Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Przemysław Strzelczak
Nr albumu: 201068

Wizualizacja systemu rozproszonego po awarii na podstawie migawek w celu tropienia błędów

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Janiny Mincer-Daszkiewicz
Instytut Informatyki

Wrzesień 2006

Oświadczenie kierującego pracą

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

Data Podpis kierującego pracą

Oświadczenie autora pracy

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

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

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

Data Podpis autora pracy

Streszczenie

W pracy poruszam kwestię skutecznego tropienia błędów w systemach rozproszonych. Przez błąd w tym kontekście rozumiem błędy programistyczne, spadek wydajności, czy szeroko pojęte nietypowe sytuacje w działaniu systemu rozproszonego. Przedstawiam również pomysł narzędzia wspierającego programistę w poszukiwaniu takich błędów. Polega on na zapisywaniu migawek z funkcjonowania systemu rozproszonego w czasie jego działania i wygodnej z punktu widzenia użytkownika wizualizacji działania systemu na podstawie zapisanych zrzutów. Pomysł ten wsparty jest rzeczywistą implementacją, którą współtworzyłem w ramach projektu dla zastosowań komercyjnych. Przyjęte rozwiązanie jest analizowane pod kątem użyteczności, sposobu działania, problemów implementacyjnych i możliwości rozszerzeń.

Słowa kluczowe

system rozproszony, wizualizacja, usuwanie błędów, wykrywanie anomalii

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.2 Software engineering
D.2.5 Testing and debugging
D.2.5.3 Distributed debugging

Tytuł pracy w języku angielskim

Post mortem distributed system visualization for debugging purposes based on its activity snapshots

Spis treści

Spis rysunków

Rozdział 1

Wprowadzenie

Skuteczne tropienie błędów w systemach rozproszonych od zawsze było uważane za zadanie ambitne i skomplikowane. Ten sam problem postawiony dla systemów jednozadaniowych (sekwencyjnych) wydaje się w miarę prosty do rozwiązania, gdyż cykliczne tropienie błędów, polegające na stopniowym zawężaniu możliwego momentu wystąpienia problemu, powoduje, że jego znalezienie jest jedynie kwestią czasu.

W przypadku wyszukiwania błędów w systemach rozproszonych napotyka się na większe trudności. Wynikają one ze złożonej natury samych systemów rozproszonych, w których pojawia się wielozadaniowość, niejednorodność węzłów, niedetereminizm pojedynczych węzłów, czy kwestie związane z siecią łącząca węzły. Liczba przeplotów, możliwych kombinacji danych wejściowych zupełnie wyklucza tzw. wyczerpujące poszukiwanie błędów. Techniki stosowane w przypadku systemów jednozadaniowych zawodzą. Dla przykładu: cykliczne tropienie błędów, tak skuteczne dla programów sekwencyjnych, zastosowane wprost (czyli polegające na próbie zreprodukowania błędu poprzez uruchomienie systemu na tych samych danych wejściowych, które spowodowały problem) może skończyć się całkowitym fiaskiem. Może się nawet zdarzyć, że symptomy problemu w ogóle się nie pojawią przy powtórnym uruchomieniu.

Główną, powszechnie akceptowaną oraz stosowaną metodą tropienia błędów w systemach rozproszonych jest technika monitorowania oraz reprodukowania niepoprawnego wykonania. Obydwa składniki tej metody przysparzają wielu problemów, które są obiektem badań akademickich. Warto na przykład wspomnieć o klasycznym problemie obserwowalności (z ang. *observability problem*), który wskazuje, że obserwatorzy systemu rozproszonego mogą w inny sposób postrzegać jego działanie. Inny klasyczny przykład, tzw. problem próbkowalności (z ang. *probeability problem*) wskazuje na wpływ zrzucania informacji o systemie rozproszonym na jego działanie. Szersze spojrzenie na te kwestie dostarczyć może lektura np. [7, 4].

Główny kierunek wysiłków akademickich jest wyznaczony przez poszukiwanie metod deterministycznego odtwarzania funkcjonowania systemu rozproszonego. Celem jest bowiem możliwość skutecznego wykorzystania cyklicznego tropienia błędów w środowisku rozproszonym. Technika ta, mając do dyspozycji deterministycznie odtworzone wykonanie programu rozproszonego, nabiera wtedy sensu. Jest to oczywiście zadanie nietrywialne, biorąc pod uwagę konieczność zmierzenia się ze wszystkimi wyzwaniami, jakich dostarcza zapewnienie dokładnie tego samego wykonania. Istnieją rozwiązania deterministycznie odtwarzające działanie systemu rozproszonego. Jednakże wymagają one dużego narzutu czasowego bądź pamięciowego w ogóle ([10]), bądź w przypadku, gdy np. odtwarzany program wymaga dużej ilości pamięci ([6]). Inne zaś nakładają na aplikacje, w których poszukujemy problemów, duże ograniczenia (np. dotyczą systemów, w których węzły wymieniają się komunikatami (*z ang. message-passing programs*) oraz zachowują się deterministycznie pomiędzy zdarzeniami

komunikacji z innymi węzłami) ([11]).

Sensowne jest więc pytanie, czy można zaprojektować mechanizm, który nie będzie miał na celu deterministycznego odtwarzania systemu (tym samym nie dążąc do wykorzystania cyklicznego tropienia błędów w środowisku rozproszonym), ale pozwoli na zwiększenie skuteczności w wykrywaniu pewnej klasy błędów? Mechanizm, którego kluczową cechą ma być relatywna prostota, a w konsekwencji możliwość praktycznego zastosowania.

