

1.	Wstęp	3
1.1	Kontekst użytkowy i technologiczny	3
1.2	Cel pracy i podział pracy	3
2.	Czym jest VMD	6
2.1	Podstawowe założenia i funkcjonalność	6
2.2	Przegląd podobnych rozwiązań	8
2.2.1	VMD jako mechanizm wirtualizacji zasobów	8
2.2.2	Virtual Network Computing	9
2.2.3	System Xen	11
3.	Technologia	13
3.1	Wybór rozwiązań programistycznych	13
3.2	Remote Method Invocation (RMI)	16
3.2.1	Wprowadzenie do technologii RMI	16
3.2.2	Mechanizm serializacji i przekazywania referencji obiektów RMI	19
3.2.3	Schemat nawiązania połączenia RMI	21
3.2.4	Ogólny schemat nawiązania połączenia RMI w VMD	22
3.2.5	SocketFactory i TwoWaySocketFactory w RMI	23
3.3	Dokumentacja	26
3.3.1	Wprowadzenie	26
3.3.2	JavaDoc	26
3.4	Standard kodowania	27
3.5	Wersjonowanie	28
3.6	Maven	29
4.	Architektura systemu VMD	30
4.1	Wprowadzenie	30
4.2	Architektura współdzielonego jądra frameworka - TwoWayCScKernel	33
4.2.1	Interfejsy zdalne	33
4.2.2	System plików zdalnych	35
4.2.3	Pluginy	38
4.2.4	Obsługa RMI	39
4.2.5	Dynamiczne ładowanie klas do maszyny wirtualnej	40
4.3	Architektura serwerowej części szkieletu - ServerKernel	41
4.3.1	Wprowadzenie	41
4.3.2	Implementacje interfejsów zdalnych	41
4.3.3	Użytkownicy, typy i specyfikacja wejścia/wyjścia	43
4.3.4	Pluginy	45
4.3.5	Zarządzanie użytkownikami	47
4.3.6	Narzędzia pomocnicze i zarządzające serwerem	49
4.4	Architektura serwerowej implementacji frameworka - VMDSERVER	52
4.4.1	Wprowadzenie	52
4.4.2	Struktura i implementacja	53
4.4.3	Parametry uruchomieniowe serwera	62
4.5	Architektura klienckiej części szkieletu – ClientKernel	62
4.5.1	Wprowadzenie	62
4.5.2	Interfejsy programistyczne interfejsu graficznego	63
4.5.3	Mechanizm zdalnego listenera	64
4.5.4	Pluginy	64
4.5.5	Klasy narzędziowe	65
5.	Implementacje pluginów	66

5.1	VmdDesktop.....	66
5.1.1	Wprowadzenie.....	66
5.1.2	Część serwerowa.....	67
5.1.3	Część kliencka.....	68
5.1.4	Opis interfejsu.....	68
5.2	VmdTalk.....	70
5.2.1	Wstęp.....	70
5.2.2	Opis interfejsu użytkownika.....	70
5.3	VmdEditor.....	70
5.3.1	Wstęp.....	70
5.3.2	Opis interfejsu.....	71
5.3.3	Używanie edytora z wykorzystaniem kontroli zmian plików zdalnych.....	72
5.4	VMDDraw.....	73
5.4.1	Wprowadzenie.....	73
5.4.2	Algorytm wypełniania.....	74
6.	Testowanie systemu.....	75
6.1	Testy jednostkowe.....	75
6.2	Mock Object.....	77
7.	Zakończenie.....	80
7.1	Istotne zalety projektu.....	80
7.2	Możliwe sposoby wykorzystania systemu.....	81
7.3	Podsumowanie.....	82
8.	Bibliografia.....	84
9.	Załączniki.....	85
9.1	Płyta CD.....	85

1. Wstęp

1.1 Kontekst użytkowy i technologiczny

W ostatnich latach można zaobserwować gwałtowny rozwój technologii informacyjnych opartych na sieciach komputerowych. Rozwój ten umożliwił powstanie takich zastosowań jak www czy email. Dostęp do takich aplikacji jest możliwy zarówno z komputerów stacjonarnych jak i mobilnych. Można oczekiwać, że również w przyszłości będą powstawały nowe zastosowania wykorzystujące powszechny dostęp do mediów i informacji. Prasa z dziedziny IT oraz sieciowe systemy informacyjne pełne są doniesień o aplikacjach. Prasa z dziedziny IT oraz sieciowe systemy informacyjne pełne są doniesień o aplikacjach tego typu. Na przykład oczekuje się, że firma GOOGLE udostępni oprogramowanie typu Office jako aplikację dostępną przez www. Kolejnym przykładem mogą być programy do pracy grupowej: kalendarze, współdzielenie informacji opisujących duże projekty itp. Ten proces prowadzi do wirtualizacji i delokalizacji zarówno zasobów informacyjnych (plików) jak i mocy obliczeniowej (procesorów), których moc wykorzystujemy.

1.2 Cel pracy i podział pracy

Postawione w pracy zadanie polegało na zaprojektowaniu i zaimplementowaniu systemu Virtual Mobile Desktop, nazywanego w dalszej części pracy systemem VMD. System VMD jest rozproszonym systemem informatycznym, który tworzy infrastrukturę dla zdalnego przechowywania informacji, rozproszonego przetwarzania danych. Metaforą usług tego systemu jest dla użytkownika desktop taki jaki zwykle widzi się w systemach GUI (Windows, KDE). Na tym desktopie umieszczone są zasoby informacyjne (pliki) i uruchamiane są aplikacje.

W VMD można wyróżnić 2 podstawowe części: część kliencką i część serwerową.

Praca inżynierska powinna cechować się dobrym i zorganizowanym podziałem pracy, dlatego naturalny podział, jakim jest rozdzielenie systemu rozproszonego na część kliencką i część serwerową, został zaadoptowany przez studentów-programistów zarówno na potrzeby projektowania systemu oraz w procesie implementacji. Czteroosobowa grupa inżynierska podzieliła się na 2 podzespoły. Adam Kozak i Tomasz Głowacki opracowywali część serwerową, natomiast Szymon Kupiński i Aleksander Stasiak część kliencką systemu. Oto podział pracy w systemie VMD:

➤ Tomasz Głowacki

- ✓ Projekt i implementacja serwera VMDServer

- ✓ Testy jednostkowe serwera

 - Testy metodą black box ServerKernel

 - Testy metodą white box VMDServer

- ✓ Implementacja Mock Object

- ✓ Testowanie przy użyciu Mock Object

- ✓ Analiza podobnych rozwiązań

- ✓ Dokumentacja serwera

- ✓ Część teoretyczna pracy dotycząca wersjonowania, Mavena, Javy, RMI, Xena oraz VNC, testowania, kodowania metodą UTF-8

➤ Adam Kozak

- ✓ Projekt i implementacja frameworka ServerKernel

- ✓ Projekt i implementacja frameworka TwoWayCScKernel

- ✓ Analiza architektury RMI, dekompilacja źródeł Suna i analiza serializacji
- ✓ Projekt komunikacji VMD
- ✓ Testy jednostkowe
 - Testy metodą black box VMDServer
 - Testy metodą white box ServerKernel
- ✓ Analiza i debug fabryki socketów `TwoWaySocketFactory` i zastosowanie jej w projekcie
- ✓ Projekt i implementacja plugina VMDDraw
- Szymon Kupiński
 - ✓ Projekt i implementacja plugina VmdDesktop
 - ✓ Projekt i implementacja plugina VmdTalk
 - ✓ Projekt i implementacja klienta VmdClient
 - ✓ Projekt i implementacja generatora deskryptorów dla pluginów
- Aleksander Stasiak
 - ✓ Projekt i implementacja frameworka ClientKernel
 - ✓ Projekt i implementacja plugina VmdEditor
 - ✓ Projekt i implementacja FileChooser
 - ✓ Analiza i poprawa implementacji biblioteki `TwoWaySocketFactory`
 - ✓ Projekt komunikacji VMD

2. Czym jest VMD

2.1 Podstawowe założenia i funkcjonalność

Definicja podana w tym rozdziale wymaga zdefiniowania pojęć zasobów dla systemu VMD. Zasobem systemu VMD jest pewna współdzielona, czyli utrzymywana na serwerze, informacja o systemie taka jak np. nazwa ikony, położenie ikony, stan pulpitu, ale także obiekty znajdujące się w pamięci jak np. stan systemu plików. Podsumowując, zasoby to wszelkie współdzielone informacje, które są zapisane na dysku lub znajdują się w pamięci serwera. Reprezentują one pewien stan systemu, który może być zmieniony przez pewne zdarzenia, które zajdą w systemie w wyniku działania serwera, lub klienta posiadającego uprawnienia do korzystania z i modyfikowania tych zasobów.

VMD to system rozproszony, który powinien umożliwiać zalogowanie się na wirtualny pulpit przez użytkownika, oraz zarządzanie tym pulpitem. Integralną część pulpitu stanowi jego wizualizacja po stronie klienta. Istnieje możliwość zalogowania się kilku użytkowników na to samo konto, wówczas współdzielą oni zasoby wirtualnego użytkownika systemu.

Na etapie analizy, a później projektowania, twórcy systemu musieli zdefiniować jak w budowanym systemie ma wyglądać korelacja między użytkownikami. Należało zdecydować się na pewien poziom propagacji zmian. Zdecydowano, że współdzielone są jedynie zasoby. Przypomnijmy, że zasoby mają utrwaloną formę. System nie współdzieli zdarzeń lokalnych zachodzących u klienta, które prowadzą do lokalnych zmian zasobów. Jednak utrwalenie zmiany zasobów jest propagowane do wszystkich uprawnionych użytkowników, czyli w naszym wypadku wszystkich użytkowników zalogowanych na to same konto. Utrwaleniem

zmiany zasobów jest np. zapisanie po stronie serwera edytowanego pliku, zmiana położenia ikony na desktopie. Założenia te oznaczają w praktyce, że np. w przypadku podsystemu umożliwiającego rysowanie zmiany w wyglądzie graficznym obrazu desktopu przesyłane są do wszystkich użytkowników, natomiast zmiany położenia kursora myszy u jednego z zalogowanych użytkowników są lokalną cechą systemu dla tego użytkownika. VMD musi umożliwiać korzystanie z zasobów znajdujących się na serwerze, a także ich modyfikację, a wszystko z zachowaniem praw dostępu. Możliwe jest więc ściągnięcie ustawień wirtualnego pulpitu, a także ich zmienianie i zapisywanie po stronie serwera.

Kolejną decyzją, którą należało podjąć na etapie projektowania, a związaną z interakcjami między użytkownikami, było określenie reguły postępowania w przypadku współbieżnego modyfikowania zasobów. Przykładowo co należy zrobić, gdy użytkownicy jednocześnie modyfikują ten sam plik tekstowy? Jak rozwikłać ewentualne niespójności w jego zawartości? Zauważmy, że w ogólności takie współbieżne modyfikowanie zasobów może mieć i pożądane i niepożądane – katastrofalne skutki. W zależności od sytuacji: w programie komunikacyjnym typu „talk” modyfikowanie strumienia informacji jest właśnie główną funkcjonalnością aplikacji. Podobnie mógłby działać program ze współdzieloną tablicą do rysowania- użytkownicy komunikują się „on line” rysując i pisząc na tablicy. Z drugiej strony, można też wyobrazić sobie takie sytuacje, gdy współbieżne modyfikowanie zasobów informacyjnych jest niepożądane. Na przykład, jeżeli piszemy tekst programu nie chcielibyśmy, aby ktoś inny jednocześnie modyfikował ten sam plik. Nie wyklucza to jednak pracy grupowej na zasadzie tur, takiej jak w systemie CVS. Podsumowując, współbieżne modyfikowanie zasobów oraz reakcja na to zależy od natury aplikacji. Dlatego zdecydowano, że system nie

będzie zawierał mechanizmów uspoólniania współbieżnie modyfikowanych zasobów. Informacje o modyfikacjach zasobów będą przekazywane do współbieżnie działających aplikacji, które muszą odpowiednio reagować na to, np. aktualizując lokalną kopię, powiadamiając użytkownika, lub ignorując zmianę.

System VMD posiada budowę modułarną - rozbudowa funkcjonalności następuje poprzez pluginy (co pozwala na łatwe dodawanie nowych funkcjonalności). Podstawowa funkcjonalność serwera systemu polega na logowaniu użytkowników, sprawdzaniu ich uprawnień, ładowaniu pluginów oraz zarządzaniu komunikacją między użytkownikami zalogowanymi na tym samym koncie. Podstawowa funkcjonalność klienta polega na wizualizacji pulpitu. Głównie poprzez część kliencką generują się zdarzenia zmiany stanu zasobów, takie jak zmiany ustawień ikon na pulpicie, dodanie lub usuwanie plików.

2.2 Przegląd podobnych rozwiązań

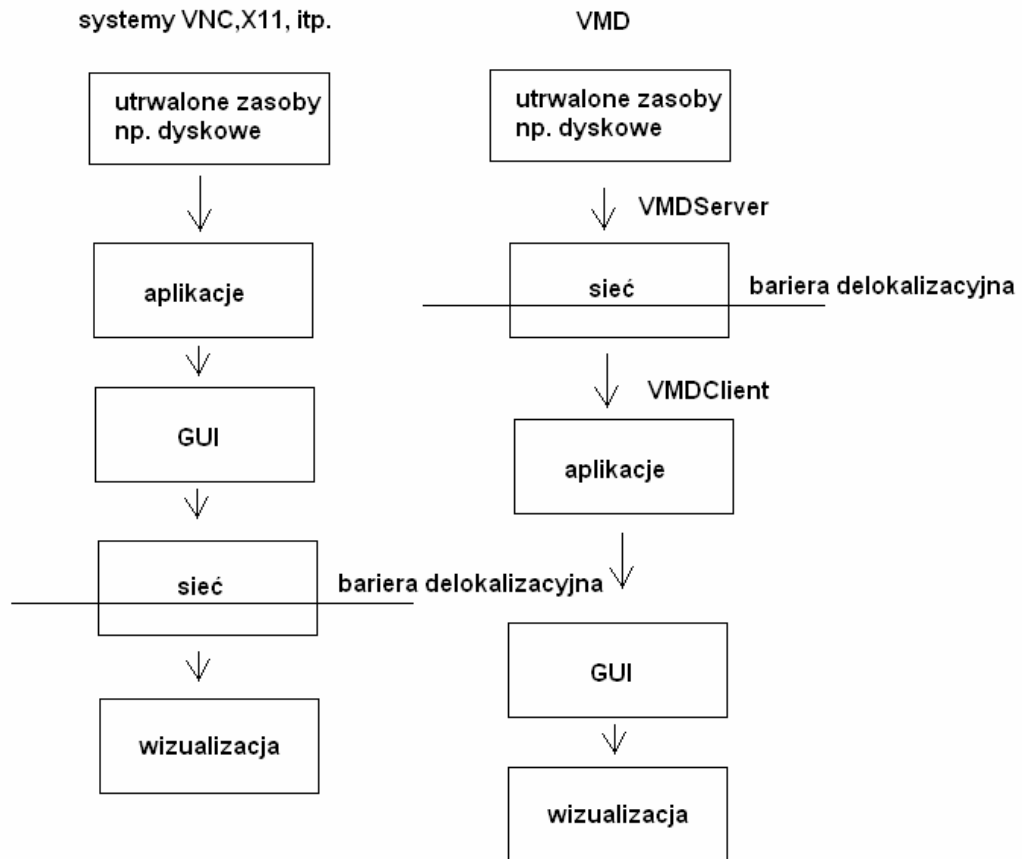
2.2.1 VMD jako mechanizm wirtualizacji zasobów

System VMD można rozważać w szerszym kontekście narzędzi i mechanizmów wirtualizacji zasobów komputerowych. Tak szeroko rozumiany kontekst obejmuje aplikacje takie jak Xen, Vmware, User Mode Linux, export GUI w systemach X11, VNC, My Virtual PC itp. W kolejnych podrozdziałach przeprowadzono porównanie VMD z systemem VNC oraz z systemem Xen. Zdecydowano się na porównanie VMD właśnie z tymi systemami, ponieważ reprezentują one całkiem inne podejście do przedstawionego problemu:

- VNC eksportuje GUI
- Xen używa parawirtualizacji do przedstawienia zasobów

2.2.2 Virtual Network Computing

Przed przystąpieniem do fazy analizy zbadano, czy istnieją podobne rozwiązania do VMD. Ich analiza i ewentualne scenariusze przypadków użycia umożliwiłyby lepsze rozwiązanie postawionego problemu. Godne uwagi i przytoczenia wydaje się rozwiązanie zastosowane w systemie Virtual Network Computing (VNC). VNC to niewielki program - klient wraz z serwerem zajmuje około 300 KB. Rozwiązanie to dostarcza do lokalnego komputera obraz z komputera zdalnego. Pozwala to na zdalne uruchamianie programów na komputerze, na którym działa serwer VNC. Serwer VNC przetwarza obraz i wysyła strumieniem do klienta VNC. Po stronie klienta następuje ponowna wizualizacja. Zdarzenia wykonane po stronie klienta na zasobach serwera są przesyłane i obsługiwane przez serwer. System taki pozwala na korzystanie ze swojego komputera na odległość, wymaga żeby komputer-serwer był włączony i działał na nim serwer VNC. Łatwo zauważyć, że system taki, pozornie podobny do VMD, działa jednak inaczej. Jest on wizualizacją prawdziwego systemu. VMD natomiast jest wizualizacją pewnych abstrakcyjnych zasobów zgromadzonych na serwerze. Klient VMD przesyła zdarzenia do serwera, gdzie zdarzenia powodują zmianę na zasobach. Informacje o tym propagują się do odpowiednich klientów, w postaci zdarzeń. Klient VNC przesyła do serwera prawdziwe przechwycone zdarzenia myszy lub klawiatury, a serwer zwraca mu obraz (wirtualizację) systemu zdalnego. Różnicę tą ilustruje rysunek:



Zasadnicze różnice między systemami to:

- Wizualizacja systemu
 - VNC wizualizuje działanie prawdziwego systemu u klienta,
 - VMD wizualizuje zasoby znajdujące się na serwerze
- Przesyłanie danych
 - Serwer VNC przesyła skompresowany obraz, który jest wizualizowany u klienta VNC
 - Serwer VMD propaguje do swoich klientów zdarzenia

- Zasoby
 - VNC pozwala korzystać zdalnie z prawdziwych zasobów komputera-serwera
 - VMD modyfikuje pewne abstrakcyjne zasoby znajdujące się na serwerze
- współbieżność
 - VNC jest typowym systemem sieciowym
 - VMD jest systemem rozproszonym, pozwala na rozproszone korzystanie z zasobów serwera

2.2.3 System Xen

Xen to monitor maszyn wirtualnych, zaprojektowany i stworzony na Uniwersytecie Cambridge. Przeznaczony jest dla komputerów kompatybilnych z x86. Produkt ma status open source. Rozwijany jest przez firmę XenSource. Xen jako monitor maszyn wirtualnych sam zużywa wyjątkowo mało zasobów systemowych. Jest bardzo wydajny. Systemy operacyjne działające pod kontrolą Xena muszą działać w trybie parawirtualizacji. Parawirtualizacja jest podobna do wirtualizacji, jednak wymaga specyficznego przygotowania uruchamianego wirtualnie systemu operacyjnego. Zmiany polegają głównie na modyfikacji sposobu zarządzania pamięcią przez ten system. Takie rozwiązanie ma swoje wady i zalety. Zaletą jest przede wszystkim wzrost szybkości systemu parawirtualnego. Główną wadą rozwiązania jest konieczność modyfikacji uruchamianego parawirtualnie systemu operacyjnego.

