Politechnika Łódzka

Instytut Informatyki

PRACA DYPLOMOWA INŻYNIERSKA

Platforma do zarządzania wiadomościami udostępnianymi przez kanały RSS

Wydział Fizyki Technicznej, Informatyki i Matematyki Stosowanej
Promotor: dr inż. Aneta Poniszewska - Marańda
Dyplomant: Krzysztof Klimczak
Nr albumu: 143001
Kierunek: Informatyka
Specjalność: Inżynieria oprogramowania i analiza danych

Łódź 4.02.2011

Instytut Informatyki

90-924 Łódź, ul. Wólczańska 215, budynek B9

tel. 042 631 27 97, 042 632 97 57, fax 042 630 34 14 email: office@ics.p.lodz.pl

Spis treści

Wstęp

W dzisiejszych czasach nie ma problemu z dostępem do informacji. Gazety, telewizja, radio serwują nam codziennie ogromną dawkę wiadomości z całego świata. Źródła te mają jedną wadę: informacje przekazywane przez nie pojawiają się z pewnym opóźnieniem, a do tego nie można do niech wrócić (wyjątek w tej grupie stanowi prasa, gdzie możemy zawsze skorzystać z egzemplarzy archiwalnych). Żyjemy jednak w XXI wieku, który może się poszczycić ogromnym rozwojem elektroniki oraz Internetu. Globalna sieć jest już dostępna praktycznie dla każdego, a wiele osób nie wyobraża sobie bez niej życia. Informacje z całego świata (a nawet wszechświata), z każdej dziedziny życia, pojawiają się w tym medium niemal natychmiastowo. Nie ma czym się martwić, jeśli coś przegapimy, zawsze możemy otworzyć odpowiednią stronę i przeczytać interesujący nas artykuł, do tego w przeciwieństwie do gazet, nie musimy wiedzieć dokładnie gdzie, kiedy i czego szukać, wystarczy wyszukać interesujący nas artykuł poprzez określenie, co może on zawierać, a w rezultacie otrzyma się grupę wiadomości, które mogą być tym czego szukaliśmy.

Tutaj jednak pojawia się problem, nie możemy przecież odwiedzać wszystkich stron co chwilę i sprawdzać, czy nie pojawił się jakiś nowy artykuł. Do tego dochodzą przecież indywidualne zainteresowania każdego użytkownika, czyli mówiąc krótko, wszyscy mamy inne zainteresowania, zatem nie każda wiadomość będzie przez nas przeczytana. Niewygodnie staje się więc ciągłe przeglądanie stron lub też rozwiązanie, by poczekać na koniec dnia i przejrzeć wszystkie wiadomości, czytając tylko te, które nas zainteresują. Dlaczego to jest niewygodne? Nie mamy dostępu do najświeższych wiadomości (sprowadzamy internetowy serwis do roli gazety, która publikuje artykuły periodycznie). Do tego dochodzi fakt, że musimy często przejrzeć wiele podstron danego serwisu, ignorując nieciekawe z naszego punktu widzenia wiadomości. Może się do tego zdarzyć sytuacja, że danego dnia nie będzie artykułu, który spełniałby nasze oczekiwania, wtedy tracimy tylko czas na przeglądanie serwisu.

Opisany powyżej problem można w dość prosty sposób rozwiązać dzięki kanałom **RSS**

(ang. *Really Simple Syndication*). Czym one są? Jest to zbiór dokumentów opartych na języku **XML** (ang. *eXtensible Markup Language*, w wolnym tłumaczeniu *Rozszerzalny Język Znaczników*), służących do publikacji treści w Internecie, które się często zmieniają (na przykład na blogach, portalach informacyjnych) [8]. Jest to technologia idealna dla osób chcących być na bieżąco z różnymi informacjami. Kanały *RSS* wykorzystywane są również do publikowania *podcastów* i *videocastów*, czyli radiowych lub telewizyjnych audycji. W zależności od serwisu mogą one zawierać jedynie skrót/fragment danej wiadomości, która ma na celu zachęcenie czytelnika do odwiedzenia głównej strony zawierającą pełny artykuł. Mogą też zawierać cały tekst wraz z dołączoną grafiką (niestety pobieranej osobno). Aby cieszyć się tymi wszystkimi dobrodziejstwami, wystarczy posiadać odpowiedni program, który potrafi przetwarzać dokumenty z wiadomościami i je wyświetlać. Oczywiście niezbędny jest również dostęp do Internetu.

Co jednak w sytuacji, gdy nie ma dostępu do sieci lub chcielibyśmy sprawdzać nasze wiadomości na innym systemie operacyjnym/urządzeniu. Tutaj pojawia się pewien problem. Sam dokument *RSS* zawiera ograniczoną listę najnowszych wpisów. Przy dłuższym braku dostępu do Internetu, po prostu przegapi się część wpisów i jedynym rozwiązaniem tej sytuacji jest wejście na daną stronę i odszukanie wiadomości, których nie czytaliśmy. Na tym samym komputerze można posiadać wiele systemów operacyjnych z zainstalowanym klientem *RSS* (może to być ten sam program, jeśli jest wieloplatformowy), ale wiadomości przez niego zbieranie i prezentowane nie będą ciągłe (sytuacja podobna, jak przy braku połączenia z siecią). Istnieją rozwiązania zdalne, czyli takie, które nie wymagają posiadania lokalnego klienta, ale ich podstawową wadą jest to, że i tak trzeba mieć dostęp do Internetu, gdyż nie oferują możliwości pracy *offline* lub też synchronizacji pomiędzy klientami (niektóre czasem oferują możliwość pobrania swojej zawartości przez jakiś program).

Cel pracy

Celem poniższej pracy dyplomowej jest stworzenie platformy, która umożliwiłaby pokonanie wyżej wymienionych problemów, czyli by niezależnie od miejsca i czasu możliwy był dostęp do tych samych danych, niezależnie od używanego systemu operacyjnego lub urządzenia. Stan wiadomości powinien być zawsze taki sam, niezależnie od miejsca używania systemu.

Zakres pracy

Z uwagi na ograniczony czas, trudno jest stworzyć kompletną platformę, czyli taką, która umożliwiłaby od razu pełny i uniwersalny dostęp do wiadomości. W związku z tym, w pracy ograniczono się jedynie do stworzenia odpowiedniego oprogramowania serwerowego oraz aplikacji klienckiej, działającej w systemie użytkownika. Zatem bez aplikacji (na przykład z poziomu przeglądarki internetowej w szkole, pracy) nie będzie możliwe przeglądanie wiadomości, jednak nie ograniczono tej możliwości, tzn. każda aplikacja internetowa będzie mogła podłączyć się do stworzonego serwera i wymieniać dane.

Struktura pracy

Poniższa praca dyplomowa została podzielona na cztery rozdziały, w których, w sposób spójny przedstawiono temat pracy.

Rozdział pierwszy zawiera opis oraz porównanie istniejących na rynku rozwiązań w dziedzinie dostępu do wiadomości *RSS*. Opisano w nim, z jakimi problemami użytkownicy aktualnie się spotykają lub mogą się spotkać.

W drugim rozdziale opisano propozycję rozwiązania problemów omówionych w rozdziale pierwszym. Zawarta w nim została też cała idea dotycząca budowy projektu oraz to, na czym należy skupić uwagę przy projektowaniu aplikacji.

Kolejny rozdział to dokumentacja techniczna projektu. Przedstawiono w niej opis realizacji aplikacji, która powstała do poniższej pracy dyplomowej.

Rozdział czwarty to dokumentacja użytkownika. Jest to przewodnik po interfejsie użytkownika stworzonego projektu. W tym rozdziale opisano jak używać aplikacji, wyjaśniono znaczenie i przeznaczenie poszczególnych opcji ustawień.

Rozdział 1

Ogólna problematyka pracy

W poniższym rozdziale opisano aktualnie występujące na rynku rozwiązania w dziedzinie aplikacji służących do odczytu kanałów *RSS*, starając się uwypuklić niedoskonałości tych projektów, bądź wskazując, w którym obszarze mogłyby się poprawić. Skupiono się przy tym jednak tylko na aplikacjach darmowych, nie wymagających żadnych opłat abonamentowych lub rejestracyjnych.

1.1 Kanały RSS

Kanały *RSS* to dokumenty oparte na języku *XML*. Zawierają one streszczoną formę wiadomości ze skojarzonej strony *WWW* lub jej pełny tekst. Umożliwia to użytkownikom bycie na bieżąco z treścią ulubionych serwisów społecznościowych w sposób automatyczny, gdyż programy obsługujące dokumenty *RSS* same pobierają ich zawartość.

Specyfikacja *RSS 2.0* jest własnością Uniwersytetu Harvarda [7, 8, 9]. Nie jest ona już rozwijana. Nie oznacza to jednak, że format ten jest doskonały i nie wymaga poprawek. Zawiera on wiele niejasności oraz błędów. Do poważniejszych można zaliczyć brak określenia, jak zamieszczać i obsługiwać w treści wiadomości znaczniki *HTML*. Istnieją dwa popularne rozwiązania tego problemu, ale nie stanowią one oficjalnych rozwiązań. Pierwszym jest zamiana wszystkich znaków nietworzących wyrazy na encje języka *XML*. Czytnik *RSS* musi jednak obsługiwać takie rozwiązanie i ,,domyślać się'', że autor wiadomości chciał zamieścić tagi *HTML*, a nie na przykład po prostu encje. Drugim rozwiązaniem problemu jest umieszczenie całej treści *HTML* w sekcji *CDATA* języka *XML*. Taka zawartość nie jest przetwarzana przez *parser XML*, może więc zawierać

dowolny tekst w dowolnym języku, oznaczony w dowolny sposób. To rozwiązanie jest najczęściej stosowane.

Innym niedociągnięciem jest brak określenia, w jaki sposób klient ma obsługiwać aktualizacje w obrębie wpisów. Może się zdarzyć sytuacja, iż w wyniku pomyłki zostaną umieszczone na kanale błędne dane. Dokument *RSS* przez klienta może z tym błędem zostać pobrany. Pojawia się jednak problem, jak poprawić ten błąd (sytuacja wygląda analogicznie w przypadku aktualizacji do danej wiadomości, gdy taka pojawia się na portalu). Specyfikacja nie określa, jak wydawca kanału ma się zachować - edytować istniejący wpis, dodać nowy lub wykonać jeszcze inną czynną. Analogicznie sytuacja odnosi się do klienta subskrybującego kanał. Dodanie nowej wiadomości z poprawionym błędem, potraktowane zostanie najczęściej jako nowy wpis. Edycja wpisu może nie zostać nawet zauważona. By uniknąć tej sytuacji stosuje się zabieg z aktualizacją daty publikacji wiadomości (choć jest to pole nieobowiązkowe). Każdy klient inaczej radzi sobie z obsługą takich sytuacji. Jest to jedynie zależne od twórców aplikacji i ich wizji rozwiązania tego problemu (na przykład podmiana danych na nowsze, przechowywanie wszystkich kopii wiadomości).

Sama konstrukcja dokumentów miała, wedle założeń, być prosta i łatwa do interpretacji. Nie zawiera więc żadnych rozszerzeń. Dopiero, kiedy format ten stał się popularny, udostępniono deweloperom możliwość dołączania własnych modułów, poprzez zastosowanie przestrzeni nazw (zgodnie ze specyfikacją do standardu *XML 1.0*). Każdy więc może rozszerzyć ten dokument o pola mu potrzebne. Nie ma jednak obowiązku ich interpretacji. Jednak takie rozwiązanie przydatne jest często w firmach do publikacji własnych wiadomości, dostosowanych pod jej profil działalności.

Do każdej wiadomości może być dołączony plik. Jest to realizowane poprzez dołączenie do wpisu jego adresu, umieszczonego w nieobowiązkowym znaczniku *<enclosure .../>*. Taka technika jest stosowana przy publikacji *podcastów*, *videocastów* lub przez serwisy torrentowe, dzięki czemu dane są pobierane automatycznie, wedle ustalonych przez użytkownika reguł.

Brak wsparcia, zaprzestanie rozwijania formatu oraz liczne błędy, doprowadziły do powstania alternatywnych kanałów pod nazwą *Atom*. Rozwiązują one problemy *RSS*, a ich specyfikacja jest określona w dokumentach RFC. Nie są jednak tak rozpowszechnione i preferowane jak *RSS 2.0*. Dlatego nie zostało uwzględnione w tej pracy.

1.2 Dostęp do sieci

Jak wspomniano we wstępie, by cieszyć się dobrodziejstwami kanałów **RSS** potrzeba stałego dostępu do Internetu. W przypadku jego braku nie pobierze się nowych wiadomości, a w zależności od rodzaju używanego klienta możliwe, że nawet nie przeczyta się już pobranych informacji, gdyż nie będzie się posiadać do nich dostępu (klient zdalny). Jest to ważna kwestia, szczególnie wśród ludzi często podróżujących, gdzie dostęp do sieci jest często ograniczony (na przykład w pociągach, gdzie Internet mobilny traci zasięg z powodu dość szybkiego przemieszczania się pomiędzy stacjami bazowymi) lub wręcz niemożliwy (na przykład w samolocie, gdzie w celach bezpieczeństwa zaleca się wyłączenie wszystkich urządzenia elektronicznych, bądź przestawienie ich w tak zwany ,,tryb samolotowy'', który dezaktywuje wszystkie nadajniki i odbiorniki w urządzeniu, lecz samo urządzenie działa poprawnie).

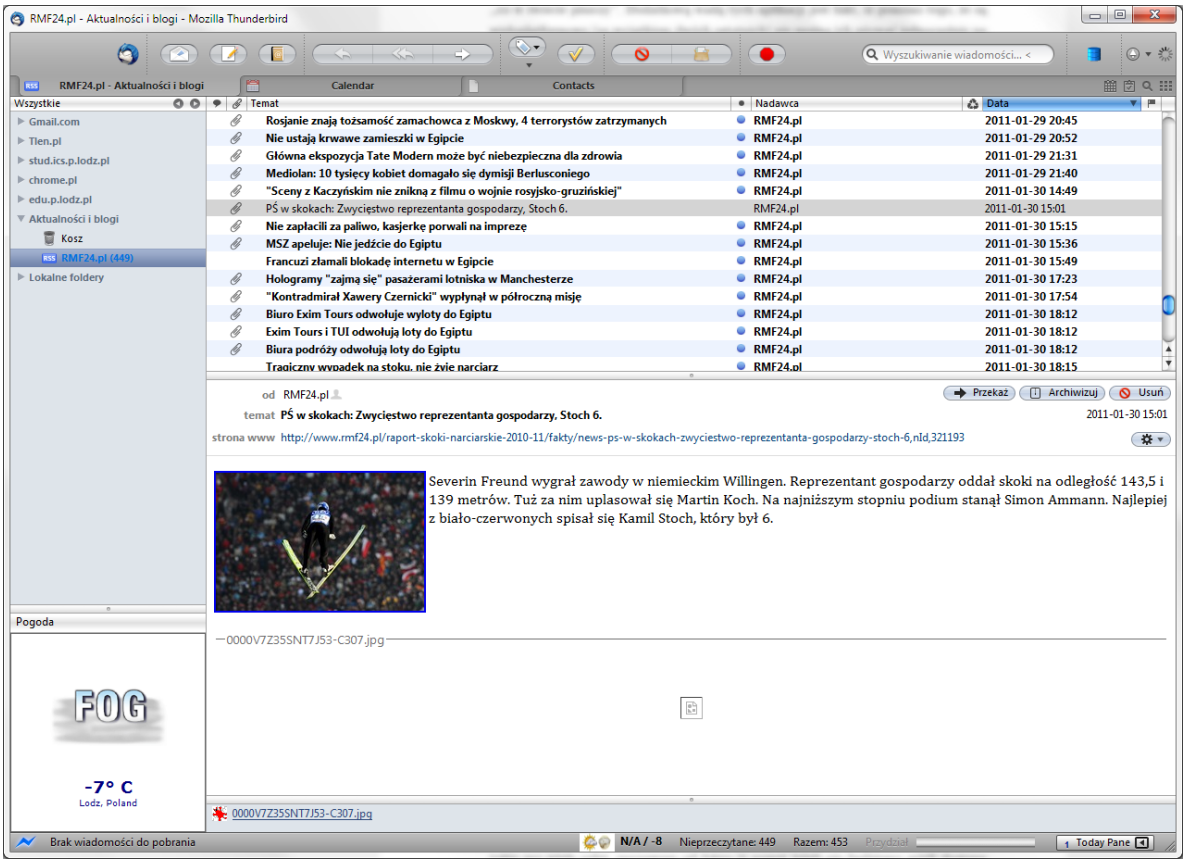
Obecnie na rynku można wyróżnić dwie grupy aplikacji (pomijając platformy, na których działają): internetowe oraz stacjonarne.

1.2.1 Aplikacje stacjonarne

Ten typ programów pozwala na pobieranie wiadomości i czytanie ich później *offline* na własnym komputerze/urządzeniu, bez wymaganego połączenia z Internetem. Jednak bez dostępu do Sieci, nie ma możliwości pobrania nowych artykułów lub też zsynchronizowania stanu czytnika z innym medium (serwer, portal). Jest to rozwiązanie dobre, jeśli korzysta się tylko z jednego komputera i systemu, a przerwy w dostępie do sieci są krótkotrwałe. Wtedy aplikacja pracująca w tle będzie zawsze sprawdzać i uaktualniać naszą lokalną bazę wiadomości. Wymaga to jednak, by komputer użytkownika był włączony bez przerwy.

