Uniwersytet Mikołaja Kopernika
Wydział Matematyki i Informatyki
Zakład Obliczeń Równoległych i Rozproszonych

Tomasz Rękawek
nr albumu: 214364

Praca magisterska
na kierunku informatyka

Dostęp do rozproszonych zasobów plikowych w systemie UNICORE

Opiekun pracy dyplomowej
dr Krzysztof Benedyczak
Zakład Obliczeń Równoległych i Rozproszonych

Toruń 2011

Pracę przyjmuję i akceptuję

.....
data i podpis opiekuna pracy

Potwierdzam złożenie pracy dyplomowej

.....
data i podpis pracownika dziekanatu

Spis treści

Wstęp

Systemy gridowe pozwalają połączyć moc obliczeniową wielu różnych maszyn (często odległych od siebie) i udostępnić ją w spójny sposób użytkownikom. Celem jest zwykle przeprowadzenie skomplikowanych obliczeń, związanych z dziedzinami takimi jak chemia, biologia, fizyka i inne. Kilka popularnych programów pozwalających na stworzenie systemu gridowego (aplikacje tego typu zwane są oprogramowaniem warstwy pośredniej) to NorduGrid's ARC, gLite, Globus Toolkit czy UNICORE. Znany jest również system BOINC, na którym opierają się projekty takie jak SETI@home.

Oprócz mocy obliczeniowej, współdzielone są także inne zasoby, na przykład przestrzeń dyskowa. W zależności od rodzaju oprogramowania warstwy pośredniej, istnieją różne sposoby dostępu do danych. Standardy te dostosowane są do potrzeb systemu gridowego w zakresie bezpieczeństwa, skalowalności, możliwości rozproszenia, itd. Z tego powodu nie używa się np. powszechnie znanego protokołu FTP czy SCP. Często spotykanym interfejsem dostępowym jest *Storage Resource Manager* (SRM), natomiast środowisko UNICORE, którego niniejsza praca w dużym stopniu dotyczy, używa usługi o nazwie *Storage Managament Service* (SMS).

Usługa SMS nie pozwala niestety na rozproszenie danych, tzn. nie jest w niej możliwe połączenie przestrzeni dyskowych kilku serwerów w jedną logiczną całość. Taka funkcja byłaby przydatna, gdyż użytkownik gridu nie musiałby się już troszczyć o wybór maszyny, na którym chce umieścić swoje pliki. Serwery udostępniałby swoje dyski w ramach jednej przestrzeni nazw, organizowanej przy użyciu katalogów, przy czym logiczne położenie danych (ścieżka do pliku) nie zależałoby od położenia fizycznego. Niniejsza praca jest właśnie próbą znalezienia rozwiązania rozproszonego przechowywania danych dla środowiska UNICORE.

Na początku, w rozdziale 1, przedstawiono podstawowe pojęcia związane z systemami gridowymi i obsługą zasobów dyskowych. Następnie opisano kilka rozwiązań rozproszonego przechowywania danych, przeznaczonych głównie dla oprogramowania gLite, ale także samodzielnych, takich jak iRODS lub GPFS.

W rozdziale 2 opisano architekturę systemu UNICORE, szczególną uwagę poświęcając komponentom odpowiedzialnym za obsługę danych. Zaprezentowano też istniejące systemy, umożliwiające rozproszone przechowywanie danych w systemie UNICORE. Rozdział 3 zawiera analizę tych rozwiązań. Sformułowano również wymagania, które powinny zostać spełnione przez poszukiwany system.

Ponieważ żadne z rozwiązań nie wydawało się spełniać tych wymagań, w rozdziale 4 zaprezentowano architekturę nowego rozwiązania, powstałego w ramach niniejszej pracy. Rozdział 5 zawiera próbę analizy wydajności i stabilności utwo-

Rozdział 1

Istniejące rozwiązania

1.1. Podstawowe pojęcia

Autorzy książki *The Grid: Blueprint for a new Computing Infrastructure* definiują system gridowy jako „strukturę obliczeniową, pozwalającą na swobodne współdzielenie zasobów takich jak moc obliczeniowa, bazy danych, interfejsy i ludzie" [9]. Jeden z autorów tej pozycji — Ian Foster — utworzył listę 3 warunków [8], które muszą być spełnione, aby daną strukturę nazwać systemem gridowym:

1. koordynuje zasoby, które nie podlegają centralnemu zarządzaniu,
2. używa otwartych protokołów, standardów i interfejsów,
3. dostarcza usług o wysokiej jakości: dużej szybkości, przepustowości, pojemności, bezpieczeństwie, itd. Użyteczność całego systemu gridowego powinna być wyższa niż użyteczność sumy jego części.

Natomiast wg. angielskiej Wikipedii „termin «grid» odnosi się do kombinacji zasobów komputerowych pochodzących z różnym domen administracyjnych służących osiągnięciu wspólnegu celu" [36].

Oprogramowanie służące utworzeniu systemu gridowego nazywane jest oprogramowaniem warstwy pośredniej (ang. *middleware*) — jest ono niejako „pomiędzy" konkretnym zadaniem, które należy wykonać a komputerami, które powinny się tym zająć. Programy tego typu zazwyczaj opierają się (zgodnie z trzecim punktem listy Fostera) na otwartych standardach. Dość powszechne jest stosowanie w nich architektury zorientowej na usługi (service-oriented) i używanie usług sieciowych *web services* jako interfejsu [10, str. 10].

Jednym z elementów zasobów współdzielonych w ramach systemu gridowego jest przestrzeń dyskowa. Dane i wyniki obliczeń przeprowadzanych przez system gridowy to często wielkie ilości ogromnych plików — przykładowo dane przechowywane w ramach omawianego niżej systemu CASTOR w CERN to ponad 40 petabajtów. Niezbędne są więc specjalne mechanizmy pozwalające na zarządzanie przestrzenią i danymi. Część oprogramowania warstwy pośredniej posiada wbudowane podsystemy zarządzania danymi, w innych trzeba wykorzystać rozwiązanie zewnętrzne. W tym rozdziale przedstawiono kilka najpopularniejszych.

Rozwiązania takie można podzielić na kilka grup. Pierwsza z nich to rozproszone systemy plików, takie jak IBM General Parallel File System (GPFS). Po zainstalowaniu takiego systemu zyskujemy jedną połączoną przestrzeń dyskową, do której możemy dostać się np. za pomocą funkcji standardu POSIX oferowanych przez system operacyjny. Kolejna kategoria to „disk-pool". Oprogramowanie tego typu wykorzystuje istniejące systemy plików na wielu serwerach i również łączy je we wspólną przestrzeń. Dostęp do tej przestrzeni odbywa się za pomocą specjalnie zdefiniowanego interfejsu. Bardziej skomplikowane oprogramowanie (jak omawiany poniżej dCache lub CASTOR) potrafi również wykorzystywać zautomatyzowane biblioteki taśm. Przedstawicielem minimalistycznego podejścia jest StoRM. Udostępnia on po prostu, za pomocą interfejsu charakterystycznego dla systemu gridowego, wybrany katalog z systemu plików. W celu rozproszenia danych należy w tej sytuacji wykorzystać odpowiedni system plików, np. wspomniany GPFS.

1.2. Usługi dostępu do danych w systemie gridowym gLite

Ponieważ większość z opisanych tu systemów rozproszonego przechowywania danych współpracuje z oprogramowaniem warstwy pośredniej gLite, niezbędne jest wprowadzenie kilku charakterystycznych dla niego pojęć. Poniższe definicje nazw i adresów są zgodne z używanymi przez usługę *LCG File Catalogue* (LFC).

Jeśli użytkownik systemu gridowego chce uzyskać dostęp do jakiegoś pliku, posługuje się jego przyjazną nazwą *Logical File Name* (LFN). Jeden plik może mieć kilka LFN, na podobnej zasadzie, jak jeden plik w Linuksie ma kilka twardych dowiązań (ang. *hard links*). LFN tłumaczona jest na *Grid Unique Identifier* (GUID). GUID to unikalny ciąg znaków tworzony na podstawie aktualnego czasu i np. adresu sprzętowego MAC karty sieciowej — jak sama nazwa wskazuje pozwala on na jednoznaczną identyfikację pliku w ramach całego systemu gridowego. Fizyczną lokalizację repliki pliku wskazuje *Storage URL* (SURL). Ponieważ jeden plik może być replikowany w wielu różnych miejscach, do jednego identyfikatora GUID może być przypisane wiele adresów SURL. SURL prowadzi do jednostki zwanej *Storage Element*, czyli serwera lub grupy serwerów, które odpowiadają za fizyczne przechowanie i odczyt pliku na dyskach lub taśmach. Serwery Storage Element udostępniają swoje zasoby za pomocą usługi *Storage Resource Manager*, do której odnosi się adres SURL. Większość rozwiązań opisanych w poniższym rozdziale to właśnie menedżerowie SRM. SRM jest Po połączeniu z usługą SRM, zwraca ona adres *Transport URL*, który umożliwia wczytanie lub zapis pliku na SE. O ile adresy LFN i GUID to elementy charakterystyczne dla usługi LFC, wykorzystywanej przez system gLite, to interfejs SRM wraz z adresami SURL i TURL są ogólnymi standardami stworzonymi przez Open Grid Forum[25].

Adres TURL zawiera informację o protokole, który powinien zostać użyty do transmisji pliku. Najczęściej spotykane to: GridFTP[24], RFIO[28], XROOTD[37]. Pierwszy z nich to rozszerzony w zakresie bezpieczeństwa protokół FTP. Dwa pozostałe umożliwiają dostęp do zdalnych plików za pomocą funkcji przypominających te ze standardu POSIX.

Usługą, która pozwala na translację adresów LFN na GUID i SURL, jest

LFC. Usługa ta przechowuje również systemowe metadane plików (wielkość, czas dostępu, itd.) i — w ograniczonym zakresie — metadane użytkownika. Ograniczenie to polega na możliwości wykorzystania tylko jednego pola, gdyż twórcy LFC wyszli z założenia, że obsługą metadanych użytkownika powinna zajmować się osobna usługa [3, rozdział *LFC Architecture*]. Autoryzacja dostępu do danych może przebiegać przy użyciu praw dostępu systemu UNIX lub list kontroli dostępu standardu POSIX. Klienci mogą połączyć się z LFC za pomocą przygotowanych programów konsolowych, takich jak `lfc-ls`, `lfc-mkdir`, itd. Dostępne jest również API dla programów w języku C i biblioteki dla języków Python i Perl.

1.3. Disk Pool Manager

Disk Pool Manager (DPM) to „lekkie rozwiązanie przechowywania danych dla systemów gridowych" [15]. Udostępnia interfejs SRM i może być używany w systemie gLite jako Storage Element, przy czym przeznaczony jest do obsługi „raczej małych zasobów opartych jedynie o dyski" [35]. Wynika to z założeń, według których DPM powinien być łatwy w instalacji, konfiguracji oraz utrzymaniu, zapewniając jednak pełną funkcjonalność wymaganą przez gridowe systemy przechowywania danych, takie jak obsługa wielu serwerów dyskowych, systemów plików i replikacji [15].

Architektura menedżera DPM zakłada podział serwerów (zwanych tu węzłami) na dwa rodzaje: *head node* oraz *disk node*. Pierwszy z nich służy jako baza metadanych oraz punkt dostępu dla klientów. Serwery drugiego typu przechowują i udostępniają pliki. Na *head node* uruchomionych jest kilka demonów. Są to:

- **DPM nameserver daemon** (DPNS) — demon zajmujący się obsługą metadanych dotyczących plików i folderów,
- **SRM daemon** — demon udostępniający klientom interfejs SRM,
- **DPM** — główny demon, który zajmuje się obsługą żądań — korzysta przy tym z DPNS i demonami z *disk nodes*. Demon posiada również swoją bazę danych, w której przechowuje żądania,
- baza danych MySQL wykorzystywana przez powyższe aplikacje.

Na węzłach dyskowych uruchomione są demony odpowiedzialne za zdalny dostęp do plików poprzez protokoły GridFTP, RFIO, XROOTD, itd. Dane przechowywane na węźle dyskowym umieszczone są po prostu w wybranym katalogu na lokalnym systemie plików. DPM musi być właścicielem takiego katalogu i plików.

Oczywiście może istnieć więcej niż jeden węzeł dyskowy. Demony przypisane do węzła *head node* mogą być rozmieszczone na różnych serwerach, jednak nie ma możliwości ich replikacji. Aby zrozumieć działanie menedżera DPM i wzajemne relacje poszczególnych demonów i węzłów, warto posłużyć się przykładem, w którym klient chce wysłać plik na zasób dyskowy za pomocą interfejsu SRM.

Pierwszą operacją będzie wywołanie operacji *SRM put* na demonie SRM. Demon SRM połączy się z demonem DPM i doda żądanie do jego bazy danych. Klient otrzyma identyfikator dodanego żądania. Następnie DPM pobiera

żądanie z bazy danych i za pomocą demona DPNS sprawdza, czy użytkownik ma uprawnienia do danej operacji. Jeśli tak, to do demona DPNS dodawana jest informacja o pliku. Kolejną operacją wykonaną przez DPM jest wybór węzła dyskowego, na którym zostanie umieszczony plik. Po tym wyborze żądanie w bazie zostaje uzupełnione o adres *Transport URL* i oznaczone jako „gotowe". Klient co jakiś czas odpytuje demon SRM używając operacji *getRequestStatus*. Gdy żądanie zostanie już zrealizowane i klient otrzyma adres TURL, łączy się z odpowiednim serwerem dyskowym i wysyła plik używając na przykład protokołu GridFTP[19].

1.4. dCache

dCache to kolejne rozwiązanie rozproszonego przechowywania danych. W porównaniu do DPM jest on o wiele bardziej skomplikowany, posiada jednak również znacznie większe możliwości. Powstał w wyniku wspólnych prac Deutsches Elektronen-Synchrotron w Hamburgu i Fermi National Accelerator Laboratory niedaleko Chicago.

Założeniem dCache'a jest to, aby zasoby dyskowe (i nie tylko) z wielu różnych maszyn były dostępne dla użytkownika końcowego jako jedna wielka przestrzeń, przy czym użytkownik nie musi wcale wiedzieć, gdzie fizycznie znajdują się jego dane [30, str. 2]. *Z tego powodu dane mogą migrować między serwerami bez żadnych przerw w działaniu systemu. Umożliwia to również usuwanie i dodawanie kolejnych serwerów w taki sposób, że użytkownik końcowy nawet tego nie zauważy* [30, str. 2]. Kolejną ważną cechą dCache'a jest obsługa pamięci trzeciorzędnej (*tertiary storage*) — czyli biblioteki taśm obsługiwanej przez robota. Pamięć taka charakteryzuje się bardzo dużym czasem dostępu, ale za to zwiększenie jej pojemności jest tanie. Dla zwiększenia wygody użytkownika aktualnie używane dane z taśm magnetycznych są kopiowane na dyski. dCache potrafi także wykrywać szczególnie często otwierane pliki (tzw. *hot spots*) i replikować je na wielu węzłach, aby rozłożyć obciążenie na większą ilość maszyn. Proces ten może być jednak kontrolowany ręcznie tak, aby uzyskać optymalne wykorzystanie zasobów [30, str. 2].

Natywnym protokołem dostępu do dCache jest dCap. Do pakietu dołączona jest biblioteka w języku C, która daje możliwość korzystania z funkcji standardu POSIX takich jak `open`, `read`, `write`, `seek`, itd. Protokół obsługuje mechanizmy bezpieczeństwa takie jak GssApi i SSL. Jest on także odporny na awarie sieci i węzłów. Oprócz protokołu dCap dCache obsługuje rozmaite odmiany protokołu FTP takie jak GssFtp[1] i GridFTP [12, str. 1108].

Podstawowym elementem budującym system dCache jest komórka (*cell*). Komórka jest modułem napisanym w Javie, wykonującym jedno proste zadanie. *W celu wykonania tego zadania komórka może komunikować się z innymi komórkami. Komórki można podzielić na typy takie jak np.: „doors" i „pools". Komórki tego samego typu wykonują podobne zadania (np. przechowują lub udostępniają pliki). Większość komórek występuje w wielu instancjach, a dCache jest tak zaprojektowany, aby rozkładać obciążenie między tymi instancjami* [30, str. 2].

[1] Protokół FTP rozszerzony o uwierzytelnienie i autoryzację przy użyciu protokołu Kerberos.

Komórki uruchamiane są w ramach domeny. Domena to wirtualna maszyna Javy, rozszerzona o pewne dodatkowe możliwości. dCache w swojej domyślnej konfiguracji ma pewną ilość domen zdefiniowanych według zadań, które każda domena ma wykonać. Zadanie te, to na przykład przechowanie danych lub nazw plików. Zwykle nie ma potrzeby modyfikacji listy komórek działających w ramach danej domeny. Na jednym węźle (serwerze) zwykle uruchomionych jest kilka domen, istnieje możliwość swobodnego ich przenoszenia między serwerami. Warto wspomnieć, że proces przeniesienia domeny z jednego węzła na drugi jest bardzo prosty, co również ułatwia równoważenie obciążenia na poszczególnych komputerach[30, str. 2].

Jak już powiedziano, konfiguracja dCache jest bardzo elastyczna, istnieją jednak pewne konwencje. Autorzy [30] zalecają, aby podzielić węzły na trzy grupy:

- **head node** — centralny węzeł, przechowujący bazę Postgres, serwer nazw i inne usługi, które występują w jednej instacji,
- **door node** — węzły dostępowe, udostępniające dane za pomocą jednego z obsługiwanych protokołów,
- **pool node** — węzły przechowujące zasoby.

Warto wspomnieć jeszcze o usłudze Chimera, pełniącej istotną rolę serwera nazw w dCache. Chimera przechowuje dane o przyjaznych nazwach plików i katalogów oraz pozwala przyporządkować je wewnętrznym identyfikatorom zasobów dCache. Obsługuje także metadane, które użytkownik może przypisać poszczególnym zasobom. Dane te przechowywane są w bazie PostgreSQL. dCache łączy się z Chimerą za pomocą specyficznego modułu o nazwie PnfsManager. Dane o przestrzeni nazw udostępnione są jednak również za pomocą protokołu NFS V3, dzięki czemu administracja nazwami plików możliwa jest za pomocą dowolnego klienta NFS i komend takich jak `ls` lub `mkdir`. Warto jednak zwrócić uwagę na to, że komenda `cp` już nie zadziała — albowiem Chimera odpowiada jedynie za nazwy plików, a nie za ich zawartość. Jeśli pojawia się konieczność bardziej wyrafinowanego dostępu do metadanych, możliwe jest wykonanie zapytań SQL bezpośrednio na bazie danych.

