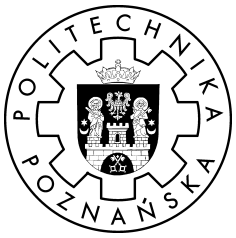
Politechnika Poznańska
Instytut Informatyki



Miłosz Kmieciak
Jan Palus
Marcin Szubert
Marek Zawirski

Rozproszone środowisko obliczeniowe oparte na bibliotece ProActive

Praca inżynierska

Promotor: dr inż. Cezary Sobaniec

Recenzent: doc. dr inż. Michał Sajkowski

Poznań, 2008

Spis treści

Rozdział 1
Wstęp

W ostatnich latach zaobserwować można znaczny wzrost zainteresowania obliczeniami rozproszonymi oraz ich zastosowaniami. Przyczyną takiego stanu rzeczy jest między innymi rosnąca wydajność oraz malejący koszt sieci komputerowych, stanowiących podstawę sprzętową rozproszonych środowisk obliczeniowych. Co więcej, przy utrzymaniu się obecnych trendów w postępie technologicznym, kiedy to wydajność procesorów, pamięci czy też sieci wciąż rośnie wykładniczo (prawo Moore'a [19]), z czasem koncepcje przetwarzania rozproszonego będą coraz powszechniej realizowane. Poza tym ostatnia dekada przyniosła dynamiczny rozwój wielu rozwiązań, które odmieniły sposób patrzenia na systemy rozproszone. Mowa tutaj m.in. o Internecie, systemie operacyjnym Linux, czy platformie Java. Wszystko to otwiera przed szeroko rozumianymi obliczeniami rozproszonymi bardzo wiele możliwości i pozwala na integrację zasobów rozprzestrzenionych fizycznie nawet po całym świecie.

Wraz z rozwojem rozległych sieci komputerowych (ang. *Wide-Area Networks* — WAN) obliczenia rozproszone i równoległe znacznie rozszerzyły swój zasięg, który wcześniej skoncentrowany był na rozwiązaniach opartych o pamięć współdzieloną i sieci lokalne. Doprowadziło to do wzrostu popularności tzw. *gridów* [6]. Grid jest rozproszonym środowiskiem obliczeniowym, łączącym w jedną infrastrukturę rozprzestrzenione geograficznie zasoby, zarządzane przez różne domeny administracyjne. Umożliwia wykorzystanie mocy obliczeniowej w sposób dynamiczny, w zależności od dostępności, wydajności, kosztów czy też wymagań użytkownika co do jakości usług (ang. *Quality of Service* — QoS). W porównaniu z gridami, gdzie komputery są całkowicie niezależnymi maszynami połączonymi siecią komputerową, wcześniejsze rozwiązania, takie jak superkomputery czy klastry są znacznie ściślej ze sobą związane (ang. *tightly-coupled*) oraz charakteryzują się bardziej homogenicznymi architekturami. Popularność rozwiązań ogólnie nazywanych gridami doprowadziła do wykształcenia pojęcia tzw. gridów lokalnych (ang. *local grids*), czyli środowisk o podobnej zasadzie działania, ale używanych w obrębie pojedynczej organizacji, uczelni czy też przedsiębiorstwa.

Dodatkowy czynnik wpływający na gwałtowny rozwój obliczeń rozproszonych to coraz trudniejsze problemy, przed którymi staje współczesna nauka. Pomimo tego, że dzisiejsze komputery osobiste są tak szybkie jak superkomputery kilkańcie lat temu, to i tak są zbyt wolne by zaspokoić stawiane im obecnie wymagania. Problemy, przed którymi stają naukowcy wiążą się głównie z olbrzymimi ilościami danych, jakie muszą przetwarzać oraz częstym powielaniem podobnych obliczeń w prowadzonych eksperymentach. Prowadzi to do poszukiwań alternatywnych rozwiązań zapewniających dodatkowe źródła mocy obliczeniowej. Takim rozwiązaniem jest właśnie przetwarzanie rozproszone. Pojęciem które opisuje wykorzystanie gridów dla celów nauki jest w ostatnim czasie *e-Science* [4]. Naukowcy zajmujący się różnymi dziedzinami wspierają inicjatywy na wielką skalę, mające na celu podjęcie próby rozwiązania skomplikowanych problemów, niejednokrotnie istotnych dla całej ludzkości (takie jak np. poszukiwanie lekarstwa na AIDS). Pomysły te zakładają wykorzystanie narzędzi rozproszonych po całym świecie, a mogących pracować razem, dzięki połączeniu wspólną infrastrukturą gridową. Nie chodzi tutaj tylko o zasoby komputerowe, ale również o urządzenia charakterystyczne dla danej dziedziny nauki. Przykładem może być chociażby unikalny na skalę światową akcelerator cząstek w szwajcarskiej instytucji CERN (LHC — Large Hadron Collider), udostępniony w ramach projektu LCG (LHC Computing Grid [13]).

Szybki rozwój gridów i ich rosnąca popularność ujawniła jednak sporo problemów jakie wiążą się zarówno z udostępnianiem, jak i z korzystaniem z takich środowisk. Niezbędny jest odpowiedni system zarządzający infrastrukturą gridową, którego zadaniami są między innymi kolejkowanie zgłaszanych zadań, przydzielanie współdzielonych zasobów zgodnie z preferencjami użytkownika czy też monitorowanie wykonanywania zadań oraz zmian w środowisku. Niezwykle ważną kwestią pozostaje bezpieczeństwo, zarówno z punktu widzenia osoby korzystającej z komputerów gridu, jak i spoglądając ze strony samych udostępnianych zasobów. Wszystko to wymaga zagwarantowania odpowiednich mechanizmów dostępowych, których wdrożenie utrudnione jest przez następujące kwestie:

- heterogeniczność infrastruktur wynikająca z różnorodności sprzętu i technologii stosowanych w różnych fragmentach środowiska rozproszonego;
- dostępność i wydajność poszczególnych maszyn zmienia się w czasie, środowiska mają dynamiczną naturę;
- niejednolity, utrudniony dostęp do zdalnych zasobów — potrzebny jest przyjazny interfejs umożliwiający stosunkowo łatwe wykorzystanie zasobów.

1.1 Cel i zakres pracy

Celem niniejszej pracy jest opracowanie koncepcji oraz implementacja systemu zarządzania lokalnym gridem obliczeniowym. Środowisko ma zostać oparte o istniejące zasoby laboratoriów Politechniki Poznańskiej. Ponieważ wykorzystywane komputery dedykowane są do celów dydaktycznych, system powinien działać na zasadzie „kradzieży cykli" (ang. *cycle-stealing*, *CPU-scavenging*), czyli wykorzystywać tzw. wolne cykle (ang. *free cycles*) dostępnych maszyn. Wymaga to szczególnego podejścia, ze względu na konieczność dynamicznego włączania i wyłączania poszczególnych hostów ze środowiska w zależności od aktualnej dostępności.

Dodatkowo udostępniony powinien zostać przyjazny interfejs umożliwiający monitorowanie trwających obliczeń oraz zarządzanie konfiguracją dla administratora. Cechą charakterystyczną lokalnych gridów jest ograniczenie dostępu dla dobrze zdefiniowanej grupy użytkowników — w tym przypadku są nimi pracownicy naukowi (oraz studenci) Politechniki Poznańskiej. Architektura platformy w założeniu jest heterogeniczna zarówno pod względem systemu operacyjnego, jak i samego sprzętu (np. wielordzeniowe procesory).

System powinien być optymalnie dostosowany do lokalnych potrzeb, zachowując jednakże uniwersalność umożliwiającą wdrożenie w podobnych środowiskach. Szczególny nacisk powinien zostać położony na możliwość zdalnego dostępu oraz łatwość wdrażania samodzielnie napisanych aplikacji. Jako warstwa pośrednia użyta ma zostać biblioteka ProActive umożliwiająca na poziomie programisty odpowiednią abstrakcję nad warstwą sprzętową.

1.2 Streszczenie rozdziałów

Struktura pracy jest następująca. **Rozdział drugi** stanowi wstęp teoretyczy wprowadzający w rozważane zagadnienia. Zostały tam opisane kwestie związane z przetwarzaniem rozproszonym i środowiskami gridowymi oraz główne koncepcje biblioteki ProActive. Przedstawiono również istniejące rozwiązania problemów takich jak systemy kolejkowe czy mechanizmy monitorowania.

Rozdział trzeci zawiera opis części projektowej rozwijanego systemu. Znaleźć tam można specyfikacje wymagań zawierającą m.in. funkcje, które system ma pełnić oraz model danych. Poza tym umieszczono tam zaproponowaną architekturę oraz typowy sposób wykorzystania systemu. Na zakończenie dokonano przeglądu fazy projektowania i innych propozycji rozwiązań, które brano pod uwagę na tym etapie.

Rozdział czwarty stanowi szczegółowy opis implementacji poszczególnych komponentów systemu. Dla każdego z nich przedstawiono w skrócie użyte technologie, ideę działania oraz szczegóły dotyczące architektury. Zwrócono przy tym uwagę na kwestię integracji danego komponentu z pozostałymi fragmentami systemu. W pewnych miejscach opisano także przeprowadzone testy zarówno jakościowe oraz porównujące dostępne rozwiązania pod kątem konkretnego sposobu użycia.

Rozdział piąty jest podsumowaniem wykonanej pracy z uwzględnieniem korzyści jakie system może przynieść oraz potencjalnych możliwości rozbudowy.

1.3 Podział zadań

Analiza wykorzystanych technologii i protokołów, jak i prace nad projektem systemu były wykonywane wspólnie, przez cały zespół. Jeżeli natomiast chodzi o implementację to obowiązywał następujący podział zadań:

- Miłosz Kmieciak — komunikacja zewnętrzna (Web Services)
- Jan Palus — aplikacja zgłaszająca zadania, aplikacja kontrolująca na węzłach
- Marcin Szubert — interfejs WWW, dostęp do bazy danych
- Marek Zawirski — centralna aplikacja zarządzająca

Wykonane zadania implementacyjne zostały następnie przedstawione przez odpowiadające za nie osoby w rozdziale 4 poświęconym szczegółom implementacji. Pozostałe części niniejszego opracowania mają następujących autorów:

1. *Wstęp* — Marcin Szubert
2. *Dziedzina problemu:*
   - 2.1. *Grid Computing* — Marcin Szubert
   - 2.2. *Systemy kolejkowe* — Marek Zawirski
   - 2.3. *Biblioteka ProActive* — Jan Palus
3. *Projekt systemu:*
   - 3.1 *Specyfikacja wymagań* — Marcin Szubert
   - 3.2 *Architektura systemu* — Miłosz Kmieciak
   - 3.3 *Typowy mechanizm użycia* — Miłosz Kmieciak
   - 3.4 *Alternatywne rozwiązania* — Miłosz Kmieciak
5. *Podsumowanie* — Marek Zawirski

Rozdział 2

Dziedzina problemu

2.1 Grid Computing

Przetwarzanie gridowe jest w ogólności usługą pozwalającą na korzystanie z dodatkowych zasobów komputerowych w postaci mocy obliczeniowej, przechowalni danych, zaawansowanych urządzeń itp. Zgodnie z założeniami idei „obliczeń na żądanie" (ang. *computing on demand*) ma to być usługa użyteczności publicznej, udostępniana na podobnej zasadzie jak np. prąd. Istnieje zresztą sporo analogii pomiędzy siecią elektryczną a gridem komputerowym. Dostęp do mocy obliczeniowej powinien być dla użytkownika transparentny, nie musi on wiedzieć jakie jest jej źródło i które węzły środowiska są przez niego wykorzystywane. Elektryczność jest w dzisiejszych czasach wszechobecna i takim też ma stać się grid computing. W niedalekiej przyszłości zasoby komputerowe w postaci usługi możliwej do wykupienia w razie potrzeby mają być dostępne na różnych platformach (również takich jak telefony komórkowe czy PDA) i w niezwykle prosty sposób (np. przez przeglądarkę internetową).

W związku z rosnącą popularnością takich systemów dokonano podziału gridów ze względu na cel jaki mają spełniać. Najbardziej popularne są tzw. gridy obliczeniowe (ang. computational grid), które mogą służyć między innymi do rozwiązywania skomplikowanych obliczeniowo problemów z różnych dziedzin nauki. Warto również wspomnieć o istnieniu tzw. gridów danych (ang. data grid) zapewniających dostęp do rozproszonych zbiorów informacji czy też gridów kontrolujących zaawansowane technicznie urządzenia (np. radioteleskopy) i przetwarzających produkowane przez nie dane (ang. equipment grid). Przykładem ostatniego może być wspomniany już wcześniej LCG [13].

2.1.1 Charakterystyka gridów

Kluczową cechą gridów jest *współdzielenie zasobów* na szeroką skalę. Dzięki temu grid daje dostęp do dodatkowej mocy obliczeniowej stwarzając tym samym iluzję potężnego superkomputera, który jest w stanie w rozsądnym czasie radzić sobie

z problemami, których rozwiązanie na pojedynczych maszynach zajęłoby całe lata. Zadaniem środowiska jest dostarczenie mechanizmów dostępowych umożliwiających czerpanie korzyści ze wspólnych zasobów. Stanowi to niejednokrotnie spore wyzwanie, ponieważ poszczególne części systemu mogą należeć do różnych osób lub instytucji. Pojawiają się więc problemy związane z różnicami w bezpieczeństwie oraz ograniczonym zaufaniem pomiędzy obcymi hostami. Nieco inaczej wygląda to w gridach lokalnych, gdzie wszystkie hosty należą do tej samej domeny administracyjnej. Wówczas poziom wzajemnego zaufania jest zazwyczaj stosunkowo wysoki, co zdecydowanie upraszcza rozwiązanie problemów związanych ze współdzieleniem zasobów.

Jednym z największych wyzwań przy projektowaniu środowiska obliczeń rozproszonych jest kwestia *bezpieczeństwa*. Należy zadbać między innymi o politykę dostępową, precyzującą kto i kiedy może korzystać ze współdzielonych zasobów oraz jaki dokładnie sprzęt znajduje się pod kontrolą gridu. Niezbędne jest także stosowanie uwierzytelniania i autoryzacji. System powinien być na tyle elastyczny, aby umożliwiać dynamiczną zmianę konfiguracji polityki bezpieczeństwa, jak i możliwość zarządzania kontami użytkowników w trakcie działania. Dodatkowym problemem jest poufność przesyłanych danych, co wymaga zagwarantowania mechanizmów szyfrowania.

Kolejną cechą jest *wydajne użytkowanie posiadanych zasobów*. Bez względu na to jak wielkimi zasobami dysponuje grid, mogą zdarzyć się sytuacje, gdy utworzy się kolejka użytkowników w oczekiwaniu na zwolnienie odpowiedniej ilości sprzętu. Bardzo ważne jest wtedy odpowiednie przydzielanie zgłaszanych zadań do dostępnych węzłów. Potrzebny jest więc system kolejkowy (który będzie brał pod uwagę m.in. oczekiwany czas obliczeń lub preferencje użytkownika) szeregujący przychodzące zadania. Ponieważ jest to dość powszechne zagadnienie związane z optymalnym przydziałem zasobów, istnieją gotowe rozwiązania radzące sobie z tym problemem.

Dzięki bardzo szybkiemu rozwojowi technologii sieciowych coraz mniejsze znaczenie ma odległość pomiędzy poszczególnymi fragmentami gridu. Nikogo nie powinno dziwić już to, że do obliczeń wykorzystywane są zasoby *rozprzestrzenione geograficznie* po całym świecie, co jeszcze kilkanaście lat temu było nie do pomyślenia. Oczywiście nadal występują pewne opóźnienia w komunikacji, jednak wraz z dalszym rozwojem i wzrostem prędkości rozległych sieci będą one miały coraz mniejsze znaczenie. Dodatkowo środowisko, dzięki zastosowaniu odpowiednich mechanizmów tolerancji błędów (ang. *fault-tolerance*), powinno być odporne na błędy związane z transmisją danych bądź też awarią sprzętu. Związane jest to z inną cechą charakterystyczną mianowicie *wysokimi wymaganiami co do jakości świadczonych usług* (ang. *quality of service*). Grid powinien udostępniać zasoby spełniające złożone wymagania użytkowników (dotyczą one parametrów takich jak dostępność, czas odpowiedzi, przepustowość itp.), dzięki czemu użyteczność całościowego systemu jest znacznie większa niż suma użyteczności jego części składowych. Każdy użytkownik dostaje bowiem dokładnie to, czego w danym momencie oczekuje.

Jedną z najważniejszych właściwości środowisk gridowych jest *wykorzystanie otwartych standardów i technologii* [6]. Gwarantuje to szybki rozwój uniwersalnych i przenośnych rozwiązań. Pojawia się jednak pytanie, kto powinien być odpowiedzialny za wybór, specyfikację i prace nad udoskonaleniem tych standardów. Jeżeli chodzi o protokoły internetowe, takie jak TCP/IP czy HTTP, od dawna istnieją już zajmujące się nimi organizacje IETF i W3C. Natomiast w przypadku standardów ściśle gridowych to są one rozwijane przez Global Grid Forum (GGF), zrzeszające kilka tysięcy pracowników naukowych i użytkowników zajmujących się i korzystających z gridów. Jednym ze standardów zdefiniowanych przez GGF, który wydaje się być kluczowy w kontekście najbliższej przyszłości jest OGSA (Open Grid Services Architecture).

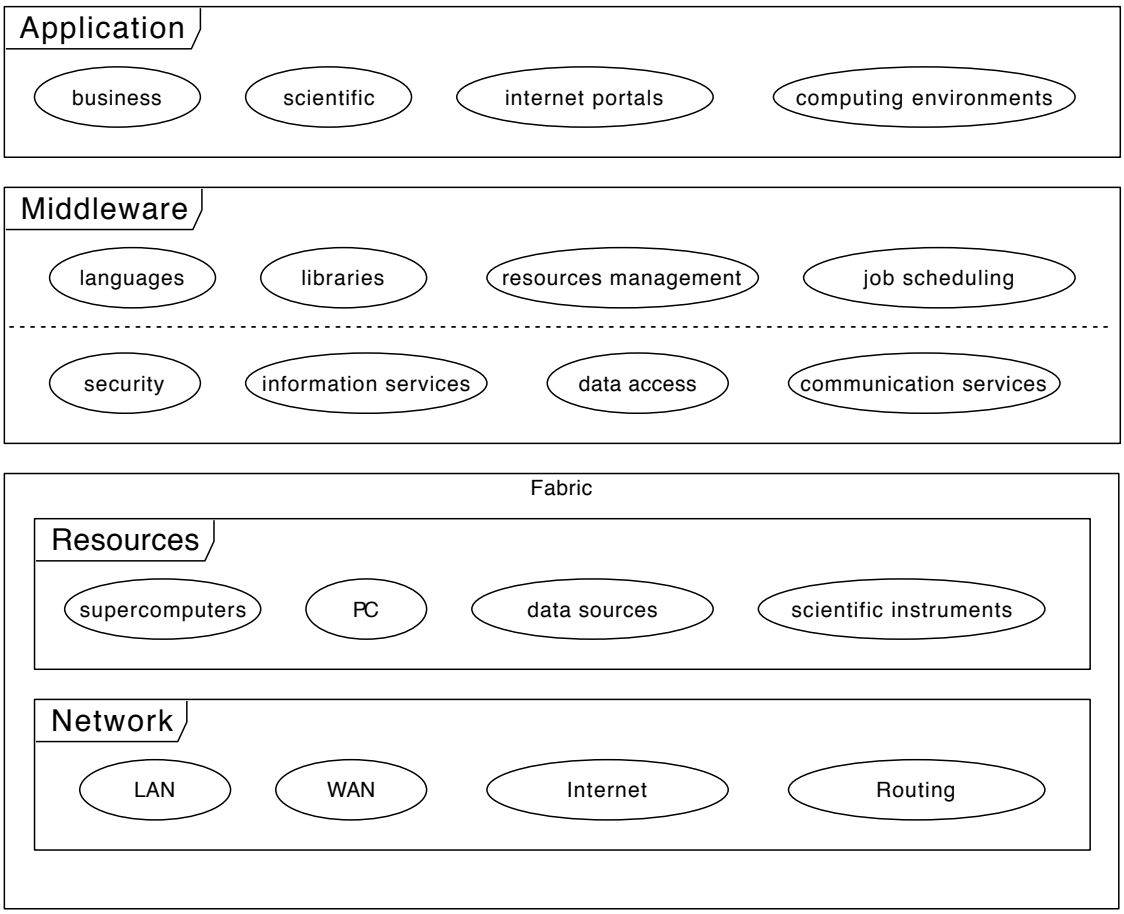
2.1.2 Architektura gridów

Zbudowanie środowiska gridowego wymaga uprzedniego zaprojektowania jego architektury, czyli wyspecyfikowania poszczególnych komponentów systemu oraz sposobów ich wzajemnej interakcji. Towarzyszy temu zazwyczaj podział modułów na warstwy. Nieodłączną częścią każdego gridu jest warstwa sprzętowa, jednak to co stanowi o użyteczności środowiska to oprogramowanie umożliwiające użytkownikowi dostęp do właściwych zasobów. Oprogramowanie takie nosi ogólną nazwę *middleware*, ze względu na to że stanowi warstwę pośrednią pomiędzy systemem operacyjnym bezpośrednio zarządzającym sprzętem a aplikacjami użytkownika, które mają zostać uruchomione w środowisku. Middleware ma na celu efektywne i niezawodne wykonywanie zleconych programów na odpowiednio wybranych komputerach, bez względu na ich fizyczne położenie w gridzie. Ukrywa ono przed użytkownikiem zbędne szczegóły dotyczące warstwy sprzętowej, ma za zadanie integrować niezależne zasoby w jedną spójną całość, sprawiając przy tym wrażenie przezroczystości. Powstało wiele projektów skoncentrowanych na rozwijaniu oprogramowania tego typu, jednym z nich jest ProActive.

2.1.2.1 Podział na warstwy

Architekturę gridów często przedstawia się w postaci warstw, z których każda pełni określone funkcje. Połączenie ich ze sobą tworzy kompletne środowisko. Ogólnie rzecz ujmując wyższe warstwy są związane z użytkownikiem i jego wymaganiami, podczas gdy niższe skoncentrowane są wokół sprzętu — komputerów i łączących ich sieci. Nieodłączną cechą każdego modelu warstwowego jest to, że każda warstwa udostępniając zdefiniowany zestaw usług opiera się na warstwie niższej. Architektura ta zilustrowana została na rysunku 2.1. Poniżej opisano rolę każdej z warstw:

- Warstwa sieciowa — łączy zasoby gridu. Jedną z głównych, wspomnianych już wcześniej cech gridów jest wykorzystanie możliwości jakie dają coraz szybsze połączenia pomiędzy oddalonymi od siebie zasobami. Niektóre testowe



Rys. 2.1: Warstwowa architektura gridów.

środowiska gridowe wykorzystują tzw. sieci wysokiej wydajności (ang. *high-performance networks*), których przepustowość sięga 10 Gbps. Pozwala to na efektywne zminimalizowanie opóźnień związanych z przesyłaniem danych. Coraz większe znaczenie zaczynają mieć także sieci bezprzewodowe, które pozwalają integrować urządzenia mobilne takie jak PDA, telefony komórkowe czy wbudowane procesory używane między innymi we współczesnych samochodach.

- Warstwa zasobów — właściwe zasoby udostępniane w gridzie. W ich skład wchodzą urządzenia takie jak: zasoby obliczeniowe, urządzenia przechowujące dane, zaawansowany sprzęt naukowy dostarczający informacji do przetwarzania (sieci sensoryczne, radioteleskopy). Zasoby obliczeniowe charakteryzują się często bardzo zróżnicowaną architekturą. W skład gridu mogą wchodzić zarówno dedykowane klastry procesorów, superkomputery, jak i zwykłe komputery osobiste działające pod kontrolą różnych systemów operacyjnych. Czasami do dwóch pierwszych warstw odwołuje się jako do jednej warstwy, nazywanej konstrukcyjną lub fizyczną.

- Warstwa middleware — dostarcza narzędzi pozwalających wszystkim urządzeniom niższej warstwy współistnieć w jednym zintegrowanym systemie. Ponieważ warstwa ta, jak sama nazwa wskazuje, znajduje się w środku modelu architektury, jej funkcje związane są z zapewnieniem możliwości współpracy wyższej warstwy — aplikacji z niższymi — sprzętowymi. Opiera się to na wykorzystaniu standardowych protokołów. W szczególności są to protokoły komunikacyjne, protokoły związane z bezpieczeństwem dostarczające mechanizmów weryfikacji tożsamości użytkowników oraz zasobów, protokoły informujące o strukturze i stanie środowiska oraz protokoły zarządzania zapewniające dostęp do wolnych zasobów czy szeregowanie zgłaszanych zadań. Oprogramowanie middleware pozwala na zdalne zarządzanie procesami, przydzielanie zasobów czy dostęp do danych. Dba o zagadnienia związane z jakością usług (QoS). Musi ono przy tym zapewniać abstrakcję od złożoności i heterogeniczności podlegających mu warstw niższych dostarczając spójny sposób odwoływania się do rozproszonych zasobów.

- Warstwa aplikacji i usług — zawiera aplikacje rozwijane przez użytkowników dla celów naukowych bądź też biznesowych. Jest to najwyższa warstwa architektury bezpośrednio związana z użytkownikami i jedyna, na którą mają oni faktyczny wpływ. Aplikacje użytkownika intensywnie wykorzystują funkcje udostępnione przez warstwę oprogramowania middleware. Pozwala im to uzyskiwać w bezpieczny sposób dostęp do pożądanych zasobów, zlecać na nich zadania polegające np. na pobieraniu danych lub przeprowadzeniu obliczeń, otrzymywać wyniki oraz monitorować ich postęp i występujące w trakcie pracy błędy.

2.2 Systemy kolejkowe

Systemy kolejkowe zadań (ang. *batch queing systems*) są najczęstszą formą udostępniania zasobów klastrów i gridów. Istnieje wiele produktów tego typu, popularne przykłady to SGE (*Sun Grid Engine*)[20] lub PBS (*Portable Batch System*)[37]. Rozwiązania middleware dla gridów często integrują w sobie systemy kolejkowe, jak jest np. w przypadku pakietu *Globus Toolkit*.

Podstawowe funkcje jakie spełniać mają takie systemy to przydział zasobów dla zadania i wykonanie go w obrębie tych zasobów. Zarządzane zasoby to w kontekście gridów obliczeniowych przede wszystkim procesory (moc obliczeniowa) i pamięć. Użytkownik zgłaszając zadanie do systemu zazwyczaj definiuje jego wymagania odnośnie zasobów, ewentualnie określa zależności od innych zadań i wskazuje aplikację do uruchomienia. Zasoby obliczeniowe najczęściej definiowane są przynajmniej przez liczbę procesorów i czas (ang. *walltime*) na który zarezerwowane będą węzły (procesory). W zależności od systemu może to być również szereg innych wymagań:

wybór kolejki dla zadania (dla systemów o wielu kolejkach), maksymalny rozmiar potrzebnej pamięci, maksymalny rozmiar pliku wymiany używany przez procesy itp. W nowoczesnych gridach używających mechanizmów *integrated services* dla zapewnienia jakości usług sieciowych (QoS), definiowane mogą być także wymagania dotyczące sieci. Wiele systemów zezwala również na definiowanie priorytetu zgłaszanego zadania i wymagań dotyczących transferu plików stanowiących często wejście i wyjście procesu obliczeniowego.

Zlecenie zadania w klasycznych systemach kolejkowych odbywa się przez aplikacje konsolową. Żądania dotyczące zasobów są w tym przypadku najczęściej określane przez zmienne środowiskowe lub argumenty przekazywane tej aplikacji. Nową tendencję stanowią wspomniane rozwiązania middleware lub systemy meta-kolejek udostępniające usługi związane z zarządzaniem zadaniami przez *API*.

Po zleceniu zadania trafia ono do określonej kolejki zadań. Kolejki te są definiowane przez administratora systemu i często wykorzystują złożone algorytmy do określania porządku wykonania zadań, np. w zależności od historii zgłoszeń użytkownika i stanu systemu. Ostatecznie, po opuszczeniu kolejki, aplikacja zadania jest wykonywana przez system na właściwych, przydzielonych węzłach. Niektóre systemy używają do tego celu swoich własnych protokołów, inne popularnych protokołów jak SSH lub RSH.

Istotną funkcją oczekiwaną wobec systemu kolejkowego jest monitorowanie stanu podlegających mu węzłów. Aby poprawnie i efektywnie przydzielać zasoby trzeba opierać się na możliwie aktualnej wiedzy o stanie systemu. Taka wiedza jest możliwa do utrzymania przez stosowanie regularnego aktywnego monitorowania. Wiele produktów wykorzystuje do tego własne protokoły, niektóre protokoły ogólnego zastosowania jak SNMP.

Standardowa dystrybucja biblioteki ProActive (opisana szerzej w punkcie 2.3) umożliwia wykorzystanie wielu istniejących systemów kolejkowych, choć powiązanie między tymi systemami a biblioteką nie jest silne (np. nie są wykorzystywane mechanizmy kontroli zadań w trakcie wykonania, powiązania między zadaniami). Możliwe jest również dołączenie własnego systemu kolejkowego. Wykorzystanie systemów kolejkowych przez używane wydanie ProActive'a (wersja 3.3) sprowadza się zazwyczaj do jednorazowej interakcji biblioteki z systemem — najczęściej przez zlecenie zadania na początku działania aplikacji. Co odróżnia ProActive'a od starszych bibliotek wykorzystywanych w gridach obliczeniowych, to fakt, że jej architektura wydaje się jednak nastawiona na obliczenia z zachowaniem interakcji z użytkownikiem wykonującym obliczenia — m.in. poprzez możliwość łatwego wdrożenia aplikacji z węzła nie znajdującego się w gridzie.

2.3 Biblioteka ProActive

Biblioteka ProActive [43] jest biblioteką warstwy middleware napisaną w języku Java przeznaczoną do tworzenia aplikacji równoległych, rozproszonych oraz współbieżnych. Dzięki zastosowanym koncepcjom tworzenie takich aplikacji stało się niezwykle proste i tylko w minimalnym stopniu różni się od tworzenia zwykłych aplikacji w języku Java. Wyraźne rozdzielenie warstwy aplikacji od warstwy fizycznej gridu oraz ograniczeniu do minimum ich interakcji z punktu widzenia programisty sprawia, że może on się skupić na funkcjonalności samej aplikacji pozostawiając ustalenie szczegółów odnośnie docelowego gridu jego administratorowi. Pomimo faktu, że biblioteka ProActive jest napisana w języku Java uchodzącym wciąż wśród wielu programistów za mało efektywną platformę, to jednak w rzeczywistych zastosowaniach biblioteka okazuje się niezwykle efektywna i bardzo skalowalna nawet w przypadku rozlokowania aplikacji na 5000 węzłów rozmieszczonych w różnych częściach świata. Co więcej biblioteka ta zawiera szereg dodatkowych możliwości takich jak wbudowana obsługa wielu popularnych systemów kolejkowych, rozszerzenia ułatwiające implementację algorytmów typu branch-and-bound oraz mechanizmy odporności na błędy.

Poniżej zaprezentowane zostaną pokrótce podstawowe koncepcje związane z biblioteką ProActive wyróżniające ją wśród innych rozwiązań tego typu. Najważniejszym elementem całej biblioteki jest pojęcie *aktywnego obiektu* (ang. *Active Object*). Aktywnym obiektem może stać się dowolny obiekt dowolnego typu stworzony przez użytkownika. Podstawową jego cechą jest podział na dwie części — część eksponującą interfejs obiektu użytkownika (tzw. *stub*) komunikującą się z częścią zawierającą właściwe ciało obiektu (tzw. *body*). Z obiektem body skojarzony jest jeden wątek w wirtualnej maszynie Javy znajdującej się na dowolnym węźle w strukturze gridu. Jednak dla programisty proces podziału na te dwie części jest zupełnie przezroczysty dzięki zaawansowanym metodom dziedziczenia i refleksji zastosowanym w bibliotece ProActive. Z punktu widzenia programisty w kodzie programu widoczny jest jedynie obiekt o żądanym przez niego typie. Należy zdawać sobie sprawę, że aktywny obiekt jest podstawową jednostką rozproszenia przetwarzania.

Kolejną własnością biblioteki ProActive bezpośrednio związaną z opisywanym mechanizmem aktywnego obiektu jest sposób wywołania metod takiego obiektu odpowiadający mechanizmowi przekazywania komunikatów w tradycyjnych bibliotekach przeznaczonych do tworzenia aplikacji rozproszonych. Dzięki rozdzieleniu funkcji obiektu na części stub i body możliwe stało się asynchroniczne wykonanie metod. Aplikacja wywołująca metodę aktywnego obiektu może kontynuować działanie nawet pomimo faktu, że wykonanie tej metody w rzeczywistości jeszcze się nie zakończyło. W przypadku metod zwracających wartości w rezultacie zwracany jest natychmiast tzw. obiekt *Future* (o ile tylko typem zwracanym nie jest klasa okraszona modyfikatorem final w przypadku którego wywołanie jest synchroniczne). Podobnie jak w przypadku aktywnego obiektu, obiekt Future eksponuje taki sam

interfejs jak typ obiektu zwracanego — jego użycie, z punktu widzenia programisty, jest zupełnie przezroczyste. Jeżeli w programie zostanie użyty obiekt „opakowany" w obiekt Future to o ile nie został dostarczony jeszcze rezultat wywołania przez część body aktywnego obiektu, dalsze przetwarzanie programu zostanie zablokowane do momentu jego uzyskania. Mechanizm odroczenia uzyskania rezultatu wykonania metody w celu uzyskania maksymalnej współbieżności przetwarzania nazwany został w bibliotece ProActive terminem *wait-by-necessity*.

Bardzo ważną cechą biblioteki ProActive jest rozdzielenie warstwy aplikacyjnej od fizycznej struktury grida. Dzięki temu możliwe jest uruchomienie dowolnej aplikacji opartej o tę bibliotekę na dowolnym gridzie bez potrzeby jakichkolwiek modyfikacji kodu programu. Jedynym połączeniem między aplikacją użytkownika a gridem na którym jest ona uruchamiana jest tzw. *deskryptor XML* (ang. *XML descriptor*). Jest to przygotowywany przez administratora gridu plik XML o ściśle określonej składni. Jedyne informacje, jakie użytkownik musi posiadać o deskryptorze XML to jego ścieżka dostępu. W trakcie działania aplikacji programista pobiera z deskryptora kolejne węzły abstrahując od tego w jaki sposób są one przydzielane. Plik deskryptora definiuje pojęcie *wirtualnego węzła* (ang. *virtual node*) stanowiące abstrakcję nad węzłami fizycznymi. Wewnątrz wirtualnego węzła natomiast zdefiniowana jest hierarchia tzw. *procesów* opisujących jednoznacznie sposób rozlokowania aplikacji. W praktyce jej zadaniem jest w ogólności stworzenie komendy, której uruchomienie spowoduje rozlokowanie aplikacji użytkownika. Zadaniem każdego z pojedynczych procesów jest zatem „doklejenie" na koniec komendy części odpowiadającej danemu procesowi przy czym zwykle na końcu tej hierarchii znajduje się proces odpowiedzialny za wystartowanie środowiska uruchomieniowego biblioteki ProActive w wirtualnej maszynie Javy. Prostym przykładem może być tu hierarchia procesów w której na najwyższym poziomie znajduje się proces wykonania komendy poprzez protokół SSH, następnie występuje proces odpowiadający za zgłoszenie wykonania komendy w dowolnym systemie kolejkowym, a na końcu wspominany proces startujący wirtualną maszynę Javy. W takim wypadku efektem przetwarzania jest przykładowa komenda o postaci: `ssh remotemachine /usr/bin/submit /usr/bin/java ProActive`.

Rozdział 3

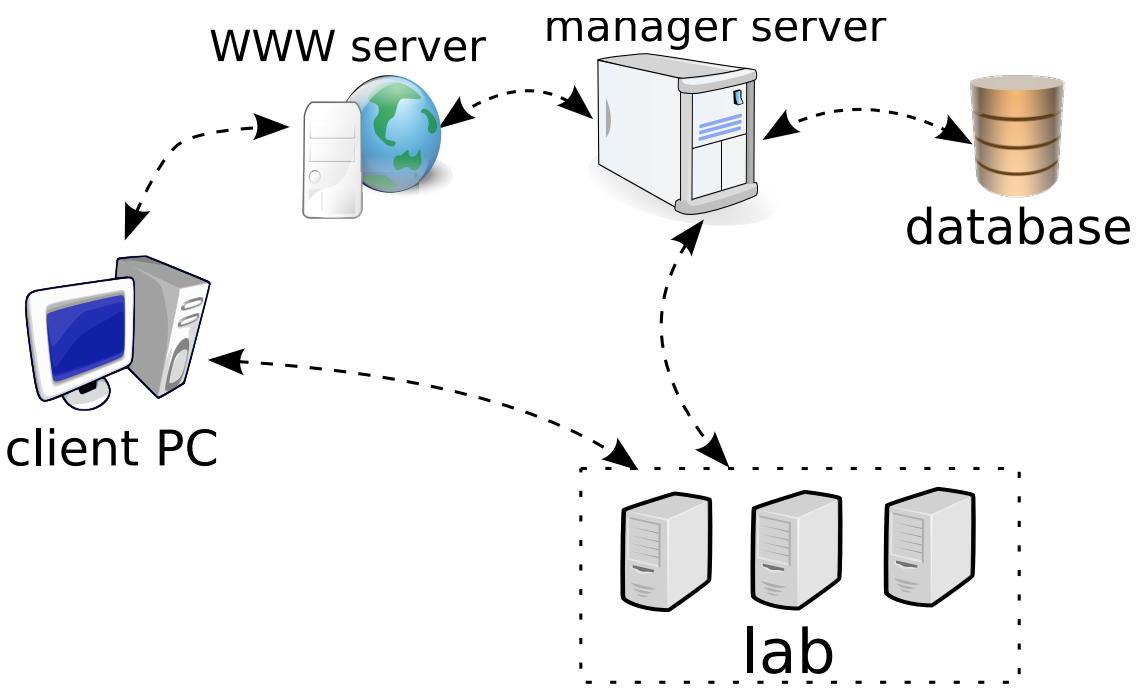
Projekt systemu

3.1 Specyfikacja wymagań

3.1.1 Ogólny opis systemu

Zadaniem systemu jest zapewnienie jednolitego dostępu do lokalnego gridu obliczeniowego opartego na istniejących zasobach sprzętowych. Umożliwić ma to uruchamianie aplikacji rozproszonych na wybranych węzłach oraz nadzorowanie ich wykonywania. Zgodnie z architekturą tego typu środowisk przedstawioną w części teoretycznej (rysunek 2.1), opisywany produkt umiejscowiony może być w dolnej części warstwy middleware. Górną cześć tej warstwy stanowi z kolei biblioteka ProActive.