Oto główne różnice między XEN a VMD:

Kompatybilność i wsparcie sprzętowe

- Platforma x86 nie wspiera w naturalny sposób wirtualizacji (systemy VMD)

- Xen jest przeznaczony dla maszyn typu x86, posiada wsparcie ze strony procesorów Intel

Wirtualizacja

- VMD jest systemem zapewniającym wirtualizację pewnego systemu plików i danych, oraz pewnego zbioru instrukcji
- Xen działa w systemie parawirtualizacji, dzięki czemu jest bardzo wydajny

Zasoby pamięciowe

- VMD wymaga uruchomienia wirtualnej maszyny Javy, która ma stosunkowo wysokie wymagania zasobowe co do pamięci operacyjnej
- Xen słynie ze swojej oszczędności, zużywa niewielkie zasoby systemowe

Przenośność

- VMD jest bardzo przenośny – działa praktycznie w każdym środowisku operacyjnym, dzięki zaimplementowaniu go w Javie
- Xen wymaga modyfikacji systemu, który ma monitorować, w celu uzyskania parawirtualizacji, jest to dość problemowe i sprowadza się do tego, że należy posiadać konkretną wersję systemu uruchamianego na maszynie wirtualnej

Dostępność systemu, prawa autorskie

- System VMD jest własnością Politechniki Poznańskiej, natomiast prawa autorskie do niego posiada grupa inżynierska w składzie powyższym
- System Xen jest systemem typu open source rozwijanym przez firmę XenSource

Możliwości

- System VMD pozwala na zarządzanie pewnym wirtualnym pulpitem, oraz jego danymi
- System Xen pozwala emulować systemy z dostępnym kodem, głównie systemy Free Libre/Open Source Software, ponieważ system ten wymaga przygotowania do parawirtualizacji

Zasoby dyskowe

- System VMD korzysta z zasobów zdalnych zgromadzonych na serwerze, wymagania dyskowe są więc minimalne.
- System Xen wymaga dostępności specjalnie zmodyfikowanego systemu operacyjnego (przygotowanego do parawirtualizacji), z tego powodu wymagania dyskowe są znaczne

Dla Xena są dostępne następujące systemy operacyjne:

- Linux
- NetBSD
- FreeBSD
- Windows XP – korzystanie z tego systemu wymaga jednak podpisania odpowiedniej umowy z firmą Microsoft

3. Technologia

3.1 Wybór rozwiązań programistycznych

W trakcie przygotowywania pracy inżynierskiej powstały 2 rozwiązania programistyczne. Pierwsze uzyskane oprogramowanie nie zadowalało jego twórców, idea została porzucona w lipcu 2005 roku. Druga wersja programu powstawała od lipca 2005 roku. Twórcy uważają jednak, że zmiana technologii

programistycznej była wysoce wskazana. Zadaniem niniejszego rozdziału jest uzasadnić tę tezę. Na etapie projektowania wykorzystano język UML. UML (Unified Modeling Language) jest to zunifikowany język modelowania, oparty o pojęcia takie jak obiekty, klasy, atrybuty, związki, metody dziedziczenie i inne. Język UML bardzo dobrze nadaje się do obiektowego zamodelowania problemu. Język UML jest notacją pośrednią między ludzkim rozumieniem struktury i problemu a oprogramowaniem. W lipcu 2005 roku stworzono diagram klas systemu (class diagrams). Są one odmianą klasycznych diagramów encja-związek (entity-relationship).

Jako język programowania wybrano Javę. Java to wprowadzony w 1995 roku przez Sun Microsystems obiektowy język programowania. Niezależność od platformy, czytelność i prostota były niewątpliwym atutem wyboru tego rozwiązania. Wysoki poziom abstrakcji w tym języku pozwala uniezależnić się od budowy sieci komputerowej. Jako środowisko pracy twórcy wybrali darmowe środowisko Eclipse. Eclipse to uniwersalne środowisko IDE, które może być wykorzystane do programowania w wielu językach, jednak jego głównym zastosowaniem jest Java. Obsługa Javy jest w zintegrowana. Eclipse powstał jako uniwersalna platforma i bardzo dobrze spełnia swoje zadanie. Eclipse ma status open-source, został jednak stworzony przez firmę IBM, która wydała na ten projekt ponad 40 milionów dolarów. Eclipse został wzbogacony o plugin RMI genady.net, który wspomaga generowanie stubów w technologii RMI, oraz pomaga w debugowaniu wątków RMI. Autorzy otrzymali pozwolenie od autora na wykorzystanie tej technologii w naszej pracy. Początkowy projekt, który rozwijał się od 1 marca 2005 roku do 30 czerwca 2005 roku opierał się na protokole operującym bezpośrednio na socketach. Wszystkie połączenia były tworzone przez sockety, komendy były przesyłane jako

strumień bajtów. Na podstawie nazwy komendy mechanizm refleksji wyszukiwał odpowiednią klasę komendy. Refleksja jest to mechanizm, zaimplementowany w javie, który pozwala na identyfikację typu w czasie wykonywania (RTTI run-time type identification). Pozwala on także na operowanie na metodach, polach i konstruktorach klasy przez bezpośredni dostęp przez nazwy za pomocą metaklasy (klasy reprezentującej i opisującej inne klasy).

Jednocześnie grupa prowadziła poszukiwania technologii bardziej odpowiedniej do problemu, który miał być rozwiązany. Na przełomie czerwca i lipca grupa podjęła się próby zaadoptowania technologii J2EE na potrzeby rozwiązania problemu. Java 2 Enterprise Edition jest zbiorem standardów tworzenia oprogramowania, zaproponowanym w celu podwyższenia jakości oprogramowania biznesowego. Grupa skupiła się na poznaniu technologii JMS (Java Message Service), która miała być wykorzystana w celu zapewnienia komunikacji między serwerem a klientami. JMS jest swego rodzaju uniwersalnym interfejsem do systemów kolejkowych, pozwala ona na obsługę komunikacji asynchronicznej. Niestety rozwiązanie to musiało zostać porzucone, ze względu na bezpośrednią możliwość przesyłania wiadomości jedynie od klienta do serwera. System VMD musiał umożliwiać wysyłanie wiadomości zarówno przez serwer, jak i przez klienta. Dodatkowym założeniem autorów było uniezależnienie się od zabezpieczeń sieciowych, takich jak ściany ogniowe. W październiku 2005 uwaga autorów skupiła się na technologii Remote Method Invocation (z której korzysta też pośrednio J2EE). Technologia RMI pozwala obiektowi istniejącemu na jednym komputerze zachowywać się tak, jakby istniał na innym komputerze. Aby dobrze zrozumieć mechanizm RMI, na potrzeby projektu zdekompilowano źródła klas SUNa, odpowiedzialne za mechanizmy RMI, ich prześledzenie pozwoliło autorom

na dokładne poznanie mechanizmu i upewnieniu się w jego wyborze. Oto przesłanki, które zadecydowały o zmianie technologii na RMI:

- jest technologią dedykowaną dla środowisk rozproszonych w Javie
- pozwala na rozpatrywanie problemu na wyższym poziomie abstrakcji, niż połączeń na socketach
- umożliwia wysyłanie całych zserializowanych obiektów, a nie tylko tekstowych ciągów, które są parsowane i analizowane u odbiorcy, jak w poprzednim rozwiązaniu
- pozwala na pisanie programu tak, jakby wykonywał się on lokalnie, należy tylko zadbać o to, aby obiekty działające zdalnie implementowały interfejs Remote
- modyfikacja sposobu budowania połączeń sieciowych pozwoliła autorom rozwiązać problem z firewallami, które mogą znajdować się po stronie klienta, gdyż cały system musi umożliwiać wykonywanie zdalnych metod także u takich klientów

3.2 Remote Method Invocation (RMI)

3.2.1 Wprowadzenie do technologii RMI

RMI jest technologią Javy pozwalającą na pisanie programów rozproszonych, których struktura semantyczna i syntaktyczna jest taka sama jak programów uruchamianych na jednej JVM. Jedną z głównych zalet RMI jest prostota, która pozwala programiście abstrahować od tego, czy dany obiekt jest obiektem lokalnym czy zdalnym. Każdy obiekt jest traktowany jednakowo.

Warto tu wspomnieć o alternatywnym rozwiązaniu, którego dostarcza CORBA. CORBA (Common Object Request Broker Architecture) jest to obiektowy standard

dla systemów rozproszonych. Obiekty nie tylko mogą się znajdować na różnych maszynach i posiadać implementacje w różnych językach. RMI umożliwia uruchomienie usług nie należących do standardu Java, jednak te usługi muszą posiadać specjalne obiekty opakowujące. Służy do tego technologia JNI. Standard CORBA rozwiązuje ten problem sam. RMI jest standardem łatwiejszym, bardziej intuicyjnym, dedykowanym dla Javy. Na etapie projektowania autorzy uznali, że będzie lepszym rozwiązaniem niż CORBA ze względu na implementacje obu stron komunikacyjnych na platformie Java.

RMI polega na oddzieleniu interfejsu obiektu od jego implementacji. Implementacja obiektu znajduje się na serwerze RMI, natomiast klient otrzymuje specjalny obiekt dostępu do obiektu serwerowego. RMI dba o zwracanie wyników wykonania do klienta, o komunikację sieciową, szczegóły dotyczące protokołu komunikacji. Technologia ta jest także odpowiedzialna za serializację i deserializację przesyłanych obiektów. W celu użycia technologii RMI należy napisać interfejs obiektu zdalnego, jego właściwą implementację oraz wygenerowaną implementację odpowiedzialną za komunikację zdalną:

- Service Implementation – uruchamiana na serwerze implementacja działania zdalnego obiektu
- Service Proxy - pośredniczący obiekt znajdujący się u klienta.

Program klienta wykonuje metody obiektu Service Proxy, który zwraca mu wyniki obiektu znajdującego się na serwerze.

RMI składa się z trzech warstw:

- stubs (Bielecki tłumaczy w swojej książce[3] stub jako namiastka- są to obiekty namiastki)

- remote reference layer- czyli warstwa referencji
- transport layer – czyli warstwa transportowa

Namiastki obiektów, czyli „stuby”, używane są przez klienta. Klient chcąc wykonać metodę na serwerze, wykonuje metodę o tej samej nazwie w lokalnej namiastce. To namiastka odpowiedzialna jest za wykonanie metody obiektu na serwerze i zwrócenie wyniku do klienta. Z perspektywy klienta nie ma różnicy, czy obiekt wykonał się lokalnie, czy zdalnie, otrzymuje on wyniki tak jakby wykonał metodę lokalną.

Warstwa referencji odpowiada za zarządzanie referencjami do obiektów przekazywanymi między serwerem a klientem. Warstwa referencji przekazuje referencje do obiektu, lub serializuje je z jednej strony i deserializuje je z drugiej strony. Namiastka wykorzystuje mechanizm refleksji do uruchamiania metod na serwerze. Metoda `invoke(obiekt, parametry[])` klasy `Method` jest wywoływana na referencji zdalnej obiektu w celu wykonania reprezentowanej metody na podanym obiekcie przekazując do niej tablice parametrów.

Warstwa transportowa używa protokołu Java Remote Method Protocol (JRMP), jego implementacja domyślnie oparta jest o protokół TCP. Protokół może być zastąpiony jednak bezpołączeniowym odpowiednikiem - protokołem UDP. Warstwa ta jest odpowiedzialna za komunikację między klientem a serwerem.

Oto przykład interfejsu zdalnego:

```
import java.rmi.*;
public interface Name extends Remote {
    String getDescription() throws RemoteException;
}
```

Klasa implementująca interfejs zdalny będzie dziedziczyć (podobnie jak sam interfejs zdalny) z interfejsu `Remote`. Metody wykonywane zdalnie muszą deklarować zgłaszanie wyjątku `RemoteException` (wyjątek sygnalizujący błąd wywołania zdalnego). W standardowym użytkowaniu technologii RMI klient odnajduje zdalne usługi za pomocą nazw (naming). Usługa działa na znanym porcie i hoście, domyślnie jest to port 1099. Program kliencki korzysta z klasy `Naming` i jej metody statycznej `lookup()`. Łącuch identyfikujący serwer ma postać:

```
rmi://<nazwa_hosta>[:<port>]/nazwa_usługi
```

Zauważmy, że postać łańcucha jest zbliżona do standardów łańcucha URL.

3.2.2 Mechanizm serializacji i przekazywania referencji obiektów RMI

Technologia RMI zapewnia automatyczne odtwarzanie parametrów i wyników metod, wyjątków oraz pól klasowych między dwoma punktami zdalnej komunikacji. Wymagane jest przy tym, żeby każdy parametr, wynik, wyjątek i pole klasy spełniało jedno z założeń:

- implementowało interfejs `Serializable`
- implementowało interfejs `Remote` (obiekty przekazywane przez referencję zdalną).

Obiekty implementujące `Serializable` są przekazywane przez wartość (wszystkie ich pola podlegają rekurencyjnemu poddaniu identycznej procedury) czyli następuje zamiana obiektu w strumień danych, który jest przekazywany następnie przez sieć.

Obiekty implementujące `Remote` są przekazywane przez referencję zdalną. Oznacza to, że obiekt przekazany zdalnie dokonuje operacji zdalnie na

„rodzimy” lokalnym obiekcie. Ze względu jednak na fakt iż ich przekazanie odbywa się wizualnie w identyczny sposób jak obiektów serializowalnych nie mogliśmy zrozumieć tego mechanizmu dopóki nie znaleźliśmy odpowiedzi w dekompilacji źródeł firmy Sun.

W klasie `MarshalOutputStream` z pakietu `sun.rmi.server` znaleźliśmy wyjaśnienie tego mechanizmu. Klasa ta jest potomkiem klasy `ObjectOutputStream` z pakietu `java.io`, która jest odpowiedzialna za standardową serializację obiektów w javie. Znaleziona klasa stanowi mechanizm serializacji obiektów zastosowany w RMI. Jej najważniejszą metodą zapewniającą właściwe przekazywanie obiektów RMI jest `replaceObject(Object obj)`:

```
protected final Object replaceObject(Object obj)
throws IOException
{
    if ((obj instanceof Remote) && !(obj instanceof RemoteStub)) {
        Target target = ObjectTable.getTarget((Remote) obj);
        if (target != null)
            return target.getStub();
    }
    return obj;
}
```

Metoda ta jest chroniona (`protected`) i nadpisuje metodę klasy `ObjectOutputStream`. Jest ona wywoływana dla każdego obiektu podlegającego serializacji RMI na każdej głębokości rekurencji. Jeśli serializowany obiekt implementuje interfejs `Remote`, nie jest już namiastką sieciową (`RemoteStub`) oraz jest zarejestrowany w serwerze RMI (`ObjectTable.getTarget((Remote) obj)` jest różne od `null`) to podmienia serializowany obiekt na jego namiastkę sieciową (serializowana jest sama namiastka posiadająca referencję zdalną). Powoduje to, że obiekty zdalne, nawet te które się znajdują „głęboko” w podklasach zostaną zawsze

przekazane jako obiekty zdalne. Obiekty które nie zostaną podmienione przez `replaceObject(Object obj)` podlegają standardowej serializacji.