Chimera posiada także mechanizm „wyzwalaczy” (*triggers*) dzięki któremu administrator może zdefiniować akcje, które powinny został wywołane np.: po usunięciu lub zmianie nazwy pliku. Katalogom można przypisać tzw. znaczniki (ang. *tag*), czyli specjalny rodzaj metadanych, które będą dziedziczone przez podkatalogi. Znaczniki mogą mieć znaczenie czysto informacyjne lub określać na przykład, na której taśmie powinny zostać zapisane pliki z danego katalogu i jego podkatalogów. Funkcją o najbardziej interesującej nazwie są tzw. *wormholes*. Są to odpowiedniki plików ukrytych, a więc takich, które nie wyświetlają się podczas wywołania komendy `ls`, a mimo to są dostępne [21].

1.5. CASTOR

Kolejnym rozwiązaniem jest CERN Advanced STORage manager czyli CASTOR (nie należy go mylić z komercjalnym produktem CAStor 5 firmy Caringo). Podobnie jak dCache, CASTOR obsługuje pamięci trzeciorzędne i używanie

dysków jako pamięci podręcznej (*cache*). Jak sama nazwa wskazuje, program jest rozwijany w CERN-ie i od samego początku projektowany był do obsługi ogromnych ilości danych. Obecnie (luty 2011) instalacja w CERN-ie obsługuje ponad 40 PT danych na dyskach i taśmach [4]. CASTOR składa się z wielu bezstanowych demonów, odpytujących centralną bazę danych. Umożliwia to ich łatwą replikację i zwiększa skalowalność całego systemu [20, str. 2].

Żądania użytkowników umieszczane są w bazie przez usługę *request handler*. Ze względu na proste zadanie, usługa ta jest bardzo lekka i potrafi obsłużyć do 100 żądań na sekundę bez żadnych widocznych opóźnień. W środowisku gridowym użytkownicy nie łączą się jednak bezpośrednio z usługą *request handler* lecz używają interfejsu SRM [20, str. 3]. Protokoły transportowe obsługiwane przez CASTOR to RFIO, ROOT, XROOT i GridFTP. Gdy żądanie trafi już do bazy, jego obsługą zajmuje się (również bezstanowy) demon *Stager*. Składa się on z wielu usług, odpowiedzialnych za różne rodzaje operacji.

Kolejny demon odpowiada za monitorowanie użycia zasobów. Wynik jego działania jest wykorzystywany podczas szeregowania zadań. CASTOR pozwala na użycie zewnętrznego oprogramowania szeregującego, takiego jak komercyjny LSF lub darmowy i otwarty Maui Cluster Scheduler [22]. Szeregowanie zadań na podstawie aktualnego użycia zasobów pozwala na optymalne rozłożenia obciążenia.

Serwer nazw odpowiada za przechowanie hierarchii katalogów. Wg. stosowanej konwencji ścieżki plików mają postać `/castor/cern.ch/dir/file`, gdzie pierwsza część (`/castor`) jest zawsze stała, zaś druga (`/cern.ch`) to nazwa węzła, na którym znajduje się serwer nazw. Hierarchia katalogów poniżej nazwy węzła zależy już w pełni od administratora [20, str. 3].

Logowanie i rozliczanie odbywa się za pomocą usługi *Distributed Logging Facility* (DLF). DLF składa się z centralnego serwera i biblioteki klienckiej, używanej we wszystkich modułach menedżera CASTOR [20, str. 3].

Odśmiecanie (*garbage collecting*) dyskowej pamięci podręcznej odbywa się za pomocą zdefiniowanych polityk, które znajdują pliki nie będą potrzebne w najbliższym czasie i mogą zostać usunięte.

1.6. StoRM

StoRM, rozwijany we włoskim CNAF[2], jest najbardziej minimalistycznym z dotychczas omawianych rozwiązań. Zrzuca on odpowiedzialność za rozproszenie danych na niższą warstwę, czyli system plików, dokładając do niego implementację interfejsu SRM. Obsługiwane są wszystkie systemy plików zgodne ze standardem POSIX. Oprócz tego StoRM obsługuje również dodatkowe funkcje rozproszonych systemów plików, takich jak IBM General Parallel File System (GPFS) czy Lustre. Możliwy jest bezpośredni dostęp do danych za pomocą operacji standardu POSIX, bez żadnego pośrednictwa menedżera StoRM. Autoryzacja dostępu do plików jest oparta o listy kontroli dostępu standardu POSIX.

StoRM składa się z dwóch głównych komponentów. Część interfejsowa (frontend) udostępnia usługę SRM, uwierzytelnia użytkowników i zapisuje żądania

[2] *Centro Nazionale per la Ricerca e Sviluppo nelle Tecnologie Informatiche e Telematiche* — Narodowe Włoskie Centrum Badań i Rozwoju Technologii Informacyjnych i Transmisji Danych

w bazie danych. Natomiast część funkcjonalna (backend) zajmuje się wykonaniem żądań, transferem plików, przechowaniem metadanych, itd. Jest ona w stanie wykorzystać zaawansowane funkcjonalności systemu plików. Obsługa nowych systemów plików jest możliwa dzięki mechanizmowi wtyczek (plugins). [6, str. 3] Baza danych używana jest do przechowania żądań i metadanych. Zastosowany system tłumaczenia nazw SURL na fizyczne ścieżki do plików na dysku jest bardzo prosty — dzięki czemu uzyskanie dostępu do pliku posiadając tylko jego adres SURL nie wymaga użycia bazy danych [6, str. 4]. W praktyce taki sposób dostępu jest wykorzystywany przy pomocy protokołu `file://` w adresach TURL. Oczywiście zadanie wykorzystujące plik musi być wówczas uruchomione na maszynie, na której zamontowano dany system plików.

Z powodu możliwości bezpośredniego dostępu do dysku, ciężar zapewnienia bezpieczeństwa również spoczywa na systemie plików. W tym celu wykorzystywany jest mechanizm ACL standardu POSIX. Ponieważ jednak listy ACL odnoszę się do systemowych użytkowników danej maszyny, a nie do użytkowników systemu gridowego, StoRM wykorzystuje zewnętrzne mechanizmy mapowania [6, str. 7].

1.7. iRODS

„i" Rule Oriented Data System jest rozwiązaniem rozproszonego przechowywania danych rozwijanym w Data Intensive Cyber Environments Center w Uniwersytecie Karoliny Północnej. Jest on następcą Storage Resource Broker. Nie jest jednak (tak jak dotychczasowo opisane systemy) związany bezpośrednio z żadnym systemem gridowym, w szczególności nie udostępnia usługi SRM. iRODS ma dość rozbudowane i ciekawe mechanizmy umożliwiające dopasowanie jego działania do konkretnych potrzeb. System pozwala na połączenie ze sobą wielu przestrzeni dyskowych. Wspólna przestrzeń, którą otrzymujemy, może być zajmowana przez kolekcje (odpowiedniki katalogów) i obiekty danych (odpowiedniki plików). Każda akcja użytkownika (taka jak próba dodania lub pobrania obiektu, utworzenie kolekcji, itd.) pociąga za sobą wykonanie pewnych *akcji* złożonych z ciągu wywołań *mikrousług*. O tym, jakie akcje zostaną wykonane, decydują zdefiniowane wcześniej *reguły* (stąd nazwa systemu). Akcje i reguły mogą być dowolnie definiowane, natomiast mikrousługi to funkcje napisane w C włączone w iRODS. Dzięki takiemu mechanizmowi iRODS jest bardzo elastyczny i w dość prosty sposób można np. utworzyć regułę, wg. której każde zdjęcie wczytane na serwer będzie miało automatycznie utworzoną miniaturę.

Centralną usługą informacji o zasobach iRODS jest iCAT — baza danych oparta o serwer PostgreSQL, MySQL lub Oracle. Użycie iCAT polega na tworzeniu zapytań do bazy, korzystając z wcześniej zdefiniowanych stałych [18]. iCAT przechowuje listy:

- zasobów dyskowych,
- użytkowników i grup,
- kolekcji i obiektów,
- metadanych systemowych i użytkownika,
- uprawnień,

- reguł wykonywanych cyklicznie.

Metadane dot. kolekcji i obiektów mogą być dowolnie definiowane, wg. schematu: nazwa, wartość, jednostka.

1.8. OGSA-DAI

OGSA-DAI to rozwiązanie rozwijane przez The Edinburgh Parallel Computing Centre. Jest to oprogramowanie umożliwiające spójny dostęp (za pomocą usług web service) do różnego typu danych — nie tylko do plików, ale też np. do bazy XML. Są to tzw. „zasoby danych" (*Data Resource*). Klient łączy się z serwerem za pomocą jednej ze zdefiniowanych „usług danych" (*Data Service*) — usługa taka może być łączona z dowolną ilością zasobów. Połączenie następuje za pośrednictwem interfejsu SOAP, dostępne jest też gotowe API (*client toolikit*), które można dołączyć do aplikacji klienckiej [2].

Ze względu na dużą różnorodność zasobów danych powstała konieczność stworzenia spójnego interfejsu. Interfejsem takim są „aktywności" (*activities*). Aktywność wykonana na określonych danych przy określonych parametrach stanowi „zadanie" (*task*). Aktywność z założenia ma spełniać jedną, podstawową funkcję. Aktywności można podzielić na 3 grupy [2]:

- *statement* — zapewnienie dostępu do bazy danych za pomocą zapytania (np. `sqlQueryStatement` lub `xPathStatement`),
- *translation* — transformacja danych (np. `xslTransform`, `zipArchive`),
- *delivery* — dostarcza lub pobiera dane z zewnętrznych zasobów (np. `deliverToUrl`, `deliverFromGFTP`).

Programiści mają możliwość tworzenia własnych aktywności.

1.9. General Parallel File System

GPFS to rozproszony system plików stworzony przez firmę IBM. Jego rozwój nie jest związany z żadnych z systemów gridowych, niemniej jednak może być w nich używany (na przykład w połączeniu z opisanym powyżej menedżerem StoRM). Jego poprzednikiem był system Tiger Shark, nakierowany głównie na multimedia.

GPFS kładzie nacisk na obsługę przetwarzania równoległego, co wiąże się z jednoczesnym dostępem do danych. Aby zapewnić wysoką wydajność, każdy duży plik umieszczany w GPFS jest dzielony na mniejsze części (bloki) i rozdzielany na serwery dyskowe przy użyciu algorytmu round-robin. Podczas odczytu danych, są one pobierane ze wszystkich dysków jednocześnie i umieszczane w buforze. Analogicznie w czasie zapisu — dane najpierw wędrują do bufora, a później zapisywane są na wszystkich dyskach jednocześnie. System GPFS rozpoznaje najczęściej stosowane sposoby dostęu do danych w pliku (np. sekwencyjny). Jeśli dana aplikacja odczytuje plik w niestandardowy sposób, istnieje możliwość dostosowania opisanego mechanizmu do swoich potrzeb. Metadane także umieszczone są w blokach i rozdystrybuowane na wszystkich serwerach, dzięki czemu nie istnieje tzw. pojedynczy punkt awarii (*single point of failure*).

Dzięki zastosowaniu mechanizmu hashowania, katalogi mogą mieścić miliony plików bez utraty wydajności. Spójność zapewnia wykorzystanie dziennika, który zapisuje wszystkie zmiany jeszcze przed umieszczeniem ich w systemie plików [29].

1.10. Google File System i Hadoop Distributed File System

Google File System (GFS) jest własnościowym systemem plików, opracowanym w firmie Google [13] w 2003 roku. GFS zostało stworzone w odpowiedzi na rosnące potrzeby Google. Założeniem GFS jest, że wykorzystywana jest duża ilość tanich (a przez to awaryjnych) serwerów. W związku z tym niezbędna jest automatyczna replikacja danych i monitorowanie stanu całego systmu.

„Klaster GFS składa się z pojedynczego serwera głównego (ang. *master*) i wielu serwerów dyskowych (*chunkserver* — od ang. *chunk*, dosł. *kawałek*)" [13, str. 2]. Pliki podzielone są na bloki o stałej wielkości 64 MB i rozdzielone między serwery chunkserver. Serwer główny odpowiada za zarządzanie replikacją bloków (domyślnie każdy blok jest przechowywany w trzech kopiach) oraz za obsługę metadanych. „Klient łączy się z serwerem głównym w celu uzyskanie informacji o plikach, jednak dane pobiera i wysyła komunikując się bezpośrednio z serwerami dyskowymi" [13, str. 2].

Hadoop Distributed File System (HDFS) jest otwartą implementacją systemu GFS stworzoną przez Apache. Powstał w ramach większego projektu Apache Hadoop[1]. Nie obsługuje standardu POSIX, dostępne są jednakże biblioteki dla języka Java (a także dla języków C++, Python, PHP, Ruby, i innych).

Rozdział 2

System UNICORE

UNICORE (Uniform Interface to Computing Resources) to proste w instalacji i użyciu, bezpieczne oprogramowanie warstwy pośredniej. Powstało w ramach dwóch projektów finansowanych przez niemieckie Ministerstwo Edukacji i Rozwoju[1], następnie było rozwijane w trakcie kilku projektów europejskich, w 2004 roku zostało udostępnione jako otwarte oprogramowanie. Aktualnie (2011) rozwojem UNICORE zajmuje się European Middleware Initiative. UNICORE używany jest między innymi w DEISA[2], National German Supercomputing Center NIC, Gauss Center for Supercomputing, a także w ramach polskiego projektu PL-Grid i europejskiej European Grid Infrastructure.

Założeniem UNICORE było oparcie o otwarte standardy, w szczególności Open Grid Services Architecture (OGSA) i Web Services Resource Framework (WSRF). Architektura tego oprogramowania bazuje na usługach sieciowych (web services), co umożliwia łatwą rozbudowę i wymianę komponentów. Jego zaletą jest fakt, że dostarcza spójne i kompletne środowisko — począwszy od interfejsu użytkownika aż do obsługi zasobów. UNICORE miał być prosty w instalacji i utrzymaniu. Stworzony został w Javie, w oparciu o kontener serwisów WSRFLite[3] [31].

2.1. Architektura systemu UNICORE

Podstawową jednostką architektury systemu gridowego opartego o UNICORE jest UNICORE Site (USite). Jest to zbiór systemów docelowych (tzn. takich, na których ostatecznie wykonywane są obliczenia) w ramach jednej domeny administracyjnej. Natomiast pojedynczy system docelowy to UNICORE Virtual Site (VSite).

Wewnętrzna budowa systemu UNICORE składa się z trzech warstw: klienta, usług i systemu. Pierwsza z nich to oprogramowanie klienckie, takie jak *UNICORE Commandline Client* UCC (klient tekstowy) i *UNICORE Rich Client* (URC), oparty o środowisko Eclipse. Warstwa usług pośredniczy między klientem a docelowymi aplikacjami, uruchamianymi w warstwie systemu. Oto lista najważniejszych usług [32]:

[1] Bundesministerium für Bildung und Forschung

[2] The Distributed European Infrastructure for Supercomputing Applications

[3] `http://www.unicore.eu/documentation/manuals/unicore6/wsrflite/`

- **Gateway** — zewnętrzny punkt dostępu do całego USite. Pośredniczy w przekazywaniu żądań do pozostałych komponentów.
- **XNJS** — serce systemu UNICORE, przyjmuje i wykonuje zadania, obsługuje przechowywanie i przesyłanie plików, itd.
- **XUUDB** — obsługuję autoryzację, przechowuje listę użytkowników identyfikowanych przez certyfikaty oraz ich role i loginy,
- **UVOS** — nowsza alternatywa dla XUUDB,
- **Registry** — przechowuje listę publicznie dostępnych usług. Każdy VSite posiada lokalny rejestr uruchomionych usług. Oprócz tego w ramach USite istnieje rejestr globalny, który łączy wpisy ze wszystkich rejestrów lokalnych. Posiadając adres rejestru globalnego, użytkownik jest w stanie odnaleźć wszystkie publicznie dostępne usługi.
- **CIS** — usługa informacyjna, zbierająca dane z dostępnych XNJS-ów i publikująca je w formacie XML lub tekstowym,
- **Workflow Engine** — przyjmuje kaskady zadań. Zlecenie poszczególnych zadań zleca usłudze **Service Orchestrator**.

XNJS wykorzystuje **IDB**, czyli bazę danych, zawierającą informacje o tym w jaki sposób abstrakcyjne zadania zapisane w formacie JSDL[4] mają być przetworzone na konkretne pliki wykonywalne na systemie docelowym[33]. Serwerem, w ramach którego uruchomiony jest XNJS i rejestr lokalny, jest UNICORE/X. Serwer ten musi być zainstalowanym na każdym systemie docelowym. Gateway, Workflow Engine i rejestr globalny uruchamiane są jako oddzielne serwery.

Głównym komponentem warstwy systemu jest Target System Interface (TSI), napisany w języku Perl. Wykonuje on na systemie VSite polecenia dotyczące uruchomienia aplikacji lub operacji na plikach. TSI nie musi być uruchomione na tej samej maszynie co UNICORE/X. Obsługuje kilkanaście systemów kolejkowych, do których może skierować zadania — może jednak pracować też bez żadnego systemu kolejkowego. Prostszą alternatywą dla TSI jest Java TSI. Zadania uruchamiane są wówczas na tej samej maszynie i z tymi samymi uprawnieniami co UNICORE/X, bez użycia systemu kolejkowego.

Usługą pośredniczącą między użytkownikiem a TSI jest Target System Service. Usługa ta dostarcza informacji o zainstalowanych aplikacjach, o zasobach dyskowych z których korzystają zadania i o samych zadaniach. To TSS jest również odpowiedzialna za przyjmowanie nowych zadań, które przesyłane są następnie do XNJS i stamtąd do TSI.

Aby wykonać zadanie na systemie gridowym opartym o UNICORE, użytkownik przy użyciu klienta takiego jak URC podaje adres HTTP bramy (Gateway) i za jej pomocą łączy się z rejestrem. Uwierzytelnienie przebiega dzięki certyfikatom i usłudze UVOS lub XUUDB. Następnie wybiera usługę TSS, której przesyła zadanie. Zadanie jest przekazywane do XNJS, a następnie — w formie gotowej do wykonania — do TSI. Użytkownik czeka na jego zakończenie zadania i pobiera wyniki. Może też stworzyć kaskadę zadań (*workflow*), którą prześle na jeden ze znajdujących się w rejestrze Workflow Engine.