Istnieje wiele aplikacji tego typu, jedne mają większą funkcjonalność inne mniejszą:

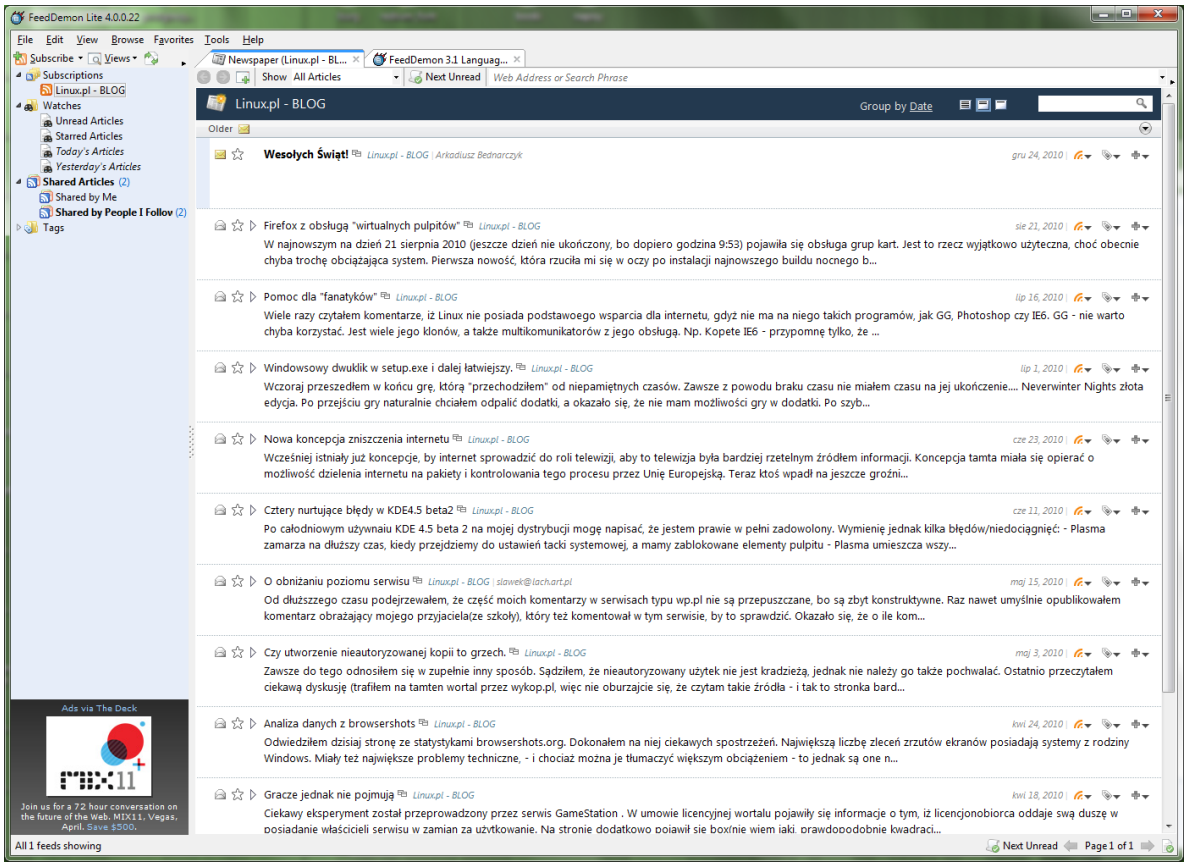
- *Mozilla Thunderbird*,
- *Mozilla Firefox* (dynamiczne zakładki),
- *Akregator* - główna platforma to Linux, ale dzięki zapałowi wolontariuszy i faktu, iż jest napisany przy użyciu biblioteki *Qt4*, działa też w systemie Windows i MacOS,
- *Opera*,
- *FeedDemon* - dostępny tylko w systemie Windows,



Rys. 1.1: Okno z wiadomościami programu *Thunderbird*

- *RssBandit* - dostępny tylko w systemie Windows.

Wszystkie te aplikacje cechują się podobnym wyglądem i układem zawartości (Rys. 1.1 i 1.2), wyjątek stanowi tutaj *Mozilla Firefox* (Rys. 1.3). Aplikacje te nie udostępniają funkcji synchronizacji oprócz dwóch ostatnich pozycji, ale działają praktycznie tylko jednokierunkowo, czyli do klienta i żadne zmiany dotyczące samych wiadomości nie są wysyłane do serwera (warto odnotować fakt, że działają tylko z jedną usługą: *Google Reader*). Potrafią jednak pracować *offline*, dzięki czemu możliwe jest czytanie wiadomości, gdy nie mamy dostępu do Internetu, czyli w samolocie można ,,nadrobić'' zaległości i dowiedzieć się ,,co w świecie piszczy''. Wyjątek stanowi *Mozilla Firefox*, która zbiera tylko tytuły, bez treści, po kliknięciu na pozycję otwierana jest docelowa strona z wiadomością. Dodatkową wadą tych aplikacji jest fakt, iż pomimo tego, że są wieloplatformowe (za wyjątkiem dwóch ostatnich), to nie można używać ich jednocześnie na różnych systemach. Na każdym z nich programy te będą posiadać inne ustawienia, a więc inne pobrane wiadomości. Zawsze istnieje możliwość skopiowania pliku wewnętrznej bazy, bądź przeniesienie całego katalogu profilu, o ile to jest możliwe i nie zakłóci

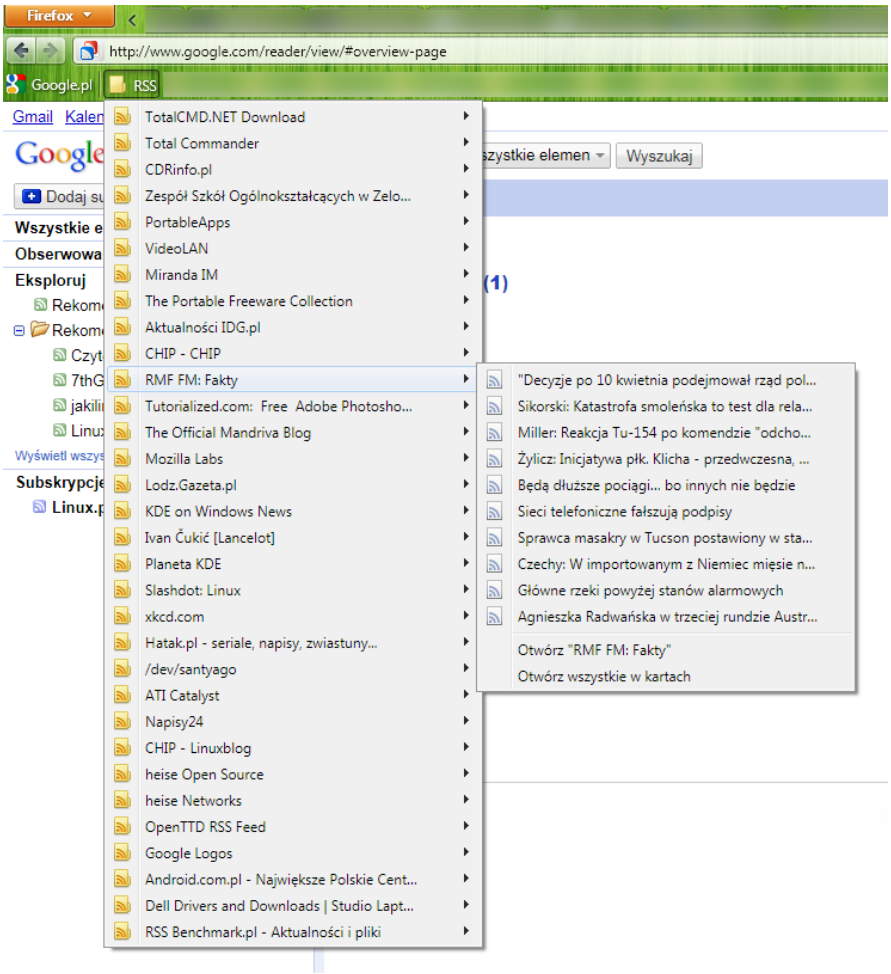


Rys. 1.2: Okno z wiadomościami programu *FeedDemon*

pracy aplikacji. Nie są to jednak czynności proste dla zwykłych użytkowników, zatem pozostaje im na każdym komputerze/systemie mieć zawsze nieco inny stan tych klientów *RSS* i pamiętać, co już zostało przeczytane, a co nie (część wiadomości na pewno będzie się powtarzać).

1.2.2 Aplikacje internetowe

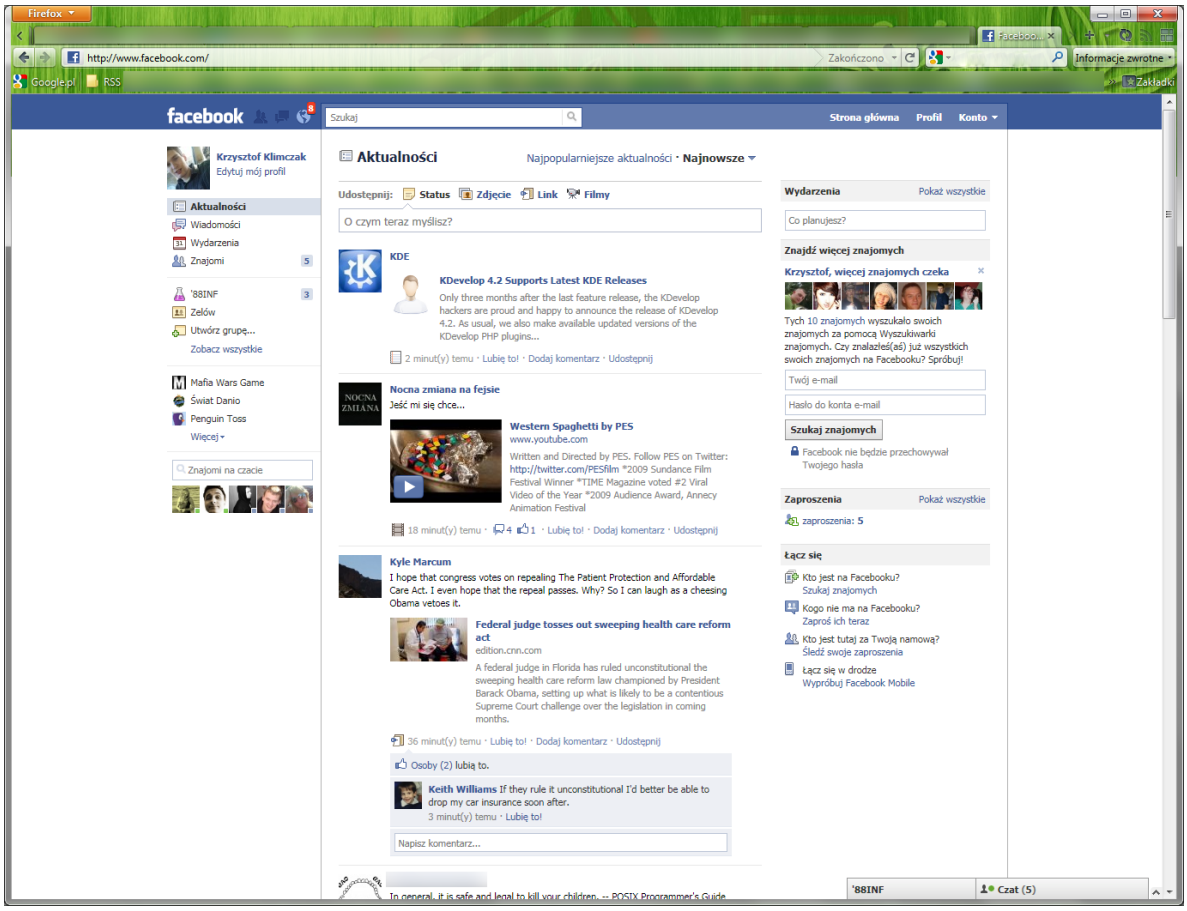
Drugą kategorią programów, dzięki którym można się cieszyć aktualnymi wiadomościami, są aplikacje działające w chmurze, czyli na zdalnym serwerze. Rozwiązanie takie ma wiele zalet, począwszy od faktu, iż nawet, jeżeli nie będzie się miało dostępu do Internetu, to na koncie zostaną umieszczone wszystkie opublikowane wiadomości w danym czasie. Serwisy takie działają zazwyczaj bez przerwy, a ewentualne awarie zdarzają się niezmiernie rzadko i są zazwyczaj dość szybko usuwane. Kolejną zaletą jest fakt dostępu z dowolnego miejsca na Ziemi do wiadomości. Wystarczy uruchomić komputer, podłączyć się do Internetu, otworzyć właściwą stronę i można odczytać



Rys. 1.3: Dynamiczne zakładki w programie *Firefox*

informacje. Zaletą jest tutaj też to, że zawsze i wszędzie posiada się taki sam stan wiadomości (status przeczytania, dodatkowe etykiety, usunięcie z listy), czy to komputer na uczelni, u kolegi, rodziny, na drugim końcu świata. Wszelkie poczynione zmiany są widoczne automatycznie, niezależnie od miejsca. Jedyna trudność to znalezienie przeglądarki internetowej i podanie adresu strony.

Wygląda to na rozwiązanie idealne, ale posiada pewną poważną wadę - wymaga stałego połączenia z Internetem. Bez niego nie da się otworzyć strony czytnika w sieci i przeglądać wiadomości. Co z tego, że będą się one pobierać i czekać aż użytkownik się zaloguje, jeśli nie może tego zrobić. Nie ma on przy tym możliwości (zazwyczaj) pobrania zawartości serwisu na dysk. Do najpopularniejszych stron/czytników można tutaj zaliczyć:



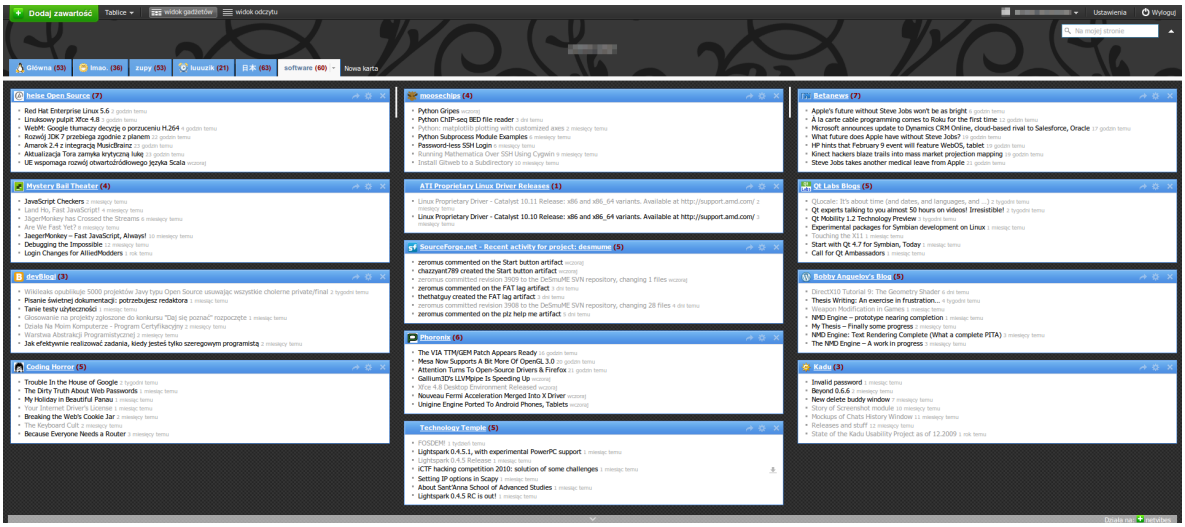
Rys. 1.4: Tablica na portalu *Facebook*

- *Google Reader* (polska nazwa: *Czytnik Google*)[1],
- *NetVibes*[2],
- *Facebook*[3].

Każdy z nich jest darmowy, wystarczy założyć konto i można przystąpić do korzystania, ale już funkcjonalność ich jest diametralnie różna, przez co i sposób użycia.

