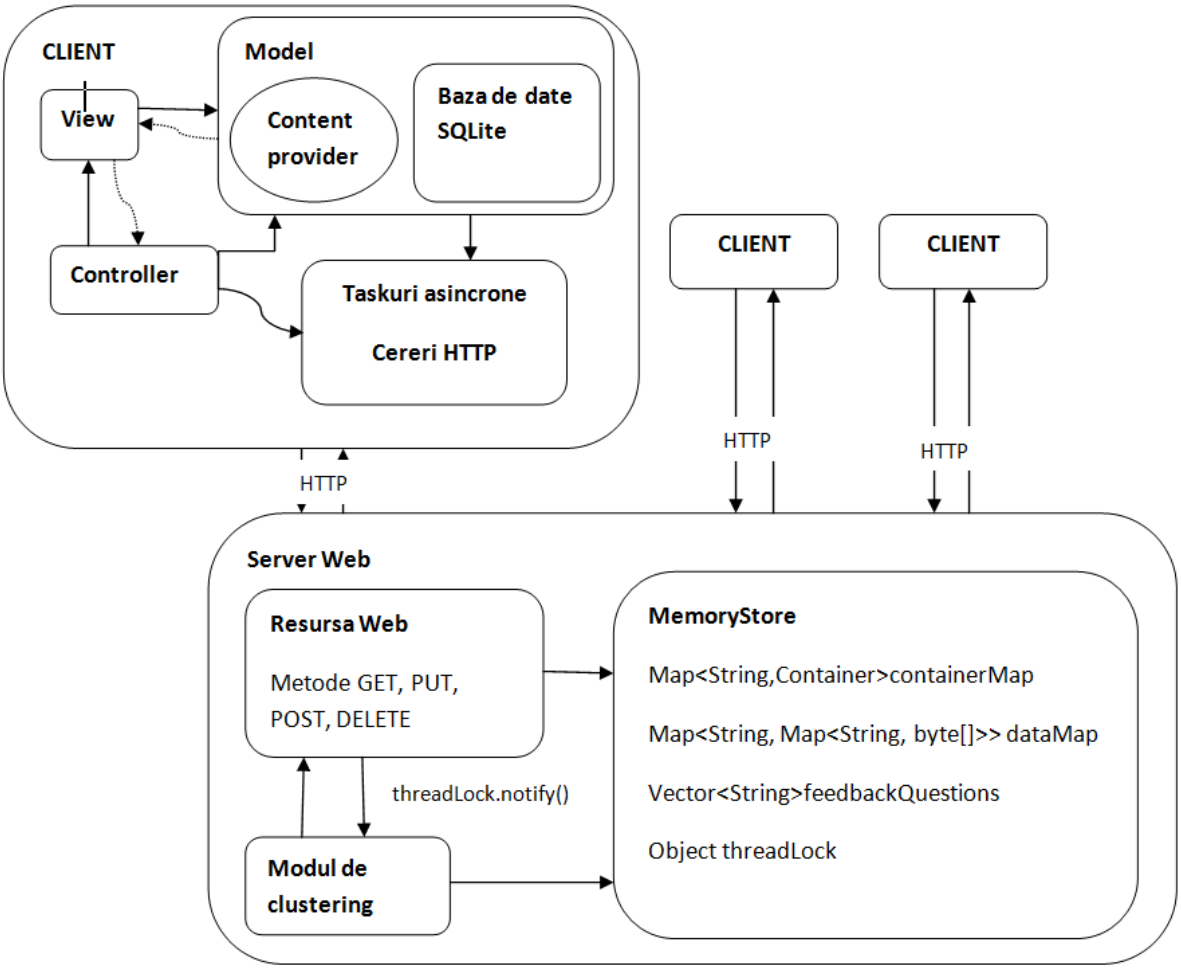


Fig. 4 Arhitectura globală a sistemului

3.2 Arhitectura clientului Android

Clientul Android implementează o arhitectură Model-View-Controller, care se caracterizează prin separarea clară a funcţionalităţii celor trei module funcţionale:

- View – interfaţa grafică, alcătuită din totalitatea butoanelor, ferestrelor şi elementelor cu care utilizatorul interacţionează direct;
- Controller – totalitatea handlerelor care sunt declanşate de utilizarea elementelor interfeţei grafice. În cadrul acestor handlere, sunt formulate interogări asincrone către model sau către server pentru obţinerea datelor.
- Model – modulul care are rolul de persistenta a datelor şi care poate face cereri pe rețea.

Interfaţa este actualizată prin returnarea răspunsurilor de la server, asincron. Aceste răspunsuri pot veni prin intermediul Modelului, dacă cererea este făcută intermediar la Model, în cazul aplicaţiei Android acesta fiind implementat prin clasa `ContentProvider`. O altă variantă a actualizării vizuale este prin întoarcerea valorii direct prin execuţia asincronă a unei cereri către server, caz în care controllerul este responsabil cu efectuarea cererilor către server.

O altă componenta de bază a modelului este baza de date, care este folosită pentru stocarea informaţiilor de feedback, sau a informaţiilor proprii clientului, cum ar fi date legate de propriul feedback. Aceste date se obţin prin interogarea content providerului, care poate decide dacă este nevoie de o actualizare a datelor prin cereri către server.

Mai jos se află o diagramă UML care descrie principalele componente ale Modelului, clasa content providerului şi cele care deservesc bazei de date comunicării cu serverul (Figura 5):

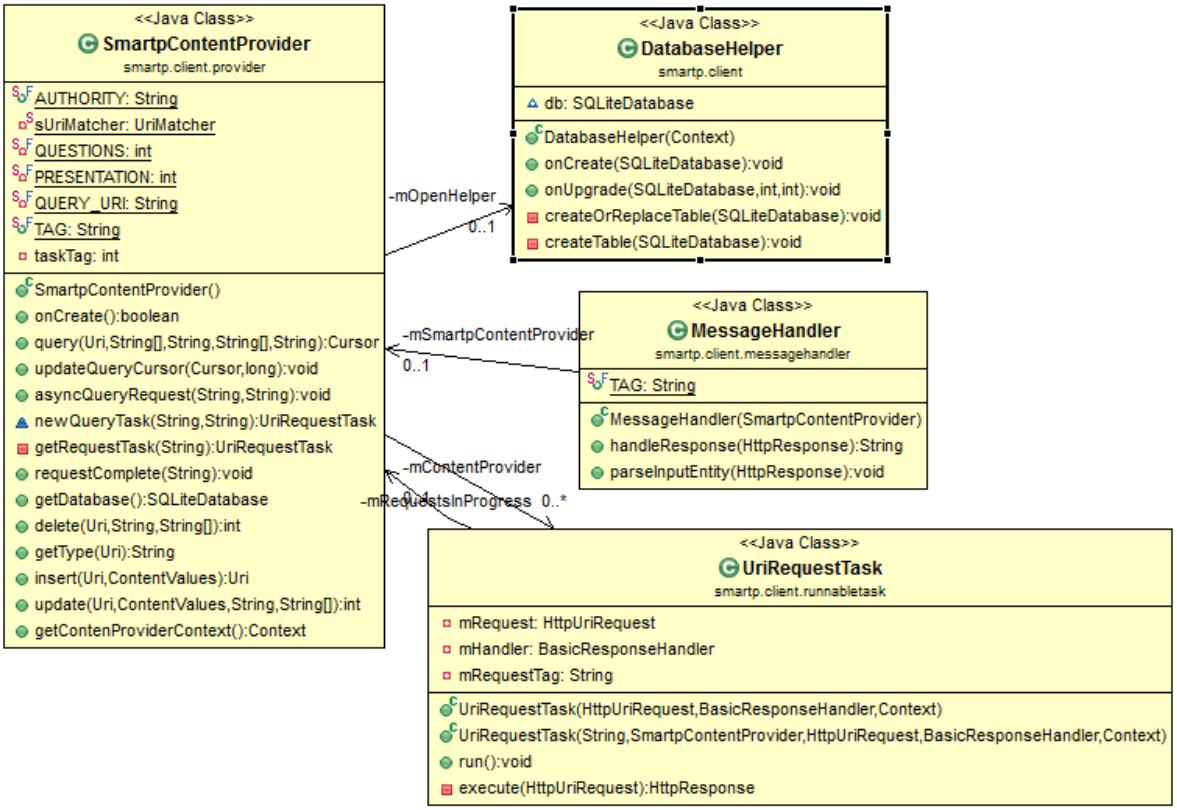


Fig. 5 Diagrama UML a arhitecturii MVC implementată prin ContentProvider

3.3 Arhitectura serverului web

Serverul central are două componente, care rulează pe threaduri separate:

- un server web, care răspunde la cereri HTTP pentru resurse;
- un modul de grupare (eng. *clustering*) pe baza similarităţilor semantice a întrebărilor de feedback. Acest modul comunică doar cu cealaltă componenta a serverului.

În cazul serverului web, există un thread principal cu un entry point din care se rulează serverul. În acest thread se iniţializează pachetele claselor resursă, care sunt încărcate în background, comunicaţia facându-se prin crearea de threaduri secundare. Responsabil cu crearea şi managementul acestor threaduri este containerul Grizzly.

Pentru stocarea datelor, am implementat o arhitectură de stocare ierarhică, formată din containere pentru diferitele tipuri de date (Figura 6).

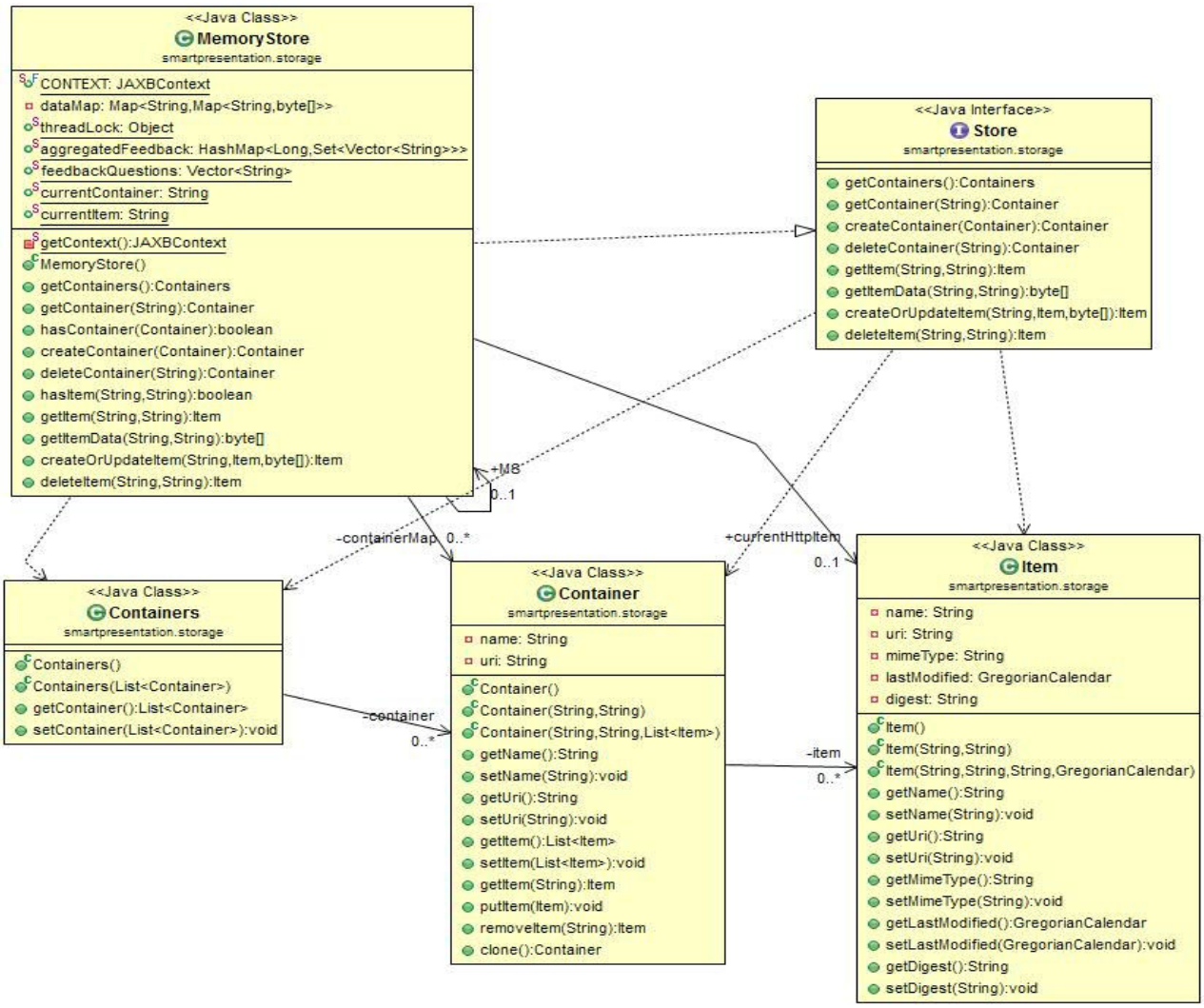


Fig. 6 Diagrama UML a arhitecturii de stocare pe server

Resursele HTTP care sunt disponibile la interogarea serverului web sunt implementate în clase specifice, care definesc metode programatice de interpretare a pachetelor HTTP şi de alcătuire a răspunsurilor în cazul unor cereri (Figura 7).