[4] oparty o XML język opisu zadań, opracowany przez Global Grid Forum[14]

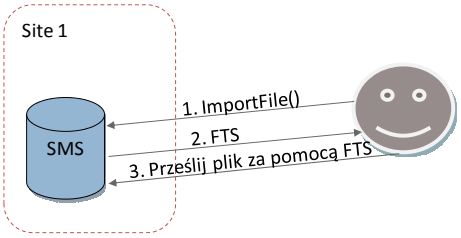
Uwierzytelnienie oparte jest o standard X.509. Każdy użytkownik (ale też i każda usługa) identyfikuje się certyfikatem, zawierającym podstawowe dane na jego temat (nazwa, e-mail, miasto, organizacja, itp.) — jest to tak zwana nazwa wyróżniająca (ang. *Distinguished Name*, DN). Oprócz tych danych certyfikat zawiera klucz publiczny, a całość podpisana jest przez urząd certyfikacji (ang. *Certificate Authority*, CA). Wszyscy użytkownicy i wszystkie usługi w ramach systemu gridowego muszą określić listę akceptowanych urzędów certyfikacji. Dzięki temu tak użytkownik jak i usługa mają pewność co do tożsamości drugiej strony połączenia.

Usługa XUUDB lub UVOS pozwala na mapowanie użytkowników gridu (identyfikowanych przez certyfikat lub nazwę DN) na użytkowników systemów docelowych (identyfikowanych przez login). Każdy użytkownik gridu ma również przypisaną „rolę", która pozwala określić, czy dany podmiot ma prawo do wykonywania określonych czynności na danych zasobach. Tego typu polityka bezpieczeństwa określona jest za pomocą standardu XACML opartego o pliki XML. Jedną z zalet usługi UVOS w stosunku do XUUDB jest możliwość tworzenia tzw. wirtualnych organizacji (ang. *virtual organization*, VO). Wirtualna organizacja jest zdefiniowana w [11, str. 2] jako grupa osób lub instytucji które dzielą w ramach gridu zasoby na tych samych warunkach. Istnieje również znacznie prostszy mechanizm autoryzacji, niewymagający uruchamiania oddzielnej usługi — jest to UUDB. Pozwala on mapować użytkowników wykorzystując plik tekstowy w formacie XML.

Zdarzyć się może sytuacja, w której użytkownik łączy się z jakąś usługą, a ta z kolei w celu wykonania żądania musi połączyć się z inną usługą. Przykład takiej sytuacji to wykorzystanie systemu Workflow Service. Aby zapisać wyniki pojedynczego zadania należącego do kaskady, system ten musi połączyć się z globalnym zasobem SMS. Pojawia się pytanie: jaka tożsamość powinna być użyta do uwierzytelnienia usługi Workflow Service podczas połączenia z owym zasobem SMS? To system Workflow Service wykonuje połączenie, ale działa on niejako „w imieniu" początkowego użytkownika. Zastosowanie znajduje tutaj mechanizm bezpośrednich delegacji zaufania (ang. *Explicit Trust Delegation* — ETD). Użytkownik łącząc się z dowolną usługą może jej wydać (i domyślnie wydaje) cyfrowo podpisane upoważnienie, które umożliwia jej pracę w jego imieniu. Usługa wykonując kolejne żądania, może dołączyć otrzymane upoważnienie do własnych danych uwierzytelniających. Dzięki temu usługa zdalna zna tożsamość usługi pośredniczącej oraz wie, na czyje konto ma wykonać daną operację.

2.2. SMS jako usługa odpowiedzialna za dostęp do zasobów dyskowych

Usługą odpowiedzialną za operacje na plikach jest *Storage Management Service* (SMS). Usługa ta, działająca jako web service w ramach UNICORE/X, udostępnia szereg metod takich jak `ListDirectory`, `ImportFile`, `Rename`, itd. W przypadku, gdy klient wywoła operację, która wymaga transferu plików, tworzony jest zasób o nazwie *File Transfer Service* (FTS). Adres zasobu FTS (*Endpoint Reference*) jest przekazywany użytkownikowi, który może kontrolować przebieg transferu. FTS tworzy instancję usługi zwanej StorageAdapter, która (w domyślnej swojej postaci) za pośrednictwem TSI dokonuje operacji na



Rysunek 2.1: Schemat działania usługi SMS na przykładzie operacji `ImportFile`

plikach.

Rysunek 2.1 przedstawia działanie usługi SMS na przykładzie operacji `ImportFile`. W pierwszym kroku klient łączy się z zasobem SMS, wywołując wspomnianą metodę. W rezultacie (2), usługa zwraca adres utworzonego zasobu FTS. Klient nawiązuje połączenie z zasobem FTS i przesyła plik (3).

Dane przesyłane są za pomocą jednego z kilku obsługiwanych protokołów: *Baseline File Transfer* (BFT), *Random access ByteIO file transfer* (RBIO), *Streamable ByteIO file transfer* (SBIO) lub UFTP. Pierwszy z nich oparty jest o HTTP. Dwa kolejne przesyłają dane w wiadomościach SOAP. Konieczność serializacji binarnych danych sprawia, że oba są mało wydajne. SBIO umożliwia dodatkowo korzystanie z operacji `seek` na otwartym pliku. UFTP, obecny w UNICORE/X od wersji 6.4, to wydajny protokół równoległej transmisji danych, oparty o Java Parallel Secure Stream [17]. Protokół ten wymaga jednak instalacji dodatkowego serwera UNICORE UFTP.

Oto lista wszystkich operacji interfejsu SMS:

- `ChangePermissions` — zmiana uprawnień do pliku,
- `Copy` — skopiowanie pliku,
- `CreateDirectory` — utworzenie nowego katalogu,
- `Delete` — usunięcie pliku lub katalogu,
- `ExportFile` — eksport (pobranie) pliku o podanej ścieżce,
- `Find` — wyszukanie plików na podstawie przekazanych parametrów,
- `ImportFile` — przesłanie nowegu pliku do wybranego katalogu,
- `ListDirectory` — pobranie listy plików w danym katalogu,
- `ListProperties` — pobranie właściwości pliku, takich jak wielkość, uprawnienia, data ostatniej modyfikacji, itd.,
- `ReceiveFile` — dodanie nowego pliku, znajdującego się na zdalnym zasobie SMS,
- `Rename` — zmiana nazwy pliku lub katalogu,
- `SendFile` — przesłanie pliku na zdalny zasób SMS.

Operacje `ReceiveFile` i `SendFile` pozwalają na wykonanie transferu bezpośrednio między dwoma zasobami SMS.

Mechanizm SMS jest dość elastyczny. Istnieje możliwość tworzenia własnych implementacji, ich instalacja sprowadza się do dodania kilku linii w pliku konfiguracyjnym `uas.conf`. Stworzenie własnego protokołu transferu plików również nie stanowi problemu. W domyślnej instalacji znajduje się kilka implementacji SMS:

- **FixedStorage** — przechowuje pliki w wybranym katalogu,
- **HomeStorage** — używa domowego katalogu użytkownika,
- **PathedStorage** — ścieżka jest ustalana podczas żądania na podstawie zmiennych wprowadzonych w konfiguracji.

Z usługą SMS powiązana jest usługa *Storage Factory* (SF). Pozwala ona na utworzenie własnych, tymczasowych zasobów SMS. Gdy zasób przestanie być potrzebny, można go łatwo usunąć wraz z całą zawartość. Aby korzystać z SF, należy wcześniej skonfigurować rodzaje zasobów SMS, które będzie można utworzyć (wskazując implementację i konfigurację Storage Management). Klient URC pozwala jedynie na wyświetlenie utworzonych zasobów SMS, klient UCC pozwala również tworzyć nowe.

Zwykłe zasoby SMS powiązane są z instancjami usług Target System Service. Zasoby te nie są dodawane do rejestru, niemniej jednak można uzyskać do nich dostęp za pomocą TSS. Domyślna implementacja to HomeStorage, istnieje też możliwość użycia dowolnych innych. Podczas uruchomienia serwera UNICORE/X istnieje też możliwość utworzenia zasobu SMS, niepowiązanego z żadnym TSS, ale obecnego w rejestrze — zasób taki nazywamy globalnym SMS.

W najnowszej (6.4) wersji UNICORE została dodana obsługa metadanych[5]. Metadane (w postaci par: klucz-wartość) przechowywane są w plikach XML, umieszczonych na zasobie SMS (pliki te wyróżniają się rozszerzeniem `.metadata`). Z każdym zasobem SMS skojarzona jest usługa *MetadataManagement*, która umożliwia dostęp do metadanych. Fizycznym zapisem metadanych do odpowiednich plików zajmuje się *MetadataManager*.

2.3. Usługi korzystające z SMS

Warto przyjrzeć się bliżej komponentom UNICORE, które korzystają ze Storage Management. Przede wszystkim są to aplikacje klienckie — tak UCC jak i URC posiadają mechanizmy umożliwiające listowanie katalogów, operacje na plikach (zmiany nazwy, kopiowanie), a także import i eksport. Potrafią też wywołać operację `Destroy`, która powoduje usunięcie całego zasobu SMS. Klient UCC potrafi też tworzyć zasoby SMS używając StorageFactory.

Usługi SMS wymaga również Service Orchestrator. Ponieważ każde zadanie w kaskadzie może być uruchomione na innym TSS, a jednocześnie jedno zadanie może przyjmować jako wejście wynik działania poprzedniego, niezbędna jest przestrzeń dyskowa, w której wszystkie pliki tymczasowe i wynikowe zostaną zapisane. Użyty może zostać globalny SMS znajdujący się w rejestrze lub Storage Factory — w takim przypadku to usługa Workflow utworzy tymczasowy

[5] http://sourceforge.net/mailarchive/message.php?msg_id=27022694

zasób SMS. Zaletą takiego rozwiązania jest to, że po wykonaniu kaskady zadań i pobraniu interesujących nas wyników, można usunąć cały zasób SMS i w ten sposób nie trzeba martwić się o powstałe tymczasowe pliki.

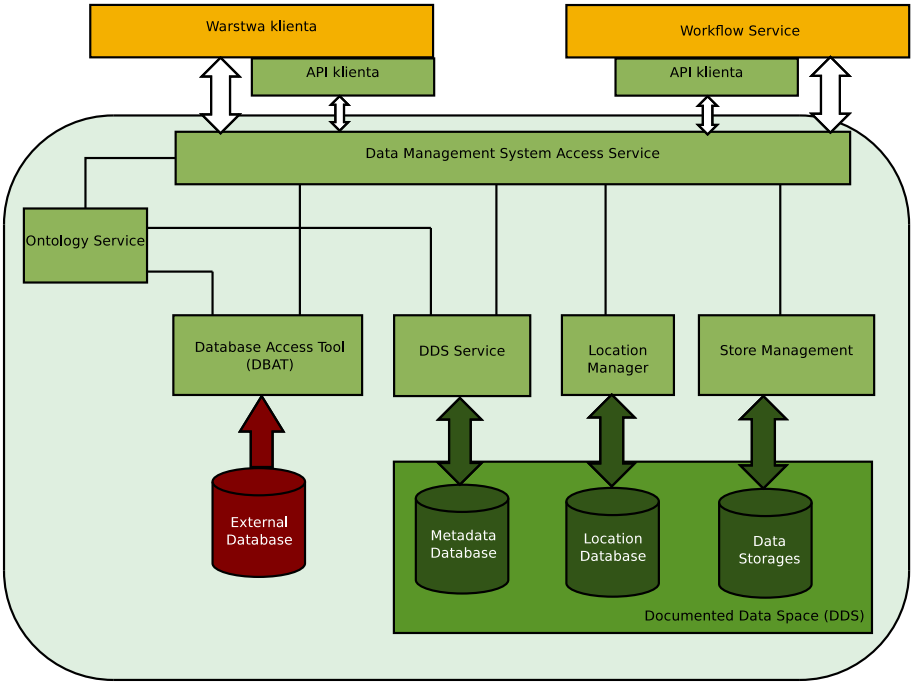
2.4. Istniejące rozwiązania rozproszonego przechowywania danych w systemie UNICORE/X

Jak wynika z poprzednich sekcji, podstawowe implementacje SMS nie obsługą rozproszonego przechowywania danych — wykorzystują po prostu wybrany katalog w systemie plików. Powstało jednak kilka zewnętrznych rozwiązań umożliwiających ominięcie tego problemu — poniżej znajduje się opis trzech z nich.

2.4.1. Chemomentum DMS

Chemomentum (C9m) to zakończony już europejski projekt, który miał na celu wykorzystanie systemu UNICORE w obliczeniach chemicznych i pokrewnych. W jego ramach powstał Chemomentum Data Management System, rozszerzający mechanizmy przechowywania danych w UNICORE o rozproszenie i obsługę metadanych. Oprócz tego C9m DMS zapewnia możliwość dostępu do zewnętrznych źródeł danych (takich jak pliki XLS lub nawet odpowiednio sformatowany HTML) za pomocą *Database Access Tool*. Narzędzie to powstało głównie z myślą o specjalistycznych, biochemicznych bazach danych, np. białek. Natomiast *Ontology Service* pozwala na przeliczanie jednostek i nazw związków, co może być przydatne podczas przeszukiwania metadanych i zewnętrznych baz — gdyż dzięki tej usłudze nie ma znacznia, czy wpisujemy np. H_2O czy „water".

Z punktu widzenia rozproszonego przechowywania danych najbardziej interesujące są jednak inne komponenty C9m DMS. Centralną usługą jest *Data Management System Access Service* (DMSAS). Pośredniczy ona między żądaniami klientów a modułami odpowiedzialnymi za wykonanie wywołanych operacji, zwraca również wyniki. DMSAS nie implementuje usługi SMS, dlatego też klienci muszą łączyć się przy użyciu specyficznego protokołu C9m DMS. Dostępne są wtyczki do standardowych klientów UNICORE, umożliwiające połączenie z DMSAS [34, str. 8]. Usługa DMSAS jest połączona z dwiema opisanymi w poprzednim akapicie usługami oraz z trzema usługami odpowiedzialnymi za przechowywanie plików. Są to: *Storage Management*, *Location Manager* i *Documented Data Space Service* (DDS Service). Pierwsza z nich obsługuje zasoby SMS, na których fizycznie umieszczane są pliki. Istnieje możliwość dodania w konfiguracji wielu SMS-ów. Wówczas usługa Location Manager, odpowiedzialna za mapowanie nazw logicznych na fizyczne, będzie wybierać zasoby SMS korzystając z algorytmu round-robin. Location Manager przechowuje informacje o plikach w bazie MySQL. DDS Service obsługuje bazę metadanych. Baza metadanych, baza lokalizacji plików i zasoby SMS tworzą razem tzw. Documented Data Space (DDS). DDS Service może współpracować z usługą Extractor Service. Usługa ta pozwala na tworzenie prostych skryptów w języku Python, których zadaniem jest analiza przesyłanych do DDS plików i automatyczne tworzenie metadanych, np. miniaturki zdjęć na podstawie plików JPG. Skrypty te mogą być przypisane do określonego (na podstawie typu MIME) rodzaju plików [34, str. 16].



Rysunek 2.2: Budowa C9m DMS, za: [26]

C9m DMS autoryzuje wszystkie żądania dotyczące danych i metadanych na podstawie nazwy DN użytkownika. Uwierzytelnienie przebiega za pośrednictwem mechanizmu UUDB, usługi XUUDB lub UVOS. Jedynie właściciel plików może kasować i modyfikować dane, natomiast całą „grupa" (określona za pomocą atrybutu `role`) może czytać metadane. Użytkownik o roli `admin` ma prawo do wszelkich modyfikacji DDS [34, str. 18].

2.4.2. UniRODS

UniRODS to implementacja usługi SMS oparta o omówiony w pierwszym rozdziale system iRODS. Powstała na UMK w ramach prac magisterskich [7] i [16]. Do komunikacji z iRODS wykorzystano bibliotekę *jargon*, będącą implementacją interfejsu programistycznego iRODS w języku Java. Możliwe jest wykorzystanie UniRODS jako globalnego zasobu SMS (na przykład w celu użycia przez system Workflow Service).

W pierwszej wersji [7] rozszerzenie UniRODS wykorzystywało własną usługę transferu plików. W przeciwieństwie do domyślnej implementacji FTS, usługa ta nie używa komponentu `StorageAdapter`, pozwalającego na dostęp do plików na maszynie docelowej, lecz łączy się bezpośrednio z systemem iRODS. Jedyny dostępny w tym przypadku protokół transferu plików to RBIO. W poprawionej wersji [16] UniRODS nie modyfikuje usługi FTS, lecz tworzy własną implementację `StorageAdapter`, która używana jest do połączenia z iRODS. Dzięki temu rozwiązaniu do transferu plików można używać dowolnego z dostępnych protokołów.

Ponieważ uwierzytelnienie w iRODS jest niekompatybilne z mechanizmem

bezpieczeństwa stosowanym w systemie UNICORE, niezbędne jest mapowanie użytkowników systemu gridowego na użytkowników iRODS — konfiguracja tego mapowania zapisana jest w pliku XML. Plik składa się z rekordów, każdy rekord zawiera nazwę DN użytkownika w systemie UNICORE oraz login i hasło w systemie iRODS.

2.4.3. UniHadoop

UniHadoop jest w swojej koncepcji rozwiązaniem podobnym do UniRODS. Również tutaj korzystamy z istniejącego, rozproszonego systemu przechowywania danych (w tym przypadku jest to, opisany w sekcji 1.10., HDFS). Utworzono w tym celu własną implementację interfejsu `StorageAdapter`, wykorzystując bibliotekę Apache Hadoop, aby umożliwić dostęp do docelowego systemu plików. Kontrola dostępu do plików przechowywanych w ramach HDFS opiera się o login właściciela oraz uprawnienia (do odczytu, zapisu i wykonania). Tak więc mechanizmy bezpieczeństwa są analogiczne do tych, które stosowane są w przypadku TSI i nie są tu potrzebne dodatkowe mechanizmy mapujące (jak ma to miejsce w przypadku UniRODS).

Rozdział 3

Cel pracy

3.1. Analiza istniejących rozwiązań

Jak powiedziano w poprzednim rozdziale, domyślna dystrybucja UNICORE nie posiada mechanizmów rozproszonego przechowywania danych. Mimo że opisano dwie usługi, niestety nie wydaje się, aby były one optymalne. Analizując ich wady i zalety można jednakże utworzyć listę cech, które musi posiadać pożądane rozwiązanie.