Facebook (Rys. 1.4) nie jest typowym czytnikiem *RSS*, jest raczej czymś w rodzaju *ściany/tablicy*, na której inni mogą umieszczać notki z informacjami/odnośnikami do swoich portali, zawierających wiadomości [12]. Tablica ta znajduje się centralnie na środku strony i zajmuje najwięcej miejsca. Dzięki takiemu umiejscowieniu jest pierwszym elementem, na który użytkownik zwraca uwagę. Pozostałe elementy nie rozpraszają

[1] `http://www.google.com/reader/`
[2] `http://www.netvibes.com/pl`
[3] `http://www.facebook.com`

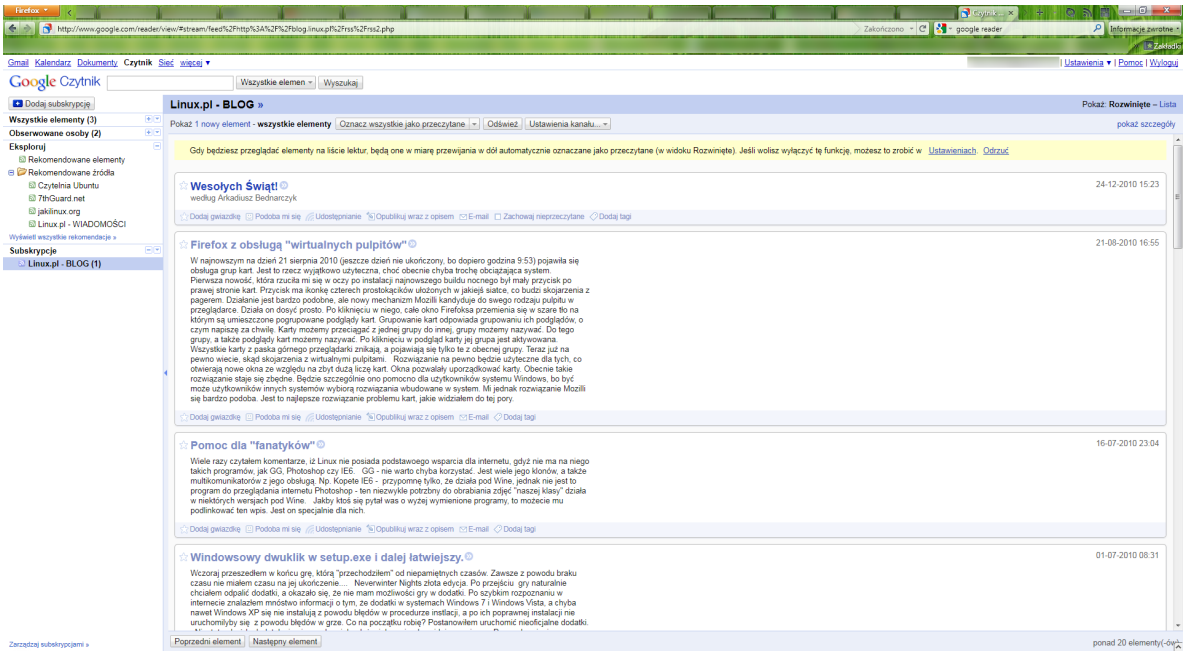


Rys. 1.5: Tablica na portalu *Netvibes*

uwagi użytkownika. Duża popularność serwisu sprawiła, że dzięki publikacji **API** (ang. *Application Programming Interface* - interfejs programowania aplikacji) powstało wiele aplikacji i dodatków do już istniejących programów na komputery stacjonarne oraz urządzenia mobilne [13], dzięki którym bez odwiedzania strony można czytać to, co ktoś umieścił na naszej tablicy. Wymaga to jednak połączenia z Internetem. Dodatkową zaletą jest też to, że tablica zawiera nie tylko wiadomości/notki z różnych serwisów, ale też informacje o poczynaniach naszych znajomych na portalu oraz o zbliżających się rocznicach. A zatem czytając wiadomości ze świata nie przeoczymy faktu, że nasz znajomy na przykład obchodzi urodziny lub też może wydarzyło się w jego życiu coś, czym chciałby podzielić się z innymi.

Druga pozycja na liście to **NetVibes** (Rys. 1.5). Podobnie, jak na portalu *Facebook*, można się tutaj również spotkać z tablicą, lecz jej lepszym określeniem byłoby słowo *,,ściana''*, do której są przypięte kanały użytkownika. Tworzy to swoiste centrum dowodzenia i wspaniale sprawuje się jako strona startowa przeglądarki, dzięki czemu można zacząć dzień od wiadomości z kraju i ze świata. Nie trzeba się przy tym martwić o miejsce, wszystkie kanały można pogrupować w zakładkach dostępnych na tablicy, dzieląc kanały tematycznie, bądź też zależnie od miejsca używania portalu (osobne zakładki do pracy, domu, uczelni).

Serwis udostępnia również swoje własne *API*, dzięki któremu można tworzyć *widżety* (ang. *widget*), czyli funkcjonalne gadżety do innych stron lub systemów operacyjnych. Można na przykład umieścić taki gadżet na pulpicie systemu Windows Vista lub Windows



Rys. 1.6: *Google Reader* - widok kanału

7 i mieć pod ręką bez otwierania przeglądarki listę opublikowanych wiadomości, bądź dodać go do własnej strony internetowej [14]. Niemniej wymagają one również stałego dostępu do Sieci, gdyż muszą się jakoś kontaktować z serwerami *NetVibes*, a do tego wymagają Internetu, by przeczytać pełną treść wiadomości. Zatem jest to rozwiązanie, które pomaga rozpocząć i zakończyć dzień odpowiednią ilością wiadomości.

Na koniec rozwiązanie najbardziej przypominające aplikacje stacjonarne, czyli *Google Reader* (Rys. 1.6). Interfejs tego serwisu został tak dostosowany, by użytkownikom było łatwo rezygnować z dotychczasowych rozwiązań i przenosić się do usługi koncernu z Mountain View. Dzięki pracy inżynierów Google, przez pewien czas istniała nawet możliwość pracy bez połączenia z siecią, dzięki dodatkowi do przeglądarek internetowych *Google Gears*[4]. Rozwiązanie to nie jest już jednak rozwijane i zostało anulowane, koncern zapowiedział, że wraz z nadejściem standardu *HTML5* funkcjonalność ta powróci. Standard ten ma umożliwiać przechowywanie u klienta dowolnej ilości danych, w przeciwieństwie do obecnego rozwiązania, gdzie możliwe jest przechowywanie danych tylko w postaci ,,ciasteczek'' w przeglądarce, które to mają ograniczoną wielkość oraz ilość dla danej domeny.

Wadą tego serwisu jest jednak brak oficjalnego *API*. Jedyne dostępne *API* zostało stworzone przy pomocy inżynierii wstecznej, poprzez analizę zapytań **AJAX** [11].

[4] `http://gears.google.com/`

Zatem programy stacjonarne mają tylko ograniczoną możliwość komunikacji z usługą *Google Reader*. Koncern wydał niedawno aplikację na ich flagowy system operacyjny na urządzenia mobilne *Android*. Kilka dni po premierze najnowszej wersji 2.3 (kodowa nazwa *Gingerbread*), ukazał się program na ten system, który umożliwia dostęp do serwisu, synchronizację z nim, a nawet pracę w trybie *offline*. Jednakże, aplikacja ta jest dostępna tylko i wyłącznie na smartfony z systemem *Android*.

1.3 Podsumowanie

Aktualnie nie ma na rynku oprogramowania, które byłoby ,,idealne''. Najbardziej zbliżone tego określenia są aplikacje internetowe, jednak brak odpowiedniego klienta stacjonarnego, a więc pracy bez połączenia z Siecią, uniemożliwia korzystanie z nich w wielu sytuacjach, na przykład w trakcie podróży samolotem, samochodem lub też pociągiem. Awaria dostawcy usług internetowych również przekreśla korzystanie z aplikacji tego typu, a co za tym idzie skutkuje to brakiem dostępu do wiadomości.

Nie lepiej jest z aplikacjami stacjonarnymi - owszem można czytać wiadomości, gdy nie ma połączenia z Internetem, ale wtedy nie pobierze się nowych wiadomości, które mogą przepaść. Do całości dochodzi jeszcze fakt, że taka aplikacja jest bardzo związana z konkretnym komputerem oraz systemem operacyjnym - o ile problemem nie są netbooki i laptopy, to już komputer stacjonarny ciężko ze sobą zabrać na przykład na uczelnię lub do pracy. Zatem, aby zachować ciągłość wiadomości nasz komputer musi być ciągle włączony i podpięty do Sieci. Do tego kłopotliwe jest używanie go w podróży, nawet w laptopie lub netbooku kiedyś kończy się bateria.

Na chwilę obecną nie ma rozwiązania, które zapewniałoby dostęp do wiadomości kiedykolwiek, gdziekolwiek, niezależnie od posiadanego urządzenia, systemu operacyjnego lub też aplikacji. Potrzebny jest stały dostęp do Sieci, ale i to nie daje gwarancji przeczytania wszystkich interesujących użytkownika wiadomości. Jedynym światełkiem w tunelu jest usługa *Google Reader* w połączeniu ze smartfonem, wyposażonym w system operacyjny *Android*. W takim zestawie w podróży można przeglądać wiadomości na urządzeniu, a po znalezieniu się w zasięgu jakieś sieci, zawartość urządzenia zostanie zsynchronizowana z serwerem. Na komputerze stacjonarnym, jeśli tylko istnieje dostęp do Internetu, można zalogować się na odpowiednią stronę i tam przeglądać subskrybowane kanały. Bez spełnienia tych warunków, dostaje się jednak tylko serwis, który pobiera wiadomości, a dostęp jest ograniczony tylko do kontaktu poprzez przeglądarkę

internetową. Oznacza to, że nie wyróżnia się on niczym nadzwyczajnym, a brak oficjalnego *API* stawia go w nie najlepszej pozycji do walki o użytkowników, a tym bardziej o programistów.

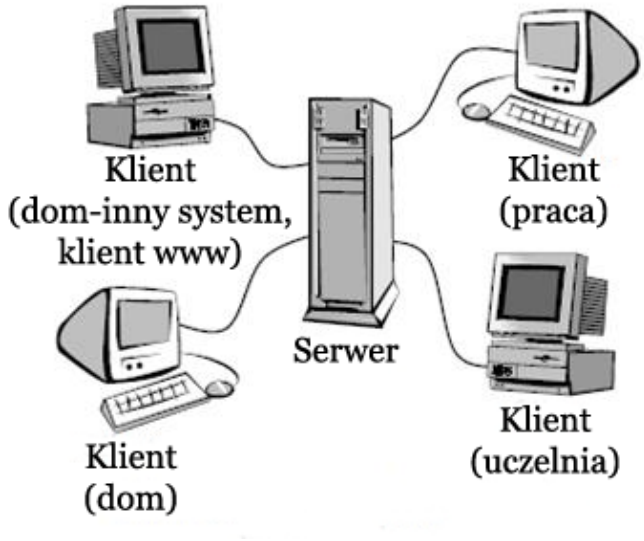
Rozdział 2

Proponowane rozwiązanie

W poniższym rozdziale omówiono rozwiązania obecnych problemów, związanych z zarządzaniem wiadomościami, udostępnionymi przez kanały *RSS*. W tym celu postanowiono połączyć zalety aplikacji internetowych oraz aplikacji stacjonarnych, które razem tworzą spójną całość, uzupełniając się. Nikt do tej pory nie stworzył takiego rozwiązania, które byłoby darmowe i łatwo dostępne. Umożliwiałoby przy tym współpracę własnego oprogramowania z oprogramowaniem stworzonym przez niezależnych twórców.

2.1 Podstawowy problem

Głównym problemem nie jest brak odpowiednich aplikacji, ale dostęp do kanałów. Jak wcześniej zaznaczono, istnieje wiele aplikacji, które spełniają zadowalająco swoje zadanie w znacznej ilości przypadków. To, co wydaje się trywialne, wręcz podstawowe, to największy kłopot. Nie każdy 24 godziny na dobę, 7 dni w tygodniu posiada stały dostęp do Internetu. Zatem to, co trzeba rozwiązać jako pierwsze, to sprawa aktualizacji kanałów tak, by każdy klient posiadał ten sam stan. Na podstawie poprzedniego rozdziału oczywistym wydaje się, że potrzebne jest jakieś rozwiązanie zdalne (pracujące poza komputerami użytkowników), które pobierałoby, aktualizowało kanały użytkownika, a następnie umożliwiałoby ich łatwą synchronizację. Rozwiązanie takie nie może być jednak zwykłą aplikacją internetową, jak na przykład usługa od Google, gdyż dublowałoby już istniejące rozwiązania, co mijałoby się z celem pracy. Nie potrzeba bowiem na rynku dwóch identycznych rozwiązań, szczególnie jeśli pierwsze ma za sobą tak wielką korporację, jaką jest Google. Pozostając dalej przy koncepcji aplikacji internetowej, nasuwa się standard *HTML5* i jego przechowywanie danych. To mógłby być przysłowiowy ,,strzał



Rys. 2.1: Architektura klient-serwer

w dziesiątkę", jednak sam standard nie jest jeszcze ukończony i nie wiadomo, kiedy to nastąpi. Sam mechanizm gromadzenia danych zmienia się jeszcze, prace wciąż trwają, przez co nie ma na rynku przeglądarki internetowej, która oferowałaby pożądaną funkcjonalność. Aktualnie tylko wersje testowe popularnych przeglądarek oferują, ale też tylko testowo i w sposób ograniczony, dostęp do lokalnej bazy danych *IndexedDB*[1]. Nie ma jednak pewności, że w przyszłości nic się nie zmieni na tym polu, prace standaryzacyjne nadal bowiem trwają. Skoro nie można stworzyć czystej aplikacji internetowej, powinno się więc powołać odpowiednią aplikację serwerową, która umożliwiałaby stworzenie odpowiednich aplikacji stacjonarnych i internetowych w oparciu o usługi przez nią oferowane (Rys. 2.1).

2.2 Serwer

Podstawowym zadaniem tej części platformy powinno być pobieranie zawartości kanałów *RSS* oraz zapewnienie łatwego dostępu do tych danych niezależnie od miejsca, z którego użytkownik chce uzyskać dostęp do swoich danych oraz aplikacji, jakiej będzie używać w tym celu. W związku z tym kluczowe w tej części projektu są:

- dostępność na różne platformy,
- łatwa integracja z różnymi SZBD (*System zarządzania bazą danych*, ang. *Database*

[1] `http://www.w3.org/TR/IndexedDB/`

Management System, DBMS),

- przejrzysta i łatwa wymiana danych z klientem,
- nieograniczanie się do zamkniętych standardów,
- otwarcie na rozwiązania innych (współpraca z różnymi klientami),
- bezpieczeństwo.

2.2.1 Dostępność na różne platformy

Często można się spotkać z rozwiązaniami, które działają tylko na konkretnej platformie, szczególnie jest to widoczne w produktach firmy Microsoft, gdzie ich aplikacje wymagają do działania tylko ich systemu operacyjnego i często zdarza się, że musi on być w określonej wersji, inaczej dana aplikacja nie będzie działać lub będzie to robić w ograniczony sposób. Zmusza to użytkownika do wydania wielu pieniędzy na nowy sprzęt oraz licencje na oprogramowanie, często też pojawia się potrzeba zatrudnienia nowego pracownika lub wynajęcie takiego, który zna daną platformę i będzie potrafił ją skonfigurować do konkretnych potrzeb. Rozwiązanie to można uznać za niekorzystne, gdyż należy też obsłużyć serwery, na których zainstalowany jest inny system operacyjny.

Rozwiązanie musi być wieloplatformowe, tak by to administrator decydował, na czym najlepiej uruchomić daną usługę (tak, by koszt wdrożenia był najniższy). Zatem aplikacja taka musi być napisana w języku programowania i technologii, które działają na każdym systemie, a kompilacja na daną platformę nie jest problematyczna.

2.2.2 Łatwa integracja z różnymi SZBD

Podobnie, jak z systemem operacyjnym, na którym działa aplikacja serwerowa, nie powinno się ograniczać SZBD, w ramach którego będą przechowywane dane. W przypadku zaimplementowania obsługi tylko jednego konkretnego serwera baz danych, zmusza się klienta końcowego do jego użycia, bądź rezygnacji z platformy. Nie każdy ma zasoby, miejsce oraz środki, by utrzymywać wiele różnych rozwiązań.

Trzeba więc wziąć pod uwagę łatwą metodę rozszerzenia tworzonej aplikacji, na przykład poprzez mechanizm wtyczek, które zapewniałyby obsługę różnych SZBD. Takie rozwiązanie jest idealne w połączeniu z publikacją odpowiedniego *API* wraz z opisem i przykładową implementacją. Dowolny programista będzie mógł stworzyć wtyczkę,

dzięki której stworzony w ramach pracy program będzie działać z dowolną bazą danych, którą posiada programista. Takie posunięcie znacząco powinno podnieść popularność całej platformy.

2.2.3 Przejrzysta i łatwa wymiana danych z klientem

Na przykładzie usługi *Google Reader* zaprezentowano, jaka może być wada nieopublikowania sposobu komunikacji klienta z serwerem. Istniejące rozwiązania są przez to niepełne i zawierają błędy, ich funkcjonalność jest ograniczona. Brak takiego opisu sprawia, że użytkownik platformy jest skazany tylko na dane rozwiązanie lub jego brak.

Należy więc stworzyć przejrzysty system komunikacji klienta z serwerem, określić rodzaj dialogu, jaki będą prowadzić. Uważam za najlepsze rozwiązanie zastosowanie metody komunikacji podobnej do tej użytej między innymi w takich protokołach, jak *POP3*, *IMAP4* lub *HTML*, czyli komunikacji otwartym teksem. Natomiast tam, gdzie wskazana jest komunikacja w trybie binarnym, należy to wyraźnie podkreślić. W ten sposób nie ograniczamy możliwości tworzenia aplikacji klienckich na platformy, których sami nie wspieramy, a co za tym idzie, grono naszych użytkowników powinno wzrosnąć.

2.2.4 Nieograniczanie się do zamkniętych standardów

W przypadku zamkniętych standardów ograniczamy przyszły rozwój naszej platformy nie tylko poprzez ścisłe związanie z danym rozwiązaniem, ale też uzależniamy się od czegoś na co, nie mamy wpływu. W razie złego wyboru może się zdarzyć, że dane rozwiązanie przestanie być rozwijane, bądź na naprawienie zauważonych błędów musimy czekać długimi miesiącami. Ograniczamy też współpracę innych programistów przy projekcie, gdyż często trzeba spełnić warunki licencji, dotyczące ilości stanowisk pracy. Nie możemy zazwyczaj opublikować całego kodu takiej aplikacji, a w przypadku błędnego działania jakiegoś modułu, nie jesteśmy w stanie dokładnie określić, gdzie i co działa wadliwie.