3.2.3 Schemat nawiązania połączenia RMI

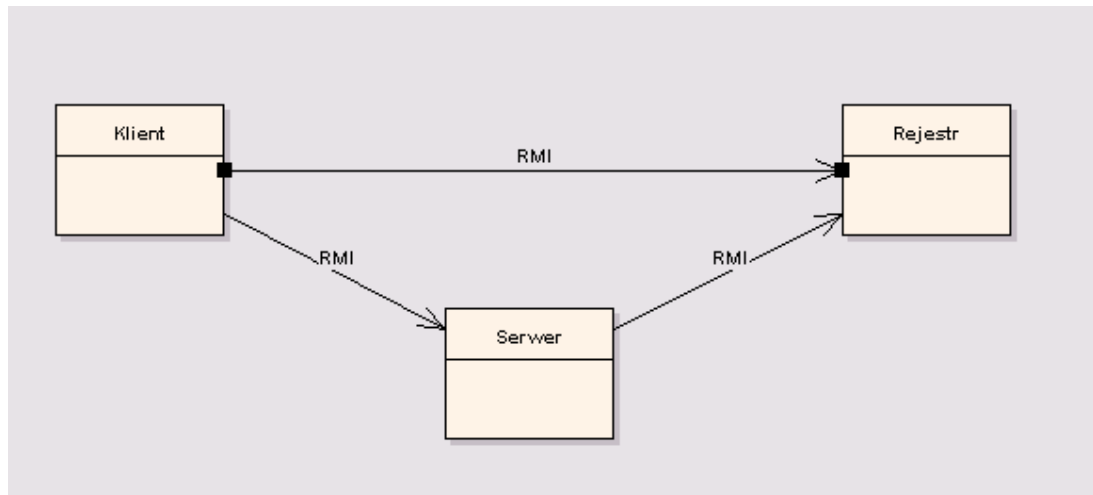
Do przechowywania i udostępniania referencji zdalnych służy rejestr RMI. Ogólny schemat pobierania referencji z rejestru oraz wykonania metody zdalnej w standardowej komunikacji RMI prezentuje się następująco:

- Klient chce wykonać metodę zdalnego obiektu na serwerze. Adres usługi jest znany,
- Klient uzyskuje referencję do zdalnego obiektu rejestru na serwerze za pomocą metody `Registry LocateRegistry.getRegistry(port);`
- Klient odwołuje się do rejestru w celu uzyskania referencji do obiektu zdalnego poprzez nazwę. Rejestr posiada HashMapę zdalnych obiektów typu nazwa → zdalny obiekt
- Klient otrzymuje namiastkę do obiektu zdalnego
- Klient wykonuje zdalnie odpowiednią metodę

W dostarczonej przez SUN implementacji `RegistryImpl` rozszerza klasę `RemoteServer` oraz implementuje interfejs `Registry`. Pełni więc równocześnie funkcję serwera usług RMI jak i rejestru. Oczywiście funkcje te mogą być, w razie potrzeby rozdzielone:

```
public class RegistryImpl extends RemoteServer implements Registry
{
    //rozszerzenie klasy serwera
    //implementacja interfejsu rejestru
}
```

Schemat nawiązania połączenia RMI:



rys. 1 Schemat nawiązania połączenia RMI

Rejestr jest jedynym obiektem zdalnym dostępnym na serwerze usług. W rejestrze znajdują się inne dostępne obiekty zdalne. Muszą być one najpierw zarejestrowane w rejestrze. Rejestracji dokonuje się przez metodę `bind()`, która dodaje referencję do obiektu do `HashMap`.

3.2.4 Ogólny schemat nawiązania połączenia RMI w VMD

W VMD oryginalne założenia SUNa względem RMI zostały zmodyfikowane i dostosowane do wizji twórców. Rejestr posiada tylko jedną metodę do pobrania zdalnego obiektu - `getRemoteServer()`. Oto interfejs zastosowanego rejestru:

```
public interface Registry extends Remote {  
    public Remote getRemoteServer() throws RemoteException,  
        NotBoundException;  
    public static final int REGISTRY_PORT = 1099;  
}
```

Oraz szkielet ideowy implementacji:

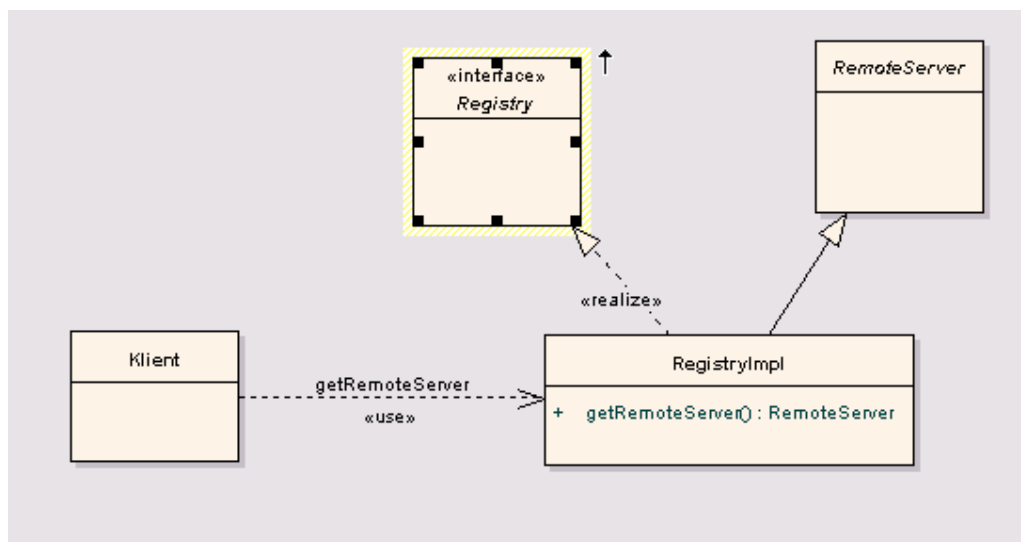
```
public class RegistryImpl extends java.rmi.server.RemoteServer  
implements Registry {  
    private RemoteServer remoteServer = null;
```

```

public Remote getRemoteServer() throws RemoteException,
    NotBoundException {
    if (remoteServer == null)
        throw new NotBoundException();
    return remoteServer;
}
}

```

Pobrany przez `getRemoteServer()` obiekt typu `RemoteServer` zapewnia użytkownikowi 1 metodę `login`, za pomocą której klient loguje się na serwerze. Oto schemat nawiązania połączenia RMI w systemie VMD:



rys. 2 Schemat nawiązania połączenia RMI w VMD

3.2.5 SocketFactory i TwoWaySocketFactory w RMI

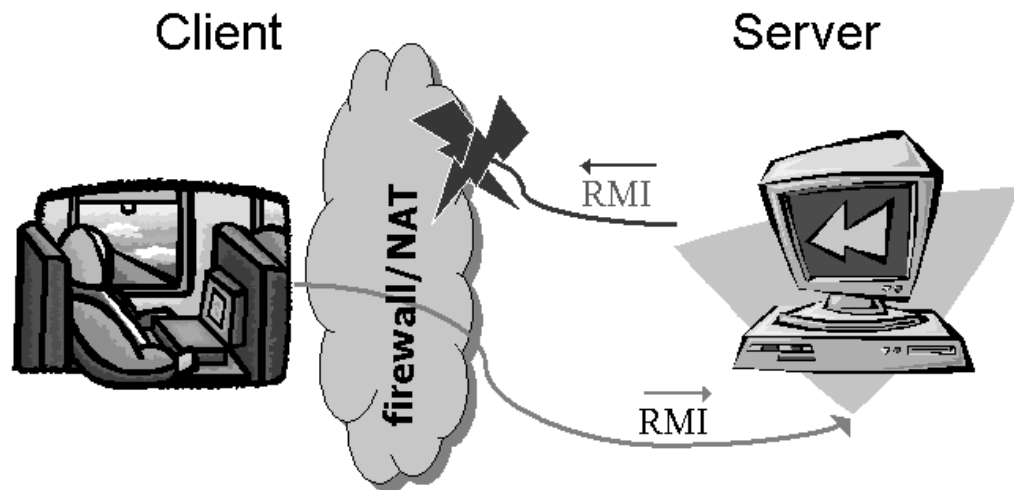
Założenia, jakie przyjęli twórcy systemu VMD na etapie projektowania wobec komunikacji były następujące:

- Dwustronność komunikacji (wywołania metod)
- Klient może przebywać za firewalleń/NATem

Standardowe mechanizmy RMI nie spełniają tych założeń. Twórcy systemu musieli bliżej przyjrzeć się klasie `SocketFactory`, odpowiedzialnej za komunikację w technologii RMI.

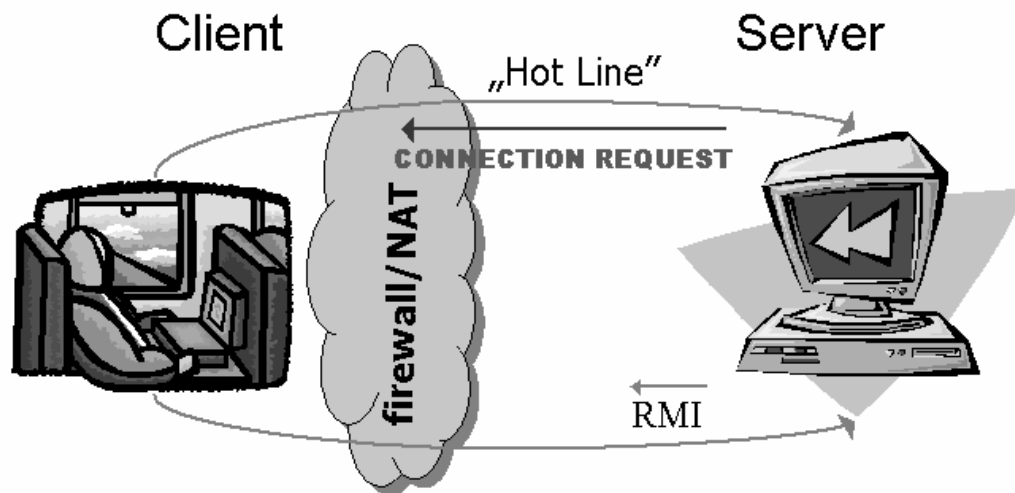
`SocketFactory` jest mechanizmem w RMI, który zajmuje się tworzeniem połączeń opartych o sockety. Fabryka socketów jest parametrem konstruktora serwera RMI (lub jest przekazywana przy tworzeniu serwera poprzez ustalenie domyślnej fabryki socketów poprzez statyczną metodę klasy `SocketFactory`). Odpowiednia fabryka socketów dba o sposób tworzenia połączeń. Domyślna fabryka socketów działa na protokole TCP, oraz korzysta z pakietu klasy `Socket` w pakiecie `java.net`. Nie spełniała ona jednak wymagań programistów systemu VMD, ponieważ nie umożliwiała udostępniania obiektów zdalnych u klienta, który posiada firewall/NAT. Poszukiwanie odpowiedniej (wolnej od ograniczeń prawnych i licencyjnych) fabryki socketów trwało około miesiąca. Poszukiwania w dedykowanej literaturze, oraz w Internecie, zakończyły się sukcesem. W projekcie wykorzystano `TwoWaySocketFactory` [11]. Autor (Tim Taylor, pracownik firmy Computer System Services) zastrzegł tylko, żeby w kodzie źródłowym znajdowała się informacja o jego autorstwie. Fabryka ta rozwiązuje problem firewalla/NAT u klienta.

Działanie standardowej fabryki:



rys. 3 Działanie standardowej fabryki socketów w RMI

Działanie fabryki `TwoWaySocketFactory`:



rys. 4 Działanie fabryki `TwoWaySocketFactory`

Fabryka ta działa w trybie analogicznym to trybu pasywnego protokołu FTP.

Rozwiązanie problemu firewallu:

- Klient nasłuchuje na specjalnym połączeniu („hot line” – kanał sygnalizujący)
- Jeśli serwer chce się połączyć z klientem zgłasza żądanie poprzez istniejący cały czas kanał sygnalizujący
- Klient inicjuje połączenie do serwera, przez które następuje wywołanie metody zdalnej Dokumentacja

3.3 Dokumentacja

3.3.1 Wprowadzenie

Każde oprogramowanie wymaga dokumentacji, która przyspiesza, ułatwia i czasem w ogóle umożliwia zrozumienie jego działania. Ze względu na możliwość rozbudowy napisanego przez nas frameworka, szczególnie niezbędna jest techniczna dokumentacja programistyczna, dzięki której potencjalni programiści będą mieli ułatwioną naukę zaprojektowanej architektury. Dokumentacja ta powinna obejmować objaśnienie architektury, kontrakty na implementacje interfejsów (jakie system przyjmuje założenia co do funkcjonowania implementacji), opisy działania poszczególnych metod w klasach i sposób interakcji między nimi. Dokumentacja została sporządzona w języku polskim ze względu na przynależność do niniejszej pracy dyplomowej.

3.3.2 JavaDoc

Przyjętą formą dokumentacji frameworka została technologia JavaDoc – specjalny język w komentarzach języka java (niewidoczny dla kompilatora), który opisuje zachowanie poszczególnych interfejsów, klas, metod i pól. Za pomocą JavaDoc można wygenerować stronę w HTML dokumentującą cały projekt. Tej właśnie techniki używa opisany pakiet Maven do generowania dokumentacji kodu

źródłowego. Opis metody w JavaDoc umieszcza się bezpośrednio przed deklaracją metody w sposób następujący:

```
/**
 * Ogólny opis działania metody.
 * @param nazwa(1) - opis pierwszego parametru metody
 * @param nazwa(n) - opis n-tego parametru metody
 * @return - opis zwracanego wyniku
 * @throws nazwaWyjątku(1) - opis rzucanego wyjątku
 */
```

We wszystkich opisach można używać tagów HTML, co ułatwia formatowanie dokumentacji na poziomie opisywania kodu źródłowego. Opis klasy/interfejsu w JavaDoc wygląda podobnie (umieszcza się przed deklaracją klasy):

```
/**
 * Ogólny opis funkcjonalności i przeznaczenia klasy.
 * @author nazwa autora - ew. nota o autorze
 */
```

3.4 Standard kodowania

Przyjętym standardem kodowania (zestawem reguł dotyczących stylu pisanie kodu źródłowego) był szeroko przyjęty standard dla języka java, dla którego moduł automatycznego formatowania posiadał sam Eclipse. Przyjęcie jednolitej konwencji pisanie kodu źródłowego pozwala na szybsze i łatwiejsze czytanie cudzego kodu, co może być również pomocne dla programistów frameworka.

Jako format kodowania znaków użyto UTF-8. Standard ten przesyła kody ASCII bez zmian. Kody powyżej 127 są modyfikowane. Dzięki użyciu tego standardu, polskie teksty powiększają się średnio tylko o kilkanaście procent, zamiast wzrosnąć dwukrotnie lub czterokrotnie, jak w przypadku użycia UTF-16 i UTF-32.

3.5 Wersjonowanie

Systemy wersjonowania umożliwiają systematyczną pracę grupową. W przypadku rocznej, czteroosobowej pracy inżynierskiej praca z systemem wersjonującym jest niezbędna. Wyniki pracy studentów-programistów VMD zapisywane były w systemie Subversion (SVN). Subversion zapewnia:

- Bezpieczeństwo – przez bezpieczeństwo rozumie się zabezpieczenie repozytorium przed skasowaniem, a także zabezpieczenie historii projektu i zmian. Hasła użytkowników są utrzymywane na serwerze, plik z hasłami jest haszowany. Zalogowanie wymaga podania hasła i nazwy użytkownika
- Historie zmian – jest utrzymywana na serwerze SVN, w razie potrzeby można powrócić do dowolnej wersji projektu,
- Współbieżność – uzyskuje się poprzez istnienie jednego współdzielonego repozytorium, użytkownicy zatwierdzają wprowadzone zmiany (commit) i uaktualniają projekt o zmiany wprowadzone przez innych użytkowników (update)
- Oszczędność miejsca – na serwerze nie są utrzymywane wszystkie wersje projektów (algorytm odtwarzania wersji jest oparty na zmianach dokonanych między kolejnymi wersjami). Do danej wersji projektu można wrócić dzięki historii zmian projektu,
- Dobrą organizację pracy.

Warto dodać, że system SVN realizuje ideę systemu multiple-checkout. System multiple-checkout jest systemem, w którym użytkownicy mogą dowolnie współbieżnie modyfikować pliki w lokalnej kopii repozytorium. Zmiany w repozytorium wprowadza się za pomocą polecenia commit. System SVN, jako system multiple-checkout, dba jedynie o uszeregowanie operacji commit i kontrolę

ewentualnych konfliktów. Z punktu widzenia systemu wersjonującego ważna jest kolejność operacji commit wykonanych przez użytkowników. W przypadku próby zapisania na serwerze wersji starszej wersji pliku, niż istnieje w repozytorium, użytkownik zostanie poproszony o połączenie (uspójnienie) tych dwóch wersji. Dobra organizacja i podział pracy pozwoliły autorom VMD zminimalizować liczbę takich sytuacji.

Środowisko Eclipse bardzo dobrze współpracuje z systemem wersjonowania SVN, dzięki dedykowanym do tego zadania pluginom. Autorzy wykorzystali plugin Subclipse 0.9.37, który pozwolił na wygodny dostęp do repozytorium z poziomu eksploratora pakietów środowiska Eclipse. Pomocny okazał się także graficzny interfejs uspójniania dwóch wersji plików. Autorzy zdecydowali się wybrać SVN, niż popularniejszy CVS, ponieważ SVN, w przeciwieństwie do CVS, pozwala wersjonować katalogi. Projekt i pakiety są reprezentowane w Javie jako katalogi. W przypadku VMD, gdzie struktura systemu – zarówno klasy, pakiety, jak i całe projekty, były często modyfikowane wybór SVN wydaje się uzasadniony.

3.6 Maven

Pracę programistów nad projektem VMD wspomagał system Maven. Jest to napisana w Javie aplikacja, wolna od ograniczeń prawnych (open source), której głównymi zadaniami jest:

- Wspomaganie zarządzaniem wytwarzania oprogramowania
- Zapewnianie jakości oprogramowania

Główną jednostką programistyczną, na której opiera się Maven, jest projekt. Wszystkie operacje, które wykonuje Maven są więc wykonywane w kontekście całego projektu. Projekt jest zdefiniowany w pliku project.xml. Posiada on swoją

nazwę, opis, identyfikator, adresy list mailingowych, listę programistów. Maven umożliwia dbanie o powiązania projektu, takie jak biblioteki i pliki *.jar. Wystarczy wpisać te zależności do pliku project.xml.

System ten wspomagał tworzenie projektu VMD poprzez dbanie o poprawność stylu i kodu Javy. Generuje on raporty, które opisują różne błędy związane ze stylem i poprawnością kodu. Na podstawie tych raportów i wprowadzanych poprawek udało się osiągnąć wysoki standard kodu. Sprzyjało to podwyższeniu zrozumiałości kodu przez grupę, i w ten sposób przyczyniło się do przyspieszenia prac nad projektem. Maven ułatwia też wyszukanie duplikatów w kodzie, które za pomocą metod refaktoryzacyjnych typu *extract method* można było usunąć. Jest to ważna funkcjonalność, ponieważ podczas pracy grupowej na tych samych pakietach i klasach programiści często nieświadomie przyczyniali się do duplikowania już utworzonego kodu. Pozbycie się duplikatów kodu pozwoliło osiągnąć bardziej czytelny i krótszy kod. Dodatkowo Maven wygenerował stronę projektu składającą się z danych o projekcie i raportów dotyczących kodu. Wybór Mavena był jednym z trafniejszych wyborów dokonanych przez grupę inżynierską na etapie planowania projektu.