Celem pracy jest udokumentowanie analizy wymagań, projektu i implementacji praktycznego mechanizmu wizualizacji funkcjonowania systemu rozproszonego po awarii w celu tropienia błędów. Mechanizm ten był współtworzony przeze mnie w ramach projektu dla zastosowań komercyjnych. W pracy szczególny nacisk kładę na użyteczność i praktyczność projektowanego rozwiązania.

W rozdziale 2 szczegółowo formułuję problem, wskazuję jego istotne aspekty oraz opisuję istniejące podejścia do rozwiązywania analogicznych problemów. W rozdziale 3, bazując na zrozumieniu natury problemu oraz doświadczeniach, wynikających z analizy istniejących rozwiązań, formułuję wymagania dotyczące projektowanego mechanizmu. Następnie opisuję sam projekt i implementację, ze szczególnym zwróceniem uwagi na wyzwania i problemy, którym należało stawić czoła podczas projektowania i realizacji założeń projektowych. W rozdziale 4 prezentuję rozszerzalność autorskiego rozwiązania, opisując ideę i projekt mechanizmu wykrywania nietypowych sytuacji w systemie rozproszonym, który został zbudowany na bazie zrealizowanego pomysłu wizualizacji. W rozdziale 5 podsumowuję problemy poruszone w tej pracy w kontekście wysiłków włożonych w realizację praktycznego mechanizmu wizualizacji systemów rozproszonych w celu tropienia błędów.

Rozdział 2

Przegląd istniejących mechanizmów do wizualizacji systemów rozproszonych

Pracując nad rozwijaniem systemu rozproszonego dla zastosowań komercyjnych zetknąłem się z praktycznymi przykładami problemów z tropieniem błędów w takich systemach. Na własnej skórze doświadczyłem kłopotów, jakich to zadanie przysparza. Zespół, w którym pracowałem miał już opracowane pewne rozwiązania ułatwiające radzenie sobie z pewnymi klasami błędów tj. używanie niedozwolonego zakresu pamięci, wchodzenie w nieprawidłową ścieżkę wykonania, czy wycieki pamięci.

Jednakże brakowało nam narzędzia, które pozwoliłoby nam mieć pewien ogląd stanu całego systemu przed awarią. Oczywiście mieliśmy pewne informacje na temat stanu i zdarzeń na każdym z węzłów osobno, jednakże synteza tych informacji była żmudna, czasochłonna i nie zawsze skuteczna. Postanowiliśmy zatem pokusić się o zaprojektowanie i implementację praktycznego mechanizmu, który pozwoliłby na wygodne śledzenie funkcjonowania systemu post mortem (łac. *po śmierci*, tutaj po awarii).

2.1. Problemy z istniejącą metodą wykrywania błędów

Podczas rozwijania systemu rozproszonego natknąłem się na różnego rodzaju problemy: od kłopotów związanych ze środowiskiem wykonania, poprzez błędy w implementacji w zakresie błędnego użycia narzędzi języka programowania, błędy w synchronizacji, po niepoprawne sytuacje wynikające z nieprawidłowego projektu. Wraz z rozwojem implementacji mnoży się liczba komponentów systemu, jednostek wykonania. W konsekwencji trudniejsze jest ogarnięcie, co w danej chwili się dzieje w sercu systemu, jakie zdarzenia zachodzą, bądź zaszły, powodując potencjalny problem. Taki problem nie ominął również zespołu, z którym rozwijałem pewien system rozproszony. W systemie tym na dużą skalę stosowaliśmy tzw. logowanie śladów (ang. *traces*), czyli tekstowych wiadomości mówiących o tym, jakie czynności system wykonuje. Wówczas jedyną metodą lokalizacji niektórych błędów było równoległe przeglądanie wielu różnych plików z tymi wiadomościami (nazywane logami), aby odtworzyć przyczynę problemu. Nietrudno się jednak domyślić, że takie postępowanie nie dość, że żmudne i czasochłonne, to również często nie dawało pożądanych rezultatów. Powodem była m. in. niewystarczająca informacja zawarta w logach, możliwość przeoczenia pewnej ważnej informacji o zdarzeniu podczas przeglądania logów, co często prowadziło do wysnuwania nieprawdziwych wniosków, bądź niedostrzeżenia przyczyny błędu. Nie bez znaczenia była również sama

świadomość programisty, że takie postępowanie prawdopodobnie nie doprowadzi go do odnalezienia źródła problemu. W rezultacie mieliśmy znużonego programistę przeszukującego kilometrowe logi na wielu maszynach, próbującego ogarnąć stan systemu, zadania, które aktualnie wykonuje, ich postęp itd.

2.2. Wizualizacja funkcjonowania systemu rozproszonego pomocą w tropieniu problemów

Pojawia się zatem potrzeba prezentacji programiście działania systemu w sposób bardziej dla niego przyjazny. Narzucającym się rozwiązaniem tego problemu wydaje się być graficzne przedstawienie człowiekowi funkcjonowania systemu, wizualizacja jego stanu i zachodzących w nim procesów. Nie od dziś bowiem wiadomo, że często tę samą treść można w szybszy i bardziej przejrzysty sposób przekazać używając obrazu, a nie np. tekstu. Ta obserwacja jest przecież od lat stosowana podczas niemal wszystkich faz wytwarzania oprogramowania, czego przykładem jest np. taka popularność języka UML.

