Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Autoreferat rozprawy doktorskiej

Self-healing, role-based and autonomous monitoring system for distributed environments

Piotr Pęgiel

Promotor
prof. dr hab. inż. Jacek Kitowski

Promotor Pomocniczy
dr inż. Włodzimierz Funika

Wydział Informatyki, Elektroniki i Telekomunikacji
Katedra Informatyki

Kraków 2014

1 Wstęp

W dzisiejszych czasach systemy komputerowe stają się coraz bardziej złożone. Stwierdzenie to jest szczególnie aktualne w przypadku systemów rozproszonych. Dobrym przykładem trendu obrazującego lawinowy przyrost takich systemów są systemy oparte o chmury (*cloud storages*, *cloud computing*). Jeszcze kilka lat temu terminy te były znane jedynie wązkiej grupie naukowców lub ludzi pracującej w branży IT. Obecnie rozwiązania oparte na chmurach są dostępne nawet dla niezaawansowanych użytkowników.

Złożone, rozproszone systemy mogą być zbudowane z komponentów które potrafią ze sobą współpracować. Monitorowanie tak skomplikowanych systemu dostarcza szczególnych wyzwań. Rozproszenie systemu monitorowanego wymusza dekompozycje samego systemu monitorującego. System taki powinien wykazywać pewną *inteligencję*, dzięki której powinien wskazywać i podpowiadać użytkownikowi które elementy aplikacji powinny być monitorowane. Najbardziej zaawansowane systemy monitorujące potrafią działać w sposób autonomiczny – oznacza to, że część (lub większość) operacji jest wykonywanych bez ingerencji użytkownika.

Podobnie jak ma to miejsce w przypadku chmur, terminy takie jak *wysoka dostępność* (ang. High Availability) i Odporność na Błędy (Fault Tolerance) stają się coraz popularne. Systemy posiadające takie cechy stają się coraz bardziej dostępne dla końcowych użytkowników. Rewolucja ta wymusza również zmiany w systemach monitorujących. Rodzi się pytanie: *Jak powinno się monitorować system wysokiej dostępności?*

Zwiększająca się złożoność systemów monitorujących powoduje zwiększone ryzyko wystąpienia błędów. W takiej sytuacji system powinien być w stanie samodzielnie odzyskać sprawność (ang. recovery). Innymi słowy, powinien być w stanie przeprowadzić **samonaprawę** (ang. **self-healing**).

Na rynku istnieje bardzo duża ilość systemów monitorujących [13]. Niestety, istniejące rozwiązanie nie zawsze odpowiadają wymogom stawianym przez współczesne aplikacje. Obecne systemy monitorujące:

- służą do monitorowania aplikacji o wysokiej dostępności pomimo że same nie spełniają kryteriów wysokiej dostępności,
- są trudne w instalacji i użyciu,
- nie posiadają możliwości samonaprawy,
- nie integrują się z aplikacjami które monitorują (lub integracja taka jest utrudniona)

W celu monitorowania wysoko dostępnych aplikacji system monitorujący również powinien być aplikacją wysoko dostępną. Dzięki temu można zminimalizować ryzyko utraty danych zebranych podczas monitorowania.

Wysoka Dostępność

Dostępność systemu jest miarą określającą czy i w jakim stopniu system jest wstanie dostarczać wymaganą usługę. Podczas projektowania **Systemu Wysokiej Dostępności** szczególny nacisk kładzie się na zmniejszenie prawdopodobieństwa wystąpienia krytycznych problemów dzięki czemu usługa może działać w sposób możliwie ciągły [2]. Dostępność systemu najczęściej jest definiowana jaka procent czasu w jakim system jest dostępny na przestrzeni roku. System którego dostępność wynosi 99.999% jest zazwyczaj uważany jako system wysokiej dostępności (ang. highly available systems). Oznacza to, że jego przestój (ang. downtime) nie powinien trwać dłużej niż 5.5 minuty w ciągu roku.

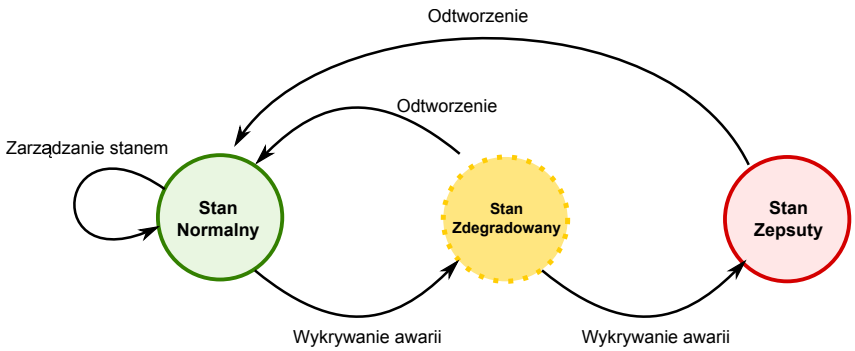
Samonaprawa

Samonaprawa jest cechą systemu który potrafi samodzielnie odzyskać sprawność po wystąpieniu awarii (ang. failure). System taki powinien również być wstanie wykryć gdy jego działanie jest nieprawidłowe [1]. Naprawa systemu może być wykonano samodzielnie (autonomicznie) lub też może wymagać interwencji użytkownika.

Samonaprawa [3] jest również rozpatrywana w kontekście autonomicznych systemów samo-zarządzających się. W takich systemach człowiek odgrywa nową rolę – nie kontroluje on bezpośrednio systemu. Skupia się on na dostarczaniu reguł dzięki którym procesy będą wstanie działać możliwie samodzielnie.

Kluczowym zagadnieniem do zdefiniowania podczas rozpatrywania systemów samo-naprawiających się jest odpowiedzenie na pytanie czy system samonaprawiający się wymaga interakcji użytkownika czy też nie. W rozprawie przyjęto, że samonaprawa nie może wymuszać na użytkowniku podjęcia jakichkolwiek działań. Aby było to możliwe, system taki musi również posiadać cechy autonomiczne.

Rysunek 1 przedstawia diagram stanu systemu samonaprawiającego się [1].



Rysunek 1: State diagram of the self-healing system

Podczas omawiania systemu samonaprawiającego się można wyróżnić czy stany w jakich może znajdować się taki system. Najbardziej pożądanym stanem jest stan normalny, określany również jako zdrowy (ang. healthy). Oznacza on, że system funkcjonuje prawidłowo i wypełnia powierzone mu zadania. Jednakowoż system powinien co pewien dokonywać sprawdzenia swojego własnego stanu. Jest wiele różnych sposobów w jakich takie sprawdzenie może się odbyć ([4, 5, 6]). Jeżeli stan systemu nie można zakwalifikować jako normalny oznacza to, że wykryliśmy awarię.

W przypadku zaistnienia awarii system jest oznaczany jako zepsuty (ang. broken). Samonaprawa powinna pozwolić systemowi odzyskać sprawność – czyli uleczyć się. Jedną z głównych strategii używanych w takich przypadkach polega na redundancji komponentów. Dzięki temu w przypadku awarii jednego z komponentów, inne komponenty mogą przejąć jego funkcje. W takim wypadku dopuszczalny jest spadek wydajności systemu.

Jednym z problemów utrudniającym wykrycie awarii komponentu w rozproszonych środowiskach jest tkz. problem bizantyjskich generałów. Do jego rozwiązania zazwyczaj stosuje się metodę głosowania.

System może również znajdować się w stanie „pomiędzy" stanem poprawnym a zepsutym. Stan taki określa się jako zdegradowany – w tym przypadku system działa ale część jego usług może zostać ograniczona. Jeżeli nie zostaną podjęte odpowiednie kroki zaradcze, to system może uleć awarii. Jako przykład można rozważyć aplikacje której użycie pamięci wynosi 99%. Jest dość prawdopodobne, że aplikacja taka dość szybko ulegnie awarii jeżeli nie zostaną podjęte kroki zmierzające do zwolnienia pamięcią.