În cazul resurselor dinamice, cum ar fi feedbackul de la utilizatori, serverul poate modifica resursa prin executarea unor operaţii interne, cum ar fi incrementarea numărului de feedbackuri pozitive asupra unei selecţii (mapate la un URL) sau reclusterizarea întrebărilor pentru o selecţie. În cazul al doilea, serverul procesează clusterizarea semantică într-un thread separat. Sincronizarea cu acesta se face printr-un lock şi prin resurse de memorie comune, aflate în MemoryStore.

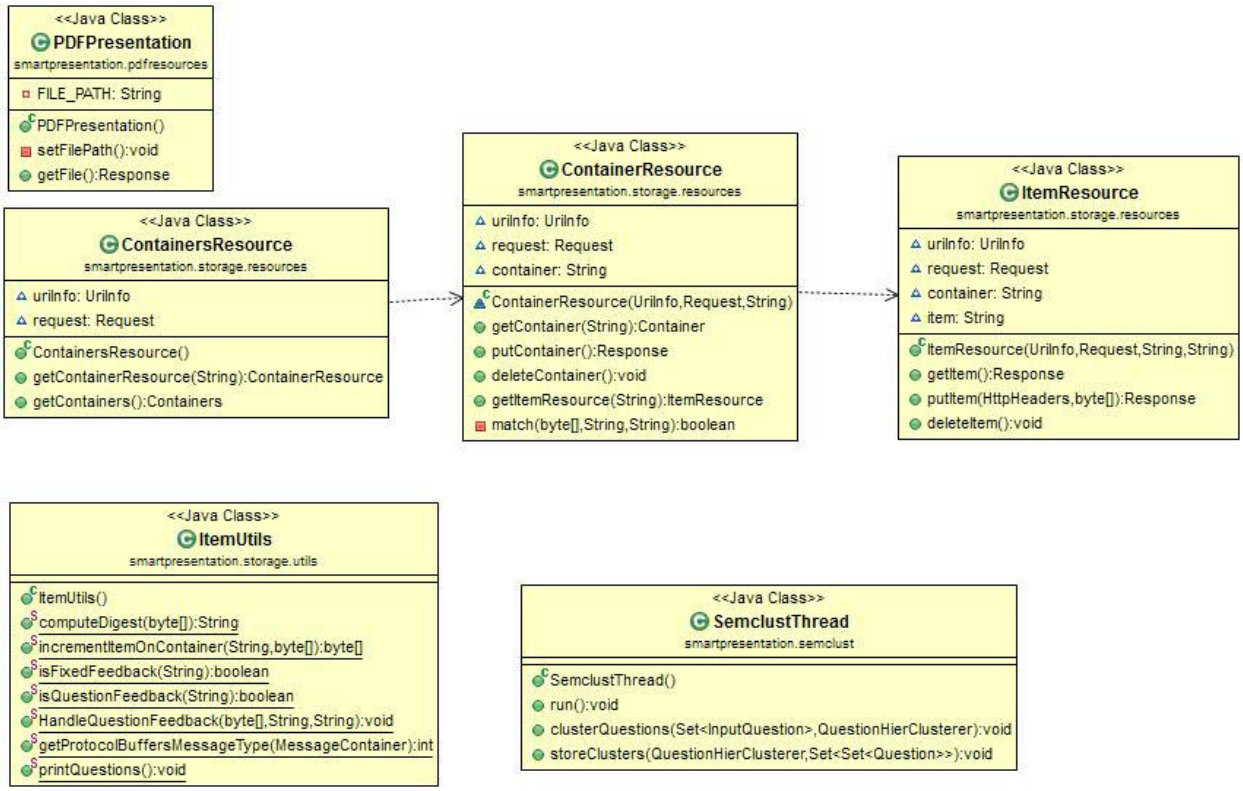


Fig. 7 Diagrama UML a claselor resursa UML şi a metodelor de prelucrare

4. Detalii de implementare

4.1 Accesarea resurselor pe baza URL şi a protocolului HTTP

Aplicaţia SmartPresentation se bazează pe o arhitectură client-server, având în centru un server cu următoarele roluri:

- stochează prezentarea pe care clienţii cu dispozitive Android o descărcă la deschiderea aplicaţiei;
- menţine evidenţa slideului curent al prezentării la care se află speakerul, oferind posibilitatea clienţilor de a-şi sincroniza propria copie a prezentării cu cea a speakerului, prin folosirea modului "Go live!";
- centralizează feedbackului primit de la clienţi, stocând toate tipurile de feedback;
- răspunde la cereri de la clienţi pentru feedbackul agregat;
- rulează modulul de grupare pe baza semantică a întrebărilor puse de cei din audienţă şi serializează structurile de date procesate de acest modul.

Pentru implementarea serverului am ales o soluţie de web service, datorită existenţei unor soluţii multiple de frameworkuri astfel de servicii şi datorită simplităţii accesării resurselor, prin Uniform Resource Locator.

4.1.1 Structura URLului

Containerul Grizzly folosit pentru stocarea resurselor http oferă posibilitatea setării la rulare a unui URL de bază, la care serverul web poate fi accesat. În cazul de fata, URL-ul de baza al serverului este format dintr-un IP şi un port. În cazul staţiei pe care rulează serverul, acest URL are forma http://localhost:9998/, numele “localhost” fiind translatat local in ip-ul 127.0.0.1. În cazul clienţilor, aceştia vor avea posibilitatea accesării serverului prin ip-ul public al acestuia, care se setează indepenent de server la rularea acestuia, în funcţie de setările ruterului wireless care asigură comunicaţia.

API-ul Jersey permite crearea unor clase resursă http, pentru care se defineşte prin adnotarea `@Path` porţiunea de URL care se adăugă URL-ului de baza al serverului. O adnotare de tipul `@Path("/myresource")` aplicată în cazul unei clase va face disponibilă resursa http la adresa `http://localhost:9998/myresource`, în cazul accesului de pe maşina locală care rulează serverul, spre exemplu dintr-un browser rulat local.

Pentru aplicaţia Smart Presentation, am definit mai mult tipuri de URL-uri, în funcţie de resursele accesate. Acestea sunt prezentate în continuare prin porţiunea caracteristică resursei, care se adăugă adresei de bază:

- `/presentation` : această adresa este accesată la pornirea aplicaţiei Smart Presentation pentru descărcarea prezentării PDF stocate pe server;
- `/containers/slidecontainer/slide` – adresa la care este reţinut slideul curent la care se află prezentatorul. Prin accesul la această resursă, utilizatorii pot sincroniza prezentarea cu cea a speakerului. Speakerul actualizează această resursa automat la schimbarea slideului;
- `/containers/positivefeedback/[selection_id]` – aceasta este resursa la care se reţine numărul de feedbackuri pozitive care au fost acordate pe o selecţie din prezentare, reprezentată de un id, număr întreg, al selecţiei;

- `/containers/ambiguousfeedback/[selection_id]` – aceasta este resursa la care se reţine numărul de feedbackuri de tip ambiguu care au fost acordate pe o selecţie din prezentare;
- `/containers/prooffeedback/[selection_id]` – aceasta este resursa la care se reţine numărul de feedbackuri de tip proof/citation needed care au fost acordate pe o selecţie din prezentare;
- `/containers/questions/[selection_id]` – aceasta este resursa la care se găseşte forma agregată binar a întrebărilor puse pe selecţie, grupate în clustere (liste) de întrebări similare din punct de vede semantic;

Pe lângă cele de mai sus, am mai definit patru resurse care sunt create şi actualizate de server, la primirea oricărui tip de feedback. Acestea sunt cele de feedback agregat pentru toată prezentarea, necesare speakerului la finalul prezentării, şi au următoarele adrese URL:

- `/containers/positivefeedback/aggregate` – aceasta este resursa la care se reţine o listă de mapari selecţie – feedback pozitiv;
- `/containers/ambiguousfeedback/aggregate` – aceasta este resursa la care se reţine o listă de mapari selecţie – feedback de tip ambiguous;
- `/containers/prooffeedback/aggregate` – aceasta este resursa la care se reţine o listă de mapari selecţie – feedback de tip proof needed;
- `/containers/questions/aggregate` – aceasta este resursa care stochează toate întrebările puse pe documentul PDF, prin gruparea acestora în selecţii, fiecare selecţie fiind un binar de tipul celei găsită la o resursă /containers/prooffeedback/[selection_id].

4.1.2 Tipurile mesajelor HTTP

Pentru transmiterea diferitelor mesaje HTTP între client şi server, am folosit setarea antetului Content-Type al mesajului http. Următoarele tipuri au fost folosite pentru definirea conţinutului:

- `text/plain`: prin această setare am definit conţinutul de tip clear text, în cazul mesajelor scurte, care nu necesitau serializare pentru o optimizare a comunicării. Astfel de mesaje sunt:
  - schimbarea slideului de către speaker respectiv accesarea slideului curent de către client, în ambele cazuri se transmite doar un număr al slideului;
  - transmiterea unui şir fix de două caractere, "+1" sau "-1", cu semnificaţia adăugării său retragerii unui feedback te ţip fix, adică unul din tipurile positive feedback, ambiguous sau proof needed în cazul unei selecţii. Selecţia este reprezentată în URL prin id-ul ei, un număr întreg de tip long care se obţine ca hashCode al unui șir de caractere reprezentativ pentru elementele selectate;
  - transmiterea ca răspuns de la server către client a unui şir de caractere reprezentând numărul total de feedbackuri din fiecare tip din cele trei descrise mai sus, pentru o anumită selecţie. Acest număr este reprezentativ pentru agregarea tipurilor fixe de feedback, frecvenţa indicând relevanţa acelui feedback;
- `application/pdf`: această setare e folosită pentru codificarea binară a prezentării pdf stocate pe server. Prin interpretarea corectă acestui mesaj http, clientul preia conţinutul binar şi îl scrie într-un fişier de tip pdf, stocat local pe dispozitivul Android.
- `application/x-protobuf`: acesta este un mime-type definit local corespunzător mesajelor http codificate binar folosind Google Protocol Buffers. Setarea conţinutului binar al fişierului se poate face în mai multe moduri: în cazul clientului Android am folosit clasa ByteArrayEntity, care se instantiaza cu un content-type dintr-un vector de octeţi, byte[]. În cazul serverului, sunt definite clase care implementează interfeţele MessageBodyWriter şi MessageBodyReader din pachetul

jersey javax.ws.rs.ext, necesare pentru construirea unei clase Response pentru un mime type definit de utilizator. Pentru deserializare am folosit şi metoda parseFrom() aplicată unui obiect Google Protocol Buffer, prin care se construieşte instanţa din vectorul de octeţi al conţinutului binar.