2.3. Przegląd istniejących rozwiązań

Pomysł wizualizacji działania systemów informatycznych nie jest nowy, więc zanim napisaliśmy pierwszy wiersz kodu, postanowiliśmy przyjrzeć się innym rozwiązaniom tego typu. Celem takiego badania było sprawdzenie, czy nie istnieją gotowe rozwiązania, które dałyby się niedużym kosztem wdrożyć do naszego projektu, rozwiązując tym samym postawiony problem.

2.3.1. Rodzaje oprogramowania do wizualizacji systemów rozproszonych

Rezultaty badania pokazały wielką różnorodność oprogramowania wizualizującego systemu współbieżne, równoległe, czy rozproszone. Istnieje wiele kategoryzacji takich rozwiązań, postaram się opisać kluczowe.

Wizualizacja na żywo vs. graficzna prezentacja systemu post mortem

Oprogramowanie wizualizujące może wspierać monitorowanie systemu w czasie jego działania, dając możliwość analizy stanu systemu, procesów zachodzących wewnątrz i potencjalnych problemów w czasie rzeczywistym. Są to mechanizmy obrazujące działanie systemu "na żywo" (ang. *on-line*). Rozwiązania, które umożliwiają wizualizację działania systemu rozproszonego wtedy, gdy obserwowany system już nie działa (w wyniku awarii czy zakończenia pracy), noszą miano wizualizujących post mortem. Ten typ zwykle w sposób specyficzny dla konkretnego rozwiązania zbiera i zapisuje informację o funkcjonowaniu systemu rozproszonego w czasie jego działania. Następnie zaś ta informacja jest odczytywana i przetwarzana w fazie wizualizacji. Moment wizualizacji jest tutaj niezależny od czasu działania samego systemu rozproszonego.

Mechanizmy wizualizacji na żywo znajdują zastosowanie np. w administracji systemami rozproszonymi. Znane są również przykłady oprogramowania do wizualizacji programów współbieżnych i rozproszonych na żywo w celach dydaktycznych. Rozwiązania wizualizujące funkcjonowanie systemu post mortem używa się również do celów dydaktycznych podczas nauki m. in. programowania współbieżnego. Jednakże wydaje się, że to zastosowanie wizualizacji post mortem jest szczególnie interesujące w przypadku prawdziwych, przemysłowych

systemów rozproszonych, gdyż może służyć jako nieoceniona pomoc przy tropieniu problemów w systemie, który jest już wdrożony u klienta. Wtedy często z powodów wydajnościowych tradycyjne logowanie śladów jest wyłączone lub pozostawione na wysokim poziomie, zatem narzędzie wizualizujące np. ostatnią godzinę działania systemu przed awarią może umożliwić skuteczne wykrycie przyczyny awarii.

Istnieją również systemy hybrydowe, które umożliwiają wizualizację na żywo z możliwością zrzutu obserwowanego wykonania i jego odtworzenia w przyszłości.

Szczegółowy cel wizualizacji

Nadrzędnym celem wizualizacji systemu rozproszonego jest prezentacja jego działania w sposób przyjazny człowiekowi. Jednakże szczegółowe cele wizualizacji mogą być różne. Objawia się to zwykle w tym, co wizualizujemy. Istnieją np. systemy wizualizujące systemy rozproszone w celu ułatwienia ogarnięcia globalnego stanu systemu, inne zaś głównie koncentrują się na wykrywaniu potencjalnych wąskich gardeł systemu w kontekście wydajności.

2.3.2. Typowe elementy systemów do wizualizacji programów rozproszonych

W rozdziale tym przedstawię typowe elementy większości systemów wizualizujących programy rozproszone. Oparłem się tutaj głównie na [5], gdzie moim zdaniem najbardziej przejrzyście przedstawiono strukturę takich rozwiązań. Typowe podejście do wizualizacji systemów rozproszonych zawiera w sobie następujące kroki:

- zebranie informacji o systemie, które chcemy wizualizować. Na tym etapie powstaje kilka podstawowych pytań, na które należy odpowiedzieć przy projektowaniu mechanizmu wizualizacji. Po pierwsze należy zdecydować, jakie informacje będziemy zbierać. Następnie na jakim poziomie będziemy je kolekcjonować, tzn. czy wybór padnie na wsparcie sprzętowe, czy też na poziomie samego oprogramowania. Trzeba również zastanowić się, jak projektowany sposób zbierania danych z systemu będzie wpływać na jego działanie;

- zapis informacji o systemie. W tym kroku fundamentalne problemy, które trzeba wziąć pod uwagę, dotyczą umiejscowienia danych, decyzji, czy mają być już przetwarzane na tym etapie oraz w jakiej formie informacje o systemie będą przechowywane. Należy tu dokonać wyboru, czy dane będą zapisywane na lokalnym dysku, czy też w pamięci dzielonej itp. Problem przetwarzania danych na etapie ich zapisu ma na celu ograniczenie zapamiętywania informacji potencjalnie nieistotnych z punktu widzenia samej wizualizacji. Jednakże idzie również w parze z dodatkowym narzutem w fazie zbierania danych. Problem formy przechowywania danych dotyczy głównie efektywności dostępu do nich oraz zajmowanego miejsca. Indeksowanie i kompresja to przykładowe techniki, które mogą okazać się pomocne przy rozwiązywaniu tych kwestii;

