Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Sebastian Zagrodzki
Nr albumu: 181071

Rozproszony system monitorowania sieci komputerowych

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Janiny Mincer-Daszkiewicz
Instytut Informatyki MIM UW

Wrzesień 2006

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora pracy

Streszczenie

W pracy przedstawiono implementację systemu powiadomień, przeznaczonego przede wszystkim do przesyłania administratorom sieci informacji na temat zdarzeń sieciowych. Zaprezentowano również kilka, napisanych w różnych językach programowania, programów klienckich współdziałających z systemem. Dołączono przykładowe aplikacje monitorujące niektóre aspekty pracy urządzeń sieciowych, korzystające z systemu powiadomień.

Słowa kluczowe

monitoring, zarządzanie siecią, powiadomienia

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

C. Computer Systems Organization
C.2 Computer-Communication Networks
C.2.3 Network Operations
C.2.4 Distributed Systems

Tytuł pracy w języku angielskim

Distributed monitoring of events in network management

Spis treści

Wprowadzenie

Jednym z elementów zarządzania siecią komputerową jest monitorowanie na bieżąco poprawności działania sieci. W dużych sieciach monitoruje się niezależnie co najmniej kilka aspektów ich funkcjonowania, np. dostępność węzłów, wymagające szczególnej uwagi wpisy w dziennikach systemowych, poziom ruchu na poszczególnych łączach, obciążenie kluczowych serwerów, jak również takie rzeczy jak napięcie zasilania w systemach UPS czy temperatura w pomieszczeniach ze sprzętem sieciowym ([10, 3]). Użytkownicy sieci oczekują, że sieć będzie działała sprawnie 24 godziny na dobę, 7 dni w tygodniu, najlepiej bez żadnych przerw. Monitorowanie wielu aspektów funkcjonowania sieci ma na celu szybkie lokalizowanie pojawiających się problemów oraz możliwie najskuteczniejscze zapobieganie awariom.

W większości firm funkcjonują systemy informatyczne, których zadaniem jest obserwowanie urządzeń sieciowych i informowanie o zauważonych nieprawidłowościach. W większych sieciach (kilkadziesiąt lub więcej urządzeń) ilość zarejestrowanych zdarzeń może dochodzić nawet do setek na dobę.

Niektórzy dostawcy usług sieciowych zatrudniają operatorów, którzy przez całą dobę weryfikują informacje zgłaszane przez systemy monitoringu. Operator na podstawie swojej wiedzy dotyczącej struktury sieci, sposobu funkcjonowania konkretnych urządzeń i możliwych konsekwencji poszczególnych zdarzeń może podjąć decyzję o powiadomieniu administratora. Ma to znaczenie zwłaszcza poza zwykłymi godzinami pracy, np. w nocy. Operator może wtedy zdecydować o tym, żeby poinformować administratora dopiero rano. Dodatkowo, przekazując informacje może je streścić tak, aby administrator nie musiał samodzielnie przeglądać dziesiątek komunikatów. Operator potrafi też kontaktować się z administratorem na różne sposoby – jeśli jest środek nocy, prawdopodobnie wysyłanie listu elektronicznego mija się z celem, dlatego lepiej skorzystać z telefonu; jeśli telefon komórkowy nie odpowiada, można zadzwonić na telefon domowy; itp., itd.

Jednak na utrzymanie operatorów stać tylko dużych dostawców usług, ze względu na wysokie koszty takiego rozwiązania. Pozostali muszą korzystać z całkowicie automatycznych systemów powiadomień.

Proste systemy powiadomień administratorów zaaplikowane w systemach, które generują duże ilości komunikatów, najczęściej się nie sprawdzają (np. [1]). Systemy takie najczęściej nie potrafią inteligentnie wybrać najważniejszych wiadomości do przekazania, zwykle ograniczając się do wysłania jednego listu elektronicznego dla każdego wygenerowanego komunikatu. Najczęściej taki system nie umie sprawdzić, czy wiadomość zosała dostarczona do adresata. Trudne jest też złączenie ze sobą w spójny sposób powiadomień generowanych z różnych systemów monitorujących.

Stąd wzięła się potrzeba wykorzystania do takich zadań systemu bardziej złożonego, który umożliwiałby łatwą współpracę wielu różnych programów do monitorowania sieci komputerowych poprzez wykorzystanie wspólnego mechanizmu powiadomień. Jednocześnie powinien on móc w pewnym zakresie decydować samodzielnie o tym, w jaki sposób i w jakim czasie najlepiej dostarczyć wiadomości. Wskazane byłoby, żeby taki system umiał wysyłać powia-

domienia na różne sposoby, tzn. nie tylko pocztą elektroniczną, ale też np. jako SMS poprzez terminal GSM lub przez bramkę WWW, jako komunikat dziennika systemowego czy poprzez umieszczenie informacji na stronie WWW. Celem mojej pracy magisterskiej jest stworzenie takiego systemu.

Praca składa się z sześciu części: szczegółowego opisu przyczyn, dla których konieczne było stworzenie nowego systemu; projektu i opisu stworzonej implementacji systemu powiadomień; opisu stworzonego przykładowego oprogramowanie klienckiego korzystającego z systemu powiadomień, napisanego w różnych językach programowania; opisu stworzonego przykładowego programu do monitorowania dostępności węzłów w sieci, wykorzystującego system powiadomień; opisu przeprowadzonych testów; podsumowania, obejmującego plan wdrożenia oprogramowania na Uniwersytecie Warszawskim oraz możliwe do wykonania w przyszłości ulepszenia i rozszerzenia.

Rozdział 1

Przyczyny stworzenia nowego systemu

Istnieje gotowe oprogramowanie, umożliwiające wysyłanie powiadomień do administratorów sieci, np. poprzez SMS ([7, 14]) czy e-mail. Istnieje też gotowe oprogramowanie monitorujące urządzenia sieciowe, np. PSI [11], SNIPS [15], JFFNMS [8], Big Sister [2], Nagios [9] i wiele innych, kontrolujących różne aspekty działania sieci komputerowej (przykłady można znaleźć np. w katalogu FreshMeat [6]).

Jednak wdrożenie wielu systemów monitorowania w połączeniu z gotowymi narzędziami do powiadomień nie jest łatwe. Różne narzędzia monitorujące wysyłają informacje o zdarzeniach na różne sposóby, przez co trudne jest złączenie ich w jeden kompleksowy system. Z kolei narzędzia do powiadomień najczęściej mają bardzo ubogą funkcjonalność, np. nie potrafią zagregować kilku komunikatów do jednej wiadomości, nie potrafią wybierać różnych adresatów i sposobów komunikacji w zależności od ważności wiadomości itd.

Kilkuletnie doświadczenia Działu Sieciowego ICM Uniwersytetu Warszawskiego, zajmującego się zarządzaniem siecią szkieletową uczelni, pokazały, że system skonstruowany z wykorzystaniem prostych aplikacji składowych ma następujące wady:

- przesyłanie każdego komunikatu w osobnej wiadomości powoduje, że administrator zasypany naraz wieloma wiadomościami (np. przy rozległych awariach sieci) bardzo często potrafi przeoczyć najważniejsze komunikaty ze względu na dużą liczbę komunikatów błahych;
- ta sama cecha powoduje wzrost kosztów funkcjonowania systemu w momencie gdy wysłanie wiadomości kosztuje (np. przy wysyłaniu SMS);
- brak zunifikowanej metody wysyłania powiadomień przez sieć powoduje konieczność budowania dość złożonych mechanizmów wykorzystujących skrypty, serwery sieciowe itp. do przesłania wiadomości z serwerów nie zajmujących się bezpośrednio wysyłaniem wiadomości;
- brak jest możliwości udostępnienia systemu wybranym osobom trzecim, które mogłyby wysyłać wiadomości do zespołu obsługi sieci poprzez system powiadomień;
- najczęściej nie jest możliwe potwierdzanie stanu wysłania wiadomości, co powoduje konieczność wysyłania wiadomości wielokrotnie w celu zapewnienia redundancji.

Stąd pojawiła się potrzeba stworzenia systemu, który posiadałby następującą funcjonalność:

- agregowanie wielu komunikatów w jedną wiadomość, z wybieraniem najważniejszych wiadomości na podstawie informacji dołączonych przez systemy wysyłające powiadomienia,
- jednolity sposób wysyłania wiadomości zgłaszanych przez różne serwery,
- uwierzytelnianie systemów wysyłających wiadomości,
- możliwość nadawania użytkownikom różnych uprawnień, pozwalając na wysyłanie wiadomości tylko do wskazanych odbiorców,
- współdziałanie kilku serwerów powiadomień, mające na celu zapewnienie możliwe najwyższego poziomu niezawodności dostarczania komunikatów.

Rozdział 2

Rozproszony system powiadamiania administratorów

2.1. Przyjęte nazewnictwo

odbiorca – osoba, do której przesyłane są wiadomości, np. Jan Kowalski, lub grupa odbiorców.

przetłumaczenie nazwy odbiorcy – przetłumaczenie nazwy grupy odbiorców na listę nazw odbiorców skonfigurowanych w tej grupie.

kanał – kanał transmisji wiadomości, np. e-mail lub SMS.

cel – para (osoba, kanał), tzn. kanał z zadeklarowanym adresem odbiorcy, np. „telefon Jana Kowalskiego", rozwijany np. do „sms: 501234567".

komunikat – informacja, którą należy przekazać odbiorcy, np. „serwer X nie działa".

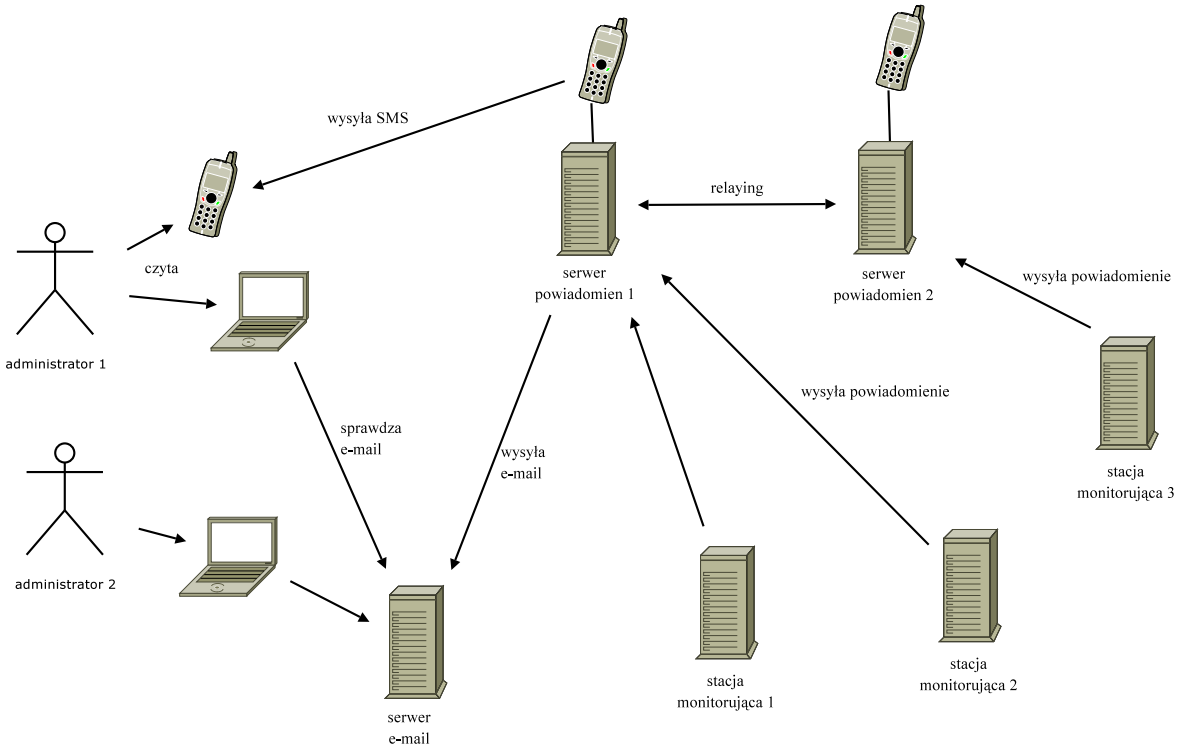
wiadomość – treść powiadomienia, którą osoba otrzymuje przez kanał. Może składać się z kilku komunikatów, może też zawierać tylko informacje o komunikatach, np. „otrzymałeś innym kanałem 3 komunikaty ważne, 5 zwykłych"

przekazanie do wysłania (ang. *relaying*) – wysłanie wiadomości natychmiast, bez wstawiania jej do kolejki. Najczęściej wiadomość jest przekazywana przez serwer na prośbę innego serwera, któremu nie udało się samodzielnie wysłać wiadomości.

2.2. Założenia systemu

W pierwszej fazie projektu przyjęto, że system powinien posiadać następujące właściwości:

- możliwość wysyłania powiadomień przez różne kanały transmisji (e-mail, SMS, systemy IM),
- modularna budowa związana z kanałami transmisji (możliwość definiowania parametrów per kanał i dodawania nowych kanałów),
- wysyłanie wiadomości do pojedynczych odbiorców lub grup odbiorców,
- określenie maksymalnego czasu, po jakim powinno nastąpić wysłanie wiadomości,



Rysunek 2.1: Ogólny model planowanego systemu powiadomień

- określenie poziomu ważności komunikatu,
- możliwość wysłania jednej wiadomości w kilku wersjach (np. inna treść w liście elektronicznym, inna w SMS),
- przydzielanie wiadomościom identyfikatorów, na podstawie których nadawca może się dowiedzieć o stan wiadomości,
- szyfrowanie komunikacji między klientem i serwerem,
- uwierzytelnianie użytkownika na podstawie hasła, adresu bądź innych danych,
- agregowanie wielu wiadomości do jak najmniejszej liczby komunikatów.