System Chemomentum DMS powstał głównie na potrzeby obliczeń biochemicznych. Wiąże się to z istnieniem szeregu dodatkowych usług, niezwiązanych bezpośrednio z przechowywaniem danych, takich jak Database Access Tool lub Ontology Service. Mimo teoretycznie modularnej budowy C9m DMS nie da się niestety w prosty sposób zrezygnować z ich wykorzystania. Sprawiają one, że cały projekt jest trudniejszy w utrzymaniu — zarówno od strony programistycznej jak i administracyjnej. Kolejnym problemem jest istnienie tzw. wąskiego gardła — jest to usługa Data Management System Access Service, przez którą przechodzą wszystkie żądania klientów, ale też i wszystkie transferowane pliki. Łatwo sobie wyobrazić, że w przypadku znacznej ilości plików o wielkościach rzędu gigabajtów, usługa DSAS może znacznie spowolnić wszystkie wykonywane operacje. Znaczną wadą jest również użycie własnego interfejsu dostępu do danych przechowywanych w ramach DMS. W związku z tym zarówno aplikacja kliencka jak i system Workflow Service należy rozszerzyć o odpowiednie wtyczki, umożliwiające dostęp do DMS.

Zaletą C9m DMS jest możliwość wykorzystania zasobów SMS jako docelowych przestrzeni, w których zostaną umieszczone pliki. Dzięki temu dowolny serwer z zainstalowaną usługą UNICORE/X może stać się serwerem dyskowym — bez konieczności instalowania dodatkowego oprogramowania, którego utrzymanie byłoby dodatkowym problemem.

Dużą wadą pierwszej wersji UniRODS był protokół używany do przesyłania plików — jak twierdzi autorka: „trudno bowiem wyobrazić sobie przesyłanie kilkusetgigabajtowego pliku z użyciem technologii SOAP i efektywnym transferem rzędu 400 kBs" [7, str. 59]. Druga wersja pozwoliła wyeliminować tę słabość. Zaletą UniRODS (w porównaniu do C9m DMS) jest wykorzystanie usługi SMS jako interfejsu dostępowego. Dzięki temu, aby skorzystać z UniRODS klient nie potrzebuje żadnych niestandardowych narzędzi. Warto zwrócić uwagę na funda-

mentalne założenie UniRODS, którym jest wykorzystanie oddzielnego systemu rozproszonego przechowywania danych i utworzenie implementacji SMS, która będzie potrafiła z niego korzystać. Rozwiązanie to posiada wiele zalet: system iRODS jest zoptymalizowany, sprawdzony i łatwo się skaluje (może obsługiwać zasoby z ponad milionem plików) [7, str. 39]. Wadą jest jednak fakt, że na wszystkich serwerach dyskowych należy zainstalować dodatkową usługę, którą trzeba skonfigurować, aktualizować, itd. Kolejną wadą jest brak kompatybilności mechanizmów bezpieczeństwa i konieczność ręcznego tworzenia mapowań użytkowników — „niezbędne wydaje się być przeniesienie mapowań użytkowników z pliku konfiguracyjnego do serwera VO" [7, str. 60].

Powyższe uwagi w dużej mierze można odnieść również do UniHadoop. Niekompatybilne mechanizmy autoryzacji nie stanowią tu jednak problemu, gdyż stosowana jest standardowa metoda mapowania nazwy DN użytkownika na jego login (który następnie jest używany w HDFS). Niestety, uwaga dotycząca konieczności instalacji i utrzymania dodatkowego oprogramowania na serwerach dyskowych pozostaje aktualna. Poza tym przy rozwiązaniach tego typu (tzn. korzystających z zewnętrznego oprogramowania, takiego jak iRODS lub Apache Hadoop) nie można zagwarantować, że wewnętrzne transfery plików są realizowane w sposób bezpieczny i szyfrowany.

3.2. Lista wymagań

Na podstawie powyższej analizy przedstawionych narzędzi można utworzyć następującą listę wymagań dotyczących rozproszonego systemu przechowywania danych dla UNICORE:

- usługa łącząca wiele przestrzeni dyskowych w jedną logiczną całość,
- owa logiczna całość ma być dostępna za pomocą standardowej usługi SMS (tak jak w UniRODS i UniHadoop),
- serwery dyskowe udostępniają swoją przestrzeń również za pomocą usługi SMS (tak jak w C9m DMS),
- brak centralnego komponentu, przez który muszą zostać przesłane wszystkie pliki,
- możliwość stosowania domyślnych mechanizmów uwierzytelnienia i autoryzacji,
- obsługa wszystkich protokołów transferu plików dostępnych w usłudze SMS,
- prostota architektury i łatwość utrzymania, dzięki czemu rozwiązanie mogłoby być dodane do standardowej dystrybucji UNICORE.

Realizacja powyższych wymagań jest celem niniejszej pracy.

Rozdział 4

Usługa dSMS

4.1. Podstawowe informacje

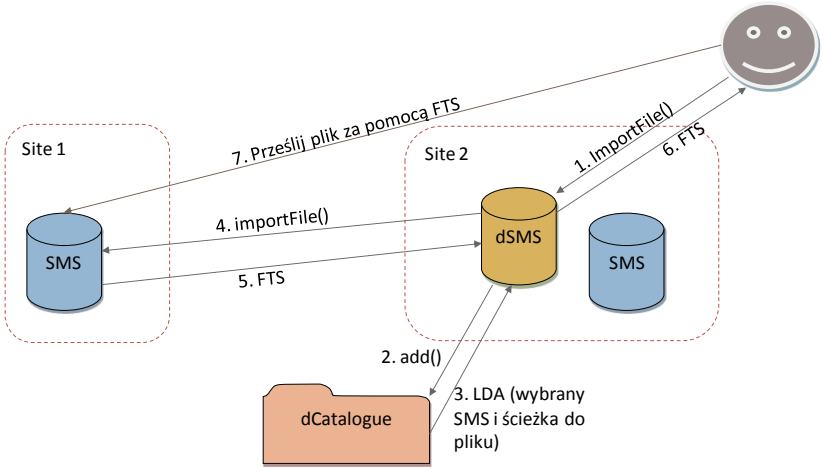
W poprzednim rozdziale przedstawiono założenia poszukiwanego rozwiązania rozproszonego przechowywania danych. W ramach niniejszej pracy i w oparciu o powyższe założenia utworzono system nazwany (od usługi dostępowej) dSMS. System wykorzystuje kilka elementów:

- dane przechowywane są na istniejących **globalnych zasobach SMS**,
- informacje o lokalizacji plików i struktura katalogów znajduje się w usłudze katalogu zwanej **dCatalogue**,
- klient łączy się z implementacją usługi Storage Management o nazwie **dSMS**.

Wykorzystana może być dowolna ilość obecnych w rejestrze zasobów SMS. Ilość punktów dostępowych, czyli usług dSMS jest również nieograniczona. dCatalogue jest usługą centralną, nie podlegającą replikacji. dSMS oprócz udostępnienia klientowi interfejsu Storage Management łączy się również z dCatalogue i docelowymi zasobami SMS.

Rysunki 4.1, 4.2 i 4.3 przedstawiają działanie systemu dSMS na przykładzie trzech operacji wywołanych przez klienta.

Operacja `ImportFile` pozwala na dodanie do zasobu SMS nowego pliku. W kroku 1. klient wywołuje tę operację na zasobie dSMS. W 2. kroku dSMS łączy się z usługą dCatalogue, aby zarejestrować nowy plik. Usługa katalogu oprócz zarejestrowania pliku w bazie danych, wybiera zasób SMS na którym zostanie umieszczony plik. Następnie informuje usługę dSMS o tym wyborze, przesyłając w kroku 3. adres docelowego zasobu Storage Management i ścieżkę. Te dwie informacje tworzą tzw. adres zależny od lokalizacji (ang. *Location-dependent address*, LDA). Usługa dSMS posiadając adres LDA może połączyć się z wybranym zasobem SMS, również wywołując operację `ImportFile`, podając jednak ścieżkę uzyskaną z dCatalogue (krok 4). W kroku 5 zasób SMS zwraca adres Endpoint Reference usługi File Transfer Service, umożliwiającej przesłanie nowego pliku. Przedostatnim, 6. krokiem jest przekazanie niezmienionego adresu EPR klientowi. Posiadając adres usługi FTS, klient łączy się z usługą SMS i bezpośrednio do niej przesyła treść pliku.

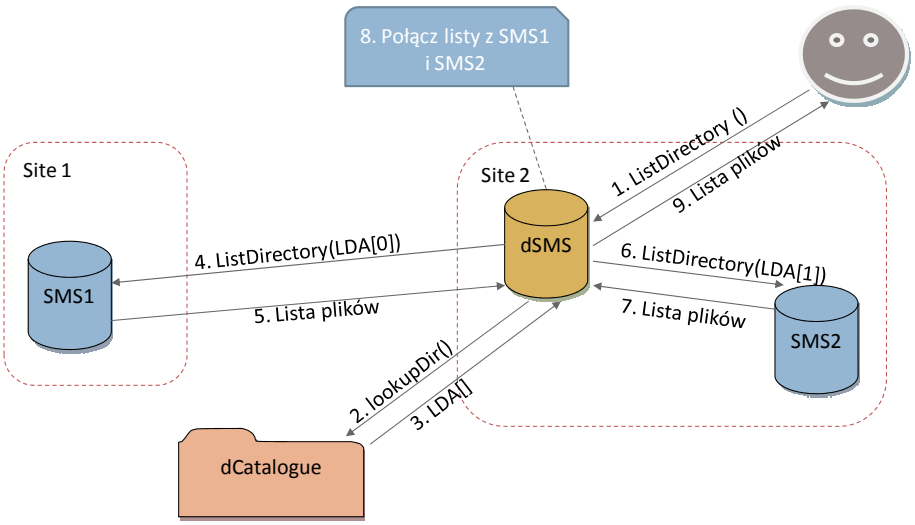


Rysunek 4.1: Schemat działania systemu dSMS na przykładzie operacji `ImportFile`

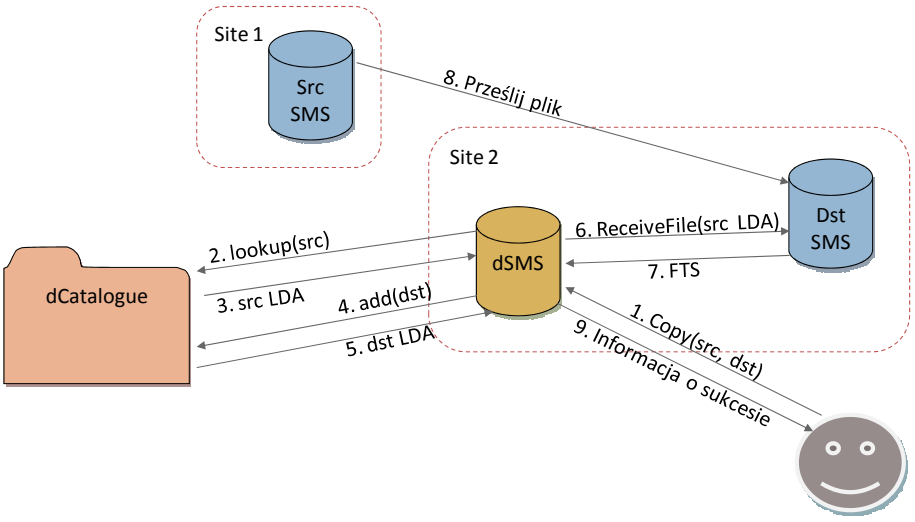
W porównaniu do tej samej operacji `ImportFile` wywołanej na zwykłym zasobie SMS (rysunek 2.1), przedstawione wyżej działania są bardziej skomplikowane. Zaangażowane są tu aż trzy usługi: dSMS, dCatalogue i docelowy SMS. Jednak z punktu widzenia użytkownika, sytuacja się nie zmieniła: wywołuje on metodę, w rezultacie dostaje zasób FTS, za pomocą którego przesyła plik. Założeniem prezentowanej usługi było zachowanie takiej kompatybilności w przypadku wszystkich zaimplementowanych metod, aby działania charakterystyczne dla systemu dSMS odbywały się w sposób niewidoczny dla klienta.

Z kolei operacja `ListDirectory`, której schemat działania przedstawia rysunek 4.2, zwraca listę plików znajdujących się w danym katalogu. Ponieważ w przypadku dSMS pliki takie mogą być rozproszone na różnych zasobach SMS, konieczne jest połączenie z każdym z nich. Operacji `ListDirectory` nie można przeprowadzić bazując jedynie na zawartości dCatalogue, gdyż oprócz nazw plików należy zwrócić użytkownikowi również metadane, takie jak wielkość i prawa dostępu. Gdy zasób dSMS otrzyma polecenie wykonania operacji (1), łączy się z dCatalogue wywołując metodę `lookupDir`. Metoda ta zwraca (3) listę wszystkich zasobów SMS, które przechowują pliki z danego katalogu. Następnie dSMS łączy się z każdym z tych zasobów, wywołując operację `ListDirectory` (4, 6). Zasoby SMS zwracają częściowe listy plików (5, 7), które dSMS łączy ze sobą (8) i zwraca klientowi (9).

Operacja `Copy`, przedstawiona na rysunku 4.3 kopiuje plik w ramach zasobu dSMS. W przypadku zasobu rozproszonego należy mieć na uwadze, że plik docelowy może być umieszczony na innym zasobie niż plik źródłowy i dlatego należy wykorzystać jedną z metod transferu plików między serwerami. Tu użyto `ReceiveFile` (6) wywołaną na docelowym zasobie SMS. Powoduje ona utworzenie zasobu FTS, który odpowiada za kontrolę transferu pliku. Zasób ten jest zwrócony do usługi dSMS (7), która kontroluje transfer i po jego zakończeniu zwraca użytkownikowi informację o sukcesie. Warto zauważyć, że usługa



Rysunek 4.2: Operacja `ListDirectory`



Rysunek 4.3: Operacja `Copy`

dSMS kontroluje transfer, ale w nim nie uczestniczy, tzn. plik przesyłany jest bezpośrednio z jednego zasobu SMS na drugi.

Pozostałe operacje to:

- `ChangePermissions`, `ListProperties`, `Delete`, `ExportFile` — działają w sposób analogiczny do `ImportFile`, choć tylko ostatnia z nich tworzy i przekazuje klientowi zasób FTS.
- `Find` — działa podobnie do `ListDirectory`,
- `Rename` — nazwa pliku jest zmieniana najpierw w dCatalogue a następnie na docelowym zasobie SMS,
- `SendFile`, `ReceiveFile` — operacje te dotyczą transferu między serwerami i realizowane są bezpośrednio między używanymi zasobami SMS.

Wszystkie powyższe metody są szczegółowo opisane na diagramach interakcji języka UML, w dodatku B.

4.2. Wykorzystanie Storage Factory

W rodziale 2. opisana została usługa Storage Factory, pozwalająca na utworzenie „prywatnych" zasobów SMS, tzn. takich, które są nieobecne w rejestrze a ich właścicielem jest tworzący je użytkownik. Zasoby takie wykorzystywane są głównie w kaskadach zadań wykonywanych przez system Workflow Service. Usługa Storage Factory może być skonfigurowana tak, aby tworzyła również zasoby dSMS. Pliki z utworzonych w ten sposób zasobów dSMS również rozproszone są w ramach gridu, a dCatalogue tworzy dla takich plików oddzielną przestrzeń nazw. Oprócz tego możliwe jest wykorzystanie usługi Storage Factory w sposób niejako symetryczny. Prywatny (stworzony przez Storage Factory) zasób dSMS potrafi utworzyć (również za pomocą fabryki) nowy zasób SMS i w nim przechowywać pliki. Można to podsumować w tabeli:

	Wykorzystywane zasoby docelowe	
Rodzaj zasobu dSMS	Globalny SMS	Storage Factory
Globalny dSMS	tak	nie
Prywatny dSMS	tak	tak

Tak więc globalny zasób dSMS wykorzystuje jedynie globalne zasoby SMS, tymczasem prywatne dSMS utworzone przez Storage Factory potrafią wykorzystać także „fizyczne" usługi Storage Factory. O tym, czy prywatny dSMS wykorzysta istniejący, globalny zasób SMS, czy też utworzy nowy za pomocą fabryki, decyduje algorytm wyboru zasobu, omówiony w sekcji 4.6. Jeśli algorytm wybierze usługę Storage Factory, na której już utworzono SMS dla danego prywatnego dSMS, wówczas nie jest tworzony kolejny zasób SMS, lecz wykorzystywany jest istniejący. W przypadku usunięcia prywatnego dSMS usuwane są także wszystkie stworzone na jego potrzeby prywatne zasoby SMS oraz odpowiednie katalogi w globalnym SMS.

4.3. Usługa dCatalogue

Usługa dCatalogue udostępnia interfejs web service pozwalający na dostęp do danych dotyczących systemu dSMS zapisanych w bazie MySQL lub H2. Dane przechowywane w bazie to:

- lista utworzonych zasobów dSMS, wraz z unikalnym identyfikatorem i nazwą DN właściciela (zapisaną w celu autoryzacji),
- lista zasobów SMS utworzonych za pomocą Storage Factory na potrzeby prywatnych dSMS. Konieczne jest przechowywanie tej listy, gdyż w sytuacji usunięcia prywatnego dSMS należy również usunąć te zasoby,
- informacje o strukturze plików i katalogów znajdujących się na wszystkich zasobach dSMS. Również tutaj obecna jest nazwa DN właściciela każdego pliku,
- informacje o tym, na jakich zasobach SMS znajdują się powyższe pliki i katalogi.

Nie są przechowywane metadane na temat plików, takie jak wielkość czy czas modyfikacji (są one każdorazowo pobierane z docelowych zasobów SMS). Poza tym dCatalogue zawiera właściwie wszystkie informacje na temat danych zawartych w zasobach dSMS. Dzięki temu możliwa jest prosta replikacja dostępowej usługi dSMS, która jest bezstanowa, a wszystkie informacje czerpie właśnie z bazy dCatalogue.