1.1 Niezawodność

Niezawodność w kontekście oprogramowania jest pewną cechą statystyczną którą można zdefiniować jako prawdopodobieństwo, że system nie ulegnie awarii w pewnym zdefiniowanym czasie. Może ona zostać przedstawiona za pomocą poniższej funkcji:

$$R(t) = Pr\{T > t\} = \int_t^\infty f(x)dx \text{ [7]}$$

gdzie

$$R(t) = \text{niezawodność}$$
$$T = \text{czas pracy bez awarii}$$
$$t = \text{wymagany czas pracy bez awarii}$$
$$f(x) = \text{funkcja gęstości prawdopodobieństwa wystąpienia awarii}$$

Do oceny niezawodności wykorzystuję się również dwie poniższe metryki:

- Średni Czas Pomiędzy Awariami (ang. MTBF) – długość czasu pomiędzy awariami systemu,
- Wskaźnik awaryjności (ang. Failure Rate) – częstotliwość występowania awarii w systemie.

Ocena niezawodności jest szczególnie przydatna w procesie tworzenia oprogramowania. Może być użyta jako metryka oceny bieżącego stanu oprogramowania i służyć do śledzenia postępów w trakcie różnych faz testowania (np. testów jednostkowych, akceptacyjnych, integracyjnych, wydajnościowych).

1.2 Cele i wkład w badania

Głównym celem rozprawy było zweryfikowanie tezy pracy:

Dzięki wykorzystaniu technik samonaprawy możliwe jest skonstruowanie niezawodnego i wysoko dostępnego systemu monitorującego wykorzystywanego do rozwoju i utrzymywania aplikacji wysokiej dostępności.

W celu zweryfikowania tej tezy został stworzony nowy system monitorujący. Podstawowe cechy systemu to:

- *samonaprawa* – pozwoli ona spełnić wymagania stawiane przed systemami o wysokiej dostępności. System nie powinien dopuścić do utraty danych zebranych w ramach sesji monitorującej.
- *rozproszona, luźno powiązana architektura* – projekt systemu powinien być oparty na rozproszonej architekturze. System powinien umożliwiać monitorowanie rozproszonych aplikacji.
- *autonomia* – powinna istnieć możliwość definiowania reguł i akcji które pozwolą na zautomatyzowanie procesu monitorowania i naprawy
- *zdolność do integracji z systemem monitorowanym* – model systemu powinien umożliwiać na integracje z aplikacją która jest monitorowana w celu jej ewentualnej naprawy.

AgeMon [14] jest systemem zaprojektowanym do zbadania poprawności tezy. W kolejnych częściach tego referatu zostaną zaprezentowane własności systemu. Opis systemu rozpoczyna się od wprowadzenia do architektury oraz implementacji. Następnie omówione zostaną koncepcje i założenia samonaprawy oraz wysokiej dostępności istniejące w systemie. Na zakończenie przedstawione zostaną wyniki testów oraz zostaną podsumowane podane wcześniej informacje.

2 Architektura i Implementacja Systemu

System AgeMon jest oparty na rozproszonej, agentowej architekturze. Rozproszenie systemu pozwala na większą elastyczność w dostosowywaniu się do złożonych aplikacji. Ponieważ aplikacje takie są często rozproszone takie podejście wydaje się uzasadnione.

Rozproszone systemy monitorujące zazwyczaj zbudowane są z następujących elementów: sensorów monitorujących oraz interfejsu użytkownika. Mogą one zostać rozszerzone poprzez dodatkowe komponenty takie jak baza danych lub podsystem reguł wspomagający podejmowanie decyzji w oparciu o rezultaty monitorowania.

Jedną z możliwych implementacji architektury rozproszonej jest system wykorzystujący agenty [9]. Podejście takie niesie wiele korzyści. Przykładowo, skalowalność systemów wieloagentowych (ang. multi-agent system, MAS) jest realizowana w bardzo naturalny – zazwyczaj wystarczy dołożyć kolejne instancje agenta.

Dodatkowo, jedną z typowych własności systemów MAS jest wysoka dostępność i niezawodność. Pojedynczy komponent ulegający awarii nie powinien mieć znaczącego wpływu na resztę systemu. Niezawodność systemu jest osiągana poprzez współpracę wszystkich agentów.

Głównym celem systemu AgeMon nie jest dostarczenie w pełni autonomicznych i inteligentnych agentów (np. uczących się i wnioskujących). Pierwsza implementacja systemu oparta jest na agentach z ograniczoną autonomią. Niezależnie od tego, podczas implementacji systemu starano się rozwiązać problemy często występujące w systemach agentowych takie jak komunikacja pomiędzy agentami, współpraca, dekompozycja złożonych problemów. Agenty są rozproszone pomiędzy różnymi systemami komputerowymi. System AgeMon można określić jako rozproszony system wieloagentowy.

2.1 Architektura rozproszonego i samonaprawiającego się systemu monitorującego

Techniki użyteczne do budowy systemów wysokiej dostępności powinny zostać wykorzystane już podczas wczesnych faz projektowania systemu. Stanowią one podstawę do budowy systemu samonaprawiającego. Kluczowe elementy to:

- redundancja komponentów
- wysoko dostępna warstwa komunikacji

Redundancja komponentów jest podstawowym sposobem na zapewnienie wysokiej dostępności. Jest ona zazwyczaj realizowana poprzez fizyczne lub programowe zduplikowanie komponentów lub zasobów w systemie. To stosunkowo proste rozwiązanie jest stosowane w wielu systemach. Niestety ma ono podstawową wadę – nie jest skalowalne. Proponowane rozwiązanie tego problemu zostanie przedstawione w dalszej części tego autoreferatu.

Sama redundancja komponentów nie jest wystarczająca – co z tego, że komponenty będą zduplikowane jeśli nie będą w stanie się ze sobą komunikować. Aby nie dopuścić do takiej sytuacji warstwa komunikacyjna powinna charakteryzować się wysoką dostępnością.

Proponowany model systemu powinien mieć na uwadze powyższe wyzwania. Przedstawiony projekt systemu skupia się na własnościach samonaprawy m. in. poprzez dynamiczną alokację i wykorzystanie zasobów w zależności od stanu systemu. Redundancja komponentów powinna być możliwa do zastosowania, jednak należy próbować stosować inne techniki w przypadku gdy są one wydajniejsze lub mnie zasobochłonne.

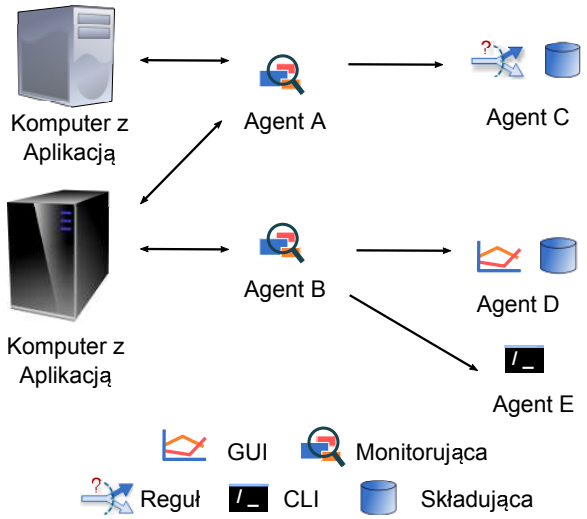
Kluczową koncepcją systemu AgeMon są *role*. Są one używane do grupowania funkcjonalności realizowanej przez poszczególne agenty. Przykładowo rola odpowiedzialna za składowanie danych (ang. persistence agent) zawiera wszystkie funkcjonalności związane z obsługą bazy danych. Dzięki takiemu podejściu możliwe będzie uproszczenie implementacji oraz dopasowanie technik samonaprawy do konkretnych ról.