Setarea câmpului content-type şi citirea acesuia se poate face foarte uşor programatic, în funcţie de pachetele şi clasele Java folosite pentru formarea şi interpretarea mesajelor http. Astfel, pentru citirea câmpului, pe server se poate inspecta câmpul headers de tip HttpHeaders accesibil într-o metodă de tip Http a unei clase resursă, prin apelarea metodei header.getMediaType(). În cazul clientului, pentru citirea antetului se apelează metoda getFirstHeader("Content-Type") pe o instanţă de tip HttpResponse. Pentru setarea acestui câmp, se poate instanția o entitate care implementează interfaţa HttpEntity cu tipul conţinutului corect sau se poate seta acel câmp din antet, printr-o metodă setter.

4.2 Implementarea protocolului de comunicaţie

Un protocol de comunicaţie se defineşte ca un set de reguli cunoscute de ambele părţi ale unei comunicaţii, prin care atât receptorul, cât şi emiţătorul pot interpreta corect mesajele transmise pentru îndeplinirea anumitor cerinţe. Un protocol defineşte de asemenea şi ordinea mesajelor schimbate, anumite operaţii necesitând schimbul unui număr consecutiv de mesaje specifice, într-o ordine particulară. Un protocol bazat pe HTTP, aşa cum este cazul la aplicaţia Smart presentation, este format din două componente:

1. URL – atât clienţii care accesează resursa cât şi serverul pe care este stocată aceasta au aceeaşi concepţie asupra resursei aflate la acea adresa. Astfel, clientul ştie ce adresa trebuie accesată pentru obţinerea resursei dorite şi are certitudinea corectitudinii mesajului primit ca răspuns. Această certitudine permite interpretarea corectă a răspunsului. La fel, serverul asigură stocarea resursei cerute la respectiva adresă, având acelaşi set de mapări resursă-URL ca cel cunoscut de client.
2. Conţinutul mesajelor – în cadrul unei resurse aflate la un URL, se încapsulează un conţinut propriu-zis al mesajului care conţine informaţia dorită. Această informaţie poate reprezenta date obţinute prin prelucrare pe server sau pur şi simplu stocate pe acesta. În funcţie de conţinutul acestor mesaje, un client poate decide transmiterea unor noi cereri sau declanşarea unor evenimente locale.

4.2.1 Componenta mesajelor şi logica protocolului

În cadrul aplicaţiei Smart Presentation, protocolul de comunicaţie este definit pe fiecare arie de funcţionalitate separată. Astfel, serverul primeşte anumite cereri la care este pregătit să răspundă cu mesajele necesare. În continuare, mă voi referi la adresa serverului cu [adresa_server], datorită caracterului variabil al acesteia, în funcţie de adresa ip la care poate fi accesat. De asemenea, mă voi referi la un URL cu termenul "adresa".

Mesajele schimbate între client şi server sunt următoarele:

- La iniţializarea aplicaţiei, fiecare client trimite o cerere http de tip GET la adresa `[adresa_server]/presentation`, cu conţinut gol. Serverul trimite ca răspuns un mesaj http cu headerul Content-type setat "application/pdf" şi conținutul binar, în format pdf, reprezentând aplicaţia stocată pe server.
- La schimbarea slide-ului de către speaker, se trimite un mesaj de tip PUT către server, la adresa [adresa_server]/containers/slidecontainer/slide, având un conţinut cu Content-type "text/plain" care este format dintr-un număr cu slide-ul curent. Clientul trimite cereri de tip GET la aceeaşi

adresă, primind ca răspuns tot un mesaj http "text/plain" cu numărul slide-ului actual. Acest mesaj este trimis la un interval mic de timp, încontinuu, atunci când clientul se află în modul "Go live!".

- Pentru trimiterea unui feedback de tip fix, positive feedback, ambiguous sau proof needed, clientul trimite un mesaj HTTP PUT cu conţinutul "text/plain" reprezentat dintr-un șir de caractere cu forma "+1" sau "-1", cu semnificaţia adăugării sau retragerii acelui tip de feedback. Adresele de acces a acestor tipuri de feedback pentru o selecție sunt:
    - `[adresa_server]/containers/positivefeedback/[id_selecţie]` - pentru accesarea feedbackului de tip positive feedback
    - `[adresa_server]/containers/ambiguousfeedback/[id_selecţie]` - pentru accesarea feedbackului de tip ambiguu
    - `[adresa_server]/containers/prooffeedback/[id_selecţie]` - pentru accesarea feedbackului de tip proof needed.

  La formularea unei cereri PUT la o astfel de resursă, serverul returnează acelaşi răspuns ca în cazul unei cereri GET, un mesaj de tip "text/plain" reprezentând numărul total al feedbackurilor de acel tip primite la acea adresa, adică pe o anumită selecţie. Id-ul selecţie este un şir de cifre care formează un număr întreg reprezentativ doar pentru o selecţie.

- Pentru mesajele ce conţin feedback de tip întrebări pentru o anumită selecţie, am folosit codificarea în formatul binar Google Protocol Buffers, datorită simplităţii şi versatilităţii sale în definirea mesajelor, dar şi datorită unei eficientizări evidente a utilizării benzii de transfer. Ca şi în cazul celorlalte tipuri de feedback, adresa la care este ţinut feedbackul agregat pe un id al unei selecţii este de forma [adresa_server]/containers/questions/[id_selecţie]. Diferenţa apare la tipul conţinutului unui mesaj şi la logica răspunsului de la server în cazul transmiterii unei forme agregate a feedbackului stocat.

  Astfel, un client trimite un feedback sub forma unui mesaj de tip HTTP PUT, cu câmpul content-type setat pe "application/x-protobuf". Conţinutul acestui mesaj este un mesaj de tip protocol buffers, având la baza o întrebare adresată pe o selecţie. La recepţia acestei cereri, serverul reface resursa pe care o are reţinută la acea adresă, prin regrupare. Această resursă conţine o formă agregată a feedbackului, reţinută în format binar Google Protocol Buffers. Conţinutul acesui mesaj este format dintr-o listă de clustere, un cluster fiind o listă de întrebări cu sens asemănător din punct de vedere semantic. Acest mesaj este cel primit ca răspuns de un client sau de către speaker la o cerere de tip GET pe adresa corespunzătoare unei selecţii.

  De asemenea, am folosit Google Protocol Buffers pentru agregarea tuturor tipurilor de feedback pentru întregul document PDF, pentru ca acestea să poate fi accesate de speaker la final. Aceste resurse sunt găsite la adresele detaliate la capitolul anterior, de tipul:

  `[adresa_server]/containers/[feedback_container]/aggregate`

  Conţinutul unui mesaj de tip Google Protocol Buffers poate varia, în funcţie de tipul mesajului. Pentru a determina tipul mesajului înainte de deserializare, am folosit un mecanism de încapsulare care va fi descris în continuare.

4.2.2 Serializarea datelor folosind Google Protocol Buffers

Mesajele de codificare a întrebărilor de feedback sunt serializate folosind mecanismul Google protocol Buffers, detaliat la capitolul Tehnologii folosite. În acest scop, am definit mai multe tipuri de mesaje într-o ordine ierarhică, în funcţie de gradul de încapsulare.

Astfel, pentru o simplă întrebare, trimisă în cadrul unui mesaj sub forma unei cereri PUT de la client către server la o adresă specifică unei selecţii, se foloseşte următorul format de mesaj:

```
message FeedbackQuestion
{
   required int32 retract = 1;
   required string selectionString = 2;
   required string question = 3;
}
```

Pentru un cluster de întrebări, reprezentat printr-o listă de întrebări apropiate semantic, din care se poate alege oricare ca centroid (întrebare reprezentativă), am definit următorul mesaj:

```
message QuestionCluster
{
   required string selectionString = 1;
   repeated FeedbackQuestion questions = 2;
}
```

Pentru reprezentarea agregată pentru o selecţie a tuturor clusterelor obţinute în urma aplicării algoritmului de grupare pentru toate întrebările adresate, am definit următorul mesaj:

```
message SelectionClusters
{
   required int64 selectionId = 1;
   required string selectionString = 2;
   repeated QuestionCluster clusters = 3;
}
```

În plus faţă de aceste mesaje, am implementat mesaje de încapsulare pentru tipurile de feedback fix sau întrebări, pentru încapsularea mesajelor de agregare transmise speakerului la finalizarea prezentării.

```
message AggregateQuestionsFeedback
{
   repeated SelectionClusters SelectionClusters = 1;
}

message FixedFeedback
{
   required string selectionString = 1;
   required string feedbackType = 2;
   required int32 numberOf = 3;
}

message AggregateFixedFeedback
{
   repeated FixedFeedback feedbacks = 1;
}
```

În cazul transmiterii unui mesaj Google Protocol Buffers, ar trebui cunoscut tipul mesajului înainte de deserializare, pentru folosirea claselor corecte generate de executabilul protoc. O tehnică potrivită

pentru acest scop este încapsularea unui mesaj din cele trei tipuri de mai sus într-un mesaj container, care să conţină un câmp cu semnificaţia unui flag care indică tipul mesajului încapsulat, şi un alt câmp cu mesajul propriu-zis. Formatul mesajului container arăta astfel:

```
message MessageContainer
{
    enum Type
    {
           SINGLE = 1;
           CLUSTER = 2;
           SELECTION = 3;
           AGGREGATE_QUESTIONS = 4;
           AGGREGATE_FIXED = 5;
    }

    required Type type = 1;

    optional FeedbackQuestion feedbackQuestion = 2;
    optional QuestionCluster questionCluster = 3;
    optional SelectionClusters selectionClusters = 4;
    optional AggregateQuestionsFeedback aggregateQuestions = 5;
    optional AggregateFixedFeedback aggregateFixed = 6;
}
```

Astfel, câmpul type evidenţiază care din cele trei mesaje opţionale este cel conţinut de mesajul container, permiţând o deserializare corectă.

Pentru definirea acestor mesaje, am folosit următoarele cuvinte cheie ale limbajului Google Protocol Buffers, cu explicaţiile:

- Required – indică necesitatea prezenţei acelui câmp în mesaj;
- Opţional – indică lipsa necesităţii prezenţei acelui câmp în mesaj;
- Repeated – indică un câmp de tip lista al tipului specificat.

De asemenea, se poate observa şi un tag care este asociat fiecărui câmp. Acest tag este utilizat la interpretarea mesajului la deserializare, indicând ordinea în care se găsesc membrii mesajului în formula binară a mesajului.

Generarea claselor, serializarea şi deserializarea mesajelor

Google Protocol Buffers este un limbaj compatibil cu majoritatea limbajelor populare, oferind suport inclusiv pentru Java. Pentru obţinerea mesajelor binare, este necesară obţinerea unor clase Java din formatele definite în fişierele .proto. Generarea acestor clase face printr-un executabil protoc.exe. În cazul utilizării mediului de dezvoltare Eclipse, există un plugin care determină generarea automată a claselor la crearea unui mesaj .proto în folderul resources al unui proiect.

Clasele generate pun la dispoziţie mai multe subclase şi metode pentru serializare şi deserializare. Pentru deserializare, cea mai facilă tehnică este apelarea metodei parseFrom() care primeşte că parametru un array de octeţi (byte[]), spre exemplu conţinutul unui mesaj HTTP. Pentru serializare, fiecare clasa conţine o subclasa builder, care se obţine astfel, pentru un exemplu din mesajele de mai sus:

```
SelectionClusters.Builder clusterBuilder = SelectionClusters.newBuilder()
```

Această instanţă este folosită apoi pentru adăugarea celorlalte câmpuri, prin metode de tip setter;

Exp: clusterBuilder.setSelectionId(id)

Pentru un câmp de tip repeated, se foloseşte metoda add().

După setarea tututor câmpurilor, mesajul se obţine prin apelul metodei build asupra instanţei builder.