Ogólny model funkcjonowania systemu przedstawiono na rysunku 2.1. Przy projektowaniu pojawiały się jeszcze inne założenia (np. możliwość dołączania dowolnych danych do każdej wiadomości, które mogłyby być wykorzystywane przez jakieś dodatkowe aplikacje), które zostały odrzucone już na etapie projektowania oprogramowania jako mało istotne.

2.3. Projekt implementacji

W projekcie przyjęto następujące założenia implementacyjne:

- wykorzystanie Perla jako języka implementacji, ze względu na przenośność oraz dostępność licznych bibliotek, np. do obsługi XML-RPC, wiadomość, której nie udało się wysłać z jednego serwera, będzie przekazana do innego serwera. SSL, SQL itp. [4],

- modularne elementy systemu mają być pisane w sposób obiektowy,
- wykorzystanie XML-RPC jako metody komunikacji klientów z serwerem. Protokół XML-RPC jest dobrze udokumentowany (np. [20, 16, 19]), oparty na bardzo popularnym protokole HTTP, prosty w obsłudze,
- do szyfrowania komunikacji i jako opcjonalną metodę uwierzytelniania wybrano protokół SSL,
- założono, że dane będą przechowywane w bazie SQL. Wykorzystany ma być moduł Perla DBI do zapewnienia abstrakcji warstwy bazy danych. Dobrym silnikiem bazodanowym do tego projektu może się okazać SQLite.
- z przesyłaną wiadomością mają być związane następujące atrybuty:
  - nadawca,
  - czas nadania wiadomości,
  - priorytet, określający pośrednio czas jaki wiadomość może czekać w kolejce,
  - waga, określający kolejność wysyłania lub agregowania wiadomości oraz ew. umożliwiający poszerzenie/zawężenie grona odbiorców wiadomości,
  - zbiór treści (kilka różnych wariantów tego samego komunikatu),
  - informacja o tym czy wiadomość ma być przekazywana do innego serwera w razie niemożliwości jej wysłania,
  - identyfikator nadawany przez serwer, umożliwiający sprawdzenie aktualnego stanu wiadomości.
- z kanałem transmisji mają być powiązane atrybuty:
  - przełożenie abstrakcyjnych wartości priorytetów wiadomości na konkretne czasy oczekiwania (różne dla różnych kanałów),
  - sposób agregacji wiadomości, uwzględniający maksymalną długość komunikatu,
  - klasa wysyłanej wiadomości (wskazanie jednej z kilku dostępnych treści),
- obsługa niektórych kanałów powinna być możliwa do zrealizowania w formie osobnego serwera, ze względu np. na wymaganą wyłączność w dostępie do sprzętu wysyłającego wiadomości lub na kosztowną inicjację wysyłania,
- komunikacja między serwerami powiadomień powinna odbywać się z wykorzystaniem tego samego sposobu dostępu (XML-RPC),
- założono dostępność następujących metod serwera XML-RPC:
  - `powiadomienia.send` – do wysyłania wiadomości,
  - `powiadomienia.relay` – do przekazywania wiadomości do innego serwera,
  - `powiadomienia.explain` – do odpytywania o konfigurację powiadomień,
  - `powiadomienia.status` – do sprawdzania stanu wysłanej wiadomości,
  - `powiadomienia.version` – do sprawdzenia wersji serwera.

  Serwer dodatkowo udostępnia standardowe metody serwera `XML::RPC::Server`, np. informacyjne `system.methodHelp` lub `system.methodSignature`.

- założono, że wiadomość może być w jednym z następujących stanów:
  - `NIE_PROBOWANO_WYSLAC` – serwer nie próbował wysłać wiadomości, ponieważ np. odbiorca nie istnieje, użytkownik nie ma prawa wysyłać wiadomości itp.
  - `CZEKA_NA_PRZETWORZENIE` – wiadomość została umieszczona na serwerze, jednak jeszcze w żaden sposób nie była analizowana. Ten stan jest widoczny jedynie wewnątrz serwera, klienci odpytujący o wiadomość w takim stanie otrzymają odpowiedź `OCZEKUJE_W_KOLEJCE`.
  - `OCZEKUJE_W_KOLEJCE` – wiadomość została przetworzona przez serwer, rozdzielona na poszczególnych odbiorców i wstawiona do kolejki w oczekiwaniu na agregację i wysłanie.
  - `NIE_WYSLANO` – nie udało się wysłać wiadomości (np. sieć GSM odmówiła przesłania SMS).
  - `WYSLANO`, `WYSLANO_NIE_DOSTARCZONO` – wiadomość została wysłana. Pierwszy komunikat ma być przekazywany w sytuacji, w której wiadomo, że kanał nie jest w stanie potwierdzić dostarczenia wiadomości (np. w wypadku wysyłania poczty elektronicznej za pomocą systemowej funkcji `mail()`). Drugi ma być przekazywany, gdy kanał jest w stanie potwierdzić doręczenie wiadomości, ale jeszcze tego nie zrobił.
  - `DOSTARCZONO`, `DOSTARCZONO_NIE_PRZECZYTANO` – analogicznie, wiadomość została dostarczona, w zależności od funkcjonalności kanału nie można będzie stwierdzić czy została przeczytana lub będzie można, ale kanał jeszcze nie potwierdził przeczytania.

Duża część projektu była precyzowana dopiero w trakcie implementacji.

2.4. Implementacja

W trakcie implementacji pojawiały się wielokrotnie problemy z wyborem właściwego rozwiązania, bądź z zaimplementowaniem konkretnej właściwości systemu. W kolejnych punktach spisane są najważniejsze problemy występujące w trakcie implementacji oraz najważniejsze podjęte decyzje.

2.4.1. Serwer XML-RPC z SSL

Dla Perla dostępna jest gotowa implementacja serwera XML-RPC o nazwie `RPC::XML` [13], z wykorzystaniem dodatkowo modułu `Net::Server` jako silnika obsługującego połączenia sieciowe, zapewniającego równoległość obsługi żądań. Jednak nie obsługuje ona wprost komunikacji szyfrowanej po SSL. Stąd pojawił się problem: czy budować serwer ze zintegrowaną obsługą SSL, czy wykorzystać wprost `RPC::XML::Server` i dodatkowo wykorzystać stunnel jako silnik SSL.

Wadą rozwiązania wykorzystującego stunnel jest utrata w aplikacji informacji dostępnych tylko na poziomie połączenia SSL: adres IP oraz opcjonalny certyfikat klienta. Utrudnia to diagnostykę połączeń (trzeba korelować logi stunnela z logami aplikacji) oraz nie pozwala wykorzystać certyfikatów SSL do uwierzytelniania klienta. Dlatego ostatecznie wybrany został serwer SSL wbudowany w aplikację.

Rozważono następujące możliwości:

- wykorzystanie bezpośrednio `Net::Server` z protokołem SSL. Ponieważ jednak moduł `Net::Server::Proto::SSL` obsługuje tylko jedno połączenie SSL naraz, rozwiązanie okazało się nie do przyjęcia.

- wykorzystanie zewnętrznego serwera WWW z obsługą SSL (np. Apache) i budowa serwera w formie aplikacji webowej np. w PHP albo jako CGI. Jednak system zbudowany w ten sposób jest zbyt złożony jak na potrzeby powiadomień. Ta opcja została potraktowana jako ostateczność.

- przerobienie modułu `RPC::XML::Server` tak, aby korzystał z połączeń SSL.

Moduł `RPC::XML::Server` wewnętrznie korzysta z modułu `HTTP::Daemon`. Istnieje implementacja `HTTP::Daemon::SSL`, zachowująca interfejs serwera, a korzystająca z połączeń SSL. Po przeanalizowaniu kodu serwera XML-RPC okazało się, że nie jest nawet konieczne przepisywanie modułu, a tylko umiejętne skorzystanie z udostępnianych funkcji. Szczególnie użyteczna okazała się funkcja `RPC::XML::Server::process_request()`, która jako argument przyjmuje deskryptor połączenia typu `HTTP::Daemon::ClientConn`.

Ostatecznie w implementacji wykorzystano moduł `HTTP::Daemon::SSL`, który dla każdego otwartego połączenia przekazuje jego deskryptor do funkcji `process_request()`. Łączność sieciowa jest w tym rozwiązaniu obsługiwana przez moduł `IO::Socket::SSL`, równoległość żądań zapewnia `Net::Server`, `HTTP::Daemon::SSL` wiąże te klasy razem oraz zapewnia obsługę protokołu HTTP, a `RPC::XML::Server` obsługuje przyjęte przez `HTTP::Daemon:SSL` żądania.

Podczas implementacji zostały znalezione dwa błędy w module `IO::Socket::SSL` w wersji 0.97:

- założenie, że otwarty deskryptor jest większy od 0. W wypadku serwera powiadomień, ponieważ wszystkie standardowe deskryptory wejścia/wyjścia są zamykane przy odłączaniu się od terminala, pierwszym nowo otwieranym był właśnie 0. Ten błąd występował również w starszych wersjach `IO::Socket::SSL`, dlatego w kodzie zostało wprowadzone obejście, polegające na sztucznym otwarciu i utrzymywaniu deskryptora 0 wskazującego na `/dev/null`,

- w wersji 0.97 `IO::Socket::SSL` zmienił semantykę funkcji read(), która zaczęła wykorzystywać `Net::SSLeay::ssl_read_all()`. W efekcie `HTTP::Daemon::SSL` nie działał poprawnie (wczytywanie żądania nigdy się nie kończyło). Autor modułu został powiadomiony o problemie (*http://rt.cpan.org/Public/Bug/Display.html?id=19705*) i w kolejnych wersjach ten problem został rozwiązany.

Główny proces serwera tworzy pewną liczbę procesów nasłuchujących (ang. *listener*), po czym nadzoruje ich działanie, tworząc nowe procesy gdy ich liczba spadnie poniżej zadanego poziomu (tryb wstępnego tworzenia procesów potomnych, ang. *prefork*). Jeśli do systemu przyjdzie więcej żądań jednocześnie niż jest procesów nasłuchujących, to klient będzie musiał czekać aż jeden z nich skończy obsługę poprzedniego żądania.

2.4.2. Konfiguracja

Początkowo dopuszczono było kilka metod zapisu konfiguracji:

- konfiguracja zapisana w bazie danych. Jednak już do połączenia z bazą danych niezbędne byłoby posiadanie jakiejś konfiguracji. Dobrze byłoby mieć wcześniej włączone logi, a to też wymaga konfiguracji. Można zrobić konfigurację dzieloną (część w pliku lub

wierszu poleceń, część w bazie danych), jednak w takim wypadku wymaga to więcej pracy niż zrobienie konfiguracji wprost w pliku tekstowym.

- konfiguracja w pliku tekstowym, na wzór plików plików konfiguracyjnych Apache, z wykorzystaniem modułu `Config::General`. Składnia takich plików jest wygodna do edycji przez administratora. Wady: sprawdzanie poprawności składni musi być zaimplementowane na poziomie kodu.
- konfiguracja w pliku XML. Takie pliki są mniej wygodne w edycji, jednak budowa XML zapewnia przynajmniej częściowe automatyczne sprawdzanie składni z wykorzystaniem plików DTD.

Ostatecznie, ponieważ w przyszłości planowane jest stworzenie interfejsu do zarządzania systemem powiadomień z poziomu aplikacji WWW, został wybrany format XML. Do implementacji użyto modułu `XML::Simple`, a dodatkowo do sprawdzania poprawności z DTD modułu `XML::LibXML`.

Obsługa plików konfiguracyjnych w całości została wydzielona do modułu `Powiadomienia::Config`, dzięki czemu ew. zmiana na format tekstowy powinna być stosunkowo prosta, jako że jedynym wymaganiem jest zbieżność generowanej struktury z obecnie używaną.

2.4.3. Moduły kanałów

Aby uczynić architekturę kanałów powiadomień modularną, został ustalony interfejs implementacji kanału. Kanał może pracować w jednym z dwóch trybów: synchronicznym, gdzie wysyłanie wiadomości jest wołane z głównego procesu serwera, a proces oczekuje aż wysyłanie się zakończy; lub asynchronicznym, gdzie proces główny tylko sygnalizuje jednemu z procesów potomnych konieczność wysłania wiadomości, a sam wraca natychmiast do obsługi kolejki i innych wiadomości. Ogólna klasa `Powiadomienia::Kanal` jest wspólnym opakowaniem dla poszczególnych kanałów, obsługującym oba tryby pracy. Wszystkie moduły kanałów (`Kanaly::*`) są ładowane dynamicznie na podstawie konfiguracji.