Wszystkie agenty w systemie mogą pełnić jedną lub wiele ról. W systemie przewidziano 5 ról:

- *Rola monitorująca* – odpowiedzialna za pomiar i odczyt danych w ramach sesji monitorowania
- *Rola składująca dane* – odpowiedzialna za zapis danych do np. bazy danych
- *Rola reguł* – potrafi dynamicznie reagować na zmiany w systemie monitorującym w monitorowanym
- *Rola graficznego interfejsu użytkownika* – pozwala użytkownikowi na interakcje z systemem, definicję sesji monitorowania, oglądanie rezultatów itp.
- *Rola interfejsu linii poleceń* – udostępnia zaawansowane funkcje systemowe poprzez tekstową linię poleceń

Każda z ról może zostać dynamicznie włączona lub wyłączona w zależności od zmian zachodzących w systemie. Jest to ważny aspekt pracy systemu – w systemie nie jest wymagana pełna redundancja ról gdyż system uruchomi nową rolę tylko wtedy gdy jest ona potrzebna.

Przykładowy diagram wdrożenia (ang. deployment diagram) jest przedstawiony na Rysunku 2.



Rysunek 2: System AgeMon – przykład użycia

W przedstawionym przykładzie monitorowane są dwie aplikacje. Strzałki na rysunku przedstawiają przepływ danych reprezentujących zmierzone parametry nazywanych również wartościami monitorowania (ang. monitoring values). W systemie obecne są dwa agenty z włączoną rolą monitorowania – A i B. Agent B monitoruje jedną aplikację, podczas gdy drugi agent monitoruje dwie aplikację w tym samym czasie. Dane przesyłane są do kolejnych agentów. W tym konkretnym przykładzie agent C pełni dwie role – reguł oraz składowania danych. Zapisuje dane do bazy danych oraz potrafi zareagować w przypadku gdy zmierzone wartości przekroczą zdefiniowany wcześniej próg. Agent D potrafi składować dane oraz udostępnia użytkownikowi graficzny interfejs. Ostatni z agentów (E) będzie wykorzystywany przez administratora systemu.

Powyższy, dość prosty przykład ilustruje kilka kluczowych aspektów systemu AgeMon. Jeden agent może wykonywać wiele ról. Z drugiej strony ta sama rola może być wykonywana przez kilku agentów. Agent monitorujący może wysyłać dane do jednego lub wielu agentów. Każda z ról jest zostanie szczegółowo opisana poniżej.

2.2 Role monitorująca

Rola monitorująca (ang. Monitoring Role) jest używana do pomiaru, odczytu pewnej własności z aplikacji (systemu). Agenty te działają jako sensory – co określony czas odpytują system monitorowany o wartość i przesyłają ją do systemu monitorującego. Agenty wspierają dwa typy pomiarów – pomiar parametrów Systemu Operacyjnago oraz pomiar parametrów aplikacji.

W świecie aplikacji napisanych w języku Java powszechnie przyjętą technologią wykorzystywaną do monitorowania i zarządzania jest Java Management Extensions (JMX) [12, 10, 11]. Technologia ta może być również wykorzystywana np. do monitorowania systemu operacyjnego lub sieci. AgeMon posiada wbudowaną obsługę JMX dzięki czemu każda własność udostępniona przez technologię JMX może być monitorowana.

Użytkownik może wybrać co chciałby monitorować poprzez graficzny interfejs użytkownika. Zmierzone wartości będą przesłane do poszczególnych agentów w zdefiniowanych interwałach czasowych. Agent potrafi również buforować dane – może je przesyłać po kilka w jednej paczce.

W przypadku gdy nie jest możliwe skontaktowanie się z docelowym agentem następuje uruchomienie algorytmu elekcji służącego do wyboru agenta zastępczego (ang. failover).

Agent monitorujący nie służy jednak tylko do monitorowania. Dzięki niemu możliwa jest również zaawansowana integracja z monitorowaną aplikacją np. w celu jej dynamicznej naprawy. Ta własność

systemu zostanie opisana w kolejnych rozdziałach.

3 Rola GUI

Rola GUI (ang. GUI Role) jest używana do prezentacji użytkownikowi danych zebranych podczas monitorowania. Służy ona również do wykonywania zadań administracyjnych – dzięki niej użytkownik może np. zdefiniować i wdrożyć określone zestawy reguł.

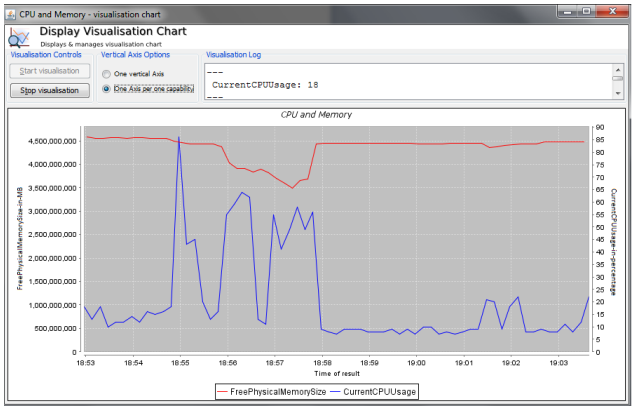
Po uruchomieniu użytkownikowi zostanie wyświetlony graficzny interfejs którego centralnym elementem jest graf agentów. Jest on skierowanym grafem w którym każdy wierzchołek reprezentuje agenta w systemie a krawędź kierunek przepływu danych pomiędzy poszczególnymi agentami. Użytkownik, klikając bezpośrednio na grafie, ma dostęp do wielu funkcjonalności – np. może wyłączyć agenta, rozpocząć monitorowanie, uruchomić wizualizacje.

Rola GUI pozwala użytkownikowi na wybranie własności którą chce monitorować, zdefiniowanie nazwy sesji monitorowania oraz wybranie interwału co jaki dane będą zbierane.

Za pomocą komponentu reguł istnieje możliwość stworzenia i zarządzania regułami. Opcja ta dostępna jest jedynie po wybraniu agenta reguł.

Zarządzanie regułą składującą dane jest realizowanie za pomocą osobnego komponentu. Możliwe jest np. przejrzenie danych zapisanych do bazy danych.

Kolejny komponent udostępniany przez tę rolę odpowiada cza zarządzanie wizualizacjami. Użytkownik może zobaczyć listę aktywnych sesji oraz utworzyć do nich wizualizację. Przykładowa wizualizacja zaprezentowana jest na Rysunku 3.



Rysunek 3: Przykładowa wizualizacja – dwa pomiary przedstawione na jednym wykresie

Przedstawiony zrzut ekranu obrazuje możliwość prezentacji rezultatów z dwóch sesji monitorowania na jednym ekranie. W powyższym przykładzie monitorowane było użycie procesora oraz wolna pamięć. Z tego powodu zostały wyświetlone dwie pionowe osie. Komponent ten posiada bardziej zaawansowane opcje takie dynamiczne skalowanie, zbliżanie i oddalanie, istnieje również możliwość drukowania.

3.1 Rola reguł

Rola reguł (ang. Rule Role) otwiera system na automatyzację. Jest to bardzo ważna własność systemu, szczególnie w złożonych, rozproszonych środowiskach. W takim przypadku użytkownik nie będzie w stanie wykryć wszystkich problemów samodzielnie.

Reguły w systemie są używane do identyfikacji zmiany stanu zarówno w systemie monitorującym jak i w systemie monitorowanym. Przykładowo, możliwe jest wykrycie nowego agenta podłączonego do systemu; możliwe jest również wykrycie faktu że użycie procesora wynosi więcej niż 70%.

W celu zdefiniowania możliwości reakcji na zmiany reguły są zawsze tworzone w połączeniu z akcjami. Akcja jest wykonywana przez system w chwili gdy warunki zdefiniowane w regule są spełnione.

AgeMon posiada dwie predefiniowane reguły – oparte na wartości wyrażenia oraz oparte na stanie systemu. Pierwsza z reguł umożliwia tworzenie warunku nałożonego na monitorowaną własność systemu – np. zużycie procesora większe niż 90%. Możliwe jest tworzenie warunków z wykorzystaniem języka Java Script.

Drugi typ reguł pozwala reagować na zmiany w samym systemie monitorującym. Możliwe jest zdefiniowanie reguły uruchamiającej akcje w przypadku gdy do systemu przyłączył się (lub odłączył się) nowy agent.