Dlaczego więc opisana baza nie zawiera pozostałych metadanych? Z pewnością znacznie uprościłoby to na przykład operację `ListDirectory` — nie byłoby konieczności łączenia się ze wszystkimi używanymi zasobami SMS. Dzieje się tak, gdyż metadane te można pobrać dopiero po zakończeniu importu nowego pliku, tymczasem usługa dSMS „nie wie" kiedy ów transfer się kończy. Całkowita kontrola nad transmisją pliku jest bowiem przekazywana (w zasobie FTS) użytkownikowi, a ten nie informuje usługi dSMS o zakończeniu transmisji.

Problem ten mogłoby rozwiązać utworzenie własnej implementacji usługi FTS, która pozwalałaby kontrolować przebieg transmisji i zapisywać w usłudze dCatalogue także pozostałe metadane. Jednak takie rozwiązanie uczyniłoby usługę dSMS trudniejszą w utrzymaniu od strony programistycznej i mniej funkcjonalną. Przykładowo — do UNICORE/X w wersji 6.4 dodano nowy, szybki protokół transferu plików o nazwie UFTP. Dzięki wykorzystaniu standardowej implementacji FTS, protokół ten może być wykorzystany w usłudze dSMS bez żadnych dodatkowych modyfikacji (mimo że w trakcie powstawania usługi protokół ten jeszcze nie istniał). Tymczasem używanie własnej wersji FTS sprawiłoby, że protokół UFTP można by użyć dopiero po dokonaniu odpowiednich zmian w kodzie dSMS.

4.4. Struktura plików

Pliki na zasobach SMS przechowywane są w ukrytym katalogu, aby uniknąć ich przypadkowej modyfikacji przez użytkownika. Pliki należące do globalnego zasobu dSMS znajdują się w katalogu `/.dsms`, natomiast pliki należące do zasobu dSMS utworzonego w Storage Factory umieszczane są w ukrytym katalogu,

którego nazwą jest wewnętrzny identyfikator zasobu. Nazwy plików są identyczne w ramach dSMS i SMS, również struktura katalogów pozostaje niezmieniona (choć oczywiście na każdym z zasobów SMS przechowywany jest jedynie fragment tej struktury).

4.5. Synchronizacja

Istotnym problemem jest synchronizacja między bazą danych dCatalogue a rzeczywistym stanem plików w zasobach SMS. Istnieją dwie możliwości utraty tej synchronizacji:

A) na zasobie SMS pojawi się plik, który nie istnieje w dCatalogue,

B) z zasobu SMS zostanie usunięty plik należący do dSMS.

Z kolei operacje w trakcie których powyższe problemy mogą się objawić można podzielić na 3 grupy:

1. operacje zwracające zawartość katalogu (metody `ListDirectory` i `Find` interfejsu SMS),
2. operacje dodające nowy plik (metody `ImportFile`, `ReceiveFile`, `Copy`, `Rename`),
3. operacje pobierające plik lub informacje o pliku (`ExportFile`, `SendFile`, `ListProperties`, `ChangePermissions`).

Operacje `ListDirectory` i `Find` zaimplementowane zostały w taki sposób, że dSMS pobiera listy plików ze wszystkich używanych zasobów SMS, scala je i odsyła klientowi. Tak więc operacje pierwszego typu zawsze odzwierciedlają aktualny stan zasobów SMS.

W przypadku próby dodania nowego pliku (operacja drugiego typu) problem synchronizacji *A)* może objawić się wtedy, gdy plik o danej nazwie istnieje na jakimś zasobie SMS (ale nie w dCatalogue) i użytkownik przesyłając ten plik ponownie, chce go nadpisać. W związku z tym przy operacji tego typu dSMS sprawdza, czy na żadnych z używanych zasobów SMS nie istnieje już plik o nazwie odpowiadającej dodawanemu plikowi. Jeśli tak, plik ten jest dodawany do dCatalogue i zostaje nadpisany przez nowy plik. Problem synchronizacji *B)* przy tego typu operacji właściwie nie występuje — nowy plik jest umieszczony w miejsce starego, usuniętego. Takie nadpisywanie jest zachowaniem standardowym dla UNICORE.

Pozostaje jeszcze operacja typu trzeciego, czyli odczyt informacji o pliku. Jeśli użytkownik chce pobrać plik który nie istnieje w dCatalogue, to przed zwróceniem wyjątku `FileNotFound` plik ten jest poszukiwany na wszystkich zasobach SMS. Jeśli zostanie odnaleziony, jest dodawany do bazy dCatalogue i zwracany użytkownikowi. W ten sposób rozwiązany jest problem *A)*. Jeśli plik istnieje w dCatalogue, to dSMS sprawdza również, czy rzeczywiście znajduje się na odpowiednim zasobie SMS. Jeśli nie, jest usuwany z dCatalogue a użytkownik otrzymuje wspomniany wyjątek.

Przedstawiony mechanizm synchronizacji może spowodować problemy z wydajnością — konieczność odpytywania wszystkich zasobów SMS przy każdej

operacji tworzącej nowy plik powoduje dość duże obciążenie systemu (zwłaszcza, jeśli używanych zasobów SMS jest wiele lub użytkownik generuje dużą ilość małych plików). W podobny sposób obciążające może być każdorazowe sprawdzanie, czy plik obecny w dCatalogue rzeczywiście znajduje się na zasobie SMS. Szczegółowa analiza wydajności znajduje się w rozdziale 5. Istnieje możliwość wyłączenia (za pomocą odpowiednich opcji konfiguracyjnych) każdego z powyżej omówionych mechanizmów synchronizacji.

Alternatywą dla powyższego sposobu zachowania synchronizacji mógłby być proces działający w tle, który w określonym czasie przeglądałby wszystkie zasoby i dokonywał ewentualnych zmian w bazie dCatalogue, aby zachować spójność danych. Niestety, pliki zachowane na zasobach SMS są standardowo dostępne tylko dla ich właścicieli, dlatego też usługa synchronizująca musiałaby albo działać z prawami administratora albo w jakiś sposób przechowywać delegacje zaufania użytkowników łączących się z usługą dSMS i używać tych delegacji podczas badania spójności systemu. Oba rozwiązania wydają się być mało bezpieczne (wiążą się z przekazaniem jednej usłudze znacznej ilości uprawnień), dlatego też ostatecznie zdecydowano się na omówiony w tej sekcji mechanizm synchronizacji.

4.6. Wybór zasobu

Jednym z zadań dCatalogue jest wybór zasobu SMS lub Storage Factory, na którym umieszczony zostanie nowy plik. Używany algorytm może zostać zmieniony w konfiguracji. Domyślnie używana jest algorytm round-robin, wskazujący po kolei wszystkie zasoby dostępne w rejestrze systemu gridowego. Dostępny jest również drugi prosty mechanizm, wybierający zasób o największej ilości wolnej przestrzeni. Dodanie własnych algorytmów nie jest trudne, interfejs `SmsFinder`, który należy zaimplementować, zawiera tylko jedną metodę. Jej argumentami są: nazwa dodawanego pliku oraz lista wszystkich znalezionych zasobów SMS (wraz z ilością wolnego miejsca na każdym z nich). Metoda powinna zwrócić adres wybranego zasobu.

4.7. Bezpieczeństwo

Proponowane rozwiązanie wykorzystuje standardowe dla UNICORE mechanizmy bezpieczeństwa, w szczególności omówiony w sekcji 2.1. mechanizm delegacji zaufania. Delegacje używane są, gdy użytkownik łączy się z jedną usługą, a ona z kolei łączy się z drugą w imieniu początkowego użytkownika. W tym przypadku usługa dSMS łączy się z docelowym zasobem SMS i z katalogiem właśnie w imieniu użytkownika, który wywołuje całą operację. dCatalogue przechowuje nazwy DN właścicieli plików i katalogów, tak więc jedynie właściciel ma możliwość odczytu i modyfikacji danych o plikach. Zasoby SMS również autoryzują dostęp do plików za pomocą opisanego w sekcji 2.1. mechanizmu mapowania nazw DN na nazwy użytkowników systemu docelowego. Dzięki wykorzystaniu delegacji zaufania autoryzacja odbywa się w identyczny sposób, jak gdyby użytkownik łączył się bezpośrednio z zasobem SMS. Usługa dCatalogue musi być autoryzowana do tego, aby wyszukiwać w rejestrze dostępne zasoby SMS i Storage Factory i odpytywać je o ilość wolnego miejsca.

Rozdział 5

Analiza rozwiązania

5.1. Testy funkcjonalne

Istotną częścią tworzenia systemu dSMS było sprawdzanie poprawności implementowanych metod za pomocą testów funkcjonalnych. Przygotowano je w oparciu o bibliotekę programistyczną JUnit w wersji 4.8. Przygotowano dwa oddzielne zestawy testów — dla usług dCatalogue oraz dSMS. Testy sprawdzają poprawność działania wszystkich metod udostępnianych przez interfejsy powyższych usług.

Metody testujące usługę dCatalogue działają według schematu, który przedstawię na przykładzie operacji `testAdd`. Przed wykonaniem zestawu testów tworzony jest kontener WSRFlite, natomiast przed wykonaniem każdego z testów inicjalizowana jest baza danych oraz tworzone są usługi dCatalogue i SMS. Następnie rozpoczyna się właściwy test: do bazy dCatalogue dodawany jest plik `/test`, po czym sprawdza się, czy plik rzeczywiście istnieje i czy nie jest katalogiem. Następnie test próbuje dodać do bazy jeszcze raz ten sam plik, czekając na wyjątek `FileExistsFault`. Na końcu następuje sprawdzenie funkcjonalności automatycznego tworzenia nadrzędnych katalogów dla nowego pliku. Po wykonaniu testu usuwane są usługi dCatalogue i SMS. Reszta operacji usługi dCatalogue testowana jest w analogiczny sposób: sprawdzane jest działanie na poprawnych danych (np. dla metody `IsDir` na utworzonym wcześniej katalogu i pliku), a następnie na niepoprawnych (zwykle jest to błędna nazwa pliku lub błędny identyfikator zasobu dSMS).

Oprócz testowania metod operujących na plikach i katalogach przechowywanych w usłudze dCatalogue, sprawdzana jest też rejestracja nowego „prywatnego” zasobu dSMS w bazie. Odpowiadają za to klasy `CatalogueWithFactory` (utworzenie zasobu dSMS gdy w rejestrze dostępna jest usługa Storage Factory) oraz `CatalogueWithStorage` (utworzenie zasobu dSMS w przeciwnym wypadku).

Sprawdzanie usługi dSMS przebiega według podobnego schematu. Przed każdym testem usuwane są pliki tymczasowe, inicjalizowana jest baza danych, a następnie tworzone są usługi: SMS, dCatalogue i dSMS. Same testy są bardzo proste — przykładowo metoda `testCreateDir` tworzy katalog, a następnie sprawdza, czy on rzeczywiście istnieje. Metoda `testImportExport` importuje, a następnie eksportuje plik i sprawdza, czy jego zawartość się zgadza.

Reszta operacji testowana jest w analogiczny sposób. Metody dSMS nie są natomiast sprawdzane pod kątem obsługi nieprawidłowych żądań. Wszystkie testy uruchamiane są najpierw w kontekście globalnego zasobu dSMS (klasa `TestDsmsBasics`), a następnie prywatnego (klasa `TestDsmsFactoryBasics`).

Oddzielna klasa `TestDsmsAutoSync` testuje dwa mechanizmy synchronizacyjne, opisane w sekcji 4.5.. Pierwsza metoda dodaje plik do usługi dSMS, a następnie usuwa go z fizycznego zasobu SMS. Później sprawdza, czy plik został usunięty również z zasobu dSMS (a więc czy stan usługi dCatalogue został automatycznie dostosowany do faktycznego stanu usługi SMS). Druga metoda analogicznie sprawdza, czy nowy plik — dodany do zasobu SMS — zostanie automatycznie dodany do bazy dCatalogue.

Testy (tak usługi dCatalogue jak i dSMS) można by rozszerzyć o sprawdzenie poprawności żądań wykonywach przez kilku klientów, identyfikujących się różnymi nazwami DN. Brakuje również sprawdzenia zachowania usług w sytuacji wielu współbieżnych żądań — choć zostało to częściowo przetestowane przy okazji testów wydajnościowych. Oprócz tego warto byłoby sprawdzać zachowanie usługi dSMS w sytuacji metod wywoływanych z nieprawidłowymi argumentami lub przy nieprawidłowej konfiguracji (np. brak usługi dCatalogue w rejestrze lub usunięcie zasobu SMS podczas pracy usługi dSMS) — aby mieć pewność co do treści pojawiających się w takich sytuacjach wyjątków i komunikatów.

5.2. Testy wydajności

Oddzielnym obszarem badań była weryfikacja wydajności. W celu sprawdzenia szybkości i skalowalności utworzonego systemu utworzono program narzędziowy o nazwie *sms-stress*, którego celem jest iteracyjne wywoływanie na wskazanych komponentach określonych metod. W trakcie działania programu zapisywany jest czas poszczególnych operacji. Aplikacja może służyć do testowania dowolnego zasobu SMS oraz usługi dCatalogue. Konfiguracja programu obejmuje możliwość zdefiniowania użytkowników, których tożsamości będą wykorzystywane, ilości tworzonych plików i katalogów (wraz z ich maksymalnym zagłębieniem) oraz ilości jednoczesnych wątków.

5.2.1. Skalowalność usługi dCatalgoue

Ponieważ od wydajności usługi dCatalogue zależy wydajność całego rozwiązania, dlatego też na początku przetestowano prędkość tego właśnie komponentu. Wykorzystana maszyna testowa to *ananiasz* — serwer oparty o system Linux, procesor Intel Core i7 920 2,67 GHz i posiadający 12 GB pamięci RAM. Oprócz wykonywania testów, na ananiaszu uruchomione są dwie maszyny wirtualne (razem zajmujące 4 GB pamięci operacyjnej). W trakcie pierwszych prób usługa Gateway przejawiała dziwne zachowanie (polegające na braku odpowiedzi przez 2–3 minuty). Zachowanie to powtarzało się kilkukrotnie w czasie jednego testu, dlatego też badania przeprowadzono bez wykorzystania tej usługi. Autoryzacja przebiegała w oparciu o pliki tekstowe, w których skonfigurowano trzy tożsamości użytkowników. Sprawdzono dwa obsługiwane systemy zarządzania bazą danych: H2 i MySQL (mechanizm składowania InnoDB). Ponieważ zarówno klient (sms-stress) jak i serwer (dCatalogue) znajdowały się na tej samej maszynie, rezultaty mogą być gorsze niż w sytuacji, w której te dwa programy

zostałyby rozdzielone. Dzieje się tak, gdyż w przypadku rozdzielenia powstałaby co prawda konieczność komunikacji między komponentami przez sieć, ale przesyłanych danych jest stosunkowo niewiele, a programy nie musiałyby się za to dzielić mocą procesora.

Test wydajności składał się z 50 iteracji. Każda iteracja polegała na dodaniu do bazy 2 tys. katalogów i 20 tys. plików. Następnie tworzono 10 wątków i każdy z nich 40-krotnie wykonywał następujące operacje: `Add`, `Lookup`, `IsDir`, `Remove`, `AddDir`, `LookupDir`, `RemoveDir`. Nazwy plików będących argumentami powyższych metod generowano przed utworzeniem wątków. Następnie łączna ilość przetworzonych w ramach każdej operacji rekordów (400) była dzielona przez czas wykonania operacji przez wszystkie wątki. Kompletny test powtórzono 4 razy, przedstawione na poniższych wykresach wyniki są średnią z tych 4 prób. Średnie odchylenie standardowe wynosi około 18 operacji na sekundę (dla H2) i 15 (dla MySQL).

Pierwsze wyniki nie były zadowalające. Prędkości usługi dCatalogue opartej o H2 dla kilku tysięcy plików w bazie wynosiła około 100 operacji na sekundę, jednak po dodaniu 20 tys. plików prędkość ta spadła do zaledwie kilku operacji na sekundę. W przypadku MySQL wydajność była większa i dla metod nie wymagających modyfikacji bazy wynosiła 150 operacji na sekundę, zaś dla metod wiążących się z modyfikacją około 20 operacji na sekundę.

Aby zoptymalizować działanie usługi, najpierw przeanalizowano plany wykonania zapytań H2 i zmodyfikowano indeksy bazy. Zamiast indeksów pojedynczych, powiązanych z kluczami obcymi (na początku istniały tylko takie), utworzono indeksy grupowe na zbiorach kolumn najczęściej używanych w zapytaniach SQL. Przeanalizowano również kod źródłowy komponentu `DBDao` usługi dCatalogue odpowiedzialnego za komunikację z bazą danych. W kilku miejscach usunięto niepotrzebne powtarzanie wywołań SQL. Zarówno pliki i katalogi przechowywane są w jednej tabeli bazy danych o nazwie `files`. Ponieważ wyszukanie katalogu jest operacją o wiele częstszą niż wyszukanie pliku, a katalogów zwykle jest mniej, wprowadzono dodatkowy indeks powiązany z kolumną rozróżniającą pliki od katalogów i zmodyfikowano `DBDao` tak, aby przy wyszukiwaniu katalogów korzystał właśnie z tego indeksu. Na koniec wprowadzono mechanizm pamięci podręcznej, która przechowuje ostatnio odczytywane z bazy informacje o plikach i katalogach. Użyto w tym celu biblioteki Ehcache. Wprowadzone modyfikacje sprawiły, że usługa dCatalogue oparta o bazę H2 stała się znacznie szybsza. Ostateczna prędkość jest zaprezentowana na wykresie 5.1.

Przystąpiono do optymalizacji usługi opartej o MySQL. Dość niska prędkość metod modyfikujących dane (20 operacji na sekundę) była spowodowana tym, że po każdej takiej operacji silnik bazy danych wykonywał operację *flush*, która powoduje zapisanie wszelkich zmian na dysk. Ustawienie parametru konfiguracyjnego MySQL `innodb_flush_log_at_trx_commit` na `0` spowodowało, że operacje te zaczęły być wykonywane z taką samą prędkością, jak operacje odczytu. Przy takiej konfiguracji zmiany zapisywane są na dysku co sekundę. Negatywnym skutkiem jest utrata właściwości ACID, gdyż może zdarzyć się, że zatwierdzona transakcja nie zostanie zapisana i po ewentualnej awarii dane zmodyfikowane podczas tej transakcji nie będą dostępne. Na przykład: informacja o zaimportowanym pliku nie znajdzie się w bazie. Ponieważ jednak ważniejszy jest faktyczny stan plików w zasobach SMS, to jeśli udało się plik z powodzeniem zaimportować, metody synchronizacyjne powinny dodać go również do bazy dCatalogue. Dodatkowo należy zauważyć, że prawdopodobieństwo wystą-

pienia awarii w takim układzie jest bardzo małe, więc taka optymalizacja jest dopuszczalna.