Zapotrzebowanie na tego typu produkt wynika z wielu potencjalnych zastosowań lokalnego środowiska obliczeniowego (ze strony pracowników naukowych) oraz brak uniwersalnego oprogramowania zapewniającego pożądaną elastyczność. Elastyczność ta związana jest głównie z dynamicznie zmieniającą się dostępnością zasobów spowodowaną dedykowaniem ich w większości celom dydaktycznym. Pomimo tych ograniczeń, w obliczu niewykorzystanej mocy obliczeniowej laboratoriów, powstały do tej pory pewne rozwiązania, podejmujące próby ułatwienia wykonywania obliczeń rozproszonych. Charakteryzują się one jednak bardzo ograniczonym zastosowaniem. Oparte są w szczególności na krótkich programach w językach skryptowych uruchamiających identyczne polecenie na wielu węzłach. Wymaga to ręcznego utworzenia listy komputerów, sprawdzania ich stanu oraz zbierania wyników. Szczególnie niewygodna wydaje się być konieczność unikania jednoczesnego uruchamiania skryptów, a w konsekwencji uzgadnianie wykorzystania komputerów z innymi oczekującymi. Poza tym niezbędne było także pozostawienie komputerów włączonych na czas obliczeń, co może być utrudnione w wypadku zajeć dydaktycznych odbywających się w danym laboratorium. Zaprojektowany system ma za zadanie sprostać tym niedogodnościom.



Rys. 3.1: Interakcje między głównymi komponentami systemu.

3.1.2 Użytkownicy

Administrator. Uprzywilejowany użytkownik odpowiedzialny za zarządzanie konfiguracją środowiska oraz utrzymywanie systemu po jego wdrożeniu. Powinien posiadać on dodatkowe uprawnienia jeśli chodzi o operacje wykonywanie na zgłoszonych zadaniach. Konfiguracja obejmuje między innymi warstwę sprzętową, konta użytkowników mogących korzystać z systemu oraz ustawienia zarządcy zasobów.

Użytkownik zlecający obliczenia — autor aplikacji rozproszonej. Użytkownik dla którego tworzony jest system. Są to głównie pracownicy naukowi oraz studenci. W założeniu autor tworzy aplikację rozproszoną przy użyciu biblioteki ProActive, a następnie przy pomocy dostarczonych mechanizmów może ją w prosty sposób wdrożyć. Poza tym powinien posiadać możliwość monitorowania stanu środowiska, w tym głównie zgłoszonych zadań.

3.1.3 Komponenty systemu

System ma działać w oparciu o architekturę wielowarstwową. Poszczególne komponenty ilustruje rysunek 3.1, gdzie zaznaczono również zachodzące między nimi interakcje. Poniżej wyróżnione zostały wymagania co do każdego z komponentów — w szczególności dotyczące interfejsów (komunikacyjnych, programowych, sprzętowych) oraz, bardzo ogólnie, realizowanych funkcji:

- **Komputer klienta** — dostarcza autorowi aplikacji możliwości wdrożenia jej w systemie (na komputerach laboratoriów) oraz graficzny interfejs użytkownika w postaci strony WWW. Niezbędna jest obecność przeglądarki internetowej ze wsparciem języka JavaScript. Aplikacje rozproszone rozwijane są z założenia przy użyciu biblioteki ProActive, więc do ich tworzenia potrzebna jest instalacja Javy oraz sama biblioteka.

- **Serwer WWW** — w szczególności kontener serwletów (ang. *web container*, *servlet container*). Zapewnia pełen interfejs usług dla klienta. Komunikacja między klientem a serwerem odbywać ma się z wykorzystaniem protokołu HTTP.

- **Serwer Managera** — główny komponent systemu, stanowi rolę łącznika pomiędzy różnymi fragmentami systemu. Dostarcza interfejs dostępowy do bazy danych oraz interfejs pozwalający zarządzać zadaniami oraz maszynami w laboratoriach (wykonywać na nich polecenia, sprawdzać ich stan). Aplikacja zarządcy napisana ma zostać w języku Java.

- **Baza danych** — wszystkie dane dotyczące konfiguracji środowiska i jego aktualnego stanu przechowywane są w zewnętrznej bazie danych. Warto podkreślić, iż nie nałożono żadnych ograniczeń dotyczących dystrybucji systemu zarządzania bazą danych.

- **Laboratoria** — komputery w laboratoriach uczelnianych (w tym przypadku są to laboratoria Instytutu Informatyki Politechniki Poznańskiej); stanowią na co dzień miejsce zajęć dydaktycznych dla studentów. Nie ma specyficznych wymagań co do homogeniczności ich architektury sprzętowej, systemu operacyjnego, ani też ich fizycznego umiejscowienia. Powinny udostępniać swój stan zarządcy, jak również umożliwiać wykonywanie zdalnych poleceń. Odbywać się to powinno przy użyciu standardowych, otwartych protokołów. Ponieważ wdrażane aplikacje są z założenia napisane w języku Java, powinna istnieć możliwość ich uruchomienia na węzłach środowiska.

3.1.4 Funkcje systemu — diagram przypadków użycia

Główne funkcje systemu prezentuje diagram przypadków użycia na rysunku 3.2. Poszczególne przypadki użycia są szczegółowo wyspecyfikowane w rozdziale 3.1.5.

Dodatkowo należy wspomnieć, że korzystanie z jakichkolwiek funkcji systemu, przedstawionych schematycznie poniżej, wymaga posiadania konta o odpowiednich uprawnieniach. Wszystkie przypadki użycia powinny być więc poprzedzone uwierzytelnianiem (realizowanym np. poprzez logowanie) oraz autoryzacją (system weryfikuje uprawnienia użytkownika przy każdej wykonywanej przez niego operacji). Nie uwzględniono tego ograniczenia na poniższym diagramie w celu zachowania dostatecznej przejrzystości.

Rys. 3.2: Diagram przypadków użycia.

Warto zwrócić uwagę, iż aktor *administrator* stanowi specjalizację aktora *użytkownik*, posiada więc z założenia takie same możliwości jak on, a te widoczne na powyższym diagramie stanowią ich rozszerzenie, czyli funkcje dostępne tylko dla administratora.

Większość wyszczególnionych tutaj przypadków użycia skoncentrowanych jest głównie na użytkownikach i wykonywanych przez nich działaniach. Kluczowe funkcje systemu natomiast, związane z utrzymaniem środowiska, realizowane są automatycznie, bez interakcji z użytkownikiem. Zostały one opisane w osobnym podrozdziale 3.1.5.5.

3.1.5 Wymagania funkcjonalne

3.1.5.1 Logowanie

UC0: Logowanie do interfejsu użytkownika

Główny scenariusz:

1. Użytkownik włącza aplikację.
2. Użytkownik wprowadza dane uwierzytelniające — swoją unikalną nazwę oraz hasło.
3. System weryfikuje poprawność wprowadzonych danych na podstawie znanej listy wszystkich utworzonych kont w systemie — następuje uwierzytelnianie.
4. System wyświetla interfejs odpowiadający roli logującego się użytkownika i jej uprawnieniom.

Rozszerzenia:

2.A. Administrator nie podał wszystkich niezbędnych do zalogowania informacji.

2.A.1. System prosi o wprowadzenie brakujących informacji.

3.A. Dane wprowadzone przez użytkownika nie są poprawne — zostało wprowadzone błędne hasło lub nieistniejąca nazwa użytkownika.

3.A.1. System wyświetla komunikat o błędzie oraz daje użytkownikowi szanse ponownego wprowadzenia informacji uwierzytelniających.

3.1.5.2 Administracja kontami użytkownika

UC1: Dodanie konta użytkownika

Główny scenariusz:

1. Administrator wybiera za pomocą GUI odpowiedni panel pozwalający na dodawanie użytkowników.
2. Administrator wypełnia formularz z danymi użytkownika; podaje jego nazwę, hasło oraz rolę nowego użytkownika w systemie — administrator lub autor.
3. System tworzy nowe konto, zgodne z podanymi informacjami.

Rozszerzenia:

2.A. Administrator nie podał wszystkich informacji o nowym koncie.

2.A.1. System prosi o wprowadzenie brakujących informacji

UC2: Usunięcie konta użytkownika

Główny scenariusz:

1. Administrator wybiera za pomocą GUI odpowiedni panel wyświetlający listę wszystkich kont użytkowników.
2. Administrator znajduje i zaznacza na liście konto użytkownika, które chce usunąć.
3. System prosi o potwierdzenie usunięcia a następnie, w przypadku potwierdzenia, usuwa konto.

Rozszerzenia:

3.A. Administrator nie potwierdza usunięcia wybranego konta.

3.A.1. System anuluje usuwanie konta, pozostawiając konta użytkowników bez zmian.

UC3: Edycja konta użytkownika

Główny scenariusz:

1. Administrator wybiera za pomocą GUI odpowiedni panel wyświetlający listę wszystkich kont użytkowników.
2. Administrator znajduje i zaznacza na liście konto użytkownika, które chce zmienić.
3. System wyświetla formularz z dotychczasowymi danymi dotyczącymi konta.

4. Administrator modyfikuje informacje o koncie, a następnie zapisuje je.

Rozszerzenia:

4.A. Administrator usuwa dotychczasowe informacje i nie wprowadza nowych.

4.A.1. System prosi o wprowadzenie brakujących informacji.

UC4: Zmiana hasła użytkownika

Główny scenariusz:

1. Dowolny użytkownik wybiera za pomocą GUI odpowiedni panel z informacjami o swoim koncie.
2. Użytkownik decyduje się zmienić swoje dotychczasowe hasło, wprowadza dwukrotnie nowe hasło oraz potwierdza zmiany.
3. System zmienia hasło dla danego użytkownika, co będzie uwzględnione przy kolejnym logowaniu.

Rozszerzenia:

2.A. Podane przez użytkownika hasła nie są identyczne.

2.A.1. System prosi o ponowne wprowadzenie hasła.

3.1.5.3 Operacje na pojedynczych węzłach

UC5: Zdalne włączenie/wyłączenie komputera

Główny scenariusz:

1. Administrator wybiera przy użyciu GUI panel prezentujący dostępne komputery podzielone na grupy.
2. Administrator wybiera grupę i komputer oraz zleca na nim wykonanie polecenia włączenia/wyłączenia. Niemożliwe jest przy tym wykonanie polecenia włączenia/wyłączenia w chwili gdy komputer już jest włączony/wyłączony.
3. System podejmuje próbę wykonania polecenia, uwzględniając przy tym aktualny stan wybranego komputera, w szczególności fakt czy komputer jest zarządzany przez system oraz czy ktoś na nim pracuje lokalnie.

Rozszerzenia:

2.A. Możliwe jest także wybranie komputera spośród komputerów związanych z wykonaniem konkretnego zadania, poprzez użycie panelu z listą zadań.

UC6: Włączenie/wyłączenie zarządzania komputerem

Główny scenariusz:

1. Administrator wybiera panel prezentujący dostępne komputery podzielone na grupy.
2. Administrator wybiera grupę, a następnie komputer oraz zleca na nim wykonanie polecenia włączenia/wyłączenia zarządzania. Niemożliwe jest przy tym wykonanie polecenia włączenia/wyłączenia zarządzania w chwili gdy komputer jest już zarządzany/niezarządzany.
3. System podejmuje próbę wykonania polecenia, uwzględniając przy tym aktualny stan wybranego komputera, w szczególności fakt czy komputer jest włączony.

Rozszerzenia:

2.A. Możliwe jest także wybranie komputera spośród komputerów związanych z wykonaniem konkretnego zadania, poprzez użycie panelu z listą zadań.

UC7: Monitorowanie stanu komputera

Główny scenariusz:

1. Użytkownik wybiera panel z listą komputerów.
2. System wyświetlając listę używa następujących kolorów dla symbolicznego oznaczenia stanu węzła:
   - szary — komputer nie jest zarządzany (wszystkie inne kolory oznaczają, że komputer znajduje się pod zarządzaniem systemu)
   - zielony — komputer dostępny dla obliczeń
   - żółty — komputer niedostępny dla obliczeń
   - szaro-zielony — komputer jest dostępny, ale jest wyłączony
   - szaro-żółty — komputer jest niedostępny, jest wyłączony i nie może zostać włączony
   - niebieski — stan nieznany, oczekiwanie na pierwszą odpowiedź od węzła
   - czerwony — błąd w pobieraniu stanu węzła
3. System wyświetla także przy poszczególnych komputerach na liście ich adresy IP, które pozwalają je fizycznie zidentyfikować oraz identyfikatory zadań na nich uruchomionych.
4. Użytkownik wybiera interesujący go komputer i zaznacza go

5. System wyświetla dodatkowe, szczegółowe informacje dotyczące danego komputera, zarówno dotyczące konfiguracji jak i jego stanu — wynikające z ostatniej odpowiedzi.

Rozszerzenia:

1.A. Możliwe jest monitorowanie stanu poszczególnych węzłów środowiska zarówno przez panel ze wszystkim komputerami, jak i za pomocą panelu z komputerami przeznaczonymi dla konkretnego zadania.

2.A. Użytkownik wyświetla dany panel z listą komputerów przez okres dłuższy niż 30 sekund.

2.A.1 System automatycznie odświeża zawartość i wygląd listy komputerów, opierając się na najświeższych informacjach o stanie hosta.

3.1.5.4 Operacje na zadaniach

UC8: Zgłoszenie zadania

Główny scenariusz:

1. Użytkownik implementuje aplikację rozproszoną, która ma stanowić zadanie obliczeniowe dla środowiska rozproszonego.
2. Użytkownik specyfikuje pewne właściwości zgłaszanego zadania, m.in. jego nazwę, minimalną i pożądaną liczbę maszyn oraz preferowane grupy maszyn (np. takie, co do których użytkownik jest pewien, że nie będą lokalnie używane).
3. Użytkownik zgłasza zadanie przy użyciu specjalnie przeznaczonej do tego aplikacji dostarczanej razem z systemem i stanowiącej jego fragment.
4. System zapamiętuje informacje o zgłoszonym zadaniu.
5. System, na podstawie wymagań użytkownika, znajduje węzły, na których zadanie może zostać wdrożone.
6. Zadanie zostaje uruchomione na węzłach środowiska. W trakcie obliczeń system pełni funkcje monitorujące i zarządzające wykonianiem zadania.
7. Użytkownik może śledzić wykonanie swojej aplikacji, która ostatecznie zwraca wyniki w sposób zależny od jej implementacji (wyświetlenie na terminalu, zapisanie do pliku).

Rozszerzenia:

3.A. Podane przez użytkownika parametry zadania są niepoprawne.

3.A.1. System weryfikuje poprawność argumentów i w przypadku wykrycia jakichkolwiek niezgodności (np. minimalna liczba maszyn podana

przez użytkownika jest większa niż całkowita liczba maszyn w systemie) przerywa proces zgłaszania i zwraca użytkownikowi informację o błędzie.

5.A. Nie ma wystarczającej liczby wolnych maszyn.

5.A.1. System kolejkuje zadanie w oczekiwaniu na zakończenie trwających obliczeń lub udostępnienie dodatkowych komputerów.

6.A. Wystąpiło zdarzenie mogące wpłynąć na przerwanie zadania — np. ktoś zalogował się lokalnie na jedną z maszyn grupy, wewnątrz której uruchomione jest zadanie.

6.A.1. Działanie aplikacji może zostać przerwane, jednak zależy to od konfiguracji środowiska oraz od wymagań zgłaszającego. Autor może chcieć przerwać zadanie przy jakimkolwiek błędzie, bądź też może sam zapewnić mechanizmy tolerancji błędów.

Jest to główny przypadek użycia z punktu widzenia użytkownika systemu, dlatego został on szczegółowo opisany w rozdziale 3.3. Poszczególne fazy zgłaszania zadań zostały tam zobrazowane w postaci rysunków.

UC9: Anulowanie zgłoszonego zadania

Główny scenariusz:

1. Użytkownik wybiera za pomocą GUI odpowiedni panel z listą wszystkich zgłoszonych do tej pory zadań.
2. Użytkownik znajduje na liście swoje zadanie (oczekujące w kolejce lub aktualnie wykonywane) i zaznacza je do anulowania.
3. System kończy wykonanie zadania na komputerach oraz zapisuje nowe informacje związane z zakończeniem zadania.

Rozszerzenia:

2.A. Użytkownik nie ma prawa do anulowania zadania.

2.A.1. Administrator może anulować dowolne zadanie. Autor aplikacji może anulować jedynie swoje zadanie. W przypadku naruszenia tych reguł system zignoruje polecenie anulowania.

UC10: Wyszukiwanie zgłoszonych zadań

Główny scenariusz:

1. Użytkownik wybiera za pomocą GUI odpowiedni panel z listą wszystkich zgłoszonych do tej pory zadań.
2. Użytkownik wybiera kryteria wyszukiwania spośród następujących:

- właściciel zadania: nazwa użytkownika będącego autorem wyszukiwanego zadania
- stan zadania: zadanie uruchomione, zakończone, zakolejkowane, przygotowane do uruchomienia
- czas zgłoszenia zadania: przedział czasowy zaznaczany na kalendarzu

3. Użytkownik potwierdza zakończenie specyfikacji poszczególnych kryteriów wyszukiwania.
4. System wyszukuje zadania spełniające kryteria oraz prezentuje je w postaci listy.

UC11: Przeglądanie szczegółów zadań

Główny scenariusz:

1. Użytkownik wybiera panel z listą zadań (może to być pełna lista, bądź już odfiltrowana, po zastosowaniu wyszukiwania).
2. Użytkownik wybiera z listy interesujące go zadanie i je zaznacza.
3. System wyświetla szczegółowe informacje o zadaniu, zarówno te podane przez użytkownika przy zgłaszaniu, jak i te uzupełnione przez system. Szczególnie istotną informacją jest lista hostów na których zadanie zostało uruchomione, z możliwością sprawdzenia szczegółów dotyczących danego komputera.

UC12: Zatrzymanie wykonania zadania na danym komputerze

Główny scenariusz:

1. Administrator przegląda listę zadań (możliwe, że już odfiltrowanych) i związanych z nimi hostów lub też listę wszystkich hostów.
2. Administrator znajduje interesujący go komputer na liście oraz wyświetla jego szczgóły.
3. Administrator wybiera z listy rozwijanej wszystkich zadań uruchomionych na danym komputerze interesujące go zadanie oraz potwierdza swój wybór — wydaje polecenie zatrzymania wykonania.
4. System zatrzymuje wykonanie konkretnego zadania na wybranym komputerze, co powoduje potencjalne zwolnienie maszyny dla innych zadań lub udostępnienie większej mocy obliczeniowej innym uruchomionym na danym komputerze zadaniom. W konsekwencji uruchomienie zadania może zostać automatycznie przerwane na wszystkich maszynach.

3.1.5.5 Zarządzanie środowiskiem

Jak już zostało wspomniane, pewne funkcje systemu, stanowiące podstawę jego poprawnego działania, wykonywane są niezależnie od działań użytkowników i bez ich wiedzy. Wiążą się one z nieustannym monitorowaniem zarządzanych zasobów i podejmowaniem na podstawie zachodzących zmian określonych działań. Działania te mają na celu zapewnienie wymaganych cech systemu, w tym głównie bezkonfliktowej współpracy lokalnych użytkowników komputerów z użytkownikami systemu. Osoby pracujące lokalnie nie powinny odczuwać, że używany przez nie sprzęt jest częścią gridu obliczeniowego i zarządzany jest przez zewnętrzny system.

Oczywiste jest, że tego typu funkcji systemu nie da opisać się przy pomocy przypadków użycia i sekwencyjnych scenariuszy. Funkcje te oparte są bowiem na zbiorze reguł i determinujących je zdarzeń oraz mają charakter ciągły. Poszczególne reguły działania opisują jak system ma reagować na takie zdarzenia jak np. lokalne zalogowanie się użytkownika. Specyfikują one kiedy na komputerze można uruchomić zadanie, tzn. kiedy można uznać dany węzeł za dostępny do obliczeń uwzględniając jego aktualny stan. Decydują również jak długo system może czekać na odpowiedź od węzła oraz co należy zrobić w przypadku przekroczenia limitu czasu. Wszystkie reguły powstawały wskutek rozpatrzenia możliwych kombinacji stanów i zdarzeń jakie mogą dotyczyć systemu, autora aplikacji rozproszonej oraz lokalnych użytkowników.

Precyzyjny opis reguł działania w ściśle określonych sytuacjach znaleźć można w punktach 4.3.3.2 i 4.3.3.3.

3.1.6 Model danych

W bazie danych systemu przechowywane są na bieżąco utrwalane informacje dotyczące stanu i konfiguracji systemu. Aby to umożliwić zaprojektowano utworzenie następujących relacji:

- **Job** — reprezentuje zadanie zgłaszane do systemu przez użytkownika. Niektóre z atrybutów dostarczane są przez zgłaszającego, pozostałe uzupełnia system. Krotki tworzone są podczas zgłoszenia zadania natomiast modyfikowane są na bieżąco w wyniku zmiany stanu zadań. Najważniejsze atrybuty zadania to:
  - *JobName* — nazwa zadania, służy do identyfikacji zadania poprzez przyjazną użytkownikowi nazwę
  - *MinMachines* — minimalna liczba maszyn niezbędna do uruchomienia zadania
  - *RequestedMachines* — żądana liczba maszyn, na których zadanie powinno zostać uruchomione według użytkownika

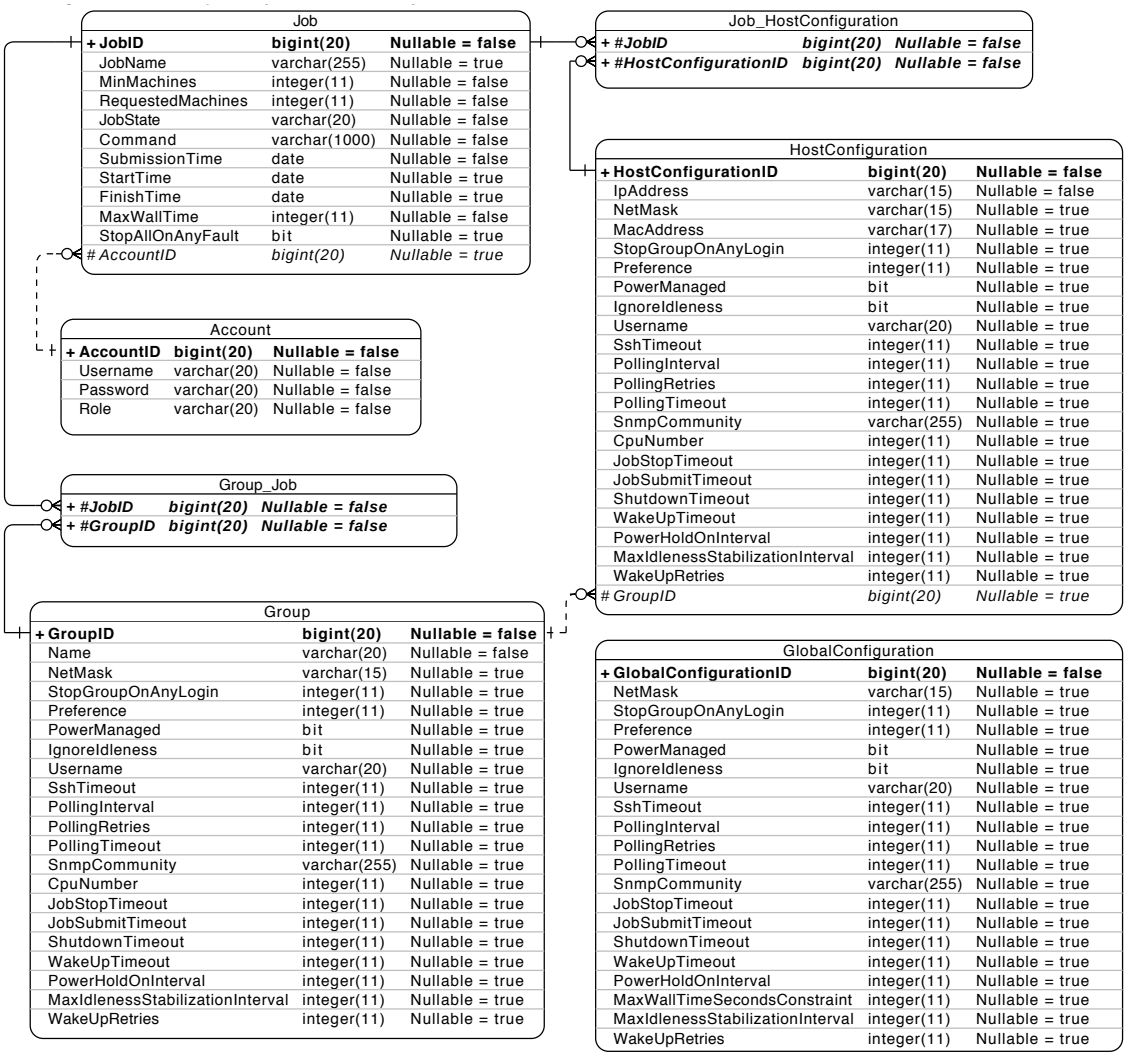
  - *JobState* — stan zadania, może przyjmować jedną z następujących czterech wartości: zakolejkowane, przygotowywane do uruchomienia, uruchomione oraz zakończone lub przerwane.
  - *Command* — dokładne polecenie uruchomienia zadania
  - *SubmissionTime, StartTime, FinishTime* — uzupełniane przez system czasy zgłoszenia, uruchomienia oraz zakończenia zadania
  - *StopAllOnAnyFault* — specyfikuje czy zadanie powinno zostać całkowicie przerwane w wyniku jakiegokolwiek błędu na choćby jednej z maszyn, na których zostało uruchomione

- **Account** — reprezentuje konta użytkowników środowiska. Konta są tworzone, modyfikowane i usuwane przez administratora. Dodatkowo każdy użytkownik może w dowolnym momencie zmienić swoje hasło, co również odpowiada modyfikacji odpowiedniego wiersza w bazie danych. Jedynym atrybutem tej tabeli, który może wymagać objaśnienia jest *Role* — specyfikuje on rolę użytkownika, a co się z tym wiąże jego uprawnienia. Atrybut może przyjmować wartości: administrator, użytkownik oraz aplikacja zgłaszająca zadania.

- **GlobalConfiguration** — jest jedną z trzech pozostałych relacji opisujących konfigurację środowiska, a przez to bardziej statycznych niż dwie poprzednie. Wartości w tych tabelach nie są modyfikowane podczas działania systemu. Relacje zawierają ogólne parametry dotyczące systemu (takie jak np. limity czasowe) i mają charakter hierarchiczny. *HostConfiguration* stanowi najniższy poziom w hierarchi i dotyczy pojedynczego komputera. Gdy jakikolwiek z obowiązkowych atrybutów nie został dla danego hosta wyspecyfikowany, jest on dziedziczony z krotki tabeli *Group* odpowiadającej grupie tego hosta. Niewypełnione pola w tej ostatniej tabeli przepisywane są z kolei z najwyższej w hierarchi relacji *GlobalConfiguration*, która zawiera część wspólną atrybutów tabel konfiguracyjnych (*Group* i *HostConfiguration* zawierają pewne dodatkowe własności) w tym następujące:

  - *StopGroupOnAnyLogin* — specyfikuje czy wszystkie hosty w ramach danej grupy powinny wstrzymać wykonywanie zadań w przypadku lokalnego logowania użytkownika na dowolny z komputerów tej grupy
  - *PowerManaged* — oznacza zarządzanie zasilaniem komputerów, tzn. stwierdza czy dany komputer ma być zdalnie włączany/wyłączany przez system
  - *Preference* — opisuje preferencję wykonania zadania na tym hoście; jeśli dostępnych jest kilka hostów, zadanie zostanie uruchomione na tym o najwyższej wartości tego atrybutu.

  Część atrybutów określających m.in. limity czasowe i przerwy między zapytaniami o stan hostów, zostanie opisana szczegółowo w części implementacyjnej.



Rys. 3.3: Diagram związków encji.

- **HostConfiguration** — reprezentuje konfiguracje pojedynczego węzła środowiska. Poza atrybutami wspólnymi dla tabel przechowujących konfigurację relacja ta zawiera także atrybut *IpAddress* odzwierciedlający adres sieciowy komputera.

- **Group** — reprezentuje grupę komputerów (w tym przypadku może to odpowiadać np. jednemu laboratorium, które składa się z wielu komputerów); jedynym specyficznym atrybutem jest nazwa grupy.

Wszystkie atrybuty opisanych tabel oraz powiązania między nimi obrazuje diagram związków encji widoczny na rysunku 3.3. Pomiędzy relacjami zachodzą następujące związki:

- *Account – Job* — asocjacja typu 1:M; każde zadanie ma swojego właściciela, który musi mieć utworzone konto w bazie danych, jeden użytkownik może być autorem wielu zadań; właściciel zadania ma co do niego specjalne prawa, których nie posiadają pozostali użytkownicy (z wyjątkiem administratora);
- *Group – Job* — asocjacja typu M:N; każde zadanie może mieć wyspecyfikowane grupy hostów, na których uruchomienie tego zadania powinno być preferowane; użytkownik zgłaszający może posiadać wiedzę na temat dostępności poszczególnych grup i na tej podstawie przedstawia swoje preferencje;
- *HostConfiguration – Job* — asocjacja typu M:N; po zgłoszeniu zadania i spełnieniu wymagań co do dostępności minimalnej liczby węzłów, zadanie powiązane jest z hostami, do których zostało przydzielone; na podstawie tego związku wiadomo również które zadania są uruchomione lub przygotowywane do uruchomienia na danym komputerze;
- *Group – HostConfiguration* — asocjacja typu 1:M; komputery powiązane są w grupy; jeden komputer może być częścią tylko jednej grupy; na podstawie zgrupowania komputerów można zarządzać w pewnych przypadkach nie tyle pojedynczymi węzłami co całymi ich grupami; grupy posiadają właściwości, stanowiące wartości domyślne dla odpowiadających atrybutów w hostach należących do danej grupy;

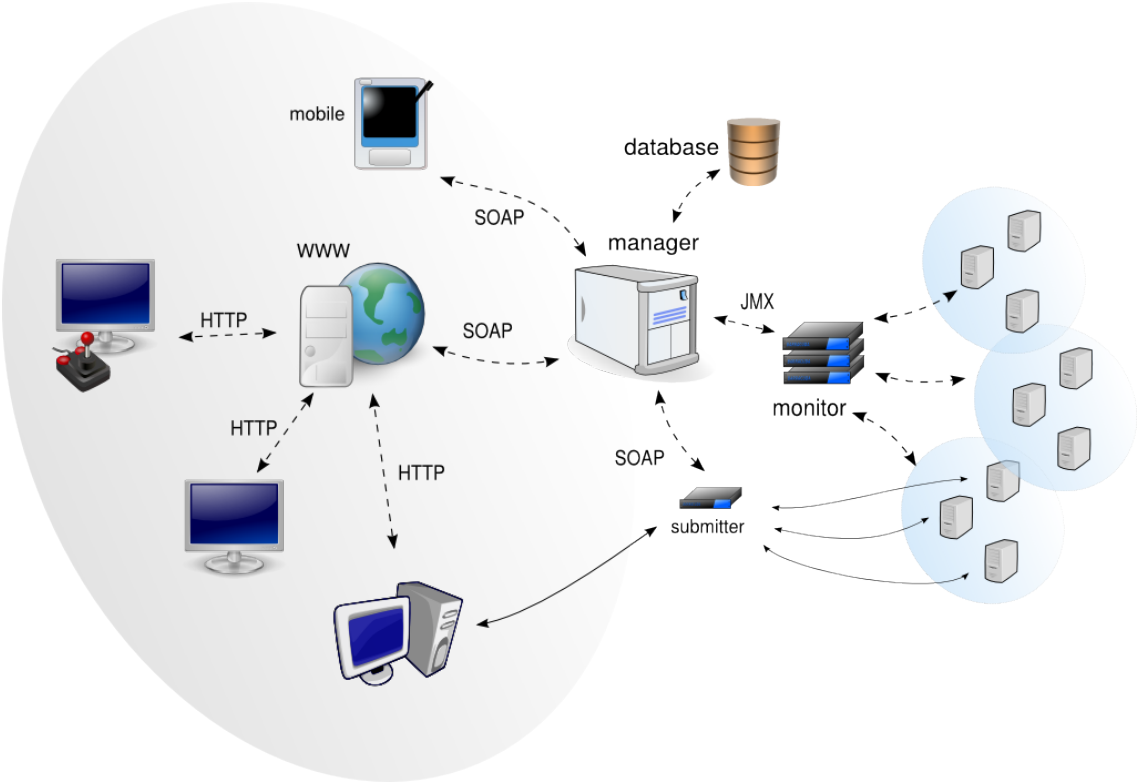
3.2 Architektura systemu

Mając na uwadze postawione cele oraz ograniczenia opisane w poprzednim podpunkcie, powstał zarys architektury systemu. Należy zaznaczyć, iż przedstawiony poniżej obraz systemu jest ostatecznym wynikiem dłuższego procesu analitycznego. Szczegółowy opis ewolucji oraz dojrzewania poszczególnych koncepcji wykracza, natomiast, poza ramy niniejszego opracowania i nie jest jego przedmiotem (por. 3.4 *Alternatywne rozwiązania*).

Architektura systemu stara się z jednej strony zaadresować wszystkie wymagania (por. 3.1), a także wpasować się w ogólnie przyjęte kanony projektowania. Zgodnie z podpunktem 2.1.2 *Architektura gridów* uzasadnione jest wyróżnienie możliwie niezależnych komponentów systemu, a także dopasowanie ich do poszczególnych warstw gridu. W poniższych podpunktach wyszczególniono moduły systemu, a także opisano umiejscowienie proponowanego rozwiązania w kontekście dostępnego środowiska.

3.2.1 Podział na moduły

Wielomodułowość omawianego systemu, czyli podział na poszczególne składowe, jest bezpośrednim następstwem przyjętych wymogów oraz założeń. Odpowiedni



Rys. 3.4: Diagram integracji składowych systemu.

dobór realizowanych zadań, technologie ich wykonania oraz szczególne założenia pozafunkcjonalne zostały zawarte w niniejszym podpunkcie. Ogólny schemat integracji komponentów wchodzących w skład omawianego systemu przedstawiono na rysunku 3.4.

3.2.1.1 Centralna aplikacja zarządzająca

Implementację głównych mechanizmów systemu zawarto w centralnej aplikacji zarządzającej. To w jej gestii leży szeroko rozumiane zarządzanie stanem monitorowanego systemu oraz zasobami wchodzącymi w jego skład. Realizuje również wymianę informacji z aplikacjami klienckimi oraz wykonuje czynności zlecane przez użytkowników. Ponadto, jest punktem centralnym przedstawianego systemu, przez co w sposób naturalny integruje większość jego komponentów.

Sama aplikacja w założeniach jest uruchamiana na serwerze, którego dostępność oraz niezawodność przekłada się na ogólną wiarygodność systemu. Zważywszy na fakt, iż to w tym miejscu udostępniana jest cała funkcjonalność, serwer powinien być elementem ogólnie dostępnej infrastruktury sieciowej, w szczególności sieci Internet. Z oczywistych względów wymagane jest również połączenie z zarządzanymi zasobami.

Przegląd modułów wchodzących w skład omawianej aplikacji należy zacząć od modułu dostępu do danych. Stanowi on punkt wyjścia dla kolejnych komponentów,

gdyż dostarcza niezbędnych informacji wymaganych do realizacji większości zadań związanych z zarządzaniem rozproszonym środowiskiem obliczeniowym. Wykonanie niniejszego komponentu oparto o bazę danych, do której zapewniono elastyczny dostęp poprzez technologię JPA (ang. *Java Persistence API*). Pozwala ona odzwierciedlić strukturę relacji bazy danych w modelu obiektowym języka Java (por. podpunkt 4.3.1). Jako system zarządzania bazą danych przyjęto dystrybucję MySQL, jednakże przewidziano możliwość jego zmiany.

Zrealizowanie funkcjonalności monitorowania oraz wykonywania zdalnych operacji na zasobach zostało oddelegowane do modułu o nazwie Monitor (zob. podpunkt 4.3.2). Implementacja oparła się o integrację wielu protokołów, wśród których wymienić należy SNMP (ang. *Simple Network Management Protocol*), służący do monitorowania węzłów oraz SSH (ang. *Secure Shell*), przez który są wykonywane polecenia na zdalnych maszynach. Z kolei w celu włączania komputerów poprzez sieć wykorzystano standard WOL (ang. *Wake On Lan*). oodatkowo, w celu podwyższenia kohezji omawianego komponentu, wykorzystano mechanizm JMX (ang. *Java Management Extensions*). Udostępnia on ujednolicony interfejs komunikacji z pozostałymi modułami centralnej aplikacji zarządzającej, a także pozwala na rozproszenie realizowanej funkcjonalności. Cecha ta przekłada się bezpośrednio na możliwość skalowalności opisywanego systemu.

Opisany wyżej komponent jest podstawą do utrzymywania w aplikacji obrazu stanu zarządzanego środowiska. Ten z kolei jest wykorzystywany przez moduł zarządzający węzłami, tzw. Disposer (zob. podpunkt 4.3.3). Wraz z danymi pozyskanymi z bazy danych możliwe jest w pełni zautomatyzowane sterowanie środowiskiem oraz reagowanie na zachodzące w nim zmiany. Ponadto, moduł ten obsługuje jawne żądania administratora, a także odbiera informacje z modułu integrującego system z biblioteką do obliczeń rozproszonych. Warto podkreślić, że moduł ten ma również za zadanie umożliwienie prowadzenia zajęć dydaktycznych w ścisłym otoczeniu zarządzanych węzłów.

Kontakt ze „światem zewnętrznym" zapewniono poprzez zastosowanie technologii usług sieciowych (zob. podpunkt 4.2 oraz 4.3.4). Zaletą podjętej decyzji jest elastyczność rozwiązania oraz możliwość jego wzbogacenia o dodatkowe moduły.

3.2.1.2 Komponenty po stronie węzłów obliczeniowych

Do kolejnej grupy omawianych komponentów systemu należą wszelkie aplikacje uruchamiane na komputerach będących przedmiotem zarządzania. Są one bezpośrednio zaangażowane w proces realizacji funkcjonalności monitorowania poprzez przygotowanie środowiska uruchomieniowego, zarejestrowanie w systemie operacyjnym odpowiednich usług oraz wykonywania czynności sterujących na rzecz centralnego systemu zarządzającego.