Powinno się więc skorzystać z otwartych rozwiązań, które mają liczną społeczność czuwająca nad danym standardem/rozwiązaniem. Dzięki temu nie musimy obawiać się zaprzestania rozwijania danego standardu. W ostateczności można kontynuować jego rozwój, wnosząc własne rozwiązania, które do tej pory się nie pojawiły i już nie pojawią. Wszelkie niejasności mogą zostać łatwo wyjaśnione, gdyż nie jest tajemnicą, jak dany element zaprojektowano oraz zaimplementowano, przez co można zrozumieć, jak działa i lepiej zaprojektować użycie danego podsystemu. Do całości dochodzi jeszcze fakt, iż

samemu można wspierać taki projekt z obopólną korzyścią. Dodatkowym faktem jest to, iż udostępnienie kodu i korzystanie z rozwiązań, które znają inni programiści, jest zmniejszeni ilości błędów w aplikacji (większa liczba osób testuje i sprawdza program/kod źródłowy).

2.2.5 Otwarcie na rozwiązania innych

Ten punkt jest kwintesencją trzech powyższych, nie powinno ograniczać się tylko do własnych rozwiązań. Niezaprzeczalnym faktem jest to, iż nie sposób samemu napisać uniwersalną aplikację na każde urządzenie. Zatem otwierając się na innych, otwieramy drogę naszej aplikacji na przykład na inne platformy, systemy lub języki.

Nie powinno utrudniać się tworzenia przez innych rozwiązań, które współpracowałyby z naszym. Otwarcie jest drogą do popularności. Przykładowo protokoł *XMPP* (ang. *eXtensible Messaging and Presence Protocol* [dawniej *Jabber*]), to protokół bazujący na języku *XML*, umożliwiający przesyłanie w czasie rzeczywistym wiadomości oraz statusu. Protokół ma zastosowanie nie tylko w komunikatorach, ale również w innych systemach natychmiastowej wymiany informacji [15]. Dzięki jego otwartości, powstała liczna gama oprogramowania i różnego zastosowania. Nie osiągniętoby tego, gdyby specyfikacja protokołu była zamknięta.

2.2.6 Bezpieczeństwo

Należy zapewnić minimum bezpieczeństwa dla przechowywanych danych oraz ich transmisji. Wielu użytkowników używa tych samych haseł w wielu miejscach, do tego to, jakie informacje czytamy (jakie wiadomości nas interesują) może zdradzić wiele o użytkowniku i narazić go na wiele niebezpiecznych ataków. Powinno się zatem zapewnić bezpieczny kanał komunikacji, przy czym powinien być on dobrze znany i spopularyzowany. Dzięki sprawdzonemu rozwiązaniu unikniemy przykrych w konsekwencjach błędów implementacyjnych, a do tego nie utrudnimy innym stworzenia rozwiązań współpracującym z naszym. Należy też zadbać o bezpiecznie przechowanie danych w bazie danych. Nie ma rozwiązań idealnych, a błędy w zabezpieczeniach zawsze się znajdą. Poufne dane powinny być przechowywane w sposób niemożliwy do odczytania przez osoby trzecie, nawet w przypadku, gdyby uzyskali oni bezpośredni dostęp do bazy danych.

Rys. 2.2: Schemat ideowy budowy platformy

2.2.7 Budowa serwera

Biorąc pod uwagę wszystkie wyżej wymienione punkty, zdecydowano, że platforma w całości powstanie w języku C++ przy użyciu biblioteki **Qt** firmy *Nokia* [1]. Jest ona wydawana na dwóch licencjach: komercyjnej oraz otwartej (LGPL), nie ma problemu z dostępem do dokumentacji [2] i kodów źródłowych. Wokół tej biblioteki zgromadzona jest ogromna społeczność, która pomoże i wytłumaczy niejasności. Zaletą *Qt* jest jego wieloplatformowość, jeden kod powinien bez większych komplikacji budować się na różnych platformach i systemach. Nie ograniczamy się tutaj do jednego typu procesora (x86, PowerPC, Sparc) lub do systemu operacyjnego (Windows, Linux, MacOS). Stworzona aplikacja, po pewnych dostosowaniach powinna także bez problemu działać na urządzeniach mobilnych wyposażonych w system operacyjny *Symbian* lub *MeeGo* (które są oficjalnie wspieranie). Powstał też odpowiedni port na system *Android* (jest to jednak rozwiązanie stworzone przez niezależnego programistę). A wszystko to dzięki otwartości tej biblioteki.

Różne serwery baz danych posiadają swoje zalety i wady, nadając się do określonych zastosowań lub środowisk. Nie można więc wymuszać konkretnego rozwiązania (Rys. 2.2). Idąc tym tokiem rozumowania, przyjęto, iż podstawowa/przykładowa implementacja wtyczki komunikacji z bazą danych będzie obsługiwać serwer **PostgreSQL**. Jest to największa otwarto-źródłowa baza danych, zapewniająca dobrą szybkość i zgodność z językiem zapytań SQL. Dodatkowo, do zapisywania samych ustawień aplikacji

zastosowana będzie prosta i lokalna baza danych **SQLite3**. Uprości ona zarządzanie ustawieniami, ograniczy zużycie pamięci oraz zmniejszy też ilość odwołań do dysku w porównaniu z rozwianiami w postaci plików XML, INI lub globalnym rejestrem systemowym.

Na koniec kwestia bezpieczeństwa. Hasła użytkowników nigdy, nie powinny być przechowywane otwartym tekstem, gdyż w przypadku nieuprawnionego dostępu do bazy danych, może to mieć poważne konsekwencje (wielu użytkowników używa tylko jednego hasła do wszystkich usług). W celu zachowania minimum bezpieczeństwa, powinno się przetrzymywać hasła w postaci ich skrótów, tak samo przesyłać je pomiędzy klientem a serwerem. Przyjęto, że w aktualnej wersji platformy zostanie użyta metoda **SHA1**, ale w przypadku późniejszej implementacji należy rozważyć wdrożenie nowszych metod, na przykład *SHA256* lub *SHA512*. Do połączenia zostanie użyty protokół **SSL/TLS**, w celu zabezpieczenia dwukierunkowej komunikacji klient-serwer, co w znacznym stopniu ograniczy możliwość podsłuchania transmisji.

Ostatnim elementem jest ilość danych, jakie będą przesyłane i gromadzone. Należy zadbać o to, by rozmiar bazy danych nie zwiększał się w zastraszającym tempie. Baza powinna być znormalizowana. Dodatkowo z uwagi na fakt, iż wiadomości *RSS* są przesyłane w postaci czystego tekstu, zastosowano kompresję głównej części wiadomości, używając przy tym otwartego i dobrze opisanego formatu **zlib** [10].

2.3 Klient

Klientowi tak, jak serwerowi przyświecają te same idee, a więc: otwartość źródeł, wieloplatformowość, przyjazność dla użytkownika/programisty. Z tego względu poniżej wymieniono tylko różnice pomiędzy serwerem a klientem.

2.3.1 Klient zawsze taki sam

Dzięki zastosowaniu już wcześniej wspomnianej biblioteki **Qt**, stworzona aplikacja kliencka działa bez problemu na systemach Windows, Linux, MacOS (te zostały przetestowane i są wspierane). Wystarczy wziąć ten sam kod i skompilować go pod daną platformę. Jedyny wymóg, jaki musi być spełniony, to kompilacja przy pomocy **GCC** (minimum wersja 4.4, starsze nie były sprawdzane). Inne kompilatory nie były testowane i nie są wspierane, może się więc zdarzyć sytuacja, kiedy jakieś wyrażenie użyte w kodzie jest w pełni poprawne pod *GCC*, zaś pod kompilatorem firmy Intel lub

Microsoft, kod nie skompiluje się, zgłaszając przy tym liczne błędy.

Podobnie, jak serwer, klient również może korzystać z różnych SZBD. Domyślnym serwerem bazy danych dla ustawień oraz dla lokalnie przechowywanych danych jest **SQLite3** [5]. Baza ta nadaje się idealnie do tego typu zastosowań, gdyż jest w postaci jednego pliku, zapewnia szybkość i łatwość zarządzania danych oraz nie wymaga wstępnej konfiguracji ani jakiegokolwiek serwera. Jest jedną z najbardziej rozpowszechnionych baz danych, a do tego jest dostępna za darmo wraz z kodami źródłowymi. Nie stoi jednak nic na przeszkodzie, by zastosować inny serwer baz danych. Podobnie, jak w przypadku serwera, wystarczy napisać i dołączyć odpowiednią wtyczkę, umożliwiającą kontakt aplikacji z bazą.

Dzięki takiemu rozwiązaniu, użytkownik pracując na różnych systemach, używa zawsze tego samego interfejsu użytkownika (oraz nieświadomie tych samych komponentów wewnętrznych aplikacji), do którego jest przyzwyczajony i z którym nie ma problemów. Aplikacja ponadto bezproblemowo integruje się ze środowiskiem graficznym użytkownika, czy to Windows, MacOS, czy Linux z KDE/Gnome, wyglądając na natywną dla danego systemu (czego nie można powiedzieć na przykład o aplikacjach napisanych w języku Java).

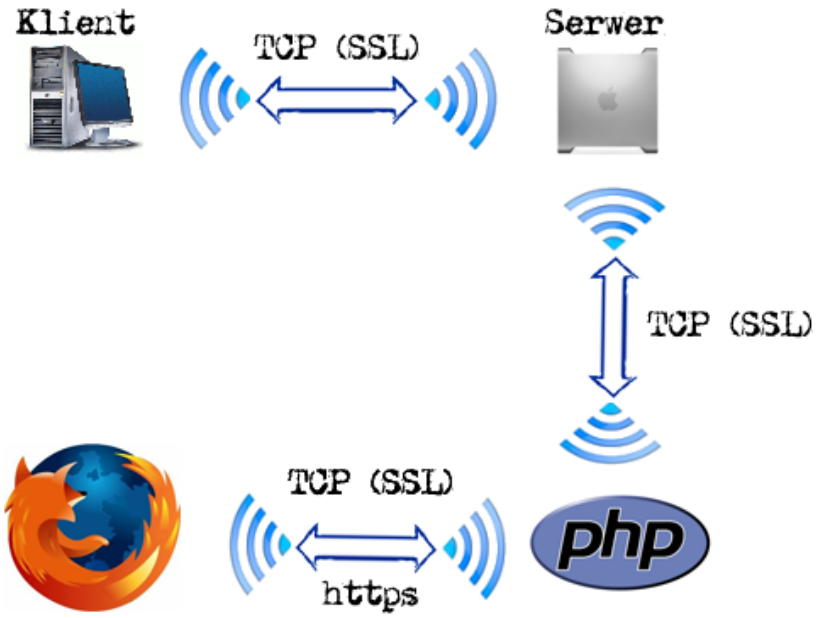
2.3.2 Klient uniwersalny?

Czy stworzony klient jest uniwersalny? Odpowiedź na to pytanie brzmi oczywiście: nie. Jak każde rozwiązanie, tak i *Qt* posiada błędy[2], a do tego dochodzi fakt, że kod został dostosowany do komputerów stacjonarnych, a nie do urządzeń mobilnych. Nie można też zapominać, że każda platforma rządzi się swoimi prawami, przez co nie jesteśmy w stanie uwzględnić i rozwiązać wszystkich możliwości/problemów.

Dlatego też dzięki otwartej budowie, każdy może zapoznać się ze sposobem wymiany danych, z budową stworzonej aplikacji, po czym stworzyć swoje własne rozwiązanie. Nie jest przy tym blokowana żadna platforma, można bez problemu napisać aplikację w języku Objective-C na urządzenia z systemem iOS od Apple (iPhone, iPad) lub w języku Java (tworząc aplikacje na system *Android*), bądź w C# i platformie .NET.

Użyte w projekcie technologie nie ograniczają twórców również pod względem tworzenia aplikacji internetowych (Rys. 2.3). Czy użyjemy *PHP, Pythona, ASP, JSP*, czy innej technologii, będziemy w stanie podłączyć się do stworzonego serwera i bez

[2] `http://bugreports.qt.nokia.com` adres gdzie można śledzić aktualnie zgłoszone błędy i status ich naprawiania



Rys. 2.3: Ogólny schemat budowy platformy

problemu wymieniać z nim dane. Każde z nich, dzięki użyciu otwartych standardów, potrafi nawiązać szyfrowane połączenie SSL z określonym hostem na określonym porcie i wymieniać z nim dane. Nie ma też problemu z rozpakowanie danych w formacie *zlib*.

2.4 Czym jest zatem platforma

Zaproponowane rozwiązanie łączy cechy aplikacji internetowych i stacjonarnych oraz eliminuje ich większość niedogodności. Dzięki skorzystaniu z biblioteki **Qt** zyskuje projekt, który działa na wielu systemach, a do tego jest uniwersalny i można go w prosty sposób łączyć z innymi technologiami.

Sercem platformy jest oczywiści serwer, który zapewnia bezproblemową komunikację pomiędzy różnymi klientami, ważne tylko by używały tych samych poleceń/języka do komunikacji. Nie ma ograniczeń, dotyczących tego w czym mają być stworzone ani na czym działać (nie będzie to problemem, jeśli ktoś napisze na przykład aplikację w *Asemblerze* i uruchomi ją na lodówce).

Zastosowanie otwartych standardów, zapewnia łatwość budowy każdego fragmentu platformy i nie ogranicza innych programistów (jak zastosowanie *zlib* do kompresji lub *SSL/TLS* do szyfrowania komunikacji, albo *SHA1* do wyznaczania skrótu hasła). Mamy gwarancję, że zastosowane w projekcie technologie będą posiadać wsparcie. Oznacza to,

że platforma nie zniknie nagle z cyfrowego świata, pozostawiając jej użytkowników bez rozwiązania zastępczego.

Zatem, co oznacza słowo **„Platforma”** w tej pracy? Oznacza kompleksowe rozwiązania, niezależne od używanego systemu, platformy lub języka. Stworzona *Platforma* to zbiór rozwiązań łatwych do implementacji poprzez wszechobecne dokumentacje i kody źródłowe do wszystkich elementów, użytych w projekcie. Pozwala łączyć różne rozwiązania, a nie dzielić je na lepsze i gorsze.

Rozdział 3

Dokumentacja techniczna

Niniejszy rozdział zawiera dokumentację techniczną stworzonego projektu. Opisano w nim użyte technologie i rozwiązania oraz napotkane w trakcie realizacji projektu błędy i problemy wraz z metodami ich rozwiązania. Niektóre moduły występują w podobnej budowie w obu aplikacjach projektu (serwer i klient), więc zostaną opisane tylko raz.

RSS

Dokument *RSS* to zwykły dokument XML przesłany czystym tekstem, zawierającym określone tagi, które to klient rozpoznaje i interpretuje. Minimalny dokument *RSS* zgodnie ze specyfikacją [8] musi zawierać określoną konstrukcję, pokazaną na listingu 3.1.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <rss version="2.0">
3     <channel>
4         <title>Tytuł kanału</title>
5         <description>Opis, co zawiera dokument, jaką tematykę</description>
6         <link>Adres strony, do której się odnosi/</link>
7
8         <item>
9             <title>Tytuł wiadomości</title>
10            <link>Adres do pełnej wiadomości</link>
11            <description>Ciało wiadomości</description>
12        </item>
13    </channel>
14 </rss>
```

Listing 3.1: Minimalny dokument RSS

W tagach <*item*> zawarte są wiadomości, nie ma ograniczeń, dotyczących ich ilości. Warto też wspomnieć o nieobowiązkowych tagach, takich jak <*guid*> oraz

<*pubDate*>. Pierwszy określa unikalny ciąg znaków/napis, jednoznacznie identyfikujący wiadomość (często jest tutaj ta sama zawartość, co w tagu <*link*>). Drugi z opcjonalnych tagów określa datę publikacja i/lub ukazania się wiadomości na portalu (pomimo jego nieobowiązkowości, stał się wręcz stałym elementem wszystkich kanałów).

Przetwarzanie dokumentów RSS

Gdy opisany już został wygląd dokumentu *RSS* (obecna wersja rozpoznaje tylko informacje umieszczone w tagach obowiązkowych oraz tagu <*pubDate*>), można przystąpić do jego przetwarzania. W bibliotece *Qt* istnieją trzy możliwości wykonania tego zadania: formaty *SAX*, *DOM* (niezalecany, najwolniejszy, ma być wycofany w przyszłości) oraz *QXmlStreamReader*. W pracy użyto tego ostatniego, gdyż z dostępnej trójki jest najszybszy, do tego jest zalecany przez programistów *Qt*.

Jak sama nazwa wskazuje, format *QXmlStreamReader* przetwarza dokument *XML*, traktując go jako strumień danych, to znaczy nie robi jego obrazu w pamięci, tak jak *DOM*, lecz przetwarza na bieżąco każdy węzeł. Jest to metoda podobna do *SAX*, lecz z tą różnicą, że nie określa się zachowania dla każdego typu węzła, jaki występuje w dokumentach *XML*. *QXmlStreamReader* przetwarzając dokument, sprawdza rodzaj węzła, pozwala na odczytanie jego nazwy i dopiero na tej podstawie można wykonać inną czynność (Listing 3.2 i 3.3). Rozwiązanie to jest znacznie szybsze od technologii *SAX*, gdyż nie trzeba za każdym razem sprawdzać wszystkich możliwości dla danego typu węzła.