- analiza zapisanych danych. Na tym etapie fundamentalne decyzje dotyczą sposobu interpretowania oraz potencjalnych mechanizmów filtrowania zdarzeń wg pewnych zasad;

- graficzna prezentacja wybranych aspektów działania systemu. Tu pojawiają się takie kwestie jak tworzenie intuicyjnych widoków, wymaganie niewielkiego zaangażowania programisty w szczegóły samej graficznej warstwy. Istotna jest również decyzja, jak mechanizm będzie sygnalizował istotne informacje, jakie będą możliwości interakcji

użytkownika z widokiem. Aspektem do rozważenia jest również skalowalność projektowanych widoków, jako że sprzęt wyświetlający grafikę ma ograniczoną rozdzielczość, więc należałoby zadbać o przejrzystość prezentowanych obrazów nawet przy dużej liczby węzłów w systemie.

2.3.3. Konkretne przykłady mechanizmów wizualizacji

W tym rozdziale postaram się zaprezentować kilka istniejących narzędzi do wizualizacji systemów rozproszonych, na które mój zespół natknął się podczas badania postawionego zagadnienia. Pokrótce omówię każde rozwiązanie zwracając uwagę na ich przynależność do wcześniej zdefiniowanych kategorii, badając zalety i wady oraz rozważając potencjalną użyteczność w rozwiązaniu postawionego problemu.

ConcurrentMentor

ConcurrentMentor ([2]) jest narzędziem stworzonym dla potrzeb dydaktycznych. Jego zadaniem jest ułatwienie studentom zrozumienia charakterystyki programów współbieżnych i rozproszonych poprzez wizualizację na żywo oraz post mortem. Rozwiązanie to należy więc do rodziny mechanizmów hybrydowych, gdyż pozwala na zapis obserwowanego wykonania i odtworzenia w późniejszym czasie.

Możliwości ConcurrentMentor pozwala na wizualizację dowolnych programów napisanych w języku C++, które korzystają z predefiniowanych kanałów komunikacyjnych (istnieje również bratni projekt ThreadMentor ([3]) służący do wizualizacji metod synchronizacji procesów współbieżnych w celach dydaktycznych). ConcurrentMentor dostarcza takie abstrakcje komunikacji jak synchroniczne, bądź asynchroniczne kanały symulujące np. komunikację sieciową. Mamy do dyspozycji implementację zegarów logicznych Lamporta ([9]), aby korzystać z częściowego porządku zdarzeń w systemie. Mechanizm umożliwia definiowanie topologii wizualizowanego systemu rozproszonego, daje również do dyspozycji predefiniowane topologie np. gwiaździstą, pierścienia, kliki. Zarówno abstrakcje kanałów komunikacyjnych, jak i topologie są dostarczane użytkownikowi w postaci klas. Ich implementacja dba o dostarczanie modułowi wizualizującemu informacji na temat zdarzeń na każdym węźle, sam program użytkownika nie wymaga żadnych modyfikacji, aby włączyć wsparcie dla wizualizacji. Jest ono już wbudowane. ConcurrentMentor w warstwie wizualizacji zapewnia predefiniowane widoki pozwalające na zobrazowanie wybranych aspektów działania programów rozproszonych. Możliwa jest graficzna prezentacja topologii obserwowanego systemu, komunikacji poprzez dostarczane kanały na widoku z osiami czasowymi (zdarzenia są porządkowane za pomocą zegarów logicznych) oraz statystyk wysłanych/odebranych komunikatów.

Ograniczenia ConcurrentMentor i ThreadMentor wydają się być doskonałymi narzędziami do wizualizacji systemów rozproszonych, czy współbieżnych w celach dydaktycznych. Jednakże ich stosowalność w większych projektach wydaje się bardzo wątpliwa. Powodów jest kilka:

- choć programy użytkownika nie wymagają żadnych modyfikacji, to muszą one najpierw powstać w oparciu o predefiniowane klasy reprezentujące kanały komunikacyjne;
- wizualizacja obejmuje jedynie kilka predefiniowanych widoków, jest ściśle zależna od kolekcjonowanych informacji o systemie, nierozszerzalna;

- jako narzędzie ukierunkowane dydaktycznie jest mało użyteczne w przypadku dużych systemów z dużą liczbą węzłów.

Środowisko PARADE

Środowisko PARADE ([12, 13]) stworzone przez pracowników Georgia Institute of Technology jest chyba najczęściej cytowanym rozwiązaniem w literaturze poruszającej problem wizualizacji systemów rozproszonych. Dlatego postaram się opisać je trochę szerzej. PARADE jest platformą mającą na celu ułatwienie graficznej prezentacji systemów rozproszonych post mortem. Pozwala na niezbyt skomplikowaną integrację zapewnianego przez nią wsparcia dla wizualizacji z rozwijanym systemem, umożliwia inteligentną prezentację graficzną zdarzeń z zachowaniem relacji częściowego porządku zdarzeń w systemie (opcjonalnie). Szczególną uwagę poświęca rozszerzalności swego rozwiązania, łatwości integracji i ubytku wydajności wizualizowanego systemu z powodu wdrożenia mechanizmu jego wizualizacji.