W systemie predefiniowane jest pięć różnych akcji. Najprostszą jest Akcja Konsoli polegająca na zalogowaniu zdefiniowanej wcześniej wiadomości na konsolę. Kolejną akcją jest Akcja Email polegająca na wysłaniu wiadomości na podany adres email.

Bardziej zaawansowana akacja polega na wykonaniu dowolnego, zewnętrznego programu. Użytkownik może zdefiniować podać ścieżkę pod którą znajduje się program/skrypt który zostanie wykonany.

Kolejne dwie akcję pozwalają na bezpośrednią interakcje z systemem monitorowanym. Pozwalają na uruchomienie zdefiniowanego kodu (np. metody) z wnętrza systemu monitorowanego. Możliwe jest wywołanie metody z klasy typu MBean lub też dowolnej statycznej metody z dowolnej klasy.

W najprostszej wersji reguły są uruchamiane na pojedynczym agencie reguł. Jednakże takie podejście nie gwarantuje wysokiej dostępności – w tym celu stworzono kooperatywny tryb uruchamiania reguł. Reguły tak stworzone mogą być wykonywane na wielu agentach równocześnie, natomiast wykonanie akcji jest zsynchronizowane i jest wykonywane tylko przez jednego agenta.

3.2 Rola składowania danych

Rola ta służy do zapisu danych otrzymanych z sesji monitorowania (ang. monitoring session) do bazy danych. Domyślnie, reguła ta korzysta z pamięciowej bazy danych (ang. in-memory database). Dzięki temu możliwe jest uruchomienie tej roli nie posiadając żadnej zewnętrznej bazy danych. Oczywiście, w przypadku gdy to konieczne możliwe jest skorzystanie z dowolnej bazy danych wspieranej przez JDBC.

3.3 CLI Role

Rola wiersza poleceń (ang. Command Line Role) jest drugą rolą umożliwiającą użytkownikowi interakcje z systemem monitorującym. Jest ona rolą uzupełniającą rolę GUI. Udostępnia ona dodatkowe funkcjonalności niedostępne poprzez rolę GUI. Przykładowo, dzięki tej roli możliwe jest uruchomienie dowolnego zapytania SQL na bazie danych obsługiwanej przez rolę składowania danych.

3.4 Warstwa komunikacji

Komunikacja w systemie AgeMon oparta jest o wymianę wiadomości pomiędzy agentami. Warstwa komunikacji dostarcza wielu ważnych funkcjonalności. Możliwa jest komunikacja bezpośrednia pomiędzy poszczególnymi agentami (punkt-punkt) oraz równocześnie pomiędzy wieloma agentami (punkt-do-wielu). Warstwa ta również umożliwia przyłączenie nowych agentów bez żadnej dodatkowej konfiguracji. Wspiera również automatyczne wykrywanie faktu dołączenia / odłączenia agentów.

Kolejne poziomy warstwy komunikacji dostarczają kolejnych abstrakcji, dzięki czemu możliwe jest traktowanie grupy agentów w sposób w pełni obiektowy.

4 Samonaprawa i wysoka dostępność

System AgeMon został zaprojektowany jako wysoko samonaprawiający się, wysoko dostępny system. Poniżej przedstawiona została lista najważniejszych funkcjonalności systemu które wspierają samona-

prawę.

4.1 Automatyczne wykrywanie

Automatyczne wykrywanie wszystkich agentów w danej grupie jest elementem wspierającym wysoką dostępność systemu. System nie posiada pojedynczego miejsca do którego rejestrowane są poszczególne agenty. Automatyczne wykrywanie agentów może być również postrzegane przez pryzmat samonaprawy. Dzięki niemu agent który odłączył się z grupy może w sposób automatyczny przyłączyć się do niej ponownie.

4.2 Niezawodne protokoły transportowe

Komunikacja pomiędzy agentami wykorzystuje niezawodne algorytmy. Oznacza to, że warstwa transportowa poinformuje wyższe warstwy aplikacji o fakcie dostarczenia *lub* niedostarczenia wiadomości. Prawidłowe dostarczenie wiadomości wymusza na odbiorcy wysłanie dodatkowej wiadomości potwierdzającej fakt odbioru. Zapewniana jest prawidłowa kolejność dostarczanych wiadomości jak również ich spójność.

4.3 Odporność na błędy sieci

AgeMon potrafi działać w przypadku problemów z siecią. Pierwszą linią obrony jest zdolność do buforowania wiadomości po stronie agenta. Jeżeli wartości monitorowania nie mogą być wysłane do agenta docelowego to zostaną one zapisane po stronie agenta źródłowego i zostanie zapoczątkowany drugi krok radzenia sobie z problemem - nastąpi wybór agenta zastępczego.

4.4 Brak pojedynczego miejsca którego awaria spowoduje awarię całego systemu

Miejsca takie są zazwyczaj najbardziej kłopotliwe i wykluczają poszczególne systemy z klasy systemów wysokiej dostępności. Można tutaj wyróżnić dwa podejścia radzenia sobie z problemem:

- unikanie problemu poprzez zastosowanie prawidłowych decyzji projektowych. Przykładowo – AgeMon jest w pełni zdecentralizowany. Agenty są homogeniczne, ale potrafią wykonywać różne role. Role mogą być dynamicznie startowanie w celu radzenia sobie z problemami w systemie
- dodatkowa redundancja komponentów. W przypadku gdy nie jest możliwe uniknięcia problemu można stosować dodatkową redundancję.

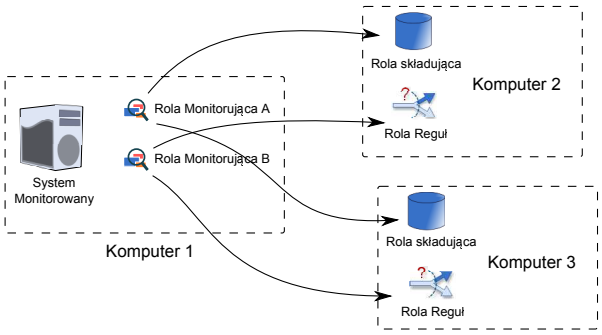
4.5 Redundancja Ról

W przypadku gdy nie jest możliwe uniknięcie problemu, jednym z rozwiązań jest dodanie redundancji. W przypadku systemu AgeMon wszystkie agenty są podobne – jedyna różnica polega na roli którą pełni agent. Istnieje kilka metod osiągnięcia redundancji w systemie AgeMon. Pierwsza stosunkowo prosta, jest pokazana na Rysunku 4.

W tym przykładzie, AgeMon jest użyty do monitorowania Systemu Operacyjnego. Wszystkie agenty są zduplikowane i są uruchomione (poza agentami monitorującymi) na osobnych fizycznych maszynach.

AgeMon wspiera tego typu konfigurację, ale ma ona kilka wad. Złożoność wdrożenia jest dość wysoka i wymaga wielu nadmiarowych połączeń. Wymagana jest również większa przepustowość sieci. W związku z tym taki rodzaj konfiguracji nie jest zalecany w większości przypadków.