Wśród omawianych modułów należy wskazać aplikację o nazwie Agent (zob. 4.1), udostępniającą stan węzła, z której korzysta wspomniany wcześniej Monitor. Istotny jest także zestaw narzędzi przygotowanych z myślą o sterowaniu systemem operacyj-

nym zarządzanego komputera. Do funkcji tego drugiego, modułu Putgridctl, należy m.in. pielęgnowanie środowiska uruchomieniowego oraz wyłączanie jednostki.

Warto podkreślić, że wymienione wyżej moduły są instalowane na każdym komputerze, który ma być zarządzany.

3.2.1.3 Aplikacja zlecająca wykonanie zadania w systemie

Potrzeba posiadania wyszczególnionego narzędzia zlecającego zadania wynika bezpośrednio z filozofii zgłaszania zadań do systemów kolejkowych ogólnego dostępu. Mianowicie, we wspomnianym procesie wyróżnia się funkcjonalny moduł odpowiedzialny za prawidłowe zarejestrowanie zadania w środowisku oraz ewentualne jego uruchomienie na przydzielonych węzłach. Dokładnie takie znaczenie posiada aplikacja zgłaszająca zadania — Submitter (zob. 4.4).

Komunikuje się ona z centralną aplikacją zarządzającą przy użyciu dedykowanych usług sieciowych (por. 4.3.4). Zgłasza żądanie wykonania zadania wraz z parametrami je charakteryzującymi, po czym oczekuje na przydział zasobów obliczeniowych.

Ważną kwestią, wynikającą z cech biblioteki ProActive, jest możliwość interaktywnego wykonywania zadań, to jest rozproszonych aplikacji, których wykonanie zlecił użytkownik. W związku z czym, aplikacja zgłaszająca zadania, musi zadbać o zestawienie odpowiedniego połączenia sieciowego pomiędzy przydzielonymi zasobami, a źródłem żądania. Wykorzystano w tym celu protokół SSH.

3.2.1.4 Integracja z biblioteką ProActive

Integracja z biblioteką dla obliczeń rozproszonych, ProActive (zob. podpunkt 2.3) wynika bezpośrednio z założeń opisywanego systemu i polega na dostarczeniu niezbędnych mechanizmów rozszerzających funkcjonalność biblioteki w taki sposób, aby możliwe było zlecenie zadania w opisywanym systemie z poziomu bilbioteki ProActive, wykorzystując do tego celu opisany już wcześniej moduł Submittera.

3.2.1.5 Aplikacje po stronie klienta

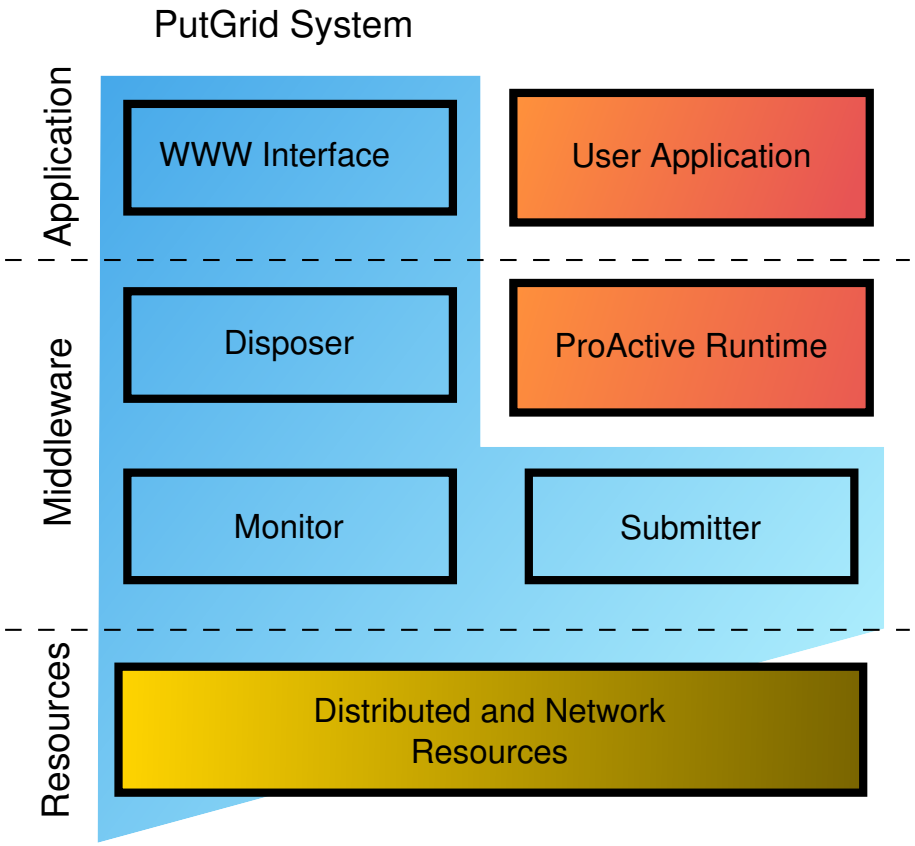
Ostatnią grupą komponentów prezentowanego systemu są aplikacje klienckie udostępniające funkcjonalność systemu w postaci dedykowanych interfejsów. Funkcję takiego interfejsu spełnia aplikacja WWW, w skład której wchodzi infrastruktura serwera oraz technologie uruchamiane po stronie klienta — przeglądarce internetowej (por. 4.6). Należy w tym miejscu również podkreślić, iż z uwagi na zastosowanie technologii usług sieciowych, istnieje możliwość dodania kolejnych aplikacji klienckich, adresujących konkretne potrzeby użytkowników. Mogą to być zautomatyzowane narzędzia administracyjne przeznaczone dla środowiska Unix/Linux, bądź też aplikacje mobilne.

Wspomniana aplikacja WWW (zob. 4.6.1) została w pełni zaprojektowana oraz wykonana w oparciu o architekturę bogatej aplikacji internetowej (ang. *Rich Inter-*

net Application). Również i tym razem zdecydowano się na użycie nowoczesnych technologii. W roli serwera wykorzystano dystrybucję Apache Tomcat, natomiast wykonanie samej implementacji oparto o środowisko rozwojowe GWT (ang. *Google Web Toolkit*).

3.2.2 Kompozycja warstw systemu

Analiza wymagań postawionych systemowi, a także kontekst jego zastosowań, pozwala wyszczególnić warstwy, w których system będzie obecny. Co więcej, przeprowadzony podział pozwala scharakteryzować pełnione przez system role, jak rownież możliwą interakcję w poszczególnych przypadkach użycia (por. rysunek 3.5). Należy w tym miejscu zaznaczyć, iż przedstawiony schemat zawiera pewne uproszczenia i prezentuje jedynie wybrane komponenty. Celowo pominięto w nim kwestie technologiczne.



Rys. 3.5: Poglądowa kompozycja warstw systemu w kontekście środowiska.

Spoglądając z perspektywy najniższej warstwy, omawiany system powinien całkowicie współpracować z zarządzanymi zasobami. W ich skład wchodzą heterogeniczne węzły obliczeniowe, przykładowo jednostki centralne dostępne w laboratoriach, a także infrastruktura sieciowa. W związku z powyższym, przygotowano

specjalne moduły przeznaczone do instalacji na zarządzanych jednostkach, umożliwiające włączenie konkretnych zasobów w skład systemu.

Wyższa warstwa zapewnia izolację pomiędzy zarządzanymi zasobami, a usługami warstw kolejnych, dostarczając jednocześnie odpowiedniej funkcjonalności. Tym samym wchodzi już w skład logicznej całości zwanej oprogramowaniem middleware. Można w niej wyróżnić submittera, o który oparto integrację z biblioteką ProActive, a także moduły centralnej aplikacji zarządzającej, przykładowo Disposer oraz Monitor. Te ostatnie realizują główny scenariusz przewidziany w systemie — nadzorowanie oraz zarządzanie zasobami.

Na najwyższym poziomie — aplikacyjnym, znajdują się wszystkie komponenty działające na styku użytkownik — system. Mogą one dostarczać funkcjonalności związanej z zarządzaniem oraz monitorowaniem zasobów, bądź też realizować obliczenia rozproszone. W pierwszej grupie wymienić należy interfejs WWW, natomiast do drugiej należą m.in. wszystkie aplikacje rozwijane w oparciu o bibliotekę ProActive, dostarczane oraz zlecane przez użytkownika końcowego (por. 3.1.2 *Użytkownicy*).

3.3 Typowy mechanizm działania

Z założenia, główną funkcją realizowaną przez system jest obsługa zgłaszanych zadań. W niniejszym punkcie przedstawiono projekt systemu w aspekcie jego realizacji. W sposób ogólny omówiony został typowy mechanizm działania, zilustrowany odpowiednimi schematami (rys. 3.6, 3.7, 3.8). Poszczególne etapy odnoszą się bezpośrednio do przypadku użycia *UC8* z podpunktu 3.1.5.4.

3.3.1 Etapy zgłoszenia oraz uruchomienia zadania

Etap 1: Submitter zgłasza zadanie do systemu (rys. 3.6).

1.A Uruchomienie przez użytkownika aplikacji opartej o bibliotekę ProActive.

1.B Na podstawie dołączonego deskryptora następuje rozpoczęcie procesu uruchamiania aplikacji w środowisku rozproszonym.

1.C Poprzez protokół SSH zostaje zestawione bezpieczne połączenie z maszyną dostępową, na której został zainstalowany Submitter, aplikacja zgłaszająca zadania.

1.D Następuje uruchomienie Submittera z odpowiednimi parametrami żądania.

1.E Poprzez protokół SOAP submitter wykorzystuje usługi sieciowe centralnej aplikacji zarządzającej do zgłoszenia żądania.

Etap 2: System podejmuje decyzje dotyczące przydziału węzłów (rys. 3.7).

2.A Centralna aplikacja zarządzająca posiada obraz stanu środowiska rozproszonego.

2.A.1 Monitor na bieżąco uaktualnia wiedzę modułu zarządzającego węzłami (pośrednio, poprzez mechanizm JMX).

2.A.2 Wiedza utrzymywana jest dzięki mechanizmowi powiadomień protokołu SNMP.

2.B Na podstawie stanu środowiska oraz parametrów żądania zostaje podjęta decyzja o przydziale węzłów. Może również nastąpić jej odroczenie.

2.B.1 W razie potrzeby zostają włączone odpowiednie węzły obliczeniowe poprzez protokół WOL.

Etap 3: Przydzielone węzły biorą udział w obliczeniach (rys. 3.8).

3.A Uzgodniony przydział węzłów zostaje przekazany oczekującemu Submitterowi.

3.B Submitter poprzez protokół SSH kolejno uruchamia aplikację rozproszoną na wyznaczonych węzłach.

3.B.1 Uruchomiona aplikacja kontaktuje się z komputerem źródłowym poprzez odpowiednio wyznaczony protokół, standardowo poprzez Java RMI.

3.C Submitter nie zamyka połączeń SSH, dopóki aplikacja rozproszona nie zostanie zakończona.

3.C.1 Następuje powiązanie deskryptorów wejściowych oraz wyjściowych pomiędzy utrzymywanymi strumieniami.

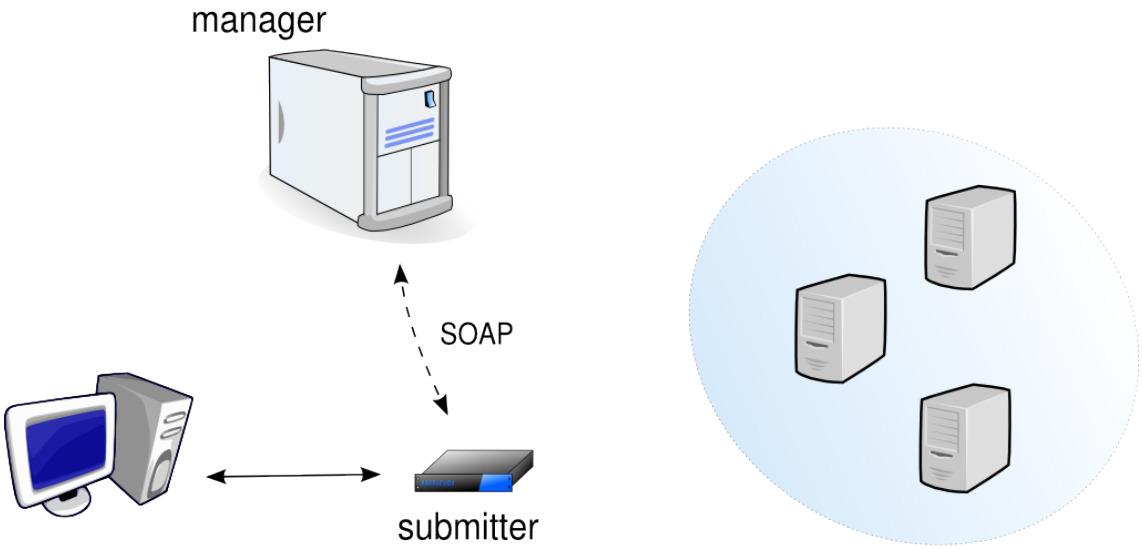
3.C.2 Możliwa jest interakcja z użytkownikiem zlecającym obliczenia.

3.D Centralna aplikacja zarządzająca monitoruje przebieg obliczeń.

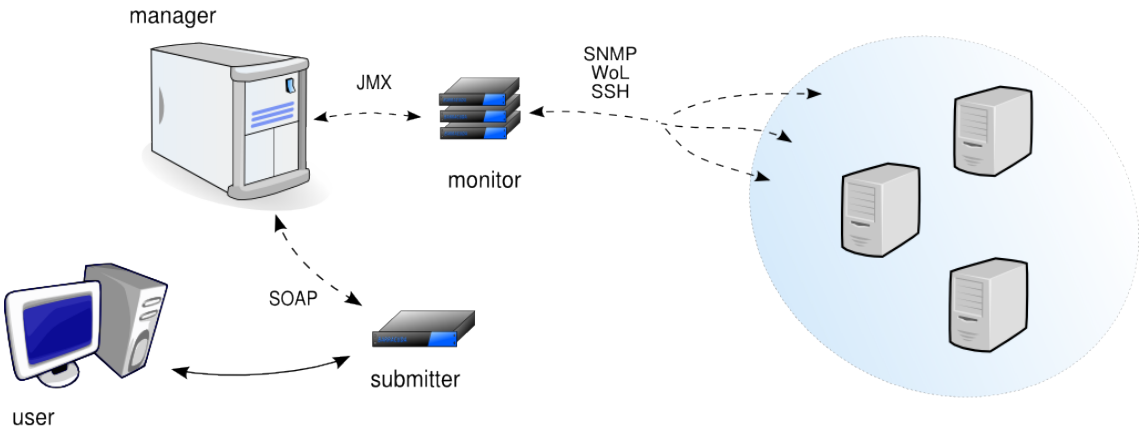
3.D.1 W razie niepowodzenia, środowisko uruchomieniowe na poszczególnych węzłach jest czyszczone.

3.D.2 W przypadku wykrycia aktywności akademickiej, następuje przerwanie obliczeń w ramach grupy węzłów. W zależności od parametrów żądania, może ono zostać przerwane całkowicie.

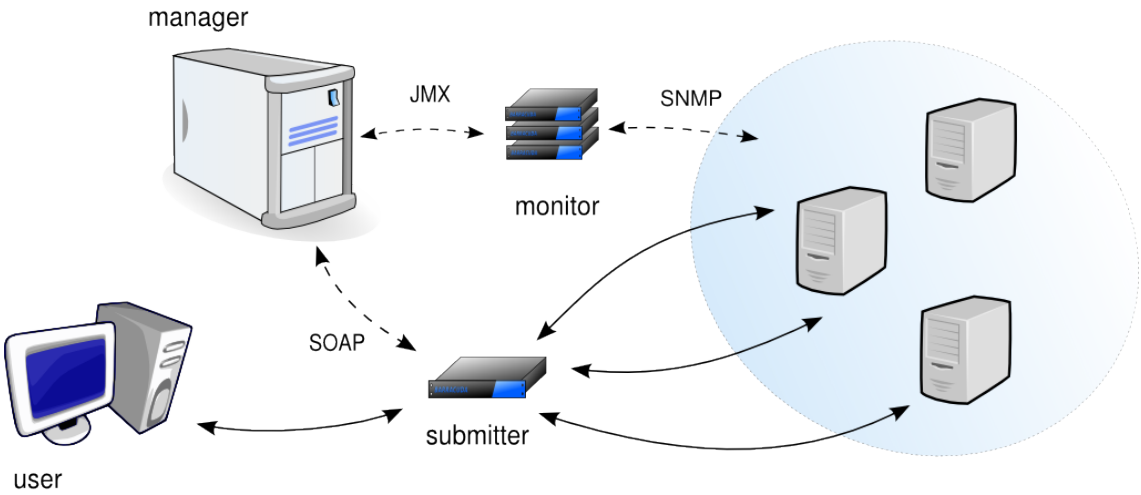
3.D.3 Po zakończonych obliczeniach może nastąpić wyłączenie, uprzednio włączonych, węzłów obliczeniowych.



Rys. 3.6: Etap 1: Submitter zgłasza zadanie do systemu.



Rys. 3.7: Etap 2: System podejmuje decyzje dotyczące przydziału węzłów.



Rys. 3.8: Etap 3: Przydzielone węzły biorą udział w obliczeniach.

3.4 Alternatywne rozwiązania

Nim ostateczna architektura została zaakceptowana, prace koncepcyjne przyniosły wiele pomysłów. Jak wspomniano, dokładny ich opis wykracza poza ramy tego opracowania, jednakże najciekawsze z nich zostaną zaznaczone.

Pierwszą koncepcją, która zostanie pokrótce omówiona w niniejszym podpunkcie, jest wdrożenie własnego protokołu, opracowanego z myślą o komunikacji pomiędzy Agentem, a modułem centralnej aplikacji zarządzającej, Monitorem. Protokołu implementującego pełną funkcjonalność monitorowania oraz wykonywania poleceń sterujących. Z uwagi na istniejące, szeroko stosowane protokoły SNMP oraz SSH, koncepcja została porzucona.

3.4.1 Wykorzystanie biblioteki ProActive wewnątrz systemu

Ciekawym rozwiązaniem, które brano pod uwagę w trakcie projektowania systemu, było rozwinięcie mechanizmu dostarczanego wraz z biblioteką ProActive, modułu *Scheduler* realizującego kolejkę (algorytm szeregowania) zadań. Moduł ten wykonany jest w oparciu o mechanizm *aktywnych obiektów* z samej biblioteki, przede wszystkim jego interfejs jest udostępniany w formie takiego obiektu, dostępnego zdalnie. Takie podejście wpłynęłoby na sposób zgłaszania zadań oraz przydzielania węzłów obliczeniowych do ich realizacji, a także skupiło dużą część systemu w jednej aplikacji.

Szczegóły takiego rozwiązania konsultowano [41] z twórcami biblioteki ProActive. W ówczesnej architekturze biblioteki rozwiązanie takie nie było jednak polecane. Tworzenie procesów-węzłów ProActive przez aplikację inną niż użytkownika traktowane było jako implementacja mechanizmu wdrożenia sprzeczna z założeniami biblioteki, istniejąca jako tymczasowe rozwiązanie pewnych problemów. Dlatego też zrezygnowano z tego pomysłu realizacji systemu.

3.4.2 Szersze użycie Java Enterprise Edition

Rozważano również szersze zastosowanie technologii JEE (ang. *Java Enterprise Edition*) [10]. W tym przypadku, zmiany architektury byłyby znaczące.

Platforma JEE jest często spotykana w biznesowych rozwiązaniach sieciowych. Dostarcza wielu użytecznych mechanizmów integrujących technologie internetowe, bazodanowe oraz szeroko rozumiane przetwarzanie transakcyjne. Znajduje zastosowanie w miejscach, gdzie istnieje potrzeba zastosowania wielowarstwowych, często rozproszonych systemów. W jej skład wchodzi wiele standardów oraz technologii, jak choćby *Enterprise JavaBeans*, *Java Persistence API*, *Java Message Service*, czy cały stos Web Services.

Przydatność JEE w kontekście omawianego systemu opierała się na założeniu łatwości rozwiązania niektórych kwestii, takich jak warstwa kliencka, czy dostęp do

baz danych. Za zaletę uznano również możliwość skorzystania z mechanizmów dedykowanego środowiska uruchomieniowego, tzw. serwera aplikacji. Mianowicie, zapewnia on wielowątkowość oraz transakcyjność przetwarzania, skalowalność, a także dojrzałe rozwiązania z dziedziny bezpieczeństwa.

Ponadto, nagromadzenie wielu technologii, których obecność na rynku wytwarzania systemów informatycznych jest w przyszłości wysoce prawdopodobna, reprezentuje pewien walor naukowo–poznawczy.

Głębsza analiza koncepcji omawianego systemu wykazała jednak, iż wprowadzenie mechanizmów transakcji do przetwarzania opartego o sterowanie procesami nieodwracalnymi, nie jest słuszne. Wykluczenie, natomiast, własności atomowości z dostarczanych mechanizmów nie przyniosłoby w ostatecznym rezultacie spodziewanych korzyści.

Biorąc także pod uwagę argument, iż rezygnacja z platformy JEE nie pociąga za sobą rezygnacji z niektórych jej udogodnień, podjęto decyzję o wycofaniu się z omawianej koncepcji. Wspomnianymi udogodnieniami jest technologia servletów, technologie *Java Persistence API*, JAX–WS (*Java API for XML Web Services*), JAX–B (*Java API for XML Binding*) oraz JMX (*Java Management Extensions*), które to ostatecznie włączono do prezentowanego rozwiązania.

3.4.3 Rozwinięcie istniejących systemów kolejkowych

Na etapie koncepcyjnym została także rozpatrzona możliwość wykorzystania gotowych systemów kolejkowych (por. punkt 2.2).

Takie rozwiązanie niewątpliwie mogłoby przynieść wiele korzyści. Opierając się o znaną dystrybucję, miałoby się gwarancję dojrzałości rozwiązania. Ta z kolei bezpośrednio przekłada się na stabilność, bezpieczeństwo oraz bogatą funkcjonalność systemu kolejkowego. Często spotyka się szereg gotowych oraz sprawdzonych rozwiązań, jak interfejsy oparte o technologie Web Services lub narzędzia monitorujące.

Wybrano jednakże inne podejście, głównie z uwagi na zgodność z przedstawioną specyfikacją systemu. Zapewnienie takich mechanizmów jak wykrywanie zajęć akademickich odbywających się w laboratiorium, włączanie oraz wyłączanie komputerów „na żądanie”, mogłoby okazać się trudne w realizacji. Obawa ta uzasadniona jest wysokim poziomem skomplikowania kodu gotowych systemów oraz zastosowaniem niskopoziomowych technologii przy ich konstrukcji.

Omawiany w niniejszym podpunkcie pomysł został rozpatrzony również w aspekcie integracji z biblioteką ProActive. Ostateczny system z założenia jest przystosowany do współpracy z procesem uruchamiania aplikacji o nią opartych. Dodatkowo, przewidziano możliwość powiązania z zaawansowanymi mechanizmami biblioteki ProActive, jak np. monitoring migracji obiektów, co byłoby trudne do zrealizowania w systemach kolejkowych rozwijanych zazwyczaj w językach programowania innych niż Java.

Rozdział 4

Implementacja systemu

4.1 Aplikacja kontrolująca na węzłach obliczeniowych — Agent

Podstawową funkcjonalnością jaką należało zapewnić na węzłach było dostarczenie mechanizmów udostępniających informacje o stanie węzła dla nadrzędnej aplikacji zarządzającej oraz metod za pomocą których aplikacja ta może wpływać na stan węzła. Funkcje te powinny być udostępniane przez węzły w sposób jednolity, czyli niezależny od architektury sprzętowej oraz systemu operacyjnego uruchomionego na poszczególnych węzłach. Tworzona jest zatem forma abstrakcji nad docelowym środowiskiem obliczeniowym pozwalająca na łatwiejszy i spójny dostęp do jego zasobów nawet w przypadku środowisk heterogenicznych. W realizowanym systemie podstawowymi wspieranymi architekturami sprzętowymi były x86 32bit oraz x86 64bit na których uruchomione były systemy Linux oraz Windows, które znacząco różnią się interfejsem programowania aplikacji (API). Dzięki temu możliwe było wyraźne wyodrębnienie tych elementów, które mogą różnicować kolejne systemy operacyjne, a tym samym zaprojektować aplikacje w taki sposób, aby łatwe było dodawanie wsparcia dla innych systemów operacyjnych.

Ze względu na udostępnianą funkcjonalność oprogramowanie uruchamiane na węzłach można podzielić na trzy części:

- *mechanizm wybudzania wyłączonych komputerów* — nie jest to specjalizowana aplikacja w ścisłym tego słowa znaczeniu, lecz w ogólności zbiór technologii oraz parametrów systemu potrzebnych do włączenia nieaktywnej stacji i zainicjowania wszystkich podsystemów tak, aby możliwe było uruchomienie na danym węźle nowych zadań.

- *aplikacja udostępniająca stan węzła* — usługa systemowa uruchomiona w tle przez cały czas działania stacji, która zbiera i udostępnia aplikacji zarządzającej informacje o stopniu obciążenia systemu oraz stanie zadań zleconych do wykonania na danym węźle.

- *aplikacja realizująca podstawowe operacje na węźle* — realizuje zlecenia od aplikacji zarządzającej takie jak wykonanie zadania, wymuszenie zakończenia zadania czy wyłączenie stacji.

Początkowa koncepcja systemu zakładała użycie biblioteki ProActive i stworzenie aplikacji w postaci aktywnego obiektu, którego metody realizowałyby wskazane funkcje i który byłby trwale uruchomiony w wirtualnej maszynie Javy. Dostęp do takiego obiektu byłby realizowany za pomocą protokołu RMI. Rozwiązanie to było stosunkowo proste w realizacji, dzięki właściwościom języka Java oprogramowanie byłoby przenośne między wszystkimi systemami wspieranymi przez ten język oraz ułatwiałoby integrację systemu z biblioteką ProActive. Napotkane zostały jednak dwa podstawowe problemy. Po pierwsze wymagane było trwałe uruchomienie wirtualnej maszyny Javy, która posiada duże zapotrzebowanie na pamięć operacyjną. Dyskwalifikowało to wymaganą z założenia, współbieżną pracę użytkownika na stacji o małym rozmiarze pamięci operacyjnej. Po drugie rozwiązanie to uzależniało cały system od konkretnej biblioteki warstwy middleware i tym samym traciło na ogólności. W przypadku zmiany biblioteki na inną należałoby zmienić również oprogramowanie na węzłach.

Druga koncepcja zakładała stworzenie aplikacji w języku niskiego poziomu i zaimplementowaniu nowego własnego protokołu komunikacyjnego między aplikacją zarządzającą, a aplikacją uruchamianą na węźle. W porównaniu do poprzedniej koncepcji wyeliminowana została potrzeba użycia konkretnej biblioteki middleware, a dzięki użyciu języka niskiego poziomu spadło również zapotrzebowanie na zasoby systemowe. Pojawiły się jednak nowe problemy. Implementacja nowego protokołu w języku niskiego poziomu była skomplikowana i powodować mogła potencjalne luki w bezpieczeństwie. Skomplikowane było również dodanie metod uwierzytelniania i poufności przesyłanych danych.

Spostrzeżenia dotyczące pierwotnych koncepcji pozwoliły wyodrębnić cechy jakimi powinno charakteryzować się oprogramowanie uruchomione na węzłach:

- posiadające małe zapotrzebowanie na zasoby systemowe
- bezpieczne
- przenośne między różnymi systemami
- niezależne od warstwy middleware

4.1.1 Mechanizm wybudzania wyłączonych komputerów

Proces włączenia komputera nie jest realizowany przez żadną specyficzną aplikację, jednak wymaga istnienia odpowiedniego sprzętu wspierającego technologie umożliwiające zainicjowanie procedury startowej. W architekturach z rodziny x86 technologią umożliwiającą wybudzenie komputera jest *Wake–on–LAN* (WOL). W celu wybudzenia wskazanego komputera wysyłany jest specjalnie spreparowany pakiet tzw.

Magic Packet zawierający sprzętowy adres karty sieciowej (ang. *MAC address*). Wymagana jest zatem karta sieciowa, która umożliwia zinterpretowanie takiego pakietu i wysłanie odpowiedniego sygnału do płyty głównej, który umożliwi rozpoczęcie procedury startowej komputera. Dodatkowo wymagana jest płyta główna pozwalająca na włączenie opcji BIOS-u (ang. *Basic Input/Output System*) odpowiadającej za możliwość włączenia komputera wykorzystując technologię WOL oraz dostarczającej minimalnego napięcia do karty sieciowej potrzebnego do przetwarzania ruchu sieciowego w czasie gdy komputer jest wyłączony. Odpowiednie wsparcie potrzebne jest również ze strony sterowników do karty sieciowej w systemie operacyjnym, które odpowiadają za ustawienie stanu nasłuchiwania „magicznych pakietów" podczas procedury wyłączania komputera.

4.1.2 Aplikacja udostępniająca stan węzła

Podstawowym zadaniem tej aplikacji jest udostępnienie aplikacji zarządzającej stanu węzła reprezentowanego przez zbiór atrybutów. Tabela 4.1 przedstawia wszystkie eksponowane atrybuty wraz z ich opisem.

Nazwa atrybutu	Opis
version	Wersja aplikacji jednoznacznie identyfikująca zbiór pozostałych atrybutów.
idleness	Stopień obciążenia systemu o zakresie wartości 0–100, gdzie wartość 0 oznacza maksymalnie obciążony węzeł na którym nie może być wykonane żadne zadanie, natomiast wartość 100 oznacza system bezczynny.
jobsRunning	Lista identyfikatorów zadań, których procesy są uruchomione w węźle.

Tab. 4.1: Udostępniane atrybuty węzła.

4.1.2.1 Użyte technologie

Do zrealizowania aplikacji udostępniającej stan węzła wykorzystany został protokół SNMP (ang. *Simple Network Management Protocol*). Jest to protokół będący obecnym standardem w zastosowaniach monitorowania urządzeń sieciowych takich jak routery, przełączniki czy komputery. Ze względu na pełnione role urządzenia dzieli się na dwie grupy: *urządzenia zarządzane* oraz *urządzenia zarządzające*. Na urządzeniach z pierwszej grupy uruchomione jest oprogramowanie tzw. *SNMP Agenta* udostępniające określone zmienne charakteryzujące stan danego urządzenia. Stan ten można odczytywać i/lub zapisywać w aplikacji uruchomionej na urządzeniu zarządzającym tzw. *SNMP Managerze*.

To jakie wartości charakteryzują stan urządzenia zarządzanego jednoznacznie identyfikuje tzw. *baza danych MIB* (ang. *Management Information Base*). Baza ta

ma strukturę drzewiastą w której liście reprezentują zmienne skalarne bądź tablicowe określające stan urządzenia, natomiast pozostałe elementy reprezentują *ścieżkę dostępu* do tych zmiennych. Poszczególne wierzchołki w drzewie reprezentowane są przez przypisany do nich numer, przy czym przestrzeń możliwych do wykorzystania numerów jest niezależna dla każdej grupy wierzchołków nie posiadających wspólnego przodka, inaczej mówiąc numery dwóch dowolnych węzłów mogą się powtarzać dopóki nie współdzielą one bezpośredniego wierzchołka nadrzędnego. Istnieje także możliwość identyfikowania poszczególnych węzłów w drzewie poprzez opisowe nazwy, które stanowią czytelny dla człowieka zamiennik numeru identyfikacyjnego. W celu uzyskania dostępu do któregokolwiek z wierzchołków należy podać jego tzw. OID (ang. *Object Identifier*). Jest to innymi słowy jego ścieżka dostępu w postaci serii numerów rozdzielonych kropkami reprezentujących poszczególne węzły na drodze do żądanego wierzchołka poczynając od korzenia drzewa. Aby uniknąć konfliktów numerów OID przypisanie numerów w bazie MIB zarządzane jest przez organizację IANA [23] (ang. *Internet Assigned Numbers Authority*), dzięki czemu ten sam numer OID zmiennej w dwóch dowolnych SNMP Agentach zawsze będzie reprezentował zmienną o tym samym znaczeniu. Opis bazy MIB zawierający m.in. hierarchię elementów, typy zmiennych, czy metody dostępu do danych zawarte są w tzw. *plikach MIB* zapisanych zgodnie z notacją ASN.1 (ang. *Abstract Syntax Notation One*). Notacja ta została specjalnie stworzona do formalnego opisu struktur danych, ich kodowania, przesyłania i dekodowania niezależnie od urządzenia, na którym są one wykorzystywane.

W celu odpytania urządzenia zarządzanego o jego obecny stan wysyłane są tzw. *komunikaty SNMP* wykorzystujące najczęściej protokół UDP. Protokół ten w znacznie mniejszym stopniu obciąża sieć w porównaniu do protokołu TCP, co ma szczególne znaczenie w przypadku istnienia w pojedynczej sieci dużej liczby urządzeń zarządzanych. Najpopularniejszymi komunikatami protokołu SNMP są:

- `Get` — służy wysłaniu żądania pobrania wartości zmiennej reprezentowanej przez podany OID.
- `Response` — odpowiedź na komunikat Get z wartością żądanej zmiennej.
- `Set` — ustawienie zmiennej o danym identyfikatorze OID na podaną wartość.
- `Trap` — wysyłany przez urządzenie zarządzane do urządzenia zarządzającego w celu powiadomienia o zmianie wartości zmiennej identyfikowanej przez dany numer OID wraz z nową wartością.

Wszystkie komunikaty protokołu SNMP posiadają bardzo mały rozmiar co dodatkowo wpływa na mniejsze obciążenie sieci.

Należy zauważyć także, że w ostatniej wydanej specyfikacji protokołu SNMP — SNMPv3 dodano rozszerzenia umożliwiające wykorzystanie silnych mechanizmów

bezpieczeństwa w zakresie uwierzytelnienia stron oraz integralności i poufności danych, co bezpośrednio przekłada się na atrakcyjność tej technologii dla opisywanego systemu.

4.1.2.2 Implementacja

Przed implementacją należało w pierwszym kroku wybrać pakiet realizujący funkcję SNMP Agenta, którego funkcjonalność możnaby rozszerzyć o zmienne charakteryzujące węzeł w opisywanym systemie. Do realizacji tego zadania wybrany został pakiet *net-snmp* [35], który charakteryzuje się implementacją wszystkich wersji protokołu SNMP, otwartym kodem źródłowym, możliwością uruchomienia na szerokiej gamie systemów operacyjnych (w tym Linux i Windows) oraz dostępnością wielu narzędzi ułatwiających rozszerzanie podstawowej funkcjonalności.

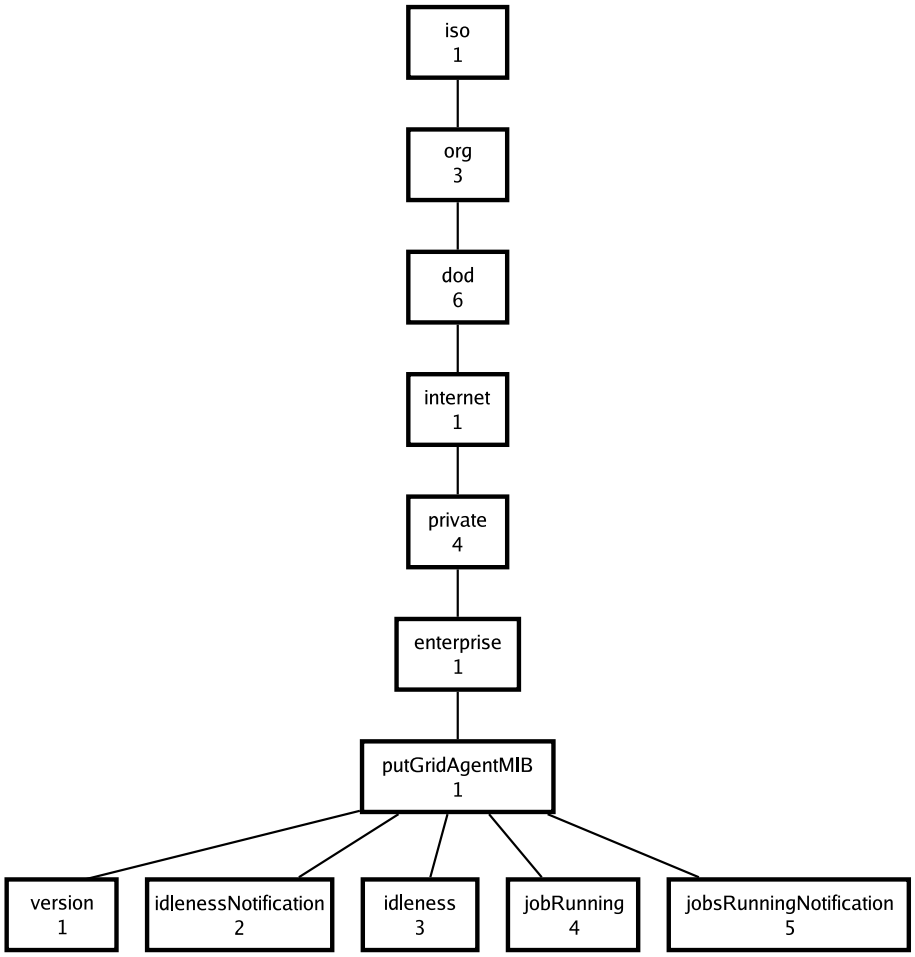
Pierwszym krokiem jaki należało wykonać w celu dodania wymaganych zmiennych do modułu SNMP Agenta było stworzenie pliku MIB zgodnie z notacją ASN.1 opisującego te zmienne oraz ich umiejscowienie w hierarchii drzewa. Zalecanym miejscem przez organizację IANA dla nowych rozszerzeń włączanych do bazy MIB jest wierzchołek `iso.org.dod.internet.private.enterprise` (OID 1.3.6.1.4.1). Utworzono zatem nowy wierzchołek podrzędny o nazwie `putGridAgentMIB` identyfikowany przez numer 1 w poddrzewie o korzeniu znajdującym się w węźle `enterprise`. Udostępniane zmienne wraz z odpowiadającym im numerami OID oraz pełnioną przez nie funkcją przedstawione są w tabeli 4.2 (por. tabela 4.1).

Nazwa zmiennej	OID	Funkcja
version	`1.3.6.1.4.1.1.1`	Wartość zmiennej version.
idlenessNotification	`1.3.6.1.4.1.1.2`	Powiadomienie o zmianie wartości. zmiennej idleness
idleness	`1.3.6.1.4.1.1.3`	Wartość zmiennej idleness.
jobsRunning	`1.3.6.1.4.1.1.4`	Wartość zmiennej jobsRunning.
jobsRunningNotification	`1.3.6.1.4.1.1.5`	Powiadomienie o zmianie wartości zmiennej jobsRunning.