Kolejny problem polegał na tym, że po dodaniu 400 tysięcy plików operacje gwałtownie zwalniały — z poziomu około 200 operacji odczytu na sekundę do 100. Lekarstwem na taki stan rzeczy okazało się ponowne uruchomienie usługi dCatalogue, która — jak się okazało — zajmowała coraz więcej pamięci operacyjnej. Winnym okazał się być przeciek pamięci w klasach odpowiedzialnych za szyfrowaną transmisję SSL, należących do standardowej biblioteki języka Java. Po aktualizacji wirtualnej maszyny Javy z wersji 1.6.0_18 do 1.6.0_26 problem zniknął. Ostateczne wyniki zaprezentowane są na wykresie 5.2.

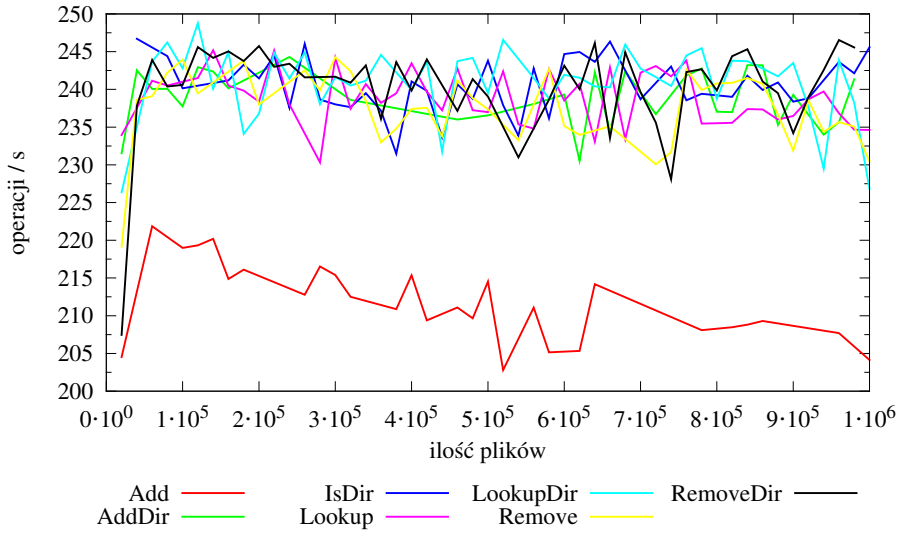
Z uzyskanych rezultatów wynika, że system H2 przy bazach o wielkości nie przekraczającej miliona plików jest bardziej wydajny niż rozwiązanie oparte o MySQL. Wraz ze zwiększaniem ilości plików spada jedynie (i to niezbyt gwałtownie) prędkość operacji `Add`. W przypadku bazy MySQL metody `Lookup`, `LookupDir` i `IsDir` (a więc najczęściej używane metody odczytujące dane z bazy) zachowują się stabilnie i działają z prędkością przybliżoną do H2, czyli około 230 operacji na sekundę. Nieco mniejszą prędkość (około 210 operacji na sekundę) mają metody `AddDir`, `Remove` i `RemoveDir`. Wyraźnie zwalnia natomiast operacja `Add`, prędkość początkowa wynosi tu prawie 160 operacji na sekundę, zaś końcowa nieco poniżej 100 operacji na sekundę. Jest to zapewne związane z koniecznością aktualizacji coraz większego zbioru indeksów.

5.2.2. Narzut dla typowych operacji usługi dSMS

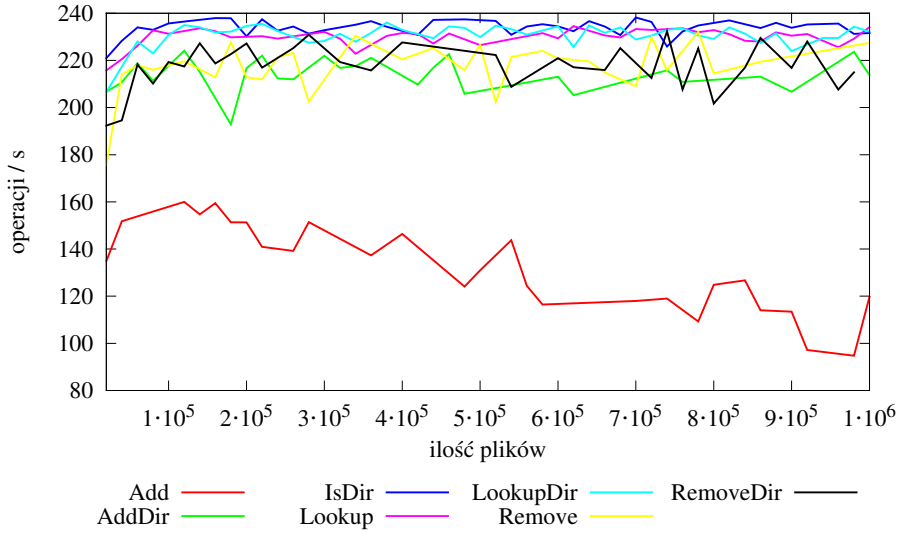
Kolejnym przeprowadzonym testem było porównanie prędkości wykonania typowych operacji przez klasyczną usługę SMS i przez dSMS. Maszyny testowe to wspomniany w poprzedniej sekcji ananiasz oraz *alfred* — serwer linuksowy wyposażony w procesor AMD Opteron 285, 2.6 GHz Dual Core i 8 GB pamięci RAM. Alfred jest dość obciążoną maszyną, uruchomiona jest na nim oddzielna instalacja UNICORE, a oprócz tego serwer bazy danych PostgreSQL i serwer Apache. Program testowy, usługa dCatalogue oraz docelowy zasób SMS był uruchomiony na ananiaszu, natomiast usługa dSMS na alfredzie. Ponownie, umieszczenie klienta (sms-stress) i dwóch serwerów (dCatalogue i SMS) na ananiaszu mogło negatywnie odbić się na wydajności poniższych testów, jednak z drugiej strony przeniesienie jeszcze jednego komponentu na już obciążonego alfreda powodowało nienaturalnie długi czas wykonywania operacji.

Na początku utworzono 100 katalogów, a następnie 10 wątków powtórzyło po 50 razy następujące operacje: `ImportFile`, `ChangePermissions`, `ListProperties`, `Find`, `ListDirectory`, `ExportFile`, `Copy`, `Rename`, `CreateDirectory`, `Delete`.

Najpierw przetestowano zasób SMS, a następnie skonfigurowaną usługę dSMS tak, aby korzystała z tego (i tylko z tego) zasobu. Opcje synchronizacyjne były wyłączone. Celem było zbadanie minimalnego „narzutu” który towarzyszy użyciu prezentowanego systemu. Wyniki znajdują się w tabeli:



Rysunek 5.1: Prędkość usługi dCatalogue w zależności od ilości plików (baza H2).



Rysunek 5.2: Prędkość usługi dCatalogue w zależności od ilości plików (baza MySQL).

	Czas trwania operacji w s			
Nazwa operacji	SMS	dSMS	Narzut w s	Narzut w %
`ChangePermissions`	0,0022	0,0384	0,0363	1686,16
`Copy`	0,0020	0,1117	0,1097	5417,65
`CreateDirectory`	0,0020	0,0943	0,0922	4565,88
`Delete`	0,0016	0,0509	0,0492	2996,80
`ExportFile`	0,0140	0,0387	0,0248	177,68
`Find`	0,0018	0,0349	0,0331	1817,80
`ImportFile`	0,0187	0,0746	0,0558	297,76
`ListDirectory`	0,0018	0,0352	0,0334	1819,55
`ListProperties`	0,0017	0,0470	0,0453	2713,91
`Rename`	0,0017	0,0618	0,0601	3527,98

Pominięto operacje `SendFile` i `ReceiveFile`, które (jeśli chodzi o implementację w usłudze dSMS) są bardzo podobne do `ExportFile` i `ImportFile`. Ostatnia kolumna wskazuje, że względny narzut jest bardzo duży. Czas wykonania operacji przez dSMS jest często kilkudziesięciokrotnie większy niż czas wykonania tej samej operacji przez SMS. Dzieje się tak, ponieważ klasyczna usługa SMS operuje na lokalnych plikach. Z kolei usługa dSMS za każdym razem musi wykonać szereg wywołań — najpierw do bazy dCatalogue, aby określić lokalizację danego pliku, a następnie do docelowego zasobu SMS. W przypadku metod `Copy` i `Rename` sytuacja jest jeszcze bardziej skomplikowana, gdyż trzeba wykonać dwa wywołania do usługi dCatalogue, a przy kopiowaniu dodatkowo przetransferować dane między zdalnymi zasobami SMS (patrz opis działania operacji `Copy` w sekcji 4.1.). Najmniejszy narzut jest widoczny w przypadku operacji `ImportFile` i `ExportFile`, gdyż dla tych metod większość czasu zajmuje transfer danych, który odbywa się z taką samą prędkością dla usług SMS i dSMS. Dla celów testowych transferowano niewielkie ilości danych (kilka bajtów). Ponieważ jednak bezwzględna różnica czasów wykonania pozostanie taka sama dla dowolnej wielkości plików, w przypadku tych dwóch operacji narzut najczęściej nie będzie w ogóle zauważalny (chyba, że transferować będziemy ogromne ilości naprawdę małych plików).

Wnioski, które możemy wyciągnąć, są takie: dSMS dobrze sprawdza się w operacjach związanych z transferem plików, w większości przypadków użytkownicy nie zauważą różnicy rzędu 0,05 sekundy (przy wyłączonej synchronizacji). Włączenie synchronizacji przy 5 zasobach SMS wydłuża operację importu o kolejne 0,07 sekundy, ale cała operacja transferu danych trwa zwykle znacznie dłużej. Jeśli jednak należy np. wykonać operację zmiany uprawnień lub zmiany nazw na wielkiej ilości plików to klasyczny SMS będzie kilkanaście lub nawet kilkadziesiąt razy szybszy od synchronizowanego zasobu dSMS. Wpływ mechanizmu synchronizacji na wydajność oraz trzecia kategoria operacji, czyli `ListDirectory` i `Find` zostaną omówione w kolejnej podsekcji.

5.2.3. Skalowalność usługi dSMS

Skalowalność usługi dSMS zbadano względem ilości składowych zasobów SMS. Konfiguracja środowiska była taka sama jak w poprzedniej sekcji. Fizyczne zasoby SMS (w ilości od 1 do 10) tworzono na alfredzie i ananiaszu. Wykonano dwa testy — w pierwszym synchronizacja była wyłączona, natomiast w drugim włączona. Z wykresu 5.3 wynika, że w przypadku braku synchroniza-

cji wszystkie operacje oprócz `Copy`, `ListDirectory` i `Find` działają z podobną prędkością niezależnie od ilości składowych zasobów SMS. Operacja `Copy` działa widocznie szybciej w przypadku jednego zasobu SMS. W tym przypadku plik kopiowany jest bez użycia usług sieciowych, za pomocą standardowych mechanizmów systemu operacyjnego. Po dodaniu drugiego zasobu SMS pliki muszą być już transferowane między serwerami, co wydłuża całą operację. Z kolei metody zwracające zawartość katalogów, tzn. `ListDirectory` i `Find` muszą połączyć się ze wszystkimi zasobami SMS, aby pobrać listę plików, połączyć ją i przekazać klientowi. Z tego powodu zwalniają one wraz ze wzrostem liczby składowych usług SMS.

Wykres 5.4 prezentuje analogiczny test przy włączonej synchronizacji. Nie zmieniła się prędkość operacji `ListDirectory` i `Find` — jak wspomniano w sekcji 4.5., w ich przypadku nie są potrzebne żadne mechanizmy synchronizujące. Dodawanie kolejnych zasobów SMS spowalnia jednak operacje `ImportFile`, `Copy` i `CreateDirectory`. Wymagają one utworzenia nowego pliku, a zastosowane mechanizmy synchronizujące sprawdzają, czy plik o tej nazwie nie istniał już wcześniej na którymś z zasobów SMS. Słowo komentarza należy się metodzie `CreateDirectory` — istniejąca implementacja najpierw tworzy katalog, a potem umieszcza w nim plik tymczasowy (który jest po chwili usuwany). Jest to związane ze specyfiką usługi dCatalogue. Z powodu takiej właśnie implementacji, metoda ta zwalnia podobnie do `ImportFile`. Pozostałe metody nie podlegają w tym przypadku synchronizacji, dlatego też ich prędkość jest podobna do zaprezentowanej na poprzednim wykresie 5.3.

Jeśli chodzi o porównanie wydajności usługi dSMS synchronizowanej i niesynchronizowanej, to przy operacjach `ImportFile` i `Copy` nie będzie widoczna znacząca różnica. Jak wspomniano w poprzedniej podsekcji, najwięcej czasu w tym przypadku zajmuje transfer pliku i 0,18 sekundy różnicy (w przypadku operacji `ImportFile` przy 10 zasobach SMS) nie jest istotną wartością, o ile nie importujemy i nie kopiujemy dużej ilości bardzo małych plików. Różnica może być natomiast widoczna przy tworzeniu skomplikowanej struktury katalogów — w takim przypadku również tracimy około 0,2 sekundy na katalogu, jednak cała operacja trwa przez to ponad 3 razy dłużej.

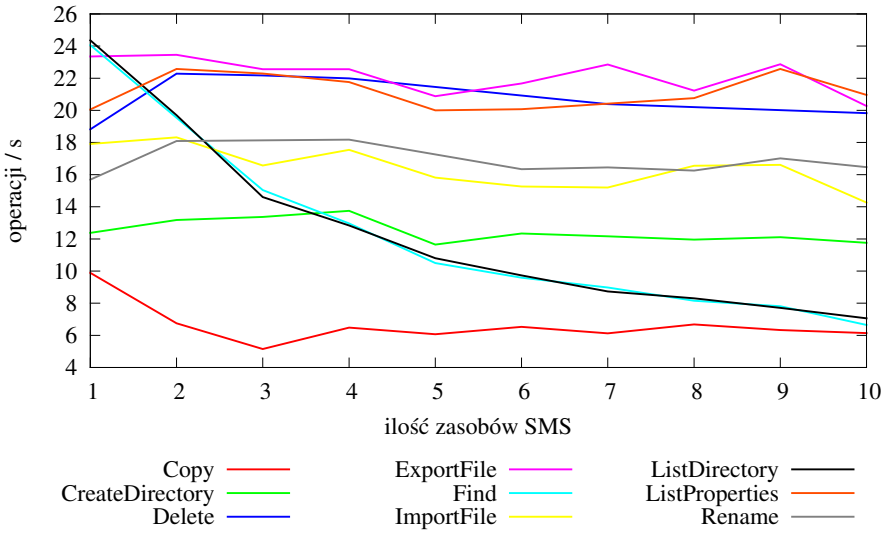
5.3. Analiza stabilności

Przypomnijmy, że prezentowane rozwiązanie składa się z:

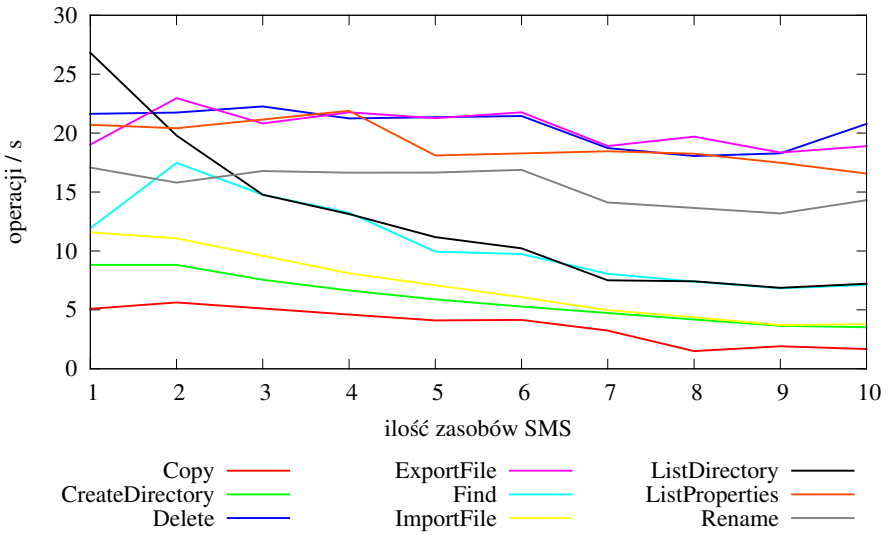
- centralnej usługi dCatalogue,
- co najmniej jednej usługi dSMS,
- dowolnej ilości fizycznych usług Storage Factory,
- dowolnej ilości fizycznych zasobów SMS.

Należy zauważyć, że dwa ostatnie elementy to standardowe komponenty UNICORE.

Jeśli awarii ulegnie usługa dCatalogue, wówczas wszystkie korzystające zeń zasoby dSMS przestają działać. Przy próbie wykonania jakiejkolwiek operacji zwracany jest wyjątek z informacją o braku usługi dCatalogue w rejestrze. Dostęp do plików można uzyskać jedynie łącząc się bezpośrednio z fizycznymi



Rysunek 5.3: Prędkość usługi dSMS w zależności od ilości składowych zasobów SMS (synchronizacja wyłączona).



Rysunek 5.4: Prędkość usługi dSMS w zależności od ilości składowych zasobów SMS (synchronizacja włączona).

zasobami SMS, próbując „ręcznie" odnaleźć odpowiednie dane. Ponieważ cała zawartość usługi dCatalogue jest przechowywana w bazie danych, można by się zastanowić nad replikacją usługi właśnie na takim poziomie. MySQL domyślnie daje możliwość replikacji danych z serwera głównego (ang. *master* — mistrz) na serwery zapasowe (ang. *slave* — niewolnik). Wówczas, w razie awarii głównego serwera MySQL można skorzystać z serwera zapasowego, natomiast w przypadku problemów z dCatalogue można uruchomić nową usługę tak, aby korzystała z istniejącej bazy.

Usługi dSMS są bezstanowe i w celu zapisania jakichkolwiek danych korzystają z usługi dCatalogue i zasobów SMS, są też w pełni replikowalne. Awaria usługi dSMS nie powinna stanowić problemu, ponieważ można uruchomić i udostępnić wiele równoważnych instancji.