```
1  void XMLParser::parseData(QByteArray data){
2      QXmlStreamReader xml( data );
3
4      while( !xml.atEnd() && !xml.hasError() ){
5          xml.readNext();
6
7          if( xml.isStartDocument() ){
8              continue;
9          }
10
11         if( xml.isStartElement() &&
12            xml.name() == QString::fromUtf8("item") )
13         {
14             parseItem(xml);
15         }
16     }
17
18     if( xml.hasError() && xml.error()
19         != QXmlStreamReader::PrematureEndOfDocumentError )
```

```
20      {
21          qWarning() << "XML ERROR =>" << xml.lineNumber() <<
22                  ": " << xml.errorString();
23      }
24  }
```

Listing 3.2: Przykładowy kod przetwarzający dokument XML

```
1  void XMLParser::parseItem(QXmlStreamReader &reader){
2      reader.readNext();//omijamy tag otwierający "item"
3
4      while( !(reader.isEndElement() &&
5              reader.name() == QString::fromUtf8("item")) )
6      {
7          if( reader.isStartElement() ){
8              if( reader.name() == QString::fromUtf8("title") ){
9                  qDebug() << "title:" << reader.readElementText();
10             }
11             if( reader.name() == QString::fromUtf8("description") ){
12                 qDebug() << "description:" << reader.readElementText();
13             }
14             if( reader.name() == QString::fromUtf8("link") ){
15                 qDebug() << "link:" << reader.readElementText();
16             }
17         }
18         reader.readNext();
19     }
20 }
```

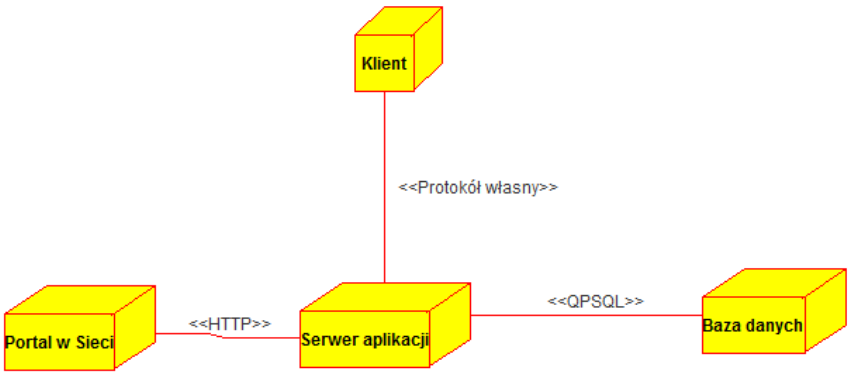
Listing 3.3: Przykładowy kod przetwarzający wiadomość w dokumencie RSS

3.1 Serwer

Wymagania

Projektując aplikację serwera, należało w szczególności uwzględnić potrzebę działania niezależną od środowiska, w którym aplikacja będzie pracować. Przyjęto, więc następujące założenia dotyczące tej części projektu:

1. aplikacja działa na różnych platformach,
2. sposób komunikacji z klientem jest ściśle określony,
3. wymiana danych realizowana jest zawsze w ten sam sposób,
4. istnieją zawsze takie same typy danych, niezależnie od użytej bazy danych,



Rys. 3.1: Diagram zewnętrznych połączeń serwera

5. możliwe jest użycie różnych SZBD z aplikacją,
6. komunikacją z bazą danych musi być bezpieczna,
7. zarządzanie danymi w bazie danych powinno być oszczędne pamięciowo,
8. aplikacja łączy się z portalami w Sieci,
9. kanały RSS są pobierane z Sieci, parsowane i składowane w bazie,
10. zapisywane są tylko te wiadomości, które są nowsze od ostatnio zapisanej (kluczem jest data publikacji),
11. aplikacja nie wymaga głównego okna.

Schemat połączeń, jakie muszą być zrealizowane w tej części projektu został przedstawiony na rysunku 3.1.

Implementacja

Całość implementacji projektu zrealizowana została w języku *C++* przy pomocy biblioteki *Qt* firmy Nokia w wersji 4.7.0 (co jest ważne w kwestii obsługi baz danych), za IDE posłużył *Qt Creator 2.0.1*.

Aplikacja do działania wymaga co najmniej jednej wtyczki, implementującej interfejs bazy danych. Sama implementacja została zrealizowana w oparciu o mechanizmy dostarczone przez bibliotekę *Qt*, a mianowicie o *QtPlugin* [18] oraz *QPluginLoader* [19]. Są to zbiory makr służące do rejestracji wtyczki oraz obiekt, który pozwala tak

eksportowaną wtyczkę umieścić w dowolnym momencie działania aplikacji (Listingi 3.4 i 3.5).

```
1 QT_BEGIN_NAMESPACE
2 Q_DECLARE_INTERFACE(IDatabase,
3                      "pl.com.momus.beaver.IDatabase/0.0.7")
4 QT_END_NAMESPACE
```

Listing 3.4: Deklaracja interfejsu wtyczki baz danych

```
1 foreach( QString fileName, m_pluginsDir.entryList( QDir::Files ) ){
2     QPluginLoader loader( m_pluginsDir.absoluteFilePath( fileName ) );
3     QObject *plugin = loader.instance();
4     if( plugin ){
5         //do something with plugin
6     }
7 }
```

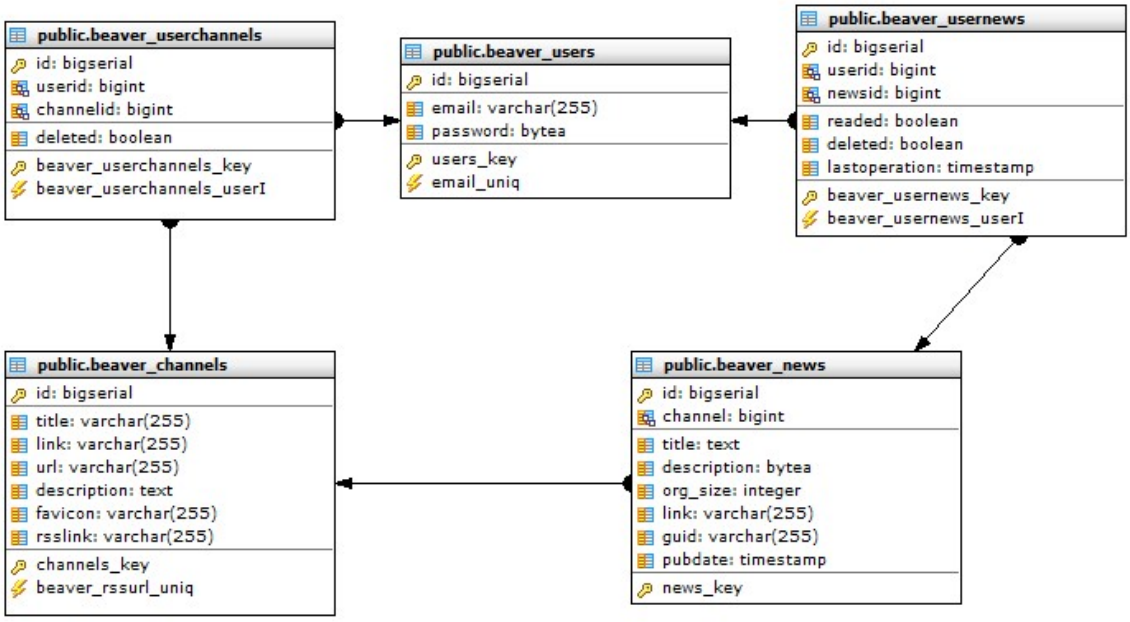
Listing 3.5: Przykład ,,ładowania'' wtyczek

Warto wspomnieć, że rozwiązanie to ułatwia jedynie ładowanie rozszerzeń do aplikacji, używając wewnętrznych poleceń danego systemu. Nie ujednolica jednak (choć lepszym określeniem jest tutaj ukrywa) typowych dla danego systemu rozwiązań, dotyczących ładowania dynamicznych bibliotek. Jako przykład takiego zachowania można podać pewien fakt związany z zarządzaniem pamięcią dla bibliotek. Jak wiadomo, wtyczka po ,,załadowaniu'' dzieli przestrzeń adresową aplikacji, jednak w przypadku systemu Windows po ,,załadowaniu'' biblioteki, ta dostaje własną przestrzeń adresową na zmienne stałe i statyczne. Fakt ten nie jest opisany w dokumentacji *Qt* i w pewnych okolicznościach może stanowić poważny problem, na przykład kiedy w aplikacji istnieje jakiś obiekt *Singletonu*, dla każdej instancji biblioteki załadowanej do aplikacji, zostanie utworzony nowy obiekt. Jednym ze sposobów uniknięcia tej sytuacji jest przekazać adres *Singletonu* z aplikacji do biblioteki tuż po jej załadowaniu.

Oprócz własnych wtyczek, w aplikacji użyto też wtyczek dostarczonych wraz z *Qt*. Są nimi na przykład wtyczki do obsługi plików graficznych *svg*, *png* lub *jpg*. Nie można też zapomnieć o wtyczkach służących do obsługi baz danych.

3.1.1 PostgreSQL

Jako domyślną bazę danych dla serwera użyto *PostgreSQL 9.0*. Jest to najbardziej zaawansowany, otwartoźródłowy SZBD na świecie. Dodatkowym faktem przemawiającym za tym wyborem, jest wsparcie jej przez *Qt*, które jest dostarczone wraz z kodami źródło-



Rys. 3.2: Diagram bazy danych po stronie serwera

wymi sterownika (a niektóre systemy/dystrybucje oferują już skompilowany sterownik w oparciu o aktualną wersję SZBD i *Qt* w repozytoriach).

Rysunek 3.2 przedstawia diagram tabel dla bazy danych, działającej po stronie serwera, uwzględniający relacje (wszystkie są typu 1-do-wielu), kolumny, klucze główne oraz indeksy, które są używane, by przechowywać dane użytkowników po stronie serwera.

Sterownik bazy PostgreSQL

Qt do komunikacji z różnymi bazami danych wymaga odpowiednich sterowników, które są rozpowszechniane jako wtyczki do aplikacji/biblioteki. Bez takiego sterownika nie ma możliwości bezpośredniego odwoływania się do bazy danych. Biblioteka *Qt* jest domyślnie dostarczona ze zbudowanym sterownikiem do bazy *SQLite3*, zaś do takich baz jak *MySQL, PostgreSQL* jest udostępniany tylko kod źródłowy. W przypadku systemu Linux instalacja odbywa się dość prosto - wystarczy korzystając z ulubionego dowolnego pakietów, pobrać odpowiednią paczkę ze sterownikiem, zainstalować i można już bez problemu używać w aplikacjach napisanych w oparciu o *Qt*. W przypadku posiadania MacOS wystarczy zbudować odpowiedni sterownik ze źródeł przy pomocy *qmake*, *make* i *gcc*.

Na platformie Windows sprawa nie jest tak oczywista i prosta. System ten nie posiada żadnego scentralizowanego repozytorium z oprogramowaniem, dodatkowo w celu za-

pewnienia maksymalnej zgodności aplikacje często są budowane za pomocą kompilatora *MSVC*, stworzonego przez Microsoft (który jest na przykład dołączany do środowiska programistycznego *Visual Studio*). Trzeba zatem albo przeszukać Internet w poszukiwaniu odpowiedniego sterownika (już zbudowanego) lub po prostu skompilować go samemu z dostarczonych źródeł. Należy jednak pamiętać, że sam *PostgreSQL* na Windows został zbudowany za pomocą kompilatora firmy Microsoft, przez co, jeśli chcemy zbudować sterownik za pomocą *MinGW*, należy odpowiednio przygotować pliki wedle poniższej instrukcji [16] (jeśli budujemy sterownik pod *Visual Studio*, te zabiegi nie są konieczne, można od razu przystąpić do kompilacji):

- otwórz ,,*Qt Command Prompt*'' i przejdź do katalogu, gdzie zainstalowany jest *PostgreSQL* (na przykład *C:\Program Files (x86)\PostgreSQL\9.0*),
- wejdź do katalogu ,,lib'' i wykonaj polecenie `reimp libpq.lib`[1], by stworzyć pliki: *liblibpq.a* oraz *libpq.def*,
- w pliku *libpq.def* usuń wszystkie znaki ,,_'' z początków definicji,
- `dlltool --input-def libpq.def --output-lib libpq.a --dllname libpq.dll`.

Stworzona została biblioteka importu, którą można użyć z *MinGW*:

- przejdź do `%QTDIR%/src/plugins/sqldrivers/psql`,
- wykonaj następujące polecenie: `qmake -o Makefile "INCLUDEPATH+=C:\Program Files (x86)\PostgreSQL\9.0\include" "LIBS+=C:\Program Files (x86)\PostgreSQL\9.0\lib\libpq.a" psql.pro`,
- uruchom *make* (*mingw32-make*) - to powinno zbudować pliki: *qsqlpsql.dll* i *libqsqlpsql.a* i umieścić je w katalogu `%QTDIR%/plugins/sqldrivers`.

Po tych wszystkich operacjach można nawiązać połączenie z bazą *PostgreSQL*. Trzeba jeszcze zadbać o to, by w zmiennej środowiskowej `PATH` umieścić ścieżkę dostępu do bibliotek bazy danych lub skopiować je do katalogu z aplikacją, w przeciwnym wypadku sterownik nie załaduje się.

Niestety używana w projekcie wersja *Qt 4.7.0* wspiera oficjalnie ten SZBD do wersji 8.2 (dopiero *Qt 4.7.2* oferuje wsparcie dla *PostgreSQL 9.0*, nie wiadomo jednak, kiedy

[1] narzędzia `reimp` oraz `dlltool` można pobrać ze strony `http://sourceforge.net/projects/mingw/files/`, jak i również sam kompilator `MinGW` i `make`

się pojawi). Oznacza to, że sterownik pracuje w trybie zgodności (*PostgreSQL 6.x*), przez co nie wszystkie funkcje działają bezproblemowo i czasem trzeba samemu szukać rozwiązania, na przykład w przypadku obsługi danych typu *BLOB/bytea*, gdzie dane należy w obie strony przesyłać przy użyciu kodowania *base64*, zaś przed wysłaniem należy taki ciąg poprzedzić odpowiednim ,,znakiem ucieczki'' (listing 3.6).

```
1 QSqlQuery query(db);
2 //odczyt przesłanych danych do serwera (zapytanie)
3 query.prepare( QString("SELECT id, password FROM %1users "
4                        "WHERE email=? AND password=E?;").arg(m_prefix) );
5 query.addBindValue( login );
6 query.addBindValue( QByteArray::fromHex(password.toLatin1()).toBase64() );
7 //odebranie danych
8 ...
9 QByteArray data = QByteArray::fromBase64( query.value(1).toByteArray() );
```

Listing 3.6: Obsługa typu danych *bytea* w trybie zgodności

3.1.2 Kompresja danych

Nazwy tabel, jak i kolumn oraz typów danych im przypisanych wydają się oczywiste, za wyjątkiem jednej. W tabeli *beaver_news* kolumna *description* jest typu *bytea*. Jest to pole, które służy do przechowywania danych binarnych. Pytanie brzmi: po co dane binarne, przecież *RSS* ma postać zwykłego tekstu. Odpowiedź jest dość prosta. Zdecydowano się na ten zabieg ze względu na oszczędność miejsca. Kolumna ta przechowuje bowiem główną część artykułu, czyli zawartość tagu <*description*>. *Qt* ma w sobie wbudowaną bibliotekę *zlib*, dzięki której można w prosty sposób zmniejszyć ilość przechowywanych danych. Dane po spakowaniu są w formacie *zlib* [10], jednak *Qt* tworzy niestandardowy nagłówek (co nie jest opisane w dokumentacji). Nie ma problemu, jeśli spakuje się dane w aplikacji, opartej o bibliotekę Nokii oraz rozpakujemy w takiej aplikacji (Listing 3.7).

```
1 QByteArray data = query.value(2).toString().toUtf8(); //jakieś dane
2 QByteArray data2 = qCompress(data, 9); //pakowanie
3 QByteArray data3 = qUncompress(data2, 9); //rozpakowanie
```

Listing 3.7: Pakowanie i rozpakowanie danych w *Qt*

Jednak, jeśli dane mają być uniwersalne, powinny być opakowane w standardowy nagłówek:

```
0   1
+---+---+
```

```
|CMF|FLG|   (more-->)
+---+---+

+=====================+---+---+---+---+
|...compressed data...|    ADLER32    |
+=====================+---+---+---+---+
```

gdzie:
CMF - metoda kompresji i flagi,
FLG - flagi,
ALDER32 - metoda wyliczająca sumę kontrolną, podobną do *CRC32*, lecz szybszą w działaniu.

To jest standardowy format danych *zlib*, *Qt* dodaje do niego na początek (przed polem CMF) cztery bajty, które określają rozmiar danych przed kompresją. Liczba ta zapisana jest w notacji *BigEndian* (informacje te nie zostały umieszczone w dokumentacji). Jest ona niezbędna do rozpakowania danych (przy jej użyciu sprawdzana jest poprawność operacji). W bazie danych liczba ta jest przechowywana (kolumna *org_size*) w notacji *LittleEndian*, na co trzeba uważać pisząc własną implementację serwera. Domyślną metodą kompresji jest użycie algorytmu *Deflate*.

3.1.3 Rozkazy

Format rozkazów jest zawsze takim sam, dopiero przekazywanie parametrów jest zależne od danego rozkazu. Ogólna budowa przedstawia się następująco:

```
<cztery bajty rozmiaru>ROZKAZ\n
opcjonalne parametry
```

Na początku wiadomości zawsze umieszczany jest w notacji *BigEndian* rozmiar rozkazu (jednak bez tego pola). Wielkość tego obszaru to cztery bajtów (liczba całkowita bez znaku). Po określeniu rozmiaru następuje rozkaz. Jest to słowo pisane wielkimi literami, zakończone znakiem nowej linii ,,\n''. Parametry nie są obowiązkowe, zatem wyżej przedstawiony schemat stanowi minimalną postać komunikatu. Warto przyjrzeć się wszystkim obsługiwanym poleceniom oraz ich odpowiedziom. Aby ułatwić czytanie, pomijane będą pierwsze cztery bajty, określające rozmiar. Należy jednak pamiętać,

że **muszą się one zawsze pojawić, dodatkowo wszystkie liczby zapisane są w notacji *BigEndian*, ciągi tekstowe kończą się zawsze znakiem „\n”**.

Wyróżniamy też trzy standardowe odpowiedzi:

1. `OK\n` - wykonanie rozkazu zakończyło się powodzeniem,
2. `FAIL\n` - wykonanie rozkazu zakończyło się niepowodzeniem,
3. `LOGIN\n` - nie można wykonać danego polecenia, gdyż użytkownik nie jest zalogowany/uwierzytelniony.

Obsługiwanymi przez klienta i serwer poleceniami są:

```
USERLOGIN\n
<email>\n
<hasło - tekst, ciąg hex, sha1>\n
```

Polecenie to jest używane do uwierzytelnienia użytkownika, w odpowiedzi otrzymuje się albo OK albo FAIL. W drugim przypadku dodatkowo zrywane jest połączenie.