Exp: `SelectionClusters clusters = clusterBuilder.build()`

```
SelectionClusters.Builder scBuilder = SelectionClusters.newBuilder();
             scBuilder.setSelectionId(new Long(MemoryStore.currentItem));

             for (Set<Question> cluster : clusterTree)
             {
                    QuestionCluster.Builder qCluster = QuestionCluster.newBuilder();
                    for (Question q : cluster)
                    {
                           FeedbackQuestion.Builder fqBuilder =
                                        FeedbackQuestion.newBuilder();
                           fqBuilder.setRetract(0);
                           fqBuilder.setQuestion(q.originalText);
                           FeedbackQuestion fq = fqBuilder.build();
                           qCluster.addQuestions(fq);
                    }
                    QuestionCluster qc = qCluster.build();
                    scBuilder.addClusters(qc);
             }

             SelectionClusters sc = scBuilder.build();
```

Exemplu de cod din serializarea feedbackului

4.2 Implementarea clientului Android

Implementarea aplicaţiei Smart Presentation include aplicaţia de client, cea de Android şi serverul central. Implementarea clientului cuprinde cele trei module funcţionale, interfaţa, modulul de manipulare a prezentării PDF şi partea de comunicare cu serverul.

Mediul de dezvoltare Android este diferit de cel desktop. O diferenţă de baza între aplicaţiile pentru telefoane mobile şi aplicaţiile desktop este reprezentată de modul în care tratează persistenţa datelor. Cel mai des, aplicaţiile desktop folosesc o implementare a patternului MVC centrată pe documente – aplicaţiile deschid un document, îl citesc în memorie şi îl transpun în obiecte care persistă în memorie. Aceste programe definesc vizualizări în care pot fi manipulate acele obiecte ale modelului de date, prin procesarea inputului de la controller. Spre deosebire de acestea, aplicaţiile Android combină modelele de date şi interfaţa utilizatorului într-un mod diferit. Aplicaţiile rulează pe un dispozitiv mobil cu memorie limitată, care poate rămâne fără baterie într-un moment imprevizibil. Întregul concept de document este absent în Android. Utilizatorul întotdeauna ar trebui să aibă datele la îndemână şi să fie încrezător că acestea sunt în siguranţă. Pentru a stoca datele aplicaţiei incremental, obiect cu obiect, şi ca acestea să fie întotdeauna la dispoziţie, Android oferă un suport centrat pe baze de date, prin clasele cuprinse în pachetul android.database.sqlite [10].

O altă diferenţă între aplicaţiile pentru telefoane mobile şi aplicaţiile desktop este accentul care cade în cazul celor mobile în primul rând pe o utilizare fluidă a aplicaţiei, astfel încât interfaţa grafică să nu fie blocată niciodată. Din acest motiv, sdk-ul Android oferă multiple pachete şi clase care permit

rularea paralelă a mai multor taskuri, cu scopul de a nu încărca suplimentar threadul GUI. Spre exemplu, ultimele versiuni de Android sdk interzic rularea de operaţii de networking, cum sunt şi cele http. Încercarea de rulare a unei astfel de cereri în threadul aplicaţiei aruncă o excepţie [2].

Cea mai des utilizată tehnica în devoltarea unor taskuri care rulează paralel este comunicarea asincronă cu threadul respectiv, astfel încât elementele vizuale, de interfață din threadul principal să fie înştiinţate de finalizarea taskului paralel. Pentru implementarea acestei tehnici, am folosit două metode. Prima şi cea mai simplă este de a implementa o clasă care extinde clasa AsyncTask, aceasta oferind metode pentru rularea în background şi pentru finalizarea rulării. Cealaltă este cea de a realiza un managedQuery() către ContentProviderul aplicaţiei, un serviciu care are rolul Modelului din arhitectura Model-View-Controller şi care are acces direct la bază de date a aplicaţiei. Ambele variante sunt detaliate în continuare.

4.3.1 Implementarea design patternului MVC

Aşa cum am descris şi la capitolul de Arhitectură, devoltarea aplicaţiei Android implică implementarea unui pattern Model-View-Controller.

Astfel, se separă cele trei module funcţionale, interfaţa grafică, modelul şi controllerul, astfel încât fiecare componentă este dependentă de celelalte. Interfaţa grafică răspunde la comenzi din partea utlizatorului, controllerul fiind responsabil cu tratarea evenimentelor şi cu actualizarea datelor din interfață. Modelul este cel care asigură persistenţa datelor din care atât controllerul, cât şi view-ul îşi actualizează conţinutul.

Şi în cadrul aplicaţiei Smart Presentation, există o separare clară a componentelor, astfel încât acestea pot fi încadrate într-una din cele trei categorii: view, model şi controller. Astfel, view-ul este constituit din totalitatea elementelor grafice ale aplicaţiei şi controllerul cuprinde totalitatea handlerelor de actualizare a interfeţei. Modelul este alcătuit din taskurile asincrone care comunică direct cu serverul web şi dintr-o componentă specială, Content provider, care asigură persistenţa datelor într-o bază de date Sqlite şi care poate fi interogată.

4.3.2 Persistenţa datelor în Sqlite şi accesarea resurselor interne prin ContentProvider

Instanţele Content providers reprezintă o componentă a aplicaţiei locale şi sunt analoage conceptului de serviciu web RESTful. API-ul content providerului permite aplicaţiilor client să interogheze sistemul de operare pentru date relevante utilizând un URI, similar modului în care un browser cere informaţii pe internet. Adresa URI a content providerului începe întotdeauna cu content://, la care se adăugă că în cazul unui web service componenta specifică resursei.

Dacă o aplicaţie implementează un provider propriu, este necesar ca acesta să fie înregistrat în fişierul de configurare al aplicaţiei, AndroidManifest.xml, astfel:

```
<provider android:name=".provider.SmartpContentProvider"
          android:authorities="smartp.client.provider.SmartpContentProvider" />
```

Utilizarea clasică a unui content provider se face prin stocarea datelor într-o bază de date SqLiteDatabase, asupra cărora acţionează metodele implementate ale clasei abstracte ContentProvider:

- Insert – este analoagă metodei POST a unui server web şi are rolul de a stoca noi date în bază de date

- Query – analoagă metodei GET, returnează înregistrări din bază de date pe baza unor cereri SQL, într-o colecţie specializată numită Cursor
- Update – analoagă metodei update. Înlocuieşte cererile din bază de date cu unele mai noi
- Delete – analoagă cererii de tip DELETE

Ca şi în cazul unui server web, un content provider mapează dinamic adrese URL la resursele aflate în baza de date. Această mapare este responsabilitatea programatorului, care la apelul metodei Query a content providerului face match pe URIul folosit în query pentru a şti ce interogare se realizează pe baza de date. Interogarea întoarce o instanţă a clasei Cursor, care oferă pe lângă informaţia propriu-zisă mecanisme de declanşare a evenimentelor atunci când datele aflate la adresa respectivă suferă modificări. Pentru a activa aceste mecanisme, se apelează metoda [2]:

```
queryCursor.setNotificationUri(getContext().getContentResolver(),uri);
```

la crearea cursorului, adică în metoda Query a content providerului, şi metoda :

```
getContext().getContentResolver().notifyChange(uri, null);
```

la modificarea conţinutului, adică în interiorul unei metode insert, delete sau update.

În cazul ambelor metode, se foloseşte metoda `getContext().getContentResolver()`, care întoarce instanţa unui `ContentResolver`, clasa care realizează interfaţa dintre aplicaţia care rulează şi content providerul înregistrat în fişierul de configurare. Atât Context, cât şi ContentResolver, sunt clase singleton, ale căror instanţe sunt create la iniţializarea aplicaţiei Android.

Clasa Content Provider a aplicaţiei Smart Presentation are scopul de stocare în bază de date a unor date care sunt preluate de pe server. Cele mai importante astfel de date sunt documentul PDF, care este preluat asincron la deschiderea aplicaţiei, după care aceasta este notificată pentru a-şi reface interfaţa, şi întrebările de feedback sub forma agregată, care sunt reţinute asemănător unui mecanism de cacheing în bază de date, pentru a rări cererile la server. În ambele cazuri, din clasa activitate se foloseşte un query de tip managedQuery, care primeşte ca parametru adresa de interogare a content providerului şi returnează un cursor cu datele din bază de date.

Clasa managedQuery are posibilitatea de mapare a unei instanţe a clasei `ContentObserver`, care implementează metoda `onChange()`, aceasta fiind metoda handler apelată atunci când un notify este executat pe URLul cursorului în cadul content providerului, cel mai uzual în cazul unui insert, delete sau update (Figura 8).

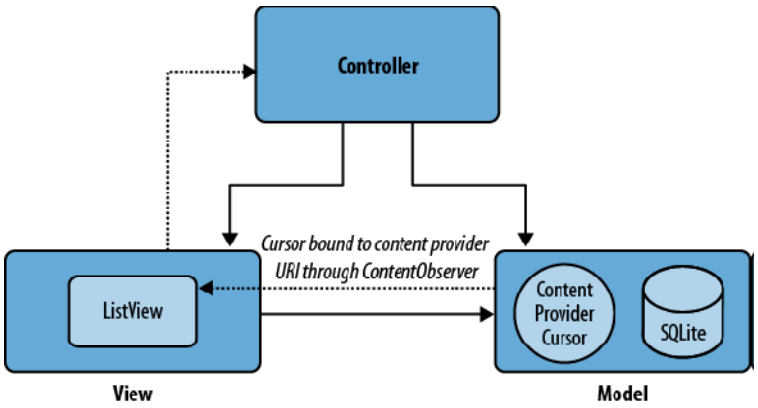


Fig 8. Componentele MVC de bază ale unei aplicaţii Android[1]

[1] Zigurd Mednieks, Laird Dornin & G. Blake Mieke: *Programming in Android 2nd Edition (O'Reilly, 2012), 80*

Metoda query() a content providerului este responsabilă cu returnarea rezultatelor din bază de date. Acest matching se face folosind o clasă ajutătoare UriMatcher.

În cazul întrebărilor feedback reţinute în baza de date, am decis să implementez o modalitate de cacheing, astfel încât în urma unei extrageri a intrării din baza de date corespunzătoare feedbackului unei selecţii, să se verifice şi timestampul ultimei actualizări, astfel încât o nouă cerere către serverul web să se facă doar atunci când resursele stocate sunt considerate prea vechi.

Extragerea căii de salvare a documentului PDF din baza de date se face doar o dată, la început, în urma unei cereri efectuate la serverul web care declanşează un notify la insertul în baza de date.

4.3.3 Accesarea resurselor de pe web server prin mesaje asincrone

Clasa AsyncTask

Accesarea resurselor de pe web server se face în două situaţii, din clasa de baza Activity sau din clasa ContentProvider, dar în ambele cazuri comunicarea cu serverul se face asincron. În cazul în care apelul se face direct din threadul UI, acest mecanism se implementează printr-o clasă care extinde clasa abstractă AsyncTask [2]. Această clasa oferă metode şi parametri prin care comunică cu threadul apelant.

Cele trei tipuri utilizate de un task asincron sunt următoarele:

- `Params` – tipul parametrilor trimişi taskului la execuţie;
- `Progress` – tipul unităţilor de progress publicate în timpul efectuării operaţiilor în background;
- `Result` – tipul rezultatului obţinut în urma rulării operaţiilor de background;

Aceste tipuri sunt setate atunci când se extinde clasa AsynTask, prin setarea celor trei parametri generici ai clasei. În cazul în care unul din cele trei tipuri nu este folosit, acesta poate fi marcat cu void. Un exemplu de declarare a unei clase copil ar fi:

```
private class MyTask extends AsyncTask<URL, Void, String)
```

care se traduce astfel: metoda de execuţie primeşte ca parametri instante ale clasei URL, unităţile de progress nu sunt folosite, iar rezultatul este întors sub forma unui String.

Când un task asincron este executat acesta trece prin patru paşi[1]:

1. `onPreExecute()`, invocat de threadul UI imediat după ce taskul este executat. Acest pas este utilizat în mod normal pentru setarea taskului, spre exemplu pentru afişarea unui progress bar în interfaţa utilizatorului.
2. `doInBackground(Params ...)` invocat de threadul de background imediat ce `onPreExecute()` termină de executat. Acest pas este folosit pentru efectuarea unor acţiuni computaţionale de background care pot dura un timp mai îndelungat. Rezultatul operaţiilor efectuate aici trebuie să fie întors în această metodă şi va fi pasat următorului pas. Acest pas poate declanşa şi metoda `publishProgress(Progress...)` pentru a publica una sau mai multe unităţi de progres. Aceste valori sunt interpretate în UI thread, în pasul `onProgressUpdate(Progres...)`
3. `onProgressUpdate(Progress...)`, invocat de UI thread după un apel al metodei `publishProgress(Progress...)`. Momentul exact al executării este nedefinit. Această metodă este folosită pentru afişarea oricărei forme de progres în interfaţa utilizatorului în timp ce operaţiile

[1] Documentatia Android, 20.06.2012, <developer.android.com/reference/android/os/AsyncTask.html>

de background sunt în execuţie. Spre exemplu, poate fi folosită pentru animarea unui progress bar sau pentru a afişa mesaje de log.