Jeśli awarii ulegnie usługa Storage Factory, a w rejestrze nie ma globalnego zasobu SMS wówczas niemożliwe stanie się tworzenie nowych, prywatnych zasobów dSMS. Dostęp do danych już wcześniej obecnych w usłudze dSMS pozostanie niezakłócony.

Ostatni scenariusz dotyczy awarii jednego z fizycznych zasobów SMS. W tej sytuacji należy przeanalizować działanie operacji dSMS:

- `ChangePermissions`, `Copy`, `Delete`, `ExportFile`, `ListProperties`, `Rename`, `SendFile` — jeśli plik znajduje się na niedostępnym zasobie SMS, operacja zakończy się zwróceniem wyjątku. W przeciwnym wypadku zostanie wykonana poprawnie,
- `ImportFile`, `ReceiveFile` — jeśli dostępny jest co najmniej jeden sprawny zasób SMS, operacja będzie wykonana poprawnie,
- `Find`, `ListDirectory` — zwrócona lista nie będzie zawierać plików z niedostępnego zasobu SMS.

Bez wprowadzenia replikacji plików nie można efektywnie rozwiązać tego problemu.

5.4. Możliwe usprawnienia

Pierwszym usprawnieniem, które wydaje się być niezbędne dla efektywnej pracy w środowisku produkcyjnym jest implementacja bardziej zaawansowanych algorytmów rozdziału plików między zasoby SMS. W tej chwili dostępne są jedynie dwa algorytmy — domyślnie używany jest round robin, gotowy jest też algorytm wybierający zasób z największą ilością wolnej przestrzeni dyskowej. Można wymyślić bardziej zaawansowane metody, biorące pod uwagę obciążenie poszczególnych serwerów, ich odległość od użytkownika i od maszyn, na których będą wykonywane obliczenia, itd. Aby zwiększyć wydajność operacji `ListDirectory` i `Find` warto rozważyć algorytm uwzględniający strukturę katalogów (tak, aby pliki z jednego katalogu były przechowywane na jednym zasobie SMS) lub informacje o właścicielach plików (grupowanie plików jednego użytkownika). Dodawanie własnych algorytmów jest dość proste, jednak zdobywanie dodatkowych informacji (np. dotyczących lokalizacji klienta i obciążenia serwerów) jest już zadaniem nietrywialnym.

Kolejną funkcją, która mogłaby być użyteczna, jest replikacja plików między zasoby SMS. Zwiększyłoby to bezpieczeństwo przechowywanych danych. Implementacja takiego rozwiązania nie jest jednak prosta. Można wyobrazić sobie dwa rozwiązania — w pierwszym z nich replika powstaje w momencie importu pliku przez klienta. Klient mógłby na przykład połączyć się z usługą dSMS za pomocą protokołu FTS, a usługa dSMS łączyłaby się z kilkoma zasobami dyskowymi i przesyłała otrzymany przez klienta plik. Wadą takiego rozwiązania jest fakt, że usługa dSMS musiałaby pośredniczyć we wszystkich transmisjach danych i mogłaby się przez to stać wąskim gardłem (nawet jeśli sama jest replikowalna). W tej chwili problem ten nie istnieje, gdyż klient przesyła plik bezpośrednio do docelowego zasobu SMS. Drugim rozwiązaniem jest tworzenie replik podczas okresów niskiej aktywności systemu (np. w nocy). Problem który tu występuje to konieczność przechowywania na serwerze delegacji zaufania wystawionych przez klientów tak, aby w ich imieniu móc utworzyć repliki na docelowych zasobach SMS.

Podobny problem występuje przy kolejnej funkcji, którą można by poprawić, tzn. przy synchronizacji. W tej chwili synchronizacja następuje przy każdym żądaniu, co — jak pokazano wyżej — zwalnia niektóre operacje. Tymczasem proces synchronizujący mógłby działać autonomicznie. Do tego jednak również potrzebne są delegacje zaufania wystawione przez klientów, które należy jakoś przechowywać.

Ciekawym pomysłem byłaby możliwość wyposażenia użytkowników w narzędzia, które pozwalają bardziej precyzyjnie określać np. na którym zasobie SMS ma się znaleźć importowany przez nich plik, w ilu replikach ma być przechowywany, itd. Ta zmiana wymagałaby z kolei rozbudowy interfejsu SMS o dodatkowe metody, specyficzne dla dSMS. W tym momencie prezentowany system straciłby jednakże jedną ze swoich głównych zalet, tzn. możliwość wykorzystania dowolnych narzędzi obsługujących interfejs SMS. Oczywiście, można by zachować wsteczną kompatybilność, tworząc na przykład rozszerzenie do klienta UCC.

Warto rozważyć również rozszerzenie operacji interfejsu SMS o możliwość działania na grupach plików. Przykładem mogłoby być usunięcie wielu plików na raz, czy też rekursywna zmiana uprawnień (przez `ChangePermissions`). Oprócz wygody użytkownika, poprawiłoby to wydajność usługi dSMS która — jak pokazano wyżej — jest dość wolna w przypadku konieczności wykonania wielu krótkich operacji. Oczywiście taka zmiana wymagałaby modyfikacji interfejsu SMS, jednak nie byłoby to rozszerzenie specyficzne dla dSMS, lecz mogłoby być zaimplementowane również przez zwykłe, fizyczne zasoby SMS.

W sekcji 5.3. opisano hipotetyczną metodę replikacji usługi dCatalogue na poziomie bazy MySQL. Należałoby zbadać działanie takiego mechanizmu w praktyce, gdyż dCatalogue jest elementem niezbędnym dla poprawnego działania całego systemu. Oprócz tego można rozważyć modyfikację usługi tak, aby była możliwość replikacji na wyższym poziomie — na przykład kilka instancji dCatalogue działających w ramach systemu gridowego, synchronizujących dane między sobą.

Zakończenie

Celem pracy było rozszerzenie oprogramowania warstwy pośredniej UNICORE o możliwość rozproszonego przechowywania danych. Taki mechanizm znacznie ułatwiłby życie użytkownikom, którzy nie musieliby się już zastanawiać, którą przestrzeń dyskową wybrać do przechowywania swoich danych i wyników. Z kolei administratorzy systemu gridowego mogliby na przykład w prosty sposób dodawać nowe serwery dyskowe — wspólna przestrzeń powiększyłaby się automatycznie.

Aby zrealizować ten cel, na początku przeanalizowano rozwiązania istniejące w innych systemach gridowych, a także dostępne możliwości w systemie UNICORE. Na podstawie tej analizy sformułowano listę wymagań. Niestety, żadne z istniejących rozwiązań nie spełniało wszystkich założeń. Utworzono więc nowy system, zaprezentowany w rozdziale 4.

System dSMS — mimo że znajduje się we wczesnej fazie rozwoju — spełnia większość sformułowanych wymagań, tzn. pozwala na stworzenie z kilku zasobów SMS jednej spójnej przestrzeni, dostępnej również za pomocą usługi SMS. Obsługiwane są wszystkie protokoły transmisji plików. System został w miarę możliwości zdecentralizowany, poza usługą dCatalogue, która jednak w założeniu jest lekka i szybka. Stworzone rozwiązanie daje możliwość uwierzytelniania za pomocą standardowych mechanizmów UNICORE. Architektura systemu jest dość prosta, a instalacja ma podobny poziom trudności, co instalacja pozostałych komponentów oprogramowania UNICORE.

Analiza przeprowadzona w rozdziale 5 wykazała, że rozwiązanie nie posiada istotnych problemów wydajnościowych. Oczywiście, optymalizacja w dalszym ciągu jest możliwa, zwłaszcza jeśli chodzi o mechanizmy synchronizujące i sytuacje, w których wykonywana jest duża ilość krótkich operacji, takie jak zmiana praw dostępu wielu plików. Oprócz optymalizacji, w sekcji 5.4. przedstawiono propozycje dalszych prac.

O zaakceptowaniu prezentowanego rozwiązania przez środowisko twórców oprogramowania UNICORE świadczy fakt, że system dSMS jest włączony do standardowej dystrybucji UNICORE od wersji 6.4.0. Zostanie on również zaprezentowany na spotkaniu UNICORE Summit 2011 w Toruniu, a jego podstawowe założenia opisano w artykule [27], który będzie wydany wraz z innymi materiałami z tej konferencji.

Dodatek A

Opis instalacji

Instalacja składa się z dwóch etapów: instalacji usługi dCatalogue i usługi dSMS. Usługa dSMS wymaga posiadania, w ramach systemu gridowego, działającego serwera UNICORE w wersji co najmniej 6.4.0.

A.1. Instalacja usługi dCatalogue

Najpierw należy pobrać pliki usługi dCatalogue. W chwili obecnej (czerwiec 2011) można je znaleźć pośród wydań projektu UNICORE Life na serwisie Sourceforge[1]. Po pobraniu i rozpakowaniu należy dokonać następujących czynności:

1. Utworzyć plik `conf/keystore.jks` zawierający certyfikat centrum autoryzacji używanego w ramach systemu gridowego. Oprócz tego plik ten powinien zawierać klucz prywatny i powiązany z nim certyfikat podpisany przez wspomniane centrum autoryzacji.

2. W pliku `conf/uas.config` wpisać adres rejestru (parametr `uas.externalregistry.url`) oraz nazwę VSITE (parametr `uas.targetsystem.sitename`).

3. W pliku `conf/wsrflite.xml` ustawić adres usługi dCatalogue przy użyciu parametru `unicore.wsrflite.baseurl` (zwykle wystarczy zmienić pierwszą część, będącą adresem usługi Gateway). Należy również skonfigurować obsługę utworzonego pliku `keystore.jks` za pomocą parametrów o prefiksach `unicore.wsrflite.ssl.key` i `unicore.wsrflite.ssl.trust`.

4. Ustawić parametry używanej bazy danych w pliku `conf/datamap.properties`. Bazę danych przed pierwszym użyciem inicjuje się komendą `bin/init_db.sh`.

5. Do pliku konfiguracyjnego `connections.properties` dodać linię z informacją o usłudze dCatalogue, np. `DCATALOGUE = https://localhost:7750`.

[1] Pliki znajdują się pod adresem `http://sourceforge.net/projects/unicore-life/files/` (19.06.2011)

6. Skonfigurować usługę źródła atrybutów i polityki bezpieczeństwa tak, aby usługa dCatalogue miała dostęp do właściwości zasobów SMS i Storage Factory. I odwrotnie — należy skonfigurować odpowiednie źródło atrybutów i politykę bezpieczeństwa usługi dCatalogue, aby użytkownicy systemu gridowego mogli się z nią łączyć.

7. Po tych operacjach można uruchomić usługę komendą `bin/start.sh`.

Dodatkowo, w pliku `conf/uas.config` można jawnie wskazać, z których zasobów SMS i Storage Factory powinna korzystać usługa dSMS. Służą do tego parametry `catalogue.smses`, `catalogue.factories`. Należy po nich wskazać listę adresów oddzielonych spacjami. Jeśli parametry te nie zostaną ustawione, usługa dCatalogue będzie wyszukiwać odpowiednie zasoby w rejestrze.

A.2. Konfiguracja usługi dSMS

Usługa dSMS jest standardowo dołączona do serwera UNICORE/X od wersji 6.4.0. Aby uruchomić ją w ramach już działającego kontenera, należy najpierw upewnić się, że usługa dCatalogue działa i jest dostępna w rejestrze systemu gridowego. Następnie, aby utworzyć globalny zasób dSMS, do którego dostęp będą mieli wszyscy użytkownicy, należy dodać nazwę klasy
`de.fzj.unicore.uas.impl.dsms.CreateDSMSOnStartup`
do parametru `uas.onstartup` w pliku `uas.config`. W tym przypadku powinien być dostępny co najmniej jeden globalny zasób SMS dostępny w rejestrze. Nie należy też zapomnieć o konfiguracji odpowieniej polityki bezpieczeństwa.

Jeśli usługa Storage Factory ma mieć możliwość tworzenia zasobów dSMS, należy dopisać do pliku `uas.config` następujące linie:

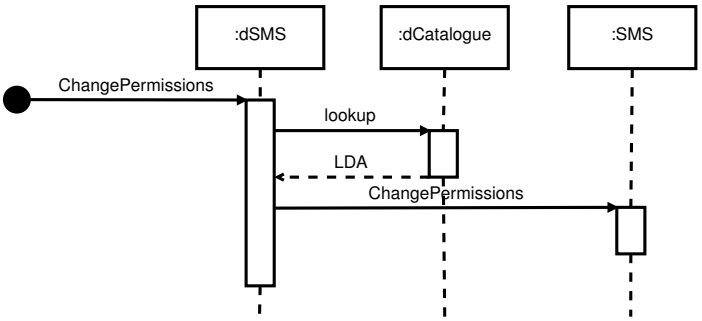
```
uas.storagefactory.DSMS.type=CUSTOM
uas.storagefactory.DSMS.class=de.fzj.unicore.uas.impl.dsms.DSMSImpl
uas.storagefactory.DSMS.infoprovider=\
de.fzj.unicore.uas.impl.dsms.DSMSInfoProvider
uas.storagefactory.DSMS.cleanup=true
```

Następnie należy znaleźć parametr `uas.storagefactory.storagetypes` i dopisać do niego `DSMS`. Aby taka konfiguracja działała poprawnie, w rejestrze powinien być globalny zasób SMS lub co najmniej jedna usługa Storage Factory pozwalająca tworzyć „fizyczne” zasoby SMS.

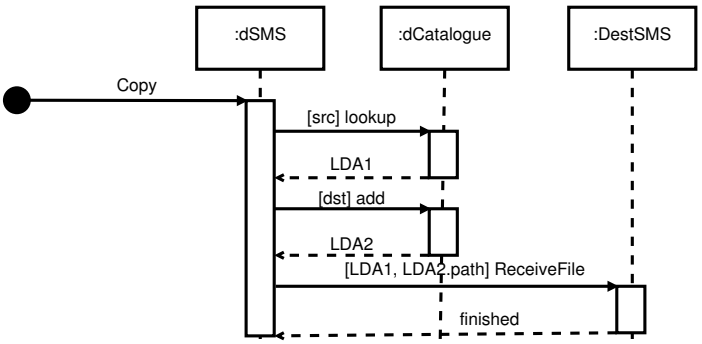
Dodatek B

Schematy UML

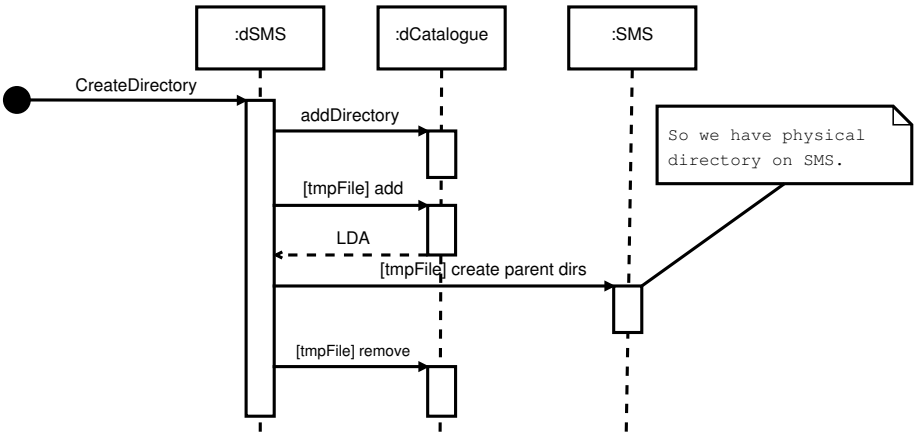
W trakcie projektowania systemu dSMS wszystkie operacje interfejsu SMS zostały opisane na diagramach interakcji języka UML. Diagramy te znajdują się poniżej.