4. Architektura systemu VMD

4.1 Wprowadzenie

Architektura VMD opiera się na stworzonym przez nas *szkielecie* aplikacji klient-serwer bazującej na „obustronnym” RMI. System VMD został podzielony na etapie analizy na następujące, samodzielne projekty:

- ClientKernel – zbiór podstawowych bibliotek do implementacji klienta (kliencka część frameworka)

- ServerKernel – zbiór podstawowych bibliotek do implementacji serwera (serwerowa część frameworka)
- TwoWayCScKernel (*Two Way Client-Server Communication Kernel*) – implementacja dwustronnej komunikacji, zdalne wykonywanie obiektów, zdalny system plików i strumieni (część frameworka współdzielona między serwerem i klientem)
- VMDClient – konkretna implementacja klienta
- VMDServer – konkretna implementacja serwera
- projekty pluginów, tj. aplikacji

Wyodrębnienie bibliotek serwera (ServerKernel) od jego konkretnej implementacji (VMDServer), oraz bibliotek klienta (ClientKernel) od jego konkretnej implementacji (VMDClient) powoduje, że system jest bardzo elastyczny. Przez elastyczność rozumiemy łatwość wprowadzania zmian i modyfikacji systemu, a także możliwość wydajnej, równoległej pracy wielu programistów. Napisany przez nas szkielet pozwala na wykorzystanie gotowej architektury praktycznie w dowolnych aplikacjach klient-serwer, które bazują na połączeniu dwustronnym (taką implementacją mógłby być np. dedykowany komunikator internetowy, serwer obliczeń rozproszonych lub system wymiany plików). Osiągamy to dzięki rozdzieleniu zagadnień na osobne projekty i wprowadzeniu dużego poziomu abstrakcji.

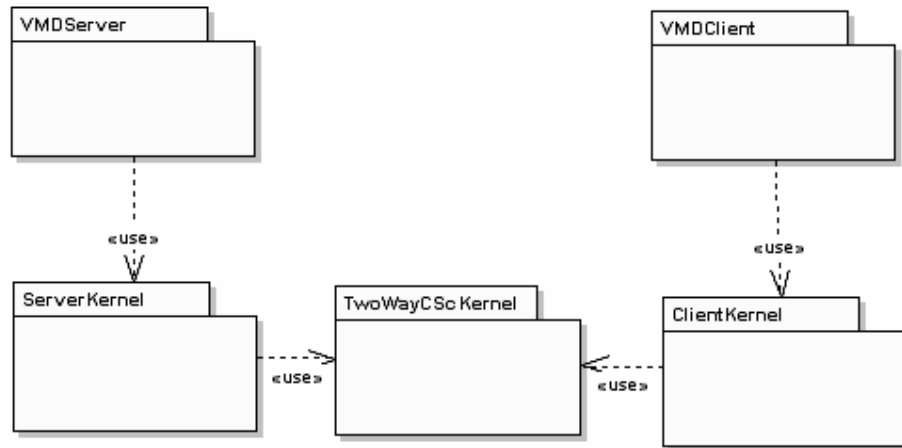
Wydzielenie aplikacji w postaci pluginów powoduje bardzo łatwą rozszerzalność systemu. Dodatkowo „architektura pluginowa” powoduje, że system może zostać użyty do rozwiązywania innych problemów rozproszonych, system taki jest bardzo ogólny. Nie jest on dedykowany konkretnemu celowi i zadaniu. Dla wytworzenia pewnej funkcjonalności (rozwiązania konkretnego problemu) należy

zaimplementować odpowiednie pluginy i związać ze sobą parę klient-serwer. Każdą dodatkową funkcjonalność implementuje się jako plugin. Dodatkowo Java zapewnia przenośność między platformami. Te wszystkie cechy powodują, że system jest

- Przenośny
- Łatwo rozszerzalny
- Elastyczny
- Ogólny

W tej pracy inżynierskiej VMDClient i VMDServer są konkretnymi implementacjami napisanego szkieletu. Autorzy chcą jednak zwrócić uwagę, że zaprojektowany system może być z powodzeniem wydajnie wykorzystany także do rozwiązania innych problemów w środowiskach rozproszonych.

Oto schemat podziału systemu na niezależne, w sensie logicznym, projekty:



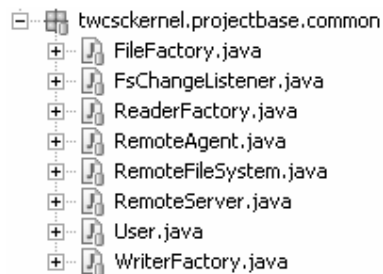
rys. 5 Podział systemu VMD na projekty

4.2 Architektura współdzielonego jądra frameworka - TwoWayCScKernel

4.2.1 Interfejsy zdalne

Interfejsy zdalne, zgodnie ze wcześniejszym opisem, to interfejsy widoczne zarówno u klienta jak i serwera. Po stronie serwerowej są implementowane, natomiast po stronie klienckiej poprzez namiastki umożliwiają wykonywanie metod zdalnych. Wszystkie interfejsy zdalne zgrupowaliśmy w pakiecie

`twcsckernel.projectbase.common`:



rys. 6 Pakiet `twcsckernel.projectbase.common`

Opisy poszczególnych interfejsów:

- `RemoteServer` - interfejs ten jest definicją serwera logowania, którą klient pobiera z rejestru serwera RMI. Posiada jedną metodę logowania: `login(name, password)` która zwraca obiekt reprezentujący użytkownika lub **null**, jeśli wystąpił błąd logowania (niepoprawne dane).
- `User` – zapewnia interfejs zdalny użytkownika. Ze względu na podstawową wagę tego interfejsu zamieszczamy opis metody:
 - `listClientPlugins()` – listuje dostępne pluginy
 - `downloadClientPlugin(clientPackagePath)` – ściąga do klienta binaria plugina wskazanego w ścieżce pakietowej
 - `getRemotePluginAgent(clientPackagePath, clientAgent)` - pobiera agenta zdalnego serwerowej części wskazanego plugina wraz z przekazaniem temu pluginowi klienckiego agenta zdalnego (wymiana agentów w celu obustronnej komunikacji)
 - `getUserFileSystem()` - klient pobiera interfejs zdalnego systemu plików (obiekt zdalny)
 - `setFsChangeListener(fsChangeListener)` – klient ustawia pewien własny obiekt nasłuchujący („listener”) zmian w serwerowym systemie plików
 - `logout()` – metoda pozwana na wylogowanie użytkownika
 - `ping()` – ping jest metodą wykorzystywaną jako uaktualnienie znacznika czasowego ostatniego wywołania od klienta. Znacznik czasowy wykorzystywany jest przy ew. zwolnieniu zasobów i wylogowaniu użytkownika w przypadku zbyt długiej bezczynności.

- `getActivityTimeout()` - pobranie maksymalnego czasu nieaktywności (w milisekundach) w którym serwer daje gwarancję, że użytkownik nie zostanie wylogowany „siłowo”.
- `RemoteAgent` – jest definicją agenta zdalnego plugina (zarówno klienckiej jak i serwerowej części)
- `FileFactory` – fabryka plików zdalnych
- `ReaderFactory`, `WriterFactory` – fabryki strumieni zdalnych (wejściowych i wyjściowych)
- `FsChangeListener` – interfejs obiektu przekazywanego do serwera w celu informowania o zmianach w systemie plików („listener”)

W javie powszechnym określeniem na obiekt nasłuchujący jest „listener” – określenie to przyjęło się w żargonie programistycznym, więc dalej dla określenia obiektu nasłuchującego używany jest właśnie ten termin.

4.2.2 System plików zdalnych

Zimplementowany w ramach pracy inżynierskiej szkielet umożliwia korzystanie z plików zdalnych niemalże w ten sam sposób jak z plików lokalnych, natomiast strumienie plików zdalnych są obsługiwane w identyczny sposób co dowolne inne strumienie w javie (co umożliwia wykorzystanie ogromnego zbioru gotowych klas pomagających w obróbce strumieni). Warto zauważyć, że umieszczenie tych klas we współdzielonym pakiecie *TwoWayCScKernel* pozwala (oprócz właściwego zdefiniowania po obu stronach) dodatkowo na udostępnienie własnego systemu plików zdalnych u klienta (może być to wykorzystane np. w celu wymiany plików między użytkownikami wykorzystując serwer jedynie jako pośredniczące repozytorium).

Bezpieczeństwo operacji na plikach oraz strumieniach zapewniają managery bezpieczeństwa danego użytkownika (domyślnie ograniczające operacje do ścieżki root użytkownika). Sprawdzają one prawa do odczytu oraz zapisu. Zastosowana implementacja ogranicza się do sprawdzenia, wymagany plik zawiera się w ścieżce root użytkownika, ale możliwe są bardziej wyszukane implementacje łącznie z funkcjonalnościami podobnymi do tych jakie zapewnia ACL (*Access Control List*) w wielu istniejących systemach.

Ze względu na wymagania jakie zostały postawione przez systemem, umożliwia on generowanie i propagację do innych użytkowników zdarzeń systemu plików takich jak:

- utworzenie pliku
- zmiana nazwy
- modyfikacje (daty, rozmiaru, atrybutu ukrycia)
- usunięcie

Klasy odpowiedzialne za obsługę systemu plików zdalnych umieściliśmy w pakiecie `twcsckernel.projectbase.io`:



rys. 7 Pakiet `twcsckernel.projectbase.io`

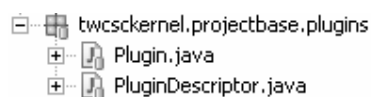
Opis funkcjonalności:

- `RemoteFileSystemImpl` – implementacja interfejsu odpowiedzialnego za zdalny dostęp do systemu plików na serwerze – posiada metodę do pobierania pliku zdalnego (`newRemoteFile`), do pobierania zdalnego strumienia wejścia (`newFileInputStream`) oraz strumienia wyjścia (`newFileOutputStream`).
- `WriterFactoryImpl`, `ReaderFacotryImpl`, `FileFactoryImpl` – implementacje wcześniej opisanych interfejsów operacji na systemach plików. Ważną funkcjonalnością wszystkich tych klas jest możliwość zwolnienia wszystkich zajmowanych zasobów, która jest wykorzystywana przez metodę wylogowania użytkownika z systemu (podstawowe założenie każdego systemu pośredniczącego w zarządzaniu zasobami). Fabryki te są przekazywane jako pola prywatne `RemoteFile`, `RemoteFileInputStream` oraz `RemoteFileOutputStream` i służą jako wewnętrzny mechanizm pracy zdalnej tych obiektów.
- `UserRootPathManager` – manager obsługi ścieżki root użytkownika, czyli ścieżki która jest widoczna u użytkownika jako najwyższy poziom drzewa katalogów. Zawiera metody służące do zarządzania i obrabiania ścieżek (zamiany ścieżek bezwzględnych użytkownika na ścieżki bezwzględne serwera i odwrotnie) – jest wykorzystywany przez manager bezpieczeństwa
- `FileSecurityManager` – opisany wyżej manager bezpieczeństwa
- `LocalFsChangeListener` – listener zmian w systemie plików działający lokalnie (stosowany przy współdzieleniu nasłuchu na danym fragmencie systemu plików),

- `RemoteFileInputStream`, `RemoteFileOutputStream` – zdalne strumienie wejścia/wyjścia. Są tworzone i pobierane za pośrednictwem opisanych fabryk i lokalnie u klienta można je traktować jak zwykłe strumienie wejścia/wyjścia (obsługują optymalizację transferu sieciowego poprzez serializację całych bloków danych, podczas gdy oryginalne implementacje Suna zapisują bloki danych w pętli). Klasa `RemoteFileInputStream` korzysta dodatkowo z klasy `ReadResult`, która zawiera pełne dane odczytanego bloku danych (tablicę oraz liczbę całkowitą). Klasy te stanowią zastosowanie wzorca projektowego *Adapter*, który dostosowuje zachowanie pewnych obiektów do istniejących interfejsów lub klas.
- `FileDescriptor` – deskryptor pliku zawierający wszystkie dane o pliku – stosowany przy listowaniu oraz zdarzeniach zmian w systemie plików
- `FsChangeDescriptor` – deskryptor zmiany w systemie plików – zawiera informacje potrzebne do odtworzenia u klienta spójnego stanu wynikającego ze zmiany

4.2.3 Pluginy

Pluginy zarówno po stronie serwera jak i po stronie klienta posiadają jeden wspólny interfejs. Jest to jednak interfejs pusty zwany w terminologii obiektowej *markerem*, który służy do klasyfikacji przeznaczenia i funkcji. Klasy związane z pluginami znajdują się w pakiecie `twcsckernel.projectbase.plugins`:



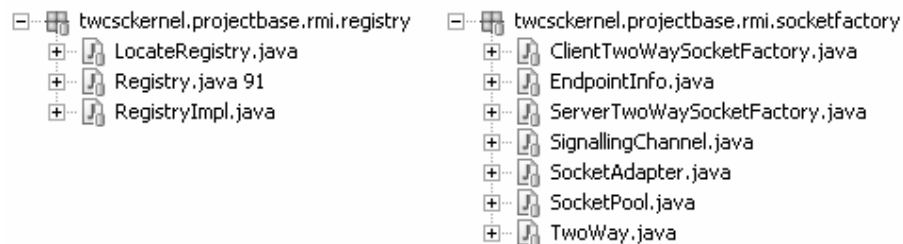
rys. 8 Pakiet `twcsckernel.projectbase.plugins`

Opis funkcjonalności:

- `Plugin` – opisany wyżej pusty interfejs plugina (*marker*)
- `PluginDescriptor` – deskryptor plugina zawierający podstawowe informacje o pluginie takie jak nazwa, opis, ścieżki pakietowe części serwerowej i klienckiej, ścieżkę pakietową tzw. *buildera* plugina (który służy do pobierania instancji na serwerze) oraz informację, czy plugin jest lokalny (posiada wyłącznie część kliencką) czy zdalny (posiada zarówno część kliencką jak i serwerową).

4.2.4 Obsługa RMI

Klasy obsługujące RMI znajdują się w dwóch pakietach. Pierwszy z nich stanowi implementację rejestru wraz interfejsem i klasą zarządzającą, natomiast drugi jest wykorzystaną przez grupę dwustronną fabryką socketów `TwoWaySocketFactory`:



rys. 9 Dwustronna fabryka socketów, pakiety

Opis funkcjonalności:

- `LocateRegistry` – jest to klasa zarządzająca rejestrem – na serwerze służy do tworzenia samego serwera i rejestru, natomiast na kliencie służy lokalizacji i pobraniu rejestru z serwera
- `Registry` – interfejs rejestru opisany w rozdziale 2.2.4

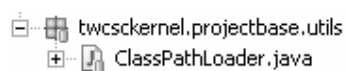
- `RegistryImpl` – implementacja rejestru oraz rozszerzenie serwera RMI opisane w rozdziale 2.2.4
- klasy pakietu `twcsckernel.projectbase.rmi.socketfactory` stanowią wykorzystaną bibliotekę `TwoWaySocketFactory`.

4.2.5 Dynamiczne ładowanie klas do maszyny wirtualnej

Ze względu na możliwość dynamicznego ściągania pluginów przez klienta, niezbędny okazał się mechanizm dynamicznego ładowania klas do maszyny wirtualnej Javy (JVM).

Z uwagi na możliwą rozszerzalność systemu zdecydowaliśmy się nie korzystać z dość sztywnych norm jakie narzuca manager bezpieczeństwa RMI (umożliwiający dynamiczne ładowanie klas do JVM przez określenie odpowiedniej własności systemowej). Wykorzystano specjalną klasę znaną na grupach dyskusyjnych, która służy do bezpośredniego ładowania klas z dysku do JVM. System opiera się zatem na założeniu, że administrator nie będzie instalował pluginów mogących czynić szkodę dla klienta (w przypadku zastosowania domyślnego managera bezpieczeństwa RMI założenie to byłoby identyczne, lecz ładowanie klas byłoby mniej elastyczne).

Dzięki tej funkcjonalności serwer ładuje biblioteki serwerowych części plugina do własnej JVM, a klient ładuje biblioteki klienckiej części plugina. Klasa odpowiedzialna za dynamiczne ładowanie klas do maszyny wirtualnej znajduje się w pakiecie `twcsckernel.projectbase.utils`:



rys. 10 `ClassPathLoader`

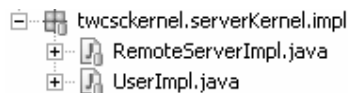
4.3 Architektura serwerowej części szkieletu - ServerKernel

4.3.1 Wprowadzenie

ServerKernel jest implementacją podstawowej funkcjonalności serwera, dla której uznaliśmy, że jest na tyle uniwersalna, żeby nie stanowić ograniczenia dla większości możliwych zastosowań. Funkcjonalności bezpośrednio związane z szkieletem, a mogące stanowić ograniczenie dla architektury konkretnej implementacji umieściliśmy w interfejsach.

4.3.2 Implementacje interfejsów zdalnych

Zdecydowano, że bez ograniczania zakresu rozbudowy można zaimplementować obsługę interfejsu użytkownika oraz serwera logowania (UserImpl oraz RemoteServerImpl). Klasy te umieszczono w pakiecie `twcsckernel.serverKernel.impl`:



rys. 11 Pakiet `twcsckernel.serverKernel.impl`

Opis funkcjonalności:

- RemoteServerImpl – odpowiada za logowanie użytkownika poprzez metodę:

```
User login(String name, String password);
```

Warto zauważyć, że namiastki obiektów zdalnych muszą występować zarówno na serwerze jak i kliencie, stąd namiastki tych dwóch klas znajdują się zarówno w bibliotece ServerKernel jak i ClientKernel. Klient otrzymuje namiastkę obiektu RemoteServerImpl. Za pomocą namiastki loguje się na serwerze VMD. Wysyła hasło i swój login.