4. `onPostExecute(Result)`, invocat de UI thread după ce operaţiile de background se termină. Parametrul Result este cel preluat de metoda `doInBackground()`.

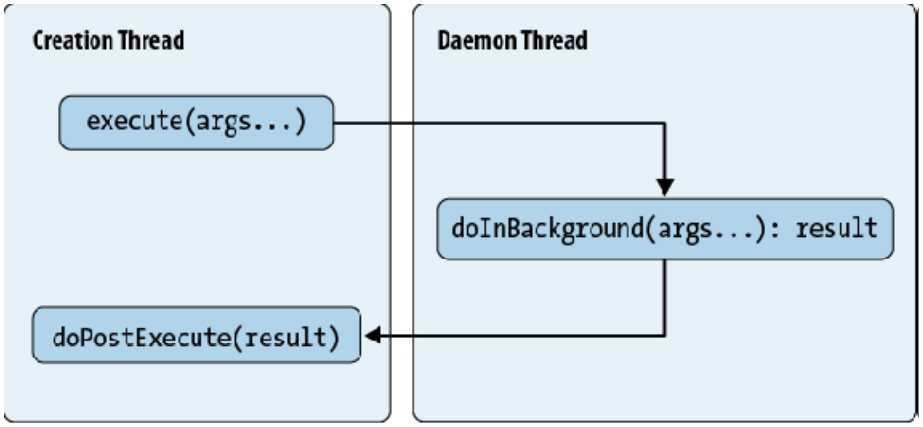


Fig. 9 Fluxul de date AsyncTask[1]

O altă facilitate oferită de clasa `AsynTask` este posibilitatea opririi acesteia din threadul apelant, prin apelul metodei `cancel(boolean)`. Invocarea acestei metode activează returnarea valorii `onCancelled(Object)` la terminarea metodei `doInBackgroud()`, în loc de `onPostExecute`. De asemenea, în timpul executării taskului se poate verifica periodic valoarea întoarsă de `onCancelled()`.

Din punct de vedere al concurenţei threadurilor, AsyncTask asigură că toate apelurile callback sunt sincronizate în aşa fel încât următoarele operaţii sunt sigure:

- setarea membrilor în constructor sau în `onPreExecute()` şi referirea lor în `doInBackground(Params...)`
- setarea câmpurilor membri în `doInBackgroud(Params...)` şi referirea lor în `onProgressUpdate(Progress...)` şi în `onPostExecute(Result)`

Aplicaţia Smart Presentation foloseşte numeroase clase particulare care implementează această clasă abstractă. În general, toate taskurile care presupun comunicare pe rețea, în acest caz efectuarea cererilor către serverul web şi întoarcerea asincronă a răspunsurilor, sunt mapate la o astfel de clasă. Următoarele sunt principalele astfel de clase folosite în cadrul aplicaţiei:

- `GetSlideAsyncTask` – este clasa responsabilă cu sincronizarea slideului curent cu cel al speakerului, a cărei executare este lansată atunci când se intră în modul Go live!. Modul de funcţionare a acesteia este următorul: funcţia `doInBackgroud(String... params)` intră într-un ciclu în care, dacă nu a s-a apelat încă metoda cancel în UI Thread, se verifică ultima actualizare a slideului. Dacă timpul scurs de la ultima actualizare este mai mare de un anumit threshold mic (2-3 secunde), se execută o cerere asincronă către serverul web pentru actualizarea numărului slideului.

  Verificarea dacă taskul continuă să ruleze se face prin apelul metodei isCancelled, care întoarce true sau false. După fiecare cerere GET efectuată, răspunsul este folosit pentru actualizarea

[1] Zigurd Mednieks, Laird Dornin & G. Blake Mieke: *Programming in Android 2nd Edition (O'Reilly, 2012), 110*

interfeţei grafice, în cazul nostru schimbarea slideului. Aceeaşi metodă de actualizare a slideului este apelată şi în final în metoda onPostExecute(String result).

Metoda doInBackground(String... params) primeşte parametru de tip String, reprezentat de URLul la care se face cererea şi returnează un răspuns de tip String, numărul slideului curent.

```
@Override
protected String doInBackground(String... params)
{
	String response = "";
	while (!isCancelled())
	{
		if (TimeUtils.getDiffTimeInSeconds(taskTimer) > 3)
		{
			taskTimer = TimeUtils.getCurrentTime();
			HttpClient client = new DefaultHttpClient();
			HttpGet httpGet = new HttpGet(params[0]);
			try
			{
				HttpResponse execute = client.execute(httpGet);
				response = HttpUtils.getStringFromResponse(execute);

				changeSlide(response);

			} catch (Exception e)
			{
				e.printStackTrace();
			}
		}
	}
	return response;
}
```

Implementare a metodei doInBackground()

- Clasa PutCurrentSlide este folosită în aplicaţie doar de speaker, şi este apelată atunci când acesta schimbă slide-ul. În cadrul metodei de background, se setează conţinutul de tip "text/plain" al unei entităţi StringEntity, reprezentând slideul curent, şi se execută o metodă de tip HttpPut.
- Clasa PutFixedFeedback este folosită pentru trimiterea unuia din cele trei tipuri de feedback fix, positive feedback, ambiguous sau proof needed. Această clasa setează în constructor un membru String al clasei, numit PutOrRetract, cu valoarea "-1" sau "+1", cu semnificaţia adăugării acelui feedback sau retragerea unui feedback făcut anterior. Acest şir de două caractere este încapsulat în conţinutul unei StringEntity şi trimis printr-un http put către server, care returnează ca răspuns numărul agregat al acelui tip de feedback pus pe selecţia respectivă. Afişarea răspunsului în aplicaţie se face la execuţia onPostExecute()
- Clasa PutFeedbackQuestionTask realizează serializarea unei întrebări de tip String şi trimiterea acesteia către server. Serializarea se face în formatul Google Protocol Buffers şi scrierea mesajului http se face folosind o entitate generică, ByteArrayEntity, care spre deosebire de StringEntity, foloseşte conţinut de tip binar:

```
ByteArrayEntity entity = new ByteArrayEntity(messsage.toByteArray());

entity.setContentType(smartp.client.AlternateMediaType.APPLICATION_XPROTOBUF);
```

Clasa UriRequestTask

Comunicarea cu serverul web se face nu doar din Activity, ci şi prin intermediul Content Provider. În acest scop, am dezvoltat o clasă care implementează interfaţa Runnable, rulând într-un thread separat. La fel că în cazul `AsyncTask`, în cadrul metodei `run()` se execută operaţiile de background. Pentru interpretarea mesajului, am implementat o clasă handler, `MessageHandler`, care parseaza mesajul http pentru a decide următorul pas. În general, datele cuprinse în răspunsul de la server sunt introduse în baza de date SQLite, astfel încât la o modificare a datelor, cursorul pe care s-a făcut o interogare pe bază de date să notifice elementele apelante din Activity.

Prima situaţie în care se întâmplă acest scenariu este la pornirea aplicaţiei, când se execută un managedQuery la contentprovider pentru descărcarea prezentării PDF de la server.

```
public void query()
      {
            Uri queryUri = SmartPresentationMessage.Pdf.CONTENT_URÎ;
            Cursor myCursor = managedQuery(queryUri, null, null, null, null);
            myCursor.registerContentObserver(new ContentObserver(new Handler()) {

                @Override
                public void onChange(boolean selfChange)
                {
                         Log.d(SmartpTAG.TAG, "On load finished");
                         super.onChange(selfChange);
                         onLoadPresentation();
                         initializeFrontend();
                }

                @Override
                public boolean deliverSelfNotifications()
                {
                    return true;
                }
            });
       }
```

Metoda de interogare a content providerului

`SmartPresentationMessage.Pdf.CONTENT_URI` este adresa la care se face o interogare către ContentProvider, şi are forma content://" + AUTHORITY + "/" + Pdf.PATH, unde AUTHORITY este calea completă a clasei Content Provider şi `Pdf.PATH` este o constanta care defineşte numele resursei. Această interogare declanşează cererea către web server. La descărcarea prezentării PDF, aceasta este scrisă în memoria internă a dispozitivului şi adresa internă, cea specifică sistemul de fişiere Android este salvată în baza de date. În funcţia care realizează insert în bază de date, se apelează metoda de notificare notifyChange pe URLul descris mai sus, astfel încât cursorul creat de metoda managedQuery

să fie atenţionat de terminarea cererii. În urma notificării, se execută metoda onChange, în care se încarcă prezentarea PDF în interfaţa grafică.

Aceeaşi succesiune de acţiuni ca şi în cazul de mai sus se efectuează în cazul interogării ContentProviderului pentru obţinerea feedbackului agregat. Aceste date sunt cerute prin intermediul instanţei ContentProvider deoarece am dorit salvarea în bază de date a feedbackului, astfel încât să fie implementat un mecansim de cacheing care să permită aplicaţiei să obţină feedbackul direct din baza de date dacă ultimul moment al actualizării sale a fost foarte recent. Acest feedback agregat este reţinut în bază de date SqlLite în tabela QUESTIONS_TABLE, în cazul întrebărilor, şi în FIXED_FEEDBACK, pentru celelalte tipuri de feedback. Interogarile pe tabele se fac după ID reprezentând selecţia, şi care conţine un blob cu binarul serializat Google Protocol Buffers şi un timestamp al ultimei actualizări.

4.4 Implementarea serverului web

Serverul central al aplicaţiei Smart Presentation este un server web RESTful, care returnează resurse pe baza unor cereri efectuate pentru anumite adrese URL. Tehnologiile utilizate la dezvoltarea acestuia sunt Grizzly, pentru container, şi Jersey, pentru implementarea claselor resursa şi a metodelor specifice HTTP. Pe lângă componenta de resurse, serverul mai are şi scopul de grupare a întrebărilor feedback de la clienţi, pentru a permite utilizatorilor o vizualizare agregată, mai compactă, a întrebărilor, în funcţie de similaritatea semantică. Astfel, partea mea de implementare a cuprins doi paşi:

1. dezvoltarea serverului web şi a claselor resursă şi maparea acestora la adrese URL

2. integrarea modulului de clusterizare şi sincronizarea acestuia cu resursa HTTP corespunzătoare feedbackului format din întrebări

4.4.1 Iniţializarea serverului şi a containerului Grizzly

Containerul Jersey oferă o interfaţă programatică de iniţializare, spre deosebire de procedura standard de înregistrare a resurselor web specifice serverelor web Tomcat sau Glassfish, adică prin introducerea claselor într-un fişier de configurare web.xml.

Astfel, serverul poate fi pornit dintr-o metodă Main, permiţând şi operaţii anexe, cum ar fi în cazul aplicaţiei Smart Presentation pornirea unui thread secundar al modulului de clusterizare a întrebărilor feedback.

Iniţializarea serverului web presupune următorii paşi [5]:

- Setarea unei adrese de bază a serverului web, spre exemplu http://localhost:9988/. Orice resursa stocată pe server are adresa formată din concatenarea acestei adrese unei porţiuni specifice resursei;
- Înregistrarea pachetelor unde se află clasele resursa;
- Crearea propriu-zisă a serverului web, prin apelul metodei:

```
GrizzlyWebContainerFactory.create(BASE_URI, initParams).
```

4.4.2 Clasele resursă şi metodele HTTP definite pe server

Serverul web pune la dispoziţie toate resursele de care clientul are nevoie pentru îndeplinirea funcţionalităţilor aplicaţiei Smart Presentation. Aceste resurse pot fi acccesate şi modificate prin cereri HTTP de tip GET, PUT, POST sau DELETE. Clasele se mapează la adrese URL prin adnotări specfice frameworkului Jersey, care defineşte de asemenea şi adnotări ale metodelor clasei pentru metodele HTTP la care acestea se mapeaza sau adnotări referitoare la tipul de content al corpului http.