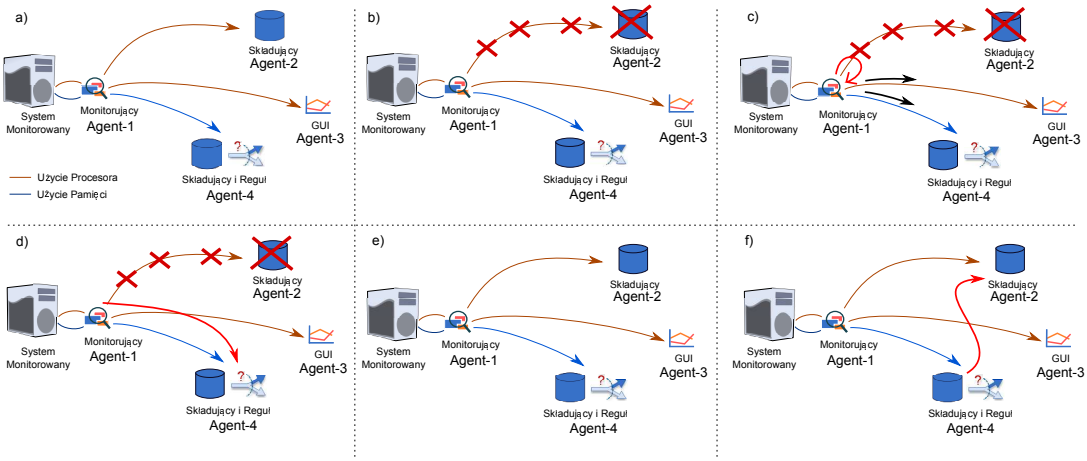
AgeMon dostarcza dużo bardziej zaawansowane mechanizmy. Przykładowo może automatycznie reagować na problemy poprzez elekcję zastępczych agentów. Proces ten został opisany w kolejnym rozdziale.



Rysunek 4: Redundancja kluczowych komponentów w systemie AgeMon

4.6 Elekcja agentów zastępczych

Wybór agentów zastępczych jest kluczową funkcjonalnością w systemie zapewniającą wysoką dostępność. Proces elekcji jest używany do wyboru agenta, który będzie zapisywał dane do bazy danych w przypadku gdy oryginalnie desygnowany do tego celu agent będzie niedostępny. Przykład takiego scenariusza jest przedstawiony na Rysunku 5.



Rysunek 5: Algorytm elekcji

W podanym przykładzie (Rysunek 5.a) użytkownik pragnie monitorować dwie własności systemu: użycie procesora i pamięci. Po pewnym okresie czasu agent *Agent 2* zostaje wyłączony (Rysunek. 5.b). System wykrywa tę sytuację i rozpoczyna procedurę naprawczą która składa się z dwóch kroków.

W pierwszym kroku (Rysunek 5.c.) agent monitorujący zakłada bufor w którym będzie składował dane dopóki problem nie zostanie rozwiązany. Dzięki temu żadne dane nie zostaną utracone.

W kolejnym kroku podjęta zostaje próba znalezienia agenta zastępczego który będzie w stanie składować dane. W tym celu zostaje uruchomiony algorytm elekcji (można definiować różne algorytmy elekcji). Zostanie również utworzone zastępcze połączenie pomiędzy agentem monitorującym a wybranym agentem (czerwona krawędź na Rysunku 5.d).

W omawianym przykładzie agent reguł został skonfigurowany aby monitorował zmiany zachodzące w systemie. W chwili wykrycia problemu z Agentem 2 zostanie wysłany email do Administratora systemu. Po pewnym czasie Administrator uporał się z problemem (zabrakło miejsca na dysku) i ponownie uruchomił agenta.

W ostatnim kroku wszystkie dane przechowywane przez agenta zastępczego zostają przetransferowane do oryginalnego agenta (Fig. 5.f). Na koniec procesu agent ten będzie zawierał wszystkie dane płynące z procesu monitorowania.

Wybór agenta zastępczego jest szczególnie ważnym procesem w systemie AgeMon. Dzięki algorytmom elekcji system wykazuje cechy systemu wysokiej dostępności.

4.7 Zaawansowane reguły w kontekście samonaprawy

Reguły potrafią uruchomić zdefiniowaną wcześniej akcje w reakcji na zmiany zachodzące w systemie monitorującym – w szczególności w przypadku odłączenia oraz dołączenia agentów. Te stosunkowo proste reguły mogą być wykorzystane do zdefiniowania zaawansowanych strategii samonaprawy. Przykładowo, w przypadku nieoczekiwanego odłączenia agenta można uruchomić skrypt analizujący plik logu z komunikatami pochodzącymi od tego agenta. W zależności od wyniku analizy można np. próbować ponownie uruchomić agenta (poprzez wykonanie zdalnego polecenia na serwerze).

To jest tylko jeden z możliwych przykładów – dzięki temu, że system potrafi uruchomić w zasadzie dowolny program, można wyobrazić sobie wiele możliwości kończących się samonaprawą systemu, zaczynając od rozwiązywania problemów połączenia pomiędzy agentami a kończąc na usuwaniu plików (np. logów) w celu odzyskania miejsca na dysku. Każdy z tych scenariuszy może zostać uruchomiony w systemie AgeMon.

4.8 Samonaprawa i Wysoka Dostępność – podsumowanie

Dzięki wykorzystaniu ról pełnionych przez agenty możliwe jest zdefiniowanie najodpowiedniejszych algorytmów i technik zwiększających dostępność i dostosowanie ich do konkretnych ról. Przykładowo – można wykorzystać redundancję w przypadku agenta monitorującego. Podejście takie można również wykorzystać w przypadku roli wykorzystywanej do zapisu danych, jednakże nie będzie to efektywne rozwiązanie z punktu widzenia zużycia zasobów dyskowych. Podobny przypadek zachodzi u agenta pełniącego rolę reguł - sama reguła może zostać sprawdzona w tym samym momencie przez wiele agentów, ale akcja powinna zostać wykonana przez pojedynczego agenta.

Z tych powodów, każda z ról powinna posiadać własne metody gwarantujące osiągnięcie wysokiej dostępności:

- *Rola Monitorująca* – w przypadku tej roli najlepsze okazuje się połączenie dwóch strategii – redundancji oraz samonaprawy. Redundacja zmieszy prawdopodobieństwo wystąpienia awarii. Nawet jeżeli ona nastąpi, dzięki zdefiniowaniu akcji naprawczej możliwe będzie poradzenie sobie z problem (np. poprzez zrestartowanie agentów).
- *Rola Reguł* – reguły używające podejścia kooperatywnego powinny być używane w przypadku środowiska wysoko dostępnego. Reguły takie mogą być uruchamiane równolegle. Jednakże, dzięki zastosowanym w systemie technikom możliwe jest zagwarantowanie, że akcja zostanie wykonana tylko przez jednego z agentów.
- *Rola składowanych danych* – podejście wykorzystujące wybór zapasowego agenta wydaje się najlepsze w tym przypadku. W przypadku awarii system AgeMon wybierze agenta który będzie zapisał dane w zastępstwie oryginalnego agenta.
- *Rola GUI* and *Rola wiersza poleceń* – w przypadku tych ról nie jest konieczne posiadanie oddzielnych mechanizmów naprawczych. Nie służą one do składowania krytycznych danych oraz nie mają bezpośredniego wpływu na proces monitorowania.

Dzięki podejściu z wykorzystaniem ról możliwe jest wykorzystanie różnych algorytmów wspierających wysoką dostępność systemu. Jak zaznaczono, specyfika każdej z ról jest inna i wymaga innego podejścia w celu naprawy ewentualnych problemów.

5 Testy systemu

W pierwszej części tego rozdziału zostaną przedstawione rezultaty testów wydajnościowych. Ten typ testów jest wykorzystywany do zweryfikowania stabilności i responsywności aplikacji poddanej różnym obciążeniom. Testy wydajnościowe dają również obraz skalowalności i niezawodności systemu. Kolejny rodzaj testów przeprowadzonych na systemie AgeMon służył ocenie czasu reakcji Systemu na zmiany w systemie monitorowanym i monitorującym.

Ostatni rodzaj testów służył ocenie mechanizmów samonaprawy oraz zapewniających wysoką dostępność systemu. Sprawdzono w jaki sposób na system wpłynie fakt awarii poszczególnych jego komponentów. Ten rodzaj testów jest nazywany testami destrukcyjnymi.

5.1 Testy wydajnościowe

W celu przetestowania wydajności systemu AgeMon, zostało stworzone przykładowe środowisko testowe. W scenariuszu zostało wykorzystanych wiele agentów monitorujących (ich ilość zależy od konkretnego przypadku) zainstalowanych na różnych maszynach. Zbierały on informacje na temat użycia procesora. Dane te były następnie wysyłane do Agenta Monitorującego – przedział czasu (nazywany również opóźnieniem) był różny dla kolejnych testów.