2.4.4. Komunikacja między procesami i synchronizacja procesów

Projekt zakładał, że niektóre części serwera muszą być samodzielnymi procesami. Wynikła z tego kwestia sposobu organizacji komunikacji między procesami. Ostatecznie założono, że przesyłanie danych między procesami będzie się odbywać poprzez bazę danych. Aby uniknąć aktywnych oczekiwań na bazie danych lub opóźnień w przetwarzaniu żądań, procesy dodatkowo komunikują się za pomocą sygnałów POSIX, głównie `SIGALRM`.

W trakcie implementacji pojawił się problem z synchronizacją procesów zajmujących się obsługą kanałów transmisyjnych, związany z koniecznością ustawienia obsługi sygnałów przed rozpoczęciem komunikacji międzyprocesowej. Przyjęto rozwiązanie polegające na utworzeniu pary gniazd połączonych potokiem, do którego proces potomny kanału zapisuje pojedynczy znak w momencie gdy zakończy inicjację. Dopiero po odczytaniu tego znaku proces macierzysty kończy procedurę tworzenia potomka.

2.4.5. Uwierzytelnianie użytkowników i uprawnienia

Pierwotnym pomysłem na uwierzytelnianie użytkowników było skorzystanie z nazw użytkowników i haseł. Ostatecznie koncepcja ta została zachowana, a dodatkowo rozszerzona o uwierzytelnianie automatyczne – na podstawie danych zawartych w certyfikacie klienta SSL, o ile taki był wykorzystany, oraz na podstawie adresu IP. Wszystkie metody serwera

dostępne są w dwóch wersjach, różniących się liczbą argumentów – w dłuższej wersji dodatkowo przekazywana jest tablicą z nazwą użytkownika i hasłem. Użytkownik nie posługujący się certyfikatem, łączący się z nieznanego adresu IP i posługujący się krótszą sygnaturą metody, ma przypisywane uprawnienia `anonymous`.

Konfiguracja praw dostępu do wołania metod została zrobiona na wzór konfiguracji linuksowego IPtables – kolejne wpisy w konfiguracji przeglądane są sekwencyjnie, dla każdego wpisu może być określona nazwa użytkownika, operacja serwera, argumenty operacji (wszystko opcjonalnie) oraz obowiązkowo rodzaj uprawnienia (dozwolone/niedozwolone).

2.4.6. Wagi i priorytety

We wstępnej fazie projektowania nie było rozróżnienia między wagą i priorytetem. Jednak pojawiła się konieczność rozdzielenia dwóch funkcjonalności – maksymalnej długości czasu jaki wiadomość może spędzać w kolejce (wpływa ona na skuteczność agregacji – jeśli komunikat może dłużej leżeć w kolejce, to będzie można upakować więcej komunikatów w pojedynczej wiadomości) oraz sortowania zagregowanych wiadomości wg ich ważności. Stąd określenia *priorytet*, które intuicyjnie jest związane z czasem dostarczenia, oraz *waga*, które mówi dokładnie o wadze wiadomości.

System priorytetów i wag został tak zaprojektowany, aby ich liczba nie była ograniczona i była możliwość ustalenia tylu różnych priorytetów przy wdrożeniu, ile tylko będzie potrzebnych. Jednak implementacje kanałów mogą być zależne od maksymalnej liczby priorytetów (np. po to, aby wstawić w wiadomości tekst „ważne" przy właściwym priorytecie). Dlatego na potrzeby przykładów konfiguracji i implementacji kanałów przyjęto następującą skalę:

priorytet 1: nieistotne, ..., 3: zwykłe, ..., 5: bardzo pilne

waga 1: nieważne, ..., 3: zwykłe, ..., 5: bardzo ważne

Dla każdego kanału z osobna definiuje się maksymalny czas oczekiwania komunikatu związany z poszczególnymi priorytetami. Dodatkowo jest możliwe wykorzystanie wagi do zmiany listy odbiorców wiadomości oraz ograniczania wysyłania wiadomości zadanymi kanałami: np. wiadomość z wagą 5 może być wysłana do większej liczby odbiorców albo większą liczbą kanałów do każdego odbiorcy niż wiadomość z wagą 3.

2.4.7. Kanały i odbiorcy

Początkowo kanał miał być specjalnym typem odbiorcy. W pierwotnej wersji projektu wiadomość mogła być przeznaczona dla grupy jednoznacznie nazwanych kanałów.

Jednak w trakcie implementacji okazało się że dobrym sposobem jest logiczne połączenie kanałów należących do fizycznych osób w większe grupy, z wybieraniem kanałów do wykorzystania na podstawie parametrów wiadomości. Odbiorca zmienił się w grupę odbiorców lub grupę kanałów. Serwer decyduje samodzielnie, które kanały zostaną wykorzystane (w pierwotnej wersji to klient decydował jakimi kanałami wysłać).

W dalszym ciągu jest możliwe zaimplementowanie pierwszej wersji organizacji odbiorców – poprzez tworzenie znacząco nazwanych odbiorców, z których każdy będzie miał przypisany tylko jeden, używany bezwarunkowo kanał.

W systemie każdy kanał reprezentowany jest jako obiekt, będący rozszerzeniem klasy `Powiadomienia::Kanal`, implementującym tylko niektóre metody prywatne. Schemat działania procesów kanałów jest z góry narzucony przez klasę `Powiadomienia::Kanal`.

2.4.8. Kolejkowanie i agregacja

Każda wiadomość w czasie przetwarzania rozdzielana jest na poszczególnych odbiorców, a następnie na kanały. W kolejce do wysłania pojawiają się trójki (odbiorca, kanał, identyfikator komunikatu), z zapisanymi obok treściami odpowiadającymi poszczególnym identyfikatorom przyznanym komunikatom przy ich wysyłaniu.

W momencie, gdy minie czas wysłania dowolnej wiadomości z kolejki, uruchamiany jest odpowiadający jej kanał. Kanał zbiera w jednym przebiegu wszystkie komunikaty do przekazania do pojedynczego odbiorcy, po czym wykonuje na nich operację agregacji. Wynikiem agregacji jest wstawienie do listy wychodzących wiadomości gotowych, skompilowanych komunikatów oraz ustawienie przy każdym komunikacie informacji, w której wiadomości komunikat został przekazany.

Sposób agregacji jest specyficzny dla każdego z kanałów. Np. kanał `sql`, który wstawia otrzymane komunikaty do bazy, może w ogóle nie agregować treści, zakładając że być może zrobi to aplikacja prezentująca wiadomości z bazy, jeśli będzie to jej potrzebne. Z kolei wiadomość zagregowana przez kanał `sms` z konieczności będzie dużo krótsza, niż wiadomość wysyłana kanałem `mail`, mimo iż obie obejmują te same komunikaty.

Ponieważ wyboru wiadomości do wysłania i agregacji dokonuje się dla każdego kanału, może się zdarzyć, że ten sam odbiorca otrzyma ten sam komunikat różnymi kanałami w zupełnie różnych momentach, np. SMS natychmiast, a pocztą elektroniczną dopiero po godzinie.

2.4.9. Logowanie

W założeniu system miał umożliwiać logowanie komunikatów diagnostycznych co najmniej na 3 sposoby: na standardowe wyjście (o ile nie pracuje w trybie asynchronicznym), do pliku z logiem oraz do interfejsu syslog. Wskazane było rozróżnianie rodzajów komunikatów (błąd, ostrzeżenie, informacja itp.)

Początkowo wykorzystywana była autorska implementacja, jednak stosunkowo szybko została ona zmieniona na moduł `Log::Log4perl`, wzorujący się na znanym komponencie Javy: `log4java`. Założono że logowanie będzie albo do pliku, albo do logów systemowych (rozłącznie), a logowanie na standardowe wyjście jest automatycznie włączane w zależności od trybu pracy serwera.

2.4.10. Obsługa bazy danych

W trakcie implementacji do testów wykorzystywany był silnik MySQL pracujący pod modułem `DBI`. Sposób dostępu do bazy danych został zapisany w konfiguracji tak, aby było możliwe zmienienie silnika bazodanowego bez potrzeby zmiany kodu. Też ze względu na przenośność zapytania wykonywane przez serwer nie korzystają z żadnych funkcji specyficznych dla konkretnego serwera bazodanowego.

Przez pewien czas odstępstwem od tego było korzystanie z pól typu datetime w bazie, jednak okazało się, że `DBI` nie zapewnia odpowiednio uogólnionego mechanizmu do operowania na datach zapisanych w bazie, dlatego ostatecznie w bazie znajdują się jedynie znaczniki czasu w formacie Unix (liczba sekund od 01.01.1970), a wszystkie operacje na datach wykonywane są w Perlu.

Szczególnym wymaganiem jest poprawne działanie funkcji `DBI::last_insert_id()` – stąd konieczność wykorzystania silnika bazodanowego który potrafi podać identyfikator ostatnio wstawionego wiersza (MySQL, PostgreSQL i SQLite w wersji 3 spełniają ten warunek) oraz wykorzystania `DBI` w wersji co najmniej 1.38 (starsze wersje nie posiadały tej funkcji).

W trakcie testów pojawił się problem z obsługą współdzielonych uchwytów do bazy danych. Wszystkie obiekty Kanal posiadają zapisany w sobie atrybut `DBH`, który jest uchwytem do bazy danych na którym wykonywane są operacje bazodanowe. Kanały synchroniczne, aby nie mnożyć połączeń do bazy, przechowywały wszystkie jeden i ten sam uchwyt, współdzielony dodatkowo z zarządcą kolejki. W momencie gdy jakiś kanał był przeładowywany (np. przy wczytywaniu zmienionej konfiguracji), serwer usuwał wszystkie kanały z pamięci. Usunięcie z pamięci kanału powodowało wołanie perlowego destruktora `DESTROY` na wszystkich jego atrybutach. Z kolei uchwyt bazy danych w swoim destruktorze miał zapisane odłączenie się od bazy. W efekcie zarządca kolejki tracił połączenie z bazą.

Rozwiązaniem okazało się wydzielenie obsługi współdzielonego połączenia bazy danych w module `Powiadomienia::DB` i zmiana destruktora tego jednego uchwytu tak, aby nie odłączał bazy danych, poprzez ustawienie opcji `InactiveDestroy`.

Zupełnie inny problem wynikł w trakcie testów z SQLite. Ze względu na wykorzystanie tej samej bazy przez różne procesy (np. procesy kanałów) konieczne jest poprawne obsługiwanie równoległych zapytań. Niestety SQLite obsługuje równoległe wykonywanie zapytań (zwłaszcza równoczesne operacje `SELECT` z `INSERT`) w sposób dość ograniczony, blokując na czas `INSERT` całą tabelę, i nie pozwalając żadnemu innemu klientowi w tym czasie czytać z tabeli. Próba wykonania równoległego zapytania kończy się błędem z informacją, że tabela jest zablokowana. Dlatego okazuje się, że SQLite nie nadaje się obecnie do pracy z wielowątkową aplikacją.

Możliwym rozwiązaniem jest przebudowa mechanizmu dostępu do bazy danych. Zapytania wykonywane przez serwer są dość proste, dlatego prawdopodobnie wystarczyłoby, gdyby jeden proces zajmował się koordynacją odwołań do bazy danych. Pozostałe procesy mogłyby przekazywać do niego treść zapytań do wykonania, a następnie odczytywać odpowiedzi.

Ponieważ obecnie niektóre fragmenty kodu zawierają wywołania, które operują w kontekście połączenia (np. `DBI::last_insert_id()`), niezbędne byłoby zaimplementowanie całych ciągów zapytań w serwerze, na wzór procedur osadzonych w bazie danych. Wykonanie zapytania byłoby de facto wywołaniem procedury serwera pośredniczącego do bazy danych.

Implementacja mogłaby być oparta np. na kolejkach komunikatów IPC Systemu V, gniazdach uniksowych lub gniazdach TCP/IP. Ze względu na już istniejącą obsługę XML-RPC w serwerze, przekazywane wywołania procedur i odpowiedzi mogłyby być właśnie w formacie XML-RPC.

Jednak obecnie jedynym przetestowanym i poprawnie działającym z aplikacją silnikiem bazodanowym pozostaje MySQL. Prawdopodobnie większe bazy, np. PostgreSQL czy Oracle, też będą nadawały się do tego celu, jednak wykorzystywanie ich do obsługi tak małego serwisu wydaje się niecelowe.

2.5. Opis publicznych interfejsów modułów i klas

Metody niepubliczne w klasach i modułach były wyróżniane znakiem _ (podkreślenie) na początku nazwy.

`Powiadomienia::Auth` MODUŁ

`auth_pass()` Argumenty: referencja do tablicy z nazwą użytkownika i hasłem. Przekazuje: nazwa użytkownika lub `anonymous`.

`auth_auto()` Argumenty: referencja do obiektu `HTTP::Daemon::ClientConn`. Przekazuje: nazwa użytkownika lub `anonymous`.

`Powiadomienia::Config` MODUŁ

`loadConfig()` Argumenty: brak. Przekazuje: 1 jeśli udało się wczytać konfigurację, wpp. przekazuje 0 lub kończy działanie procesu (jeśli było to pierwsze wczytanie konfiguracji).

`$config` drzewo aktualnej konfiguracji (referencja do tablicy asocjacyjnej).

`$configfile` nazwa pliku z konfiguracją w formacie XML.

`$dtdfile` nazwa pliku z DTD opisującym konfigurację.

`Powiadomienia::DB` MODUŁ

`sharedConnect()` Argumenty: brak. Przekazuje: uchwyt współdzielonego połączenia do bazy danych, tworząc go jeśli jeszcze nie istniał.