Resursele expuse de serverul web sunt grupate în pachete care sunt înregistrate la iniţializarea containerului Grizzly, pentru ca ele să poate fi accesate de class loader. Resursele specifice aplicaţiei Smart Presentation sunt:

- Class `PDFPresentation` – este clasa care pune la dispoziţie prezentarea PDF stocată pe server. Aceasta este adnotată cu @Path("presentation"), indicând adresa la care poate fi accesată. Clasa implementează doar o metodă GET, adnotată cu @Produces("application/pdf"), indicând tipul conţinutului pachetului http. În interiorul metodei, este citit într-un obiect File fişierul PDF aflat în directorul rădăcina al serverului . Acest obiect este încapsulat în răspunsul http prin metoda `Response.ok((Object) file).`
- Clasa `ContainersResource` – adonatată cu `@Path("containers")`, este clasa care încapsulează totalitatea containerelor, întorcând un xml cu containerele definite.
- Clasa `ContainerResource` – nu are adnotare de Path, şirul de caractere de după /containers/ fiind considerat un parametru care se parseaza la apelul metodei http. Metoda GET returnează răspuns doar dacă există acel container.
- Clasa `ItemResource` - nu are adnotare de Path, numele său este şirul de caractere de după /containers/[container_name] şi este instanţiată din metoda `geItemResource` a clasei ContainerResource. Are definite metode GET şi PUT, pentru extragerea resursei, respectiv pentru crearea/actualizarea acesteia.

Ultimele trei clase descrise fac parte dintr-o arhitectură exemplificată în aplicaţia Storage Server definită ca exemplu al utilizării Jersey[1]. Acest design reprezintă o alternativă foarte generică de stocat resurse, întrucât ierarhia adresei permite separarea logică a acestora. De asemenea, resursele sunt reţinute în format binar, din care se poate obţine orice alt tip de date, în funcţie de resursa stocată.

Din punct de vedere programatic, arhitectura Storage Server permite stocarea unor date care pot fi accesate în comun de resurse. Clasa care asigură stocarea se numeşte MemoryStore, şi membrii clasei au atributul *static,* pentru ca aceştia să fie instantiati la încărcarea clasei şi nu la o instantiere a sa. Excepţie fac structurile de date de tip *HashMap* care realizează maparile containerelor şi a numelor resurselor item la containere:

```
    private Map<String, Container> containerMap = new HashMap<String, Container>();
    private Map<String, Map<String, byte[]>> dataMap = new HashMap<String,
Map<String, byte[]>>();
```

Acestea sunt membri ale instanţei singleton `MemoryStore`, care conţine şi metode accesor pentru aceste structuri de date.

În primul subcapitol al detaliilor de implementare am detaliat URL-urile la care pot fi accesate resursele serverului. Cu excepţia documentului PDF şi a clasei aferente, celelalte resurse sunt accesibile prin acest format al URLului:

[1] Exemple Jersey, Storage-Service, 20.06.2012,
<http://docs.oracle.com/cd/E19776-01/820-4867/ggrby/index.html>

[adresa_server/containers/[nume_container]/[nume_item]

Containerele sunt: slidecontainer, positivefeedback, ambiguousfeedback, prooffeedback şi questions, cu semnificaţiile prezentate anterior. Deşi pachetele http întoarse ca răspuns pot avea formate diferite (plain/text sau Google protocol buffers), toate resursele sunt reţinute în forma byte[], pentru a beneficia de genericitatea designului. Serializarea şi deserializarea datelor din binar se face la nivelul clasei ItemResource, în funcţie de numele containerului.

4.4.3 Serializarea şi deserializarea datelor

Întrucât datele sunt reţinute în binar sub forma unui vector de octeţi în sistemul de stocare a serverului, este necesară deserializarea acestora atunci când se doreşte interpretarea sau prelucrarea datelor, respectiv serializarea lor atunci când se introduc sau reintroduc datele în structurile de date de stocare.

În cazul resurselor de tip "text/plain", deserializarea se face uşor datorită constructorului String(byte[]). Serializarea este simetrică, prin metoda `toByteArray()` a clasei String.

În cazul întrebărilor de feedback şi a celorlalte tipuri de feedback agregate, acestea sunt serializate şi deserializate folosind metodele clasei Google Protocol Buffers generate de executabilul protoc la crearea mesajelor .proto. Deoarece la introducerea sau retragerea oricărei întrebări este necesară o regruparesemantică a întrebărilor, la efectuarea unei cereri PUT sau DELETE sunt necesari paşi atât de serializare, cât şi deserializare, deşi în mod normal continiutul binar transmis în mesajul HTTP ar putea fi stocat direct. Acest lucru nu e posibil şi din motivul că formatul mesajului primit de la client printr-o metodă PUT este diferit de cel reţinut in binar pe server şi transmis clientului la o cerere GET.

Serializarea mesajelor binare a fost detaliată la capitolul referitor la mesajele transmise între clienţi şi server. Acest proces se face prin clasa intermediară Builder a clasei respective mesajului Google Protocol Buffers. Deserializarea acestor mesaje se face cu metoda `parseFromData(byte[])`.

4.4.4 Modulul de grupare a întrebărilor şi sincronizarea cu acesta

Pe lângă funcţionalitatea de serviciu web, care răspunde la cereri http, serverul aplicaţiei are şi rolul de a rula gruparea întrebărilor de feedback de fiecare dată când o nouă întrebare este pusă de o persoană de audienţă, deci o cerere PUT este efectuată pentru o întrebare pe o anumită selecţie. Fluxul logic în acest caz este următorul:

1. utilizatorul formulează o întrebare pe dispozitivul Android, care este încapsulată într-un mesaj FeedbackQuestion (mesaj Google Protocol Buffers) şi trimisa într-o cerere PUT la adresa selecţiei pentru care este adresată întrebarea
2. serverul web preia cererea, observă tipul cererii şi este apelată o metodă a modului de clusterizare
3. modulul de clusterizare preia întrebarea ca input, citeşte binarul deja existent stocat în server (într-o formă deja agregată, în formatul `SelectionsClusters`), îl deserializeaza pentru a obţine întrebările şi realizează din nou clusterizarea, preluând şi întrebarea de de input.
4. modulul de cluster serializează înapoi în formatul SelectionsClusters cu noile clustere de întrebări, apoi este refăcut mesajul container şi binarul obţinut este salvat în `MemoryStorage`.

Modulul de clusterizare a fost devoltat separat, astfel încât eu l-am integrat într-un thread separat, care este pornit odată cu serverul.

Problemele evidente apar la sincronizarea între threadurile create în background pentru cererile web şi threadul modulului de clustering. Pentru a rezolva această situaţie, am folosit un obiect simplu ca lock, acesta fiind vizibil tuturor claselor serverului ca membru al clasei `MemoryStore`. De asemenea, toate datele modificate la execuţia clusterizarii sunt membri ai clasei `MemoryStore`. Toate aceste resurse accesate în comun sunt modificate într-o zonă sincronizată după `MemoryStore.threadlLock [11]`.

Astfel, threadul de clusterizare se blochează în ciclul de rulare la apelul `MemoryStore.threadlLock.wait()`, urmând ca acesa să fie atenţionat cu notify din threadul serverului web atunci când o nouă întrebare este adresată. În threadul cererii http, atunci când soseşte o nouă întrebare, se intră într-o zonă sincronizată în care întâi se construieşte întreg setul de întrebări, deserializându-se binarul deja reţinut la care se adăugă noua întrebare, apoi se apelează `notify` pentru a reporni threadul clusterer. Acesta preia setul de întrebări din `MemoryStore` şi reface clusterizarea, apoi serializează noile clustere într-un nou mesaj `SelectionClusters`, care este suprascris peste binarul stocat la URL-ul selecţiei. La sfârşit, acest thread iese din synchronized şi repetă operaţia, blocându-se în wait.

```
public void run()
{
	QuestionHierClusterer clusterer = new QuestionHierClusterer();
	clusterer.init();

	while (true)
	{
		clusterQuestions(clusterer);
	}
}
public void clusterQuestions(, QuestionHierClusterer clusterer)
{
	synchronized (MemoryStore.threadLock)
	{
		try {
			MemoryStore.threadLock.wait();

			Set<InputQuestion> input  =
clusterer.setInputQuestionsFromArray(MemoryStore.feedbackQuestions);
			Set<Set<Question>> tree = clusterer.cluster(input);

			storeClusters(clusterer, tree);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
}
```

Exemplu cod de rulare al threadului de clustering

4.5 Interfaţa de utilizare a clienţilor Android

Interfaţa aplicaţiei oferă clientului posibilitatea utilizării întregii funcţionalităţi a modului de comunicaţie client-server al aplicaţiei Smart Presentation. Elementele vizuale permit utilizatorului să acorde oricare tip de feedback dintre cele suportate de server, unul din cele trei tipuri fixe sau feedback sub forma unei întrebări. De asemenea, clientul are posibilitatea sincronizării prezentării cu cea a speakerului, prin apăsarea butonului "Go live!". Odată intrat în acest mod, clientul poate ieşi prin apăsarea oricărui buton care obligă păstrarea slideului curent, cum ar fi intenţia de selecţie a unor elemente din prezentare sau acordarea unui feedback.

Speakerul accesează serverul în mod indirect, prin schimbarea slideului, sau în mod direct, la sfârşit, când doreşte vizualizarea întregului feedback. Scenariile de utilizare sunt detaliate la capitolul următor.

4.5.1 Interfaţarea cu prezentarea, selecţia elementelor

Pentru acordarea feedbackului pe o selecţie a documentului PDF, trebuie întâi realizată selecţia şi reprezentată într-un mod în memorie. Selecţia se declanşează la apăsarea unui buton, pentru a nu se încerca o selecţie eronată, sau pentru necesitatea resetării selecţiei făcute.

Selecţia se face prin alegerea a doua puncte din slide, cel de capăt stânga sus şi cel de capăt dreapta jos. În momentul realizării selecţiei, este generat un șir de caractere reprezentativ pentru selecţie. Acest script cuprinde numărul slide-ului şi id-uri ale elementelor din slide astfel încât să se poată reconstitui selecţia din șirul de caractere, necesară la prezentarea feedbackului pentru speaker.

Id-ul selecţiei folosit pentru accesarea şi stocarea resurselor pe serverul web este numărul întreg obţinut din aplicarea metodei hashCode() asupra stringului. Atunci când se realizează o selecţie pe documentul PDF, se reţine această mapare între string şi hash-codul sau, pentru a fi posibilă operaţia inversă de determinare a șirului de caractere şi, deci, a selecţiei vizuale din id-ul selecţiei.

4.5.2 Metodele handler ale butoanelor

Butoanele aplicaţiei intră, la fel ca şi ferestrele, la modulul de interfata Android. Legarea acestora de modulul de comunicaţie cu serverul presupune apelarea interogărilor atunci când un buton este apăsat. În clasa MuPDFActivity, clasa de startup a aplicaţiei Smart Presentation, există definită o clasa handler comună pentru toate butoanele. Alegerea handlerului potrivit fiecărui buton se realizează în metoda `OnClick`, prin verificarea callerului [13].

Handlerele butoanelor aplicaţiei pot efectua două tipuri de operaţii, din punct de vedere al comunicaţiei şi al accesării datelor: pot instanția o clasă AsyncTask şi să apeleze execute pe aceasta, sau pot realiza o interogare managedQuery a Content Providerului, asupra căruia cade responsabilitatea efectuării cererii asincrone către server, dacă datele interogate nu sunt în baza de date sau sunt expirate (eng. *)*.

Butoanele care efectuează cereri directe care web server prin clase AsyncTask sunt cele care trimit feedback de tip fix către server. La fel este şi cazul butonului "Go live!", care declanşează o acţiune repetitivă în cadrul unei clase AsyncTask, de sincronizare a slide-ului cu cel existent pe server, până la apăsarea unui alt buton care provoacă ieşirea din modul Live.