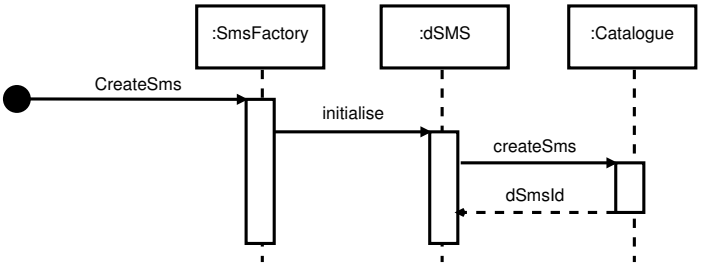
Rysunek B.1: Operacja zmiany uprawnień



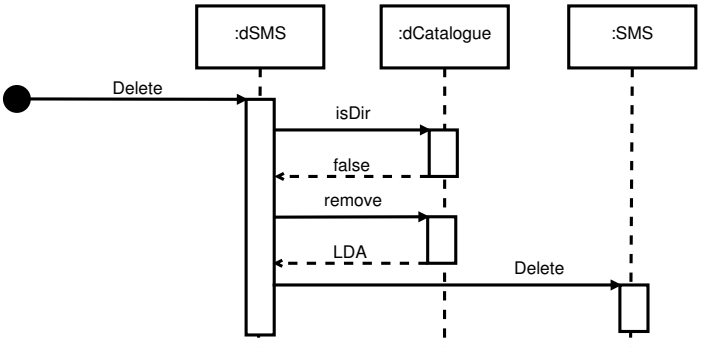
Rysunek B.2: Operacja kopiowania pliku



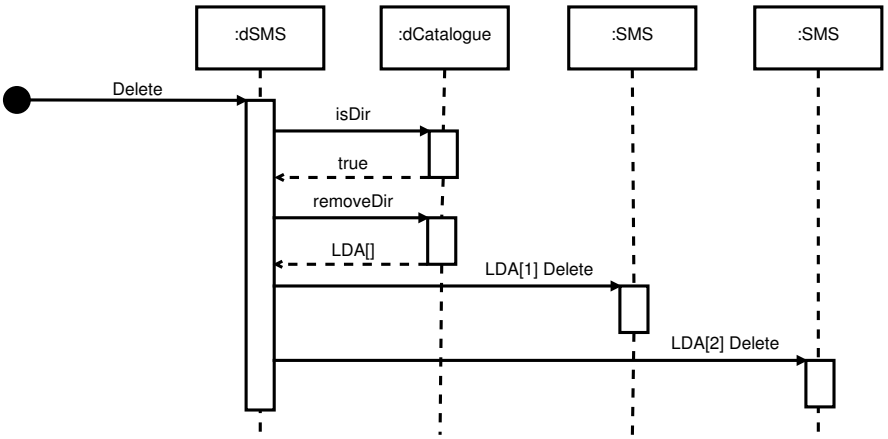
Rysunek B.3: Operacja utworzenia katalogu



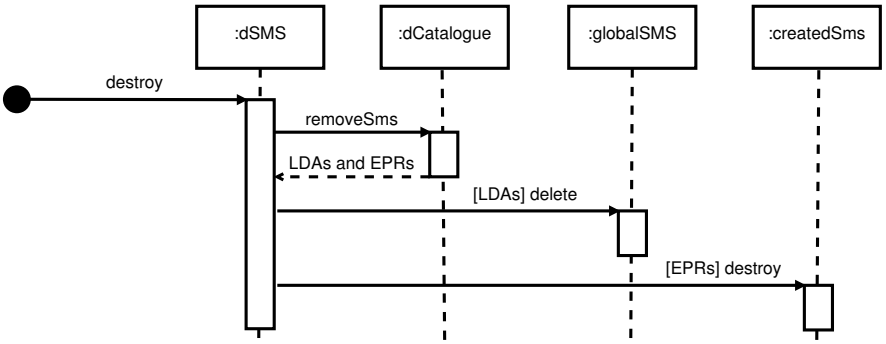
Rysunek B.4: Operacja utworzenia nowego zasobu dSMS wywołana na usłudze Storage Factory



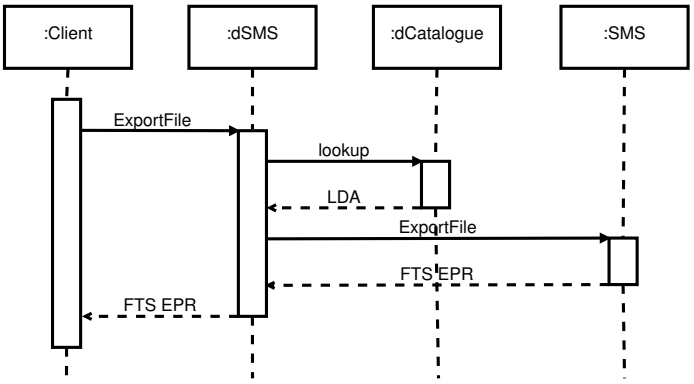
Rysunek B.5: Operacja usunięcia pliku



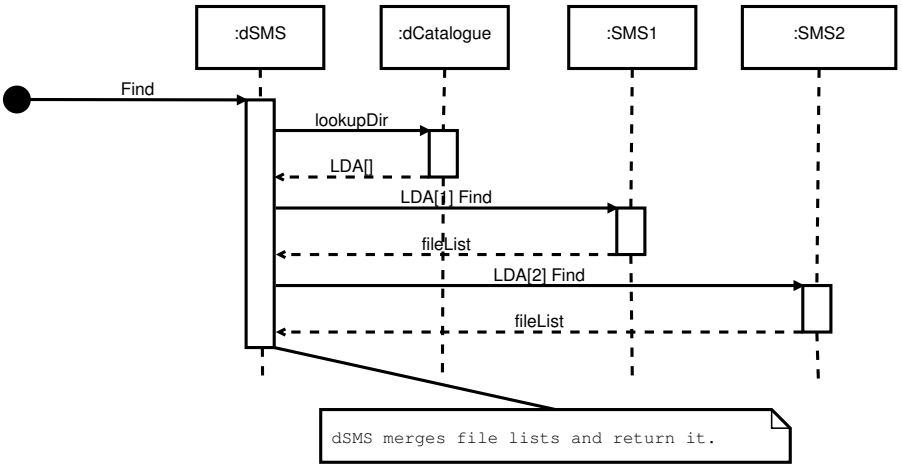
Rysunek B.6: Operacja usunięcia katalogu



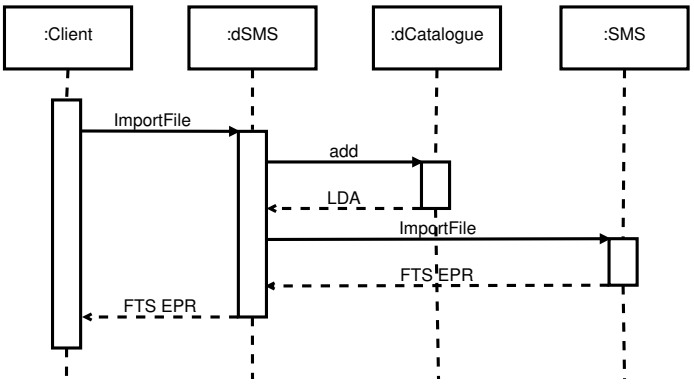
Rysunek B.7: Operacja usunięcia zasobu dSMS



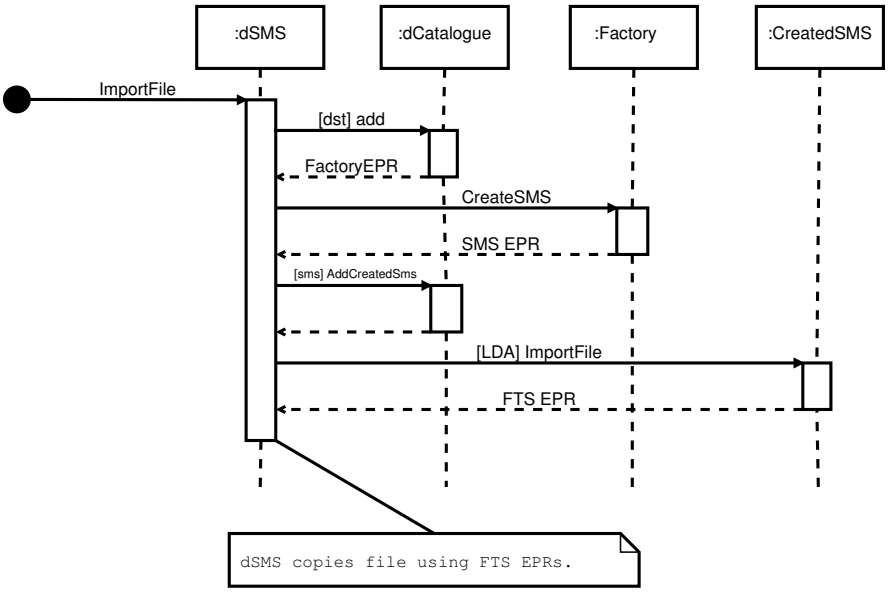
Rysunek B.8: Operacja eksportu pliku



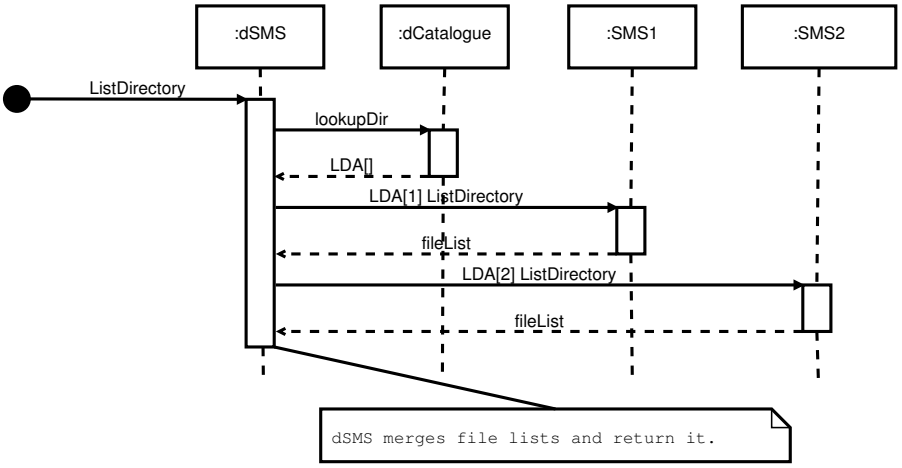
Rysunek B.9: Operacja wyszukania plików



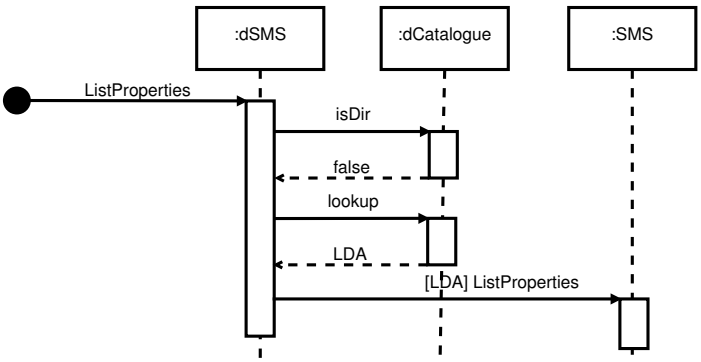
Rysunek B.10: Operacja importu pliku



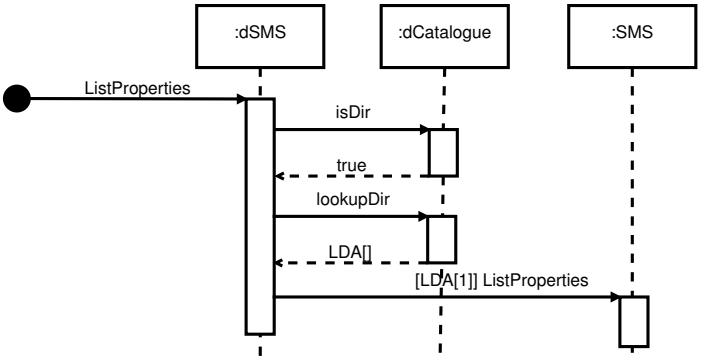
Rysunek B.11: Operacja importu pliku w sytuacji, gdy należy utworzyć nowy, fizyczny zasób SMS



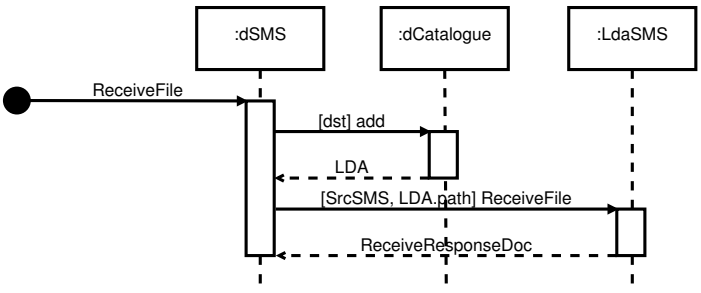
Rysunek B.12: Operacja pobrania zawartości katalogu



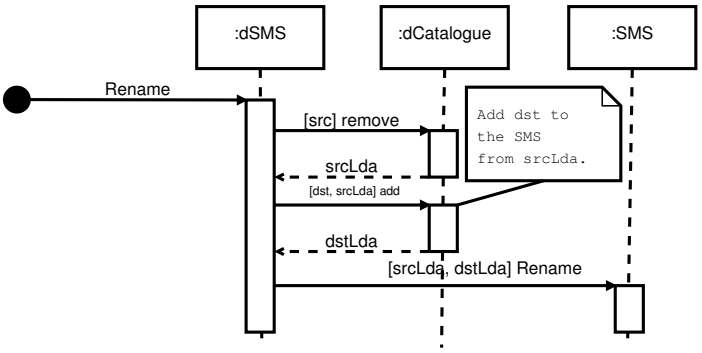
Rysunek B.13: Operacja pobrania informacji o pliku



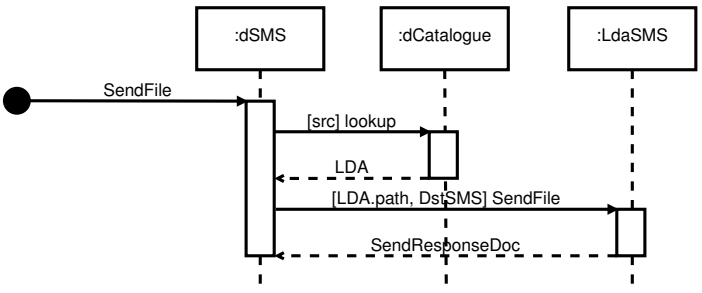
Rysunek B.14: Operacja pobrania informacji o katalogu



Rysunek B.15: Operacja przesłania pliku ze zdalnego zasobu SMS



Rysunek B.16: Operacja zmiany nazwy



Rysunek B.17: Operacja przesłania pliku na zdalny zasób SMS

Bibliografia

[1] Apache Hadoop, URL: `http://hadoop.apache.org/` (10.04.2011)

[2] Atkinson M., Karasavvas K., Antonioletti M., Baxter R., Borley A., Hong N.C., Hume A., Jackson M., Krause A., Laws S., Paton N., Schopf J., Sugden T., Tourlas K., Watson P.: A New Architecture for OGSA-DAI, *UK e-Science All Hands Meeting*, EPSRC, 2005.

[3] Baud J., Casey J., Lemaitre S., Nicholson C.: *LCG File Catalog (LFC) administrators' guide*, URL: `https://twiki.cern.ch/twiki/bin/view/LCG/LfcAdminGuide#Introduction` (01.04.2011)

[4] CASTOR — CERN Advanced STORage Manaer, URL: `http://castor.web.cern.ch/castor/` (26.02.2011)

[5] CNAF — Italian National Center for Research and Development about Information and Data transmission Technologies, URL: `http://www.cnaf.infn.it/main/index.php/en/` (28.02.2011)

[6] Corso E., Cozzini S., Forti A., Ghiselli A., Magnoni L., Messina A., Nobile A., Terpin A., Vagnoni V., Zappi R.: StoRM: A SRM solution on disk based storage system. *Proceedings of the Cracow Grid Workshop 2006*, Kraków 2006.

[7] Derc K.: *Integracja systemu iRODS ze środowiskiem gridowym UNICORE 6*, Toruń 2008.

[8] Foster I.: What is the Grid? A Three Point Checklist. *GRIDToday*, 20.07.2002.

[9] Foster I., Kesselman C. (red): *The Grid: Blueprint for a New Computing Infrastructure*. Wyd. 2. Morgan Kaufmann Publishers 2004.

[10] Foster I., Kesselman C., Nick J., Tuecke S.: The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, *Open Grid Service Infrastructure WG*, Global Grid Forum, 20.07.2002.

[11] Foster I., Kesselman C., Tuecke S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations, *International J. Supercomputer Applications*, 15(3), 2001.

[12] Fuhrmann P., Gülzow V.: dCache, Storage System for the Future. *Euro-Par 2006 Parallel Processing*, tom 4128 z serii Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2006.

[13] Ghemawat S., Gobioff H., Leung S.: The Google File System *19th ACM Symposium on Operating Systems Principles*. Lake George, NY, 2003.

[14] Global Grid Forum: *Job Submission Description Language (JSDL) Specification*, URL: `www.gridforum.org/documents/GFD.56.pdf` (27.06.2011)

[15] Grid Data Management — DPM, URL: `https://svnweb.cern.ch/trac/lcgdm/wiki/Dpm` (19.02.2011)

[16] Janiszewski K.: *Rozproszone bazy danych w systemach gridowych*, Toruń 2010.

[17] Java Parallel Secure Socket (Stream), URL: `http://www.jlab.org/hpc/?id=jparss` (31.03.2011)

[18] iCAT attributes, URL: `https://irods.sdsc.edu/index.php/icatAttributes` (28.06.2011)

[19] Lemaitre S., Baud J.: DPM Administration for Tier2, URL: `http://www.uibk.ac.at/austriangrid/manuals/dpm-tier2-tutorial-15\_06\_2006.pdf` (21.02.2011)

[20] Lo Presti G., Barring O., Earl A., Garcia Rioja R.M., Ponce S., Taurelli G., Waldron D., Coelho Dos Santos M.: CASTOR: A Distributed Storage Resource Facility for High Performance Data Processing at CERN. *The 24th IEEE Conference on Mass Storage Systems and Technologies*, IEEE, 2007.

[21] Mkrtchyan T., Fuhrmann P., Gasthuber M.: Chimera — a new, fast, extensible and Grid enabled namespace service, URL: `http://www.dcache.org/manuals/chep06/Chimera-paper-chep06.pdf` (25.02.2011)

[22] New stager architecture and deployment, URL: `http://castor.web.cern.ch/castor/presentations/2005/External-Operation-Worshop-20050614/CASTOR2-overview-20050614.pdf` (28.02.2011)

[23] OGSA-DAI — About, URL: `http://www.ogsadai.org.uk/about/index.php` (28.06.2011)

[24] Open Grid Forum: *GridFTP v2 Protocol Description*, URL: `www.gridforum.org/documents/GFD.47.pdf` (27.06.2011)

[25] Open Grid Forum: *The Storage Resource Manager Interface Specification Version 2.2*, URL: `www.gridforum.org/documents/GFD.129.pdf` (27.06.2011)

[26] Ostropytskyy V.: Data Management in Chemomentum, URL: `ftp://ftp.cordis.europa.eu/pub/fp7/ict/docs/ssai/20070926-27-cm-d2-par4-data-management-chemomentum_en.pdf` (22.03.2011)

[27] Rękawek T., Bała P., Benedyczak K.: *Distributed Storage Management Service in UNICORE*, 2011.

[28] RFIO/IB project page, URL: `http://hikwww2.fzk.de/hik/orga/ges/infiniband/rfioib.html` (27.06.2011)

[29] Schmuck F., Haskin R.: GPFS: A Shared-Disk File System for Large Computing Clusters *Proceedings of the FAST'02 Conference on File and Storage Technologies*. Monterey, USA, 2002.

[30] The dCache Book 1.9.5, URL: `http://www.dcache.org/manuals/Book-1.9.5/Book-a4.pdf` (25.02.2011)

[31] UNICORE — Objectivces, URL: `http://www.unicore.eu/unicore/` (17.03.2011)

[32] UNICORE — Service Layer, URL: `http://www.unicore.eu/unicore/architecture/service-layer.php` (17.03.2011)

[33] UNICORE — XNJS IDB Configuration, URL: `http://www.unicore.eu/documentation/manuals/unicore6/unicorex/xnjs-idb.html` (27.06.2011)

[34] UNICORE Team: The Chemomentum Data Management System — Admin guide, 18.12.2008.

[35] Wikipedia — GLite, URL: `http://en.wikipedia.org/wiki/GLite` (19.02.2011)

[36] Wikipedia — Grid Computing, URL: `http://en.wikipedia.org/wiki/Grid_computing` (05.03.2011)

[37] XRootD, URL: `http://xrootd.slac.stanford.edu/` (27.06.2011)