Test polegał na zmierzeniu całkowitego czasu trwania scenariusza i porównaniu go z czasem oczekiwanym. Jeżeli różnica była zbyt duża – nie jest wytłumaczalna przez opóźnienia związane z transmisją przez sieć, oznacza to że osiągnęliśmy granice wydajności agenta.

Następująca konfiguracja została użyta we wszystkich scenariuszach testowych (komputery były połaczone siecią Gigabit Ethernet):

- 5 × SunOS 5.10, 48GB pamięci RAM, 2 × Intel® Xeon® CPU X5650 @ 2.67GHz (zwany dalej jako Sun)
- 5 × Linux RedHat 5.5 (Tikanga), 30GB pamięci RAM, 24 × Intel® Xeon® CPU X5650 @ 2.67GHz (zwany dalej jako Linux)
- 1 × Windows 7 Enterprise, 16GB pamięci RAM, 1 × Intel® Core™CPU i5-3527U @ 2.30 GHz (4 rdzenie) (zwany dalej jako Win7)

Podczas testów zostały zmierzone podstawowe metryki takie jak użycie procesora, pamięci, ilość uruchomionych wątków, charakterystyka pracy GC. Monitoring odbywał się za pomocą drugiej instancji systemu AgeMon używającej innego zestawu portów.

Przeprowadzono 12 testów przy różnej ilości agentów w celu zbadania różnicy w czasie pomiędzy „przetwarzaniem idealnym" a obserwowaną wydajnością. Na potrzeby pracy założono, że wydajność gorsza o więcej niż 20% jest nieakceptowalna.

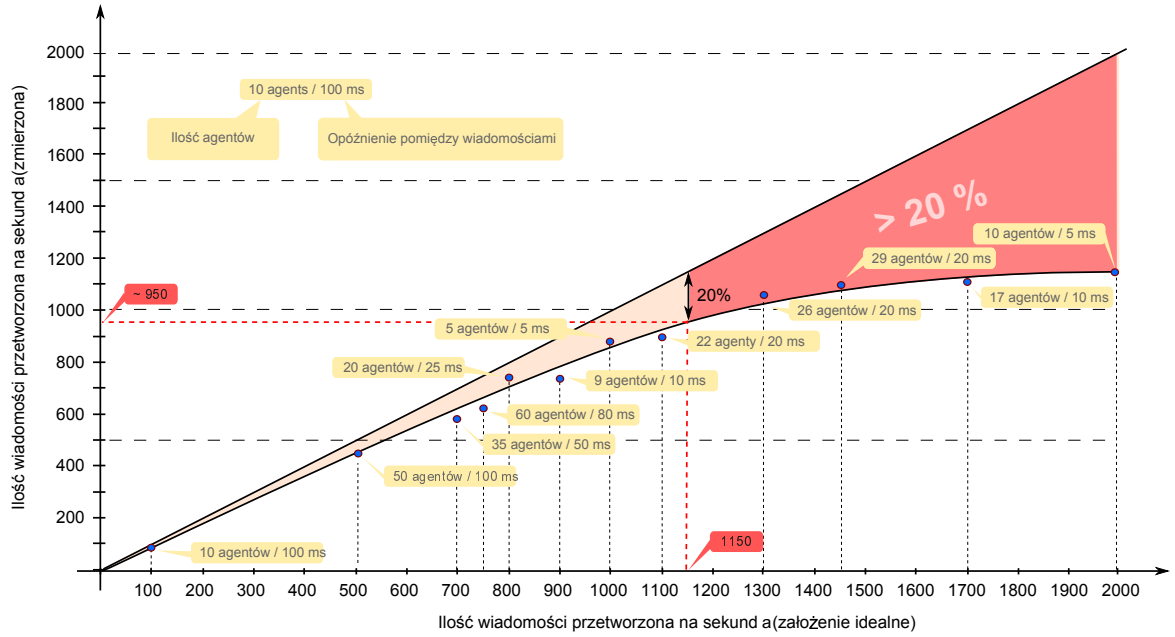
„Przetwarzanie idealne" jest obliczone wyłącznie na podstawie ilości wiadomości i sumie czasu co jaki wysyłano wiadomości. Nie zawiera on zatem czasu potrzebnego na przesłanie wiadomości przez sieć i faktycznie przetworzenie jej przez system. W wysoko wydajnym systemie czas ten może być zbliżony ale nigdy nie osiągnie wartości idealnej.

Rysunek 6 przedstawia resultaty testów. Na rysunku można zaobserwować różnicę pomiędzy stanem idealnym a zaobserwowanymi rezultatami. Można na jego podstawie wyciągnąć następujące wnioski:

- wydajność systemu jest bezpośrednio związana z ilością wiadomości do przetworzenia. Olbrzymia ilość wiadomości do przetworzenia powoduje spadek wydajności,
- agent składujący dane potrafi przetworzyć około 950 wiadomości z akceptowalną wydajnością. Powyżej tej liczby spadek wydajności przestaje być akceptowalny

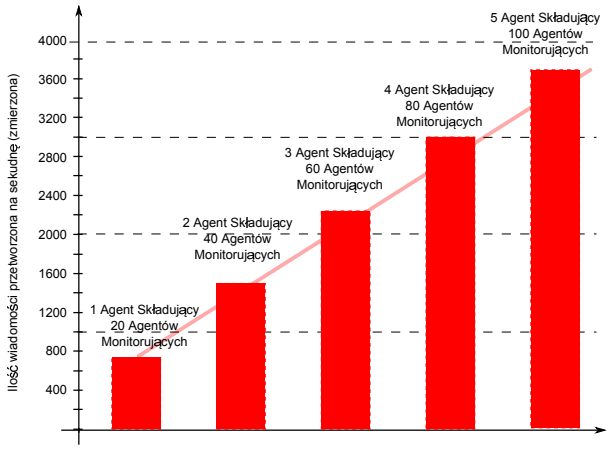
W przypadku użycia systemu do przetwarzania dużej ilości wiadomości (większej niż 1000/s) należy zastosować inne podejście. Można zwiększyć wydajność procesora (poprzez np. zakup nowego). Innym rozwiązaniem jest użycie większej liczby agentów bazodanowych.

W większości przypadków wydaje się, że wydajność ok 1000/s jest w zupełności wystarczająca. W



Rysunek 6: Wydajność agenta składującego dane

pozostałych przypadkach sugerowane jest użycie większej liczby agentów. W celu weryfikacji zachowania systemu w tym przypadku przeprowadzono dodatkowe testy. Do pierwszej fazy testów użyto jednego agenta bazodanowego i 20 agentów monitorujących. Agenty monitorujące przesyłały wiadomości co 25 ms. W każdym kolejnym cyklu zwiększano liczbę agentów. Rezultaty pomiarów wydajności są zaprezentowane na Rysunku 7.



Rysunek 7: Skalowalność w systemie AgeMon

System AgeMon potrafi się skalować horyzontalnie – wąskim gardłem okazuje się być wydajność procesora. W celu zwiększenia wydajności najlepszym rozwiązaniem wydaje się zwiększenie liczby agentów w systemie (nowe agenty powinny być uruchomione na nowych maszynach).

6 Opóźnienia w systemie / czas reakcji

W tej części przedstawiono wyniki dwóch scenariuszy, których celem było zmierzenie opóźnień (ang. latency) w systemie AgeMon. Pierwszy przykład wydaje się szczególnie interesujący – dzięki niemu poznajemy odpowiedź na pytanie „Jak długo czasu zajmuje systemowi podjęcie decyzji w celu reakcji

na zmiany w systemie monitorowanym?". Drugi przykład jest równie ważny – dzięki niemu wiadomo ile czasu zabiera systemowi zapisanie danych z monitorowania.

Średni czas decyzji (Avg_{td}) jest ważną cechą opisującą system AgeMon. Definiuje ona długość czasu potrzebnego na podjęcie decyzji. Czynnik ten opisuje jaka jest maksymalna szybkość zmian w systemie monitorowanym która pozwala na podjęcie prawidłowej akcji.