`sharedDisconnect()` Argumenty: brak. Przekazuje: brak. Wymuszone zakończenie połączenia na współdzielonym uchwycie.

`connect()` Argumenty: brak. Przekazuje: uchwyt do nowo otwartego połączenia do bazy danych.

`Powiadomienia::Kanal` KLASA

`new()` konstruktor. Argumenty: nazwa kanału oraz interwał okresowego sprawdzania kolejki (w sekundach). Jeśli interwał jest równy 0, to kanał będzie pracował w trybie synchronicznym.

`respawn()` Argumenty: brak. Przekazuje: brak. Wołane w momencie gdy proces macierzysty stwierdzi, że proces potomny nieoczekiwanie zmarł.

`run()` Argumenty: brak. Przekazuje: 1 jeśli uruchomienie kanału się udało, 0 jeśli nie. Uruchomienie kanału polega na zagregowaniu i wysłaniu wiadomości w wypadku kanałów synchronicznych lub przekazaniu sygnału do procesu obsługi w wypadku kanałów asynchronicznych.

`stop()` Argumenty: brak. Przekazuje: brak. Kończy działanie procesu obsługi kanału.

`pid()` Argumenty: brak. Przekazuje: PID procesu obsługującego kanał, jeśli taki jest, 0 jeśli proces nie działa, lub -1 jeśli kanał pracuje w trybie synchronicznym

`Powiadomienia::ListenerMethods` MODUŁ

`init_db()` Argumenty: brak. Przekazuje: brak. Przygotowuje zapytania do wykonania w bazie danych. Każda instancja procesu nasłuchującego musi wywołać tę metodę po otwarciu połączenia do bazy danych.

`send()` Argumenty: lista odbiorców w formacie tablicy XML-RPC, priorytet, waga, zbiór treści komunikatu w formacie tablicy asocjacyjnej (struct) XML-RPC, czy przekazywać wiadomość do wysłania do innych serwerów w razie niepowodzenia. Przekazuje: identyfikator przydzielony komunikatowi lub ujemny kod błędu. Obecnie wykorzystywane są kody błędów: `-400` – nieprawidłowe żądanie (brak domyślnej treści komunikatu), `-401` – nieprawidłowa nazwa użytkownika, nieprawidłowe hasło lub brak uprawnień.

`status()` Argumenty: identyfikator komunikatu. Przekazuje: listę odbiorców, przetłumaczonych stosownie do uprawnień pytającego użytkownika oraz wagi oryginalnego komunikatu. Dla każdego elementu listy dodatkowo przekazuje maksymalną wartość statusu wysłania komunikatu wśród odbiorców. Patrz `Powiadomienia::Router::maxStatus()`.

`relay()` Argumenty: brak. Przekazuje: kod błędu `-501`. Funkcja miała służyć do implementacji przesyłania wiadomości między serwerami, jednak ostatecznie nie została wykorzystana.

`explain()` Argumenty: nazwa odbiorcy, waga komunikatu. Przekazuje: listę odbiorców, będącą wynikiem przetłumaczenia podanej nazwy z wykorzystaniem zadanej wagi komunikatu, stosownie do uprawnień pytającego użytkownika.

`Powiadomienia::Listener` KLASA

`new()` konstruktor. Argumenty: brak. Tworzy procesy potomne, które w pętli oczekują na połączenie SSL na zadanym w konfiguracji porcie, a potem obsługują żądanie.

`hasPID()` Argumenty: PID procesu. Przekazuje: 1 jeśli jeden z procesów potomnych ma taki PID, 0 jeśli żaden.

`listenerKilled()` Argumenty: PID procesu. Przekazuje: brak. Wywoływane w momencie gdy proces macierzysty zauważy że jeden z procesów nasłuchujących zakończył działanie lub zginął.

`respawnDead()` Argumenty: brak. Przekazuje: brak. Wywołuje kolejne procesy potomne tak aby ich liczba w sumie wynosiła tyle, ile podano w konfiguracji.

`shutdown()` Argumenty: brak. Przekazuje: brak. Zamyka wszystkie nasłuchujące procesy potomne.

`Powiadomienia::Logging` MODUŁ

`init()` Argumenty: brak. Przekazuje: brak. Inicjuje system logowania `Log4perl` zgodnie z konfiguracją.

`logger()` Argumenty: brak. Przekazuje: wartość wywołania funkcji `Log::Log4perl::-get_logger()` dla pustej kategorii (`Log4perl` umożliwia definiowanie różnych logów dla różnych kategorii. Opisywany system wykorzystuje tylko jedną, domyślną kategorię pustą).

`close()` Argumenty: brak. Przekazuje: brak. Zamyka pliki z logami.

`Powiadomienia::QMgr` KLASA

`new()` konstruktor. Argumenty: brak. Tworzy nową pustą instancję.

`init()` Argumenty: brak: Przekazuje: brak. Przygotowuje bazę danych.

`run()` Argumenty: brak. Przekazuje: referencja do listy kanałów, które wymagają uruchomienia. Wstawia nowe wiadomości do kolejki.

`czas_oczekiwania()` Argumenty: brak. Przekazuje: zalecany czas oczekiwania w sekundach na kolejny przebieg zarządcy kolejki. Z góry ograniczony do 60 sekund.

`Powiadomienia::Router` MODUŁ

`canSend()` Argumenty: nazwa użytkownika, nazwa odbiorcy. Przekazuje: 1 jeśli użytkownik jest uprawniony do wysyłania wiadomości do tego odbiorcy, wpp. 0.

`canExplain()` Argumenty: nazwa użytkownika, nazwa grupy lub odbiorcy. Przekazuje: 1 jeśli użytkownik jest uprawniony do przetłumaczenia nazwy grupy na elementy składowe pierwszego rzędu, wpp. 0. Nazwa odbiorcy nie będącego grupą jest traktowana jak jednoelementowa grupa.

`canRelay()` Argumenty: nazwa użytkownika, nazwa odbiorcy. Przekazuje: 1 jeśli użytkownik jest uprawniony do przekazywania wiadomości adresowanych do zadanego odbiorcy, wpp. 0.

`explain()` Argumenty: nazwa użytkownika, nazwa odbiorcy, waga. Przekazuje: referencję do przetłumaczonej, zgodnie z uprawnieniami użytkownika, listy odbiorców przy zadanej wadze komunikatu.

`kanaly()` Argumenty: nazwa odbiorcy, waga. Przekazuje: referencję do listy kanałów, którymi powinien być przekazany do odbiorcy komunikat o zadanej wadze.

`maxStatus()` Argumenty: referencja do listy tekstowych statusów wiadomości. Przekazuje: maksymalny status, zgodnie z porządkiem:
`NIE_PROBOWANO_WYSLAC` < `OCZEKUJE_W_KOLEJCE`
< `NIE_WYSLANO` < `WYSLANO` < `WYSLANO_NIE_DOSTARCZONO`
< `DOSTARCZONO` < `DOSTARCZONO_NIE_PRZECZYTANO`
< `PRZECZYTANO`

`Powiadomienia` APLIKACJA

`prestart()` Argumenty: brak. Przekazuje: brak. Inicjuje konfigurację i logi.

`start()` Argumenty: brak. Przekazuje: brak. Inicjuje współdzielone połączenie bazy danych, inicjuje i uruchamia kanały, uruchamia zarządcę kolejki, uruchamia procesy nasłuchujące.

`stop()` Argumenty: brak. Przekazuje: brak. Zatrzymuje i czyści kanały, zarządcę kolejki, procesy nasłuchujące, połączenie do bazy danych.

`RELOAD()` obsługa sygnału `HUP`. Przeładowuje cały serwer.

`REAPER()` obsługa sygnału `CHLD`. Sprawdza jaki proces zmarł i ponownie go uruchamia.

2.6. Schemat działania głównego procesu serwera

Główny proces serwera po wykonaniu czynności związanych z wczytaniem konfiguracji, stworzeniem wewnętrznych struktur danych i początkowym stworzeniem procesów potomnych (nasłuchujących oraz obsługujących kanały asynchroniczne) zajmuje się:

- przywracaniem do życia procesów potomnych, które zakończyły swoje działanie. Obejmuje to utrzymywanie liczby działających procesów nasłuchujących na stałym poziomie,
- przetwarzaniem nowych komunikatów (tłumaczeniem nazw odbiorców, ustalaniem terminów wysyłki),
- wysyłaniem wiadomości kanałami pracującymi w trybie synchronicznym,
- sygnalizowaniem procesom, obsługującym kanały w trybie asynchronicznym, konieczności wysłania wiadomości

Schemat działania głównego procesu serwera można zapisać następująco (pseudokod w Perlu):

```
prestart();
if (ma_byc_daemon()) {
        przejscie_w_tryb_daemon();
```

```
}
ustawienie_obslugi_sygnalow();
start();
while (1) {
        $listener->respawnDead();
        my $listakanalow = $qmgr->run();
        foreach my $kanalname (@{$listakanalow}) {
                $kanaly{$kanalname}->run();
        }
        sleep $qmgr->czas_oczekiwania();
}
```

2.7. Prezentacja struktur w działającym serwerze

Na schemacie 2.2 niektóre elementy struktur zostały celowo pominięte, a niektóre zostały dodane, tak aby ułatwić zrozumienie struktury działającego serwera. Nazwy klas zostały zapisane prostą czcionką, nazwy modułów pochyłą.

W wypadku kanałów pracujących w trybie asynchronicznym istnieją dwie instancje klasy `Powiadomienia::Kanal`: jedna w procesie głównym, która zawiera informacje kontrolne dla procesu potomnego i w której wołane są metody typu `run()`, `start()`, `stop()`, oraz jedna w procesie potomnym, która chodzi cały czas w pętli o schemacie:

```
sub main_loop {
        while (1) {
                _send_all();
                _check_status();
                sleep;
        }
}
```

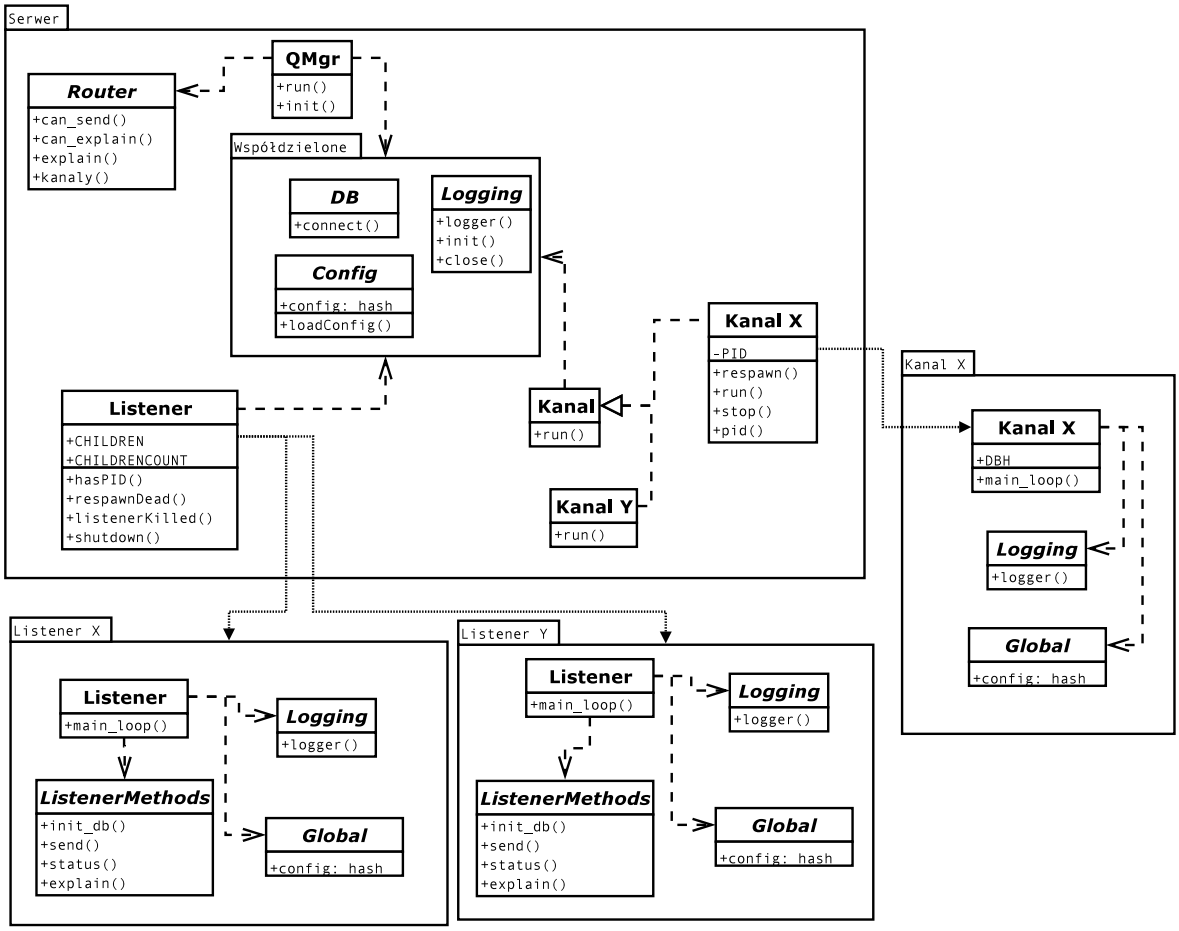
Każdy z procesów nasłuchujących ma własną instancję klasy `Powiadomienia::Listener`, a dodatkowo proces główny ma jedną instancję przechowującą identyfikatory procesów potomnych, w której wołane są metody typu `respawnDead()`.