Tab. 4.2: Zmienne udostępniane w bazie MIB.

Strukturę udostępnianych zmiennych w drzewie MIB można zobrazować w sposób graficzny (zob. rysunek 4.1).

Następnym krokiem było wygenerowanie na podstawie pliku MIB szkieletu modułu w języku C. Było to możliwe dzięki narzędziu dostarczanemu wraz pakietem net-snmp o nazwie *mib2c*. Program ten tworzy pliki źródłowe modułu rozszerzającego SNMP Agenta w taki sposób, że programista może skupić się na implementacji funkcji dostarczających wartości wymaganych zmiennych, natomiast o wszystkie szczegóły dotyczące integracji z pakietem net-snmp „troszczy się" narzędzie mib2c. Szczególną uwagę przywiązano do maksymalizacji współdzielonej bazy kodu między



Rys. 4.1: Hierarchia drzewa MIB opisywanego systemu.

różnymi systemami operacyjnymi, tak aby dodanie nowych systemów wymagało minimalnych nakładów pracy i sprowadzało się do zaimplementowania krótkich pojedynczych funkcji charakterystycznych dla danego systemu.

Ostatnim etapem było zaimplementowanie funkcji odpowiedzialnych za udostępnianie wartości zmiennych. Wartością zmiennej `version` jest zawsze ta sama liczba identyfikująca wersję dostarczanego modułu oraz zbiór pozostałych zmiennych. Wartością zmiennej `idleness` jest stopień obciążenia węzła, która w przyjętej implementacji może przyjmować dwie wartości 0 lub 100. Wartość 0, oznaczająca maksymalne obciążenie węzła, ustawiana jest w momencie wykrycia sesji logowania jakiegokolwiek użytkownika w systemie oprócz specjalnego użytkownika przeznaczonego do wykonywania operacji na rzecz opisywanego systemu. Wartość 100, oznaczająca system bezczynny, ustawiana jest w sytuacji przeciwnej, kiedy żaden z użytkowników nie jest zalogowany. W systemie Linux wykrywanie zalogowanych użytkowników odbywa się przy użyciu mechanizmu systemowego *utmpx* stworzonego specjalnie do tego celu. Pozwala on na pobranie listy nazw użytkowników zalogowanych w systemie (włączając w to sesje zdalne), dzięki czemu po odfiltrowaniu

nazwy użytkownika specjalnego otrzymywana jest liczba zalogowanych użytkowników. W systemie Windows wykrywanie zalogowanych użytkowników jest procesem bardziej złożonym z uwagi na brak wbudowanych mechanizmów do wykonywania takich operacji. Do realizacji tego zadania wykorzystano informacje przechowywane w *rejestrze systemowym* w kluczu HKEY_USERS dotyczące istniejących sesji logowania. Po przetworzeniu zawartych tam danych można było uzyskać rezultat w postaci listy zalogowanych użytkowników. Kwestią otwartą pozostaje zmiana funkcji wyliczającej wartość zmiennej `idleness` w sposób bardziej wyrafinowany np. wykorzystując stopień wykorzystania procesora i/lub pamięci. Zmiana taka wymaga jedynie zmiany implementacji funkcji `is_idle()`.

W funkcji obsługującej zmienną `idlenessNotification` odpowiadającej za powiadamianie SNMP Managera o zmianie wartości zmiennej `idleness` sprawdzana jest okresowo wartość zmiennej `idleness` i w przypadku zmiany wartości między wywołaniami wysyłany jest komunikat typu `Trap` do aplikacji SNMP Managera. Wartością zmiennej `jobsRunning` jest lista identyfikatorów zadań rozdzielona pojedynczym odstępem reprezentująca procesy uruchomione w systemie, natomiast zmienna `jobsRunningNotification` informuje SNMP Managera o zmianach na liście uruchomionych procesów. W przypadku wielu procesów należących do tego samego zadania identyfikator zadania wystąpi na liście tyle razy, ile uruchomionych jest jego procesów. W celu rozróżnienia które z procesów systemowych wykonywane są na rzecz opisywanego systemu wprowadzona została koncepcja *katalogu runfile*. Jest to specjalny katalog wyróżniony w konfiguracji SNMP Agenta, w którym przechowywane są pliki, których nazwa stanowi identyfikator procesu uruchomionego w systemie. Pliki te zawierają informację o identyfikatorze zadania, do którego dany proces należy. Dzięki temu możliwe jest monitorowanie tego katalogu w poszukiwaniu nowopowstałych plików, co wskazuje na fakt uruchomienia nowego procesu zleconego przez opisywany system. Monitorowane są również uruchomione procesy i w przypadku, kiedy któryś z nich zakończy się usuwany jest odpowiadający mu plik z katalogu runfile.

Dołączenie zaimplementowanego modułu do istniejącego SNMP Agenta wiąże się w przypadku systemu Linux z kompilacją modułu do dynamicznie ładowanej biblioteki oraz określeniem ścieżki do niej w pliku konfiguracyjnym. Pozwala to na dynamiczne rozszerzenie funkcjonalności SNMP Agenta bez jakiejkolwiek ingerencji w zmianę aktualnej konfiguracji. W przypadku systemu Windows nie ma możliwości ładowania dynamicznych bibliotek i aby dołączyć stworzony moduł należy skompilować cały program SNMP Agenta z dołączonymi plikami źródłowymi nowego modułu.

4.1.3 Aplikacja realizująca operacje na węźle

Podstawową funkcjonalnością realizowaną przez aplikację o nazwie *putgridctl* jest wykonywanie operacji mających na celu zmianę stanu docelowego węzła. Program

ten powinien eksponować spójny interfejs na każdym węźle należącym do systemu, niezależnie od jego architektury i uruchomionego na nim systemu operacyjnego. Operacje możliwe do zrealizowania na rzecz stacji zarządzającej wymienione są w tabeli 4.3.

Nazwa operacji	Opis działania
`run`	Uruchomienie komendy użytkownika na rzecz zadania o danym numerze identyfikacyjnym.
`kill`	Wymuszenie zakończenia wszystkich procesów związanych z wyspecyfikowanym numerem zadania.
`shutdown`	Wyłączenie stacji.
`clear`	Przygotowanie spójnego środowiska startowego przed uruchomieniem pozostałych komponentów systemu.
`test`	Sprawdzenie poprawności środowiska.

Tab. 4.3: Operacje realizowane przez program putgridctl.

Należało również dostarczyć na każdym węźle mechanizmów zdalnego wykonania komend umożliwiających uwierzytelnienie użytkownika oraz poufność i integralność przesyłanych danych.

4.1.3.1 Użyte technologie

W celu zapewnienia możliwości zdalnego wykonania programu putgridctl zdecydowano się na wybór protokołu SSH (ang. *Secure Shell*) w wersji drugiej [47], który spełnia wszystkie wymagania stawiane przez opisywany system. Ze względu na pełnione role oprogramowanie uczestniczące w komunikacji używającej protokołu SSH można wyróżnić podzielić na dwie grupy — oprogramowanie *serwera SSH* przyjmujące nowe zgłoszenia nawiązywania sesji i realizujące zadania wykonania programów na rzecz tych sesji, oraz oprogramowanie *klienta SSH* które nawiązuje sesje z serwerem SSH i zlecające mu nowe zadania. Na każdym węźle zatem powinno być dostarczone oprogramowanie serwera SSH, który umożliwiłby zdalne wykonanie programu putgridctl na rzecz systemu zarządzającego, który w tym wypadku pełniłby rolę klienta SSH. Wybranym serwerem przeznaczonym do uruchomienia na węzłach został serwer SSH z dystrybucji OpenSSH [38] będący pełną implementacją niezbędnych narzędzi do komunikacji z użyciem protokołu SSH o otwartym kodzie źródłowym. Implementacja pakietu OpenSSH dostępna jest na szeroką gamę systemów, a dzięki projektowi Cygwin [18] implementującemu standard POSIX [39] oraz projektowi SSHWindows [45] możliwe jest również uruchomienie serwera SSH w systemach Windows.

Zaletą zastosowania protokołu SSH jest możliwość uwierzytelnianie użytkownika na różne sposoby:

- jawne podanie hasła przez użytkownika

- uwierzytelnianie za pomocą pary kluczy publicznego i prywatnego wykorzystujących algorytmy DSA lub RSA
- uwierzytelnianie przy wykorzystaniu serwera Kerberos

Szczególnie użyteczną metodą jest opcja uwierzytelniania parą wzajemnie zależnych kluczy kryptograficznych — publicznego i prywatnego. Charakteryzują się one zasadą symetryczności — to co zostanie zaszyfrowanie kluczem publicznym może być odszyfrowane tylko i wyłącznie kluczem prywatnym, natomiast to co zostanie zaszyfrowane kluczem prywatnym może być odszyfrowane tylko i wyłącznie kluczem publicznym. Dzięki tej własności możliwe jest przyznawanie praw dostępu do danego konta użytkownika poprzez umieszczenie w specjalnym pliku (tzw. *authorized_keys*) znajdującym się w katalogu domowym tego użytkownika, klucza publicznego osoby mającej mieć możliwość dostępu do wskazanego konta.

Dodatkowo cała komunikacja odbywająca się z wykorzystaniem protokołu SSH jest zawsze szyfrowana kluczem symetrycznym ustalanym podczas negocjacji sesji. Możliwe jest wykorzystanie takich algorytmów jak AES, Blowfish czy DES.

4.1.3.2 Implementacja

Aplikacja putgridctl z uwagi na wykonywane niskopoziomowe zadania została zrealizowana w języku C oferującym bezpośredni dostęp do funkcji systemowych. Poszczególne funkcje realizujące komendy wymienione w rozdziale 4.1.3 są współdzielone między różnymi systemami operacyjnymi natomiast odwołują się one do niskopoziomowych funkcji implementowanych dla każdego systemu osobno. Kwestia dodania obsługi nowego systemu operacyjnego sprowadza się więc do implementacji kilku małych funkcji odnoszących się tylko i wyłącznie do mechanizmów charakterystycznych dla tego systemu. Program putgridctl odczytuje plik konfiguracyjny `putgrid.conf` umiejscowiony w globalnym katalogu konfiguracyjnym definiowanym podczas procesu kompilacji (tzw. `sysconfdir`) oraz w katalogu domowym użytkownika. Możliwe do użycia opcje w tym pliku omówione zostaną w dalszej części tego rozdziału. Do przetwarzania struktury pliku konfiguracyjnego użyta została biblioteka *libconfig* [34] charakteryzująca się prostym interfejsem programisty, czytelną składnią plików konfiguracyjnych, wsparciem dla wielu systemów operacyjnych oraz otwartym kodem źródłowym.

Najbardziej złożoną z punktu widzenia implementacji komendą była komenda `run` z uwagi na potrzebę powiadomienia aplikacji SNMP Agenta (zob. rozdział 4.1.2) o uruchomieniu nowego procesu w ramach zleconego zadania oraz obsłużenie różnic, jakie mogą wystąpić między poszczególnymi węzłami z powodu różnych konfiguracji systemów lub też różnic wynikających z konwencji przyjętych w danym systemie operacyjnymi. Pierwszy z problemów rozwiązano dzięki zastosowaniu katalogu runfile opisanemu szerzej w rozdziale 4.1.2.2. Zadaniem narzędzia putgridctl jest w tym wypadku uruchomienie procesu w systemie, pobranie jego identyfikatora

oraz stworzenie nowego pliku w katalogu runfile o nazwie odpowiadającej identyfikatorowi uruchomionego procesu. Zawartość pliku stanowi natomiast identyfikator zlecenia, przekazywany przez aplikację zlecającą zadanie jako argument wywołania putgridctl. Ścieżka do katalogu runfile podawana jest w obowiązkowej opcji pliku konfiguracyjnego o nazwie `run_file_dir`. Drugi z problemów przy uruchamianiu komendy użytkownika w postaci różnic między systemami występuje w przypadku m.in. różnic w ścieżkach do plików — npp̀akiet ProActive może być zainstalowany na jednym z systemów w katalogu `c:\ProActive` natomiast na innym w katalogu `/opt/ProActive`. Innym przypadkiem, w którym mogą ujawnić się różnice między systemami są różnice w przyjętych konwencjach — np. w systemie Windows znakiem używanym do separacji ścieżek w zmiennej środowiskowej `PATH` jest znak średnika, natomiast na większości systemów z rodziny Unix znakiem tym jest dwukropek. Różnica ta ma szczególne znaczenie w momencie zlecania zadania przez bibliotekę ProActive w którym podawane są ścieżki do wymaganych bibliotek (tzw. `CLASSPATH`) rozdzielonych właśnie wspomnianym znakiem. Aby zaradzić temu problemowi wprowadzono mechanizm zastępowania wzorców występujących w komendzie przekazanej przez użytkownika na wartości zdefiniowane w pliku konfiguracyjnym. Wzorce te definiowane są w sekcji `path_patterns` w postaci
tekst wzorca = tekst jakim powinien być zastąpiony wzorzec.
Pozwala to na uruchomienie procesu użytkownika bez podawania bezpośrednich ścieżek, a zamiast tego umożliwia wyspecyfikowanie *meta–ścieżki* np. `PROACTIVE\_HOME`, której definicja specyfikowana byłaby przez administratora systemu w pliku konfiguracyjnym. Dzięki temu system staje się bardziej niezależny od konfiguracji poszczególnych węzłów.

Kolejną operacją wykonywaną przez pudgridctl jest możliwość wymuszenia zakończenia procesów należących do zadania o identyfikatorze wskazanym w linii komend. W tym celu wykorzystywany jest ponownie katalog runfile w ten sposób, że wczytywane są wszystkie pliki znajdujące się w tym katalogu oraz odczytuje się ich zawartość porównując ją z przekazanym identyfikatorem zadania. Jeżeli te dwie wartości są identyczne następuje zakończenie procesu o identyfikatorze wskazanym przez nazwę pliku.

Operacja `shutdown` powoduje wyłączenie komputera i realizowana jest w odmienny sposób na każdym z systemów. Dla systemu Linux jest to wykonanie programu `/sbin/shutdown -h now`, natomiast w przypadku systemu Windows wykonanie funkcji systemowej `InitiateSystemShutdown`.

Operacje `clear` oraz `test` mają za zadanie, odpowiednio, przygotowanie środowiska pracy oraz sprawdzenie jego poprawności. W obecnej implementacji przygotowanie środowiska polega na wyczyszczeniu katalogu runfile z ewentualnych plików pozostałych po poprzednich sesjach i powinno być wykonane na etapie startowania systemu przed uruchomieniem usług związanych z opisywanym systemem.

4.2 Komunikacja zewnętrzna — Web Services

Każdy system wielomodułowy, którego poszczególne składowe z założenia mają charakter rozproszony, musi dostarczać mechanizmy komunikacji międzymodułowej. Realizację tego postulatu oparto w omawianym systemie o otwarte oraz przenośne standardy, których całość składa sie na technologię tzw. usług sieciowych (ang. *Web Services*). Dzięki temu uzyskano elastyczny dostęp do funkcji dostarczanych przez moduły główne systemu. Ponadto, nie nałożono szczególnych wymogów, bądź też ograniczeń na różnorodność oraz mnogość aplikacji klienckich, korzystających z udostępnionych funkcji. Niniejszy podrozdział poświęcono na omówienie części zastosowanych standardów oraz przedstawienie szczegółów implementacyjnych.

4.2.1 Przegląd technologii oraz standardów

Miano *Web Services* kryje pełną gamę standardów służących do zapewnienia komunikacji sieciowej opartej w głównej mierze o infrastrukturę oraz technologie internetowe. Najczęściej spotykane w dzisiejszych zastosowaniach są technologie wykorzystujące XML (ang. *Extensible Markup Language*, Rozszerzalny Język Znaczników), uniwersalny język do opisu danych w sposób strukturalny. Należy podkreślić, iż jest to format tekstowy, dzięki czemu z założenia gwarantuje przenośność. Mając ten fakt na uwadze, również w omawianym systemie wykorzystano technologię XML do reprezentowania danych opisujących sposób komunikacji, jak i będących jej przedmiotem. Warto również zauważyć, iż *Web Services* zostało oficjalnie zatwierdzone przez konsorcjum standaryzujące W3C (ang. *World Wide Web Consortium*). Dostarczono w ten sposób szereg w pełni zdefiniowanych, otwartych standardów, które są tym samym gwarantem ogólnodostępności omawianych technologii.

Usługi sieciowe reprezentują pewien interfejs udostępniany przez serwer. Interfejs jest w tym przypadku niczym innym jak zbiorem funkcji czy procedur — ogólniej usług, do których mają dostęp klienci. Wynika z tego, że usługi sieciowe są realizacją potrzeby interakcji sieciowej pomiędzy komputerami [49]. Interfejs należy również rozumieć jako kontrakt, którego obowiązek implementacji nakłada się na konkretny serwer. Idąc dalej, kontrakt ten jest wykorzystywany przez klientów, jako opis usługi sieciowej z której zamierzają skorzystać.

Model wykorzystywany do opisu kontraktów pomiędzy serwerem a klientami, jest tworzony przy użyciu języka WSDL (ang. *Web Services Description Language*) opartego o technologię XML. WSDL, jako standard W3C, daje możliwość definicji protokołów oraz sposobu wymiany danych, a także wszelkich niezbędnych informacji służących do ich uściślenia. Sam dokument napisany w języku WSDL (w skrócie dokument WSDL) w założeniach ma naturę hierarchiczną, dzięki czemu w sposób niezależny może opisywać kolejne warstwy komunikacji.

W praktyce kontrakty są punktem wyjścia dla aplikacji klienckich, chcących korzystać z danej usługi sieciowej. Dlatego też bardzo ważne jest przestrzeganie

standardów oraz zaleceń W3C [50] przez każdą implementację technologii *Web Services*. W trakcie tworzenia niniejszego systemu zwrócono na ten fakt szczególną uwagę, dokonując testowych integracji wybranych implementacji.

U podstaw usług sieciowych leży idea zdalnego wywoływania procedur (RPC, ang. *Remote Procedure Call*), której owocem był standard XML–RPC, należący do pierwszej generacji protokołu wymiany danych pomiędzy serwerem oraz klientami. Standard ten czerpał z korzyści jakie daje technologia XML, jednakże wprowadzał pewne obostrzenia jeśli chodzi o różnorodność typów przesyłanych danych. Odpowiedzią na ograniczenia protokołów pierwszej generacji stał się standard SOAP (ang. *Simple Object Access Protocol*), również oparty o XML. W przedstawianym systemie zdecydowano się na użycie właśnie tego rozwiązania, z uwagi na jego elastyczność, obecną oraz prawdopodobnie również przyszłą popularność.

Sam protokół SOAP definiuje wiadomości w czterech podstawowych wariantach. Styl wiadomości może przyjąć formę *RPC*, bądź też *Document*, natomiast kodowanie formę *Encoded* lub *Literal*. Wariantem zastosowanym w przedstawianym systemie jest kombinacja Document/Literal. Ponadto, by spełniać wymagania stawiane przez dokument WS–I Basic Profile (ang. *Web Services Interoperability Basic Profile*) [51], zastosowano dodatkowo styl *Wrapped* przekazywanych danych [52]. Jak wspomniano miejscem, w którym dokonuje się opisu przesyłanych danych jest dokument WSDL, tak więc podjęte decyzje dotyczące kształtu wiadomości SOAP, są w nim zdefiniowane. Technologicznie, w celu uściślenia wyglądu wiadomości SOAP, będącymi strukturą XML, dokument ten posiłkuje się definicjami w języku *XML–Schema*.

Technologie usług sieciowych nie ograniczają różnorodności protokołów, które mogą stanowić warstwę transportującą wiadomości SOAP. Do najpopularniejszych należy oczywiście protokół HTTP (ang. *Hypertext Transfer Protocol*), na którym oparto komunikację w omawianym systemie.

4.2.2 Przetestowane biblioteki

Z uwagi na główne cechy technologii *Web Services*, przenośność oraz wieloplatformowość, przetestowano jej wybrane implementacje pod kontem integracji. Dokonano tego w celu zapewnienia możliwości rozbudowy przedstawianego systemu o dodatkowe moduły klienckie. Innym, równie ważnym powodem wykonanej pracy, była potrzeba przetestowania poprawności zaimplementowanych serwisów, mechanizmów uwierzytelniania oraz autoryzacji. Opracowano przykładowy scenariusz testowy, którego poprawna realizacja przy pomocy integrowanych bibliotek była celem samym w sobie. Poniżej przedstawiono krótką charakterystykę poszczególnych rozwiązań oraz wnioski płynące z realizacji wyżej wymienionych czynności.

4.2.2.1 Strona serwera

Po stronie serwera dokonano rozeznania w bibliotekach implementujących technologię *Web Services* w kontekście udostępnienia konkretnych funkcji realizowanych

przez moduł zarządzający węzłami (por. 4.3.3). Z uwagi na równoczesne przygotowywanie implementacji usług sieciowych oraz wspomnianych funkcji, należało wymagać od zastosowanej biblioteki elastyczności w definiowaniu kontraktu. Z kolei częste zmiany interfejsu wymagały adaptacji ze strony zaimplementowanych mechanizmów oraz rozwiązań. Przedstawione potrzeby znakomicie adresuje technologia JAX–WS (ang. *Java Api for XML Web Services*), którą zaproponował Sun Microsystems dla platformy Java Enterprise Edition [10].

Samą technologię opisano w dokumencie standaryzującym JSR 224 [32]. Główną jej cechą jest wykorzystanie mechanizmu adnotacji (ang. *Java Annotations*) znanego z platformy Java 5. Dzięki temu umożliwiono programistom tworzenie usług sieciowych już na poziomie języka Java. Utrzymywanie kontraktu wynikającego z dostarczonej implementacji usług, stał się tym samym mniej czasochłonny. JAX–WS jest jednocześnie naturalnym następcą wcześniejszej technologii firmy Sun, JAX–RPC [30].

Warto również zwrócić uwagę na powiązanie standardu JAX–WS z technologią JAX–B (ang. *Java Api for XML Binding*) [31], służącą do wiązania zawartości dokumentów XML z klasami w języku Java, na podstawie dostarczonych schematów. Standard JAX–B obwieszczono właściwą technologią dla bibliotek usług sieciowych współpracujących z dokumentami w formie XML. Stał sie on alternatywą do dotychczas stosowanych podejść, jakimi był SAX (ang. *Simple Api for XML*) oraz DOM (ang. *Document Object Model*). Alternatywą, jednakże nie zamiennikiem. JAX-WS dostarcza bowiem mechanizmy, które w ogólności opierają sie o wspomniane technologie, prezentując jednocześnie nową jakość.

Poniżej opisano konkretny pakiet bibliotek będący implementacją omawianej technologii.

JAX–WS Reference Implementation, JDK 6 Release. Biblioteki wchodzące w skład pakietu stanowią „wzorcową" implementację opisanego wyżej standardu, zaproponowaną przez Sun Microsystems. W omawianej pracy wzięto pod uwagę edycję dołączaną wraz z JDK 6 (ang. *Java SE Delevopment Kit*). Należy podkreślić, iż ten *otwarty* projekt stanowi z założenia platformę produkcyjną, przeznaczoną do ostatecznego zastosowania w rzeczywistych systemach. W trakcie przygotowywania omawianego systemu napotkano jednakże na pewne utrudnienia związane z integracją usług JAX–WS RI z klientami opartymi o inne biblioteki.

Istotnym wymogiem z punktu widzenia założeń architektury systemu była również możliwość opublikowania usług sieciowych z poziomu samodzielnej aplikacji JSE, z pominięciem tzw. *Web Containera*. Potrzeba ta była wynikiem odrzucenia (por. podpunkt 3.4.2) platformy Java Enterprise Edition [10], w której wspomniany *Web Container* byłby nieodzownym elementem systemu. Ostatecznie więc planowano uzyskać architekturę wymagającą mniejszej ilości zasobów systemowych. Realizacja tego postulatu była możliwa przy użyciu pakietu JAX–WS RI, gdyż dostarcza on wbudowany, jednakże prosty serwer protokołu HTTP (ang. *Hypertext*

Transfer Protocol) w wersji 1.1 [22].

Integracja usług sieciowych zbudowanych na podstawie omawianego pakietu ujawniła pewne nieścisłości dotyczące żądań HTTP w kwestii kompatybilności wstecznej oraz interpretowania parametrów nagłówkowych (por. [22]). Ich konsekwencją były utrudnienia w komunikacji ze stroną kliencką (por.). W trakcie realizacji szczegółów systemu, zauważono także brak wsparcia dla niektórych mechanizmów, których szerszy opis zawarto w podpunkcie 4.3.4 *Moduł realizacji usług sieciowych — Web Services*, głównie tzw. *Handlerów*.

Warto również podkreślić fakt, iż z uwagi na zastosowanie popularnych standardów w implementacji usług sieciowych, JAX–WS oraz JAX–B, otrzymany kod powinien współdziałać z innymi bibliotekami wspierającymi wspomniane standardy. Przykładem takiego pakietu może być *Apache CXF*.

4.2.2.2 Strona kliencka

W czasie testów, strona kliencka usług sieciowych brała udział nie tylko w realizacji zadanego scenariusza testowego, ale przede wszystkim w weryfikacji zgodności implementacji ze standardami. Przyjęto założenie, że weryfikacja współdziałania różnych bibliotek z usługą sieciową zaimplementowaną przy użyciu JAX–WS RI, zagwarantuje kompatybilność rozwiązania. Jest to kluczowe zagadnienie z punktu widzenia rozbudowy całego systemu o aplikacje klienckie, które mogą korzystać z nieprzewidzianych w niniejszym opracowaniu implementacji. Warto zwrócić uwagę na szeroką gamę bibliotek przeznaczonych dla rozwiązań mobilnych, których przetestowanie wykraczało poza ramy wykonanej pracy.

W trakcie rozwoju oraz testowania aplikacji klienckich należało zrealizować kontrakt usługi sieciowej, którą implementował serwer. W kolejnej fazie integracji, wzbogacano aplikacje o dodatkowe mechanizmy wykorzystywane w celu uwierzytelniana oraz autoryzacji. Mechanizmy te oparto o nagłówki wiadomości SOAP (ang. *SOAP Headers*), których poprawna realizacja również była przedmiotem weryfikacji.

JAX–WS Reference Implementation, JDK 6 Release. Naturalnym podejściem było wykonanie scenariusza testowego w oparciu o biblioteki, które zastosowano również po stronie serwera. Integracja, zgodnie z oczekiwaniami, przebiegła pomyślnie w początkowej fazie tworzenia usług sieciowych. Jednakże wraz z dostarczeniem funkcjonalności uwierzytelniania oraz autoryzacji, zaprzestano dalszego rozwoju aplikacji. Stało sie tak z powodu braku wsparcia dla wspomnianego mechanizmu ze strony narzędzi dostarczonych wraz z JAX–WS RI.

Apache Axis. Jest to *otwarty* pakiet bibliotek oparty o technologię JAX–RPC zaproponowaną przez Sun Microsystems [30], wspierany przez fundację Apache. Z powodu bezproblemowej integracji z *Web Containerem* Apache Tomcat, jest dosko-

nałym narzędziem do realizacji usług sieciowych, zwłaszcza tam, gdzie wymagane jest zastosowanie stylu RPC/Encoded [52] przesyłanych wiadomości SOAP. W opisywanym systemie zastosowano jednak pakiet Axis nie w roli serwera, a klienta usługi dla testów integracyjnych. Ostatecznie zastosowano go także w implementacji aplikacji WWW (por. 4.6).

W procesie integracji klienta opartego o Apache Axis z serwerem wykonanym w technologii JAX–WS RI, uwidoczniły się pewne niezgodności dotyczące obsługi protokołu HTTP. Zaimplementowany klient domyślnie wykorzystuje protokół HTTP w najprostszej wersji — 0.9. Serwer, natomiast, konsekwentnie udziela odpowiedzi zgodnie z wersją 1.1 protokołu, dodając przy tym nagłówek *keep-alive* [22], czego następstwem jest zawieszenie klienta. Opracowanym rozwiązaniem jest dodanie do nagłówka każdego żądania dodatkowego pola *Connection: close*, które wymusza zamknięcie połączenia po nadaniu odpowiedzi przez serwer.

Dostarczone wraz z Apache Axis narzędzia niestety również nie wspierają nagłówków SOAP, których obecność jest wymagana w celach uwierzytelniania oraz autoryzacji. Jednakże możliwości biblioteki pozwoliły ostatecznie na poprawną implementację tego mechanizmu, poprzez odpowiednią manipulację modelem DOM (ang. *Document Object Model*).

By zautomatyzować proces tworzenia aplikacji klienckiej opartej o bibliotekę Apache Axis, przygotowano odpowiedni skrypt dla narzędzia Ant, który reaguje na modyfikacje kontraktu usługi sieciowej.

Apache Axis 2. Kolejny zestaw bibliotek jest „odświeżoną" linią poprzedniej propozycji, gdyż został zaprojektowany oraz wykonany praktycznie od podstaw. Realizuje on cały przekrój standardów, w tym również omówiony wcześniej JAX–WS. Rozwój aplikacji klienckiej opartej o Apache Axis 2, posłużył głównie jako element doświadczalny przy testach integracyjnych.

Napotkaną trudnością w trakcie realizacji scenariusza testowego były ponownie pola w nagłówkach żądań protokołu HTTP. W przypadku, gdy serwer zwracał wczesny błąd realizacji funkcji serwisu, spowodowany przez próbę błędnego uwierzytelnienia, aplikacja kliencka zawieszała się. Działo sie tak, gdyż serwer prawdopodobnie przez pomyłkę zwracał odpowiedź z polem *transfer–coding = chunked* nagłówka HTTP, nie realizując zadeklarowanego kodowania [22]. Należy zatem wykluczyć pole *transfer–coding = chunked* z nagłówka żądania klienta.

W kontekście nagłówków wiadomości SOAP, Apache Axis 2 dostarcza narzędzi pozwalających na automatyczną implementację tychże nagłówków, zgodnie z nałożonym kontraktem usługi sieciowej.

4.3 Centralna aplikacja zarządzająca — Manager

Centralnym elementem implementowanego systemu jest aplikacja zarządzająca. Jej zasadniczą rolą jest monitorowanie systemu, reagowanie na zdarzenia zachodzące

w węzłach i przetwarzanie zgłoszeń nadchodzących z innych aplikacji — klientów Web Services. Zgłoszeniami tymi są przede wszystkim operacje na zadaniach (np. zgłoszenie lub wycofanie zadania), węzłach sieci, jak i udostępnianie informacji nt. monitorowanych węzłów. Wszystkie operacje w systemie są dokonywane lub kontrolowane przez aplikację zarządzającą.

Aplikacja zarządzająca została zaimplementowana w języku Java, uruchamiana jest wewnątrz jednej wirtualnej maszyny (JVM) — w standardowej platformie Java SE 1.6.

Ze względu na złożoność tej aplikacji, wydzielono w niej szereg modułów funkcjonalnych, o możliwie minimalnych zależnościach pomiędzy nimi:

- *Moduł dostępu do danych (DB)* — udostępnia pozostałym modułom model danych przez łatwe w użyciu intefejsy, zarządza współbieżnym dostępem do danych i ich utrwalaniem. Jest niezależny od pozostałych modułów.

- *Moduł monitorująco-wykonawczy (Monitor)* — utrzymuje bieżący stan węzłów w systemie (rola komplementarna w stosunku do bazy danych — przechowuje tylko informacje bieżące, nietrwałe) przez komunikację z nimi. Wykonuje zlecone operacje na węzłach. Jest niezależny od pozostałych modułów.

- *Moduł zarządzający węzłami (Disposer)* — opierając się na informacjach z modułów DB i Monitora podejmuje decyzje odnośnie węzłów (włączenie, wyłączenie, kontrolowanie uruchomionych zadań), jak i zadań (wystartowanie, zakończenie, przydział do węzłów). Deleguje wykonanie operacji do modułu wykonawczego — *Monitora*, zapisuje i odczytuje dane korzystając z modułu DB.

- *Moduł realizacji usług sieciowych (Web Services)* — odpowiada za komunikację aplikacji zarządzającej z innymi aplikacjami, udostępniając usługi w sposób oparty o otwarte oraz przenośne standardy. Deleguje otrzymane zgłoszenia do modułu Disposer. W jego skład wchodzi również moduł weryfikacji tożsamości (ang. *Identity Provider*), dostarczający mechanizm uwierzytelniania oraz autoryzacji zgłoszeń.

4.3.1 Moduł dostępu do danych — DB

Konfiguracja środowiska, a także pewne kluczowe informacje m.in. o zgłoszonych zadaniach przechowywane są w bazie danych. Ponieważ wśród początkowych koncepcji zakładano rozwiązanie oparte o platformę JEE (*Java Enterprise Edition*), dostęp do bazy danych odbywać miał się za pośrednictwem interfejsu JPA (*Java Persistence API*). Pomimo tego, iż odstąpiono od standardów z rodziny JEE takich jak JMS czy EJB, które to wymagały użycia serwera aplikacji JEE, nie była konieczna rezygnacja z użycia JPA, jako że framework ten może być wykorzystany również wewnątrz standardowej platformy Java SE.

4.3.1.1 Użyte technologie

Specyfikacja JPA w wersji 1.0 została wydana razem z trzecią wersją specyfikacji EJB (*Enterprise JavaBeans*)[14] i zakładała odejście od wcześniejszych mechanizmów utrwalania (entity beans jako część EJB), a przejście w stronę rozwiązań znacznie "lżejszych" zaproponowanych przez frameworki takie, jak np. Hibernate czy TopLink. Interfejs JPA jest abstrakcją nad JDBC (*Java Database Connectivity*) i opiera się na użyciu prostych obiektów POJO (*Plain Old Java Object*) i ich odwzorowaniu na odpowiednie relacje w bazie danych zgodnie z zasadami ORM (*Object-Relational Mapping*) opisanymi szczegółowo w specyfikacji. Umożliwia to automatyczne wykonywanie operacji w bazie danych bez konieczności umieszczania w kodzie źródłowym konstrukcji nawiązujących połączenie z bazą danych i samodzielnego synchronizowania. Dodatkowo JPA dostarcza język zapytań EJB QL (*EJB Query Language*) umożliwiający formułowanie nawet złożonych zapytań, a zorientowany bardziej na obiekty encyjne niż na relacje w bazie danych. Ponieważ JPA jest jedynie zbiorem interfejsów, niezbędna była także implementacja. Jako dostawcę implementacji wybrano Hibernate, natomiast jako systemu zarządzania relacyjną bazą danych użyto MySQL. Więcej informacji o technologii JPA można znaleźć w książce [1].

4.3.1.2 Szczegóły implementacji

Odwzorowania relacyjne. Diagram związków encji dla bazy danych opisywanego systemu znaleźć można w rozdziale 3.1.6. W implementacji klas encji, odwzorowanych w bazie danych, wykorzystano mechanizm adnotacji, który stanowi alternatywę dla deskryptorów wdrożenia poświęconym mapowaniom obiektowo-relacyjnym (*orm.xml*). Pola klas encji w większości są typów prymitywnych bądź wrapperów na te typy. Dla takich pól istnieją w specyfikacji domyślne odwzorowania. W kilku miejscach używane są stałe wyliczeniowe (najczęściej opisujące stan — zadania lub hosta), dla których sprecyzowano dodatkowo semantykę utrwalania — w formie reprezentacji łańcuchowej. Poza tym dostęp do atrybutów relacji określony został za pośrednictwem metod (ang. *property-based access*) a nie pól danej klasy (ang. *field-based access*). Umożliwiło to ukrycie wewnętrznej implementacji bez definiowania dodatkowych klas jako encji lub jako obiektów osadzonych — `@Embedded`. Przykładem może być chociażby atrybut `MacAddress`, który pomimo tego, że w klasie `HostConfiguration` widnieje jako pole typu referencyjnego `MacAddress`, dla którego nie istnieje domyślne odwzorowanie, został przedstawiony w bazie danych jako atrybut typu *VARCHAR*. Zostało to osiągnięte dzięki dostępowi do pola poprzez metodę zwracającą obiekt typu `String`, wewnątrz której dokonywana jest odpowiednia walidacja oraz konwersja. Oczywiście, jednocześnie zachowano możliwość odczytu i zmiany właściwego pola typu `MacAddress`, z tym że metody dostępowe zostały oznaczone adnotacją `@Transient`, co oznacza, że nie wymuszają one utrwalania tej własności. Klucze główne wszystkich relacji są automatycznie generowane przy

użyciu tzw. generatorów sekwencji, co gwarantuje adnotacja `@GeneratedValue`.

Związki pomiędzy encjami. Jak można zauważyć na wcześniej przedstawionym diagramie związków-encji, w schemacie bazy danych istnieje kilka różnego rodzaju związków. Zostały one zdefiniowane również za pomocą adnotacji umieszczonych bezpośrednio w kodzie źródłowym klas powiązanych ze sobą komponentów encyjnych. Dzięki temu odpowiednie klucze obce czy tabele łączące związki wiele do wielu są generowane automatycznie. Dodatkowo prostsza jest też nawigacja na poziomie modelu obiektów przy użyciu języka EJB QL. Encja `Job` zawiera referencję do kolekcji grup hostów preferowanych przez użytkownika (jednokierunkowa relacja wiele do wielu). Obrazuje to sytuację, gdy użytkownik wysyłający zadanie może wybrać grupy hostów (reprezentujących np. laboratoria), co do których wie, że będą one wolne na czas obliczeń. Jeden użytkownik może wysyłać wiele zadań (stąd relacja jeden do wielu pomiędzy tabelą `Account` a `Job`). Po udanym wdrożeniu, zadanie wskazuje również na grupę konkretnych hostów, które zostaną wykorzystane dla jego celów. Oczywiście pojedynczy host może być powiązany z wieloma zadaniami (biorąc pod uwagę zadania, które są już zakończone), nie może natomiast być częścią więcej niż jednej grupy. Fakty te ilustrują odpowiednie związki, w skład których wchodzi komponent encyjny `HostConfiguration`.