Przesłane hasło jest haszowane algorytmem MD5, a następnie porównywane z hasłem przechowywanym w odpowiednim kontenerze zarejestrowanych użytkowników (`RegisteredUsers`). Podobny system zabezpieczeń (w postaci przechowywania na serwerze nie samego hasła lecz jego jednostronnej funkcji skrótu) jest z powodzeniem stosowany w większości dostępnych systemów klient-serwer. Algorytm MD5 uważany jest za algorytm bezpieczny. Jest stosowany między innymi w podpisie elektronicznym. Cechą funkcji haszującej jest jej jednostronność. Wielomianowo uzyskuje się skrót danego łańcucha, ale znalezienie łańcucha, dla którego mamy skrót, jest obliczeniowo trudne. Długość skrótu MD5 wynosi 128 bitów, zatem znalezienie hasła metodą *brute-force* wymaga średnio sprawdzenia 2^{128} możliwych haseł (może być mniej jeśli szybciej trafimy na hasło, lub więcej, gdy dojdzie do licznych kolizji między sprawdzanymi skrótami – odporność na kolizje zgodnie z paradoksem dnia urodzin wynosi ok. 2^{64}).

Jeśli użytkownik podał zatem odpowiednie hasło, metoda login zwraca klientowi namiastkę do utworzonego obiektu użytkownika, w przeciwnym wypadku zwraca wartość `null` (oznaczającą błąd logowania).

- `UserImpl` – jest klasą reprezentującą zalogowanego do systemu użytkownika. W przypadku pozytywnego zalogowania się na serwerze klient otrzymuje namiastkę obiektu `UserImpl` (która posiada metody zdalne definiowane przez interfejs `User`). Zauważmy, że `UserImpl` jest logiczną reprezentacją w systemie danego użytkownika. Dla każdego zalogowanego użytkownika jest tworzona instancja obiektu `UserImpl` i

oprócz swej funkcjonalności jest traktowany jako identyfikator użytkownika.

4.3.3 Użytkownicy, typy i specyfikacja wejścia/wyjścia

ServerKernel dostarcza podstawowe interfejsy administracji serwera, dzięki którym można dodawać i usuwać zarejestrowanych użytkowników oraz zarejestrowane typy użytkowników. Typ użytkownika określa zestaw pluginów dostępnych dla danego użytkownika. Podział taki został wprowadzony ze względu na elastyczność możliwej rozbudowy i podział funkcji jakie mogą pełnić poszczególni użytkownicy. Inne funkcje pełni np. administrator, inne zwykły użytkownik, a jeszcze inne goście. Wymaga to zdefiniowania zakresu odpowiedzialności, który jest w istocie określany przez funkcjonalność przyporządkowanego zbioru pluginów. Administrator systemu może posiadać plugin, którym zarządza listą dostępnych pluginów posiadając pełny zakres dostępu. Zwykli użytkownicy będą natomiast posiadali pluginy, które pełnią rolę dodatkowego funkcjonalnego oprogramowania na serwerze.

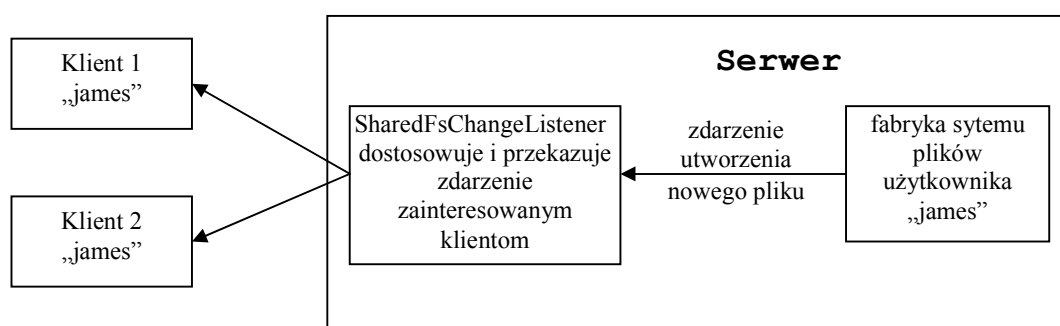
Klasy odpowiedzialne za operacje związane obsługą tego zarządzania (w tym wejścia/wyjścia) oraz współdzieleniem pewnych zasobów (listenerów zmian w systemie plików) umieściliśmy w pakiecie `twcsckernel.serverKernel.io`:



rys. 12 Pakiet `twcsckernel.serverKernel.io`

Opis funkcjonalności:

- `SharedFsChangeListener` – jest to interfejs współdzielonego listenera zmian w systemie plików. Przechowuje listenery zdalne zgłoszone przez klientów zdalnych. Jest on przekazywany do fabryk plików poszczególnych użytkowników (nie zaś same listenery zdalne) i pełni rolę pośrednika (proxy), przez co można realizować w różny sposób politykę wymiany zdarzeń pomiędzy różnymi użytkownikami. Jego działanie przedstawia poniższy schemat:



rys. 13 Schemat działania współdzielonego listenera

Schemat przedstawia zastosowaną przez nas politykę współdzielenia listenera między użytkownikami o tym samym loginie. Należy zaznaczyć, listenery nie gwarantują pełnej spójności obrazu systemu plików i służą jedynie usprawnieniu synchronizacji w pracy grupowej. Spójność informacji o systemie plików jest tracona w przypadku opóźnień w transferze zdarzeń przez sieć, lub w przypadku fizycznej modyfikacji pliku w systemie operacyjnym serwera (bez użycia fabryk). Ponadto dodatkowe ograniczenia spójności może narzucić sama polityka współdzielenia listenerów.

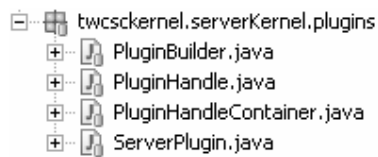
- `TypeIO`, `UserIO` – są to interfejsy implementacji dostępu do typów użytkowników i użytkowników. Posiadają metody `addNewUser` (`addNewType`) i `removeUser` (`removeType`), które służą do administrowania

zarejestrowanymi typami i użytkownikami oraz metodę `readUsers` (`readTypes`) odczytujące wszystkich użytkowników/wszystkie typy. Interfejs ten pozwala konkretnej implementacji na wykorzystanie zarówno systemu plików do administracji, jak i baz danych oraz innych nośników.

- `LogWriter` – jest to interfejs rejestratora logów, który pozwala na zapisywanie logów systemowych serwera w trzech różnych kategoriach:
 - logi błędów (błędy systemowe, logowania i inne)
 - logi użytkowników (informacje o udanym zalogowaniu i wylogowaniu)
 - logi informacyjne (pomocne w zapisywaniu ważnych informacji, które nie są ani błędami, ani informacjami o logowaniu)

4.3.4 Pluginy

W projekcie `ServerKernel` umieszczono podstawowe klasy do zarządzania pluginami serwera i klienta. Są to reprezentacje pluginów, ich kontenery, *buildery* (pobieranie instancji) oraz sama definicja pluginu serwera. Klasy te zawarte są w pakiecie `twcsckernel.serverKernel.plugins`:



rys. 14 Pakiet `twcsckernel.serverKernel.plugins`

Opisy funkcjonalności:

- `PluginBuilder` – jest to interfejs z jedną metodą `getInstance(userImpl, clientAgent, globalState)` służącą do pobierania instancji pluginu. Wprowadzono ten sposób otrzymywania instancji definiowany przez sam plugin, gdyż nie założono z góry, czy

plugin będzie unikalny dla każdego użytkownika, czy też ma być współdzielony globalnie (lub dla pewnej grupy użytkowników). Ze względu na brak możliwości definiowania metod statycznych w interfejsach, które mogłyby tą funkcjonalność zapewnić, wprowadzono właśnie buildery odpowiedzialne za to zadanie.

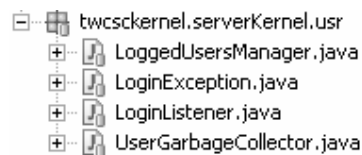
- `PluginHandle` – jest uchwyttem plugina, klasą utrzymującą informację na temat plugina serwerowego. Zawiera deskryptor (`PluginDescriptor`) pluginu a także ścieżki do serwerowej i klienckiej wersji pluginu. Dodatkowo, dla pluginów pracujących w trybie zdalnym, czyli dla takich, których pole `deskryptor.isRemote` przyjmuje wartość `true`, posiada wczytaną instancję buildera pluginu (`PluginBuilder`).
- `PluginHandleContainer` – kontener uchwytów pluginów pozwalający na pobieranie i poszczególnych pluginów użytkownika lub listowanie wszystkich pluginów dostępnych dla danego użytkownika:
 - `getUserTypePlugins(userType)` – zwraca tablicę uchwytów pluginów dostępnych dla danego typu użytkownika
 - `getUserPluginsDescriptors(userType)` – zwraca tablicę deskryptorów pluginów dostępnych dla danego typu użytkownika
 - `getUserPluginByClientPackagePath(userType, clientPckgPath)` – zwraca uchwyt konkretnego pluginu użytkownika (wyszukując po ścieżce pakietowej części klienckiej).
- `ServerPlugin` – interfejs ten stanowi definicję pluginu serwera. Posiada tylko dwie metody:
 - `getOwnAgent(userImpl)` – metoda zwraca agenta zdalnego dla podanego użytkownika (parametr użytkownika jest podawany, gdyż tak

jak pisaliśmy, nie zakładamy sposobu tworzenia instancji plugina – jeśli `PluginBuilder` zwracałby zawsze unikalne instancje parametr ten byłby niepotrzebny.

- `logoutUser(userImpl)` – metoda informująca plugin, że użytkownik został wylogowany i nie będzie więcej korzystać z plugina (parametr ma takie same znaczenie jak w metodzie `getOwnAgent(userImpl)`).

4.3.5 Zarządzanie użytkownikami

Podstawowe klasy służące zarządzaniu użytkownikami, zwalnianiu zasobów nieaktywnych użytkowników oraz udostępnianiu informacji o zalogowanych użytkownikach zostały zgromadzone w pakiecie `twcsckernel.serverKernel.usr`:



rys. 15 Pakiet `twcsckernel.serverKernel.usr`

Opis funkcjonalności:

- `LoggedUsersManager` – jest to interfejs definiujący metody zarządzania zalogowanymi użytkownikami (wewnętrzny login, logout ze zbioru użytkowników), udostępniający informacje o zalogowanych użytkownikach z możliwością pobrania aktualnie zalogowanych użytkowników danego typu oraz służy do zarządzania listenerami logowania i przydziałem współdzielonych listenerów zmian w systemie plików. To właśnie w implementacji tego interfejsu realizowana jest polityka współdzielenia opisanego wyżej listenera zmian. Listenerzy logowania służą do nasłuchiwanie zdarzeń informujących o logowaniu/wylogowaniu klienta.

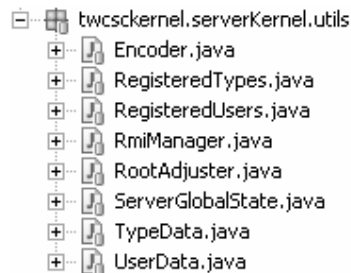
Dodatkową funkcją tej klasy jest tworzenie instancji menedżera bezpieczeństwa dla danego użytkownika konkretnej implementacji szkieletu.

- `LoginException` – wyjątek błędu logowania do zarządcy `LoggedUsersManager`. Może wystąpić wskutek próby wielokrotnego logowania tego samego użytkownika (identyfikowanego przez `UserImpl`) lub błędu tworzenia menedżera bezpieczeństwa (np. błędna ścieżka root użytkownika).
- `LoginListener` – interfejs listenera logowania służący do informowania o logowaniach / wylogowaniach użytkowników z menedżera `LoggedUsersManager`.
- `UserGarbageCollector` - jest odpowiedzialna za wylogowywanie nieaktywnych przez pewien zadany czas użytkowników. Pełni funkcje *garbage collector*a dla użytkowników. Metody obiektu `UserGarbageCollector` są odpalane w osobnym wątku i synchronizowane na zbiorze użytkowników zarejestrowanych jako *zdalne obiekty w serwerze RMI* (klasa `RmiManager`) nie zaś w zarządcy `LoggedUsersManager`, co pozwala na ukrycie jego funkcjonowania przed systemem. Wątek ten sprawdza wartość pola `timeStamp` użytkownika i porównuje z bieżącym czasem. Jeśli różnica między bieżącym czasem a ostatnią operacją wykonaną przez klienta na serwerze jest większa niż zadany przez administratora interwał czasowy, to użytkownik jest wylogowywany.

4.3.6 Narzędzia pomocnicze i zarządzające serwerem

ServerKernel posiada najważniejsze „narzędzia” związane z działaniem serwera.

Zawarte są one w pakiecie `twcsckernel.serverKernel.utils`:



rys. 16 Pakiet `twcsckernel.serverKernel.utils`

Opis funkcjonalności:

- `Encoder` – klasa odpowiedzialna za haszowanie haseł użytkownika algorytmem MD5.
- `RegisteredTypes` – klasa odpowiedzialna za zarządzanie typami. Pozwala dodawać i usuwać typy użytkowników. Jako parametr konstruktora musi otrzymać klasę typu `TypeIO`, która jest odpowiedzialna za fizyczny zapis/odczyt typów z konkretnego wejścia/wyjścia (w systemie VMD zaimplementowany jest `CsvTypeIO`, odpowiedzialny za odczyt/zapis typów do plików csv)
- `RegisteredUsers` – klasa odpowiedzialna za zarządzanie typami użytkowników. Pozwala dodawać i usuwać dane konkretnych użytkowników. Jako parametr konstruktora musi otrzymać klasę typu `UserIO`, która jest odpowiedzialna za fizyczny zapis/odczyt typów z konkretnego wejścia/wyjścia (w systemie VMD zaimplementowany jest `CsvUsersIO`, odpowiedzialny za odczyt/zapis użytkowników do plików csv).

- `TypedData` – dane konkretnego typu użytkownika
- `UserData` – dane konkretnego użytkownika
- `ServerGlobalState` – jest to reprezentacja globalnego stanu serwera, którą rozumiemy przez:
 - menadżera zalogowanych użytkowników (`LoggedUsersManager`)
 - listę zarejestrowanych typów użytkowników (`RegisteredTypes`)
 - listę zarejestrowanych użytkowników (`RegisteredUsers`)
 - kontener uchwytów pluginów (`PluginHandleContainer`)
- `RmiManager` - menadżer serwera RMI, służący do zarządzania serwerem RMI oraz rejestrowaniem obiektów zdalnych i klasą `UserGarbageCollector`. Odpowiedzialność za włączanie/wyłączanie serwera RMI sprowadza się do metod:
 - `startRegistryServer(remoteServer)` – uruchamia serwer RMI (będący jednocześnie rejestrem) na konkretnym porcie, dodatkowo rejestruje na nim konkretny serwer logowania `RemoteServer`.
 - `stopRegistryServer()` – wyłącza serwer RMI i wyrejestrowuje wszystkich zalogowanych użytkowników.

Ważną funkcjonalność `RmiManagera` stanowią także metody eksportujące i obiekty zdalne (rejestrująca w serwerze RMI) i wyrejestrowania z serwera RMI:

- `exportObject(object)` – jest to metoda rejestrująca obiekt zdalny w serwerze RMI. Tylko ona powinna być używana do rejestrowania obiektów zdalnych.

- `unexportObject(object)` – jest to metoda służąca do wyrejestrowania obiektu z serwera RMI. Tylko ona powinna być używana do wyrejestrowania obiektów zdalnych.

Metody te wyróżniają dwie klasy rejestrowanych obiektów, na które reagują oddzielnie:

- `UserImpl` - metoda zapisuje dodatkowo obiekt w zbiorze dostępnym dla *garbage collector*a użytkowników, przez co podlegają jego kontroli
- `RemoteAgent` - obiekty te można rejestrować i wyrejestrowywać wielokrotnie, gdyż pluginy mogą potencjalnie stosować tego samego agenta dla wielu użytkowników. Ze względu na ukrycie mechanizmów RMI przed samymi pluginami (które zapewniają tylko agentów zdalnych) wprowadzony został mechanizm wielokrotnego rejestrowania. Agenci nie są jednak fizycznie rejestrowani wielokrotnie, lecz prowadzone są liczniki liczby rejestracji dla każdego agenta (rejestracja ta działa analogicznie do mechanizmu twardych dowiązań do plików w systemie Linux/Unix). Pseudokod rejestracji `RemoteAgent` wygląda następująco:

```
if (agent.RegisterCount == 0) {  
    rmiRegister(agent);  
    agent.RegisterCount = 1;  
} else  
    agent.RegisterCount++;
```

Analogicznie wygląda sytuacja przy wyrejestrowaniu:

```
if (agent.RegisterCount > 1)  
    agent.RegisterCount--;  
else {  
    rmiUnregister(agent);  
    agent.RegisterCount = 0;  
}
```

- `RootAdjuster` – klasa posiadająca metody statyczne dostosowujące ścieżkę, deskryptor pliku lub deskryptor zmiany w systemie plików do odpowiedniej ścieżki root związanej z podanym managerem ścieżek root `UserRootPathManager` (np. dla managera ścieżki root = `c:\users\anna`, deskryptora pliku zawierającego ścieżkę path = `c:\users\anna\dokumenty` otrzymamy wynik result = `\dokumenty`)

`ServerKernel` jest jedynie biblioteką, na podstawie której powstaje konkretna implementacja serwera. Autorzy postawili opisać więc jedynie jej funkcjonalność w postaci pakietów, klas i niektórych metod. Działanie całego serwera, wraz ze scenariuszami użycia, zostanie przedstawione przy jego konkretnej implementacji, czyli w rozdziale opisującym projekt `VMDServer`.