2.8. Schemat bazy danych

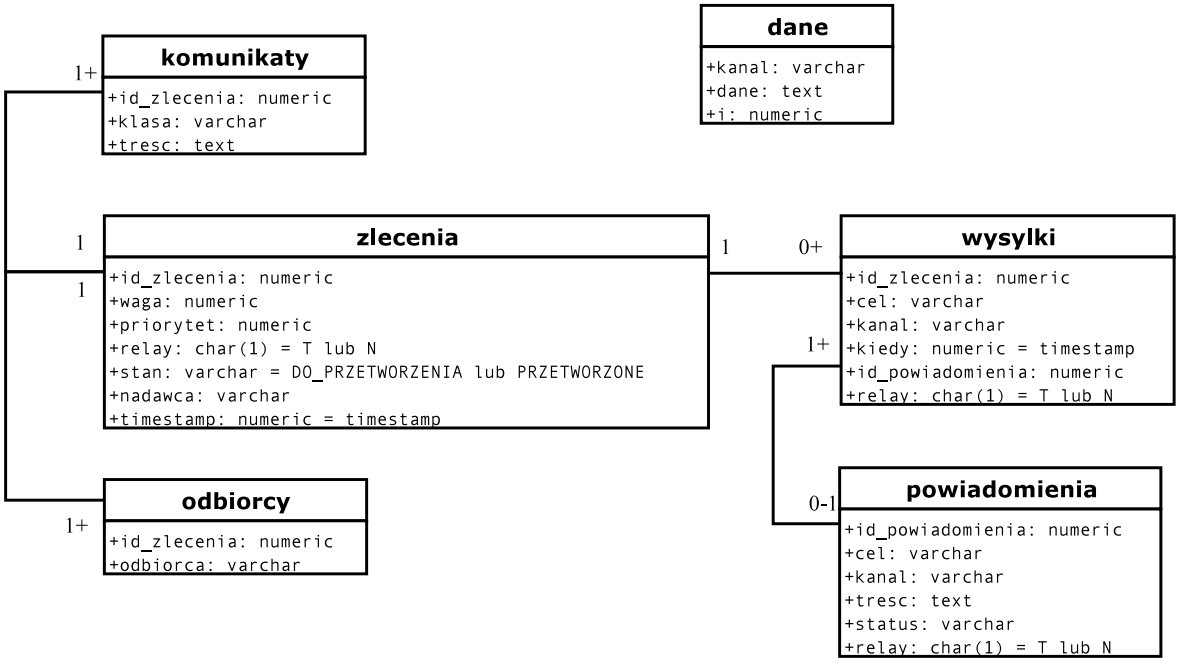
Schemat bazy danych przedstawiono na rys. 2.3. Tutaj opisano poszczególne tabele oraz występujące w nich pola:

`zlecenia` zawiera informacje o przysyłanych komunikatach.

- `id_zlecenia` – unikatowy numer komunikatu, nadawany przy jego wysyłaniu,
- `waga` – waga komunikatu,
- `priorytet` – priorytet komunikatu,
- `relay` – powinno być ustawione na `T`, jeśli w razie niepowodzenia należy przekazać wiadomość do wysłania przez inny serwer,
- `stan` – `DO_PRZETWORZENIA`, jeśli wiadomość jeszcze nie była oglądana przez zarządcę kolejki, lub `PRZETWORZONE` jeśli wiadomość została wstawiona do wysyłki (patrz tabela `wysylki`),



Rysunek 2.2: Zależności między modułami i obiektami



Rysunek 2.3: Schemat bazy danych

`nadawca` – nazwa użytkownika, który wysłał komunikat,

`timestamp` – uniksowy znacznik czasu wysłania.

`odbiorcy` – oryginalny spis odbiorców komunikatu, zadany przez nadawcę.

`id_zlecenia` – dowiązanie do komunikatu (`zlecenia.id_zlecenia`),

`odbiorca` – nazwa odbiorcy.

`komunikaty` – spis treści przypisanych do jednego komunikatu.

`id_zlecenia` – dowiązanie do komunikaty (`zlecenia.id_zlecenia`),

`klasa` – określenie rodzaju treści. Domyślna klasa jest pusta (pusty napis). Każdy komunikat musi mieć zdefiniowaną treść dla domyślnej klasy;

`tresc` – treść komunikatu.

`wysylki` – komunikaty przygotowane do agregacji i wysłania.

`id_zlecenia` – powiązanie z oryginalnym komunikatem (`zlecenia.id_zlecenia`),

`cel` – nazwa ostatecznego odbiorcy,

`kanal` – kanał, którym będzie wysłany komunikat,

`kiedy` – uniksowy znacznik czasu, określający do kiedy komunikat może leżeć w kolejce. Po przekroczeniu tego czasu powinien zostać wysłany;

`id_powiadomienia` – informacja o tym, która wiadomość zawierała w sobie dany komunikat. Może wskazywać na `powiadomienia.id_powiadomienia`, być `NULL` (komunikat jeszcze nie został zagregowany) lub -1 (komunikat intencjonalnie nie będzie wysłany);

`relay` – czy komunikat powinien być w razie problemów z wysłaniem przekazany do wysłania przez inny serwer.

`powiadomienia` – wysyłane wiadomości.

`id_powiadomienia` – unikatowy numer wysyłanej wiadomości, nadawany przy agregowaniu komunikatów,

`cel` – nazwa ostatecznego odbiorcy,

`kanal` – kanał, kórym będzie wysłana wiadomość,

`tresc` – treść wysyłanej wiadomości,

`status` – status wiadomości. Jedno z:

- `DO_WYSLANIA`,
- `WYSLANO`,
- `WYSLANO_NIE_DOSTARCZONO`,
- `NIE_WYSLANO`,
- `DOSTARCZONO`,
- `DOSTARCZONO_NIE_PRZECZYTANO`,
- `PRZECZYTANO`.

Znaczenie statusów jest takie jak w rozdziale 2.3;

`relay` – czy wiadomość powinna być przekazana do wysłania przez inny serwer w razie problemów z wysłaniem. To pole powinno być ustawione na `T`, jeśli choć jeden zagregowany w wiadomości komunikat miał pole `relay` ustawione na `T`.

`dane` – lokalne dane specyficzne dla kanałów.

`kanal` – nazwa kanału,

`dane` – dane kanału,

`i` – pole numeryczne, np. do rozróżniania wpisów przez kanał.

2.9. Instrukcja dla administratora systemu

2.9.1. Wymagania

Wymagany jest zainstalowany na serwerze Perl (tesowana wersja 5.8.8) z modułami:

`DBI` w wersji co najmniej 1.38 (testowana 1.48),

`DBD::*` w zależności od wykorzystywanego silnika bazodanowego, np, `DBD::Pg`, `DBD::mysql`. Przetestowane z `DBD::mysql` 3.0006. Wymagane właściwości silnika: obsługa `DBI::last_insert_id()`, równoległych zapytań do bazy, wielokrotnych lewostronnych złączeń,

`RPC::XML` testowana wersja 0.58,

libwww (`HTTP::Daemon`) testowana wersja 5.803

`IO::Socket::SSL` przy korzystaniu z połączeń SSL, w wersji innej niż 0.97, testowana wersja 0.96,

`HTTP::Daemon::SSL` przy korzystaniu z połączeń SSL, testowana wersja 1.02,

`Log::Log4perl` testowana wersja 0.52,

`Math::Expression` testowana wersja 1.14,

`XML::Simple` testowana wersja 2.14,

`XML::LibXML` testowana wersja 1.58,

`Data::Dump` testowana wersja 1.03,

2.9.2. Instalacja

Archiwum z serwerem należy rozpakować w dowolnie wybranym katalogu. Dodatkowo należy wybrać silnik bazodanowy (np. MySQL, postgreSQL), a następnie założyć bazę danych na podstawie pliku struktury w katalogu `sql/` (dostępne są wersje struktury dla różnych silników bazodanowych).

2.9.3. Uruchamianie

Należy wejść do katalogu z rozpakowanym archiwum i wydać polecenie:

```
perl ./Powiadomienia.pl
```

2.10. Instrukcja dla administratora powiadomień

2.10.1. Plik konfiguracyjny

Plik konfiguracyjny (`Powiadomienia.xml`) jest plikiem XML, którego struktura jest określona w pliku DTD (`Powiadomienia.dtd`). Struktura pliku konfiguracyjnego opisana jest poniżej:

`config` główny element, zawierający całą konfigurację. Składa się z elementów: `serverName`, `daemonMode`, `database`, `listener`, `logging`, jednego lub więcej elementów `obslugaKanalu`, jednego elementu `odbiorcy`, `users`.

`serverName` nazwa procesu serwera. Po uruchomieniu na liście procesów widoczna będzie nazwa tu podana.

`daemonMode` jeśli jest ustawione na 1, to serwer po uruchomieniu odłączy się od terminala kontrolującego i będzie pracował dalej w tle, zapisując informacje o swoim działaniu do logów. Jeśli jest ustawione na 0, to serwer pracuje na terminalu, dodatkowo wypisując na standardowe wyjście kopię logów.

`database` konfiguracja bazy danych. Składa się z elementów: `uri`, `username`, `password`.

`uri` identyfikator bazy danych, zgodny z nomenklaturą `DBI`. Najczęściej ma postać `dbi:nazwa_silnika:opcje_silnika`, np. `dbi:mysql:database=powiadomienia`.

`username`, `password` nazwa użytkownika i hasło do bazy danych.

`listener` konfiguracja procesów nasłuchujących. Składa się z elementów: `listenerPrefork`, `listenerLifeTime`, `IP`, `port`, `SSLCert`, `SSLKey`, `SSLCACert`.

`listenerPrefork` ile procesów nasłuchujących powinno jednocześnie oczekiwać na przychodzące połączenia.

`listenerLifeTime` po obsłużeniu ilu żądań proces nasłuchujący powinien zakończyć działanie.

`IP`, `port` adres IP i port TCP, na których należy oczekiwać na nowe połączenia.

`useSSL` 1 jeśli serwer powinien korzystać z SSL, 0 jeśli nie.

`SSLCert` jeśli używany jest SSL, to tutaj powinna się znajdować ścieżka do pliku certyfikatu SSL serwera.

`SSLKey` jesli używany jest SSL, to tutaj powinna się znajdować ścieżka do pliku z kluczem prywatnym serwera. Może wskazywać na ten sam plik co `SSLCert`, wtedy certyfikat i klucz powinny być umieszczone w tym pliku jeden pod drugim.

`SSLCACert` jeśli podana zostanie ścieżka do pliku z certyfikatami, to serwer będzie weryfikował certyfikat użyty przez klienta, sprawdzając czy został podpisany przez jeden z zadanych certyfikatów. Jeśli podpis będzie nieprawidłowy, to serwer odmówi połączenia. Wskazane jest korzystanie z `SSLCACert` przy wykorzystywaniu identyfikatora klienta zawartego w certyfikacie do uwierzytelniania.

`logging` konfiguracja logowania. Składa się z elementów: `log`, `debug`, `syslogFacility`, `logLayout`.

`log` ścieżka do pliku z logami lub słowo kluczowe `syslog`, jeśli komunikaty mają być wysyłane do serwera syslog zamiast do pliku.

`debug` 1, jeśli mają być włączone dodatkowe komunikaty diagnostyczne, 0 jeśli logowanie ma być na normalnym poziomie.

`syslogFacility` jakim znacznikiem (ang. *facility*) oznaczać wiadomości w logach systemowych, zgodnie ze stroną podręcznika systemowego opisującą `Log::Dispatch::Syslog`.

`logLayout` format logowanych komunikatów, zgodnie z sekcją Log Layouts podręcznika do `Log::Log4perl`.

`obslugaKanalu` konfiguracja kanałów. Musi zawierać atrybut `name`, definiujący nazwę kanałów. System automatycznie dołączy obsługę kanału, włączając plik `.pm` o nazwie takiej jak nazwa kanału z katalogu `Kanaly/`. Składa się z dowolnej liczby elementów `opoznienie` i `opcja`.

`opoznienie` definiuje maksymalne opóźnienie wysyłania (w minutach) dla zadanego priorytetu wiadomości. Przyjmuje opcjonalny argument `priorytet`, który musi być listą priorytetów, oddzieloną przecinkami, gdzie każdy element listy jest liczbową wartością priorytetu albo zakresem wartości z wstawionym myślnikiem między dolną a górną granicą zakresu. Jeśli priorytet nie został podany, to element jest dopasowywany do dowolnej wiadomości. W trakcie ustalania opóźnienia komunikatu wpisy `opoznienie` przeglądane są w takiej kolejności, w jakiej są wpisane, przekazywany jest pierwszy pasujący.

`opcja` definiuje wartości opcji, z których korzystać może implementacja kanału. Musi mieć zdefiniowany atrybut `name`. Dostępne opcje zależą od kanału.

`odbiorcy` zawiera informacje o odbiorcach wiadomości. Składa się z dowolnej liczby elementów cel i grupa.