```
ADDLOGIN\n
<email>\n
<hasło - tekst, ciąg hex, sha1>\n
```

Polecenie to jest używane do dodania nowego użytkownika, w odpowiedzi dostajemy OK albo FAIL, gdy nie uda się go dodać.

```
CHANGEPASSWORD\n
<hasło stare - tekst, ciąg hex, sha1>\n
<hasło nowe - tekst, ciąg hex, sha1>\n
```

Polecenie to jest używane do zmiany hasła istniejącego i zalogowanego użytkownika. W odpowiedzi dostajemy OK, gdy zmiana się powiedzie lub FAIL, gdy się nie uda albo LOGIN, gdy ktoś próbuje wykonać ten rozkaz, a nie jest zalogowany/uwierzytelniony w danym momencie.

```
GETCHANNELS\n
```

Tym poleceniem pobiera się listę kanałów subskrybowanych przez użytkownika. OK, gdy się powiedzie, FAIL w przeciwnym wypadku. Natomiast zwracany jest LOGIN, gdy użytkownik jest nieuwierzytelniony.

```
GETNEWS\n
<id kanału cztery bajty>
<czas ostatniej synchronizacji - cztery bajty - timestamp>
```

Polecenie służy do pobierania nowych wiadomości. Możliwe odpowiedzi to: OK, FAIL, LOGIN.

```
ADDCHANNEL\n
<url kanału, tekst>\n
```

Subskrybuj nowy kanał. Odpowiedzi to: OK, FAIL, LOGIN.

```
UPDATENEWS\n
<id newsa cztery bajty>
<czas ostatniej zmiany - cztery bajty - timestamp>
<jeden bajt - bool - stan flagi przeczytana>
<jeden bajt - bool - stan flagi usunięta>
```

Uaktualnij wiadomość, czyli wymień oznaczenia. Polecenie zwraca LOGIN, gdy użytkownik nie jest uwierzytelniony, OK w przypadku, gdy zmiany się powiodą, natomiast FAIL, gdy nie uda się uaktualnić wiadomości (na przykład wersja docelowa na serwerze jest nowsza, jeśli to klient wysyła rozkaz lub na odwrót, jeśli robi to serwer).

Formaty odpowiedzi zwracające dane na niektóre rozkazy:

```
TAKECHANNEL\n
<id kanału - cztery bajty>
<tytuł - tekst>\n
<link - tekst>\n
<cztery bajty - bigendian, rozmiar tablicy bajtów>
<tablica bajtów o zadanym rozmiarze, jest to opis kanału
w formie tekstu, ale przesyłany jako tablica bajtów>
```

Tym poleceniem przesyłany jest każdy kanał subskrybowany przez użytkownika. Ilość takich wiadomości zależy od ilości subskrybowanych kanałów (odpowiedź na GETCHANNELS).

```
TAKENEWS\n
<id newsa cztery bajty>
<id kanału cztery bajty>
<tytuł - tekst>\n
<link - tekst>\n
<org rozmiar wiadomości - cztery bajty - BigEndian>
<4 bajty rozmiar danych>
<spakowane dane wiadomości>
<czas publikacji - cztery bajty - timestamp>
<czas ostatniej zmiany - cztery bajty - timestamp>
<jeden bajt - bool - stan flagi przeczytana>
<jeden bajt - bool - stan flagi usunięta>
```

Powyższe polecenie przesyła wiadomość do klienta, jest odpowiedzią na GETNEWS.

3.1.4 Moduł ustawień

Aplikacja do poprawnego działania wymaga ustalenia kilku niezbędnych wartości, jak na przykład numer portu, na którym będzie oczekiwać połączeń od klientów. Domyślnie w *Qt* mamy do dyspozycji klasę *QSettings* [20]. Ma ona jednak kilka ograniczeń, do których należą:

- brak obsługi baz danych, co oznacza, że wszystkie dane są przetrzymywane w plikach tekstowych bądź w rejestrze w przypadku systemu Windows,
- dane z tych plików są wczytywane przy tworzeniu obiektu do pamięci w całości,
- zapis odbywa się, gdy obiekt kończy życie.

Oznacza to, że chcąc dodać jedno ustawienie musimy zapisać na dysku cały plik. Analogicznie do odczytu - musimy wczytać cały plik. Tę niedogodność rozwiązano w projekcie tworząc nową klasę, w oparciu o już stworzoną (twórcy *Qt Creatora* stworzyli

odpowiednią klasę udostępniając ją na licencji LGPL). Miała ona jednak kilka braków, które uzupełniono, dzięki czemu obiekt tej klasy stał się bardziej uniwersalny i oferuje większe możliwości konfiguracji. Można na przykład określić nazwę pliku, miejsce jego przechowywania. Dodano domyślne ścieżki przechowywania ustawień dla aplikacji na różne systemy.

Do modułu ustawień można się odwoływać z dowolnego miejsca w aplikacji, dzięki stworzonemu obiektowi klasy **Core** będącego *Singletonem* (Listing 3.8).

```
1 QVariant setting = Core::instance()->settings()->value(
2                                   "plugin/PostgreSQL/port", //klucz
3                                   QVariant() ); //wartość, gdy nie znaleziono klucza
```

Listing 3.8: Przykładowe pobranie wartości ustawienia dla klucza

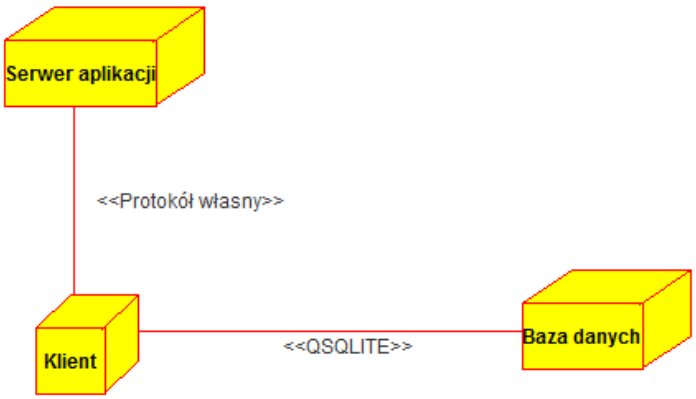
Obiekt klasy **Core** pełni rolę centralnego zarządcy w aplikacji. Dzięki niemu możliwy jest odczyt ustawień, rejestracja ,,załadowanej'' wtyczki (dzięki temu może ona dodać swoją pozycje do okna konfiguracji) lub też możliwe jest ładowanie/odłączanie wtyczek.

3.2 Klient

Wymagania

Aplikacja klienta w przeciwieństwie do serwera będzie obsługiwana przez użytkownika. Wiąże się to z pewnymi modyfikacjami funkcjonalności tej części projektu w porównaniu do serwera. Klient sam nie pobiera bezpośrednio z portali wiadomości RSS. Aplikacja zatem musi spełniać następujące wymagania:

1. aplikacja musi posiadać okno główne,
2. wygląd okien musi być niezależny od danych,
3. interfejs powinien być jednakowy na każdej platformie,
4. program powinien umożliwiać przeglądanie wiadomości RSS,
5. każda wiadomość RSS może być oznaczona jako *przeczytana* lub *usunięta*,
6. aplikacja ma możliwość kontaktu z różnymi SZBD,
7. program komunikuje się tylko z serwerem,



Rys. 3.3: Diagram zewnętrznych połączeń klienta

8. synchronizacja odbywa się automatycznie,

9. z serwerem wymieniane są niezbędne dane, dzięki którym na każdej aplikacji klienckiej, na której zalogowany jest ten sam użytkownik są identyczne dane.

Na podstawie przedstawionych wyżej założeń, można określić diagram połączeń klienta niezbędnych do prawidłowego działania (Rys. 3.3).

Implementacja

Klient podobnie, jak serwer, został napisany przy użyciu *Qt 4.7.0*. Dzieli ten sam moduł do obsługi wtyczek i ustawień, co serwer. Rożnica polega na tym, że w tej części aplikacji należało stworzyć interfejs użytkownika. Zostało to zrealizowane przy pomocy wzorca projektowego *Model-Widok-Kontroler* (Rys. 3.4).

Biblioteka stworzona przez *Nokię* wspiera ten wzorzec [17]. Klasami ,,kontrolera'' są używane w projekcie *QTableView* (postać tabeli, z kolumnami i wieloma wierszami) oraz *QListView* (lista/drzewo, wiele wierszy, ale jedna kolumna). Obiekty tych dwóch klas są odpowiedzialne za wyświetlanie danych, interakcję z użytkownikiem. O tym, jakie dane będą wyświetlane, decyduje klasa modelu danych, która musi bazować na abstrakcyjnej klasie *QAbstractTableModel/QAbstractListModel*. Obiekt tej klasy musi być dołączony do kontrolera. Ostatnim elementem układanki jest tak zwana ,,delegacja''. Obiekt tej klasy odpowiedzialny jest za to, jak kontroler prezentuje dane. W tym celu należy ,,podłączyć'' do niego obiekt klasy dziedziczącej po *QStyledItemDelegate* (Listing 3.9).

Wszystkie operacje opierają się o ,,role'', dla każdej jest zdefiniowane domyśle zachowanie, które oczywiście można zmieniać. Przykładowo *Qt::DisplayRole* oznacza rolę, dla której w modelu danych przypisany jest obiekt klasy *QString*, czyli zawartość jest tekstem, który należy wyświetlić (Listing 3.10 i 3.11).

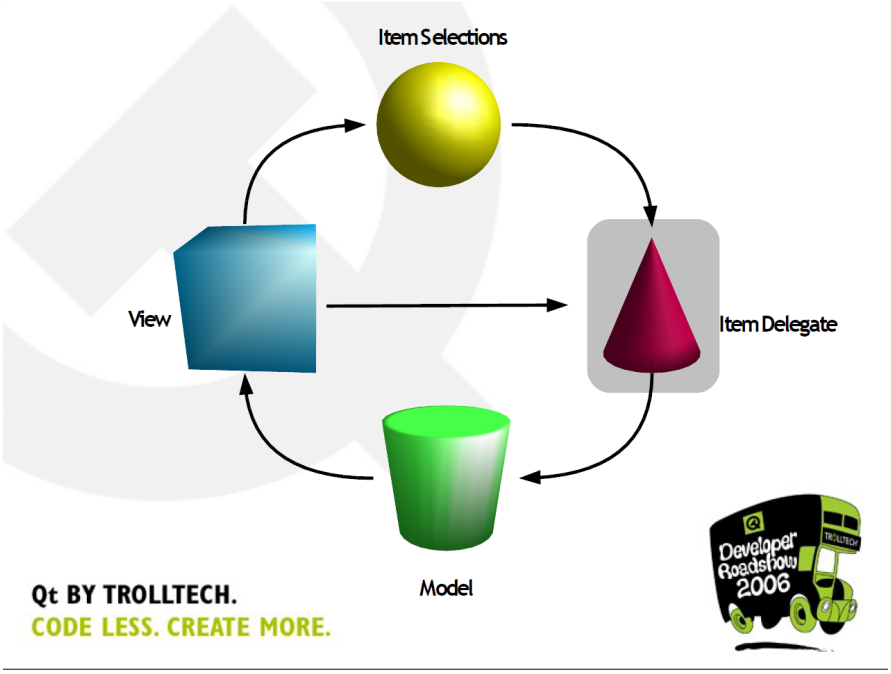
```
1 ui->listView->setModel( models->getListModel() ); //model danych
2 ui->listView->setItemDelegate( models->getListDelgate() ); //delegacja
```

Listing 3.9: Ustawienie modelu danych i delegacji dla listy

```
1 //klasa ta w zależności od roli zwraca określone dane
2 QVariant ListModel::data(const QModelIndex &index, int role) const {
3     switch( role ){
4     case Qt::DisplayRole:
5         return static_cast<const ListModelPrivate::ListEntry&>
6                                     (d->m_channels.at( index.row() )).name;
7     case (Qt::UserRole + 1):
8         return static_cast<const ListModelPrivate::ListEntry&>
9                                     (d->m_channels.at( index.row() )).unread;
10    case Qt::DecorationRole:
11        return static_cast<const ListModelPrivate::ListEntry&>
12                                    (d->m_channels.at( index.row() )).favicon;
13    }
14    return QVariant();
15 }
```

Listing 3.10: Role w klasie modelu danych

```
1 void TableModel::paint(QPainter *painter,
2                         const QStyleOptionViewItem &option,
3                         const QModelIndex &index) const
4 {
5     //domyślne zachowanie
6     QStyledItemDelegate::paint(painter, option, index);
7
8     //umieszcza w tabeli znak przeczytania, bądź nie
9     if( index.column() == 1
10        && qVariantCanConvert<bool>( index.data(Qt::UserRole) ) )
11    {
12        QRect r = option.rect;
13        if( index.data( Qt::UserRole ).toBool() ){
14            r.setX( r.x() + (r.width() - m_read.width())/2 );
15            r.setY( r.y() + (r.height() - m_read.height())/2 );
16            r.setSize( m_read.size() );
17            painter->drawPixmap( r, m_read );
18        }else{
19            r.setX( r.x() + (r.width() - m_unread.width())/2 );
20            r.setY( r.y() +(r.height() - m_unread.height())/2 );
21            r.setSize( m_unread.size() );
```



Rys. 3.4: Model-Widok-Kontroler [21]

```
22                 painter->drawPixmap( r, m_unread );
23             }
24         }
25 }
```

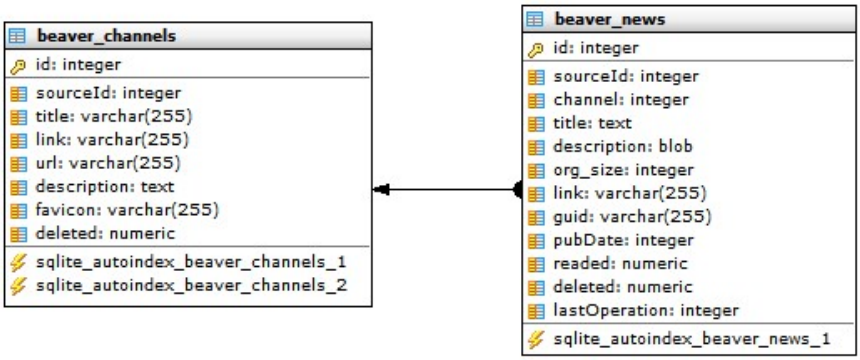
Listing 3.11: Role w delegacji, własne malowanie danych

3.2.1 SQLite

Obsługa bazy *SQLite* przy pomocy biblioteki *Qt* jest dostępna zaraz po instalacji. W tym celu wystarczy określić, że będzie się z niej korzystać oraz podać nazwę bazy danych, czyli w tym wypadku ścieżkę dostępu do pliku z bazą (Listing 3.12).