Cele Autorzy PARADE na podstawie swych badań zaobserwowali niewielkie wykorzystanie wizualizacji systemów rozproszonych w celach zrozumienia istoty jego działania, weryfikacji poprawności, optymalizacji wydajności, czy tropienia błędów. Wysunęli oni hipotezę, że główny problem leży w uzyskaniu informacji potrzebnych do graficznego wyświetlenia, oraz powiązanie zachowanych informacji z widokami narzędzi graficznych. Bazując na tych obserwacjach postawili sobie kilka celów do osiągnięcia w ich rozwiązaniu. Oto one:

- minimalizacja wysiłku użytkownika rozumiana na kilku płaszczyznach. Pierwsza z nich dotyczy możliwie jak najmniejszego trudu związanego z instrumentowaniem kodu w celu zapisania istotnych, przez co interesujących późniejszego obserwatora, zdarzeń. Wszędzie, gdzie to możliwie, zrzut zdarzeń powinien odbywać się automatycznie, bez interwencji programisty. Druga zaś stawia mechanizmowi wymaganie, aby tworzenie widoków dla systemu nie wymagała od programisty dużej wiedzy w zakresie programowania aplikacji z interfejsem graficznym;

- wykorzystanie mechanizmu na żądanie. Oznacza to, że mechanizm powinien być w prosty sposób możliwy do użycia, bądź przeciwnie, do wyłączenia bez modyfikacji kodu źródłowego wizualizowanego systemu;

- minimalny narzut na funkcjonowanie mechanizmu. Oznacza to, że silnik do wizualizacji powinien w zaniedbywalny sposób wpływać na działanie samego systemu;

- wsparcie dla porządkowania zdarzeń zarówno na podstawie lokalnych stempli czasowych (pochodzących od zwykłego zegara), bądź w wyniku użycia logicznych zegarów Lamporta ([9]).

Architektura przyjętego rozwiązania PARADE składa się z trzech podstawowych komponentów:

- komponent odpowiadający za monitorowanie działającego węzła systemu rozproszonego. Jego zadanie polega na trwałym zapisie zdarzeń zachodzących w węźle. W PARADE został on zrealizowany jako funkcja biblioteczna `c_parade_log()`. Funkcja ta, mająca składnię podobną do standardowej funkcji `printf`, pozwala zapisać typy zdarzeń, których struktura musi zostać wcześniej zdefiniowana w tekstowym pliku konfiguracyjnym;

- silnik graficzny. Został on zaimplementowany jako oddzielna biblioteka oraz nazwany POLKA. Zawarto w niej pewne predefiniowane widoki dla typowych przypadków użycia wizualizacji systemów rozproszonych (np. obrazujące komunikację programów wymieniających komunikaty), jednakże daje możliwość tworzenia własnych widoków bądź to na bazie istniejących bloków budulcowym zapewnianych przez bibliotekę POLKA, bądź to zupełnie od zera;

- tzw. "Choreograf" (z ang. *Choreographer*). Jego zadaniem jest integracja zapisywanych zdarzeń systemu ze zmianami widoku. Ta warstwa pośrednia spajająca poprzednie dwa komponenty jest, jak autorzy uzasadniają, niezbędna do poprawnej implementacji wizualizacji. Poprawnej to znaczy z zachowaniem porządku indukowanego przez zegary logiczne.

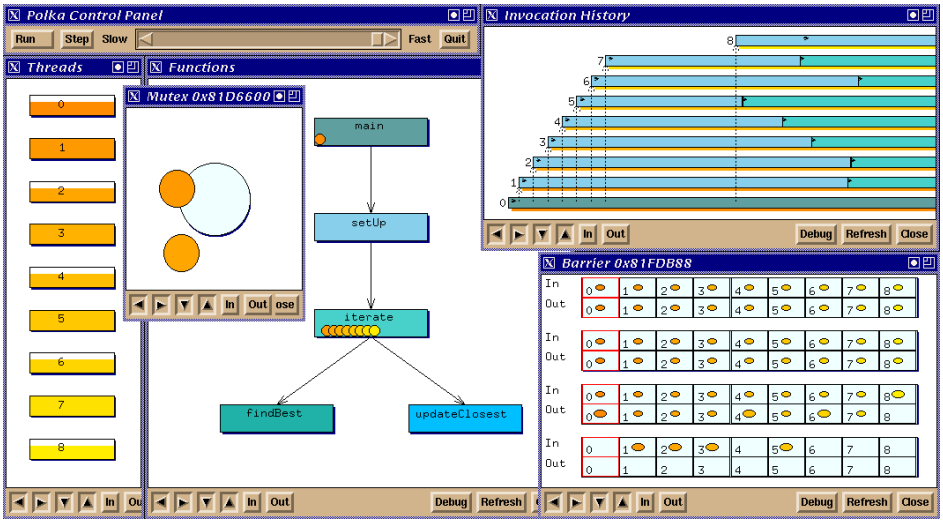
Realizacja założeń Twórcom PARADE udało się zrealizować postawione cele, które w ich mniemaniu są wyznacznikiem dobrego mechanizmu wizualizacji systemów rozproszonych.