`cel` zawiera informacje o pojedynczym odbiorcy. Posiada obowiązkowy atrybut `name`, zawierający nazwę odbiorcy. Składa się z elementów `kanal`.

`kanal` zawiera informacje o pojedynczym kanale. Posiada obowiązkowe atrybuty `name`, z nazwą kanału, oraz `adres`, zawierający adres (numer telefonu, identyfikator IM, ...), przekazywany do obsługi kanału. Dodatkowo posiada opcjonalny atrybut warunek, będący wyrażeniem matematycznym (parsowanym przez moduł `Math::Expression`), z predefiniowanymi zmiennymi:

`$w` – waga komunikatu.

Jeśli warunek jest ustawiony, to kanał zostanie użyty tylko wtedy, gdy wyliczone wyrażenie ma wartość różną od zera.

`grupa` zawiera informacje o grupie odbiorców. Składa się z elementów `member`. Posiada obowiązkowy atrybut `name`, będący nazwą grupy (odbiorcy).

`member` nazwa odbiorcy będącego członkiem grupy. Posiada opcjonalny atrybut `warunek`, obsługiwany identycznie jak atrybut `warunek` elementu `kanal`. Jeśli warunek nie jest spełniony, to członek grupy jest z niej wykluczany.

`users` konfiguracja użytkowników wysyłających wiadomości. Składa się z elementów `user` i `acl`.

`user` informacje o użytkowniku. Posiada obowiązkowy atrybut `name` (nazwa użytkownika) oraz opcjonalne argumenty `ip` (adres IP z którego łączy się użytkownik), `cn` (identyfikator klienta w certyfikacie SSL) oraz `password` (hasło).

`acl` pojedynczy wpis listy dostępowej. Obowiązkowy jest atrybut `result`, którego dopuszczalne wartości to `allow` i `deny`. Dodatkowo można skorzystać z opcjonalnych atrybutów: `action`, określającego czynność którą użytkownik próbuje wykonać, przyjmującego wartości `send`, `explain`, `status` lub `relay`; atrybutu `user` z wpisaną nazwą użytkownika wykonującego czynność, atrybutu `odbiorca` z wpisaną nazwą odbiorcy którego dotyczy czynność. Listy dostępowe sprawdzane są w kolejności wpisania, przekazywany jest pierwszy pasujący wynik.

Przykładowy plik konfiguracyjny:

```
<config>
  <serverName>powiadomienia</serverName>
  <daemonMode>1</daemonMode>
  <database>
    <uri>dbi:mysql:database=powiadomienia</uri>
    <username>mysql</username>
    <password></password>
  </database>
  <listener>
    <listenerPrefork>3</listenerPrefork>
    <listenerLifeTime>100</listenerLifeTime>
    <IP>127.0.0.1</IP>
    <port>1986</port>
    <useSSL>1</useSSL>
    <SSLCert>certs/server-cert.pem</SSLCert>
    <SSLKey>certs/server-key.pem</SSLKey>
    <SSLCACert>certs/ca.pem</SSLCACert>
  </listener>
  <logging>
    <log>syslog</log>
    <debug>0</debug>
    <syslogFacility>daemon</syslogFacility>
    <logLayout>[%P:%p] %m%n</logLayout>
  </logging>
  <obslugaKanalu name="sms">
          <opoznienie priorytet="1,2">-1</opoznienie>
          <opoznienie priorytet="3">15</opoznienie>
  </obslugaKanalu>
  <obslugaKanalu name="mail">
          <opoznienie>10</opoznienie>
  </obslugaKanalu>
  <odbiorcy>
      <cel name="osoba1">
          <kanal name="sms" warunek="$w >= 4">501234567</kanal>
```

```
            <kanal name="mail">osoba1@net.icm.edu.pl</kanal>
        </cel>
        <cel name="osoba2">
            <kanal name="sms">502345678</kanal>
            <kanal name="mail">osoba2@net.icm.edu.pl</kanal>
        </cel>
        <grupa name="obsluga_sieci">
            <member>osoba1</member>
            <member warunek="$w_>=_5">osoba2</member>
        </grupa>
    </odbiorcy>
    <users>
        <user name="localhost" ip="127.0.0.1" />
        <user name="system1" password="password1" />
        <user name="system2"
          cn="/C=PL/OU=Uniwersytet_Warszawski/CN=system2" />
        <acl user="system1" odbiorca="osoba1" result="allow" />
        <acl user="system1" result="deny" />
        <acl action="explain" result="allow" />
        <acl result="allow" />
    </users>
</config>
```

2.10.2. Dostępne moduły kanałów

`syslog` Stworzony jako przykład implementacji kanału asynchronicznego. Wysyłanie komunikatów polega na odpowiednim ich sformatowaniu i zapisaniu w dzienniku systemowym za pomocą metody systemowej syslog(). Wiadomości nie są agregowane, tzn. tworzone jest osobne powiadomienie dla każdego komunikatu. Wykorzystuje domyślną klasę wiadomości. Nie ma konfigurowalnych opcji. Nie sprawdza statusu wiadomości.

`terminal` Stworzony jako przykład implementacji kanału synchronicznego. Wysyłanie komunikatów polega na wypisaniu ich na standardowe wyjście (uwaga: działanie kanału będzie widoczne tylko po ustawieniu opcji serwera `daemonMode` na zero). Każdy komunikat powoduje wygenerowanie jednego powiadomienia. Wykorzystuje domyślną klasę wiadomości. Nie ma konfigurowalnych opcji.

`mail` Wysyła pocztę elektroniczną za pomocą programu `sendmail`. Pracuje w trybie synchronicznym. Agregacja polega na wstawieniu wszystkich wiadomości do pojedynczego listu, każda w osobnym wierszu opatrzonym datą. Konfigurowalne opcje: `sendmail` – ścieżka do programu `sendmail`, najczęściej `/usr/lib/sendmail`.

`sql` Wstawia komunikaty do bazy danych, wykorzystując silnik DBI i zadany format polecenia SQL. Pracuje w trybie asynchronicznym. Nie ma agregacji, tzn. każdy komunikat powoduje wykonanie osobnego zapytania. Konfigurowalne opcje: `db` – identyfikator DBI bazy, `username` – nazwa użytkownika bazy danych, `password` – hasło do bazy danych, `insertsql` – składnia zapytania służącego do wstawiania, w formacie `DBI->prepare()`, `checksql` – składnia zapytania służącego do sprawdzania stanu wiadomości. Sprawdza stan wiadomości (do poziomu „przeczytano").

2.11. Instrukcja dla programisty

2.11.1. Tworzenie własnych kanałów

Obsługa własnych kanałów powinna być realizowana w formie modułów `.pm` w katalogu `Kanaly/`. Kanał powinien być podklasą klasy `Powiadomienia::Kanal`. W trybie synchronicznym konstruktor kanału powinien wołać konstruktor `SUPER::new()` z drugim argumentem ustawionym na zero. W takim wypadku klasa powinna implementować metody (poza konstruktorem):

`_do_send()` przyjmującą dwa argumenty, adres kanału oraz treść wiadomości. Funkcja powinna przekazywać 1 jeśli wiadomość została wysłana, 0 jeśli nie została wysłana;

`_do_aggregate()` przyjmującą dwa argumenty, nazwę odbiorcy oraz referencję do listy tablic, gdzie tablica posiada pola w kolejności: identyfikator zlecenia, nadawca, znacznik czasu, treść, klasa, priorytet, waga, czy przekazywać wiadomość do innego serwera. Funkcja powinna wykorzystywać uchwyt do zapytania SQL `$self->{INSERT_POWIADOMIENIE}` do wstawiania powiadomień do bazy, a na koniec działania przekazać referencję do listy dwuelementowych tablic, gdzie pierwszym elementem tablicy jest numer komunikatu, a drugim – numer wiadomości, która zawiera ten komunikat.

Kanał pracujący w trybie asynchronicznym powinien mieć drugi argument konstruktora klasy nadrzędnej niezerowy, a dodatkowo implementować metody:

`_init_daemon()`, która zainicjuje zapytania do bazy danych (kanały synchroniczne robią to w konstruktorze klasy);

`_check_status()`, która sprawdza stan wysłanych wiadomości, a następnie ustawia ich stan korzystając z uchwytów zapytań zdefiniowanych w kontruktorze klasy `Kanal`;

`_stop()`, która wykonuje dodatkowe operacje przed zakończeniem procesu obsługującego kanał.

Oto przykładowy kanał pracujący w trybie synchronicznym, nie agregujący wiadomości i wypisujący komunikaty na standardowe wyjście:

```
use strict;
package Kanaly::terminal;
use Powiadomienia::Kanal;
our @ISA = qw(Powiadomienia::Kanal);

sub new {
    my $class = shift;
    my $self = $class->SUPER::new('terminal', 0);
    bless ($self, $class);
    return $self;
}

sub _do_send {
    my ($self, $cel, $tresc) = @_;
    print "Wiadomosc dla $cel: $tresc\n";
    return 1;
}
```

```
sub _do_aggregate {
    my ($self, $cel, $zleceniaref) = @_;
    my $result = [];
    foreach my $zlecenie (@{$zleceniaref}) {
        if ($zlecenie->[4] eq "") {
            my $wiadomosc = sprintf("<%d:%d> %s %s: %s",
                $zlecenie->[5], $zlecenie->[6],
                $zlecenie->[1], $zlecenie->[2],
                $zlecenie->[3]);
            $self->{INSERT_POWIADOMIENIE}->execute($cel,
                $self->{KANAL}, $wiadomosc, "DO_WYSLANIA", 0);
            push @{$result}, [$zlecenie->[0],
                $self->{DBH}->last_insert_id(undef, undef,
                    'powiadomienia', 'id_powiadomienia')];
        }
    }
    return $result;
}
```

Rozdział 3

Oprogramowanie klienckie

3.1. POSIX shell

W katalogu `clients/shell` znajduje się przykładowa implementacja klienta XML-RPC w języku systemowego interpretera poleceń. Implementacja została stworzona w zgodzie ze standardem POSIX, dzięki czemu działa w różnych implementacjach intepretera, m.in. Bourne Again Shell (bash), Korn shell (ksh), Z shell (zsh), ash.

Przykładowy klient składa się z dwóch plików: `xmlrpc-client.sh` – biblioteka udostępniająca funkcje do wysyłania żądań XML-RPC, oraz `powiadom.sh`, wykorzystujący bibliotekę do wysyłania prostych powiadomień.

3.1.1. Biblioteka `xmlrpc-client.sh`

Biblioteka udostępnia następujące funkcje interpretera:

`xmlrpc_request()` Funkcja wypisuje na standardowe wyjście strukturę XML tworzącą gotowe zapytanie.

Pierwszym spodziewanym argumentem funkcji jest nazwa metody do wywołania. Argumenty zdalnej procedury są następnie przekazywane jako następujące po sobie sekwencje argumentów funkcji, każda sekwencja zgodna z jednym z formatów:

int `wartość` – argumentem zdalnej procedury będzie liczba całkowita o wartości `wartość`,

string `wartość` – argumentem zdalnej procedury będzie napis o treści `wartość`,

array `lista_argumentów .` – argumentem zdalnej procedury będzie tablica, o wartościach zdefiniowanych w `lista_argumentów`. `Lista_argumentów` musi być serią sekwencji zgodnych z formatem sekwencji xmlrpc_request(),

struct `nazwa1 wartość1 nazwa2 ... .` – argumentem zdalnej procedury będzie tablica asocjacyjna, której kluczami będą kolejne argumenty `nazwa`, a wartościami kluczy będą odpowiadające im argumenty `wartość`. Każda `wartość` musi mieć format zgodny z jedną z sekwencji funkcji `xmlrpc_request()`.

`http_xmlrpc_request()` Funkcja wypisuje na standardowe wyjście odpowiedni nagłówek HTTP, a po nim strukturę XML z żądaniem. Wyjście tej funkcji nadaje się do wysłania bezpośrednio do serwera XML-RPC.

Pierwsze dwa argumenty tej funkcji to adres serwera w postaci: *host:port* oraz ścieżka HTTP wykorzystywane do wołania XML-RPC na serwerze. Reszta argumentów jest

identyczna z argumentami funkcji `xmlrpc_request()` i jest do niej przekazywana bez zmian.

`send_http_xmlrpc_request()` Wysyła żądanie do serwera XML-RPC, bez korzystania z SSL. Na standardowe wyjście wypisywana jest odpowiedź serwera.

Do obsługi połączenia sieciowego wykorzystywany jest program NetCat (`nc`), który musi być zainstalowany na serwerze.

Argumenty funkcji są identyczne jak w wypadku `http_xmlrpc_request()`.

`send_ssl_http_xmlrpc_request()` Funkcja wysyła żądanie do serwera XML-RPC, wykorzystując do tego połączenie SSL. Na standardowe wyjście wypisywana jest odpowiedź serwera.

Do obsługi połączenia sieciowego wykorzystywany jest program `openssl`, który musi być dostępny na serwerze.

Funkcja oczekuje listy argumentów w postaci:

```
{ opcje\_openssl } -- { argumenty http\_xmlrpc\_request }
```

Opcje OpenSSL są zgodne z opisem zamieszczonym w podręczniku systemowym do opcji `s_client` polecenia `openssl`. Można je wykorzystać np. do przekazania certyfikatu klienta lub certyfikatów CA do weryfikacji tożsamości serwera. Dwa minusy rozdzielają opcje OpenSSL od argumentów XML-RPC. Reszta argumentów jest zgodna z listą argumentów dla funkcji `http_xmlrpc_request()`.