```
1 QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE", //sterownik
2                                              "SQLite::Beaver::Plugin");
3                                              //nazwa połączenia
4 db.setDatabaseName( "dbfile.sqlite" );
5 if( !db.open() ){//otwórz bazę
6     //nie udało się otworzyć pliku (ani go utworzyć)
7 }
8 //mamy połączenie
```

Listing 3.12: Użycie bazy SQLite



Rys. 3.5: Diagram bazy danych po stronie klienta

Baza ta jest popularna, ale zarazem prosta. Ma ograniczoną ilość typów danych (NULL, BLOB, TEXT, INTEGER), jest dostępna dla każdego, nie ma żadnego uwierzytelniania.

Diagram bazy danych aplikacji klienta przedstawiony został na rysunku 3.5. W obecnej implementacji są to dwie tabele, które przechowują wszystkie informacje.

Podsumowanie

Dzięki zastosowaniu biblioteki *Qt* obie aplikacje projektu stały się wieloplatformowe. Każdy z programów może korzystać z różnych SZBD, dzięki systemowi wtyczek. Domyślnie zostały dostarczone rozszerzenia do obsługi baz danych *SQLite3* dla aplikacji klienta i *PostgreSQL* dla serwera. Jednakowy wygląd sprawia, że użytkownik na każdym systemie operacyjnym nie musi się uczyć od nowa obsługi programu. Przejrzysty schemat komunikacji sprawia, że wymiana danych pomiędzy różnymi klientami jest prosta i bezproblemowa.

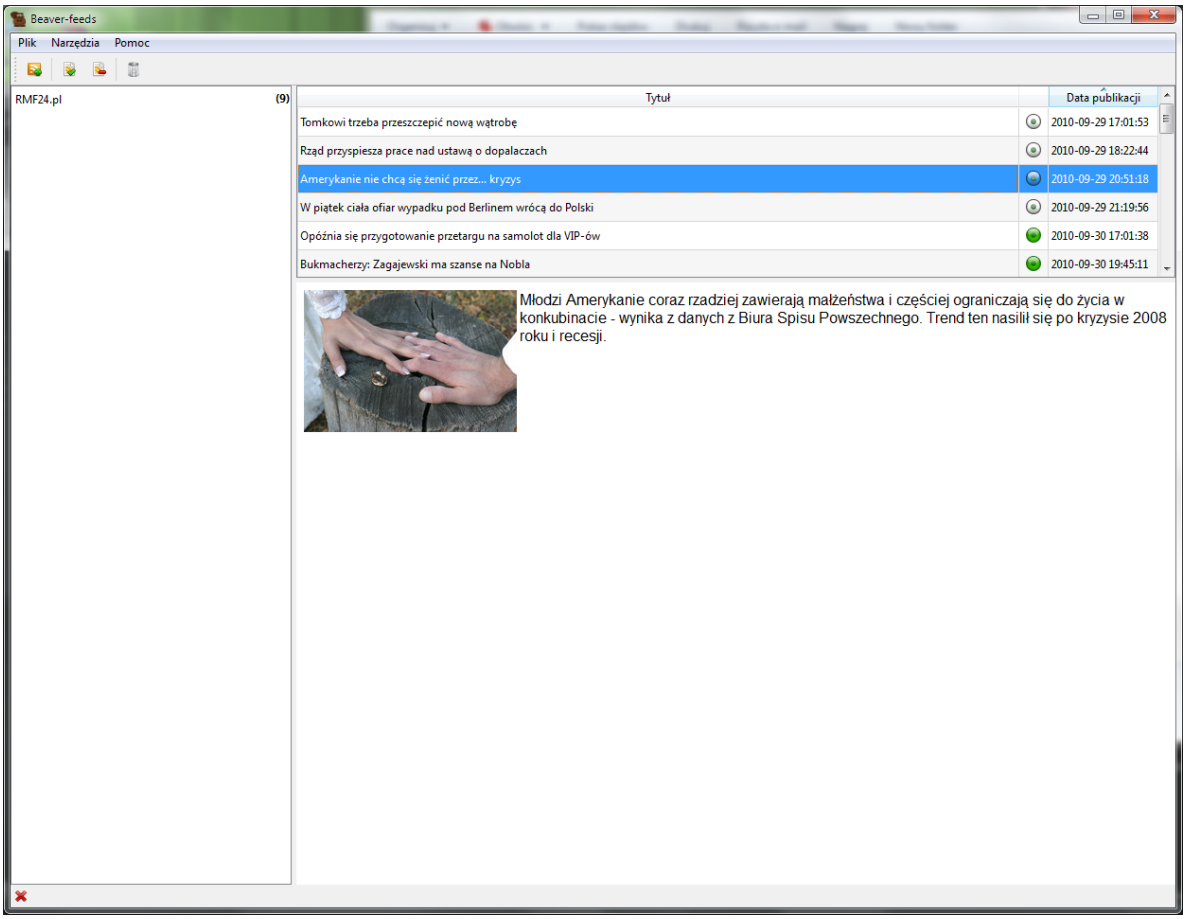
Rozdział 4

Opis użytkowania stworzonej aplikacji

Poniższy rozdział przedstawia opis sposobu używania stworzonych w ramach pracy aplikacji. Stworzony system składa się z dwóch części: aplikacji klienta, która służy do przeglądania wiadomości RSS oraz aplikacji serwera, która w sposób ciągły pobiera wiadomości z subskrybowanych kanałów. Aplikacja serwera jest niezależna i bezobsługowa, uruchomiona sama wykonuje wszystkie niezbędne. Klient jest aplikacją obsługiwaną przez użytkownika. Dzięki niej może on czytać subskrybowane kanały lub dodawać nowe kanały do subskrypcji. Aplikacja klienta, jeśli jest nawiązane połączenie z serwerem, automatycznie dokonuje synchronizacji danych z tym serwerem.

4.1 Klient

Aplikacja kliencka przeznaczona jest głównie do uruchamiania na komputerach użytkowników końcowych (zwanych też dalej czytelnikami). Osoby takie swoją pracę z komputerem opierają często nie na wiedzy, a na przyzwyczajeniach, przez co nowe aplikacje muszą przypominać już istniejące, bądź wprowadzać nowe, lecz proste rozwiązania. Zwykli użytkownicy oczekują jedynie, by dany program działał, zaś jakikolwiek błąd pojawiający się w trakcie działania aplikacji jest, według nich, najczęściej z winy twórcy aplikacji.

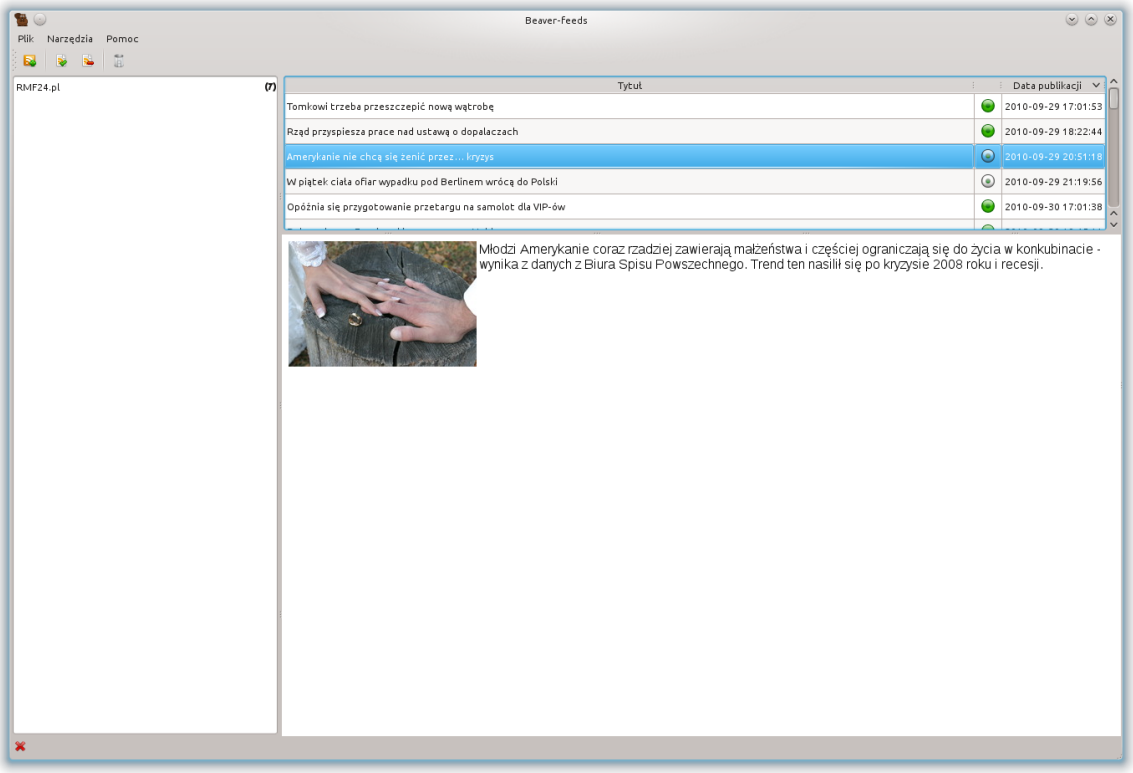


Rys. 4.1: Główne okno aplikacji klienta w systemie Windows

4.1.1 Okno główne

Okno główne aplikacji zostało stworzone na wzór już istniejących rozwiązań czytników RSS lub klientów pocztowych (Rys. 1.1 lub Rys. 1.2). Jego wygląd został przedstawiony na rysunkach 4.1, 4.2, 4.3. Jak można łatwo zauważyć, prezentowane ekrany pochodzą z trzech różnych systemów operacyjnych, a mimo to wyglądają niemal identycznie. Mają ten sam układ, różnica występuje tylko w stylach systemowych. Po lewej stronie okna znajduje się *lista kanałów* subskrybowanych przez danego użytkownika. Obok nazwy kanału, po lewej stronie, może pojawić się ikona portalu, z którego pochodzi kanał, zaś po prawej stronie w nawiasach, umieszczona została liczba nieprzeczytanych wiadomości.

Centralnie na górze okna aplikacji klienta występuje *lista wiadomości danego kanału*. Widoczne są przy tym takie informacje, jak tytuł, czas publikacji oraz symbol przeczytania (zielone kółeczko oznacza, że wiadomość jest nieprzeczytania, a białe, częściowo

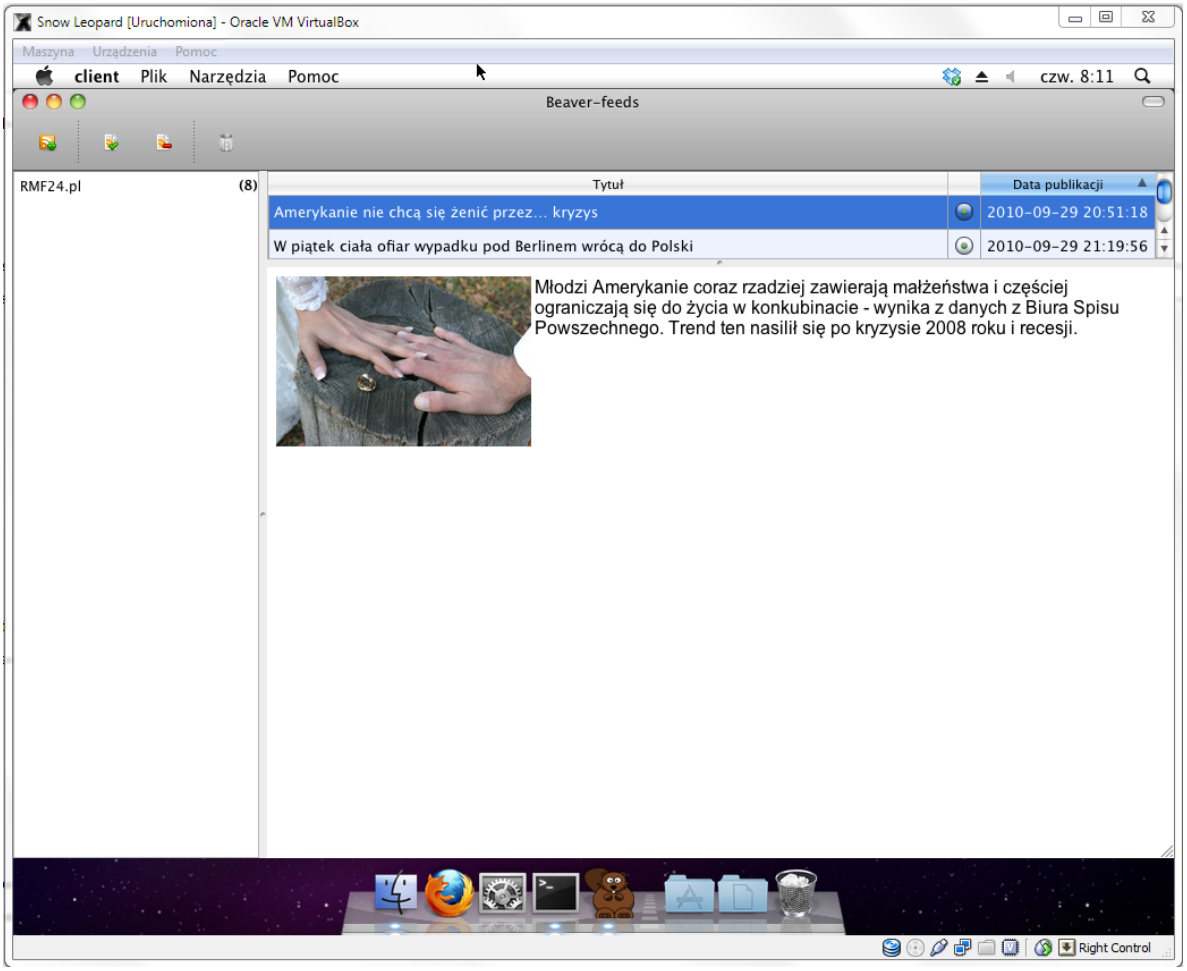


Rys. 4.2: Główne okno aplikacji w systemie Linux (KDE 4.6)

przezroczyste - wiadomość jest przeczytana). Kliknięcie jakiegokolwiek tematu, oznacza go jako przeczytany, powoduje też ,,załadowanie'' wiadomości do ,,ramki'' poniżej listy. Natomiast kliknięcie na belkę tytułową kolumny na liście, sortuje dane według tej kolumny. Pole z treścią jest interaktywne, to znaczy po załadowaniu wiadomości można klikać odnośniki, które przeniosą użytkownika na przykład na stronę główną portalu, który publikuje dany kanał, bądź do pełnej postaci wiadomości. Za wyświetlanie i poprawne przetwarzanie stron odpowiedzialny jest silnik *WebKit*, taki sam, jak w przeglądarkach *Apple Safari* lub *Google Chrome*, a zatem nie trzeba się obawiać, że dana witryna spowolni działanie programu, bądź będzie wyświetlana z błędami.

Na dole okna można zauważyć niewielki pasek statusu z ikoną po lewej stronie. Symbol ten reprezentuje status połączenia aplikacji z serwerem. Zaprezentowane na rysunku 4.4 ikony kolejno reprezentują możliwy stan połączenia z serwerem:

- rozłączony,
- trwa próba połączenia (łączenie),



Rys. 4.3: Główne okno aplikacji w systemie MacOS X

Rys. 4.4: Ikony reprezentujące stan połączenia aplikacji z serwerem

Rys. 4.5: Ikony paska narzędzi

- połączono, lecz bez szyfrowania,
- nawiązano bezpieczne połączenie (SSL).

Po ,,najechaniu'' na ikonę pojawia się tekst na pasku z opisem stanu połączenia. Zaś samo kliknięcie na nią powoduje połączenie lub rozłączenie z serwerem.

Pod paskiem menu widoczny jest pasek narzędzi z czterema przyciskami (Rys. 4.5). Ich ikony symbolizują akcję, jaka nastąpi po naciśnięciu danego przycisku:

- *Subskrybuj kanał* - po jego wciśnięciu istnieje możliwość podania adresu nowego kanału, który użytkownik chce subskrybować. Po zatwierdzeniu operacji, serwer prześle użytkownikowi zawartość kanału. Jeśli był on już kiedyś dodany przez innego użytkownika, to dostaniemy także wpisy wcześniejsze, jeśli nie, to tylko aktualne.

- *Oznacz jako przeczytane* - możliwe jest zaznaczenie wielu wiadomości, zaś po kliknięciu tego przycisku zostaną one oznaczone jako przeczytane (zielone kółeczko z listy zmieni się w białe).

- *Oznacz jako nieprzeczytane* - oznacza wiadomości jako nieprzeczytane.

- *Usuń wiadomość* - służy do usuwania wiadomości. Tutaj również można zaznaczyć wiele pozycji[1]. Warto dodać, że one tak naprawdę nie są usuwane, a jedynie ukrywane, tak aby w przyszłości można było je przywrócić (o ile pojawi się taka opcja).

Pasek menu podzielony został na trzy kategorie: *Plik*, *Narzędzia* oraz *Pomoc*. W menu *Plik* zdefiniowano akcję do połączenia z serwerem oraz możliwość wyjścia z aplikacji. Dzięki menu *Pomoc* użytkownik może uzyskać pewne informacje o samym programie (na przykład wersja, autor, wersja *Qt*). Menu *Narzędzia* zawiera aktualnie tylko jedną pozycję: *Preferencje*, która otwiera okno z ustawieniami aplikacji.

4.1.2 Okno ustawień

Na rysunkach 4.6, 4.7 i 4.8 przedstawiono wygląd okna konfiguracji aplikacji klienta na trzech różnych systemach operacyjnych. Jak widać ich wygląd jest również identyczny.

Po lewej stronie okna, wyświetlanego w każdym systemie, znajduje się lista kategorii ustawień. Ich liczba może być zmienna, zależnie od ilości ,,załadowanych'' wtyczek. Niemniej dwie pozycje zawsze będą się na niej pojawiać (są stałe): *Sieć* i *Wtyczki*.

Zakładka *Sieć* służy do określania parametrów połączenia z serwerem. Gdy jest ona aktywna, można podać adres, pod którym znajduje się serwer oraz port, na którym działa. Można też określić nazwę użytkownika i hasło. Warte dokładniejszego wyjaśnienia są trzy opcje, którą posiadają tylko dwa stany: *włączony* albo *wyłączony*:

[1] usuwanie odbywa się również poprzez wciśnięcie *Delete* na klawiaturze

- *Ignoruj błędy SSL* - w przypadku, gdy serwer posiada certyfikat z podpisem własnym, aplikacja nie połączy się z nim, gdyż nie można stwierdzić, czy certyfikat jest autentyczny. Jednak, jeśli użytkownik ma pewność na temat danego serwera, to może zignorować część ostrzeżeń, a wtedy bez problemu nawiąże szyfrowane połączenie.

- *Łącz przy uruchamianiu* - określa, czy w trakcie uruchamiania aplikacji ma zastać nawiązane połączenie z serwerem. Jeśli nie, to trzeba to zrobić ręcznie.

- *Pokazuj błędy połączenia* - zaznaczenie tej opcji powoduje, że o wszelkich błędach z połączeniem użytkownik będzie informowany za pomocą ,,wyskakujących'' okienek z informacją. Gdy opcja ta jest nieaktywna, to informacje te pokazują się przez kilka sekund na pasku statusu.

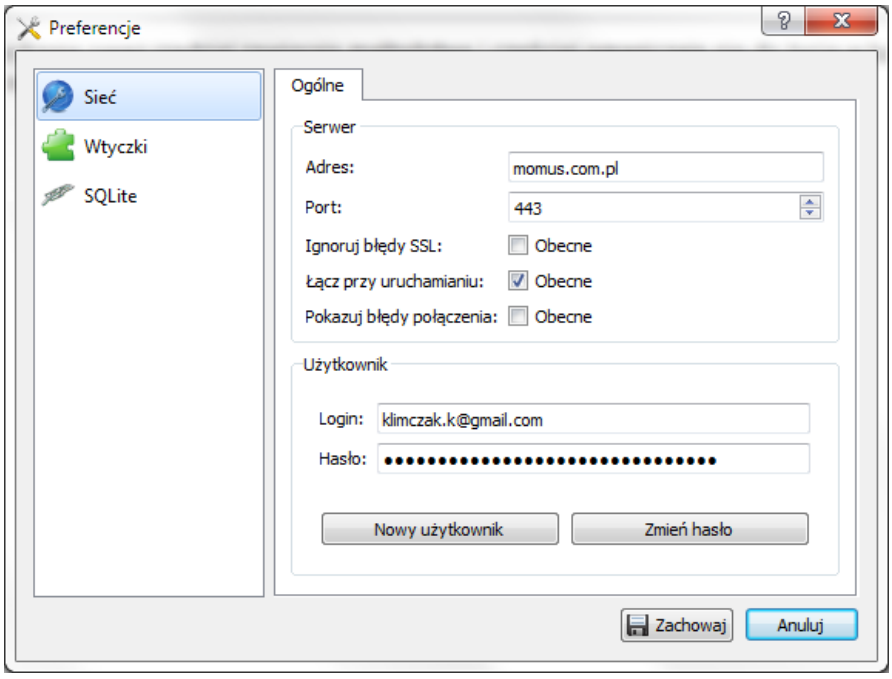
W kategorii *Wtyczki* można określić, którą wtyczkę bazy danych aplikacja będzie ładować. **Wszystkie wtyczki powinny być umieszczone w folderze ,,plugins'' znajdującym się w katalogu z aplikacją. W przeciwnym wypadku wtyczka nie zostanie odnaleziona**. Jeśli użytkownik posiada więcej niż jedną wtyczkę, to może ją wybrać klikając przycisk na liście.

Ostatnią kategorią jest *SQLite*. Pozycja ta została ,,załadowana'' z wtyczki bazy danych i służy do jej konfiguracji. Tylko zaawansowani użytkownicy powinni zmieniać ustawienia znajdujące się w tym miejscu. Domyślne wartości są prawidłowe dla większości czytelników. Na tej karcie można zmienić położenie pliku z bazą danych lub prefiks tabel, jeśli użytkownik używa jednej bazy do kilku programów.

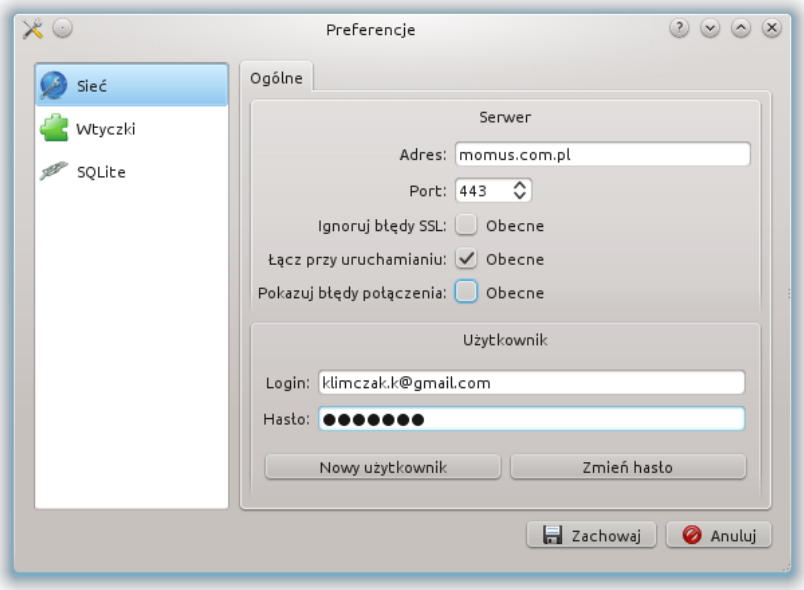