Wymaganie minimalizacji wysiłku programisty udało się zrealizować najskuteczniej przy okazji systemów, w których istotna jest wizualizacja pewnego typowego jej elementu. Należy bowiem zauważyć, że jeżeli system rozproszony wymaga bardzo wysublimowanych widoków, bądź obrazowane graficznie informacje są bardzo skomplikowane, bądź informacje potrzebne do wizualizacji są bardzo porozrzucane po kodzie, to dodatkowego wysiłku programisty po prostu nie da się uniknąć. Nawet używając PARADE należy wtedy samodzielnie zmodyfikować kod źródłowy rozwijanego systemu, aby zachowywał istotne dla wizualizacji ślady, następnie zdefiniować wyspecjalizowane widoki oraz określić odwzorowanie zapisanych zdarzeń do zmiany widoku. Jednakże wykorzystanie PARADE w systemach, w których istotna jest np. tylko ilustracja komunikacji przez sieć (czyli np. kto, kiedy i do kogo wysyła komunikaty), jest bardzo proste. Autorzy PARADE, integrując swój mechanizm wizualizacji z pewnym systemem obliczeń rozproszonych, musieli zmodyfikować jedynie użyte tam funkcje `Send` i `Receive`. Modyfikacje polegały na trwałym zapisie zdarzenia wysłania/odebrania komunikatu oraz na doklejaniu (z ang. *piggybacking*) licznika potrzebnego do implementacji zegarów logicznych Lamporta. Co ważne zmiany kodu źródłowego wizualizowanego systemu były proste i skupione w jednym miejscu. Dzięki temu niewielkim wysiłkiem udało się twórcom PARADE zintegrować wsparcie dla graficznej prezentacji z kilkoma systemami rozproszonymi (owocem tych prac jest implementacja pod nazwą PVaniM). Podobny efekt osiągnięto podczas implementacji wizualizacji metod synchronizacji i cyklu życia wątków (biblioteka GThread). Rysunek 2.1 przedstawia przykładową wizualizację wielowątkowego programu współbieżnego. Rysunek prezentuje między innymi, jaką funkcję aktualnie wykonuje dany wątek, który z nich jest w sekcji krytycznej, obrazuje "historię życia wątków", wizualizuje proces przekraczania bariery.

Wymaganie możliwości wykorzystania mechanizmu na żądanie zostało zrealizowane na poziomie konsolidacji binariów (na etapie linkowania, bądź ładowania biblioteki dynamicznej) wizualizowanego systemu z biblioteką zawierającą implementację `c_parade_log()`. Chcąc uruchomić system ze wsparciem dla późniejszej wizualizacji należy tylko zlinkować kod wykonywalny z biblioteką zapewniającą prawdziwą implementację `c_parade_log()`, w przeciwnym przypadku należy dołączyć bibliotekę z pustą implementacją tej funkcji.

Jeśli chodzi zaś o narzut funkcjonowania mechanizmu wizualizacji na działanie samego systemu, to autorzy przywołują wyniki testów wydajnościowych porównujące czasy wykonania z załadowaną implementacją komponentu monitorującego oraz bez niego. Wyniki ich testów pokazały, że wpływ jest zaniedbywalny.

Autorzy PARADE szczególną uwagę zwrócili na poprawną implementację wizualizacji



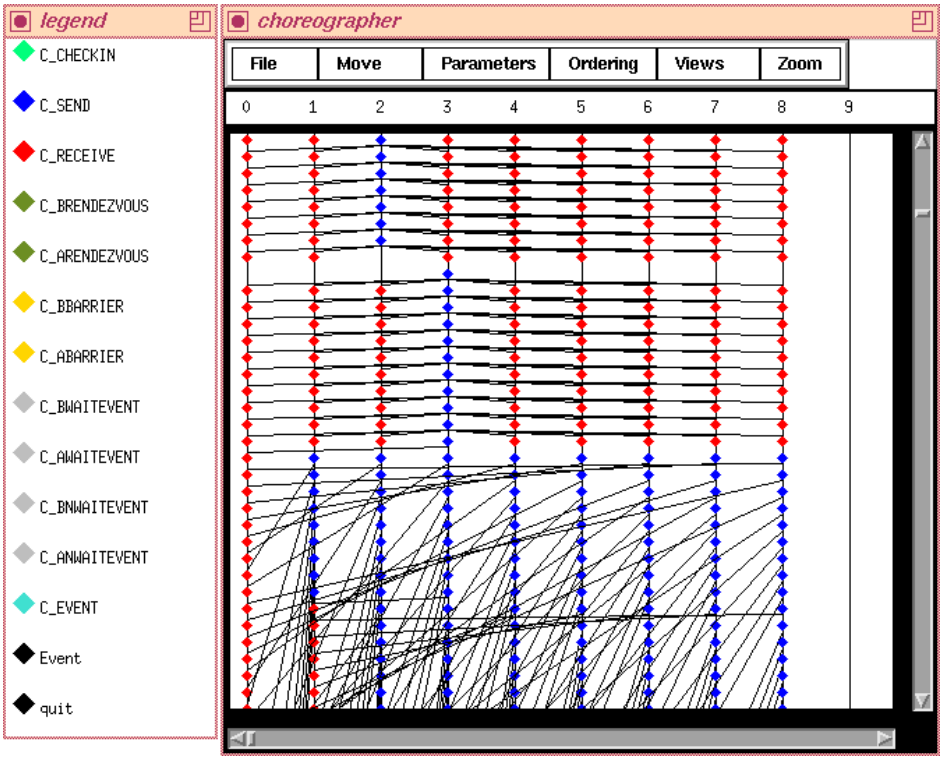
Rysunek 2.1: Wizualizacja za pomocą biblioteki GThread. Zaczerpnięte z [1]

działania systemu rozproszonego w kontekście porządkowania zapisanych zdarzeń. Po to właśnie stworzono komponent "Choreograf", aby najpierw ustawić zdarzenia w porządku zażądanym przez użytkownika. Co więcej możliwa jest dynamiczna modyfikacja widoku ze względu na wybrany rodzaj porządkowania. Rysunek 2.2 zawiera przykładowy widok obrazujący uporządkowanie zdarzeń pomiędzy procesami.