Bibliotekę interpretera poleceń można dołączyć wykorzystując polecenie `.` (kropka), podając mu ścieżkę do biblioteki:

```
. ./xmlrpc-client.sh
```

3.1.2. Program `powiadom.sh`

Program wysyła wiadomość ze zdefiniowaną domyślną treścią o zadanej wartości do zadanego odbiorcy. Przyjmuje sześć argumentów, w kolejności: adres serwera w postaci *host:port*, ścieżka HTTP, nazwa odbiorcy, treść wiadomości, priorytet, waga. Połączenie wykorzystuje SSL, wykorzystywany jest certyfikat klienta w pliku `client.pem`.

Przykładowe uruchomienie programu:

```
sh ./powiadom.sh localhost:1986 / sebek "testowa wiadomosc" 3 3
```

3.2. PHP

W katalogu `clients/php` znajduje się przykładowa implementacja klienta XML-RPC napisana w PHP. Bazuje ona na zmodyfikowanej bibliotece `xmlrpc-utils.php`, będącej częścią dystrybucji xmlrpc-epi [18]. Modyfikacje polegały na zamianie wołania funkcji fsockopen(), która w PHP w wersji 5 nie obsługuje nadmiarowego argumentu z kontekstem strumienia, który działał w PHP w wersji 4, na opcjonalne wołanie funkcji `stream_socket_client()`, o ile taka funkcja jest zdefiniowana, oraz na dodaniu przekazywania kontekstu do funkcji `xu_rpc_http_concise()`. Semantyka i sposób wołania funkcji się nie zmienił względem oryginalnego, poza dodaniem obsługi klucza `context` w przekazywanej tablicy asocjacyjnej z opcjami.

Dokumentacja dostępnych funkcji biblioteki znajduje się w kodzie samej biblioteki. Do poprawnego działania wymagane jest PHP w wersji co najmniej 4.3, z dodaną obsługą SSL.

Przykładowy klient wykorzytujący bibliotekę znajduje się w pliku `xmlrpcclient.php`. Uruchomienie klienta w PHP spowoduje wysłanie powiadomienia do jednego odbiorcy, o nazwie „obsługa sieci", z wagą 5 i priorytetem 5, ze zdefiniowanymi dwoma klasami komunikatu: domyślna klasa (pusta) z treścią „test", oraz klasa „dlugi" z treścią „to jest test", oraz z wyłączonym przekazywaniem wiadomości do innych serwerów. Otrzymaną odpowiedź klient przetwarza i wyświetla wynikowe dane na standardowe wyjście.

Podając adres serwera XML-RPC w kliencie można podać jego nazwę domenową wprost – wtedy połączenie nie będzie szyfrowane, lub poprzedzając ją prefiksem `ssl://` na początku, co wymusi skorzystanie z SSL przy połączeniu.

Chcąc skorzystać z SSL warto najpierw stworzyć kontekst strumienia poprzez `stream_context_create()`, a następnie ustawić żadane opcje SSL poleceniem `stream_context_set_option()`. W ten sposób można ustawić m.in. ścieżki do certyfikatów klienta oraz CA do weryfikacji certyfikatu serwera. Stworzony w ten sposób kontekst należy przekazać do `xu_rpc_http_concise()` w kluczu `context` tablicy asocjacyjnej.

3.3. Perl

W katalogu `clients/perl` znajduje się przykładowy klient `Client.pl`, zaimplementowany w Perlu z wykorzystaniem modułów `RPC::XML`, `Crypt::SSLeay` oraz `Data::Dump` [4].

Klienta można uruchomić wpisując polecenie:

```
perl Client.pl
```

Spowoduje to wysłanie powiadomienia do jednego odbiorcy („sebek"), z wagą 5, priorytetem 5, treścią domyślną „test perla" oraz wyłączonym przekazywaniem wiadomości do innych serwerów. Powiadomienie zostanie wysłane do serwera *https://localhost:1986*, z wykorzystaniem certyfikatu klienta w pliku `client.pem`. Odpowiedź serwera zostanie wypisana na standardowe wyjście w postaci struktury przekazywanej przez `RPC::XML::Client->send_request()`.

3.4. C

W katalogu `clients/c` znajduje się przykładowy klient napisany w języku C, wykorzystujący bibliotekę xmlrpc-c[17] w wersji co namniej 1.04. Program należy skompilować wydając polecenie `make`. Jeśli kompilacja przebiegnie poprawnie, to wynikiem będzie plik wynikowy `client`, który można uruchomić wpisując:

```
./client
```

Testowy klient wyśle do serwera powiadomień na *https://localhost:1986* powiadomienie dla odbiorcy „obsługa sieci", z wagą 5, priorytetem 5, domyślną treścią wiadomości „test klienta C" i wyłączonym przekazywaniem wiadomości do innych serwerów, wykorzystując certyfikat klienta z pliku `client.pem`.

Rozdział 4

Oprogramowanie do zarządzania siecią

4.1. Monitor ICMP

4.1.1. Opis aplikacji

W katalogu `monitoring/icmp` znajduje się aplikacja do testowania dostępności urządzeń w sieci za pomocą pakietów ICMP echo-request.

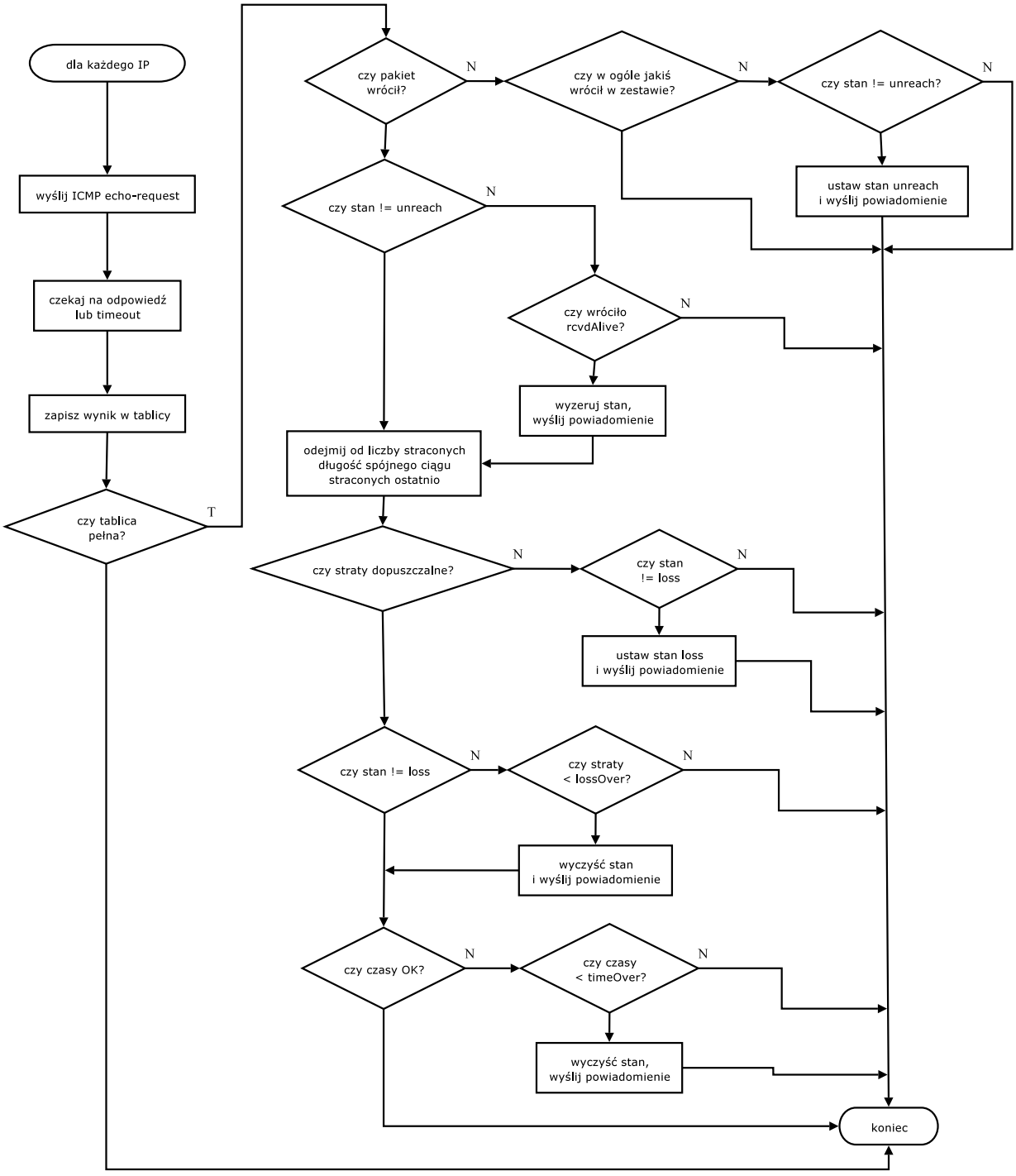
Serwer rozpoczyna działanie od wczytania konfiguracji oraz zamiany nazw domenowych na adresy IP. Następnie w pętli wysyła do określonej grupy adresów IP pakiety ICMP, zbiera odpowiedzi i na ich podstawie zapisuje odpowiedzi w formie stanów „nieosiągalny", „straty pakietów", „duże opóźnienia", „wszystko ok". Zmiany stanów są sygnalizowane powiadomieniami.

Aplikacja korzysta nie tylko z ostatniego otrzymanego wyniku, ale z określonej liczby ostatnich wyników, wyliczając z nich np. średni czas, poziom strat pakietów. Algorytm zaprojektowany jest w taki sposób, żeby tablica służyła za pewnego rodzaju bufor, który pozwala zapobiegać sytuacji, w której pojedynczy zgubiony pakiet skutkuje w krótkim czasie dwoma zmianami stanu („nieosiągalny" – „wszystko ok"). Zapobieganiu temu problemowi służy również osobne określenie progów wejścia do danego stanu oraz wyjścia z niego. Próg wejścia, określany jako maksymalny akceptowalny poziom strat pakietów lub akceptowalny czas powrotu odpowiedzi, zazwyczaj jest niższy od progu wyjścia. Dzięki temu sytuacje, w których czas bądź straty oscylują na poziomie wartości granicznej, są sygnalizowane raz w formie zmiany stanu na „źle", a powrót do „dobrze" odbywa się gdy oscylacja spadnie poniżej innej wartości granicznej.

Schemat pojedynczego przebiegu pętli dla zadanego adresu IP przedstawiony jest na rysunku 4.1.

4.1.2. Budowa aplikacji

Aplikacja zbudowana jest z wykorzystaniem środowiska Perla POE [12], które pozwala tworzyć aplikację w postaci opisywania reakcji na zdarzenia. Wykorzystywane są gotowe komponenty `POE::Component::Client::DNS` i `POE::Component::Client::Ping`, które tworzą zdarzenia w momencie pojawienia się odpowiedzi na wysłane wcześniej żądanie znalezienia nazwy lub wysłania ICMP echo-request. Kernel POE uruchamia odpowiednie procedury zapisane w programie.



Rysunek 4.1: reakcja na zdarzenie związane z ICMP

4.1.3. Wymagania

Do poprawnego działania aplikacja potrzebuje modułów Perla:

`POE` testowana wersja 0.3007,

`POE::Component::Client::DNS` testowana wersja 0.98,

`POE::Component::Client::Ping` testowana wersja 1.13,

`RPC::XML` testowana wersja 0.58,

`XML::Simple` testowana wersja 2.14,

`XML::LibXML` testowana wersja 1.58.

Dodatkowo aplikacja potrzebuje przy uruchamianiu uprawnień administratora, ze względu na specyfikę obsługi pakietów ICMP w systemie Unix.

4.1.4. Konfiguracja

Konfiguracja znajduje się w pliku XML `Monitor.xml`, którego składnia określona jest w pliku DTD `Monitor.dtd`. Najważniejsze opcje wymagające opisania to:

`interval` przerwa pomiędzy kolejnymi odpytaniami,

`timeout` maksymalny czas oczekiwania na odpowiedź ICMP,

`store` ile wyników należy przechowywać w tablicy,

`rcvdAlive` ile pakietów musi wrócić, aby wyjść ze stanu „nie odpowiada”,

`time` maksymalne dopuszczalne opóźnienie (RTT) pakietów ICMP, w milisekundach,

`timeOver` maksymalne dopuszczalne opóźnienie pakietów przy opuszczaniu stanu „długie czasy”,

`loss` maksymalny dopuszczalny poziom strat pakietów (w procentach),

`lossOver` maksymalny dopuszczalny poziom strat przy opuszczaniu stanu „duże straty”,

`przypominaj` co jaki czas przypominać o stanie węzła sieci (o ile jest inny niż „wszystko ok”). Obecnie opcja niewykorzystywana;

`lossClass`, `timeClass`, `unreachableClass`, `runtimeErrorClass` do jakiej klasy odbiorców wysłać powiadomienia odpowiednio o: dużych stratach, dużych czasach, nieosiągalności węzła, błędach aplikacji;

`class` element definiujący klasę odbiorców. Posiada obowiązkowe atrybuty: priorytet, waga, odbiorca, odpowiadające wartościom podstawianym w powiadomieniach;

`host` element definiujący węzeł do odpytywania. Posiada obowiązkowy atrybut `name`, zawierający adres IP lub nazwę domenową komputera. Dodatkowo dopuszczalne są atrybuty: `opis`, który będzie wykorzystywany w powiadomieniach; `every`, który oznacza że dany węzeł ma być testowany raz na tyle cykli, ile wynosi wartość argumentu; `class`, definiujący dla tego węzła indywidualnie klasę odbiorców dla powiadomień (wszystkich typów); `store`, `timeout`, `przypominaj`, `loss`, `lossOver`, `time`, `timeOver`, `rcvdAlive`, `lossClass`, `timeClass`, `unreachableClass`, które nadpisują wartości domyślne konfiguracji.