Wszystkie zmiany są zapamiętywane po wciśnięciu przycisku *Zachowaj*. Naciśnięcie przycisku *Anuluj*, bądź klawisza *ESC* na klawiaturze spowoduje przywrócenie wartości wcześniej zapisanych.

4.2 Serwer

Aplikacja serwera przeznaczona jest do działania w sposób automatyczny i niewymagający interwencji użytkownika lub administratora. Nie posiada ona interfejsu głównego okna, zaś po uruchomieniu pojawia się tylko ikona ,,bobra'' na pasku zadań w obszarze powiadomień. Po kliknięciu na nią prawym klawiszem pojawia się menu z trzema pozycjami: *Wyjście* - zamyka aplikacje, *O serwerze* - wyświetlane są informacje o serwerze (autor, wersja, wersja Qt, port nasłuchu). Trzecia pozycja to *Preferencje*.



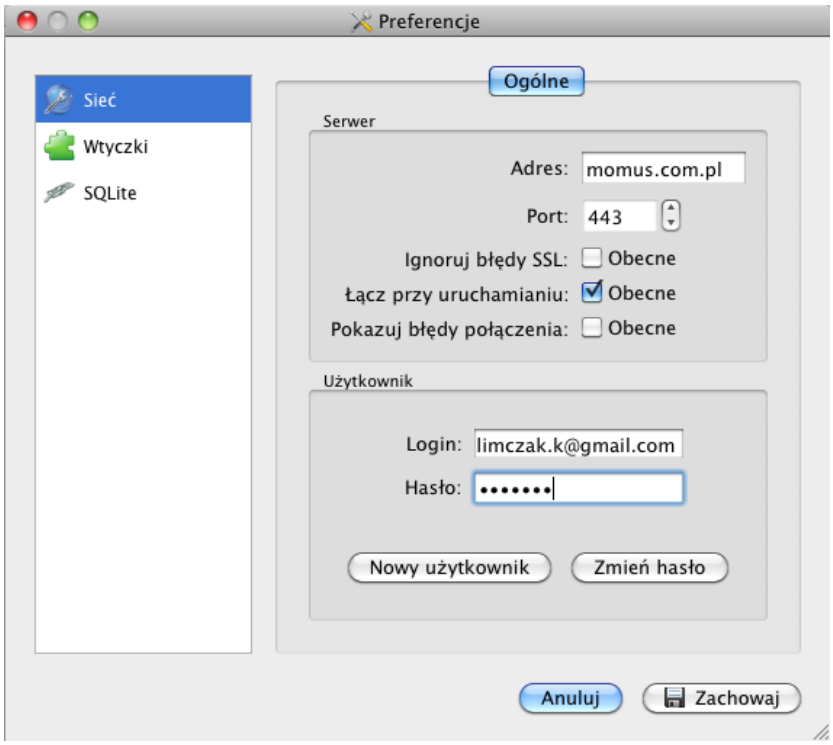
Rys. 4.6: Okno preferencji - system Windows



Rys. 4.7: Okno preferencji - system Linux (KDE 4.6)

Po jej kliknięciu na ekranie komputera pojawia się okno ustawień aplikacji. Jest ono identyczne, jak w aplikacji klienta, ma tylko nieco inną zawartość.

Kategoria *Wtyczki* jest identyczna, jak w aplikacji, opisanej w poprzednim podroz-



Rys. 4.8: Okno preferencji - system MacOS X

dziale. W kategorii *Sieć* można określić parametry niezbędne, by aplikacje klienckie mogły nawiązać połączenie: port nasłuchu (taki sam musi być ustawiony w aplikacji klienta) oraz ustawienia certyfikatów. Aplikacja domyślnie jest wyposażona w certyfikat z podpisem własnym, jednak może być on używany tylko do testów (podpis dla *localhost*). Należy więc wskazać ścieżkę dostępu do plików, gdzie w formacie *PEM* są zapisane: klucz prywatny, certyfikat oraz certyfikat CA dla certyfikatu użytkownika.

Ostatnią pozycją na liście jest *PostgreSQL* (w domyślnej instalacji). Dzięki opcjom umieszczonym na tej karcie można połączyć się z serwerem baz danych *PostgreSQL*. W celu nawiązania udanego połączenia należy określić: położenie serwera, nazwę użytkownika, hasło dostępu oraz port, na którym SZBD oczekuje na połączenia. Można też określić, czy połączenia ma być szyfrowane, o ile serwer bazy danych je obsługuje. Dodatkowo, należy podać nazwę bazy danych (domyślnie aplikacja próbuje utworzyć bazę o nazwie *beaver*, jeśli nie jest nic podane). Ważne jest określenie prefiksu tabel, z jakich korzysta serwer. Może się zdarzyć, że użytkownik ma do dyspozycji tylko jedną bazę, z której korzysta już kilka aplikacji. Bez określenia prefiksu może dojść do konfliktu nazw i któraś aplikacja nie działałaby poprawnie.

Ostatnim elementem, o którym należy pamiętać w obu aplikacjach, to dodanie

odpowiednich reguł do zapory ogniowej działającej w systemie, tak by połączenia pomiędzy klientem a serwerem nie były blokowane.

Podsumowanie

Aplikacje stworzone w ramach projektu zachowują się zgodnie z założeniami. Na każdym z obsługiwanych systemów operacyjnych oferują identyczny interfejs i funkcjonalność. Użytkownik może bez problemu przełączać się pomiędzy różnymi systemami operacyjnymi, które są zainstalowane na jego komputerze. Identyczny interfejs aplikacji sprawia, że nie musi on zmieniać przyzwyczajeń. Komunikacja pomiędzy aplikacją klienta a aplikacją serwera, z punktu widzenia użytkownika, jest zawsze taka sama, a co ważne, nie wymaga jego ingerencji w ten proces.

Dobrze skonfigurowana aplikacja powinna działać bez większych problemów. Problemem nie będzie przenoszenie konfiguracji na różne maszyny lub systemy. Jedyną czynnością, jaką należałoby wtedy wykonać, to zmiana ścieżki dostępu do bazy danych i ewentualnie zmiana prefiksu nazw tabel.

Podsumowanie

Na podstawie powyższej pracy dyplomowej można stwierdzić, że udało się osiągnąć zamierzony cel. Platforma, którą stworzono jest dobrym połączeniem zalet obecnych rozwiązań dostępnych na rynku. Dzięki otwartości kodu, opisowi ,,języka'' komunikacji klienta z serwerem dość łatwe jest zaprojektowanie innych klientów łączących się ze stworzoną platformą. W przyszłości możliwe jest zatem stworzenie dodatkowego klienta internetowego, dzięki któremu stwierdzenie dostępności zawsze i wszędzie będzie prawdziwe (albo też innego klienta na komputery stacjonarne lub urządzenia mobilne). Obecnie kontakt z wiadomościami możliwy jest tylko za pomocą aplikacji klienckiej, bez niej nie uzyskamy dostępu do danych. Będą one jednak aktualizowane i gdy tylko będzie możliwy dostęp do klienta i Internetu ten zsynchronizuje dane, więc nie trzeba obawiać się, że coś nas ominie, bądź będzie nieaktualne.

Niestety na chwilę obecną projekt nie nadaje się do wdrożenia na rynek. Brakuje mu wielu funkcjonalności. Z bardziej potrzebnych funkcji należałoby wymienić chociażby obsługę *podcastów/videocastów*, obsługę znaczników, grupowanie kanałów w podkatalogach, resetowanie hasła po jego zapomnieniu lub jego bardziej bezpieczną wymianę. Natomiast wartą do wdrożenia powinna być możliwość integracji z serwisami, takimi jak *Facebook, Netvibes* lub *Google Reader* (na przykład za pomocą systemu wtyczek). Równie przydatną oraz ułatwiającą obsługę programu byłaby implementacja technologii *OpenID*, czyli umożliwienie logowania się do serwera za pomocą jednego globalnego loginu i hasła, używanego na wielu stronach. Takie możliwości oferują chociażby *Facebook* lub *Google*, gdzie za pomocą nazwy użytkownika i hasła do ich portali można też logować się na innych stronach obsługujących *OpenID*. Na chwilę obecną użytkownik musi pamiętać do stworzonej platformy kolejny identyfikator i hasło, które prawdopodobnie są takie same, jak w innych miejscach w Sieci.

Aktualnie aplikacje oferują podstawowe działania, niezbędne do przeglądania wiadomości, udostępnianych przez kanały *RSS*. Oprócz wspomnianych niedostatków w funkcjonalności, brakuje również przeprowadzenia zakrojonych na szeroką skalę testów przez

innych użytkowników na różnych konfiguracjach systemów. Ich brak powoduje, iż nie można wydać aplikacji na rynek, gdyż nie można zapewnić, że będzie ona działać poprawnie na komputerach innych użytkowników.

Pomimo tych niedoskonałości można stwierdzić, iż praca może być wykorzystana w przyszłości. Pokazano bowiem, iż nie trzeba ograniczać się tylko do jednego systemu operacyjnego lub platformy. Stworzenie takiej aplikacji nie jest łatwe i wymaga znajomości różnic pomiędzy systemami, przez co ciężko jest jednej osobie zrobić w pełni funkcjonalną aplikację na wiele platform. Jednakże brak przywiązania do konkretnego środowiska daje użytkownikowi wybór, zamiast zmuszać go do używania określonego oprogramowania lub systemu operacyjnego. To użytkownik decyduje, co jest dla niego najlepsze, a nie wydawca aplikacji. Dlatego, powinno się zaoferować to, co naprawdę jest mu potrzebne. Ważnym elementem jest tutaj danie możliwości dostosowania aplikacji do upodobań użytkownika, na przykład poprzez zmianę wyglądu lub rozszerzenie funkcjonalności za pomocą systemu wtyczek. Jest to jednak zadanie na przyszłość, dotyczące rozbudowy projektu.

Spis rysunków

Spis listingów

Bibliografia

[1] Qt - Cross-platform application and UI framework http://qt.nokia.com/ (stan na 2.2011)

[2] Dokumentacja techniczna Qt 4.7 http://doc.trolltech.com/4.7/ (stan na 2.2011)

[3] PostgreSQL: The world's most advanced open source database http://www.postgresql.org/ (stan na 2.2011)

[4] Dokumentacja techniczna bazy danych PostgreSQL 9.0 http://www.postgresql.org/docs/9.0/static/ (stan na 2.2011)

[5] SQLite Home Page http://www.sqlite.org/ (stan na 2.2011)

[6] Dokumentacja techniczna bazy danych SQLite 3.6.19 http://www.sqlite.org/docs.html (stan na 2.2011)

[7] RSS 2.0 at Harvard Law http://cyber.law.harvard.edu/rss/rss.html (stan na 2.2011)

[8] Oficjalna specyfikacja standardu RSS 2.0 http://www.rssboard.org/rss-specification (stan na 2.2011)

[9] Wikipedia - RSS http://en.wikipedia.org/wiki/RSS (stan na 2.2011)

[10] ZLIB Compressed Data Format Specification version 3.3 - dokumentacja RFC formatu danych zlib http://www.faqs.org/rfcs/rfc1950.html (stan na 2.2011)

[11] Nieoficjalne (jedyne) API dla Google Reader http://code.google.com/p/pyrfeed/wiki/GoogleReaderAPI (stan na 2.2011)

[12] Opis tablicy na portalu Facebook http://www.facebook.com/help/?page=443 (stan na 2.2011)

[13] Zbiór narzędzi dla programistów od portalu Facebook http://developers.facebook.com/ (stan na 2.2011)

[14] Zbiór narzędzi dla programistów od portalu Netvibes http://dev.netvibes.com/ (stan na 2.2011)

[15] Wikipedia, wolna encyklopedia - Extensible Messaging and Presence Protocol http://pl.wikipedia.org/wiki/Extensible_Messaging_and_Presence_Protocol (stan na 2.2011)

[16] Building the QPSQL plugin on Windows using MinGW http://www.qtcentre.org/wiki/index.php?title=Building_the_QPSQL_plugin_on_Windows_using_MinGW (stan na 2.2011)

[17] Model/View Programming http://doc.qt.nokia.com/4.7/model-view-programming.html (stan na 2.2011)

[18] Macros for Defining Plugins http://doc.qt.nokia.com/4.7/qtplugin.html (stan na 2.2011)

[19] QPluginLoader Class Reference http://doc.qt.nokia.com/4.7/qpluginloader.html (stan na 2.2011)

[20] QSettings Class Reference http://doc.trollteck.com/4.7/qsettings.html (stan na 2.2011)

[21] Marius Bugge Monsen, MSc (NTNU),Software Developer, prezentacja ,,PRACTICAL MODEL VIEW PROGRAMMING'' http://www.slideshare.net/mariusbu/practical-model-view-programming-roadshow-version (stan na 2.2011)