PARADE – podsumowanie Środowisko PARADE jest dobrze przemyślanym mechanizmem wizualizacji programów współbieżnych i systemów rozproszonych. Sprawdza się szczególnie przy typowych zastosowaniach, takich jak np. graficzne śledzenie wysyłanych/odbieranych komunikatów, czy monitorowanie wątków. Mimo że PARADE daje możliwości rozszerzeń, to razem z zespołem uznaliśmy, że koszt implementacji takich rozszerzeń niestety byłby zbyt wysoki. Powodem był fakt, że jednym z ważniejszych obiektów naszego zainteresowania była graficzna prezentacja statystyk zbieranych w naszym systemie. Liczba statystyk była rzędu kilku tysięcy. Analiza środowiska PARADE pokazała, że musielibyśmy wyspecyfikować je wszystkie w pliku konfiguracyjnym oraz dla każdej z nich oddzielnie modyfikować kod tak, aby każdą z nich zapisywał jako informację potrzebną do późniejszej wizualizacji. Ponadto mieliśmy gotowe narzędzie do wizualizacji naszego systemu na żywo, które w żaden sposób nie przystawało do technologii i metodologii użytej w bibliotece POLKA, nie wspominając już o tym, że kod źródłowy PARADE jest wiekowy i wymagałby dostosowania do obecnie obowiązujących standardów języka C++. Powyższe wady skutecznie odstraszyły nas od wykorzystania PARADE do wizualizacji naszego systemu, choć wydawało się na początku, że jest to rozwiązanie trafiające w dużą część naszych potrzeb.

2.3.4. Podsumowanie cech istniejących rozwiązań i możliwości integracji

Przedstawiłem kilka pomysłów na wizualizację systemów rozproszonych. Wskazałem cele graficznej prezentacji działania systemu rozproszonego oraz używane techniki do ich osiągnięcia. Jednakże żadne ze zbadanych narzędzi nie spełniło naszych oczekiwań. Głównym powodem był potencjalny nakład pracy, który należałoby włożyć w integrację zewnętrznego narzędzia do naszych celów. Jak się spodziewaliśmy, wykorzystanie posiadanego już przez nas narzę-



Rysunek 2.2: Wizualizacja uporządkowanych zdarzeń w środowisku PARADE. Zaczerpnięte z [8]

dzia graficznie prezentującego system w czasie jego działania w połączeniu z którymkolwiek z rozwiązań wydawało się skomplikowanym zadaniem. Natomiast wizja wykorzystania predefiniowanych widoków bądź żmudne definiowanie na nowo widoków zgodnych z potencjalnym narzędziem wydawało się być powtórzeniem pracy, która już została wykonana przy okazji implementacji narzędzia prezentującego nasz system graficznie w czasie jego działania. To doświadczenie oraz realna potrzeba skłoniła nas do projektu i implementacji własnego narzędzia.

Rozdział 3

Projekt i implementacja autorskiego rozwiązania

Istniejące rozwiązania z zakresu wizualizacji systemów rozproszonych nie spełniły oczekiwań naszego zespołu. Przeprowadzone badanie pozwoliło nam sformułować argumenty zniechęcające do ich użycia w naszym projekcie. Najważniejsze z nich to niemożność łatwej integracji z istniejącymi narzędziami graficznie wizualizującymi działanie naszego systemu na żywo, ograniczony zakres wizualizacji, konieczność użycia kanałów komunikacyjnych danych przez autora rozwiązania, czy konieczność modyfikowania kodu w wielu miejscach w celu zrzucania pożądanych informacji. Doświadczenia dotychczasowej praktyki i analiza istniejących rozwiązań stały się podstawą do sformułowania wymagań dla projektowanego mechanizmu wizualizacji funkcjonowania systemu w celu tropienia błędów.

3.1. Wymagania względem projektowanego mechanizmu

Mechanizm wizualizacji powinien:

- pozwalać prezentować w sposób przyjazny człowiekowi działanie systemu post mortem;
- pozwalać na wygodne skoki pomiędzy punktami na osi czasowej wykonania;
- wymagać niewielkiego narzutu funkcjonowania mechanizmu na działanie samego systemu;
- wymagać praktycznie akceptowalnych zasobów do prezentacji działania systemu po awarii;
- pozwalać na jego wykorzystanie na żądanie, to znaczy w prosty sposób możliwe ma być jego wyłączenie;
- być w miarę możliwości łatwy w implementacji i wdrożeniu;
- być łatwy w użyciu i nie wymagać od użytkownika bolesnego wysiłku w celu jego uruchomienia;
- być w miarę możliwości ogólny i nie przywiązany do charakterystyki danego systemu rozproszonego;
- być niezależny od architektury, na której jest wdrażany wynikowy system.