5. Utilizarea aplicaţiei

5.1 Descrierea aplicaţiei

Aplicaţia Smart Presentation permite utilizatorilor unor dispozitive Android care iau parte la o prezentare să interacţioneze în timp real cu aceasta, existând două moduri de utilizare: speaker şi simplu uşer, persoana aflată în audienţă. Scopul principal al aplicaţiei este să permită celor din audienţă să poată selecta elemente din slide-ul curent pe care să acorde diverse tipuri de feedback, cum ar fi: feedback pozitiv, în semn de apreciere, feedback de ambiguitate, care indică necesitatea unor clarificări suplimentare sau feedback de proof sau citation needed, pentru exprimarea necesităţii citării sursei. În plus, utilizatorii pot formula întrebări legate de selecţia făcută adresate speakerului, sau pot vedea întrebările deja reţinute de la alţi ascultători pentru a-şi exprima acordul cu una din ele. Întrebările sunt grupate semantic pe un server central, astfel încât toţi userii să poate vedea doar o forma compactă a întrebărilor, fiind selectate doar cele reprezentative din punct de vedere semantic.

O altă facilitate a aplicaţiei este posibilitatea de navigare liberă prin prezentarea descărcată pe propriul dispozitiv, sau folosirea modului live, care sincronizează permanent prezentarea cu cea a slide-ului speakerului până când modul este dezactivat.

La sfârşitul prezentării, speakerul poate vedea pe documente diversele feedbackuri primite din audienţă pe timpul prezentării astfel încât să poată răspunde mai eficient la întrebări şi să îşi îmbunătăţească prezentarea pentru viitor.

5.2 Scenarii utilizare – screenshots

Pentru exemplificarea scenariilor de utilizare, vom presupune existenţa următoarelor persoane, cu următoarele roluri: Dan este prezentatorul, Alina, Andrei şi Elena sunt simpli ascultători. Toţi patru deţin un dispozitiv Android şi documentul PDF este stocat pe un server aflat în apropiere care poate fi accesat prin mediul wireless.

În momentul pornirii aplicaţiei, este declanşată o cerere către server de descărcare a documentului PDF stocat pe acesta. La primirea răspunsului şi scrierea documentului în memorie, cei patru utilizatori sunt întâmpinaţi cu o fereastră care le permite să selecteze rolul pe care îl vor avea: speaker sau simplu uşer. Dan alege să fie speaker, ceilalţi trei aleg să fie simpli uşeri. După alegerea rolului, cei patru îşi aleg numele cu care vor fi reţinuţi în sistem (Figura 10).

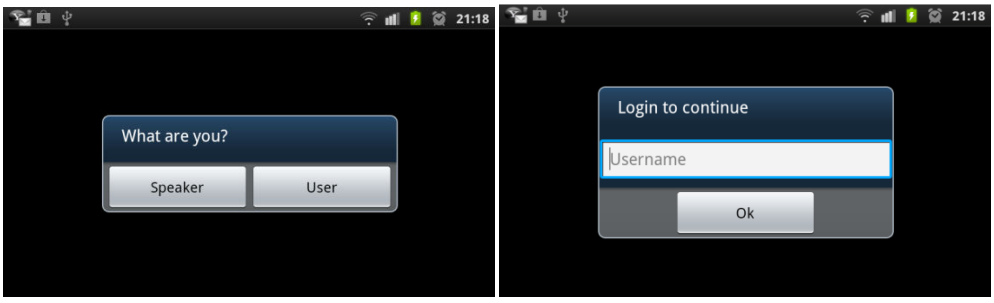


Fig. 10 Ferestrele iniţiale ale aplicaţiei

În următoarele situaţii sunt descrise activităţile userilor din audienţă, adică Alina, Andrei şi Elena. După alegerea numelui, este încărcată prezentarea de la început şi aceştia pot vedea interfaţa aplicaţiei (Figura 11).

Fig. 11 Interfaţa clientului în aplicaţie

În continuare, Alina hotareşte să parcurgă aşa cum doreşte prezentarea, pentru ca este curioasă de conţinutul acesteia. Andrei şi Elena hotaresc să fie sincronizaţi cu prezentarea ţinută de Dan, pentru a fi mai atenţi la ceea ce spune acesta. Astfel, cei doi apăsă primul buton din stânga jos, cel de Go live!, care le permite intrarea în modul sincronizat.

Pe parcursul prezentării, Alina hotareste să facă o selecţie pentru a trimite un feedback de ambiguous. Pentru a face asta, ea apasă pe al doilea buton, cel marcat cu T, şi realizează selecţia. Dacă nu este mulţumită cu selecţia făcută, ea poate reseta selecţia prin apăsarea aceluiaşi buton.

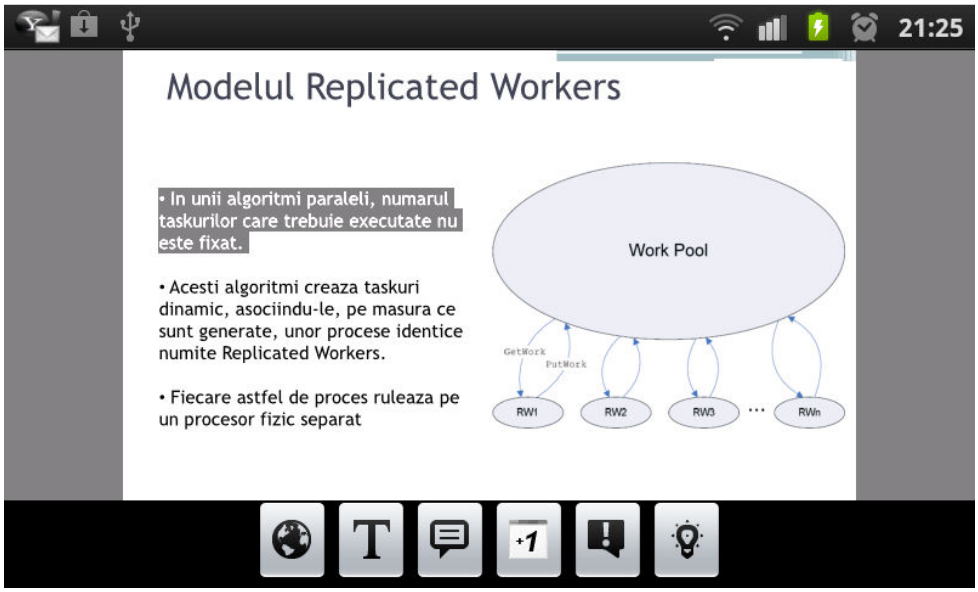


Fig. 12 Selecţie făcută pe o porţiune de text

În acest timp, Andrei şi Elena se află la alt slide, cel la care se află şi Dan, aceştia fiind în modul Live. Andrei se hotareste să pună o întrebare legată de titlul slideului. Pentru aceasta, el selectează întâi titlul (Figura 13).

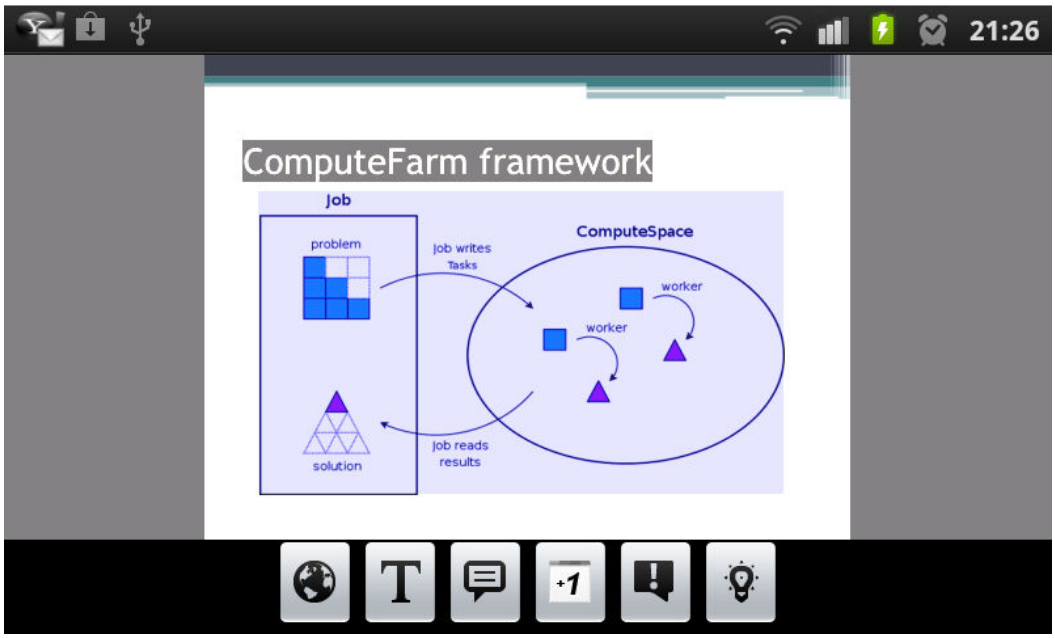


Fig. 13 Selecţie făcută pe titlu

Apoi, acesta apăsa pe al treilea buton de jos, care îi arăta întrebările semnificative deja formulate pentru selecţia făcută:

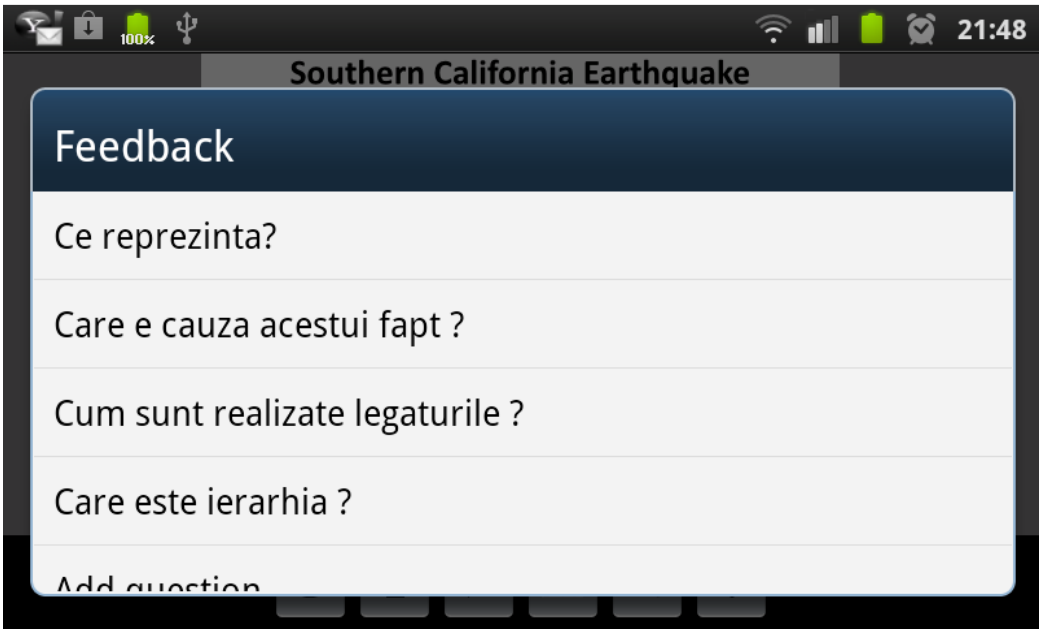


Fig. 14 Lista de intrebari

Andrei are posibilitatea de a alege o întrebare cu care să-si exprime acordul, sau de a alege elementul Add question din listă, care-i permite scrierea unei întrebări noi.