4.1.5. Uruchamianie

Aplikację uruchamia się poprzez wejście do katalogu z plikami aplikacji i wpisanie polecenia:

```
perl MonitorICMP.pl
```

Rozdział 5

Testy

Za pomocą jednego z dołączonych programów klienckich zostały przetestowane następujące funkcje zaimplementowanego serwera powiadomień:

- poprawne uruchamianie w trybie z terminalem i bez terminala kontrolującego,
- poprawne działanie systemu logowania, również z włączeniem komunikatów diagnostycznych,
- poprawne działanie metody `powiadomienia.version` (przekazanie zapisanego w kodzie numeru wersji serwera),
- poprawne działanie metody `powiadomienia.send` w kilku wariantach, sprawdzających:
  - poprawne identyfikowanie użytkowników na podstawie hasła, adresu IP, certyfikatu SSL,
  - funkcjonowanie list dostępowych dla czynności `send`,
  - generowanie właściwego komunikatu o błędzie w wypadku braku uprawnień,
  - generowanie właściwego komunikatu o błędzie w wypadku braku domyślnej treści komunikatu,
  - poprawne przetwarzanie wiadomości w kolejce,
  - poprawne tłumaczenie nazw odbiorców, z uwzględnieniem zadanej wagi komunikatu,
  - poprawną dystrybucję komunikatów do kanałów,
  - poprawną agregację poszczególnych kanałów, zgodnie z priorytetem komunikatu,
  - poprawne wybieranie preferowanego rodzaju treści z kilku alternatywnych zawartych w komunikacie,
  - poprawne dostarczanie komunikatów z wykorzystaniem kanałów: `terminal`, `syslog`, `mail`.
- poprawne działanie metody `powiadomienia.status` w kilku wariantach, sprawdzających:
  - poprawne identyfikowanie użytkowników, jak w metodzie `powiadomienia.send`,
  - funkcjonowanie list dostępowych dla czynności `status`,
  - poprawne tłumaczenie nazw odbiorców, zgodnie z priorytetem komunikatu i uprawnieniami użytkownika,

  - poprawne przekazywanie kodów reprezentujących stan komunikatów w systemie.

- poprawne działanie metody `powiadomienia.explain` w kilku wariantach, sprawdzających:
  - poprawne identyfikowanie użytkowników, jak w metodzie `powiadomienia.send`,
  - funkcjonowanie list dostępowych dla czynności `explain`,
  - poprawne tłumaczenie nazw odbiorców, zgodnie z zadanym priorytetem i uprawnieniami użytkownika,
  - poprawne przekazywanie list wynikowych.

Po przeprowadzeniu testów systemu powiadomień zostały również przeprowadzone testy przykładowej aplikacji do monitorowania dostępności węzłów w sieci. Testowano system przez krótki czas na pojedynczej stacji monitorującej, na której również uruchomiony był system powiadomień, sprawdzając co 10 sekund dostępność kilku węzłów. Jeden z węzłów odpowiadał po czasie zdecydowanie dłuższym niż pozostałe. Dzięki temu możliwe było dobranie progu maksymalnego czasu odpowiedzi tak, aby w wypadku tego konkretnego węzła generowany był komunikat o długich czasach. Innym testowanym węzłem był komputer z zainstalowanym systemem Linux i filtrami IPtables, które odrzucały przez pewien czas co drugi pakiet ICMP, a po pewnym czasie zaczynały odrzucać wszystkie pakiety. Dzięki temu przetestowano generowanie komunikatów o dużych stratach oraz o niedostępności węzła. Komunikaty były agregowane w kanale `mail`. Listy elektroniczne z komunikatami zostały dostarczone prawidłowo. Przetestowano tym samym praktycznie przetwarzanie, agregację i wysyłanie komunikatów.

Rozdział 6

Podsumowanie

6.1. Wyniki pracy

Stworzony w ramach niniejszej pracy system powiadomień spełnił wszystkie wymagania określone w rozdziale 1. Wstępne testy pokazały, że rzeczywiście usprawnia on pracę administratora sieci dzięki lepszemu przedstawieniu informacji o zdarzeniach w sieci, a także dzięki szybkiemu i pewnemu ich dostarczeniu.

6.2. Wdrożenie

W najbliższym czasie planowane jest wdrożenie stworzonego systemu oraz aplikacji do monitorowania dostępności węzłów w Dziale Sieciowym ICM [5]. Dział ten zajmuje się utrzymaniem szkieletowej sieci komputerowej Uniwersytetu Warszawskiego.

Dotychczas wykorzystywany był tam autorski system monitoringu oraz powiadomień, który nie posiadał cech wymienionych w rozdziale 1, w szczególności nie potrafił agregować komunikatów, a wysyłanie komunikatów z serwerów nie będących jednocześnie serwerami powiadomień było bardzo utrudnione. Wykorzystywana aplikacja do monitoringu dostępności reagowała na zmiany w sieci z bardzo dużym opóźnieniem (rzędu 20 minut).

Istnieje nadzieja, że wymiana systemu na nowy pozwoli usprawnić funkcjonowanie szkieletowej sieci komputerowej UW dzięki szybszemu sygnalizowaniu awarii oraz lepszemu zorganizowaniu dostarczania wiadomości.

6.3. Usprawnianie obecnej implementacji serwera powiadomień

W trakcie tworzenia oprogramowania lub w trakcie testów, pojawiały się pomysły na kolejne usprawnienia w kodzie, rozbudowę funkcjonalności, kolejne aplikacje towarzyszące. Były to między innymi:

- zmiana modelu obsługi połączeń z bazą danych tak, aby wykluczyć jednoczesny dostęp wielu procesów do bazy danych, np. wykorzystując oddzielny proces pośredniczący we wszystkich operacjach bazodanowych. Umożliwi to wykorzystanie silnika bazodanowego SQLite, który ze względu na swoją wysoką wydajność, niewielkie wymagania i trywialną instalację wydaje się idealny do zastosowania w niewielkich aplikacjach.
- dodanie obsługi pliku kontrolnego z identyfikatorem procesu (ang. *pidfile*), mające na celu usprawnienie obsługi serwera przez skrypty startowe systemu. Można do tego wykorzystać jedną z gotowych bibliotek Perla, np. `File::Pid`.

- rozbudowanie wiersza poleceń serwera o argumenty – ścieżka do pliku konfiguracyjnego, do pliku DTD, ścieżka do plików serwera. Przy dodanej obsłudze pliku kontrolnego dodatkowo można zaimplementować w wierszu poleceń argumenty typu: `start`, `stop`, `reload`.

- lepsza kontrola procesów potomnych. Obecnie, jeśli proces potomny się zawiesi i nie będzie reagował na sygnał `TERM` (np. proces kanału, który ma problem z zewnętrzną aplikacją do wysyłania wiadomości), to proces główny będzie wisiał w nieskończoność czekając na jego zakończenie. Wskazane byłoby dodanie okresu oczekiwania na zakończenie procesu, a następnie wysłanie mu sygnału `KILL`.

- obsługa dodatkowych kanałów, bądź usprawnienie obecnych. Nowe przydatne kanały to np: GaduGadu (z wykorzystaniem Perlowej biblioteki `Net::Gadu` oraz biblioteki C libgadu, będącej częścią projektu EKG), Jabber (np. z wykorzystaniem `Jabber::Lite`). Możliwe jest zaimplementowanie komunikatów głosowych za pomocą programu `powiedz`, wykorzystania dedykowanych klas treści komunikatów, przeznaczonych do powiadomień głosowych, oraz z użyciem jakiegoś klienta VoIP, lub `vgetty` i modułu `Modem::Vgetty`.

  W obsłudze już istniejących kanałów wskazane jest np. rozbudowanie kanału `sms` o obsługę raportów doręczeń.

- dodanie obsługi błędów przy wszystkich metodach XML-RPC. Metody powinny móc przekazać wynik lub komunikat o błędzie. Obecnie tylko metoda `send` udostępnia kody błędu przy wysyłaniu, jednak nawet tutaj brakuje dodatkowych komunikatów diagnostycznych.

- stworzenie aplikacji WWW do zarządzania konfiguracją odbiorców powiadomień.

- zaimplementowanie pełnej funkcjonalności przekazywania. W obecnej implementacji istnieją jedynie odpowiednie punkty wejścia w kodzie (metoda `powiadomienia.relay`, wydzielony kawałek struktury `if()` w `Powiadomienia::Kanal`, obsługa ACL dla czynności `relay`), które umożliwiają stosunkowo łatwe zaimplementowanie pełnej funkcjonalności.

- zaimplementowanie serwerów podrzędnych (ang. *slave* lub *secondary*), na wzór mechanizmu, którym posługują się serwery DNS. Serwery podrzędne nie posiadałyby własnej konfiguracji odbiorców, a jedynie informację o serwerze głównym (ang. *master* lub *primary*), z którym komunikowałyby się w celu uzyskania pełnej konfiguracji. Z kolei serwer główny posiadałby informacje o serwerach podrzędnych i w momencie zmiany głównej kopii konfiguracji potrafiłby poinformować je o konieczności ściągnięcia nowej wersji (odpowiednik mechanizmu powiadomień występującego w DNS).

- zmiana sposobu wołania metody `_do_aggregate()` kanałów, tak aby klasa implementująca kanał nie musiała zawierać odwołań do bazy SQL. Obecnie metoda `_do_aggregate()` przekazuje listę par (*id_komunikatu*, *powiadomienie*), a wstawianie powiadomień implementowane jest wewnątrz metody. Można zmienić tę implementację, tak aby `_do_aggregate()` przekazywało listę powiadomień do wstawienia, a z każdym z nich wiązało listę komunikatów.

- metody `_check_status()` oraz `_init_daemon()` kanałów powinny być opcjonalne. Gdy nie jest zdefiniowana metoda `_check_status`, system powinien obsługiwać status wiadomości tak jak w wypadku kanałów synchronicznych.

Również aplikacje korzystające z systemu powiadomień mogą być rozbudowywane o dodatkowe opcje (np. monitoring ICMP – obsługa plików PID, obsługa grup węzłów traktowanych w powiadomieniach jak jeden węzeł), jednak nie jest to przedmiotem tej pracy, jako że aplikacje są jedynie przykładowym wykorzystaniem systemu powiadomień w systemach do zarządzania siecią.

Bibliografia

[1] *3 Fatal Mistakes In Managing Alarms in Your Communications and Data Networks*, DPS Telecom, *http://www.dpstele.com/white-papers/fatal-network-mistakes/*

[2] Big Sister, *http://www.bigsister.ch/*

[3] *Choosing a Monitoring Solution for Your IT Infrastructure*, eG Innovations, *http://www.webbuyersguide.com/bguide/Whitepaper/WpDetails.asp?wpId=MjE5MA*

[4] CPAN, katalog modułów Perla, *http://www.cpan.org*, *http://search.cpan.org*

[5] Dział Sieciowy ICM Uniwersytetu Warszawskiego, *http://www.net.icm.edu.pl*

[6] FreshMeat, *http://www.freshmeat.net/*

[7] Gnokii, *http://www.gnokii.org/*

[8] Just For Fun Network Management System, *http://www.jffnms.org/*

[9] Nagios, *http://nagios.org/*

[10] *Network Alarm Monitoring Fundamentals*, DPS Telecom, *http://www.dpstele.com/white-papers/monitoring-fundamentals/*

[11] PHP Simple Informer, *http://www.godel.com.ar/psi/*

[12] POE, platforma do tworzenia aplikacji w Perlu sterowanych zdarzeniami, *http://poe.perl.org/*

[13] RPC::XML, moduły Perla do obsługi XML-RPC, *http://www.blackperl.com/RPC::XML/*

[14] Skrypty SMS, *http://sms.jfiok.org/*

[15] SNIPS, dawniej NOCOL, *http://www.navya.com/snips/*

[16] Wikipedia, strona poświęcona XML-RPC, z opisem składni komunikatów XML-RPC, *http://en.wikipedia.org/wiki/XML-RPC*

[17] XMLRPC-C, biblioteka C dla XML-RPC, *http://xmlrpc-c.sourceforge.net*

[18] xmlrpc-epi, biblioteka XML-RPC wysokiego poziomu dla PHP, *http://xmlrpc-epi.cvs.sourceforge.net/xmlrpc-epi/xmlrpc-php-epi/sample/utils/utils.php?view=markup*

[19] XML-RPC HOWTO, *http://tldp.org/HOWTO/XML-RPC-HOWTO/index.html*

[20] XML-RPC, strona główna standardu, *http://www.xmlrpc.com/*