Współbieżny dostęp do bazy danych. Jednym z podstawowych problemów jakie trzeba rozwiązać wykorzystując bazę danych jest współbieżny dostęp do danych. Równolegle wykonywane transakcje mogą naruszyć spójność systemu i spowodować wiele nieprzyjemnych konsekwencji. Co prawda w zaprojektowanym systemie dostęp do bazy danych nie jest zjawiskiem częstym — współbieżny dostęp może pojawić się jedynie przy okazji korzystania z interfejsu WWW — jednakże mimo wszystko należało zminimalizować ryzyko wystąpienia anomalii. Istnieje wiele algorytmów zarządzania współbieżnym wykonywaniem transakcji, większość rozwiązań opartych jest na mechanizmach blokowania, które zakładają pesymistyczny scenariusz konfliktów z czego wynika konieczność zakładania blokad i wycofywania transakcji — algorytmy pesymistyczne. Jednakże w systemach, w których obciążenia bazy danych są niskie a transakcje krótkie, prawdopodobieństwo wystąpienia konfliktu, który spowoduje nieuszeregowalność realizacji jest wyjątkowo niskie. Takie właśnie założenie stanowi podstawę tzw. algorytmów optymistycznych (ang. *Optimistic Locking*), których użycie wspierane jest przez JPA i które zostały zastosowane w systemie. Specyfikacja zakłada dodatkowo, że poziom izolacji transakcji w systemie zarządzania bazą danych będzie ustawiony na *read-committed*. W celu zapewnienia przenośności i wsparcia dla algorytmów optymistycznych każda relacja została wzbogacona o atrybut z adnotacją `@Version`, używany jedynie przez dostawcę implementacji JPA.

Jawne zarządzanie transakcjami. Używając interfejsu JPA w środowiskach JEE zarządzanie transakcjami odbywa się automatycznie, niejawnie, na poziomie wywoływanej metody. Kontekst utrwalania (ang. *persistence context*) propagowany jest w ramach aktualnej transakcji JTA (*Java Transaction API*) i synchronizowany pomiędzy managerami encji (ang. *Entity Manager*). Wszystkim tym zarządza kontener JEE — tryb ten nazywany jest *container-managed*. Alternatywnym podejściem — które jest jedyną możliwością w środowiskach Java SE i jako takie zostało użyte w opisywanym systemie — jest tryb *application-managed*. W tym przypadku zarządzaniem transakcjami zajmuje się sama aplikacja, przy użyciu `Entity Transaction API`.

Aby ujednolicić sposób korzystania z bazy danych wyspecyfikowano interfejsy udostępniające metody pozwalające na przeszukiwanie, aktualizowanie, usuwanie oraz odczytywanie obiektów w bazie danych — np. `AccountDBService`. Implementacją każdego z tych interfejsów jest odpowiedni zarządca — np. `AccountDBManager`, w którego metodach odbywa się pobieranie referencji do `EntityManagera` z fabryki `EntityManagerFactory`, zarządzanie transakcjami oraz obsługa wyjątków związanych z bazą danych. W bardziej zaawansowanych metodach managerów konstruowane są zapytania języka EJB QL przy użyciu `Query API`.

4.3.2 Moduł monitorująco-wykonawczy — Monitor

Moduł ten pełni dwie główne role. Pierwszą z nich jest monitorowanie aktualnego stanu zdalnych węzłów (definiowanego przez aplikację Agenta) dzięki komunikacji z nimi z użyciem określonych protokołów oraz wygodne udostępnianie tych informacji. Za wygodny sposób udostępniania informacji wybrano ogólnie system niezawodnych asynchronicznych powiadomień o zmianie stanu zdalnego węzła. Usługa ta eliminuje problem odpytywania Agenta przy każdej potrzebie uzyskania informacji — realizując w pewnym stopniu koncepcję pośrednika (ang. *proxy*) w dostępie do tych informacji. Wyodrębnia również kod odpowiedzialny za obsługę protokołów sieciowych poza interfejsy dostępu do danych prezentujących stan węzła.

Drugą rolą tego modułu jest wykonywanie zleconych mu operacji na zdalnych węzłach (z użyciem innego zbioru protokołów) — takich jak włączenie lub wyłączenie węzła, zatrzymanie zadania. Informacje na temat powodzenia lub niepowodzenia tych operacji, są również udostępniane za pomocą asynchronicznych powiadomień.

Oba te zabiegi skutkują skoncentrowaniem w tym module kodu odpowiedzialnego za obsługę stosu protokołów — w tym wielu połączeń, asynchronizmu i awaryjności kanałów komunikacyjnych, gniazd sieciowych (ang. *sockets*). Wydzielenie tego kodu w module Monitora czyni dostęp do danych i operacji łatwiejszy dla korzystającego z jego usług Disposera.

Protokoły używane przez Monitor do komunikacji z węzłem zdeterminowane są decyzjami projektowymi dotyczącymi komunikacji z modułem Agenta oraz jego implementacją i zostały opisane w rozdziałach dot. architektury systemu, jak i aplikacji

Agenta. Do pobierania danych o stanie węzła oraz do powiadomień o zmianie jego stanu używany jest protokół SNMP, do bezpiecznego wykonywania poleceń na węźle używany jest protokół SSH, a w celu wybudzenia — włączenia węzła — wysyłane są pakiety w standardzie WOL.

4.3.2.1 Użyta technologia

Wczesne plany projektowe dopuszczały możliwość uruchamiania pozostałych modułów Managera jako zewnętrznej dla Monitora aplikacji — np. w platformie JEE. Biorac pod uwagę możliwość działania Monitora jako samodzielnej aplikacji, jak i analizując pozostałe wyspecyfikowane wymagania wobec tego modułu, zdecydowano się na użycie technologii JMX (*Java Management Extensions*)[29] w wersji 1.4 do implementacji interfejsu udostępniającego usługi modułu Monitora. JMX jest aktualnie częścią platformy Java (zarówno wersji SE jak i EE), zdefiniowaną wcześniej w ramach JCP (*Java Community Process*)[28], jako JSR 3 (*Java Specification Request 3*)[27, 26].

JMX definiuje zestaw narzędzi i standardów dla procesu zarządzania aplikacjami oraz, rzadziej, urządzeniami — jak w przypadku tego systemu. W kontekście JMX mówi się często o 3–4 warstwowej architekturze systemu zarządzania [11], nawiązującej do ogólnych architektur rozproszonych systemów zarządzania. Koncepcję tę przedstawia rysunek 4.2.

Poszczególne warstwy i ich role w architekturze JMX to:

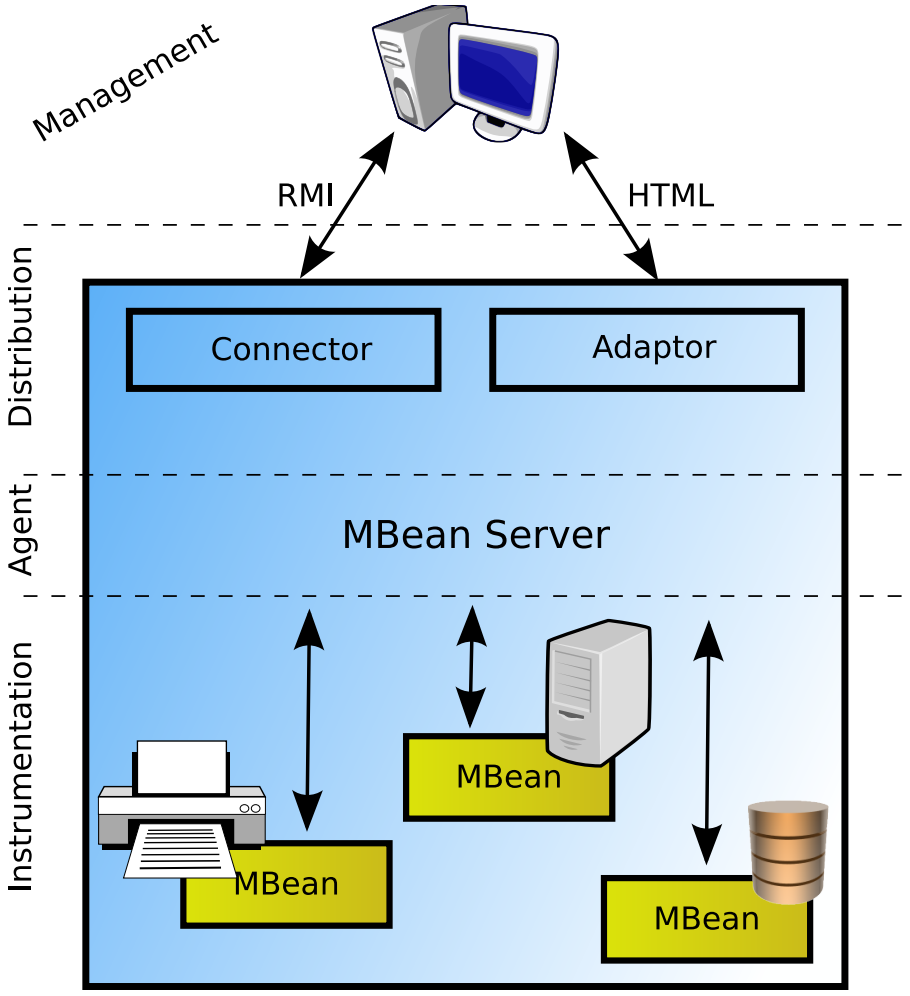
- *Warstwa instrumentacji* — silnie powiązana z *zarządzanymi zasobami*, zawierająca ich bezpośrednie modele-instrumentacje w postaci tzw. *managed beans* (w skrócie: *MBeans*). MBeany nawiązują ideą do komponentów *JavaBeans* lub *EnterpriseJavaBeans* poprzez eksponowanie określonej funkcjonalności w jednym obiekcie (klasie). MBean to w JMX elementarny byt reprezentujący zarządzany zasób, udostępniający zbiór *atrybutów* reprezentujących stan lub konfigurację zasobu oraz *operacji* które można wykonać na zasobie. Możliwe jest również generowanie przez MBeany tzw. *powiadomień* (ang. *notifications*).

- *Warstwa agenta* — składa się przede wszystkim z *serwera MBeanów* udostępniającego lokalnie ich usługi: odczyt-zapis atrybutów, wykonywanie operacji, subskrypcję powiadomień oraz usługi zarządzania MBeanami: tworzenie, niszczenie, odnajdywanie za pomocą identyfikatora obiektu itp.

- *Warstwa dystrybucji* — rolą tej warstwy jest wyeksportowanie usług agenta JMX poza zasięg wyłącznie lokalny (wirtualnej maszyny), za pomocą uruchamianych z serwerem MBeanów *złączy* (ang. *connectors*) oraz *adapterów* (ang. *adaptors*) — umożliwiających odpowiednio pełną zdalną kontrolę nad serwerem MBeanów np. poprzez wykorzystanie protokołu RMI-IIOP oraz wy-

eksportowanie usług w postaci protokołu, który z założenia nie służy z założenia do wykonywania operacji np. SNMP lub serwisu WWW (HTTP i HTML).

- *Warstwa zarządzania* (dwie ostatnie warstwy są niekiedy traktowane jako jedna warstwa *zdalnego zarządzania*) — w tej warstwie znajdują się same aplikacje monitorujące lub zarządzające, tj. prezentujące stan zarządzanych zasobów lub wykonujące określone operacje zdeterminowane stanem tych zasobów. Aplikacje te mogą znajdować się na innych węzłach niż zarządzane zasoby oraz agent JMX.



Rys. 4.2: Proces zarządzania w 4-warstwowej architekturze JMX.

Podstawową rolą implementującego interfejs zarządzania istniejącymi zasobami jest, w technologii JMX, zapewnienie implementacji MBeanów. Istnieją różne rodzaje MBeanów różniące się m.in. możliwościami użycia w uniwersalnych aplikacjach zarządzających i sposobem implementacji. Najprostsza wersja MBeanów, zwana *standardową*, polega na statycznym zdefiniowaniu interfejsu MBeana — zgodnego z konwencją tradycyjnych JavaBeans [24] — i utworzenie klasy implementującej ten

interfejs. *Metody dostępowe* (tzw. *gettery-settery*) zdefiniowanego interfejsu będą służyć do odczytywania bądź zapisywania atrybutów, a pozostałe metody do wykonywania operacji. Wyposażenie MBeana w mechanizm subskrybowanych powiadomień sprowadza się do implementacji interfejsu `NotificationEmitter` i wywoływania odpowiednich metod odpowiedzialnych za emisję powiadomienia wewnątrz kodu implementującego MBeana.

W technologii JMX instancje MBeanów na serwerze są identyfikowane poprzez przypisany `ObjectName`, składający się z opcjonalnej nazwy domeny zarządzania i zbioru par klucz-wartość, np. `dummy.net:id=1`. MBean jest rejestrowany w serwerze pod określoną nazwą-identyfikatorem i przy pomocy tej konstrukcji wykonywane są również zapytania o MBeana, wszelkie operacje na nim.

Platforma Java SE dostarcza standardową implementację serwera MBeanów, jak i zestaw złączy. Dostarczana jest również prosta aplikacja monitorująca *JConsole*, umożliwiająca monitorowanie zasobów maszyny wirtualnej za pomocą mechanizmu JMX, jak i niesystemowych MBeanów — np. podczas ich testowania. Wykorzystanie MBeanów we własnej aplikacji zarządzającej jest możliwe przez użycie klasy będącej stroną kliencką złącza JMX — implementującej interfejs `MBeanServerConnection`, która pełni rolę pośrednika w połączeniu z właściwym serwerem. Klasy do obsługi standardowych złączy są także dostarczane z platformą Java SE. Możliwe jest również używanie tworzonych w czasie wykonania pośredników implementujących interfejsy konkretnych MBeanów.

4.3.2.2 Realizacja architektury JMX

Jako, że w implementowanym systemie aplikacja Agenta (nie w rozumieniu JMX, a modułu implementowanego systemu) uruchamiana na zdalnym węźle nie wykorzystuje platformy Java, nie wykorzystano technologii JMX na węzłach obliczeniowych[1]. Jest ona jednak używana jako pośrednik w dostępie do informacji i operacji zdalnego węzła. Integracja modułów Monitora i Disposera, poprzez złącza JMX umożliwia spełnienie stawianego celu — wyodrębnienia implementacji mechanizmów komunikacji i monitorowania węzła w niezależnym module, opisywanym Monitorze.

Warstwa instrumentacji. Zdecydowano się na zdefiniowanie jednego typu MBeana, rodzaju *standardowego*, który modeluje pojedynczy zarządzany węzeł obliczeniowy. Różne jego instancje reprezentują więc różne węzły, a identyfikowane są przez `ObjectName` — przyjęto konwencję identyfikacji przez parę o kluczu `address`, a wartości odpowiadającej tekstowej reprezentacji adresu IP zarządzanego węzła. Interfejs nazwano `RemoteHostMBean`. Dziedziczy on z innych zdefiniowanych interfejsów —

[1] W rzeczywistości agent JMX jest uruchamiany na węzłach, gdy są na nich wykonywane obliczenia, ponieważ jest on składową biblioteki ProActive uruchamianą razem z Nodem ProActive. Informacje z agenta JMX ProActive nie są jednak wykorzystywane przez aktualną wersję systemu, natomiast mogą być wykorzystywane przez użytkownika np. przez użycie aplikacji IC2D dostarczanej wraz z biblioteką ProActive.

MBeanów, których znaczenie jest następujące:

- `SnmpRemoteHostMBean` — udostępnia parametry i operacje związane z mechanizmem monitorowania stanu węzła z użyciem protokołu SNMP (zaimplementowanym również w Agencie), jak i stan operacji monitorowania.

- `WolRemoteHostMBean` — udostępnia parametry i operacje służące do zdalnego wybudzania węzła z użyciem standardu WOL.

- `SshRemoteHostMBean` — udostępnia parametry i operacje służące do wykonywania zbioru komend na zdalnym węźle, implementowanych przez aplikację Agenta i wywoływanych za pomocą protokołu SSH.

- `TimeoutGeneratorMBean` — pomocniczy MBean, udostępnia operacje do generowania powiadomień po upływie pewnego czasu.

- `NotificationEmitter` (dostarczany z JMX) — umożliwia subskrypcję powiadomień z MBeana i pobranie informacji o dostępnych powiadomieniach.

- `AddressableMBean` — udostępnia atrybut z adresem węzła.

Implementacja tego interfejsu — `RemoteHost`, wykorzystuje szereg klas pomocniczych implementujących używane protokoły, opisanych szczegółowo w punkcie 4.3.2.3. Specyfikacja serwera MBeanów nie daje żadnych gwarancji dotyczących kontroli współbieżności w wywoływaniach metod implementowanego MBeana. Dlatego implementacja sama zapewnia bezpieczny współbieżny dostęp do metod (operacji, atrybutów) MBeana przez wykorzystanie standardowych mechanizmów synchronizacji w języku Java.

Zdefiniowany MBean dostarcza zbioru *atrybutów*:

- `address` (do odczytu) — jako jedyny podawany w konstruktorze; nie można go zmieniać, ponieważ służy jako wspomniany identyfikator MBeana oraz wiąże się z połączeniami na konkretne węzły, których nie można przerywać w trakcie monitorowania.

- `stateSnapshot` (do odczytu) — najważniejszy atrybut, reprezentujący migawkę (ang. *snapshot*) stanu zarządzanego węzła. Określa ona dostępność węzła (odpowiada, nie odpowiada, odpowiada błędnie lub jeszcze nie odpowiedział) oraz, w przypadku dostępności węzła, uruchomione procesy zadań i dostępność do obliczeń (czy na węźle pracują użytkownicy).

- *Parametry protokołów* (do odczytu i zapisu) — związane z implementacją konkretnych protokołów, np. parametr `sshTimeout` określający maksymalny czas na wykonanie polecenia na zarządzanym węźle; szerzej opisane w dalszej części oraz w dokumentacji Javadoc[25] do klasy `RemoteHost`.

Interfejs dostarcza również zbiór operacji na węźle. Większość z nich posiada wersje blokujące oraz nie blokujące, które nie różnią się semantyką poza możliwym blokowaniem wywołującego wątku. Udostępniany zbiór operacji zawiera:

- *operacje związane z monitorowaniem* — aktywacja i dezaktywacja monitorowania stanu węzła (np. `startMonitoring()`), wymuszenie natychmiastowego odpytania o stan;
- *operacje wykonujące polecenia na węźle* — realizowane z użyciem protokołu SSH lub standardu WOL, np. `shutdown()` wyłącza węzeł przez zdalne wywołanie polecenia;

Założona koncepcja asynchronicznych powiadomień w module mogła zostać zrealizowana przez mechanizm powiadomień JMX — realizację wzorca projektowego obserwatora (ang. *observer*)[7] dla tej technologii. Wysłanie powiadomienia polega na wywołaniu kodu obsługującego powiadomienie u subskrybentów powiadomienia. Ze względu na stosowane mechanizmy synchronizacji w implementacji `RemoteHost`, aby uniknąć możliwości zakleszczenia (ang. *deadlock*) dwóch wątków, wszystkie powiadomienia emitowane były przez osobny wątek, w kolejności ich wygenerowania.

`RemoteHostMBean` używa dwóch typów powiadomień, które są w technologii JMX klasami:

- `AttributeChangeNotification` (dostarczana z JMX) — powiadomienie emitowane w celu informowania o zmianach monitorowanego `stateSnapshot`. Obiekt tego powiadomienia zawiera m.in. starą i nową wartość atrybutu.
- `OperationFinishedNotification` — powiadomienie emitowane po zakończeniu jakiejkolwiek operacji (zarówno wersji blokującej jak i nie blokującej) wykonującej polecenie na węźle. Obiekt tego powiadomienia zawiera m.in. nazwę wykonywanej operacji, jak sposób zakończenia operacji — powodzenie (np. wysłanie pakietu WOL) lub niepowodzenie (np. zerwanie sesji SSH), wraz z opcjonalnym rezultatem — np. informacją tekstową nt. błędu.

Warstwa agenta i dystrybucji. W implementacji użyto standardowego serwera MBeanów dostarczanego z platformą Java SE, którego uruchomienie odbywa się przez użycie stosownej klasy ze standardowego zbioru bibliotek. Główny kod agenta umieszczono w klasie `Monitor`, której jedyna instancja jest współdzielona (*singleton*). Pełni ona trzy główne funkcje:

- Uruchamia serwer MBeanów i przechowuje jego instancję.
- Przechowuje *współdzielone zasoby* używane przez instancje MBeanów:
  - serwer odpytywania SNMP ze swoim wątkiem i gniazdem sieciowym;
  - wątek emitujący powiadomienia JMX;

  - pule wątków dla sesji SSH;
  - wątek wysyłający pakiety WOL;
  - usługa alarmu dla generowania powiadomień po określonym czasie;
  - gniazdo sieciowe używane do wysyłania pakietów WOL;

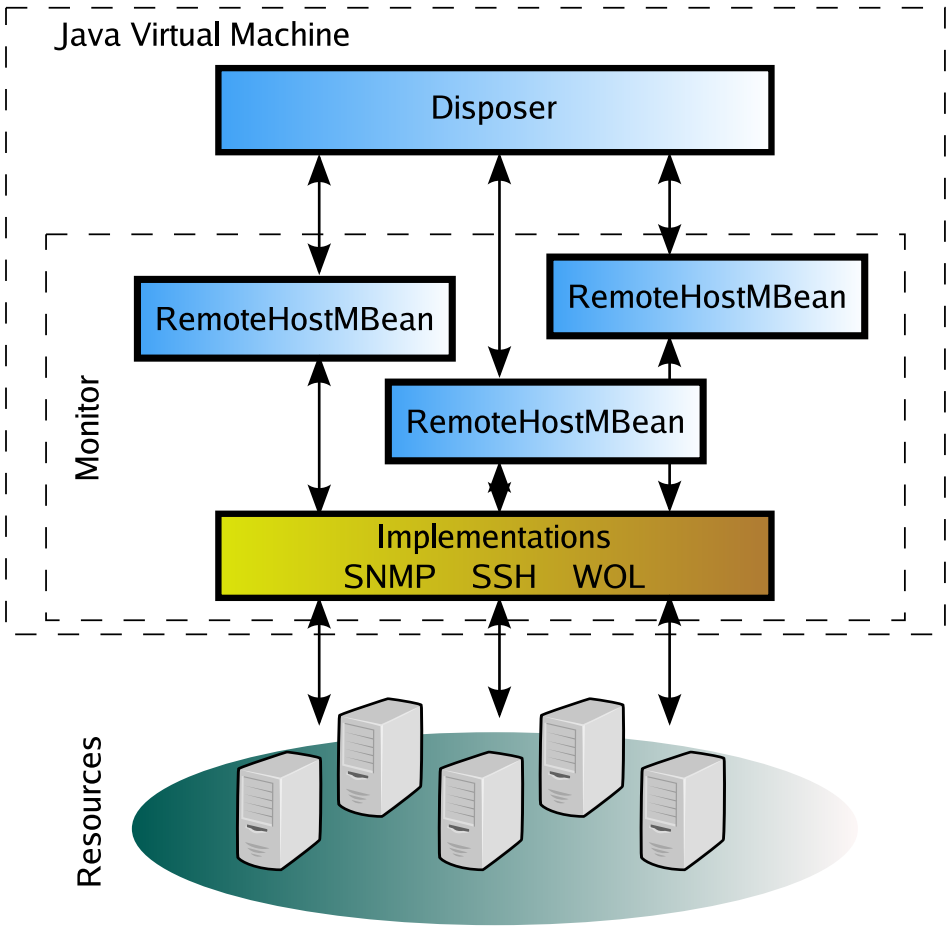
- Udostępnia metody dostępowe dla Disposera znajdującego się w tej samej maszynie wirtualnej; metody te umożliwiają tworzenie i usuwanie MBeanów `RemoteHost` dla określonych adresów węzłów.

Ze względu na ostateczne uruchamianie modułów Monitora wraz z pozostałymi wewnątrz tej samej aplikacji (maszyny wirtualnej), nie było konieczne używanie złączy JMX do komunikacji pomiędzy Disposerem a Monitorem. Wystarczyło odwoływanie się wprost do lokalnego serwera z użyciem metod z klasy `Monitor`. Klasą implementującą klienta złącza JMX jest w tym przypadku sam serwer MBeanów. Mimo tego, przy znikomych zmianach w modułach Monitora i Disposera, nadal potencjalnie możliwe jest uruchamianie Monitora na innym komputerze niż reszty Managera, przez zastosowanie złączy JMX. Miałoby to sens przy założeniu ścisłego powiązania obu komputerów (ta sama, niezawodna, bezpieczna podsieć; małe opóźnienia w komunikacji). Co więcej, możliwe wydaje się użycie w takim wypadku kilku komputerów z uruchomionymi modułami Monitora, z mechanizmem prostego równoważenia obciążenia przez Managera np. na bazie liczby MBeanów w każdym z Monitorów. Mogłoby to zwiększyć skalowalność systemu zarządzania — umożliwić obsługę większej liczby węzłów.

Rysunek 4.3 prezentuje ostateczny obraz architektury modułu i połączenia z Disposerem — w postaci lokalnych odwołań wewnątrz jednej maszyny wirtualnej.

4.3.2.3 Implementacja funkcji i protokołów sieciowych

Monitorowanie stanu węzła. Agent SNMP uruchamiany na węzłach (zob. punkt 4.1.2) udostępnia jego stan w dwojaki sposób. Pasywnie — przez udostępnienie reprezentujących stan zmiennych w drzewie MIB, o których wartość można odpytać Agenta, jak i aktywnie — przez samodzielne wysyłanie przez Agenta asynchronicznych powiadomień o zmianie stanu, nazywanych w nomenklaturze SNMP *traps*. Trapy udostępniają informacje o zmianie stanu, krótko po jego faktycznej tranzycji. Przy założeniu niezawodnych kanałów komunikacyjnych i niezawodnych węzłów (tutaj: np. nie wyłączanych bez zapowiedzi), odbieranie trapów wysyłanych przez Agenta byłoby wystarczające do utrzymania aktualnego stanu węzła w Monitorze. Jednak w bliższych rzeczywistości modelach węzłów i komunikacji, dla utrzymania dobrego estymatora stanu węzłów potrzebne jest dodatkowo regularne odpytywanie o ich stan (ang. *polling*). Jest to w założeniu redundantny, w stosunku do trapów, sposób pozyskiwania informacji nt. stanu węzła, który staje się potrzebny, gdy np. zaginie datagram z trapem lub węzeł zostanie fizycznie wyłączony przez użyt-



Rys. 4.3: Architektura modułu Monitora w kontekście Disposera.

kownika. Taki model monitorowania z wykorzystaniem trapów i odpytywania jest typowy dla aplikacji monitorujących sieci komputerowe.

Monitorowanie stanu węzłów realizowane jest w oparciu o otwartą implementację protokołu SNMP dla języka Java — bibliotekę SNMP4J[44]. Biblioteka ta udostępnia zestaw klas umożliwiających m.in. manipulację PDU (*Protocol data unit*) SNMP na względnie wysokim poziomie abstrakcji, oraz wysyłanie i odbieranie PDU z wykorzystaniem pomocniczych klas. Rolą programisty używającego biblioteki jest przede wszystkim konstruowanie wysyłanych PDU, jak i przetwarzanie odbieranych PDU. Udostępnione są przejrzyste mechanizmy asynchronicznego przetwarzania odpowiedzi lub trapów przez wykorzystanie wzorca obserwatora — zdefiniowane są interfejsy `ResponseListener` oraz `CommandResponder`. Przetwarzanie trapów jest więc stosunkowo proste. Możliwe jest również łatwe zlecanie asynchronicznego wysyłania wiadomości (bez blokującego oczekiwania na odpowiedź). Brakuje natomiast implementacji mechanizmów regularnego odpytywania — konieczne było więc zaimplementowanie takiej funkcji.

Główną częścią implementacji jest generyczna klasa `SnmpVariableMonitor<T>`, której zadaniem jest udostępnienie usługi monitorowania pewnych obiektów w drze-

wie MIB (*Management Information Base*) określonego węzła. W celu efektywnego monitorowania łączy ona opisane techniki — nasłuchiwania na nadchodzące trapy i regularnego odpytywania. Instancje `SnmpVariableMonitor<T>` związane są ze zbiorem parametrów definiowanych przez programistę, m.in.:

- `address` — adres monitorowanego węzła;
- `server` — serwer odpytywania, opisany dalej;
- `pduProcessor` — implementacja zdefiniowanego interfejsu przetwarzającego PDU: `SnmpPDUProcessor<T>`. Jej rolą jest dostarczanie PDU wysyłanych w celu pobierania wartości zmiennych (w takim PDU umieszcza się zapytania o interesujące zmienne), jak i przetwarzanie nadchodzących PDU — odpowiedzi na zapytania o zmienne i trapów.
- *parametry* SNMP i odpytywania:
  - `snmpCommunity` — parametr protokołu SNMP (wersji 2c) używany do prostego uwierzytelniania jawnym tekstem;
  - `pollingInterval` — interwał odpytywania;
  - `pollingRetries` — liczba ponawianych prób pojedynczego zapytania o stan (np. w przypadku zaginięcia datagramu UDP);
  - `pollingTimeout` — limit czasu na pojedynczą odpowiedź;

Pondato klasa dostarcza kilku operacji, z których ważniejsze to włączanie i wyłączanie monitorowania (`startMonitoring()`, `stopMonitoring()`) — w szczególności aktywnego odpytywania. Celem stosowania tej klasy jest dostarczenie aktualnej wersji obiektu reprezentującego stan zdalnego węzła, o generycznym typie `T` — określanym przez programistę. Dostarczany procesor `SnmpPDUProcessor<T>` przy przetwarzaniu odebranego PDU zwraca m.in. obiekt typu `T` określający odczytany z wiadomości stan. `SnmpVariableMonitor<T>` enkapsuluje jednak typ `T` w klasie `Snapshot<T>`, tj. migawce (ang. *snapshot*) zmiennej. Klasa ta definiuje trzy składowe:

- `snapshotType` — rodzaj migawki:
  - `VALUE` — odpowiedź związana z wartością, gdy monitorowany węzeł odpowiada na zapytania.
  - `NO_ANSWER` — brak odpowiedzi, gdy monitorowany węzeł nie odpowiada na zapytania (np. jest wyłączony).
  - `UNKNOWN` — stan nieznany, gdy monitorowany węzeł nie odpowiedział jeszcze na pierwsze zapytanie, ani nie minął jeszcze `pollingTimeout`.
  - `ERROR` — błąd wraz z opisem, gdy nie udaje się ustalić stanu monitorowanego węzła (np. zwraca on nieoczekiwane PDU).

- `value` — opcjonalna wartość, typu `T`;
- `error` — opcjonalny opis błędu;

Tak zamodelowana migawka reprezentuje możliwy stan mechanizmu odpytywania, jak i (w typowym przypadku) zawiera obiekt reprezentujący stan zdalnego węzła. `SnmpVariableMonitor<T>` kontroluje bieżący stan migawki, udostępnianej pod składową `variableSnapshot`. W przypadku odnotowania jakiejkolwiek zmiany w migawce (tj. jej typu, lub związanej z nią wartości, błędu), informuje o tym rejestrowalnych obserwatorów, implementujących interfejs `SnapshotListener<T>`.

Problemem w mechanizmie odpytywania jest jednak konieczność jego *aktywnego* wykonywania, tj. periodycznego wysyłania zapytania. Aby nie tworzyć dużej liczby wątków, które przez większość czasu pozostawałyby nieaktywne, utworzono współdzielony między instancjami `SnmpVariableMonitor<T>` serwer odpytywania — `SnmpVariableMonitorPoolServer`. Każda instancja `SnmpVariableMonitor<T>` rejestruje się w momencie aktywowania monitorowania do serwera, a wyrejestrowuje w momencie dezaktywowania monitorowania. Gdy `SnmpVariableMonitor<T>` jest zarejestrowany w serwerze, jest przez niego wybudzany co zażądany przez nią czas — w celu wysłania zapytania o stan zmiennej. Wymaga to zastosowania mechanizmów synchronizacji w obu klasach. Serwer wykorzystuje współdzielone gniazdo UDP do wysyłania zapytań i odbierania odpowiedzi na nie, oraz wyszczególnione gniazdo UDP do odbierania trapów, deleguje nadchodzące przez nie PDU do odpowiednich instancji `SnmpVariableMonitor<T>`. Trapy odbierane są na niestandardowym porcie 1162 (typowo jest to port 162), aby uniknąć konieczności uruchamiania aplikacji z przywilejami niezbędnymi do wykorzystania portu o numerze mniejszym niż 1024.

Implementacja wykorzystywanego MBeana — `RemoteHost`, dziedziczy po klasie `SnmpVariableMonitor<T>` w celu łatwego użycia mechanizmu odpytywania. Parametrem generycznym `T` jest tutaj reprezentacja stanu węzła zdefiniowana w klasie `RemoteHostState`. Składają się na nią pozyskiwane z Agenta wartości (por. tabela 4.2 w punkcie 4.1.2.2):

- `idleness` (o wartościach 0–100) — określa czy węzeł jest dostępny do obliczeń (wartość 100), czy też są na nim uruchomione interaktywne sesje (wartości mniejsze niż 100).
- `jobsProcessesRunning` — lista uruchomionych procesów obliczeniowych na węźle, w postaci listy identyfikatorów zadań.

`RemoteHost` używa do wysyłania i przetwarzania PDU SNMP zdefiniowaną klasę `PutgridSnmpPDUProcessor`, oraz obserwatora konwertującego powiadomienia o zmianie migawki na powiadomienia JMX, typu `AttributeChangeNotification`.

Zdalne wybudzanie węzła. Implementację zdalnego wybudzania węzła zawarto w klasie `WakeOnLanTask`, modelującej zadanie wysłania pakietu WOL. Klasa ta

implementuje interfejs `Runnable`, co pozwala na wykonanie zadań w dowolnej implementacji usługi wykonawcy — `Executor`, np. w puli wątków.

Wysyłany w `WakeOnLanTask` pakiet WOL jest odpowiednio przygotowanym datagramem protokołu UDP. Zawiera on sekwencję sześciu bajtów o wartości `FF` *hex* każdy, po której następuje szesnastokrotnie powtórzony adres MAC karty sieciowej adresata oraz opcjonalne hasło — niewykorzystywane w tej implementacji (por. tabela 4.4). Pakiet o takiej zawartości zostaje następnie rozgłoszony w obrębie danej podsieci na port 7 lub 9. Adres rozgłoszeniowy podsieci jest wyznaczany na podstawie adresu węzła i maski podsieci. Gdy pakiet zostanie rozpoznany przez adresata, który pozostaje w odpowiednim stanie poboru mocy [36], spowoduje włączenie komputera. Po bliższe informacje dotyczące ustawień specyficznych dla danej konfiguracji zestawu komputerowego, należy sięgnąć do punktu 4.1.1 oraz dokumentacji użytkownika właściwej płyty głównej.

Część stała (6B)	Adresat (96B)
`FF FF FF FF FF FF`	16 x (adres MAC karty sieciowej)

Tab. 4.4: Zawartość pakietu Wake On LAN.

Zadanie `WakeOnLanTask` tworzone jest przez MBeana `RemoteHost` przy żądaniu wybudzenia węzła. Następnie zlecane jest uruchomienie zadania w zdefiniowanej usłudze wykonawcy — `WakeOnLanExecutorThread`. Po zakończeniu zadania, jego wynik (powodzenie lub niepowodzenie) jest przekazywany przez asynchroniczne powiadomienie do obiektu-obserwatora wewnątrz MBeana. Obiekt ten przetwarza uniwersalną odpowiedź z zadania na powiadomienie JMX — opisanego wcześniej typu `OperationFinishedNotification`.

Usługa wykonawcy `WakeOnLanExecutorThread` uruchamia zlecone jej zadania `WakeOnLanTask` w kolejności zgłoszeń, korzystając przy tym tylko z jednego wątku. Nie ogranicza to prędkości wysyłania pakietów ze względu na bezpołączeniową naturę protokołu UDP. Zadania są jednak wykonywane z przerwą wynoszącą minimalnie 100ms (wartość arbitralnie ustalona). Ma to zapobiec sytuacji jednoczesnego włączania dużej liczby komputerów, która ze względu na duży chwilowy pobór prądu podczas uruchamiania komputerów mogłaby spowodować przeciążenie sieci energetycznej laboratorium, a w konsekwencji przerwanie obwodu przez zabezpieczenie elektryczne.

Zdalne wykonywanie poleceń na węźle. Do zdalnego wykonywania poleceń Agenta wykorzystano otwartą bibliotekę implementującą protokół SSH-2 w języku Java: Trilead SSH-2 for Java[48]. Udostępnia ona interfejs wysokiego poziomu dla operacji takich jak połączenie, uwierzytelnienie, zdalne wykonanie poleceń z użyciem protokołu SSH-2. Użycie biblioteki opiera się na wywoływaniu blokujących metod dla tego typu operacji. Operacje te wykorzystują niejawnie tworzony (w im-

plementacji API Trilead SSH-2) wątek obsługujący połączenie.

Biorąc pod uwagę charakter interakcji z używaną biblioteką, zdefiniowano klasę `SshCommandTask` modelującą zadanie wykonania polecenia przez SSH, na określonym węźle. Klasa ta implementuje interfejs `Runnable`, co umożliwia wykonywanie takich zadań w puli wątków. Zadanie składa się z kilku kroków:

1. Nawiązanie połączenia z zarządzanym węzłem i uwierzytelnienie metodą klucza publicznego[47] — zalogowanie się na konto o konfigurowalnej nazwie. Aby uwierzytelnianie mogło odbywać się w trybie nieinteraktywnym, używany jest klucz prywatny RSA lub DSA konta Monitora, oczekiwany w standardowych lokalizacjach — `.ssh/id_rsa`, `.ssh/id_dsa` w katalogu domowym lub wyspecyfikowanej w konfiguracji. Klucz publiczny tego konta musi znajdować się w zbiorze kluczy autoryzowanych na docelowym koncie węzła (por. 4.1.3).

2. Utworzenie kanału SSH do wykonywania poleceń i wywołanie żądanego polecenia (patrz punkt 4.1.3), oczekiwanie na zwrócenie kodu przez uruchamiany program lub anulowanie operacji po upłynięciu pewnego (konfigurowalnego) czasu.

3. Poinformowanie obserwatorów o poprawnym wykonaniu polecenia lub błędzie.

4. Zamknięcie kanału, połączenia.

Takie zadanie jest tworzone przez MBeana `RemoteHost` przy każdym żądaniu wykonania operacji przez SSH. Następnie zlecane jest uruchamianie zadania we współdzielonej między MBeanami puli wątków. Wynik (powodzenie lub niepowodzenie) jest przekazywany przez asynchroniczne powiadomienie do obiektu-obserwatora wewnątrz MBeana, który przetwarza uniwersalną odpowiedź z zadania na powiadomienie JMX, typu `OperationFinishedNotification`.

4.3.2.4 Testy modułu

Ze względu na ilość przetwarzania sieciowego w implementowanym module, utrudnione było jego testowanie, w szczególności w zautomatyzowany sposób. Zdefiniowano kilka zestawów testów jednostkowych z użyciem pakietu JUnit[33] — tam gdzie przetwarzanie jest lokalne, np. w przypadku klasy reprezentującej maskę sieciową lub klasy Monitor tworzącej MBeany. Przede wszystkim jednak przeprowadzano wymagające obserwacji testy implementacji poszczególnych protokołów — z użyciem sprawdzonych serwerów SNMP, SSH oraz z wykorzystaniem analizatora pakietów (ang. *sniffer*). Te klasy testów wykorzystywane były również do kontrolowania regresji w implementacji.