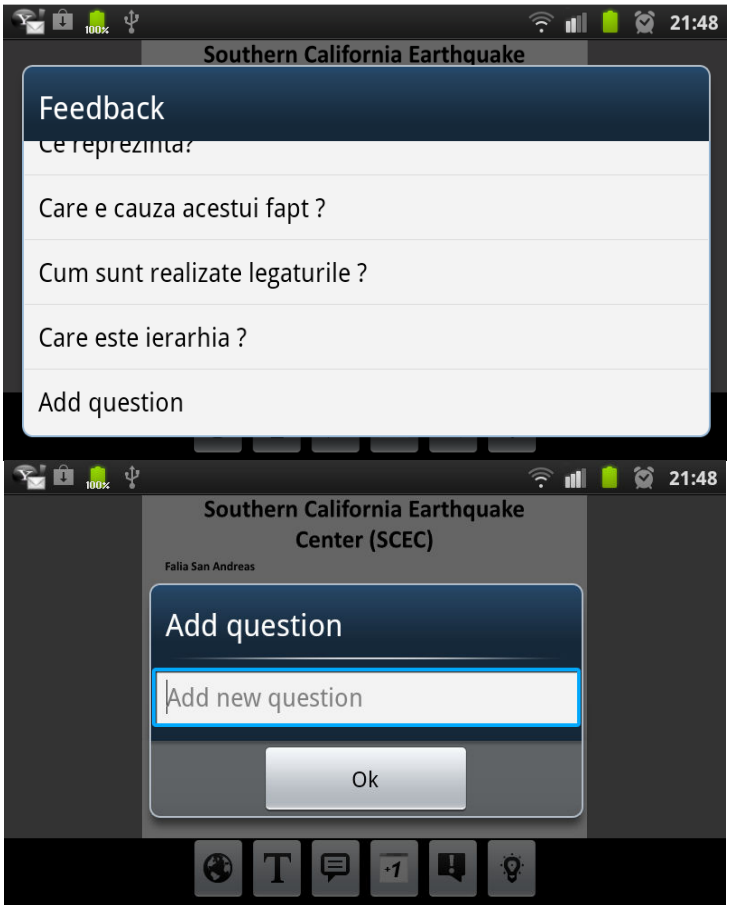


Fig. 15 Adăugarea unei noi întrebări

Pe parcursul prezentării, Elena a acordat mai multe feedbackuri de tip ambiguous (penultimul buton) sau proof needed (ultimul buton), şi a remarcat că, pentru unul din concepte, explicaţia se află deja în prezentare, motiv pentru care se întoarce la slide-ul respectiv şi îşi retrage feedbackul pe aceeaşi selecţie.

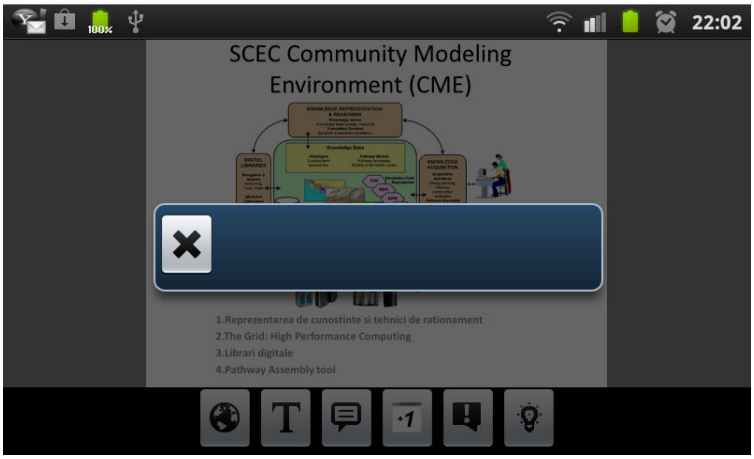
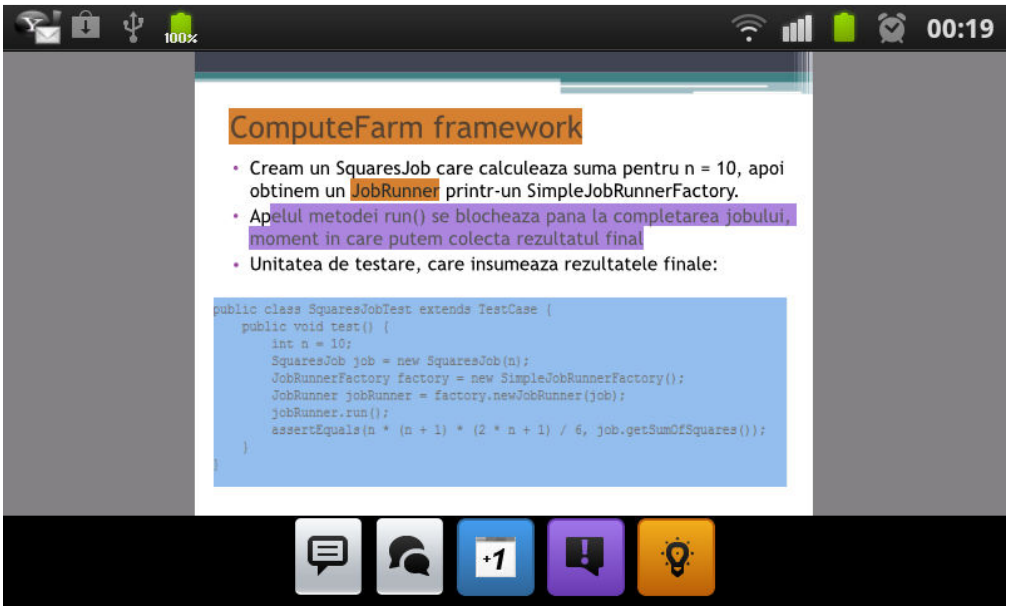


Fig. 15 Adăugarea unei noi întrebări

În timpul prezentării, Andrei, Elena şi Alina acordă mai multe feedbackuri de tip +1, pentru a-şi exprima o părere pozitivă legată de elementele selectate.

La sfârşitul prezentării, Dan poate utiliza interfaţa proprie, care îi arată pentru fiecare tip de feedback selecţiile făcute pe document, pentru a asocia corect feedbackul cu elementele asupra cărora acesta a fost acordat.



6. Concluzii

Dezvoltarea aplicaţiei Smart Presentation a necesitat utilizarea multor cunoştinţe practice şi teoretice învăţate în facultate, cum ar fi: protocoale de comunicaţii, programare şi design orientate obiect, structuri de date, calcul paralel şi sincronizare între threaduri, baze de date sau ingineria programelor. În plus, au fost necesare acumularea unor cunoştinţe teoretice noi, cât şi învăţarea utilizării unor tehnologii noi.

Dezvoltarea serverului mi-a permis să studiez arhitectura RESTful pentru servicii web şi să învăţ să implementez un astfel de serviciu folosind framework-ul Jersey şi utilizarea containerului Grizzly. Prin studierea diverselor exemple am ales să implementez un design ierarhic al claselor resursa http, şi o clasă de stocare a datelor, accesibilă din toate resursele serverului. De asemenea, am folosit elemente de detaliu ale protocolului http, cum ar fi metodele specifice protocolului şi headerul de content-type, care mi-a permis să trimit diferite tipuri de date în pachetele http.

Implementarea modulului suplimentar de grupare a ridicat probleme de sincronizare, datorită accesului comun la date din threaduri separate şi a necesităţii creării unui mecanism de comunicare între threaduri.

Din punct de vedere al clientului, am învăţat concepte importante de programare pe sistemul de operare Android. În primul rând, ideea de entry point sau clasa main sunt inexistente pe Android, unde serviciile şi procesele lansate la execuţia unei aplicaţii sunt definite într-un fişier de configurare. În al doilea rând, din punct de vedere al gândirii arhitecturale, programarea pe Android forţează utilizatorul să separe funcţionalităţile aplicaţiei după modelul arhitectural Model-View-Controller. Pentru respectarea acestui pattern, am folosit numeroasele clase oferite de sdk-ul Android care facilitează rularea paralelă a taskurilor şi sincronizarea cu threadul principal. Stocarea datelor se face într-un mod canonic într-o bază de date Sqlite care asigură persistenţa datelor. În cazul meu, am folosit o astfel de bază de date pentru persistenţa unor informaţii, similar cu păstrarea unor structuri de date, dar într-un mod mai robust şi care permite accesul simplu, prin adrese URI.

Pentru a realiza cererile HTTP de la client către server, am explorat versatilitatea clasei AsyncTask, prin care dezvoltatorului i se oferă toate mecanismele necesare sincronizării threadului apelant cu cel de background şi actualizarea datelor şi a interfeţei cu cele incluse în răspunsul primit de la server.

Comunicaţia între client şi server mi-a lăsat la alegere conceperea diverselor tipuri de mesaje, serializate în mod diferit şi cu semnificaţii diferite, care alcătuiesc protocolul aplicaţiei. Pentru mesajele largi, de agregare a feedbackului de întrebări, am ales să folosesc mecanismul de serializare binară Google Protocol Buffers, o soluţie optimă şi foarte uşor de implementat, datorită generării automate a claselor Java dintr-un format de definire a mesajelor proprietar, user-friendly.

Rezultatul final al lucrării s-a transpus într-o aplicaţie care îndeplineşte obiectivele formulate iniţial:

- posibilitatea sincronizării prezentării unui client Android din audienţă cu cea a speakerului;
- acordarea de feedback de tip positive feedback, ambiguous, proof needed sau sub forma unei întrebări pe o selecţie a unui slide sau pe întregul slide curent al prezentării;
- vizualizarea feedbackului agregat în timp real şi posibilitatea exprimării acordului cu feedback deja formulat de alte persoane;
- retragerea feedbackului propriu efectuat;
- interfeţe diferite de utilizare pentru speaker şi audienta;
- vizualizarea de către speaker a unei variante finale, agregate a tuturor tipurilor de feedback acordate de audienţă.

În viitor, aplicaţia poate fi dezvoltată, pentru a permite diverse facilităţi noi. În primul rând, este necesară gruparea mai eficientă a întrebărilor, bazată pe un algoritm care să grupeze mai inteligent întrebările pe baza similarităţii semantice şi care să aleagă un centroid al clusterelor reprezentativ pentru întrebările grupate. În prezent, gruparea se bazează în special pe cuvinte de bază şi nu pe cuvinte cheie care pot schimba sensul unei întrebări.

Alte îmbunătăţiri care pot fi aduse aplicaţiei sunt: o interfata mai interactivă şi informativă a aplicaţiei, care să permită o vizualizare mai intuitivă şi mai cuprinzătoare a feedbackului, devoltarea unui sistem de autentificare pentru a acorda privilegii diferite uşerilor, posibilitatea încărcării în realtime a unui document nou de prezentare sau dezvoltarea unei interfeţe grafice a serverului, care să permită o configurare a acestuia.

De asemenea, ar fi interesant/util de implementat o clasificare a userilor, în funcţie de o reputaţie a acestora, care să le acorde o pondere mai mare propriilor feedbackuri în faţa unor utilizatori noi.

7. Bibliografie

[1] *Wikipedia, Android Operating systems*, 20.06.2012, <http://en.wikipedia.org/wiki/Android_%28operating_system%29>

[2] Zigurd Mednieks, Laird Dornin, G. Blake Meije, *Programming Android, 2nd Edition,* O'Reilly, 2011

[3] *Documentaţia oficială Android*, 20.06.2012, <http://developer.android.com>

[4] International Organization for Standardization, *ISO 32000-1:2008 Document management -- Portable document format -- Part 1: PDF 1.7*, 20.06.2012, <http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/PDF32000_2008.pdf>

[5*] Website Grizzly*, 20.06.2012, <http://grizzly.java.net>

[6] Sameer Tyagi, *RESTful Web Services,* < http://www.oracle.com/technetwork/articles/javase/index-137171.html>

[7] *Documentaţia oficială Jersey,* 20.06.2012, < http://jersey.java.net/nonav/documentation>

[8] *RESTful Web Services Developer's Guide*, 20.06.2012, <http://docs.oracle.com/cd/E19776-01/820-4867/ggrby/index.html>

[9] *Documentaţia oficială Google Protocol Buffers*, 20.06.2012, <https://developers.google.com/protocol-buffers/>

[10] *Vogella tutorials*, 20.06.2012, <http://www.vogella.com/articles/AndroidSQLite/article.html>

[11] Scott Oaks, Henry Wong*, Java Threads, 3rd Edition,* O'Reilly, 2004

[12] RESTful Web Services Developer's Guide, *Chapter 5 - Jersey Sample Applications*, 20.06.2012 <http://docs.oracle.com/cd/E19776-01/820-4867/ggrby/index.html>

[13] Marko Gargenta*, Learning Android - Building Applications for the Android Market*, O'Reilly, 2011