W przypadku gdy w zmiany w systemie monitorowanym zachodzą częściej niż Avg_{dt} system monitorujący nie będzie w stanie podjąć decyzji która odpowiada aktualnemu stanowi systemu. Podczas testów zmierzony średni czas na podjęcie decyzji wynosi 36 milisekund.

W kolejnym scenariuszu zmierzono czas po jakim dane są zapisane do bazy danych. Test ten został uruchomiony w podobnej konfiguracji do poprzedniego scenariusza. Jedyną różnicą było wykorzystanie agenta reguł zamiast agenta składującego dane. W czasie testu mierzono czas jaki upłynął od pomiaru danej wartości do jej zapisania (zatwierdzenia, ang. commit) w bazie danych.

Testy wykazały, że agent monitorujący potrzebuje 33 milisekundy na wysłanie zmierzonej wartości. Po kolejnych 11 milisekundach dane zostają zapisane w bazie danych. Cały proces (wraz z wysłaniem potwierdzenia) zajmuje 47 miliseconds.

Podsumowując, system potrzebuje 36 milisekund do podjęcia decyzji i 47 milisekund na zapis danych otrzymanych z monitorowania do bazy danych. W przypadku gdy wartości opóźnień mogą mieć wpływ na sam proces monitorowania zaleca się uprzednie zbadanie wydajności wartsty sieciowej (np. przy użyciu polecenia `PING`).

W całym procesie przesyłania wiadomości najwolniejszy okazał się proces serializacji i deserializacji wykorzystywany przez Javę. W kolejnych wersjach systemu można pokusić się o przyspieszenie tego procesu poprzez dostarczenie własnej implementacji tych algorytmów.

6.1 Testy samonaprawy

Poprzednie testy weryfikowały zachowanie systemu podczas zwykłego lub dużego obciążenia. Oczywiście testy takie są niezwykle istotne, jednakże w celu weryfikacji czy system można traktować jako system wysokiej dostępności należy przeprowadzić dodatkowe pomiary.

W tym celu przeprowadzono testy dwóch rodzajów. W pierwszej kategorii znalazły się *testy destrukcyjne*. W czasie tych testów przeprowadzono symulację awarii różnych komponentów w celu weryfikacji niezawodności systemu. W drugiej grupie znalazły się testy typu *soak*. Używa się ich do sprawdzenia jak zachowuje się system będący pod dużym obciążeniem w długim okresie czasu.

W przypadku pierwszego testu użyto systemu złożonego z 50 agentów. Połowa z nich pełniła role monitorujące a druga połowa była używana do zapisywania danych do bazy danych.

Testy symulowały błąd w agencie składującym dane. Mechanizmy samonaprawy powinny wykryć taką sytuację i podjąć próbę wyboru agenta który będzie zapisywał dane do czasu gdy oryginalny agent nie zostanie naprawiony.

Proces elekcji rozpoczyna się w chwili gdy agent monitorujący nie jest w stanie wysłać danych do docelowego agenta. Dzięki specjalnie zaprojektowanej warstwie komunikacyjnej każdy z agentów zna stan wszystkich agentów w grupie. W tym samym czasie gdy algorytm elekcji próbuje wybrać agenta zastępczego, wszystkie wiadomości są zapisywane w lokalnym buforze. Na zapis każdej z takich wiadomości potrzeba ok 1-2 kB – co oznacza, że agent potrafi przechować do ok pół miliona wiadomości w przypadku gdy na dostępne 1GB pamięci na stercie.

28 milisekund po wykryciu awarii, wiadomość inicjująca proces elekcji zostaje wysłana do wszystkich agentów. Każdy z agentów przetwarza wiadomość i bazując na polityce elekcji wysyła odpowiedź dzięki której będzie możliwe wybranie agenta zastępczego. Agent monitorujący czeka na odpowiedzi od wszystkich pozostałych agentów (nie dłużej niż zdefiniowany wcześniej czas). Po otrzymaniu odpowiedzi, agent wybiera zapasowego agenta i rozgłasza tą informacje do całego systemu. Wszystkie wiadomości z danymi zostają wysłane do wybranego agenta.

Drugi scenariusz testowy skupiał się na badaniu przypadku w którym agent monitorujący ulega awarii (np. wskutek restartu maszyny) i jest konieczne jego ponowne uruchomienie. Scenariusz prezentuje zdolność systemu do wykrycia problemu i elastyczność w definiowaniu reguł dzięki którym można uruchomić dowolny skrypt.

W przypadku tego scenariusza wykorzystano system złożony z 20 agentów. Połowa z nich pełniła funkcje monitorujące, 9 składowało dane a jeden z nich był agentem reguł. W systemie zdefiniowano jedną regułę – uruchomi ona zewnętrzny skrypt w przypadku gdy którykolwiek z agentów zostanie odłączony od systemu. Skrypt ten połączy się ze zdalnym serwerem i podejmie próbę restartowania agenta monitorującego.

Podczas testu agent monitorujący został zatrzymany (przy użyciu polecenia `kill -9`). Po około 100 milisekundach agent reguł wykrył awarie agenta. Po następnych 140 milisekundach agent wykonał zewnętrzny skrypt. Agent monitorujący został ponownie uruchomiony po kolejnych 50 milisekundach. Całkowity czas naprawy agenta wyniósł **295 ms**. Po dodatkowych 170 ms wszystkie agenty w grupie otrzymały informacje o ponownym uruchomieniu agenta.

6.2 Niezawodność

Całkowita ilość wiadomości wysłanych w czasie wszystkich testów była większa niż 40 milionów. Przeprowadzono różne rodzaje testów – testowano system w przypadku zarówno typowego jak i bardzo dużego obciążenia. Dodatkowo przeprowadzono testy destrukcyjne.

Podczas tych wszystkich testów sprawdzano również niezawodność systemu (równocześnie z weryfikacją innych niefunkcjonalnych wymagań takich jak wydajność oraz skalowalność). Rezultaty testów są bardzo obiecujące – nie utracono ani jednej wiadomości z danymi płynącymi z monitorowania systemu. System mógł pracować ze zmniejszoną wydajnością (np. gdy obciążenie było za wysokie) jednakże nie wpłynęło to na jego niezawodność.

Występują tylko dwa przypadki w których pojedyncza wiadomość może zostać utracona. Pierwszy zachodzi wówczas gdy agent monitorujący zdążył wysłać wiadomości przed wstąpieniem awarii jego samego. Drugi przypadek ma miejsce gdy zachodzi awaria agenta składującego dane (po otrzymaniu wiadomości a przed zatwierdzeniem zapisu). Można rozważyć taką implementację komunikacji agent monitorujący – agent składujący dane, która nie dopuści do wystąpienia powyższej sytuacji. Agent monitorujący powinien poczekać na informacje o zatwierdzeniu informacji do bazy danych. W przypadku jej braku powinien: a) rozpocząć retransmisję wiadomości lub b) rozpocząć wybór agenta zastępczego. Minusem takiego rozwiązania jest zdecydowany spadek wydajności systemu. Z tego powodu nie zdecydowano się na implementację tego rozwiązania.

Na szczęście, architektura systemu umożliwia uniknięcie obydwu problemów. Przykładowo, można rozważyć zduplikowanie agentów. Oczywiście takie podejście zaleca się stosować jedynie wówczas gdy zapis wszystkich wiadomości jest niezwykle krytyczny. Przeprowadzone testy wykazały, że scenariusz w którym następuje utrata wiadomości jest niezwykle rzadki.

Podsumowując – w czasie wszystkich testów wysłano ponad 40 milionów wiadomości (10 milionów podczas testów destrukcyjnych i 32 miliony podczas testów typu soak) i nie zaobserwowano utraty wiadomości.

7 Podsumowanie

Głównym celem badań było zbadanie i zaprojektowanie różnych technik i algorytmów które mogą być używane do budowy wysoko dostępnych systemów monitorujących. W pracy opisano różne możliwe podejścia. Skupiono się również na praktycznej weryfikacji wymagań. W tym celu stworzono nowy system monitorujący o nazwie AgeMon. System ten jest oparty o architekturę rozproszoną zbudowaną w oparciu o agenty. Każdy agent może wykonywać różne role w systemie. Rola grupuje podobne funkcjonalności.