W ostatnim etapie implementacji przeprowadzono, pomyślnie zakończone, testy integracyjne z modułem Agenta. Testy polegały na monitorowaniu z użyciem MBeana węzła z uruchomionym Agentem oraz wywoływaniu operacji dostępnych w MBeanie, przy różnym stanie węzła (a w konsekwencji MBeana).

4.3.3 Moduł zarządzający węzłami — Disposer

Posiadając dostęp do usług pozostałych modułów, Disposer może realizować swoje zasadnicze cele, tj. kontrolować zasoby — węzły obliczeniowe, oraz przetwarzać zgłoszenia zewnętrzne — dotyczące operacji na zadaniach lub węzłach. Moduł ten używa Monitora jako źródła informacji o stanie węzłów oraz jako wykonawcy operacji na węźle. Moduł DB jest używany zarówno do pobierania pewnych danych (np. konfiguracji węzłów), jak i zapisywania informacji dotyczących stanu przetwarzania zadań. Moduł Web Services jest natomiast źródłem zgłoszeń od użytkowników lub innych aplikacji.

Zarządzanie węzłami i zadaniami jest w kontekście implementowanego systemu dość złożonym procesem, m.in. ze względu na dynamikę zarządzanego środowiska. Zmiany w tym środowisku nie są tylko wynikiem operacji na węzłach wykonywanych przez sam system — takich jak uruchomienie zadania lub włączenie komputera, ale też działań lokalnych użytkowników. Co więcej, w większości przypadków działania lokalnych użytkowników (użytkowanie komputera) węzłów mają priorytet nad obliczeniami lub ingerują w nie (np. poprzez wyłączenie komputera), ponieważ używane węzły nie są dedykowane tylko do obliczeń. Algorytmy zarządzania węzłami i zadaniami muszą więc brać pod uwagę dwa *założenia*:

- Manager, a w nim właśnie algorytmy zarządzajace, nie jest jedynym decydentem w systemie. Są nimi także lokalni użytkownicy komputerów. Ponadto inne, niezależne od zarządcy zjawiska (takie jak wyłączenie zasilania lub błędna konfiguracja węzłów) również mogą skutkować stanem węzłów innym niż wymuszany przez Managera.

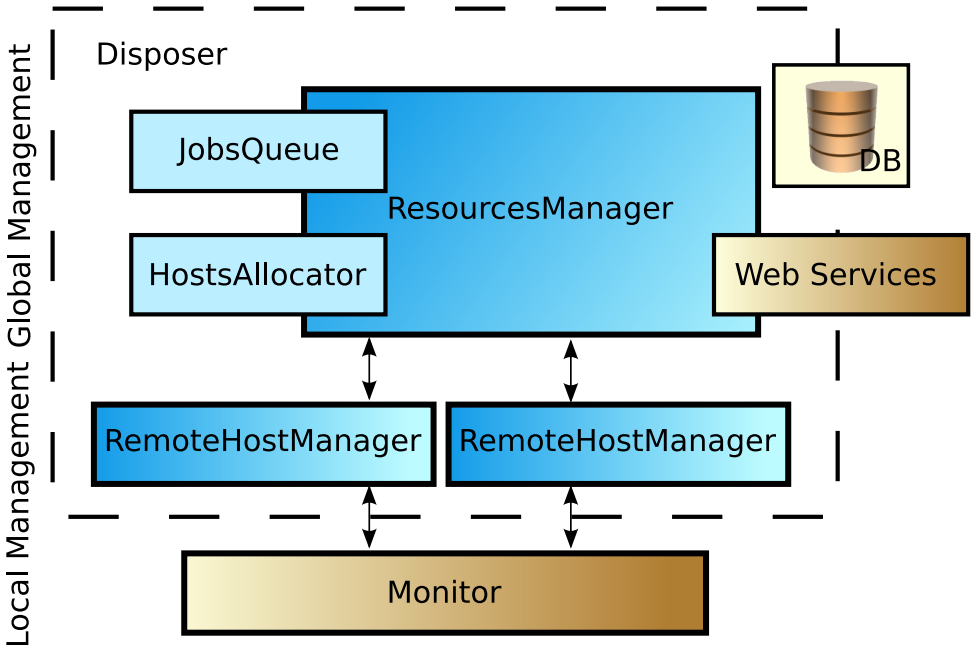
- Wykonywanie obliczeń nie jest jedynym celem zarządzania. Jest nim również umożliwienie normalnej pracy lokalnych użytkowników komputerów.

Te założenia komplikują algorytmy odpowiedzialne za zarządzanie węzłami.

4.3.3.1 Architektura modułu

Biorąc pod uwagę przewidywaną złożoność procesu zarządzania węzłami, zdecydowano się na prostą dekompozycję problemu poprzez zastosowanie dwupoziomowego (hierarchicznego) systemu zarządzania. Schemat tej koncepcji przedstawia rysunek 4.4.

Niższy poziom to *zarządzanie pojedynczymi węzłami*. Każdy zarządzany węzeł ma przypisanego jednego zarządcę tego rodzaju. W naturalny sposób umożliwia to integrację z warstwą wykonawczą (Monitorem) — przez połączenie z MBeanem, odpowiadającym również jednemu węzłowi. Decyzje podejmowane na tym poziomie są rozpatrywane wyłącznie w kontekście tego węzła. Możliwe jest przyjmowanie zleceń operacji takich jak np. przygotowanie węzła do obliczeń (włączenie węzłą i oczekiwanie na jego obudzenie lub błąd), zatrzymanie zadania, oczekiwanie na uruchomienie



Rys. 4.4: Architektura modułu Disposer (w kontekście reszty Managera).

zadania. Zlecenia te przyjmowane są od zarządcy wyższego poziomu lub bezpośrednio z Web Services, gdy administrator wymusza pewne operacje. Zarządca reaguje również na informacje z węzła — np. zatrzymuje uruchomione w nim procesy zadań gdy zaloguje się użytkownik, albo wyłącza włączony przez siebie komputer, gdy nie jest on używany przez dłuższy czas. Do zarządcy wyższego poziomu przekazywane są jedynie proste informacje — o potencjalnej *dostępności* węzła do obliczeń lub o *stanie uruchamiania zleconego zadania obliczeniowego*. Informacje na temat aktualnego stanu węzła są również udostępniane z tego poziomu przez Web Services (a w konsekwencji mogą być obserwowane np. przez serwis WWW).

Wyższy poziom to *globalne zarządzanie zasobami*. Istnieje tylko jeden zarządca na tym poziomie, przechowuje on referencje do zarządców węzłów. Na tym poziomie zlecenia przyjmowane są tylko z Web Services i dotyczą zadań — ich zakolejkowania lub anulowania. Przetwarzane są również informacje z zarządców węzłów. Na podstawie obu tych rodzajów informacji podejmowane są decyzje dotyczące przydziału zasobów, uruchamiania zadań obliczeniowych. Decyzje te wiążą się ze zlecaniem wykonania operacji zarządcom węzłów (takich jak przygotowanie ich do obliczeń) oraz dokonywaniem zmian w bazie danych (np. usunięcie zadania z kolejki, zmiana jego stanu).

4.3.3.2 Zarządzanie pojedynczym węzłem

Zdefiniowane reguły zarządzania. Proces zarządzania węzłem bazuje na ustalonym zbiorze reguł definiujących dozwolone wykorzystanie pojedynczego węzła w zależności od jego stanu i konfiguracji. Wskazują one konsekwencje wystąpie-

nia określonych stanów. Reguły te zostały ustalone biorąc pod uwagę wymagania stawiane wobec systemu. Zapisane w języku naturalnym brzmią następująco:

1. Zarówno węzeł uruchomiony na żądanie systemu (przez WOL), jak i przez użytkownika, może być wykorzystany do obliczeń.

2. Węzeł włączony przez system jest przez niego wyłączany (po pewnym, konfigurowalnym czasie braku aktywności `powerHoldOnInterval`), chyba że w międzyczasie zalogował się do niego użytkownik — wówczas odpowiedzialność za wyłączenie jest po stronie użytkownika.

3. Węzeł włączony przez użytkownika nigdy nie jest wyłączany przez system.

4. Węzeł może być włączony lub wyłączony przez system tylko jeśli pozwala na to jego konfiguracja `powerManaged`. Przez węzeł włączony przez system rozumie się taki, który uruchomił się krótko po wysłaniu pakietu WOL. Podejmowanych jest maksymalnie `wakeUpRetries` powtórzeń prób włączenia węzła.

5. Zadania obliczeniowe mogą być wykonywane na węźle tylko wtedy, kiedy jest on bezczynny (żaden użytkownik nie jest na nim zalogowany) lub pozwala na to jego konfiguracja `ignoreIdleness`.

6. Zalogowanie użytkownika powoduje przerwanie wykonywania uruchomionych na nim procesów zadań, chyba że inaczej stanowi ustawienie `ignoreIdleness`.

7. Jakiekolwiek niezgłoszone zarządcy procesy obliczeniowe wykryte na węźle (uruchomione za pomocą Submittera) są zatrzymywane.

8. Maksymalna dopuszczalna liczba procesów zadań obliczeniowych na węźle to liczba jego procesorów (rdzeni) `cpuNumber`.

9. Na wykonanie poszczególnych operacji przewidziane są ramy czasowe określone przez konfiguracje `*Timeout`. Są to maksymalne czasy po których oczekiwane są konsekwencje wykonania danej operacji. Np. `wakeUpTimeout` określa maksymalny czas od zlecenia wysłania pakietu WOL, po którym komputer powinien się uruchomić. Przekroczenie któregoś z tych czasów oznacza błąd w konfiguracji lub działaniu węzła. Powoduje to natychmiastowe wyłączenie mechanizmów zarządzania węzłem, ponieważ potencjalnie nie można podejmować dalej spójnych decyzji. Błąd w wykonaniu jakiejś operacji (np. przy uruchamianiu zdalnego polecenia shutdown przez SSH) skutkuje jedynie ostrzeżeniem, jednakże zazwyczaj konsekwencją takiego błędu jest przekroczenie limitu czasu na reakcje węzła (ponieważ operacja prawdopodobnie nie została wykonana), co kończy się również wyłączeniem z zarządzania. Jedynym wyjątkiem jest limit czasu na detekcję uruchamianego przez Submitter zadania, który nigdy nie powoduje wyłączenia węzła z zarządzania.

10. Węzeł może być zarządzany lub (tymczasowo) niezarządzany. Zarządzany węzeł jest potencjalnie dostępny do obliczeń gdy nie jest zajęty. Jest on monitorowany w kontekście logowania użytkownika, uruchomionych procesów obliczeniowych. Węzeł niezarządzany nie jest monitorowany, nie są na nim wykonywany żadne operacje, ani nie jest używany do obliczeń.

11. Sytuacja, w której przez dłuższy czas (`shutdownTimeout`) niemożliwe jest ustalenie stanu węzła (zwraca on błędną odpowiedź SNMP), jest również traktowana jako błąd i powoduje natychmiastowe wyłączenie zarządzania węzłem.

12. Węzeł jest potencjalnie *dostępny do obliczeń* gdy:

    - jest wyłączony i zarządca ma prawo go uruchomić (`powerManaged`), lub
    - jest włączony i od czasu min. `maxIdlenessStabilizationInterval` nie pracuje na nim lokalny użytkownik (służy to wyeliminowaniu sytuacji uruchamiania zadań gdy użytkownik wyłącza komputer), lub
    - został właśnie włączony przez system i żaden użytkownik nie jest na nim zalogowany, lub
    - jest włączony i ustawiona jest opcja `ignoreIdleness`.

    W przeciwnym razie węzeł jest niedostępny do obliczeń — albo z powodu aktywnej sesji użytkownika, albo braku praw (ustawień) do uruchomienia komputera. Informacja na temat potencjalnej dostępności do obliczeń jest przekazywana do globalnego zarządcy.

Implementacja. Interfejs zarządcy tego poziomu został zdefiniowany pod nazwą `RemoteHostManager`. W pewnym stopniu zbliżony jest on do interfejsu MBeana z modułu Monitora, jednak semantyka operacji jest inna i nie są to jedynie delegacje operacji do MBeana. Implementacja tego interfejsu powinna działać zgodnie z przedstawionymi wyżej regułami. Metody tego interfejsu można podzielić na trzy główne grupy:

- *Metody przetwarzające powiadomienia* — `handleNotification()` przetwarza powiadomienia JMX z MBeana — dotyczą one zakończenia wykonywania operacji lub zmiany stanu węzła. Może prowadzić to do wykonania pewnych operacji na węźle w reakcji na powiadomienie.

- *Metody wykonujące operacje na węźle* — wykonują zlecone operacje takie, jak np. zatrzymanie zadania lub włączenie komputera. Taki proces polega często na przesłaniu komunikatu stosownym protokołem przez MBeana i wprowadzeniu zarządcy w stan oczekiwania na spodziewane konsekwencje danej operacji, kontrolę wykonania operacji. Większość operacji jest dostępna dla administratora przez interfejs WWW, część dla globalnego zarządcy. Dostępne operacje to:

  - `startMonitoring()` / `stopMonitoring()` — służą do aktywowania lub dezaktywowania monitorowania i zarządzania węzłem. Wyłączenie węzła z zarządzania (monitorowania) powoduje wykonanie operacji mających na celu zostawienie go w stanie zastanym, tj. np. zatrzymywane są zadania lub zostaje on wyłączony. Aktywacja lub dezaktywacja zarządzania powoduje, odpowiednio, zarejestrowanie bądź wyrejestrowanie zarządcy węzła z rejestru globalnego zarządcy.
  - `pollNow()` — wymusza natychmiastowe odpytanie o stan węzła.
  - `wakeUp()` — wybudza (włącza) komputer.
  - `shutdown()` — wyłącza komputer.
  - `stopJobExecution()` — zatrzymuje (zabija) określony procesy obliczeniowe na węźle.
  - `prepareForJobSubmit()` — (dostępna tylko dla globalnego zarządcy) przygotowuje węzeł do uruchomienia obliczeń: włącza komputer, informuje zarządcę wyższego poziomu o stanie wykonania tej operacji.
  - `cancelPrepareForJobSumit()` — (dostępna tylko dla globalnego zarządcy) odwołuje przygotowania do wykonania obliczeń, informuje o tym zarządcę wyższego poziomu.
  - `submittingJob()` — (dostępna tylko dla globalnego zarządcy) informuje zarządcę węzła o spodziewanym uruchomieniu zadania na węźle przez Submittera.
  - `testJob()` — o ile maszyna jest dostępna, wykonuje testowe zadanie w celu sprawdzenia łączności i konfiguracji SSH w węźle oraz Monitorze.

- *Metody pobierające informacje* — udostępniają aktualny stan węzła oraz statyczne informacje takie jak konfiguracja, identyfikator.

Rozpatrując możliwe implementacje interfejsu zauważono, że działania zarządcy mogą być wyzwalane powiadomieniami lub żądaniami wykonania operacji. Wszelkie zmiany wewnętrzne lub operacje na węźle są wykonywane pod ich wpływem lub po upływie pewnego czasu — co również można zamodelować jako powiadomienie, przez wykorzystanie MBeana implementującego interfejs `TimeoutGeneratorMBean`. Korzystając z tej obserwacji, postanowiono zastosować do implementacji zarządcy technikę nazywaną przetwarzaniem *sterowanym zdarzeniami* (ang. *event-driven processing*)[12]. W tym przypadku jako zdarzenia można traktować zarówno powiadomienia, jak i żądania. Jest to użyteczne podejście m.in. właśnie w kontekście asynchronicznych systemów rozproszonych, gdzie niektóre zdarzenia zachodzą w niedeterministycznym momencie. Zastosowanie tej techniki pozwala znacząco obniżyć ilość zasobów wymaganych przez aplikację, ponieważ nie ma potrzeby tworzenia wątku dla każdego uruchomionego zarządcy (potencjalnie kilkadziesiąt lub kilkaset)

— kod będący obsługą zdarzenia może być wykonywany przez jeden lub kilka wątków używanych przez wszystkich zarządców. Dodatkowo, przy odpowiedniej kontroli wywoływania obsługi zdarzeń, jak i założeniu że kod obsługujący zdarzenie wykonuje się krótko, można osiągnąć wysoką prędkość przetwarzania poprzez zmniejszenie ilości mechanizmów synchronizacji potrzebnych wewnątrz kodu obsługi zdarzeń. Takie podejście stosowane jest często w sieciowym oprogramowaniu (np. [21]), w którym krytyczna jest wydajność. W przypadku implementowanego zarządcy może to mieć znaczenie przy obsłudze wielu zdarzeń jednocześnie — np. w momencie uruchamiania zadania na dużej liczbie węzłów.

Przetwarzanie sterowane zdarzeniami ma jednak i swoje wady. Przy dużej liczbie związanych ze sobą rodzajów zdarzeń i złożonym stanie obiektu, znacząco zwiększa się złożoność implementacji, ponieważ wiele zdarzeń może nadejść w dowolnym momencie, co musi uwzględniać kod obsługujący zdarzenia. W rezultacie fragmenty kodu obsługujące zdarzenia stają się zależne od siebie nawzajem — kod jest trudny do rozwijania, zarządzania. Implementacja była więc czasochłonna, wymagała przeglądów kodu.

Realizując koncepcję przetwarzania zdarzeń, implementację interfejsu zarządcy można by przedstawić jako *maszynę stanową*, o tranzycjach stanu zdefiniowanych w metodach obsługujących zdarzenia. Jawne zdefiniowanie wszystkich możliwych stanów i tranzycji wydaje się w przypadku zarządcy trudnym zadaniem ze względu na ich bardzo dużą liczbę. Z tego powodu w implementacji poszukiwano pewnych uproszczeń: zamiast jawnej specyfikacji każdej tranzycji dla każdego stanu, zastosowano dla części z nich funkcje tranzycji zaimplementowane przez reguły — by uniknąć powtarzania kodu.

Podzielono również zbiór możliwych *zdarzeń* na dwa podzbiory:

- *Nadejście żądań* — ich obsługa zawarta jest w opisanych wyżej *metodach wykonujących operacje na węźle* implementacji `RemoteHostManager`. Obsługa żądania może wiązać się z dłuższym procesem — wówczas zarządca, po wyjściu z kodu obsługi żądania, jest wprowadzany w *stan przejściowy*, co oznacza, że żądanie zostało przetworzone, ale jego realizacja jeszcze trwa. Np. w przypadku operacji `wakeUp` w kodzie obsługi żądania zawarte jest wysłanie pakietu WOL przez MBeana i zlecenie wygenerowania po pewnym czasie powiadomienia o limicie czasu, po czym zarządca wprowadzany jest w stan przejściowy, oczekując na wybudzenie węzła lub przekroczenie limitu czasu.

- *Nadejście powiadomień* — ich obsługa zawarta jest w opisanych wyżej *metodach przetwarzających powiadomienia* implementacji `RemoteHostManager`. Obsługa powiadomienia może potencjalnie wyprowadzić zarządcę ze stanu przejściowego. Dzieje się tak np. gdy nadejdzie powiadomienie informujące o wybudzeniu węzła. Powiadomienie może również potencjalnie wprowadzić węzeł w stan przejściowy. Np. gdy wymagane jest wykonanie operacji zatrzymania zadań po otrzymaniu powiadomienia o zalogowaniu się użytkownika.

Definicja 1. *Zdarzenie nazywa się zdarzeniem dostępnym, gdy nadeszło (żądanie zostało zlecone lub powiadomienie zostało dostarczone), a nie zostało jeszcze przetworzone.*

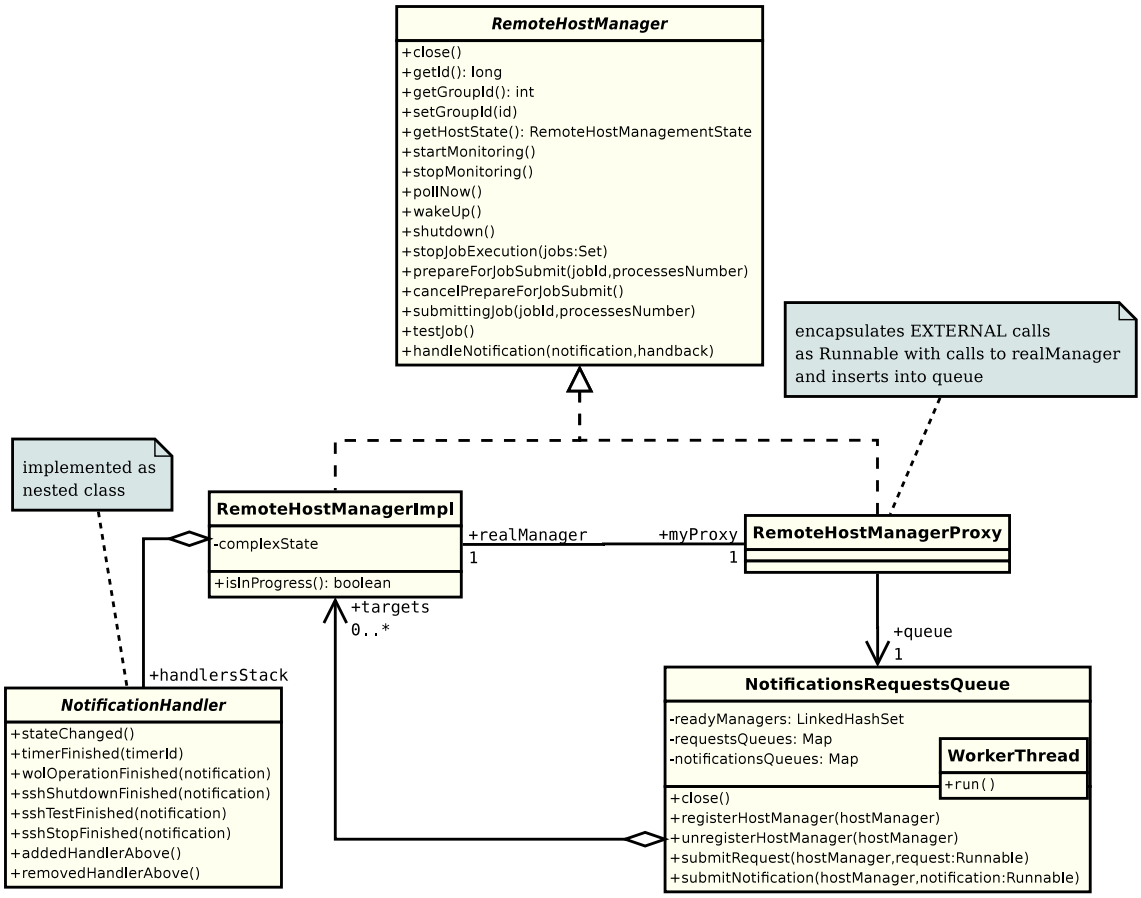
Zdecydowano się na założenie istotnie upraszczające implementację — możliwość wykonywania jednej operacji (procesu) na raz. Oznacza to, że gdy przetwarzone będzie pewne żądanie i spowoduje ono wprowadzenie zarządcy w stan przejściowy, to dopóki zarządca nie opuści stanu przejściowego poprzez przetworzenie któregoś z dostępnych powiadomień, nie zostanie przetworzone kolejne dostępne żądanie. Żądanie które nie mogło być obsłużone trafi do kolejki zdarzeń dostępnych. W każdej chwili jest natomiast możliwe przetwarzanie dostępnych powiadomień. To założenie nie ogranicza znacząco funkcji zarządcy, ponieważ wykonywanie równolegle dwóch operacji nie wydaje się być istotną własnością, a w przypadku niektórych par operacji nie ma sensu — np. `wakeUp` i `shutdown`.

Dodatkowo, założono że w stanie przejściowym obsługa powiadomień nie będzie pełna, tzn. będzie skoncentrowana na kontekście aktualnego stanu przejściowego, a ewentualne reakcje związane z ogólnym kontekstem będą obsłużone po wyjściu ze stanu przejściowego. Może to powodować pewne opóźnienia w reakcji na powiadomienia, jednak przy typowym funkcjonowaniu systemu powinny być one prawie niezauważalne.

Przy tych założeniach, metody obsługujące żądania mogą być zaimplementowane *statycznie*, a ich zawartość może być dość prostymi regułami — ponieważ nie są wywoływane w trakcie stanu przejściowego. *Dynamiczne* są natomiast metody obsługujące powiadomienia, które zmieniają się w zależności od stanu przejściowego.

Ostateczna implementacja wykorzystuje przedstawione pomysły, schematycznie zaprezentowana jest na rysunku 4.5. Stworzono klasę implementującą *kolejkę powiadomień i żądań* `NotificationsRequestsQueue`, *współdzieloną* między zarządcami. Kolejka ta przyjmuje powiadomienia i żądania dla wszystkich zarządców, przetwarza je w globalnym (w kontekście wszystkich zarządców) porządku opartym o FIFO. Przechowywana jest w niej lista zarządców, dla których dostępne są zdarzenia do przetworzenia oraz indywidualne kolejki powiadomień i żądań dla każdego z zarządców. Gdy kolejka przyjmuje powiadomienie lub żądanie, umieszcza zarządcę na końcu globalnej listy, a nadchodzące zdarzenie na końcu jego odpowiedniej indywidualnej kolejki.

Do przetwarzania nadchodzących zdarzeń `NotificationsRequestsQueue` wykorzystuje tylko jeden wątek, synchronizowany (w efektywny sposób) z metodami przyjmującymi nadchodzące zdarzenia. Zdarzenia są więc obsługiwane sekwencyjnie — ułatwia to implementację zarządcy, nie ma potrzeby synchronizacji wewnątrz kodu obsługi zdarzeń, konieczne jest natomiast utrzymywanie jego stanu. Bieżący zarządca do obsługi w wątku kolejki wybierany jest na podstawie listy gotowych zarządców. Dla takiego zarządcy uruchamiana jest obsługa pierwszego powiadomienia z jego kolejki — o ile takie istnieje. Jeśli nie ma dostępnych powiadomień i zarządca nie jest w stanie przejściowym, przetwarzane jest żądanie — o ile jest



Rys. 4.5: Uproszczony diagram UML klas służących do zarządzania węzłem.

dostępne. Przetworzenie nadchodzącego zdarzenia oznacza wywołanie odpowiedniej metody z implementacji `RemoteHostManager`. Po przetworzeniu zdarzenia zarządca trafia na koniec kolejki — o ile są jeszcze dla niego zdarzenia do przetworzenia.

Klasa kolejki nie implementuje jednak interfejsu `RemoteHostManager`. Ten interfejs implementują dwie klasy. `RemoteHostManagerImpl` jest właściwą implementacją zarządcy: zawiera kod obsługi zdarzeń, referencje do MBeana, przechowuje stan zarządzania. `RemoteHostManagerProxy` jest natomiast pośrednikiem, który opakowuje wywołania metod interfejsu `RemoteHostManager` w obiekty `Runnable` i wstawia je do kolejki `NotificationsRequestsQueue`. Pośrednik przyjmuje zdarzenia od Web Services i globalnego zarządcy, przez co mają oni dostęp do przyjaznego interfejsu zarządcy, a kwestia właściwego przetwarzania zdarzeń rozwiązana jest w implementacji kolejki.

Kod obsługi żądań to w `RemoteHostManagerImpl` zwykłe metody — jest więc to statyczny kod. Inaczej jest zaimplementowana obsługa powiadomień. Powiadomienia delegowane są na podstawie ich rodzaju do właściwej metody obsługującej — np. `sshOperationFisnished`. Aby kod obsługi powiadomień nie był statyczny, a zmieniał się w raz z przejściem zarządcy w określony stan przejściowy, stwo-

rzono abstrakcyjną klasę wewnętrzną `NotificationHandler`. Zawiera ona wspomniane metody, które obsługują powiadomienia poszczególnych rodzajów. Obsługa powiadomień delegowana jest więc przez zarządce do klas rozszerzających `NotificationHandler`. W zwykłym stanie zarządcy jest to implementacja w postaci `DefaultNotificationHandler`, natomiast poszczególnym procesom do wykonania na węźle odpowiadają kolejne implementacje, np. dla obsługi procesu wybudzania `WakingUpNotificationHandler`. Zmiana kodu odpowiedzialnego za obsługę odbywa się przez stos implementacji `NotificationHandler`. Ponieważ pewne procesy powodujące stan przejściowy zarządcy są złożone, pozwala to na ich dekompozycję oraz znalezienie elementów wspólnych. Za obsługę powiadomienia odpowiedzialna jest zawsze implementacja na szczycie stosu. Implementacja sama decyduje kiedy usunąć siebie ze stosu (za pomocą odpowiednich metod pomocniczych zarządcy), jak również może zażądać wykonania pewnych operacji podczas zdejmowania ze stosu, przesłaniania na stosie itd. Implementacje `NotificationHandler` są dodawane na stos przez kod obsługujący żądania i powiadomienia.

`RemoteHostManagerImpl` zawiera obiekty reprezentujące jego bieżący stan, takie jak lista zarejestrowanych zadań, ostatnia migawka stanu węzła z MBeana, informacja o tym czy system włączył węzeł itp. Są one udostępniane przez opisane wcześniej *metody udostępniające informacje*, a dalej przez Web Services i WWW. Metody te nie podlegają kolejkowaniu — dostęp do informacji jest natychmiastowy.

Rolą implementacji zarządcy jest również utworzenie MBeana `RemoteHost` i zaaplikowanie hierarchicznej konfiguracji opisanej w punkcie 3.1.6. Zarządca rejestruje lub wyrejestrowuje się u globalnego zarządcy w momencie, odpowiednio, aktywowania lub dezaktywowania zarządzania. Kiedy węzeł jest zarządzany, wyznacza swoją dostępność zgodnie z opisem określonym w zbiorze reguł i informuje o niej globalnego zarządcę. Podobnie udostępniane są informację nt. stanu uruchamiania, bądź przygotowań do uruchamiania zadania.

4.3.3.3 Globalne zarządzanie zasobami

Zdefiniowane reguły zarządzania. Na poziomie globalnego zarządzania zasobami również zdefiniowano reguły zarządzania — rozpatrywane w kontekście przydziału zasobów. Uzupełniają one zbiór zasad z punktu 4.3.3.2. Podobanie jak tamte reguły bazują na zadaniach stawianych wobec systemu i stanowią opis działania zarządcy wyższego poziomu:

1. Węzły obliczeniowe są połączone w rozłączne grupy, które mogą reprezentować wykorzystywane sale laboratoryjne lub węzły o podobnych ustawieniach.

2. Węzeł obliczeniowy jest *dostępny do obliczeń*, gdy:

   - Jego zarządca zgłosił węzeł jako dostępny w jego lokalnym kontekście.
   - Żaden z węzłów w jego grupie nie ma jednocześnie aktywnych: ustawienia `stopGroupOnAnyLogin` i niedostępności z powodu zalogowanego

użytkownika.

3. Węzeł jest *zajęty obliczeniami* kiedy uruchomiony jest na nim proces jakiegoś zadania.

4. Zadania w systemie są kolejkowane zgodnie z przyjętym *algorytmem kolejkowania*. Jego rolą jest wyznaczenie kolejnego zadania do uruchomienia.

5. Pierwsze zadanie z kolejki jest uruchamiane i usuwane z niej, jeśli liczba procesorów (rdzeni) dostępnych i nie zajętych węzłów w systemie jest większa lub równa wymaganej liczbie procesorów `minMachines` dla tego zadania. Zadanie jest uruchamiane na co najmniej `minMachines` procesorach, a maksymalnie na `requestedMachines`. Węzły wybiera *algorytm przydziału węzłów dla zadań*, który powinien uwzględniać możliwe preferencje użytkownika i wiedzę systemu.

6. Jeśli proces zadania na pewnym węźle zakończył się lub nie uruchomił poprawnie, a zadanie miało aktywne ustawienie `stopAllOnAnyFault` to zatrzymywane są wszystkie procesy tego zadania — również na innych węzłach.

7. Jeśli *zajęty przez obliczenia* węzeł staje się *niedostępny do obliczeń*, to zadania na nim uruchomione zostają zatrzymane. Ta sama zasada dotyczy węzłów, które stają się niedostępne ze względu na ustawienie `stopGroupOnAnyLogin` (reguła 2).

8. Jeśli po czasie `maxWalltime` zadanie nie zakończyło się samoczynnie, jest zatrzymywane przez system.

Implementacja. Ze względu na dużą funkcjonalność zapewnioną w implementacji `RemoteHostManager`, zarządca globalny nie musiał być tak złożony. Zdefiniowano go w klasie `ResourcesManager`. Również został zrealizowany jako obiekt sterowany zdarzeniami — nie jest związany z żadnym wątkiem, poza prostą usługą systemową `Timer` do kontroli czasu wykonywania zadań. Jednakże w tym przypadku nie było konieczne stosowanie pośrednika, ani kolejkowania zdarzeń. Zdarzeniami są wszystkie wywołania metod na obiekcie:

- *Zgłoszenie lub anulowanie zadania* — zgłoszenia z Web Services, czyli pośrednio z Submittera lub WWW. Zgłoszenie zadania powoduje jego zakolejkowanie lub przygotowanie do wykonania jeśli w systemie jest dostępna odpowiednia liczba węzłów. Zadania o wymaganej liczbie procesorów większej niż istniejąca w systemie nie są przyjmowane. Anulowanie zadania zatrzymuje procesy zadania na węzłach. Zadanie może być też anulowane przez usługę `Timer` po upłynięciu maksymalnego czasu działania. Obie operacje aktualizują dane dot. zadania w bazie danych.

- *Zarejestrowanie lub wyrejestrowanie zarządcy węzła* — informacja z zarządcy węzła o aktywacji lub dezaktywacji jego usług. Powoduje dodanie lub usunięcie informacji z wewnętrznych struktur.

- *Informacja o zmianie dostępności węzła* — przesyłana z zarządcy. Informacja ta może wpłynąć zarówno na uruchomione zadania — ich zatrzymanie, gdy węzeł zostaje zajęty przez użytkownika, jak i na zadania oczekujące w kolejce — zwiększenie liczby dostępnych węzłów może spowodować przydział węzłów, przygotowanie zadania do uruchomienia.

- *Informacja o zmianie stanu przygotowania węzła* — przesyłana z zarządcy. Przygotowanie wszystkich węzłów powoduje zmianę stanu zadania na uruchomione, a w konsekwencji przekazanie tej informacji Submitterowi. Błąd w przygotowaniu któregoś z węzłów (np. zalogowanie użytkownika) anuluje proces przygotowań i wymaga jego powtórzenia. W praktyce nie wiąże się to jednak z poważnymi opóźnieniami, ponieważ węzły nie są wyłączane natychmiast po zwolnieniu.

- *Informacja o zakończeniu zadania* — przesyłana z zarządcy, uaktualnia dane dot. zadania i zajętości węzłów. Przy odpowiednich ustawieniach zadania może spowodować jego zatrzymanie.

Obsługa zdarzeń jest w prosty sposób synchronizowana: w jednej chwili przetwarzane jest jedno zdarzenie. W przypadku zgłaszania zadania wywołujący (Submitter) jest blokowany na monitorze aż do czasu zezwolenia na uruchomienie zadania lub jego anulowania. Istotne jest, że `ResourcesManager` komunikuje się z zarządcami węzłów przez ich pośredników, czego efektem ubocznym jest uniknięcie możliwości wystąpienia zakleszczeń, gdy w tym samym czasie odbywa się komunikacja w drugą stronę.

Klasa `ResourcesManager` zawiera szereg struktur przechowujących informacje nt. stanu węzłów i zadań. Część z nich jest zapisywana w bazie danych i dostępna dla Submittera lub WWW. Z tego powodu zakłada się, że zarządca posiada wyłączny dostęp do relacji `Job` w bazie danych — jego transakcje nie mogą zostać wycofane.

Dla uproszczenia implementacji przyjęto, że zarządca może w jednej chwili przygotowywać węzły dla jednego zadania. Przy dużym obciążeniu — dużej liczbie zadań, przygotowanie węzłów trwa krótko (są już włączone po poprzednim zadaniu), więc nie powinno to istotnie zwalniać obsługi zadań, a zapobiega konfliktom w rezerwacji zasobów. Nie eliminuje to także możliwości wykonywania wielu zadań jednocześnie, ponieważ dotyczy tylko etapu przygotowania węzłów dla zadania. Sam zarządca może być w jednym z trzech stanów związanych z przygotowywaniem węzłów. Możliwe stany są zdeterminowane zmienną `servingState`, która wymusza właściwą sekwencję zadań (obsługę jednego zadania naraz). Możliwe wartości tej zmiennej to:

- `NOT_SERVING_ANY_JOB` — gdy zarządca nie przygotowuje węzłów dla żadnego zadania z kolejki. Stan taki ma miejsce, gdy nie ma odpowiedniej liczby wolnych węzłów w systemie lub nie ma zadań w kolejce.

- `PREPARING_HOSTS_FOR_JOB` — gdy przygotowywane są węzły dla pierwszego zadania z kolejki.

- `CANCELLING_PREPARATION_FOR_JOB` — gdy anulowane są przygotowania. Powodem może być błąd w którymś węźle lub anulowanie zadania.

W przypadku błędu w przygotowaniach na którymś węźle (np. zalogowanie użytkownika lub wyłączenie komputera), proces jest anulowany i zadanie trafia z powrotem do kolejki — aby faktycznie zapewnić zadaniu działające węzły w odpowiedniej liczbie. Po powrocie zadania do kolejki, możliwe jest szybkie, w praktyce niezauważalne, powtórzenie procesu.

Algorytmy *kolejkowania zadań* i *przydziału węzłów dla zadań* dostarczane są do klasy `ResourcesManager` w postaci implementacji prostych interfejsów `JobsQueue` oraz `HostsAllocator`, co umożliwia ich łatwą zmianę.

W aktualnej implementacji `ResourcesManager` istnieje ograniczenie umożliwiające uruchamianie na jednym węźle (wielu) procesów tylko jednego zadania. Zakładając, że w systemie nie jest uruchamianych wiele bardzo małych zadań, nie powinno to znacząco ograniczać funkcjonalności.

4.3.3.4 Algorytm kolejkowania zadań

Jedyną metodą interfejsu `JobsQueue` jaką musi implementować algorytm kolejkowania zadań, jest bezargumentowa metoda `getNextJob()`. Zakłada się, że zadanie zostanie wybrane spośród zadań w bazie danych, których stan to `JobState.QUEUED`. W bieżącej wersji systemu zakłada się istnienie jednej kolejki, nie istnieje możliwość definiowania zależności między zadaniami. Wymagałoby to zmian w relacji `Job` i usłudze zgłaszania zadań.