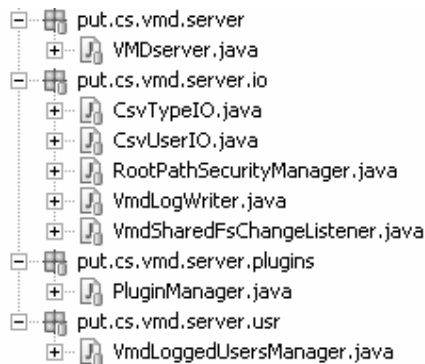
4.4 Architektura serwerowej implementacji frameworka - `VMDServer`

4.4.1 Wprowadzenie

`VMDServer` jest implementacją serwera opartą na bibliotece `ServerKernel` oraz `TwoWayCScKernel`. Dostarcza implementacji dla wszystkich potrzebnych systemowi interfejsów i zajmuje się uruchomieniem RMI, przekazaniem odpowiednich parametrów do klas i fizycznym wczytaniem zewnętrznych parametrów systemu takich jak pluginy, zarejestrowani użytkownicy i typy.

4.4.2 Struktura i implementacja

Struktura projektu jest następująca:

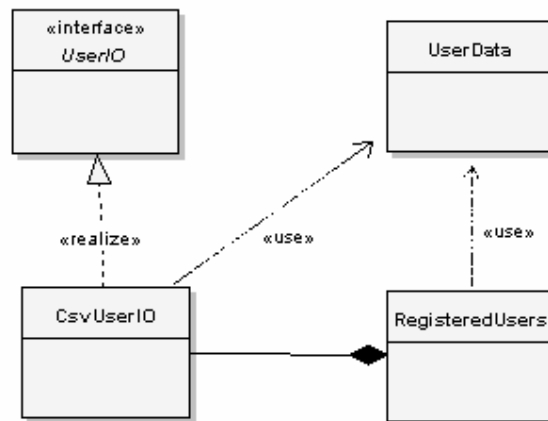


rys. 17 Struktura projektu VMDServer

Opis funkcjonalności:

- `CsvTypeIO`, `CsvUserIO` to implementacje interfejsów `TypeIO` i `CsvIO` z `ServerKernel`. Są one odpowiedzialne za odczyt/zapis typów użytkowników i samych użytkowników. Przyjrzyjmy się bliżej mechanizmowi odpowiedzialnemu za odczyt/zapis użytkowników.

Oto diagram klas:



rys. 18 Mechanizm odczytu/zapisu użytkowników

Interfejs `UserIO` jest interfejsem dostępu do użytkowników. Posiada metody dodania/usunięcia użytkownika oraz wczytania wszystkich

użytkowników. W systemie VMD jego realizacją jest `CsvUserIO`, czyli dodawanie/usuwanie i wczytywanie użytkowników z pliku `*.csv`. `RegisteredUsers` jest klasą zawierającą mapę użytkowników. W celu stworzenia mapy użytkowników oraz modyfikowania jej `RegisteredUsers` korzysta z konkretnej implementacji `UserIO`, czyli w tym wypadku z `CsvUserIO`. Dane pojedynczego użytkownika są przechowywane w klasie `UserData`. Dostęp do prywatnych pól klasy jest kontrolowany przez settery i gettery. Taki podział zapewnia oddzielenie fizycznej implementacji dostępu do konkretnego źródła danych (którym może być baza danych, plik csv, zserializowane dane itp.) od klasy `RegisteredUsers`, odpowiedzialnej za utrzymywanie danych. Ustawiając w konstruktorze klasy `RegisteredUsers` konkretną implementację `UserIO` zmieniamy sposób dostępu do danych. Oddzielenie interfejsu wejścia/wyjścia od konkretnej implementacji dla określonego systemu pozwala na prostą rozszerzalność systemu o różne nośniki danych i różne formaty. Muszą one implementować interfejs `UserIO`.

- Klasa `VmdLogWriter` pełni funkcje log managera w systemie. Domyślnie cały log systemu jest wysyłany na standardowe wyjście. Informuje on o zmianach w systemie takich jak:

- zalogowanie nowego użytkownika
- logout
- wyrzucenie nieaktywnego użytkownika (kickout)

`VmdLogWriter` informuje także o błędach, które zaszły w systemie, takich jak:

- *ERROR LOGIN PASSWORD* – niewłaściwe hasło podczas logowania

- *ERROR PLUGIN LOADING* – błąd załadowania plugina
- *ERROR UNEXPECTED EXCEPTION* – błąd nieprzewidzianego wyjątku

Dzięki obecności logu istnieje możliwość śledzenia historii działania serwera. Może to być znaczące udogodnienie dla administratora, pozwala na przykład wyśledzić próby nieudanego zalogowania się na konto użytkownika i podjąć odpowiednią politykę bezpieczeństwa jak wybór nietrywialnego hasła, lub częste zmiany haseł.

- *RootPathSecurityManager* – pełni sprawdza prawa dostępu użytkownika do danego pliku lub katalogu. Dla systemu VMD zdefiniowano 2 podstawowe prawa
 - prawo do odczytu – użytkownik posiada prawo do odczytu, jeśli jego ścieżka roota zawiera się w sprawdzanej ścieżce oraz plik istnieje, należy analizować ścieżki kanoniczne, aby zawieranie się sprawdzane na łańcuchach znakowych było poprawne; jako ścieżkę kanoniczną rozumie się w tym kontekście ścieżkę o najmniejszej długości reprezentującą pewien system plików
 - prawo do zapisu – użytkownik posiada prawo zapisu, jeśli jego ścieżka roota zawiera się w sprawdzanej ścieżce (można dodatkowo zabronić prawa zapisu do katalogu zawierającego ustawienia systemowe użytkownika)
- *VmdSharedFsChangeListener* – jest to klasa współdzielonego listenera. Jest to bardzo rozbudowany mechanizm propagowania zmian w całym systemie rozproszonym. Dla każdego zarejestrowanego użytkownika tworzony jest osobny współdzielony listener. W tym kontekście

użytkownicy rozróżniali są po nazwie. Jeśli z dwóch stanowisk nastąpi logowanie na to same konto, obaj użytkownicy będą mieli przydzielony ten sam listener. Stanowiska te współdzielą więc informacje o zmianach w systemie plików, o których informują listenery. Listener ten jest podpinany pod fabryki I/O (`RemoteFileFactory`, `WriterFactory`) każdego użytkownika. W przypadku zmian wprowadzonych przez użytkownika w systemie plików, zmiany te są wychwytywane przez listener i pozostałe instancje klienta są o nich informowane. Schemat propagowania zmian jest przedstawiony w opisie `SharedFsChangeListener` w rozdziale opisującym `ServerKernel`.

Współdzielony listener lokalny zawiera listenery zdalne klientów zalogowanych na tej samej nazwie (w schemacie jest to `anna`). Za pomocą listenerów zdalnych informuje klientów o zmianie w systemie plików. Takie rozwiązanie minimalizuje ruch w sieci. Zauważmy, że liczba sygnałów w sieci jest minimalna, aby poinformować zainteresowanych klientów o zmianach. Informacje otrzymują jedynie instancje klienta, który wprowadził zmiany (przy założeniu pustego przekroju między ścieżkami `root` poszczególnych użytkowników). Oto scenariusz prowadzący do poinformowania innych klientów o zmianie w systemie plików:

- klient 1 loguje się do serwera jako „anna”, tworzona jest instancja lokalnego listenera zmiany w systemie plików na serwerze
- klient 2 loguje się do serwera jako „anna”, podpinany jest do istniejącego listenera
- klient 1 tworzy plik zdalnie na serwerze przy użyciu fabryki `RemoteFileFactory`, fabryka informuje o tym współdzielonego listenera

- listener propaguje zmiany do wszystkich instancji klienta „anna”
- każdy klient lokalny wprowadza lokalnie zmiany w obrazie systemu plików

Listener współdzielony utrzymuje stuby do listenerów zdalnych u klienta, aby informować ich o wprowadzonych zmianach w systemie plików. Taki system jest bardzo oszczędny, ponieważ nie jest konieczne fizyczne sprawdzanie zmian w systemie plików, co wiązałoby się to z przymusem istnienia dodatkowych wątków, które śledziłyby takie zmiany i propagowały je do instancji klienta. Zakłada się istnienie pewnego kontraktu, który pozwala by zmiany w systemie plików były tylko i wyłącznie wynikiem działania fabryk i/o (jednak tak jak było to opisywane wcześniej system tych listenerów nie gwarantuje posiadania spójnego obrazu systemu plików i wspomaga jedynie bieżącą jego rekonstrukcję). Niesie to pewne ograniczenia na system, zauważmy że systemy plików dostępnych dla danych klientów powinny być rozłączne. Oto przykład, który to wyjaśnia:

- Użytkownik A ma dostęp do ścieżki a,b,c
- Użytkownik B ma dostęp do ścieżki c,d,e

W systemie znajdują się 3 instancje klientów, dwie zalogowane na konto użytkownika A i jedna zalogowana na konto użytkownika B. Zmiana systemu plików w podścieżce ścieżki c, dla użytkownika A przepropaguje się na jego drugą instancję, jednak nie przepropaguje się na instancję użytkownika, który także powinien być poinformowany o zmianach w podścieżce ścieżki c. System VMD zakłada, że przekrój ścieżek dla użytkowników jest rozłączny. System jest jednak na tyle ogólny, że zmiana

jednego parametru (parametr `int` w konstruktorze `VmdLoggedUsersManager`) powoduje wprowadzenie globalnego współdzielonego listenera dla całego systemu, przez co propagowanie zmian jest analizowane dla wszystkich użytkowników. Zwiększa to jednak znacznie złożoność obliczeniową systemu, ponieważ wymusza sprawdzanie dla każdego zdarzenia który użytkownik byłby zainteresowany otrzymaniem informacji o zmianach w systemie plików. W razie potrzeby problem ten można częściowo rozwiązać.

Twórcy VMD proponują następujący algorytm:

- tworzy się mapę ścieżek root użytkowników
- dla każdego użytkownika $X[A]$ (użytkownik X ze ścieżką A) tworzy się listę użytkowników $Y_i [B_i]$ dla których zachodzi relacja:

$$A \subseteq B_i \vee B_i \subset A$$

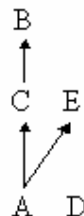
Relacja zawierania jest tutaj rozumiana jako zawieranie podrzewa katalogowego (np. `/usr/bin` zawiera się w `/usr`). Dodatkowo dla użytkowników dla których zachodzi relacja $A \subseteq B_i$ zaznaczamy, że przekazanie zdarzenia jest obligatoryjne.

W przypadku relacji $B_i \subset A$ przekazanie zdarzenia podlega rozpatrzeniu, czy zostało ono wygenerowane w podścieżce ścieżki B_i . W celu optymalizacji liczby takich decyzji, można uwzględnić przypadki gdy występują relacje $B_i \subseteq B_j$ poprzez utworzenie ciągu użytkowników $(Y_i, Y_{i+1}, Y_{i+2}, \dots)$ dla których występuje relacja $B_i \supseteq B_{i+1} \supseteq B_{i+2} \dots$. Jeśli dla użytkownika Y_k z tego ciągu nastąpi decyzja o przekazaniu zdarzenia, to automatycznie jest przekazywane do wszystkich użytkowników, Y_l dla których $l > k$.

Przykładowo:

Niech w systemie będzie 5 ścieżek root:

User 1 – ścieżka „A”, User 2 – ścieżka „B”, User 3 – ścieżka „C”, User 4 – ścieżka „D”, User 5 – ścieżka „E” dla których zachodzą relacje: $C \subset A$, $B \subset C$, $E \subset A$, $E \not\subset C$, $D \not\subset A$, $A \not\subset D$:



Mamy więc listy (apostrof oznacza obligatoryjność wysłania):

$$\begin{aligned} L(U1) &= \{U1', (U2, U3), U5\} & L(U4) &= \{U4'\} \\ L(U2) &= \{U1', U2', U3'\} & L(U5) &= \{U1', U5'\} \\ L(U3) &= \{U1', U2, U3'\} \end{aligned}$$

Jeśli listener użytkownika U1 otrzymał zdarzenie o zmianie w podścieżce ścieżki B, to wysłał on informację do użytkowników należących do zbioru $L(U1)$ w następujący sposób:

- wysłał automatycznie do użytkownika U1,
- stwierdza, że należy zdarzenie przekazać do użytkownika U2 co implikuje automatyczne przekazanie do U3,
- stwierdza, że zdarzenie nie dotyczy użytkownika U5.

Takie rozwiązanie znacznie ogranicza liczbę użytkowników dla których następuje rozpatrzenie możliwości wysłania (w tym przypadku występują dwie decyzje dla pięciu użytkowników). Rozwiązanie to zwiększa jednak złożoność implementacji mechanizmu listenera.

- `PluginManager` – jest klasą zarządzającą pluginami. Utrzymuje kontener pluginów, klasę typu `PluginHandleContainer`. `PluginManager` odpowiedzialny jest za ładowanie klas pluginów. Ładuje on do JVM część serwerową pluginu „server.jar” za pomocą opisanej klasy `ClassPathLoader`. Część kliencka jest ładowana do pamięci jako tablica bajtów. W takiej formie będzie ona zwracana klientowi. Dodatkowo plugin umieszczany jest w kontenerze pluginów (klasa `PluginHandleContainer`). Przydzielany jest typom użytkowników, zgodnie z ich uprawnieniami. Złożoność algorytmu, który tego dokonuje wynosi :

$$\sum_{type \in UserTypes} L(type) \cdot |Plugins| \text{ gdzie:}$$

- $L(type)$ to liczba pluginów przydzielona dla typu użytkownika $type$,
- $|Plugins|$ to liczba zainstalowanych pluginów

Przydział ładowanych pluginów do typów wygląda więc następująco:

```
for (plugin in plugins)
  for (typ in types)
    for (lista dozwolonych pluginów w typie)
      jeśli plugin jest na liście, załaduj go do listy
      pluginów dozwolonych dla typu typ
```

- `VmdLoggedUsersManager` – jest menadżerem zalogowanych użytkowników do systemu. Zależnie od polityki serwera utrzymuje on albo zbiór listenerów współdzielonych między tymi samymi użytkownikami (loginami), lub jeden listener globalny (współdzielony dla wszystkich użytkowników). Metoda `getSharedUserListener(userImpl)` pozwala na pobranie listenera współdzielonego dla danego użytkownika. Klasa ta posiada ponadto implementację mechanizmu listenerów logowania. Nie jest

ona jednak bezpośrednio używana w serwerze VMD pozostawiając wykorzystanie tej funkcjonalności ewentualnym pluginom.

Oto opis działania podstawowych etapów pracy systemu:

- `VMDServer`, główna klasa serwera uruchamia serwer i ustala ew. parametry systemu wprowadzone z linii poleceń
- `VMDServer` tworzy instancję klasy `RmiManager`, obiekty utrzymujące listę typów i listę użytkowników (`RegisteredUsers` i `RegisteredTypes` przy użyciu `CsvUserIO` oraz `CsvTypeIO`), `LogWritera` oraz globalne ustawienia serwera `GlobalState`.
- startowany jest serwer RMI za pomocą klasy `RmiManager` (uruchomiona metoda `startRegistryServer` która rejestruje serwer logowania `RemoteServer` - obiekt dostępny zdalnie)
- klient pobiera z rejestru `RemoteServer` zawierający metodę login z serwera logowania. Przesyła hasło i login
- po pozytywnej weryfikacji otrzymuje namiastkę interfejsu `User`
- za pomocą obiektu namiastki `User` ściąga listę dostępnych pluginów za pomocą metody `listClientPlugins()`. W tym celu:
 - Ze znanego sobie stanu globalnego wyciąga kontener pluginów
 - uruchamia na nim metodę pobrania pluginów po typie użytkownika
- użytkownik ściąga potrzebny plugin za pomocą metody `downloadClientPlugin`, otrzymując on wersję binarną pluginu, w postaci tablicy bajtów

4.4.3 Parametry uruchomieniowe serwera

- gui - wskazuje na uruchomienie graficznego edytora parametrów serwera
- upath - ustawia ścieżkę do pliku zarejestrowanych użytkowników, np.
upath=users.csv
- elog - ustawia plik logu błędów, np. elog=error.log
- ilog - ustawia plik logu info, np. ilog=info.log
- tpath - ustawia ścieżkę do pliku zarejestrowanych typów, np.
tpath=types.csv
- shl - ustawia rodzaj współdzielonego listenera (global/user), np. shl=user
- gc - ustawia interwał user garbage collector(ms), np. gc=6000
- plugins - ustawia katalog pluginów, np. plugins=/etc/Plugins
- port - ustawia port serwera, np. port=1410
- ct - ustawia timeout dla próby połączenia pasywnego(ms), np. ct=2000
- ulog - ustawia plik logu użytkowników, np. ulog=user.log

4.5 Architektura klienckiej części szkieletu – ClientKernel

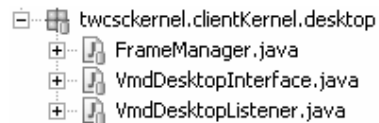
4.5.1 Wprowadzenie

ClientKernel podobnie jak ServerKernel jest częścią składową napisanego szkieletu i pełni funkcję podstawowej biblioteki funkcjonalnej klienta. Podobnie jak w ServerKernel funkcjonalności, które mogłyby ograniczać możliwość rozbudowy i strukturę końcowego klienta zostały pozostawione do późniejszej implementacji poprzez zdefiniowanie odpowiednich interfejsów. Przyjęto ponadto założenie, że w ramach funkcjonalności klienta zawsze wystąpi interfejs graficzny. ClientKernel jest więc podstawową biblioteką służącą budowie klienta z graficznym interfejsem. Jeśli klient miałby mieć postać konsoli, to można pominąć

ClientKernel i zbudować aplikację wyłącznie w oparciu o gotowe rozwiązania z *TwoWayCScKernel* oraz *ServerKernel*. Przykładem do rozbudowy takiego klienta może być *obiekt mock* opisany w rozdziale „Testowanie systemu”.