Koncepcja ról jest istotna w kontekście tworzenia niezawodnego systemu z elementami samonaprawy ponieważ pozwala ona na zastosowanie różnych taktyk w zależności od konkretnego problemu. Rola składująca dane może zostać zastąpiona poprzez agenta wybranego dynamicznie spośród innych agentów w systemie. Rola reguł potrafi współpracować z innymi agentami w celu dostarczenie niezawodnej metody na wykonywanie akcji. Rola monitorująca może zostać zrestartowania w przypadku awarii.

System nie posiada pojedynczego komponentu, którego awaria może doprowadzić do awarii całego systemu. Komunikacja pomiędzy agentami jest oparta na niezawodnym i szybkim protokole zapewniającym również automatyczne wykrywanie podłączonych agentów. Dzięki temu nie jest konieczne stosowanie dodatkowych serwerów „spotkań" (ang. gossip/rendezvous severs). Każdy z komponentów systemów może posiadać redundancję ale nie jest to warunek konieczny.

System AgeMon jest dowodem koncepcji bardziej ogólnego modelu budowy systemów. Model ten może być stosowany w przypadku budowy innych wysoko dostępnych systemów (nie musi być wykorzystywany jedynie do systemów monitorujących). Koncepcje takie jak role, warstwa komunikacji, abstrakcja agentów mogą być z powodzeniem wykorzystywane do budowy innych systemów.

Mechanizm monitorowania wykorzystany przez system AgeMon jest dość elastyczny. Domyślnie może być on wykorzystywany do monitorowania elementów systemu operacyjnego lub dowolnej aplikacji napisanej w języku Java. W przypadku gdy aplikacja taka wykorzystuje technologie JMX do udostępniania pewnych metryk, są one automatycznie widoczne w systemie AgeMon. W innych przypadkach możliwe jest zbudowanie adaptera udostępniające dane z innego systemu monitorującego.

Ważną częścią systemu monitorującego jest zaprezentowanie danych użytkownikowi. Wizualizacje (wykresy) wbudowane w systemie AgeMon umożliwiają prezentacje danych z jednego lub wielu źródeł na jednym wykresie. Możliwe jest zbliżanie, zmienianie zakresu oraz wydruk wykresów.

System wspiera wiele typów reguł i akcji. AgeMon dostarcza wbudowane reguły takie jak „agent podłączony", „agent odłączony" oraz reguły budowane na podstawie wartości z monitorowanego systemu. Istnieje również możliwość stworzenia własnych reguł. Podobnie – istnieje pewna ilość predefiniowanych akcji podczas gdy kolejne mogą być dostarczane przez użytkownika systemu.

System jest w pełni rozproszony. Istnieje zarówno możliwość uruchomienia wielu agentów na jednej maszynie jak i posiadanie dedykowanych maszyn dla każdego agenta. System nie wymaga żadnej konfiguracji w przypadku działania w ramach tej samej podsieci LAN. Możliwe jest również używanie systemu przez sieć WAN lub Internet. W takim przypadku wymagane jest użycie tunelu lub serwera typu gossip.

Rola składujące dane umożliwia zapis rezultatów do bazy danych. AgeMon dostarcza wbudowaną, pamięciową relacyjną bazę danych (HSQL). Istnieje możliwość użycia innej bazy danych SQL.

Użycie AgeMon jest nieskomplikowane. Nie jest wymagana żadna dodatkowa konfiguracja lub skomplikowana instalacja. System jest dystrybuowany jako jeden plik jar i żadna instalacja nie jest wymagana. W przypadku wykorzystania domyślnej konfiguracji system jest gotowy do użycia od razu po skopiowaniu pliku jar – agenty zostaną automatycznie wykryte i możliwe będzie rozpoczęcie budowy systemu.

Podsumowując, możliwe jest zbudowanie niezawodnego systemu monitorującego a techniki samonaprawy mogą być w tym celu niezwykle pomocne.

7.1 Nowatorskie koncepcje wykorzystane w pracy

System AgeMon jest jednym z pierwszych systemów monitorujących wykorzystujących mechanizmy samonaprawy. Jest również system wysokiej dostępności; nie posiada single-point-of-failure oraz wspiera redundancje wszystkich komponentów.

Model użyty do budowy systemu, który jest oparty o role, może być wykorzystany do innych systemów wspierających samonaprawę. Takie podejście może być wykorzystane również w przypadku systemów nie związanych z monitorowaniem. Techniki samonaprawy mogą być również wykorzystane w przypadku komponentów z innych systemów.

Jednym z najważniejszych osiągnięć pracy była ewaluacja różnych strategii samonaprawy i technik umożliwiających budowę systemu wysokiej dostępności. Różne problemy wymagają różnych rozwiązań, wydaj się, że podejście wykorzystujące role jest tutaj bardzo pomocne.

Dodatkowo, model wymiany informacji (warstwa komunikacji) używany w systmie może być wykorzystany w innych aplikacjach. Wykorzystanie tego modelu nie jest ograniczone do systemów monitorujących, może być one z powodzeniem zastosowany w przypadku innych systemów rozproszonych. Wspiera on automatyczne wykrywanie komponentów, niezawodną i wydajną komunikację.

Literatura

[1] Ghosh, D., Sharman. R., Rao, H. R., Upadhyaya, S. *Self-healing systems – survey and synthesis.* Department of CSE, SUNY, Buffalo, United States, Department of Management Science and Systems, School of Management, SUNY, Buffalo, Decision Support Systems 42 (2007) 2164-2185

[2] Weygant, P. S. *Clusters for High Availability: A Primer of HP Solutions.* ISBN-10: 0130893552 ISBN-13: 9780130893550 2001 Prentice Hall Paper, 336 pp Published 05/07/2001

[3] IBM. *The IBM autonomic computing initiative* `http://www-03.ibm.com/systems/z/os/zos/features/sysmgmt/autonomic/index.html`

[4] George, S., Evans, D., Marchette, S. *A biological programming model for self-healing.* First ACM Workshop on Survivable and Self-Regenerative Systems, 2003

[5] Aldrich, J., Sazawal, V., Chambers, C., Nokin, D. *Architecturecentric programming for adaptive systems.* Proceedings of the First Workshop on Self-Healing Systems, 2002.

[6] Dabrowski, C., Mills, K. L. *Understanding self-healing in servicediscovery systems.* Proceedings of the First Workshop on Self-Healing Systems, 2002.

[7] Goel, A. L. *Software reliability models: Assumptions, limitations, and applicability.* Software Engineering, IEEE Transactions on 12 (1985): 1411-1423.

[8] The Complex Adaptive Systems (CAS) Group. *Multi-Agent System.* 16 February 2011. `http://wiki.cas-group.net/index.php?title=Multi-Agent_System`

[9] The Intelligent Software Agents Lab – Home Page – `http://www.cs.cmu.edu/softagents/intro.html`

[10] Perry, J. S. *Java Management Extensions.* O'Reilly, ISBN 0-596-00245-9.

[11] Fleury, M., Lindfors, J. *JMX: Managing J2EE with Java Management Extensions.* Sams Publishing, ISBN 0-672-32288-9.

[12] Java Management Extensions (JMX) Technology. Home Page `http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`

[13] Funika, W., Godowski, P., Pegiel, P., Krol, D. *Semantic-oriented performance monitoring of distributed applications.* COMPUTING AND INFORMATICS Volume: 31 Issue: 2 Pages: 427-446 Published: 2012. ISSN: 1335-9150

[14] Funika, W., Pegiel, P. *A role-based approach to self-healing in autonomous monitoring systems.* PARALLEL PROCESSING AND APPLIED MATHEMATICS, PART II Book Series: Lecture Notes in Computer Science Volume: 6068 Pages: 125-134 Published: 2010. ISSN: 0302-9743. ISBN: 978-3-642-14402-8