W aktualnej wersji systemu dostarczono `JobsQueueImplFIFO` — prostą *kolejkę FIFO* realizowaną w oparciu o atrybut czasu zgłoszenia zadania do systemu, tj. `submissionTime`. Preferowane są zadania najdłużej oczekujące w systemie. Może to doprowadzić do niewykorzystywania wolnych zasobów przez długi czas, gdy za zadaniem wymagającym dużej liczby węzłów oczekują zadania o mniejszych wymaganiach. Z drugiej strony, eliminuje to jednak potencjalny problem głodzenia (ang. *starvation*) zadania o dużych wymaganiach.

Algorytm kolejkowania jest elastyczny, więc w zależności od obserwowanych potrzeb można do niego dodać np. system priorytetów dla zadań (wymagałby prawdopodobnie określenia maksymalnych dozwolonych priorytetów dla kont) albo połączyć istniejący algorytm wybierania zadania najstarszego ze strategią wybierającą pierwsze zadanie które da się zrealizować. Tę ostatnią strategię prawdopodobnie na-

leżałoby zastosować z mechanizmem zapobiegającym głodzeniu dużych zadań opierając się np. na parametrze `maxWalltime` zadania.

4.3.3.5 Algorytm przydziału węzłów dla zadań

Implementacja algorytmu przydziału węzłów również sprowadza się do zaimplementowania interfejsu `HostsAllocator` o jednej metodzie `allocateHostsForJob()`. Udostępniane w argumentach metody informacje, na podstawie których podjęta musi zostać decyzja to *zadanie*, *lista wolnych węzłów* oraz *lista wszystkich węzłów*. W rzeczywistości węzły są reprezentowane przez ich zarządców. Algorytm jest wywoływany tylko, gdy istnieje wystarczająca liczba węzłów by obsłużyć zadanie.

Implementacja algorytmu została zdefiniowana w klasie `HostsAllocatorImpl`. Wykorzystuje ona komparator implementujący interfejs `Comparator` dla węzłów. Komparator definiuje relację porządku liniowego na węzłach. Na podstawie tego porządku wybieranych jest pierwszych `requestedMachines` węzłów dla zadania.

Komparator wykorzystuje w swej definicji relację $\equiv$ dzielącą zbiór węzłów na *klasy równoważności* na podstawie preferencji węzła i grupy (parametrów konfiguracji). Ma to służyć agregowaniu zadań w małej liczbie grup (przy założeniu jednakowej konfiguracji preferencji wewnątrz grupy), a więc potencjalnie zmniejsza to szansę przerwania zadania przez zalogowanie użytkownika, zakładając, że grupy reprezentują pomieszczenia. Zwiększa to także liczbę szybkich, homogenicznych maszyn dla zadania.

Definicja 2. *Niech $\equiv$ będzie relacją równoważności określoną na zbiorze węzłów M taką że:*

$$\forall a, b \in M : a \equiv b \Leftrightarrow preference(a) = preference(b) \wedge group(a) = group(b)$$

Dzięki tej definicji możemy określić liczność klasy równoważności każdego węzła $|[m]_{\equiv}|$.

Pełny komparator bierze pod uwagę kilka kryteriów wybranych intuicyjnie: starając się spełniać preferencje użytkownika, administratora, zapewnić stabilność działania i możliwie szybkie homogeniczne środowisko. Działanie komparatora dla węzłów a i b przedstawione zostało w algorytmie 1.

Podobnie jak w przypadku algorytmu kolejkującego, mechanizm wyboru węzłów dla zadania jest elastyczny — można go łatwo zmodyfikować, biorąc pod uwagę obserwacje z działania systemu. Można również zaproponować np. algorytm bazujący na historii przetwarzania, stabilności węzłów.

4.3.3.6 Testy modułu

Testy tego modułu były dość ograniczone. Ze względu na rozmiar kilku głównych klas i silną zależność od operacji Monitora utrudnione było testowanie jednostkowe.

Algorytm 1 Komparator porządkujący węzły do przydziału

Require: hosts $a, b \in M$
  {zwraca wartość > 0 gdy preferowany jest węzeł a, 0 gdy żaden, < 0 gdy b}
  if $group(a) \in UserPreferred \neq group(b) \in UserPreferred$ **then**
    {preferuj węzeł z grupy preferowanej przez użykownika}
    if $group(a) \in UserPreferred$ **then**
      return 1
    else
      return -1
    end if
  else if $preference(a) \neq preference(b)$ **then**
    {preferuj węzeł o wyższej preferencji wynikającej z konfiguracji}
    return $preference(a) - preference(b)$
  else if $|[a]_{\equiv}| \neq |[b]_{\equiv}|$ **then**
    {preferuj węzeł należący do liczniejszej klasy równoważności}
    return $|[a]_{\equiv}| - |[b]_{\equiv}|$
  else if $id(group(a)) \neq id(group(b))$ **then**
    {preferuj węzeł z grupy o mniejszym id}
    return $id(group(b)) - id(group(a))$
  else
    {preferuj węzeł o większej liczbie CPU (kryterium wewnątrz klasy rów.)}
    return $cpus(a) - cpus(b)$
  end if

Wykonano ograniczone testy integracyjne z udziałem modułów DB, Monitora i dwóch węzłów z uruchomionymi Agentami. Obserwowano (z użyciem mechanizmu logów opisanym w punkcie 4.3.5) reakcje zarządców węzłów na różne operacje wykonywane na węźle. Wymuszano ich wyłączenie, włączenie, logowano się na nie — symulując zachowania użytkownika. Zlecano zarządcy wykonywanie szeregu operacji (procesów) na węźle w celu przetestowania różnych implementacji `NotificationHandler`.

Gdy zaimplementowano zarządcę wyższego poziomu, przeprowadzono próby przydziału zasobów, przygotowywania węzłów zlecając nieistniejące (bez Submittera) zadania.

4.3.4 Moduł realizacji usług sieciowych — Web Services

Dostęp do wszystkich funkcji udostępnianych przez moduł zarządzający realizowany jest poprzez technologię usług sieciowych. Z punktu widzenia Managera, jest to środek komunikacji zewnętrznej. Mając jednak na uwadze system w sensie całościowym, komunikacja ta odbywa się pomiędzy jego poszczególnymi elementami składowymi, nabierając charakteru wewnętrznego. Trudno jednakże przewidzieć oraz ograniczyć możliwości rozbudowy systemu o dodatkowe komponenty, czego z założenia nie dokonano. Dlatego też implementacja zrealizowana w ramach modułu usług sieciowych powinna być zgodna ze standardami oraz gwarantować choćby minimalne mechanizmy bezpieczeństwa i kontroli dostępu.

4.3.4.1 Implementacja interfejsu usług sieciowych

Użyta technologia JAX–WS pozwoliła przyjąć jako punkt wyjścia do realizacji usług sieciowych gotową implementację przygotowaną w języku Java (por. 4.2 Komunikacja zewnętrzna: Web Services).

W pierwszym kroku, należało zatem przygotować pewną abstrakcję zestawu funkcji każdej usługi, która w praktyce sprowadziła się do konkretnych interfejsów w języku Java. Następnie, wykorzystując mechanizm adnotacji (ang. *Java Annotations*), wprowadzono informacje specyficzne dla usług sieciowych, które znajdują swoje odzwierciedlenie w dokumencie WSDL. Mianowicie, były to:

- Docelowa przestrzeń nazw, opisana przez element `targetNamespace` adnotacji `@WebService`. Innymi słowy przestrzeń nazw dla metadanych zawartych w wiadomości SOAP, odpowiadających nazwie operacji oraz jej argumentów. Identyczna przestrzeń obowiązuje tym samym w kontrakcie WSDL.

- Wyspecyfikowanie sposobu formowania wiadomości SOAP, czyli ustalenie szczegółów powiązania danych z plikiem XML, odbywa się poprzez zastosowanie adnotacji `@SOAPBinding`. Jej elementy określają styl wiadomości, styl parametrów oraz kodowanie. W przypadku omawianego systemu przyjęto kolejno wartości: *Document, Wrapped, Literal*.

- Jawne nazwy argumentów ustalano również poprzez adnotację `@WebParam`. Jej brak spowodowałby wygenerowanie identyfikatorów domyślnych, gdyż interfejsy w języku Java nie zawierają w sobie nazw argumentów poszczególnych metod.

Kolejny krok polegał na dostarczeniu kodu realizującego poszczególne funkcje udostępniane przez usługi, poprzez utworzenie klas implementujących opisane wyżej interfejsy Javy. Również i w tym miejscu pojawiły się dodatkowe informacje przydatne z punktu widzenia tworzenia usług sieciowych. Przy pomocy adnotacji `@WebService` ustalono interfejs, który implementuje klasa, nazwę usługi oraz nazwę portu [50]. Ta ostatnia jest wspólna dla wszystkich klas usług i koresponduje z nazwą modułu je udostępniającego — Managera: *putgrid_mgr*.

Ostatecznie, należało wygenerować kontrakt WSDL oraz odpowiednie artefakty dla biblioteki JAX–B. W tym celu użyto narzędzia *wsgen* dostarczanego wraz z pakietem JAX–WS RI.

W wyniku opisanego procesu powstało pięć interfejsów oraz pięć klas będących ich implementacją zawartych w pakietach `services` oraz `services.impl`. Odpowiadają one pięciu osobnym usługom, które zostały wyróżnione ze względu na logiczną funkcjonalność udostępnianą przez moduł. Tabela 4.5 przedstawia nazwy poszczególnych usług sieciowych, ich przestrzeń nazw oraz krótki opis funkcji, których dotyczą.

Nazwa usługi	Przestrzeń nazw	Opis
Account Service	`urn:putgrid:mgr:as:1.0`	zarządzanie kontami użytko–wników oraz sesją
Group Service	`urn:putgrid:mgr:gs:1.0`	informacje o grupach węzłów dostępnych w systemie
Host Configuration	`urn:putgrid:mgr:hcs:1.0`	udostępnianie konfiguracji węzłów
Host Service	`urn:putgrid:mgr:hs:1.0`	aktualne informacje dotyczące węzłów
Job Service	`urn:putgrid:mgr:js:1.0`	zgłaszanie oraz zarządzanie zadaniami

Tab. 4.5: Usługi sieciowe udostępniane przez moduł Managera.

Publikowanie usług sieciowych. Ostatecznym krokiem implementacyjnym jest udostępnienie usług sieciowych poprzez wbudowany serwer HTTP (por. podpunkt 4.2.2.1 Strona serwera). Do tego celu służy klasa `Endpoint`, którą dostarcza pakiet JAX–WS RI, a która poprzez wzorzec projektowy fabryki (ang. *Factory Pattern*) daje możliwość skojarzenia usługi z określonym zasobem (ang. *Resource*) serwera HTTP. Serwer może być dostępny pod ustalonym portem, co pozwala na dostosowanie aplikacji do ograniczeń lub wymogów środowiskowych. Ponadto dostarczono

klasę `EndpointFactory`, której zadaniem jest tworzenie obiektów wspomnianej klasy `Endpoint`, zgodnie z zadanym schematem postępowania. Takie obiekty są dzięki temu spójne w kontekście parametrów, a także dodatkowo dołączonych mechanizmów modułu weryfikacji tożsamości.

4.3.4.2 Moduł weryfikacji tożsamości

Jak wspomniano uprzednio, moduł realizujący usługi sieciowe powinien dostarczać choćby najprostsze mechanizmy weryfikacji żądań. Taka weryfikacja powinna zapewniać kontrolę tożsamości nadawcy, czyli uwierzytelnianie, oraz także kontrolę upoważnień dostępu przyznanych nadawcy do poszczególnych funkcji, autoryzację. Tę potrzebę adresuje moduł weryfikacji tożsamości, który został zaprojektowany pod kątem współdziałania z usługami sieciowymi. Jego szczegółowy opis został zawarty w niniejszym podpunkcie.

Tworzenie oraz utrzymywanie sesji. Identyfikacja użytkownika chcącego nawiązać sesję z omawianym systemem odbywa się w momencie wywołania funkcji zalogowania. Jako parametry przekazywane są nazwa użytkownika (ang. *Login*) oraz wartość skrótu otrzymana z hasła użytkownika. Algorytmem zaproponowanym do jej wyliczenia jest MD5. W wyniku podania poprawnych danych nadawca otrzymuje tzw. *identyfikator sesji*, którego dołączanie do każdego żądania pozwoli na autoryzację oraz uwierzytelnienie bez ponownego użycia loginu i skrótu hasła. W celu poprawy kwestii bezpieczeństwa, rozważano również mechanizm „zawołanie — odzew" (ang. *challenge — response*), który eliminuje potrzebę przesyłania hasła. Mechanizm ten w ogólności może zostać dołączony w przyszłości rozwoju systemu, bez wpływu na przedstawione w niniejszym podpunkcie rozwiązania.

Identyfikator sesji jest losową sekwencją bajtów o ustalonej długości, zapisaną przy pomocy kodowania Base64 [40]. Użyto w tym miejscu generatora liczb losowych `java.security.SecureRandom` dostarczonego przez Sun Microsystems. Wykorzystano także zewnętrzną bibliotekę zawierającą klasę kodera formatu Base64, `org.apache.commons.codec.binary.Base64`, dostępną w pakiecie Apache Commons Codec [15]. Tak przygotowany identyfikator jest zapamiętywany w systemie w celu późniejszej weryfikacji.

Identyfikatory sesji, zwane czasem tokenami, składowane są w odpowiednich strukturach danych, pozwalających na optymalną z punktu widzenia złożoności obliczeniowej weryfikację ich istnienia oraz ważności. Sprawdzane jest również powiązanie token — identyfikator użytkownika, który w sposób unikalny przedstawia nadawcę, dla którego token został ustalony. Ważność sesji utrzymywana jest przez pewien ustalony czas braku aktywności ze strony nadawcy, jednakże wygasa po jego upływie. Jest to jednoznaczne z potrzebą ponownego zalogowania do systemu. Warto podkreślić, że ważność tokenu jest przedłużana przy każdorazowym wykazaniu aktywności przez nadawcę, to jest przy każdej pozytywnej próbie jego

weryfikacji. Identyfikatory sesji, których ważność upłynęła, są usuwane ze struktur danych.

Omówiony mechanizm zaimplementowano w klasie `IdentityProvider` oraz pozostałych klasach z pakietu *idp*.

Identyfikowanie sesji. Po prawidłowym zalogowaniu do systemu, tj. nawiązaniu sesji, każde kolejne zgłoszenie do usługi sieciowej powinno zawierać w sobie identyfikator sesji, który posłuży serwerowi do uwierzytelnienia oraz autoryzacji. Warto podkreślić charakter identyfikatora sesji, który jest z założenia semantycznie różny od informacji zawartej w głównej wiadomości SOAP. Co więcej, adresatem tych dodatkowych informacji nie jest implementacja usługi sama w sobie, lecz bardziej ogólny mechanizm uwierzytelniania. Dlatego też zaproponowano użycie nagłówków SOAP (ang. *SOAP Headers*), które z założenia są dedykowane do takich potrzeb.

Z technicznego punktu widzenia, dodanie nagłówka do wiadomości SOAP, polega na dołączeniu do dokumentu XML węzła *<header></header>*. Węzeł ten reprezentuje strukturę odpowiadającą postaci informacji, która jest przesyłana. W omawianym przypadku jest to identyfikator sesji, chociaż nie ma ograniczeń na rozbudowę wspomnianej struktury w razie potrzeby.

W kontekście zastosowanego pakietu JAX–WS RI, obsługa nagłówków SOAP nie jest w pełni wspierana. Element *Header* adnotacji `@WebParam` nie wpływa na generowany kontrakt WSDL, dlatego też należało przygotować narzędzie skryptowe do jego odpowiedniej modyfikacji. Wprowadzone modyfikacje są zgodne ze specyfikacją języka WSDL [50], co potwierdzają przeprowadzone testy integrujące zaproponowaną implementację serwera z klientami.

Należy jednakże podkreślić, iż zaproponowany przez Sun Microsystems pakiet nie nałożył żadnych sztucznych ograniczeń na zastosowane technologie, a jedynie zwiększył nakład pracy. Stało się tak gdyż ostatecznie zastosowano mechanizm łańcuchów obsługi (ang. *Handler Chains*) opisany w dalszej części niniejszego podpunktu. Musiano w nim zaimplementować kwestie pozyskiwania informacji z nagłówków wiadomości SOAP, przy użyciu technologii JAX–B.

Powiązanie z interfejsami usług. Przedstawiono tutaj mechanizmy uwierzytelniania oraz autoryzacji stosowane w trakcie komunikacji z usługą sieciową.

Jednym z głównych założeń pozafunkcjonalnych, z jakimi musiał się zmierzyć moduł weryfikacji tożsamości, była elastyczność oraz prostota modyfikacji parametrów kontroli dostępu do poszczególnych funkcji. Odrzucono wobec tego wszelkie pomysły na rzecz adnotacji, które zastosowane w interfejsach omówionych w poprzednim podpunkcie, dają jednoznaczny obraz wymaganej kontroli.

W skład zaproponowanych adnotacji wchodzą:

- `@AuthLevel` służąca do wyspecyfikowania ról użytkowników, które mają zezwolenie na wykonanie danej metody — funkcji udostępnianej przez usługę

sieciową. Jest to element kształtujący politykę kontroli dostępu w aspekcie autoryzacji.

- `@UserIsolation` informuje, że oznaczona metoda posiada parametry, których wartość powinna być zgodna z tożsamością nadawcy. Innymi słowy, dany nadawca nie może wykonać metody z parametrami dotyczącymi innych użytkowników systemu. Omówiony mechanizm gwarantuje izolację działań użytkowników, kontrolując dostęp w aspekcie autoryzacji.

- `@PublicMethod` informuje moduł weryfikacji tożsamości, że w przypadku metody opatrzonej tą adnotacją nie należy uwierzytelniać nadawcy, gdyż prawdopodobnie nie został on jeszcze zidentyfikowany. Naturalną funkcją, której niniejsza adnotacja dotyczy, jest logowanie do systemu.

- `@LogoutHandler` — ostatnia adnotacja, której semantyka ma jedynie znaczenie techniczne. Wskazuje, że wywołanie danej funkcji usługi sieciowej powinno powodować wylogowanie nadawcy oraz unieważnienie sesji.

Ponadto, wymienione adnotacje mogą posiadać parametry, zgodnie z informacjami zamieszczonymi w dokumentacji kodu.

Dzięki przedstawionym adnotacjom możliwe stało się sprawowanie kontroli nad dostępem do funkcji w sposób automatyczny, bez poważniejszego zaangażowania programisty.

Łańcuchy obsługi. Działanie opisanych wyżej mechanizmów w kontekście przetwarzania wiadomości SOAP wymagało zaproponowania na tyle ogólnego podejścia, by implementacja usług była od nich niezależna. Zdecydowano się, zatem, na wykorzystanie łańcuchów obsługi (ang. *Handler Chains*). Ich filozofia polega na utworzeniu zespołu filtrów, które umieszczone kolejno na drodze klient — serwer, pozwalają na przetwarzanie wiadomości SOAP. Dzięki temu można kaskadowo nałożyć dodatkowe mechanizmy, w celu zapewnienia np. uwierzytelnienia oraz autoryzacji. Ponadto, łańcuch obsługi może pojawić się zarówno po stronie serwera, jak i klienta. Przetwarzanie następuje na dwóch warstwach, warstwie protokołu SOAP, bądź też logicznej zawartości wiadomości. Prezentowane rozwiązanie dotyczy tej pierwszej.

Przygotowano klasę `AuthenticatorHandler` dostarczającą funkcjonalności weryfikacji tożsamości oraz uprawnień. Obiekt tej klasy umieszczony w łańcuchu przetwarza oraz odrzuca wszelkie żądania klienta, które nie spełniają przyjętych reguł bezpieczeństwa. Reguły te, są pozyskiwane z klas interfejsów usług sieciowych, z adnotacji w nich umieszczonych. Wykorzystano w tym celu mechanizm odzwierciedleń języka Java (ang. *Java Reflection*), dzięki któremu można odczytać metainformacje zawarte w klasach w trakcie wykonania programu (ang. *Run Time*).

Obsługa wiadomości na tym kroku polega na odczytaniu ewentualnego identyfikatora sesji z nagłówka wiadomości SOAP, nazwy metody, której wykonania

zażądano oraz potencjalnie wartości jej argumentów. Pierwsza informacja służy do uwierzytelnienia nadawcy wiadomości na podstawie danych zgromadzonych przez moduł weryfikacji tożsamości. Kolejne, natomiast, pozwalają na sprawdzenie czy próba wykonania żądanej metody jest autoryzowana w kontekście przyznanej użytkownikowi roli w systemie, jak i ograniczeń nałożonych na wartości jej aktualnych argumentów. Ten ostatni etap posiłkuje się obiektem klasy implementującej interfejs `ArgumentComparator`, dostarczanym wraz z implementacją usług sieciowych. Pozwala on zweryfikować wartości argumentów w kontekście izolowania działań użytkowników.

Dodatkowo w łańcuchu obsługi znajduje się implementacja obsługi wylogowania, czyli unieważnienia sesji. Zrealizowano ją w klasie `LogoutMethodHandler`, której obiekty przetwarzając wiadomość, dokonują wyrejestrowania identyfikatora sesji ze struktur danych utrzymywanych w module weryfikacji tożsamości. Warto podkreślić, że opisane przetwarzanie następuje dopiero po dokonaniu uwierzytelnienia oraz autoryzacji.

W trakcie przetwarzania wiadomości przez łańcuch obsługi, może dojść do sytuacji wyjątkowej, która narusza jedno z przyjętych założeń. W takim przypadku generowany jest wyjątek, który następnie w postaci błędu wiadomości SOAP (ang. *SOAP Fault*) jest przesyłany do klienta. W celu utworzenia odpowiedniej struktury w dokumencie XML reprezentującym wiadomość, ponownie posłużono się technologią JAX–B.

Ostatecznie, poprawne utworzenie łańcucha obsługi spoczywa na dostarczonej klasie `AuthEndpointDecorator`, która realizuje wzorzec dekoratora (ang. *Decorator Pattern*), względem obiektów wspomnianej już klasy `Endpoint`.

4.3.5 Cykl działania aplikacji

Główna klasa aplikacji jest zdefiniowana pod nazwą `Manager` — służy do uruchomienia poszczególnych modułów, przechowuje referencje do obiektów usług całej aplikacji. Sekwencja uruchomienia Managera jest ściśle określona ze względu na zależności w modułów i wygląda następująco:

1. Utworzenie podstawowych struktur.
2. Zarejestrowanie kodu *obsługującego zamknięcie systemu* (ang. *shutdown hook*).
3. Utworzenie połączenia z bazą danych (moduł DB).
4. Uruchomienie globalnego zarządcy zasobów (moduł Disposer).
5. Uruchomienie modułu Monitora.
6. Wczytanie konfiguracji.
7. Uruchomienie zarządców zasobów dla tych węzłów, które zdefiniowano w konfiguracji (Disposer).

8. Uruchomienie serwerów usług (moduł Web Services).

Po uruchomieniu aplikacji dalsze je działanie jest sterowane wewnętrznymi i zewnętrznymi zdarzeniami. Moduły Managera intensywnie wykorzystują mechanizm logowania (generowania logów) używając do tego celu otwartą bibliotekę log4j [16]. Zdefiniowano hierarchiczną strukturę tzw. *loggerów* odpowiadającą funkcjonalności systemu. Logowanie na najwyższym poziomie *gadatliwości* (ang. *verbosity*) pozwala na szczegółowe śledzenie pracy aplikacji i umożliwia rozwiązywanie ewentualnych problemów. Ułatwia to zarówno procedurę niezautomatyzowanego testowania, jak i umożliwia logowanie informacji dla administratora systemu.

Aplikacja ma zdefiniowaną klasę zamykającą ją w kontrolowany sposób po otrzymaniu sygnału od systemu. Zamknięcie aplikacji powoduje m.in. anulowanie wszystkich wykonywanych zadań w systemie, aby mimo zamknięcia lub awarii zarządcy pozostawić środowisko w stanie umożliwiającym pracę lokalnym użytkownikom. Moduły aplikacji są zamykane (jeśli zostały zainicjalizowany) w kolejności odwrotnej do uruchamiania.

Kod zamykający aplikację jest również wywoływany w momencie niespełnienia asercji, które umieszczone są w niektórych miejscach w kodzie, w połączeniu z mechanizmem logowania. Ma to przede wszystkim zapobiegać sytuacji, gdy z powodu błędu lub awarii aplikacji zarządcy lokalny użytkownik nie mógłby w normalny sposób pracować na komputerze.

Szczegółowy sposób uruchomiania aplikacji znajduje się na załączonej do pracy płycie.

4.4 Aplikacja zgłaszająca zadania — Submitter

Podstawową funkcjonalnością realizowaną przez aplikację tzw. submittera jest dostarczenie użytkownikowi możliwości zlecenia nowego zadania w systemie. Do zadań użytkownika należy wyspecyfikowanie parametrów zgłaszanego zadania, natomiast aplikacja submittera zajmuje się komunikacją z modułem managera, zgłoszeniem u niego nowego zadania zgodnie z zadanymi parametrami, pobraniem listy węzłów oraz wykonaniem na nich wskazanej przez użytkownika komendy.

4.4.1 Implementacja

Do realizacji zadania użyty został język Java z uwagi na łatwość tworzenia oprogramowania oraz dostępność gotowej infrastruktury do komunikacji z managerem poprzez mechanizm Web Services. Program submittera został stworzony jako aplikacja wykonywana z linii poleceń, której przekazywane są opcje określające parametry zgłaszanego zadania. Możliwe do przekazania opcje — wraz z opisem i klauzulą czy dana opcja jest wymagana — podane zostały w tabeli 4.6. Argumenty następujące po parametrach zadania traktowane są jako komenda (wraz z opcjonalnymi jej argumentami) do wykonania na przydzielonych węzłach.

Opcja	Opis	Wymagana
`-j`	Opisowa nazwa zadania.	Nie
`-m`	Minimalna liczba węzłów na których powinno być wykonane zadanie — gwarantowane jest przydzielenie co najmniej takiej liczby węzłów.	Tak
`-g`	Nazwa grupy węzłów preferowana przy wyborze węzłów przydzielanych do zadania. (Może wystąpić wielokrotnie.)	Nie
`-r`	Sugerowana liczba węzłów przydzielonych do zadania (wartość ta powinna być nie mniejsza od wartości opcji `-m`) – – nie jest gwarantowane przydzielenie takiej liczby węzłów.	Tak
`-s`	Czy całe zadanie powinno być przerwane w momencie zakończenia przetwarzania na którymkolwiek z węzłów.	Nie
`-u`	Adres URL wskazujący serwis managera.	Tak
`-w`	Maksymalny czas trwania zadania w sekundach.	Tak

Tab. 4.6: Opcje aplikacji submittera.

Cała funkcjonalność submittera została zaimplementowana w jednej klasie `PUTGridSubmitter` z której można korzystać w dwojaki sposób. Pierwszy z nich odpowiada wspomnianej już sytuacji w której parametry zadania podawane są jako argumenty w linii poleceń, czyli do konstruktora klasy przekazywane są wszystkie argumenty aplikacji. Drugi z nich polega na wykorzystaniu konstruktora domyślnego i ręcznym ustawieniu wszystkich parametrów. Co prawda drugi ze sposobów nie jest wykorzystywany w prezentowanym systemie, jednak pozwala w prosty sposób wykorzystać funkcjonalność submittera w innych aplikacjach pisanych w języku Java bez potrzeby uruchamiania nowego procesu.

Do przetwarzania argumentów linii poleceń użyto pakietu *args4j* [17]. Biblioteka ta wykorzystuje prosty w użyciu mechanizm adnotacji języka Java w postaci adnotacji `@Option` umieszczanej nad polem klasy w którym zapisywana jest wartość związana z daną opcją. Pewną niedogodnością biblioteki było zgłaszanie błędów w przypadku niezadeklarowanej w programie opcji (jako opcję rozumiany jest tutaj argument aplikacji zaczynający się znakiem `-`). Z uwagi na charakter aplikacji submittera dopuszczana jest możliwość podania opcji również dla komendy, która ma być wykonana na docelowych węzłach. Należało zatem zmienić zachowanie biblioteki args4j tak, aby dopuszczała nieznane opcje. Zostało to zrealizowane poprzez modyfikację mechanizmu przetwarzania argumentów tak, że pierwszy argument niebędący opcją wyłączał dalsze przetwarzanie opcji programu i wszystkie pozostałe opcje dopisywał do listy stanowiącej komendę do wykonania na węzłach. W związku z wprowadzoną modyfikacją należy mieć na uwadze, że w docelowym systemie powinna być zawsze używana biblioteka dostarczona wraz z systemem.

Po fazie przetworzenia argumentów, następuje faza zlecania zadania wyzwalana metodą `submit()`. Pierwszą czynnością wykonywaną w tej metodzie jest spraw-

dzenie poprawności wszystkich parametrów zadania przed wysłaniem ich do modułu managera. Następnie występuje próba nawiązania połączenia z serwisem managera, zalogowanie w systemie (dane potrzebne do zalogowania — użytkownik i hasło znajdują się odpowiednio w własnościach `putgrid.submitter.username` i `putgrid.submitter.password`) oraz próba zlecenia zadania. Należy w tym miejscu zwrócić uwagę na fakt rozdzielenia ról użytkowników systemowych, którzy biorą udział w procesie zlecania zadania. Ze względu na fakt, że uwierzytelnienie w usłudze SSH uruchomionej na węzłach dokonywane jest na podstawie klucza publicznego pojawia się konieczność udostępnienia pary kluczy: publicznego i prywatnego wszystkim użytkownikom, którzy mieliby prawo zlecania zadań. Jednak w przypadku dużej liczby takich użytkowników nadzór nad nimi staje się znacznie trudniejszy, a dostęp do każdego węzła i prawo zmiany jego stanu poprzez użycie aplikacji putgridctl może prowadzić do potencjalnych nadużyć. Aby zapobiec takiemu scenariuszowi w systemie wydzielany jest jeden specjalny użytkownik, który jako jedyny posiada odpowiednie klucze uwierzytelniające na wszystkich węzłach i z którego prawami powinna być wykonywana aplikacja submittera. Cel ten został osiągnięty dzięki użyciu pakietu *sudo* [46], który umożliwia wykonanie wskazanego polecenia przez wyszczególnionych użytkowników z prawami innego użytkownika. Narzędzie to umożliwia ograniczenie listy użytkowników uprawnionych do zlecania zadań, a przy tym nieposiadających dostępu do kluczy uwierzytelniających. Problem identyfikacji użytkownika zlecającego zadanie został rozwiązany poprzez odpowiednie skonfigurowanie pakietu sudo, w taki sposób, że nazwa użytkownika uruchamiającego program submittera zapisywana jest w zmiennej środowiskowej `USER` i ta właśnie nazwa przekazywana jest jako właściciel zlecanego zadania do modułu managera.

Jeżeli wszystkie kroki w komunikacji z managerem zakończą się sukcesem w odpowiedzi uzyskany zostanie identyfikator zleconego zadania. Na podstawie otrzymanego identyfikatora pobierana jest lista danych dotyczących węzłów przydzielonych do zadania. Każdy element na liście zawiera m.in. informację o adresie IP komputera oraz nazwę użytkownika której należy użyc w sesji SSH. Submitter przystępuje zatem do uruchomienia komendy na węzłach. Wykorzystując bibliotekę Trilead SSH-2 ustanawiana jest sesja SSH z każdym przydzielonym węzłem. Klucze uwierzytelniające wyszukiwane są w następującej kolejności:

- `$HOME/.ssh/id_rsa`
- `$HOME/.ssh/id_dsa`
- Ścieżka wskazywana przez wartość Java property o nazwie `putgrid.submitter.keyfile`

Jeżeli klucz zabezpieczony jest hasłem to hasło odczytywane jest z Java property o nazwie `putgrid.submitter.keyfilepassword`. Po ustaleniu sesji SSH uruchamiana jest na wszystkich węzłach jednocześnie komenda użytkownika używając do tego celu aplikacji putgridctl.

Z uwagi na duże znaczenie informacji pojawiających się na standardowym wyjściu komunikatów oraz standardowym wyjściu komunikatów błędów dla aplikacji wykorzystujących bibliotekę ProActive należało zapewnić przekierowanie tych wiadomości ze wszystkich węzłów. Funkcję tą realizuje metoda `handleSessions()`, która okresowo odpytuje każdą sesję o nowe dane na ich wyjściach komunikatów i jeśli takie dane wystąpią przekierowuje je na standardowe wyjścia uruchomionego procesu submittera. W celu zapobieżenia mieszania się danych z różnych sesji, dane są dodatkowo buforowane i wypisywane są tak, aby zapewniony był warunek, że jedna linia na standardowym wyjściu ma odzwierciedlenie w dokładnie jednej linii standardowego wyjścia pojedynczej sesji. Ze względu na fakt przekazywania standardowych wyjść komunikatów proces submittera trwa tak długo jak długo pozostaje aktywna, którakolwiek z sesji. Dlatego też w momencie przerwania przez użytkownika procesu submittera wysyłane jest do managera żądanie przerwania zadania.

4.5 Integracja z biblioteką ProActive

Aby umożliwić uruchamianie aplikacji wykorzystujących bibliotekę ProActive w opisywanym systemie, należało stworzyć w infrastrukturze tej biblioteki nowy proces odpowiedzialny za uruchomienie aplikacji submittera wraz z odpowiednimi parametrami zadania. Proces ten został zaimplementowany w klasie `PUTGridProcess` dziedziczącej po klasie abstrakcyjnej `AbstractExternalProcessDecorator`. Podstawowym zadaniem tej klasy jest przechowywanie parametrów zadania dotyczących opisywanego systemu oraz zbudowanie polecenia uruchamiającego proces submittera z wyspecyfikowanymi wszystkimi wymaganymi parametrami. Polecenie to może być dołączone hierarchicznie do poleceń innych procesów zgodnie z opisem zamieszczonym w deskryptorze XML. Zadanie tworzenia polecenia submittera jest zasadniczo proste i sprowadza się do konstrukcji napisu z nazwą programu submittera oraz ewentualnym dołączeniem odpowiednich opcji oraz wartości tych opcji.

W celu wypełnienia obiektu klasy `PUTGridProcess` parametrami użytkownika, należało dodać do mechanizmu przetwarzania deskryptora XML obsługę nowego procesu. Aby to zrealizować należało po pierwsze uzupełnić tzw. *schemat XML* (ang. *XML Schema*) opisujący składnię całego deskryptora XML na podstawie którego dokonywane jest sprawdzenie jego poprawności pod względem składniowym. Dodany został do niego nowy element o nazwie `putgridProcess` grupujący opcje przekazywane do procesu `PUTGridProcess`. Element ten posiada dwa atrybuty: `url` będący atrybutem obowiązkowym i wskazujący adres URL serwisu managera — odpowiada on opcji `-u` submittera (por. tabela 4.6) oraz `commandPath` będący atrybutem opcjonalnym wskazującym na ścieżkę do programu submittera — jeżeli taka ścieżka nie będzie podana przyjmowana jest domyślna wartość `/usr/bin/pgsub`. Opcje służące do ustalania parametrów zgłaszanego zadania znajdują się w elemencie podrzędnym o nazwie `putgridOption`. Element ten zawiera dalsze elementy podrzędne, których

wartości definiują wartość odpowiadającego parametru zadania. Tabela 4.7 przedstawia wszystkie możliwe nazwy znaczników jakie można wykorzystać w elemencie `putgridOption` oraz odpowiadające im opcje submittera (por. tabela 4.6).

Nazwa znacznika	Opcja submittera
`<jobName>`	`-j`
`<minMachines>`	`-m`
`<preferredGroups>`	`-g`
`<requestedMachines>`	`-r`
`<stopAllOnAnyFault>`	`-s`
`<walltime>`	`-w`

Tab. 4.7: Parametry zadania w deskryptorze XML.

Dodatkowo należało stworzyć klasę pośredniczącą między klasą `PUTGridProcess`, a plikiem deskryptora XML, która zajmowałaby się wczytaniem danych zwartych w poprawnie skonstruowanym deskryptorze i ustawiała opcje odpowiednie opcje procesu. W tym celu stworzono klasę `PutgridProcessExtractor` dziedziczącą po klasie `ProcessExtractor`. Klasa ta otrzymuje w konstruktorze wskazanie na element w deskryptorze XML odpowiadający elementowi `putgridProcess` zgodnie z modelem przetwarzania XML DOM (ang. *XML Document Object Model*) dzięki któremu możliwe jest odniesienie się do wszystkich atrybutów tego elementu oraz do jego elementów podrzędnych. Klasa ta pobiera na początku wszystkie atrybuty elementu i ustawia odpowiadające im wartości w procesie `PUTGridProcess`, a następnie analogiczne postępowanie przeprowadzane jest dla wszystkich elementów podrzędnych elementu `putgridOption`, czyli w procesie ustawiane są parametry zlecanego zadania. Dalszą częścią konstrukcji ostatecznej komendy submittera zajmuje się już wewnętrzny mechanizm biblioteki ProActive wykokrzystując do tego celu przygotowany obiekt klasy `PUTGridProcess`.

4.6 Interfejs WWW

Jedną z najważniejszych funkcji projektowanego systemu, szczególnie z punktu widzenia skoncentrowanego na obliczeniach użytkownika, jest możliwość monitorowania w możliwie łatwy sposób zarówno zleconych przez siebie zadań, jak i dostępności całego środowiska. Szczegółowych informacji o bieżącym stanie systemu i jego konfiguracji dostarczać ma przejrzysty interfejs, który pozwala na przeglądanie zawartości bazy danych oraz wydawanie poleceń związanych z poszczególnymi hostami i zadaniami.

4.6.1 Użyte technologie

Jeżeli zdecydowanoby się na użycie platformy JEE, interfejs udostępniony zostałby zapewne przy użyciu technologii JSP (*JavaServer Pages*) lub JSF (*JavaServer Faces*). Mogłoby to być o tyle wygodne, że tzw. kontener komponentów webowych (ang. *web container*) współistniałby razem z kontenerem komponentów EJB zarządzajacych systemem (w tym komunikujących się z bazą danych) w ramach jednego serwera aplikacji. Byłoby to typowe wykorzystanie wielowarstwowej architektury aplikacji Java Enterprise Edition w pełni wspieranych ze strony tej platformy [10].

W nowej koncepcji, po rezygnacji z użycia JEE, należało podjąć decyzję co do modelu aplikacji internetowej. Ze względu na miejsce działania kodu programu aplikacje webowe można bowiem podzielić na *fat client applications*, czyli tradycyjne aplikacje desktopowe, działające samodzielnie na maszynie użytkownika oraz *thin client applications*, które dla odmiany działają na serwerze, podczas gdy komputer użytkownika wymagany jest jedynie do wyświetlenia graficznego interfejsu (zazwyczaj w postaci strony internetowej w przeglądarce). Ostatecznie zdecydowano się na rozwiązanie pośrednie, czyli tzw. *Rich Internet Application* (RIA) w postaci technologii **Google Web Toolkit** (GWT [8] [2] [5]).