4.5.2 Interfejsy programistyczne interfejsu graficznego

Zgodnie z założeniem o możliwości rozbudowy, ClientKernel nie narzuca żadnego konkretnego interfejsu graficznego a jedynie definiuje odpowiednie interfejsy do zarządzania środowiskiem graficznym. Umieszczono je w pakiecie `twcsckernel.clientKernel.desktop`:



rys. 19 Pakiet `twcsckernel.clientKernel.desktop`

Opis interfejsów:

- `FrameManager` – jest to interfejs który odpowiada za zarządzanie oknami – posiada metody służące ustawianiu pozycji, maksymalizacji, minimalizacji, trybu pełnego ekranu etc.
- `VmdDesktopInterface` – jest to interfejs desktopu jako wyróżnionego plugina (z punktu widzenia serwera implementacja `VmdDesktopInterface` będzie identyczna jak pozostałe pluginy, lecz ze względu na szczególną funkcjonalność u klienta wymagał od osobnej definicji)
- `VmdDesktopListener` – interfejs służy do wspomagania zarządzaniem pluginów bezpośrednio z poziomu pulpitu.

4.5.3 Mechanizm zdalnego listenera

Klient chcąc nasłuchiwać zmiany w zdalnym systemie plików implementuje interfejs `FsChangeListener` z jądra frameworka *TwoWayCScKernel*. Implementacja ta, aby zachować ogólność rozumowania i funkcjonalności stanowi jedynie kontener listenerów lokalnych rejestrowanych przez różne aplikacje. Zdarzenie informujące o zmianach w systemie plików na serwerze jest przesyłane jednorazowo do klienta i jest propagowane przez listener zdalny na listenery lokalne aplikacji. Pozwala to ograniczyć ruch sieciowy w przypadku gdyby wiele aplikacji żądało dostępu do informacji o zmianach. Implementację listenera zdalnego umieściliśmy w pakiecie `twcsckernel.clientKernel.io` oraz `twcsckernel.clientKernel.io.fsChangeListenerClasses`:



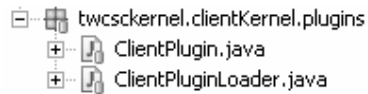
rys. 20 Implementacja listenera zdalnego

Klasa `FsChangeListenerServer` jest odpowiedzialna za rejestrowanie listenerów lokalnych aplikacji i przekazywanie im zdarzeń o zmianach w systemie plików otrzymane od serwera.

Elementami listy listenerów w klasie `FsChangeListenerServer` są instancje klasy `RegisteredListenersListElem`, które przekazują informacje do samych aplikacji.

4.5.4 Pluginy

`ClientKernel` dostarcza również definicję interfejsu plugina klienta oraz klasę narzędziową służącą pobieraniu instancji plugina. Znajdują się one w pakiecie `twcsckernel.clientKernel.plugins`:



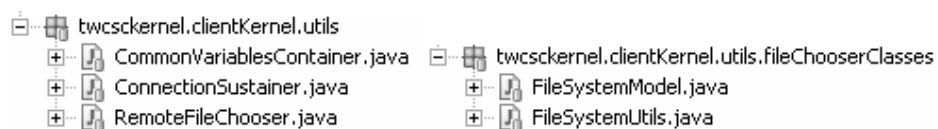
rys. 21 Pakiet `twcsckernel.clientKernel.plugins`

Opis funkcjonalności:

- `ClientPlugin` – interfejs definiujący podstawowe metody obsługiwane przez implementację plugina klienckiego (inicjacja plugina, pobranie agenta sieciowego, ustawienie zdalnego agenta części serwerowej jeśli plugin jest zdalny oraz pobranie panelu plugina).
- `PluginLoader` – jest to klasa służąca do tworzenia instancji plugina za pomocą mechanizmu refleksji – zakłada się, że plugin posiada konstruktor bezparametrowy.

4.5.5 Klasy narzędziowe

Klasy pomocnicze wraz z definicją podstawowego modelu tabeli dla systemu plików zostały zawarte w pakietach `twcsckernel.clientKernel.utils` oraz `twcsckernel.clientKernel.utils.fileChooserClasses`:



rys. 22 Pakiet `twcsckernel.clientKernel.utils` i `fileChooserClasses`

Opisy funkcjonalności:

- `CommonVariablesContainer` – klasa ta stanowi kontener zdalnej fabryki systemu plików i posiada metodę rejestracji agenta plugina w lokalnym serwerze RMI klienta (żeby obiekt mógł być udostępniony zdalnie) oraz metodę rejestrującą plugin w kontenerze lokalnym aktywnych pluginów

- `ConnectionSustainer` – klasa ta kontroluje “pingowanie” serwera w ramach dozwolonego czasu nieaktywności po to, aby klient miał gwarancję, iż nie zostanie wylogowany przez serwer (`UserGarbageCollector`) z powodu przekroczenia maksymalnego czasu nieaktywności
- `RemoteFileChooser` – jest to narzędzie do graficznego wyboru plików zdalnych (przeglądu plików zdalnych)
- `FileSystemModel` – model danych tabeli reprezentującej zdalny system plików
- `FileSystemUtils` – klasa zawiera metody narzędziowe do obróbki nazw plików (wyodrębnienia rozszerzenia, nazwy pliku i uzyskania serwerowej ścieżki kanonicznej poprzez obróbkę łańcucha)

5. Implementacje pluginów

5.1 VmdDesktop

5.1.1 Wprowadzenie

`VmdDesktop` jest pluginem będącym konkretną implementacją graficznego interfejsu użytkownika zdalnego wirtualnego desktopu. Implementuje on interfejsy zawarte w pakiecie `twcsckernel.clientKernel.desktop` (opisane w punkcie 4.5.2)

Plugin ten pozwala na wizualizację wirtualnego pulpitu. Jako pulpit rozumiany jest obszar roboczy na którym umieszczane są okna innych aplikacji. `VmdDesktop` pozwala na personalizację tła pulpitu oraz definiowanie aktywatorów innych pluginów w formie ikon umieszczonych na pulpicie.

VmdDesktop pozwala na współdzielenie pulpitu wielu użytkownikom korzystającym z tego samego konta w systemie.

Jedną z ważniejszych decyzji projektowych było ustalenie zakresu synchronizacji poszczególnych instancji współdzielonego pulpitu. Ze względu na efektywność pracy użytkowników postanowiono, że jedynie tło oraz stan ikon na pulpicie będą propagowane w czasie pracy.

VmdDesktop pracuje więc jako plugin sieciowy. Część serwerowa odpowiada za przechowywanie ustawień pulpitu oraz propagację zmian do wszystkich klientów korzystających z tego samego konta podczas logowania do serwera.

5.1.2 Część serwerowa

Odpowiada ona za wczytanie ustawień z pliku konfiguracyjnego oraz udostępnienie agenta sieciowego który zostanie przekazany do klienckiej części pluginu.

Dla każdego z kont użytkowników jest powoływana instancja klasy VmdDesktopPlugin podczas próby pobrania instancji przez użytkownika korzystającego z danego konta. Kolejni użytkownicy korzystający z tego samego konta otrzymują tę samą instancję pluginu.

Ustawienia są przechowywane w katalogu domowym użytkownika w postaci zserializowanego pliku. Plik ten zawiera obiekt klasy DesktopState, przechowujący informacje o rozmiarze wirtualnego pulpitu, tle oraz ikonach znajdujących się na pulpicie.

Te informacje są następnie udostępnione klientom pluginu, którzy w oparciu o nie tworzą graficzny interfejs wirtualnego pulpitu.

Każda zmiana ustawień tła oraz ikon jest propagowana do wszystkich klientów danej instancji serwera Desktopu. Jedynie ustawienia wielkości wirtualnego desktopu obowiązują po ponownym uruchomieniu plugina dla danego konta użytkownika.

Po zakończeniu pracy z desktopem przez ostatniego użytkownika ustawienia są zapisywane do pliku konfiguracyjnego.

5.1.3 Część kliencka

VmdDesktop jest pierwszym pluginem jaki zostaje uruchomiony po zalogowaniu się użytkownika do serwera VMD. Tworzy główne okno aplikacji wraz z wirtualnym desktopem w oparciu o ustawienia pobrane z serwera. Pozwala na uruchomienie innych pluginów oraz personalizację desktopu.

5.1.4 Opis interfejsu

Interfejs składa się z:

- menu pozwalającego na zamknięcie desktopu oraz uruchomienie dowolnego z dostępnych dla użytkownika pluginów,
- pasków przewijania, które pojawiają się jeśli wirtualny desktop jest większy od rozmiaru okna, pozwalają na zmianę widocznego obszaru pulpitu
- ikon uruchamiających określony plugin, których lokalizację można zmienić metodą “przeciągnij i upuść”

Za pomocą kontekstowego menu nad obszarem pulpitu użytkownik może:

- dodać nową ikonę powiązaną z aplikacją wybraną z listy dostępnych pluginów

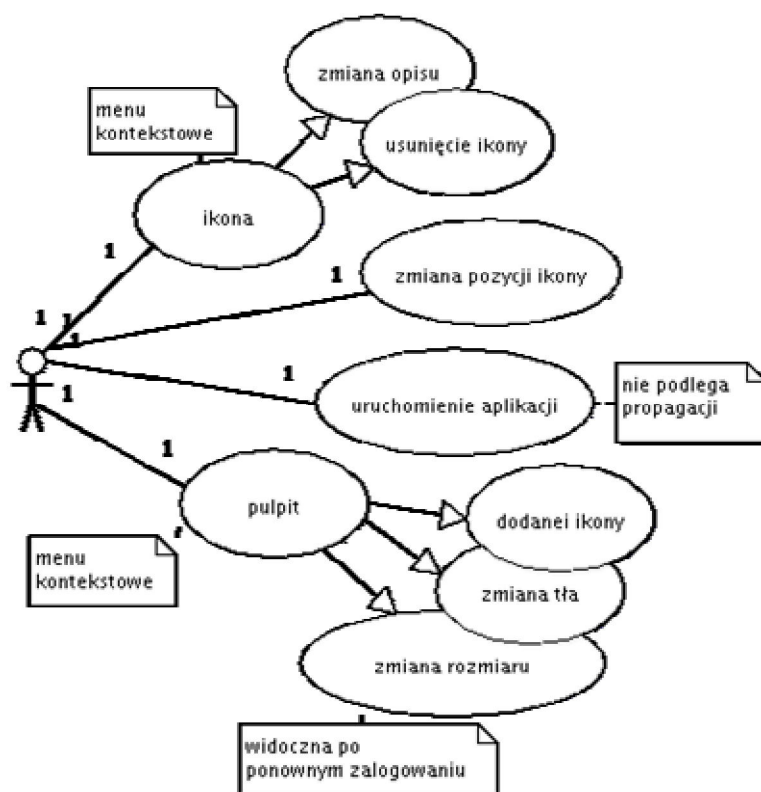
- zmienić tło pulpitu podając lokalny plik graficzny
- zmienić rozmiar wirtualnego pulpitu (zmiana nastąpi po wylogowaniu z danego konta wszystkich użytkowników)

Menu kontekstowe ikony pozwala na

- usunięcie jej z pulpitu,
- zmianę opisu znajdującego się pod ikoną

Wszystkie elementy znajdujące się na wirtualnym pulpicie mogą znaleźć się w dowolnym jego miejscu. Ikony jak i okna aplikacji mogą być położone poza obszarem widzianym przez użytkownika. Każde zminimalizowane okno aplikacji jest reprezentowane przez przycisk znajdujący się w dolnym lewym rogu pulpitu.

Poniższy rysunek (rys 23) przedstawia przypadki użycia wirtualnego pulpitu zaimplementowanego w VmdDestop.



Rys 23. Przypadki użycia zaimplementowane w VmdDesktop

5.2 VmdTalk

5.2.1 Wstęp

VmdTalk jest prostym komunikatorem tekstowym dla środowiska VMD. Pozwala na prowadzenie rozmów tekstowych z dowolną liczbą użytkowników czyli tzw. konferencje. Umożliwia on komunikację między wszystkimi użytkownikami serwera, niezależnie od konta na którym aktualnie pracują

Uruchomienie VmdTalk powoduje automatyczne połączenie aplikacji z serwerem VmdTalk będącym pluginem serwera VMD. Serwer VmdTalk jest uruchamiany w momencie próby podłączenia pierwszego użytkownika.

5.2.2 Opis interfejsu użytkownika

Interfejs składa się z listy zalogowanych użytkowników oraz zakładek rozmów prowadzonych przez użytkownika.

Użytkownik rozpoczyna rozmowę poprzez wybór jednej lub więcej nazw użytkowników z listy i naciśnięcie przycisku “czat”. W nowo otwartej zakładce rozmowy może w polu tekstowym wpisać treść komunikatu zatwierdzić wysłanie za pomocą klawisza Enter. Zakładkę z rozmową można zamknąć wybierając z menu kontekstowego opcje “zakończ”. Nowa zakładka otwierana jest także gdy użytkownik otrzyma komunikat od innego, nowego członka czatu.

5.3 VmdEditor

5.3.1 Wstęp

VMDEditor jest prostym edytorem tekstu dla środowiska VMD. Umożliwia on odczyt zarówno plików lokalnych jak i zdalnych, posiada także funkcję monitorowania zmian plików umieszczonych na serwerze.

5.3.2 Opis interfejsu

Obszar roboczy edytora stanowią zakładki związane z każdym otwartym plikiem. Wszystkie operacje wykonywane przy pomocy przycisków paska narzędziowego bądź menu głównego związane są z aktualnie aktywną zakładką bądź prowadzą do otwarcia kolejnej. Możliwe operacje na zakładce:

- ✓ Utworzenie nowej, pustej zakładki przez wybranie z menu pozycji „Nowy” lub wybranie z paska narzędzi przycisku z ikoną nowego pliku.
- ✓ Utworzenie nowej zakładki i wyświetlenie pliku dostępnego lokalnie przez wybranie z menu pozycji „Otwórz plik lokalny” lub zdalnie przez wybranie z menu pozycji „Otwórz” lub wybranie z paska narzędziowego przycisku z ikoną otwierania pliku.
- ✓ Zapisanie zmian dokonanych w pliku przez wybranie z menu pozycji „Zapisz” lub wybranie z paska narzędzi przycisku z ikoną zapisywania pliku.
- ✓ Zamknięcie zakładki wraz z kontrolą zapisu zmian przez wybranie z menu pozycji „Zamknij” lub wybranie z paska narzędzi przycisku z ikoną.
- ✓ Zapisanie pliku w nowej lokalizacji lokalnej lub zdalnej przez wybranie z menu pozycji „Zapisz lokalnie jako..” lub „Zapisz jako..”

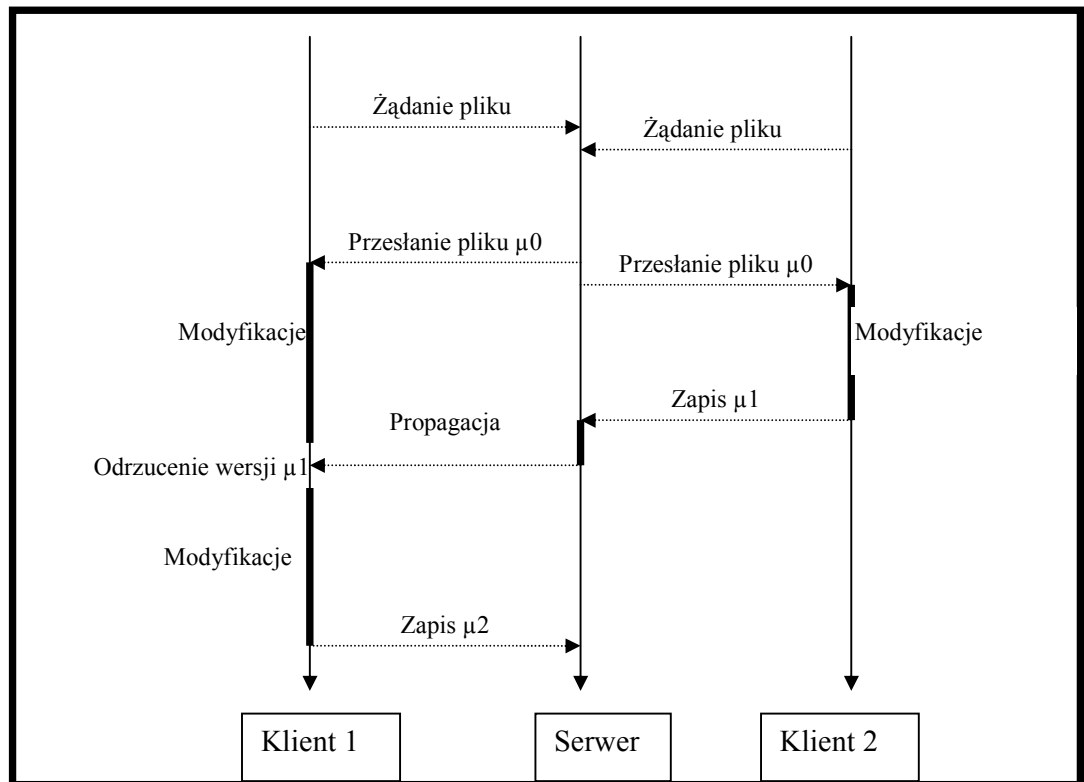
Interfejs edytora informuje także użytkownika czy aktualnie wyświetlana zawartość pliku jest zgodna z jego fizyczną zawartością. Pojęcie „fizyczna zawartość pliku” oznacza to, co aktualnie znajduje się na nośniku danych. Wyświetlenie gwiazdki przy nazwie pliku oznacza, że bieżący użytkownik dokonał

zmian w pliku, bądź też inny użytkownik zapisał swoje zmiany w pliku zdalnym, zaś bieżący użytkownik nie wczytał ich z serwera.

5.3.3 Używanie edytora z wykorzystaniem kontroli zmian plików zdalnych

Edytor implementuje interfejs `LocalFsChangeListener` co umożliwia monitorowanie zmian w otwartych zdalnych plikach. Jeśli taka zmiana zostanie dokonana przez innego użytkownika systemu VMD o takiej samej nazwie, zostanie ona przepropagowana i informacja o niej zostanie przekazana użytkownikowi edytora, który będzie mógł zdecydować, czy chce pozostać przy swojej wersji pliku czy też chce wczytać nową wersję pliku, tracąc jednak wersję poprzednią. Jeśli zdecyduje się na nie wczytywanie nowej wersji pliku będzie on mógł podejrzeć zmiany otwierając nową zakładkę z nową wersją pliku.