4.6.1.1 Rich Internet Applications i AJAX

Model Rich Internet Application łączy w sobie zalety obu wspomnianych już wcześniej używanych modeli. Podobnie jak *thin client applications* nie wymaga instalacji żadnego oprogramowania poza standardową przeglądarką internetową. Fakt ten wspiera przenośność tego typu aplikacji — może być używana dowolna przeglądarka pod dowolnym systemem operacyjnym. Niemniej jednak RIA posiada także pewne cechy *fat client applications*. Część kodu wykonywana jest po stronie klienta, co pozwala odciążyć serwer, a graficzne interfejsy użytkownika mogą być równie atrakcyjne jeśli chodzi o interakcję z użytkownikiem — wspierane są między innymi akcje typu *drag-and-drop*.

Pozostałe cechy charakterystyczne RIA to między innymi pobieranie większości danych z serwera na początku sesji oraz prezentacja ich przez cały czas w postaci jednej strony (zmieniane są jedynie jej fragmenty) — czyli tzw. *one-screen-application*. Skutkuje to brakiem ciągłego odświeżania zawartości i uciążliwego przeładowywania strony. Sprawia to wrażenie pracy w typowej aplikacji desktopowej. Dodatkowe dane pobierane są tylko w razie żądania użytkownika. Jednymi z głównych reprezentantów modelu RIA są rozwiązania bazujące na technologii AJAX (*Asynchronous JavaScript and XML*), w której to wspomniana obsługa poleceń użytkownika i pobieranie danych z serwera odbywa się w sposób asynchroniczny. Pozwala to na wymiane informacji pomiędzy klientem a serwerem bez konieczności przerywania pracy użytkownika — może on nadal używać interfejsu w oczekiwaniu na pobranie danych. Dość popularną techniką związaną z technologią AJAX jest *prefetching*, czyli wcześniejsze pobieranie danych bez wiedzy użytkownika, co do których wystę-

puje wysokie prawdopodobieństwo, że będą wkrótce potrzebne. Technika ta wykorzystywana jest np. w aplikacji Google Maps, gdzie podczas przeglądania danego fragmentu mapy automatycznie pobierane są fragmenty sąsiednie.

Oczywiście aplikacje RIA nie są pozbawione wad. Niektóre z nich związane są ściśle z cechami stanowiącymi o wcześniej wspomnianych korzyściach. Między innymi to, że większość danych (głównie skryptów) pobieranych jest na początku sesji wpływa niekorzystnie na czas pierwszego ładowania aplikacji. Poza tym RIA opiera się zazwyczaj na wykorzystaniu po stronie klienta (jako tzw. *client engine*) języka skryptowego — w przypadku AJAX jest to JavaScript. Prawidłowe działanie wymaga więc włączenia obsługi skryptów w przeglądarce, co z kolei powoduje pewne zagrożenia bezpieczeństwa (szczegółowo opisano to w szczegółach implementacji). Co więcej JavaScript jako język interpretowany jest znacznie wolniejszy niż języki kompilowane stosowane w typowych desktopowych aplikacjach (również mogących działać w roli klientów webowych). Pewne problemy powodować mogą również różnice występujące pomiędzy przeglądarkami jeżeli chodzi o wspieranie modelu DOM (*Document Object Model*), który jest podstawą działania języka JavaScript. Na zakończenie opisu wad i zalet aplikacji RIA warto jeszcze wspomnieć, że ich architektura narusza paradygmat działania tradycyjnych stron internetowych. Klasyczne aplikacje internetowe mogą być bowiem postrzegane jako seria zapytań HTTP GET, powodujących pobieranie nowych stron, podczas gdy RIA w wersji AJAX opiera się na asynchronicznych zapytaniach i modyfikacjach jednej złożonej strony pobranej tylko raz, na początku działania.

4.6.1.2 GWT

Google Web Toolkit jest otwartym zestawem narzędzi znacznie ułatwiających rozwijanie aplikacji internetowych opartych o technologię AJAX. Pozwala ono na pisanie całego kodu (zarówno po stronie serwera, jak i klienta) wyłącznie w języku Java (niestety w tej chwili tylko w wersji 1.4), poprzez udostępnienie kompilatora tłumaczącego aplikację do kompatybilnego ze wszystkimi przeglądarkami JavaScriptu i HTMLa. Pozwala to niemalże całkowicie abstrahować od uciążliwych szczegółów związanych z modelem DOM. Dzięki dostarczonej bibliotece kontrolek (ang. *UI widgets*) implementacja przypomina pracę z biblioteką Swing, a co więcej Google promuje upowszechnianie własnych komponentów interfejsu użytkownika, więc gotowych rozwiązań przybywa. Ponieważ kod jest pisany w Javie, umożliwia to wykorzystanie wszystkich obiektowych cech tego języka, jak również stosowanie zorientowych obiektowo wzorców projektowych.

Wśród zaawansowanych funkcji dostarczonych przez GWT można wyróżnić mechanizm RPC (*Remote Procedure Call*) odpowiający za asynchroniczną komunikację z serwerem i działającymi tam serwletami oraz możliwość łączenia implementacji z fragmentami języka JavaScript dzięki JSNI (*JavaScript Native Interface*). Dodatkowo zadbano o integrację z narzędziem JUnit pozwalającym na testowanie jednostkowe wytwarzanego oprogramowania. Dużo prostsze niż w klasycznych, czystych

rozwiązaniach AJAX jest też zarządzanie historią w przeglądarce. Dostosowywanie stylu graficznego interfejsu aplikacji oprzeć można na wykorzystaniu kaskadowych arkuszy stylu (CSS).

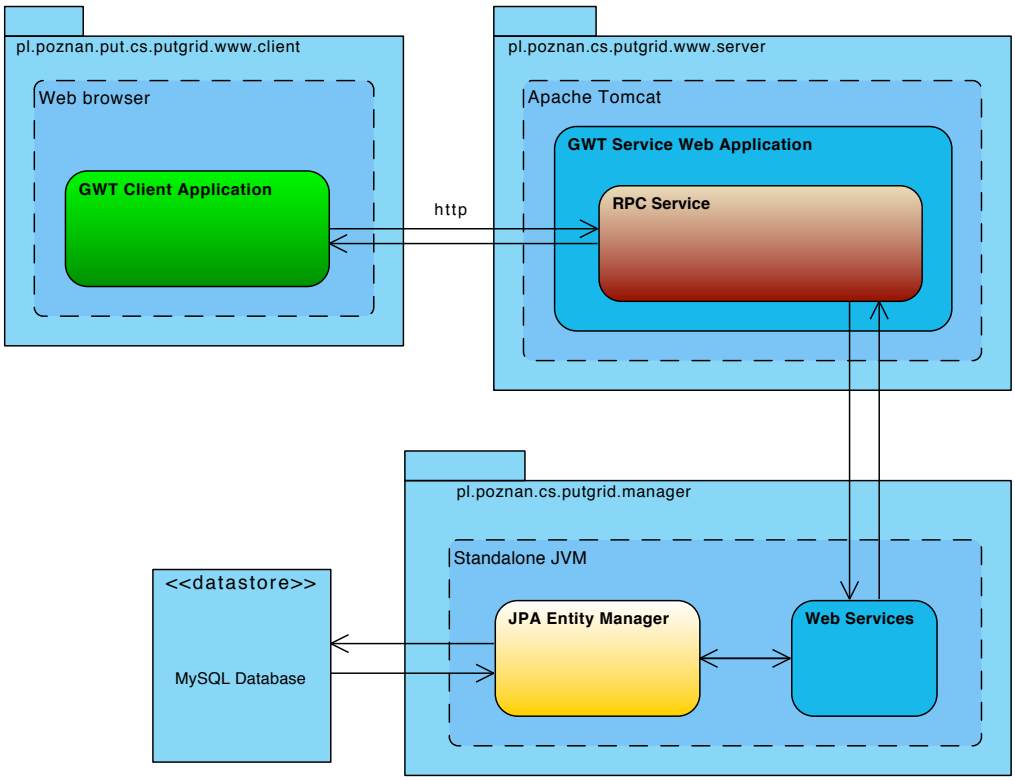
4.6.1.3 Web services

Opisywany interfejs WWW realizuje koncepcję klienta usługi sieciowej udostępnianej przez moduł zarządzający. W naturalny sposób zapewnia użytkownikom końcowym dostęp do głównych funkcji systemu. Dlatego też w ramach aplikacji należało zrealizować implementację klienta Web Services, którą oparto o biblioteki Apache Axis. Dokładny opis technologii zawarto w punkcie 4.2.

4.6.2 Architektura

W odróżnieniu od większości biznesowych aplikacji internetowych operujących na trzech warstwach (klient, serwer, baza danych), architektura opisywanej aplikacji jest czterowarstwowa. W dostępie do bazy danych pośredniczy bowiem jeszcze Manager, który udostępnia uniwersalny interfejs w postaci usług Web Services (WS). Wynika to z dwóch istotnych przyczyn. Po pierwsze funkcje jakich dostarczać ma interfejs WWW nie są ograniczone jedynie do odczytywania i modyfikacji zawartości bazy danych. W skład zdefiniowanych usług WS wchodzą także polecenia wydawane poszczególnym hostom (np. shutdown), jak również metody związane z autoryzacją i uwierzytelnianiem realizowane przy rozpoczęciu sesji. Po drugie natomiast udostępnienie spójnego zestawu usług jest podejściem bardzo elastycznym, umożliwiającym wykorzystaniem dowolnego rodzaju klienta.

Na diagramie 4.6 przedstawiono przepływ informacji w systemie przy realizacji żądania użytkownika lub odświeżaniu zawartości strony. Zapytanie inicjowane jest więc bezpośrednio przez klienta (wyzwolone zdarzeniem na poziomie GUI) lub przez obiekt klasy `Timer` regularnie odpytujący serwer w celu aktualizacji danych (stosowana technika nosi nazwę *polling*). Żądanie wędruje z wykorzystaniem protokołu GWT-RPC do skojarzonej z nim usługi RPC zaimplementowanej w postaci serwletu wdrożonego w kontenerze Apache Tomcat. Poszczególne metody serwletów stosują w roli delegatów odpowiadające metody usług Web Service, które to wykorzystują między innymi wcześniej już opisanych zarządców odpowiadających za połączenie z bazą danych. Uzyskany wynik (zazwyczaj w postaci obiektu klasy odpowiającej danej relacji w bazie danych lub wyjątku w razie niepowodzenia) propagowany jest drogą powrotną do klienta, gdzie jego dalszym przetwarzaniem zajmuje się metoda wywołania zwrotnego (ang. *callback*). Wszystko to realizowane jest w sposób asynchroniczny względem interfejsu użytkownika, co umożliwia kontynuowanie pracy w trakcie oczekiwania na wynik.



Rys. 4.6: Przepływ informacji przy zapytaniu klienta.

4.6.3 Szczegóły implementacji

4.6.3.1 Bezpieczeństwo

Nie od dziś wiadomo, że wykorzystanie języka JavaScript wymaga od programisty zwrócenia szczególnej uwagi na niektóre kwestie bezpieczeństwa. Często aplikacje pisane w tym języku nie są wystarczająco zabezpieczone przed szczególnymi rodzajami ataków np. *Cross Site Scripting* (XSS) czy *Cross Site Request Forgery* (XSRF lub CSRF). Ponieważ GWT z założenia daje możliwość operowania wyłącznie na wysokim poziomie bez zaglądania w języki skryptowe, do pewnego stopnia ułatwia też radzenie sobie ze wspomnianymi problemami. Dobrze jest jednak mieć je nadal na uwadze zwłaszcza projektując system, którego jedną z głównych cech ma być właśnie bezpieczeństwo użytkowników i udostępnianych zasobów. O ile nie było większych problemów z podatnością na ataki typu XSS, ponieważ nie stosowano własnych fragmentów kodu JavaScript (przy użyciu JSNI lub zewnętrznych bibliotek) a kod generowany przez GWT jest pod tym względem bezpieczny, to należało uważnie rozpatrzyć potencjalne luki umożliwiające XSRF [9] [3].

4.6.3.2 Logowanie

Dostęp do całego środowiska, a w szczególności do interfejsu WWW powinien być ograniczony, niezbędne jest więc wykorzystanie odpowiednich mechanizmów uwie-

rzytelniania i autoryzacji także na poziomie GWT. Pierwszą czynnością wymaganą od użytkownika jest więc logowanie — podanie nazwy użytkownika oraz hasła. Wcześniej administrator — również za pomocą interfejsu WWW — może założyć zainteresowanym osobom konta w systemie. Informacje o kontach przechowywane są w bazie danych (hasła nie są oczywiście zapisywane w postaci jawnej, lecz zaszyfrowanej przy użyciu algorytmu haszującego MD5). Po wprowadzeniu przez użytkownika wymaganych informacji następuje weryfikacja poprawności poprzez porównanie skrótów — znajdującego się w bazie danych ze świeżo obliczonym na podstawie podanego hasła. Odbywa się to po stronie Managera, który to zarządza uwierzytelnianiem oraz w przypadku poprawności danych generuje zwracany użytkownikowi identyfikator sesji, który sam zapamiętuje.

Za możliwość logowania odpowiada usługa RPC o nazwie `LoginService`. Udostępnia ona metodę o nazwie `login` zwracającą w przypadku powodzenia identyfikator sesji w postaci łańcucha znaków. Identyfikator ten jednoznacznie określa użytkownika oraz jego rolę, a co się z tym wiąże prawa do wykonywania poszczególnych operacji na poziomie interfejsu WWW. Zapisywany jest on w pliku *Cookie*, co pozwala na zapamiętanie zalogowanego użytkownika na pewien czas nawet jeśli opuści on stronę bądź wyłączy przeglądarkę. Przy ponownej próbie użycia interfejsu nastąpi sprawdzenie, czy takie ciasteczko istnieje i czy jest jeszcze ważne (okresem ważności zarządza Manager).

4.6.3.3 Zarządzanie sesją

Identyfikator sesji jest dodatkowo przechowywany w statycznej zmiennej aplikacji GWT po stronie klienta. Dołączany jest on do każdego zapytania RPC w celu autoryzacji użytkownika oraz sprawdzenia ważności sesji. Pewne operacje udostępnione przez interfejs WWW są przeznaczone tylko dla administratora (np. zarządzanie kontami użytkowników). Oczywiście wygląd interfejsu dostosowany jest do roli zalogowanego użytkownika, jednak poza tym każde AJAXowe wywołanie również weryfikuje prawa do pobrania danych informacji — następuje to po stronie Managera, który pamięta wysłane identyfikatory oraz skojarzone z nimi role użytkowników. Gdyby opierano się przy tym tylko na identyfikatorze wysyłanym w nagłówku Cookie byłaby to poważna luka w bezpieczeństwie, umożliwiająca atak XSRF — zwany też bardziej obrazowo *session riding* lub *one click attack*. Dlatego też informacje przesyłane w nagłówku Cookie nie są brane pod uwagę a sam identyfikator jest jawnie przesyłany razem z danymi. Potencjalna próba ataku wymagałaby więc znajomości przez atakującego wartości Cookie, którego uzyskanie (m.in. ze względu na *Same Origin Policy*) jest znacznie utrudnione.

4.6.3.4 Komunikacja GWT-RPC

Mechanizm zdalnych wywołań GWT-RPC można podzielić na trzy części. Pierwsza to usługa udostępniona w ramach serwletu działającego na serwerze, druga to

przeglądarka internetowa jako klient wywołujący metody interfejsu usługi, a trzecia to dane transmitowane pomiędzy klientem i serwerem. Poniżej opisano w sposób bardziej szczegółowy każdą z tych części i jej implementację w opisywanej aplikacji.

Usługi GWT-RPC. Dostarczenie usługi wykonywanej na serwerze składa się z definicji interfejsu opisującego daną usługę oraz jego implementacji. Interfejs ten powinien rozszerzać `RemoteService`, a wszystkie metody powinny zwracać i przyjmować jedynie wartości typów serializowalnych. GWT na jego podstawie potrafi wygenerować pośrednika — *proxy*, który pozwala klientowi odwoływać się do zdalnej usługi jak do lokalnego obiektu. Implementacją tego interfejsu jest z kolei serwlet dziedziczący po `RemoteServiceServlet`.

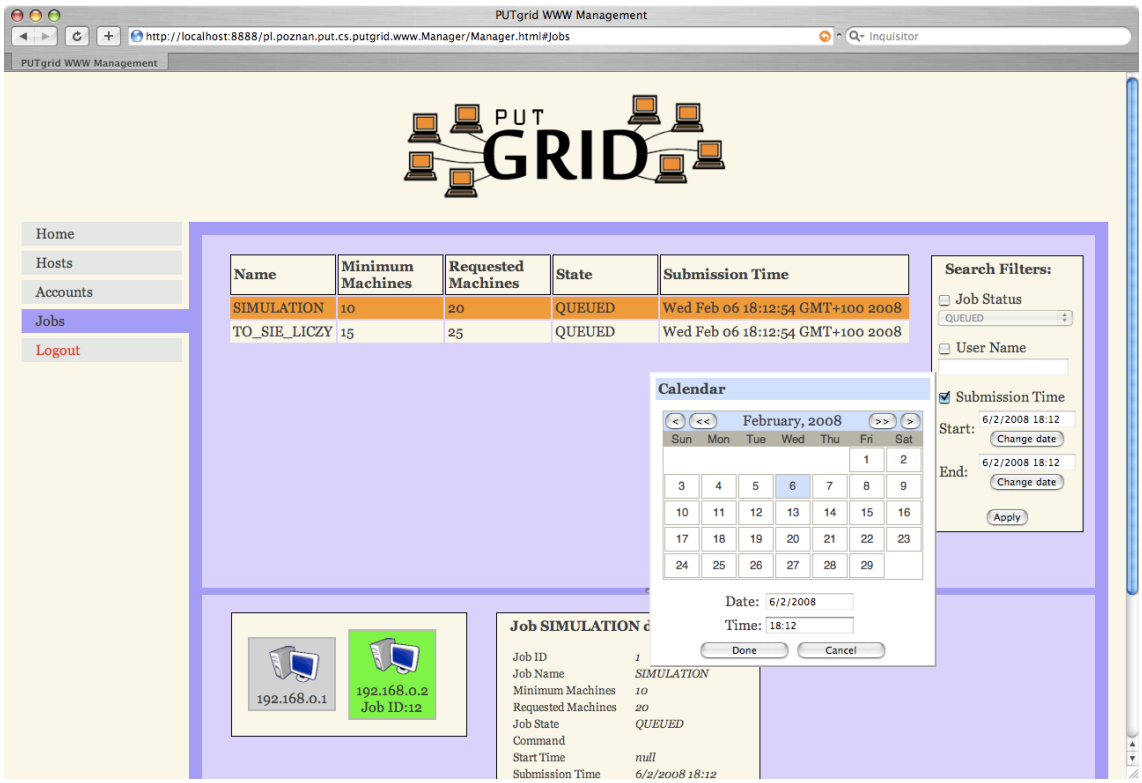
W opisywanej aplikacji, poszczególne usługi RPC odpowiadają zazwyczaj opisanym wcześniej komponentom encyjnym w bazie danych. Każda usługa (taka jak np. `AccountService` czy `JobService`) dostarcza metod pozwalających na pobieranie krotek z bazy danych oraz w pewnych przypadkach ich modyfikację. Poza tym usługa `LoginService` pozwala na uwierzytelnianie logującego się użytkownika, a `HostService` na wykonywanie poleceń takich jak np. *shutdown* czy *wake-up* na poszczególnych komputerach. Interfejsy wszystkich usług są oczywiście zgodne z interfejsami Web Services, a ich implementacja w większości przypadków sprowadza się jedynie do delegacji wywołania danej metody.

Klient. Wywoływanie metod interfejsów zdalnych usług polega na pobraniu obiektu pośrednika danej usługi oraz zleceniu mu wywołania danej metody. Implementację pośrednika dostarcza GWT, a jego instancję można uzyskać przy pomocy techniki zwanej opóźnionym wiązaniem (ang. *deferred binding*). Wymagane jest przy tym podanie punktu wyjścia (ang. *entry point*) danego serwisu, zdefininiowanie interfejsu wywołań asynchronicznych oraz dostarczenie obiektu wywołania zwrotnego (ang. *callback object*) implementującego interfejs *AsyncCallback*.

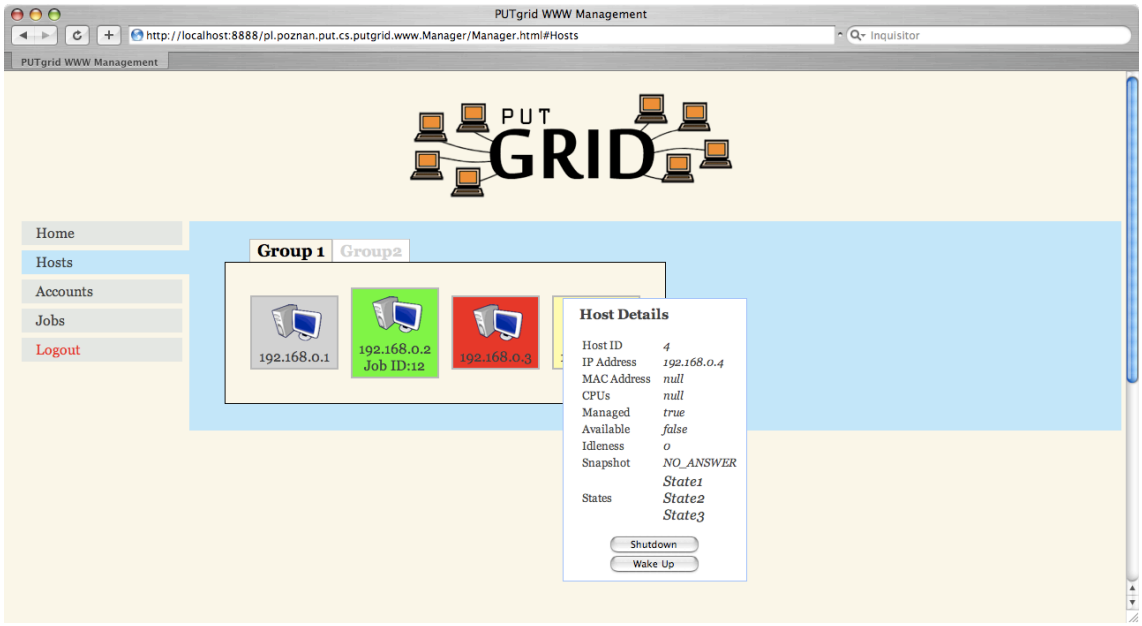
Jak widać samo wywoływanie metod zdalnych usług serwera jest działaniem dość złożonym. Dlatego też, w celu oddzielenia konstrukcji programistycznych związanych z nawiązywaniem połączenia i uzyskiwaniem dostępu do pośrednika od pozostałej logiki aplikacji klienckiej posłużono się wzorcem projektowym *Facade* [7]. Zmniejszono tym samym złożoność, redukując liczbę klas z którymi występowały bezpośrednie związki, a co ważniejsze uzyskano wysokopoziomowy interfejs ułatwiający znacznie używanie zdalnych wywołań. Wszystkie szczegóły związane z inicjalizacją wywołania zawarto w metodach klas `*DataProvider` zaimplementowanych zgodnie z wzorcem *Singleton*. Część z tych klas implementuje także interfejs `TableDataProvider`, co pozwala na prezentację pobieranych krotek w przejrzystej postaci tabelarycznej.

Przesyłanie danych. Dane przesyłane zarówno w postaci parametrów, jak i rezultatów zdalnych wywołań muszą spełniać określone wymagania — możliwa po-

winna być ich efektywna serializacja. W skład spełniających te wymagania standardowych typów Javy wchodzą wszystkie typy z dostarczonej w ramach GWT *JRE Emulation Library* (podzbioru funkcji środowiska uruchomieniowego, dla których kompilator potrafi wygenerować odpowiadający kod w języku JavaScript). Dodatkowo pozostaje możliwość definiowania własnych typów implementujących interfejs `IsSerializable`. Tak czy inaczej jednak dane powinny być zgodne ze składnią Javy w wersji 1.4, co wyklucza stosowanie typów wyliczeniowych i generycznych. Rodzi to pewne problemy związane z wcześniej opisanymi klasami reprezentującymi komponenty encyjne. Nie można ich przesyłać bezpośrednio do klienta GWT, ze względu na obecność adnotacji oraz składowych typu `enum`. Niezbędne stało się więc zaimplementowanie dodatkowych klas adaptujących klasy encji do postaci zgodnej ze starszą wersją języka. Klasy te, których nazwy opatrzono przyrostkiem `Data` w stosunku do oryginalnych nazw encji, zawierają jedynie pola typów serializowalnych oraz metody dostępowe.



Rys. 4.7: Wygląd interfejsu WWW.



Rys. 4.8: Wygląd interfejsu WWW.

Rozdział 5

Podsumowanie

5.1 Stan realizacji

W ramach pracy wykonano implementację opisanych modułów aplikacji. Sprawdzono także wiele rozwiązań takich jak mechanizm wdrażenia w ProActive z zastosowaniem prototypowych aplikacji, jeszcze przed implementacją właściwych modułów. Poznania wymagało wiele technologii, takich jak GWT, JPA czy JMX. Analizie podlegały także protokoły sieciowe i ich implementacje: m.in. SOAP (Web Services), SNMP, WOL. Tworzenie prostych aplikacji sprawdzających możliwość wykonania danego zadania w orkeślonej technologii (tzw. *spike solutions*) miało na celu obranie właściwej architektury na możliwie wczesnym etapie projektu.

System wymaga aktualnie integracji modułów i przeprowadzania intensywnych testów systemowych. Oprócz sprawdzenia zdolności systemu do produkcyjnego wykonywania obliczeń, wymaga on też kontroli w kontekście reakcji lokalnych użytkowników komputerów — czy nie przeszkadza w ich codziennej pracy.

Opis konfiguracji i uruchomiania poszczególnych aplikacji systemu znaleźć można na załączanej do pracy płycie CD.

5.2 Pomysły rozszerzeń systemu

W trakcie projektowania i rozwoju systemu, pojawił się szereg pomysłów na jego rozszerzenie. Wynikały one z obserwacji zachowania aplikacji — jej potencjalnych niedogodności, jak i badania możliwości zwiększenia efektywności pracy z systemem.

W aktualnej wersji centralnego zarządcy większość konfiguracji wczytywana jest z pliku XML. Przez Web Services oraz WWW możliwa jest aktualnie edycja kont w systemie. Użyteczna byłaby *możliwość dynamicznej zmiany konfiguracji* węzłów, bez potrzeby przeładowywania zarządcy. Interesujące wydaje się umożliwienie wykonywania tych zmian zarówno przez interfejs WWW, jak i aplikację konsolową (klienta Web Services), gdyby administrator chciał stworzyć własne skrypty do konfiguracji środowiska. Mając na względzie realizację tego pomysłu, już teraz konfi-

guracja węzłów jest w trakcie wykonania aplikacji ładowana do bazy danych. W module WWW nie są też wyświetlane logi systemu, ponieważ nie są one zapisywane w ustrukturyzowanej postaci przez moduł Manager. W rezultacie wywoływane (asynchronicznie) przez administratora polecenie zakończone błędem nie pozostawia po sobie śladu w interfejsie (poza sytuacją gdy poważny błąd powoduje wyłączenie zarządcy), a jedynie w logach pliku tekstowego. *Delegowanie logów log4j do bazy danych* w uschematyzowany sposób mogłoby rozwiązać ten problem, umożliwić prezentację logów w WWW, powiązanych z kontekstem, np. węzłem którego dotyczą.

Interesująca wydaje się też *rozbudowa systemu kolejek dla zadań* w sposób zależny od obserwowanego zachowania systemu przy znaczącym obciążeniu. W bieżącej wersji Manager i Submitter nie stosują systemu priorytetów zadań. Możliwe wydaje się dodanie takiej funkcji, jak i próby optymalizacji algorytmu kolejkowego przez wykorzystanie wiedzy o maksymalnym czasie wykonywania zadania. Gdyby system był intensywnie używany można zaimplementować obsługę więcej niż jednej kolejki zadań, jak również podjąć próbę wyeliminowania dwóch mniej istotnych ograniczeń w zarządcach węzłów. Ograniczenia te to brak możliwości wykonywania różnych zadań na jednym węźle, jak i uruchamiania (nie wykonywania) na raz więcej niż jednego zadania.

W aplikacji Submitter można podjąć próbę *zwiększenia wydajności i bezpieczeństwa*. Zwiększenie wydajności (zmniejszenie obciążenia komputera, na którym jest zainstalowany Submitter) można spróbować osiągnąć przez zastosowanie współdzielonej sesji SSH dla wielu procesów jednego zadania na jednym węźle, jak również przez zaimplementowanie opcji braku przekazywania wyjścia z procesu. Zabiegi potencjalnie zwiększające bezpieczeństwo Submittera to przeniesienie określania reguł konwersji poleceń do Managera, kontrola składni wykonywanych poleceń, jak i zdefiniowanie reguł *Java Security Policy* na docelowym węźle. Można również uruchamiać aplikacje obliczeniowe użytkownika na docelowych węzłach na prawach innego użytkownika, niż używany przez moduły Monitor i Submitter. Działania te nie mają jednak większego sensu, dopóki w samej bibliotece ProActive nie zostaną zaimplementowane *kompletne* metody uwierzytelniania i autoryzacji dostępu do węzła, jak i nie zostaną określone wymagania biblioteki w języku reguł Security Policy w platformie Java.

Bardziej zaawansowane możliwości rozwoju projektu to skonstruowanie systemu informującego aplikację użytkownika, że zostanie wywłaszczona z węzła ze względu na zalogowanie się w nim użytkownika. Mechanizm ten jest potencjalnie do zrealizowania z zastosowaniem *migracji obiektów* i tzw. usług technicznych dla węzłów (ang. *technical services*). Szczegółowego zbadania wymagałby jednak sposób działania *obiektu przekazującego* (ang. *forwarding-objects*), pozostawianego po migracji aktywnego obiektu, . W aktualnej wersji systemu możliwość obsługi awarii jest możliwa jedynie na poziomie aplikacji użytkownika lub poprzez zastosowanie mechanizmów *tolerowania awarii* (ang. *fault-tolerance*) implementowanych na poziomie biblioteki ProActive.

Inną możliwością przeciwdziałania błędom spowodowanym zachowaniami użytkownika lokalnego, jest unikanie wyboru węzłów na których może on potencjalnie pracować. Interesujące wydaje się podjęcie próby zaimplementowania algorytmu przydziału węzłów do zadania, wykorzystującego techniki wspomagania decyzji — opierając się na historii dostępności węzła.

Aktualna wersja systemu nie posiada również własnych mechanizmów zarządzania plikami — przede wszystkim transferu plików. Opiera się natomiast na implementacji transferu plików w bibliotece ProActive: *PFTP* (*ProActive File Transfer Protocol*). Poza tym, styl programowania w bibliotece ProActive zakłada najczęściej wykonywanie operacji na plikach w aplikacji użytkownika — wczytywanie i zapisywanie plików w głównej aplikacji na jego komputerze, a dalej przesyłanie instancji problemów do obliczeń lub ich wyników w postaci ustrukturyzowanych wiadomości (np. macierzy lub obiektów, w języku Java) z wykorzystaniem aktywnych obiektów, a nie transferu plików. Dodanie własnego mechanizmu transferu plików wymagałoby głównie rozbudowy modułu Submitter i mogłoby przyczynić się do potencjalnego wykorzystania implementowanego systemu dla aplikacji rozproszonych nie korzystających z biblioteki ProActive (nie posiadających własnych mechanizmów tego typu).

5.3 Równolegle rozwijany projekt w ramach biblioteki ProActive

W chwili powstawania pracy, tj. w styczniu 2008, wydana została[42] nowa wersja 3.9 biblioteki ProActive. Nowością jest w niej rozbudowany *Scheduling Framework* działający na poziomie biblioteki. Nie zbadano jeszcze szczegółowo funkcjonalności nowego modułu. Wiadomo, że jego architektura jest podobna koncepcyjnie do implementowanego systemu, ale o innej realizacji — w ramach biblioteki ProActive. W starszych wersjach biblioteki taki sposób zarządzania węzłami (wewnątrz aplikacji, w inny sposób niż za pomocą tradycyjnego modelu z użyciem procesów ProActive) nie był zalecany i traktowany był jako sposób agregowania zasobów *ad hoc*. Być może zmieniona została wewnętrzna architektura ProActive lub planowana jest taka zmiana. Scheduling Framework nie jest dedykowany do pracy w trybie „kradzieży cykli", ale nie zbadano jeszcze czy możliwa jest integracja takich mechanizmów.

Zaprezentowane rozwiązanie nie jest silnie zależne od ProActive i nie działa w obrębie biblioteki. Dlatego powinno być możliwe współdziałanie systemu z każdą wersją tej biblioteki. Implementacja różnych mechanizmów poza ProActive uniezależnia system od wewnętrznego API biblioteki, która nie wydaje się być stabilne ze względu na nieustanny jej rozwój.

Bibliografia

[1] B. Burke and R. Monson-Haefel: *Enterprise JavaBeans 3.0*, Helion, 2007

[2] E. Burnette: *Google Web Toolkit — Taking the pain out of Ajax*, The Pragmatic Bookshelf, 2006

[3] J. Burns: *Cross Site Reference Forgery — An intriduction to a common web application weakness*, Information Security Partners, 2005 http://isecpartners.com/documents/XSRF_Paper.pdf

[4] R. Buyya and S. Venugopal: *A Gentle Introduction to Grid Computing and Technologies*, CSI Communications, 2005 http://www.buyya.com/papers/GridIntro-CSI2005.pdf

[5] P. Chaganti: *GWT Java AJAX Programming*, Packt Publishing, 2007

[6] I. Foster: *What is the Grid? A Three Point Checklist* Argonne National Laboratory & Univeristy of Chicago, 2002 http://www-fp.mcs.anl.gov/~foster/Articles/

[7] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994

[8] R. Hanson, A. Tacy: *GWT in Action*, Manning, 2007

[9] K. Higgins: *CSRF Vulnerability: A 'Sleeping Giant'*, Dark Reading, 2006 http://www.darkreading.com/document.asp?doc_id=107651

[10] E. Jendrock, J. Ball, D. Carson, I. Evans, S. Fordin, K. Haase: *The Java EE 5 Tutorial, Third Edition*, Addison-Wesley, 2006

[11] A. Jasnowski: *JMX Programming*, Willey, 2002

[12] D. Luckham: *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*, Addison-Wesley Professional, 2002

[13] CERN LHC Computing Grid Project, http://lhcgrid.web.cern.ch/LHCgrid/

[14] EJB 3.0 Expert Group *JSR 220: Enterprise JavaBeans, Version 3.0 — Java Persistence API*, Sun Microsystems, 2005

[15] *Apache Commons Codec API* http://commons.apache.org/codec/

[16] *Apache Logging Services Project - Apache log4j* http://logging.apache.org/log4j/

[17] *args4j — Java command line options parser* https://args4j.dev.java.net

[18] *Cygwin — Linux-like environment for Windows* http://www.cygwin.com

[19] *Excerpts from A Conversation with Gordon Moore: Moore's Law*. Intel Corporation, 2005. ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf

[20] *gridengine* http://gridengine.sunsource.net/

[21] *HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer* http://haproxy.1wt.eu/

[22] *Hypertext Transfer Protocol — HTTP/1.1*, Request For Comment 2616 http://www.w3.org/Protocols/rfc2616/rfc2616.html

[23] *IANA — Internet Assigned Numbers Authority* http://www.iana.org

[24] *Java Beans* — API specification version 1.01, Sun Microsystems Inc., 1997

[25] *Javadoc Tool* http://java.sun.com/j2se/javadoc/

[26] Java Community Process *Java Management Extensions (JMX) Specification, version 1.4*, Sun Microsystems, Inc., 2006

[27] Java Community Process *Java Specification Request 3: Java Management Extensions (JMX) Specification* http://jcp.org/en/jsr/detail?id=3

[28] Java Community Process Program http://www.jcp.org/

[29] Sun Microsystems, Inc. *Java Management Extensions Technology*, http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/

[30] *JSR 101: Java APIs for XML based RPC*, Java Community Process Program

[31] *JSR 222: Java Architecture for XML Binding (JAXB) 2.0*, Java Community Process Program

[32] *JSR 224: Java API for XML-Based Web Services (JAX-WS) 2.0*, Java Community Process Program

[33] *JUnit testing framework* http://www.junit.org/

[34] *libconfig — C/C++ Configuration File Library* http://www.hyperrealm.com/libconfig

[35] *Net-SNMP package* http://net-snmp.sourceforge.net

[36] *Network Connectivity, Understanding WOL*, Intel Support Solution ID: CS-000084, http://www.intel.com/support/network/sb/CS-000084.htm

[37] *OpenPBS* http://www.openpbs.org/

[38] *OpenSSH — SSH connectivity tools* http://wwww.openssh.org

[39] *POSIX — Portable Operating System Interface* http://standards.ieee.org/regauth/posix/

[40] *Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures*, Request For Comment 1421 http://www.w3.org/Protocols/rfc2616/rfc2616.html

[41] *ProActive mailing list archive* http://mail-archive.objectweb.org/proactive/2007-04/msg00017.html

[42] *ProActive 3.9 Release Note* http://proactive.inria.fr/release_notes.htm#release21

[43] *ProActive User and Developer Manual*, http://proactive.inria.fr/manual_proactive.htm

[44] *SNMP4J — Free Open Source SNMP API for Java* http://www.snmp4j.org/

[45] *SSHWindows — OpenSSH for Windows* http://sshwindows.sourceforge.net

[46] *sudo — Allow command execution as root for specified users* http://www.sudo.ws/sudo/

[47] *The Secure Shell (SSH) Authentication Protocol*, Request for Comments: 4252 http://tools.ietf.org/html/rfc4252

[48] *Trilead SSH-2 for Java* http://www.trilead.com/Products/Trilead-SSH-2-Java/

[49] *Web Services Architecture*, W3C Working Group Note 11 February 2004, http://www.w3.org/TR/ws-arch/

[50] *Web Services Description Language (WSDL) 1.1*, W3C Note 15 March 2001 http://www.w3.org/TR/wsdl

[51] *Web Services Interoperability Basic Profile* http://ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile

[52] *Which style of WSDL should I use?* http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/

[53] *Wireshark — network protocol analyzer* http://www.wireshark.org/