Na poniższym rysunku (rys. 23) pokazano przykład wykorzystania edytora tekstu przez dwóch użytkowników działających na tym samym pliku. Początkowo serwer przechowuje plik w wersji oznaczonej $\mu 0$. Po modyfikacjach przez klienta nr 2 do serwera wysłana zostaje wersja oznaczonej $\mu 1$. Informacja o nowej wersji zostaje przekazana użytkownikowi nr 1, który decyduje się (po obejrzeniu lub nie) odrzucić nowszą wersję i pozostać przy swojej. Po dalszych modyfikacjach zapisuje on swoje zmiany. Na serwerze znajduje się już tylko wersja oznaczona $\mu 2$. Użytkownik nr 1 miał jednak szansę skonfrontować zmiany w pliku dokonane przez innego użytkownika



rys. 23 Diagram przepływu danych dla edytora

5.4 VMDDraw

5.4.1 Wprowadzenie

VmdDraw jest pluginem zapewniającym podstawowe funkcje programu do rysowania. Posiada takie funkcje jak: ołówek, gumka, pisanie tekstu, spray, kreślenie figur geometrycznych takich jak prostokąt i elipsa. Posiada też narzędzie, które pozwala na kreślenie promieni z jednego punktu zaczepienia (idea zaczerpnięta ze starych programów graficznych na komputery Atari). Plugin posiada ponadto funkcje odczytu i zapisu lokalnego oraz zdalnego, funkcję

drukowania oraz wyboru języka (obecnie polski i angielski). Obsługiwane formaty graficzne to JPG i PNG.

5.4.2 Algorytm wypełniania

Podobnie jak większość programów do rysowania posiada on przybornik wypełniania spójnych obszarów o jednolitym kolorze. Algorytm został zaprojektowany i zaimplementowany przez twórców plugina. Polega on na równomiernym rozchodzeniu się obszaru wypełniania we wszystkich kierunkach poprzez utworzenie punktów obwiedni. Algorytm korzysta z dwóch tablic przechowujących gotowe do kolorowania punkty obwiedni. Łatwo zauważyć że długość takiej tablicy dla równomiernego rozchodzenia nigdy nie przekroczy długości obwodu obrazka $2(a+b)$. Wynika to z tego, że równomierne, koncentryczne rozchodzenie wypełnienia zapewnia że obszar wypełniony jest zawsze wypukły, a największym obszarem wypukłym jest sam obraz (którego obwiednia ma właśnie długość obwodu). Zastosowanie dwóch takich tablic powoduje zatem zajętość pamięci rzędu dwóch długości obwodu - $4(a+b)$.

Działanie algorytmu polega na iteracyjnym, zamiennym procesie budowania obwiedni w jednej tablicy na podstawie obwiedni zawartej w jednej tablicy:

```
for (punkt in tablicaA) {  
    koloruj(punkt);  
    dodaj do tablicaB wszystkie punkty sąsiadujące, które  
    mogą zostać wypełnione;  
}
```

Następnie tablicaA jest zamieniana z tablicaB i proces iteracyjny trwa tak długo, aż nowo wypełniana tablica będzie pusta (brak punktów do wypełnienia).

6. Testowanie systemu

6.1 Testy jednostkowe

Jedną z głównych motywacji stworzenia Javy było stworzenie systemu wspierającego tworzenie aplikacji niezawodnych i bezpiecznych. Mechanizmem podwyższającym bezpieczeństwo i niezawodność aplikacji w językach obiektowych jest mechanizm wyjątków. Wyjątki, mimo że zmniejszają liczbę niezauważonych błędów w systemie, nie są w stanie usunąć ich wszystkich.

Podczas testowania systemu VMD użyto testów jednostkowych. Zadaniem testów jednostkowych jest sprawdzenie poprawności kodu programu. Java zapewnia standardową bibliotekę (JUnit) do testowania aplikacji. Za pomocą JUnita testowaliśmy pojedyncze klasy, a także grupy kilku klas. Dobieranie kilku klas w celu przeprowadzenia testu jednostkowego nie było przypadkowe. Klasy były tak dobierane, aby reprezentować pewną funkcjonalność systemu. Jako funkcjonalność systemu rozumie się w tym kontekście pewne pojedyncze czynności, które może wykonywać system, takie jak: logowanie użytkownika, sprawdzenie typu, wysłanie plugina, wylogowanie użytkownika. Dzięki testowaniu funkcjonalności systemu a nie pojedynczych klas możemy zbadać system w aspekcie jego konkretnych działań. Testowanie systemu względem funkcjonalności wymaga więcej czasu na przygotowanie testów, szczególnie że przekrój klas dla pewnych testów nie jest zbiorem pustym. Jednak takie potraktowanie testowania przyczynia się do kontekstowego badania poprawności danego kodu, i wykrycia większej liczby błędów.

Oto przeprowadzone testy:

- dodawanie i usuwanie użytkowników – testowanie klas `CsvUserIO` oraz `RegisteredUsers`. Testowanie tych klas pozwoliło na znalezienie błędów związanych z utrzymywaniem użytkowników na dysku; błąd pierwszy polegał na błędzie w kolejności zapisu składowych użytkownika do pliku, błąd drugi polegał na zachowaniu się programu w przypadku pliku pustego, lub w przypadku pliku z niepełną informacją o użytkowniku,
- dodawanie i usuwanie typów – nie odnaleziono błędów w klasach `CsvTypeIO` i `RegisteredTypes`,
- przetestowano (lokalnie) logowanie się użytkownika – w ten sposób przetestowano klasy `RemoteServerImpl` oraz klasę `Encoder` (odpowiedzialną za haszowanie hasła). W ten sposób przetestowano współdziałanie klas `ServerGlobalState` oraz `RegisteredUsers`,
- przetestowano klasę `FsChangeListenerServer` sprawdzając poprawność dodawania listenerów, ścieżek nasłuchu do już zarejestrowanych listenerów, rozszerzanie typów nasłuchu już istniejących ścieżek nasłuchu, a także usuwanie listenerów i ograniczanie typów nasłuchu
- jako klasę o podwyższonym ryzyku, w związku z wykorzystaniem wątków, przetestowano osobno `UserGarbageCollector`. Odnaleziono i poprawiono niewielkie błędy związane z synchronizacją dostępu do zmiennej reprezentującej zbiór użytkowników (część błędów wskazał sam mechanizm wyjątków dostępu współbieżnego w javie).

6.2 Mock Object

System VMD, jako system typu klient-serwer niesie pewne problemy związane z testowaniem. Testowanie zachowań serwera często wymaga zaimplementowanych funkcjonalności klienta. Grupa inżynierska postanowiła skorzystać z możliwości jakie dają obiekty typu *mock*, w celu przetestowania zachowań serwera. *Mock Object* jest obiektem, który służy do przetestowania innego systemu. Zastępuje on system, który w chwili testowania jest jeszcze niedostępny. Mock Object dostarcza najmniejszej wystarczającej implementacji pewnego obiektu, która pozwala zbadać zachowanie drugiego obiektu. Aby zbadać zachowanie serwera VMD w prawdziwych warunkach, podczas gdy klient nie był jeszcze gotowy, zaimplementowano obiekt mock klienta. Dostarcza on podstawowej implementacji jaką było logowanie do systemu, dzięki niemu można było zweryfikować poprawność komunikacji i logowania na serwer. Oto przykładowa implementacja prostego obiektu mock klienta:

```
public class ClientMock {  
    private static final int DEFAULT_SERVER_PORT = 1410;  
    public static void main(String argv[]) {  
        RemoteServer server = null;  
        Registry vmdRegistry = null;  
        ClientTwoWaySocketFactory fac = new ClientTwoWaySocketFactory();  
        try {  
            RMISocketFactory.setSocketFactory(fac);  
            fac.establishSignallingChannel("localhost", DEFAULT_SERVER_PORT);  
        } catch (IOException e) {  
            System.out.println("Cannot establish signalling channel");  
            System.exit(0);  
        }  
        try {  
            vmdRegistry = LocateRegistry.getRegistry("localhost",  
                DEFAULT_SERVER_PORT, fac);  
        } catch (RemoteException e) {
```

```

        System.out.println("Cannot get remote registry because of: "
            + e.getMessage());
        System.exit(0);
    }
    try {
        server = (RemoteServer) vmdRegistry.getRemoteServer();
    } catch (RemoteException e) {
        System.out.println("Cannot get RemoteServer from registry");
        e.printStackTrace();
        System.exit(0);
    } catch (NotBoundException e) {
        System.out.println("Remote Server is not bound");
        e.printStackTrace();
        System.exit(0);
    }
    try {
        User user = server.login("login", "haslo");
        if (user == null) {
            System.out.println("Wrong login/password");
            System.exit(0);
        } else {
            System.out.println("User successfully logged in");
            RemoteFileFactory rff = user.getUserFileFactory();
            RemoteFile rmf = rff.newRemoteFile("plik.txt");
            if (rmf == null) {
                System.out.println("Given RemoteFile is NULL");
            } else {
                System.out.println(rmf.getPath());
            }
        } catch (RemoteException e) {
            System.out.println("RemoteException (user/fileFacotory):"
                + e.getMessage());
            e.printStackTrace();
            System.exit(0);
        } catch (IOException e) {
            System.out.println("cannot create given file");
        }
    }
    System.exit(0);
}
}

```

Obiekt mock pozwolił przetestować podstawową funkcjonalność serwera i komunikacji. Dzięki niemu w czasie gdy nie było jeszcze działającej implementacji klienta mogliśmy przetestować działanie projektów:

- VMDServer
- ServerKernel
- TwoWayCSsKernel

Mock Object zweryfikował następujące funkcjonalności :

- możliwość zalogowania się
- możliwość rejestracji obiektów na serwerze
- możliwość ściągnięcia pliku zdalnego
- możliwość pobrania obiektu zdalnego z rejestru
- możliwość odwołania się do rejestru
- działanie `UserGarbageCollectora`

Przykładowy obiekt mock był implementacją pewnego scenariusza użycia, który zakładał następujące czynności wykonywane przez klienta:

- Logowanie
- Pobranie zdalnego pliku
- Brak aktywności, która spowodować miała uruchomienie garbage collector na serwerze i wylogowanie użytkownika

Testowanie za pomocą Mock Object tego scenariusza użycia zakończyło się pełnym sukcesem. Za pomocą Mock Object przetestowano jeszcze 3 krótkie scenariusze

- Logowanie przy pomocy złego hasła
- Logowanie nieznanego użytkownika
- Logowanie kilku użytkowników o tym samym loginie i wymianę zdarzeń

Scenariusz użycia polegający na logowaniu kilku użytkowników o tym samym loginie pozwolił zweryfikować poprawność działania interfejsu współdzielonego listenera zmian w systemie plików.

W celu zweryfikowania poprawności klas zastosowano także metodę czarnej skrzynki (black box). Polega ona na testowaniu klasy na podstawie znajomości jedynie jej interfejsu i wymaganej funkcjonalności. W tym celu programiści zamienili się testowanymi klasami, tak aby nie testować klas swojego autorstwa. Jest to metoda przeciwna do white box, gdzie testy oparte są o znaną implementację. Testowanie odbyło się przy użyciu JUnita. Autorzy uznali, że połączenie testowania funkcjonalności za pomocą testowania kilku klas i użycia Mock Object, w połączeniu z nowym spojrzeniem na klasy, jakie da zastosowanie metody black box i drugiego testera, a także korzystanie z raportów Mavena, pozwoli zminimalizować liczbę błędów w projekcie.

7. Zakończenie

7.1 Istotne zalety projektu

System opisany w niniejszej pracy spełnia wymogi przed nim postawione. Uzyskano oprogramowanie, które z powodzeniem można zastosować do wspólnej pracy grupy użytkowników za współdzielonych zasobach. Para aplikacji klient – serwer zapewnia synchronizację pracy i współdzielenie zgromadzonych na serwerze zasobów. Zestaw aplikacji dołączonych w postaci pluginów do systemu zapewnia podstawową funkcjonalność aplikacji klienckiej jako desktopu –

interfejsu dostępowego do plików użytkowników. Projekt zakłada jednak, że pakiet aplikacji nie jest kompletny. Przyjęta architektura zapewnia możliwość łatwej rozbudowy systemu o kolejne aplikacje, zarówno lokalne jak i posiadające części serwerowe i klienckie. W pracy często używa się wobec projektu stwierdzenia „framework” w rozumieniu szkielet, podstawa. Określenie to w kontekście systemu VMD jest jak najbardziej słuszne. Komplet klas i interfejsów jądra systemu oraz dokumentacja programistyczna w postaci javadoców dołączone na płycie CD mogą być podstawą przyszłego rozwoju systemu i jego rozbudowy o kolejne aplikacje. Zaimplementowane pluginy , ale także pliki klas klienta są jedynie przykładem wykorzystania stworzonego frameworka do wizualizacji zasobów. Jednak programiści chcący rozwijać w przyszłości system VMD mogą łatwo uniezależnić się od programu klienckiego czy serwerowego tworząc jego własne implementacje oparte o jądro systemu. Jest ono właściwie mechanizmem zapewnienia spójności rozproszonych operacji, interfejsem dostępu do zasobów oraz warstwą komunikacyjną systemu. Wizualizacja może jednak ulec całkowitej zmianie przez zmianę implementacji choćby samej aplikacji Desktopu jako pulpitu, który sam jest pluginem do systemu.

7.2 Możliwe sposoby wykorzystania systemu

Podstawowym i domyślnie zaimplementowanym sposobem wykorzystania VMD jest współdzielenie plików i ustawień wizualizacji. Jest to jednak tylko przykład, który pokazuje jak szeroko można wykorzystać opisywany system. Można wyobrazić sobie plugin, którego faktycznym wykorzystaniem nie będzie łączenie klienta z serwerem, ale w dwóch i więcej serwerów ze sobą. Korzystając z obiektów reprezentujących stan serwera może on wymieniać pomiędzy nimi dane dotyczące listy użytkowników, pozwalając logować się do grupy serwerów,

widzianych pod adresem dowolnego serwera z grupy. Mechanizm ten jest podobny do *single sign-on*, stosowanego w systemach sieciowych w celu ułatwienia dostępu użytkownikom do zasobów zgromadzonych na grupie serwerów. W systemach typu *single sign-on* użytkownik jest jednak zmuszony do przechodzenia między kolejnymi serwerami w celu osiągnięcia pożądanego zasobu. W opisywanym scenariuszu to plugin serwera troszczyłby się o dostępność zasobów zgromadzonych na innych komputerach.

Inną możliwą ścieżką wykorzystania systemu mogłoby być monitorowanie stanu zdalnego komputera. Plugin kliencki rejestrowałby zdarzenia zachodzące w systemie na którym działa aplikacja kliencka, a następnie przy wykorzystaniu agenta serwerowego przekazywał je do drugiej aplikacji klienckiej umożliwiając monitorowanie tego, co aktualnie dzieje się w systemie. System taki mógłby na przykład monitorować wykorzystanie służbowego sprzętu w godzinach pracy czy też pomóc rodzicom w monitorowaniu wykorzystania komputera przez dzieci.

Kolejną koncepcją wykorzystania VMD jako systemu przekazywania zdarzeń jest zdalna administracja wieloma komputerami dedykowane do konkretnych aplikacji systemu pluginy mogą, kierowane przez innego klienta uruchamiać bądź zatrzymywać aplikacje takie jak konserwacja czy porządkowanie systemu. Stosując mechanizm serializacji i odtwarzania można by również zaimplementować system archiwizacji plików pozostawianych na komputerze klienckim bądź przez serwer systemu VMD bądź przez dedykowanego klienta.

7.3 Podsumowanie

Twórcy projektu wierzą, że zaprojektowana architektura sprosta różnym wymaganiom i zastosowaniom, będąc jednocześnie narzędziem wygodnym dla

programistów. Gotowe rozwiązania zastosowane we frameworku pozwalają na szybkie pisanie konkretnych systemów, które wymagają głównie implementacji gotowych interfejsów oraz budowy specyficznej funkcjonalności dla danego systemu.

8. Bibliografia

1. Alistair Cockburn, „Writing effective use cases” , Addison-Wesley, 2000
2. Andrzej Jaskiewicz, Inżynieria Oprogramowania, Helion 1997
3. Jan Bielecki, „Java 3 RMI. Podstawy programowania rozproszonego”, Helion 1999
4. Bruce Eckel, „Thinking in Java”. Wydanie 3. Edycja polska, 2003
5. Bruce Eckel, „Thinking in Enterprise Java”, Revision 1.1 - May 6, 2003
6. BruceEckel, “Thinking in Patterns”, Revision 0.9 - May 20, 2003
7. <http://maven.apache.org/index.html>, Maven documentation
8. Adam Bochenek, „Prosty przepis na J2EE: JBoss, Eclipse i komponenty EJB”, Mikom 2005
9. Matt Zandstra, „PHP5. Obiekty, wzorce, narzędzia”, Helion 2004 (w zakresie wzorców)
10. Dokumentacja bibliotek Suna: <http://java.sun.com/j2se/1.5.0/docs/api/>
11. Dokumentacja, opis i kod źródłowy biblioteki **TwoWaySocketFactory**:
<http://www.cssassociates.com/rmifirewall.html>
12. Dokumentacja systemu Xen:
<http://www.cl.cam.ac.uk/Research/SRG/netos/xen/documentation.html>

13. “Xen and the Art of Virtualization”, Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield, University of Cambridge Computer Laboratory
14. “Java. Wprowadzenie”, Patrick Niemeyer, Jonathan Knudsen, O’Reilly 2003, tłum. Rafał Jońca, wyd. Helion, Gliwice 2003

9. Załączniki

9.1 Płyta CD

- Kod źródłowy oprogramowania
- Wersja dystrybucyjna
- Elektroniczna wersja pracy dyplomowej
- Dokumentacja JavaDoc do jądra frameworka
- Narzędzia
 - Eclipse Java IDE
 - Genady RMI plugin for Eclipse v 1.6.5.4
 - Maven 2.0.2