3.2. Idea

Idea naszego rozwiązania nie różni się od typowych systemów wizualizacji post mortem, gdyż w tego typu mechanizmach zwykle są dwie fazy: nagrywania, czyli zrzutu migawek z działania systemu, oraz wizualizacji (odtwarzania) na podstawie zapisanych zrzutów. W tekście czasami będę używał terminów "odtwarzanie" i "wizualizacja" jako synonimów, mimo że pierwszy termin można odnosić tylko do interpretowania zapisanych migawek, zaś drugi do graficznej ich prezentacji. Dla ustalenia uwagi nazwijmy komponenty odpowiedzialne za każdą z faz. Komponentem odpowiedzialnym za fazę nagrywania będzie tzw. DUMPER, zaś za odtwarzanie odpowiedzialność obejmie PLAYER, którego zadaniem będzie odczyt zapisanych zrzutów i umiejętność komunikacji z komponentem VISUALIZER, który to posłuży do samej graficznej prezentacji odtwarzania.

Rysunek 3.1: Idea nagrywania (komponent DUMPER)

Nasz mechanizm opiera się na umieszczeniu na każdym węźle jednostki (DUMPER), która będzie co jakiś czas wypytywać lokalny węzeł w czasie jego działania o pewne informacje na temat jego lokalnego stanu (por. rys. 3.1). Informacje te będą zapisywane trwale na urządzeniu pamięci stałej w sposób rozproszony (każdy węzeł zapisuje migawki lokalnie np. na dysku). Potem zaś w fazie odtwarzania informacje zapisane w pamięci stałej będą zbierane ze wszystkich węzłów i interpretowane przez komponent PLAYER aby ostatecznie posłużyć jako wejście dla komponentu wizualizującego VISUALIZER (por. rys. 3.2). W czasie wizualizacji użytkownik będzie mógł swobodnie sterować czasem i szybkością odgrywania, tak aby zlokalizować moment, w którym nieprawidłowe zachowanie daje się zaobserwować. Można obrazowo powiedzieć, że PLAYER będzie suwakiem, a VISUALIZER silnikiem graficznym odtwarzacza filmów. Filmem będzie zapis działania systemu rozproszonego. Widz będzie mógł oglądać film w zwykłym trybie, przeglądać go klatka po klatce, oglądać go w przyspieszonym tempie, czy przewijać taśmę do dowolnie wybranego miejsca.

3.3. Wymagania wobec rozwijanego systemu

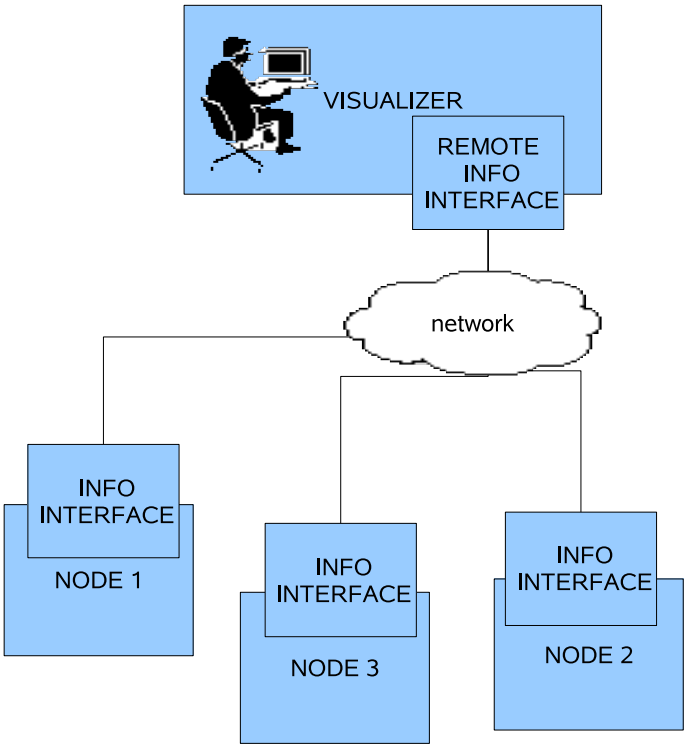
Projektowany przez nasz zespół mechanizm powstawał w celu rozwiązania konkretnych problemów, napotkanych podczas rozwijania pewnego systemu rozproszonego. Kluczowym założeniem była jego skuteczność osadzona w realiach projektu, nie zaś jego ogólność implikująca możliwość bezbolesnego zastosowania w innym projekcie. Jednakże postaram się uzasadnić, że opracowane rozwiązanie nie jest sztywno związane z konkretnym systemem, aczkolwiek nakłada na niego pewne obostrzenia, które warunkują jego stosowalność. Aby można było wykorzystać nasz mechanizm do wizualizacji systemu, powinien on:

Rysunek 3.2: Idea odgrywania (komponent PLAYER i VISUALIZER)

- udostępniać metodę do uzyskiwania informacji na jego temat. Musi zatem posiadać coś, co na potrzeby tej pracy nazwiemy interfejsem informacyjnym i będziemy zapisywać dalej jako INFO INTERFACE. System musi implementować ten interfejs, aby komponent DUMPER mógł pytać system o jego stan w celu zrzucenia na urządzenie pamięci stałej;
- zapewnić, że każdy element stanu węzła zwracany w wyniku interakcji komponentu DUMPER z interfejsem informacyjnym posiada metodę zapisania swojego stanu jako ciąg bajtów (może ulec tzw. serializacji) oraz metodę odtworzenia jego stanu z tegoż ciągu bajtów (może powstać w wyniku tzw. deserializacji);
- zapewniać bibliotekę, dzięki której będzie można używać interfejsu informacyjnego INFO INTERFACE na zdalnych węzłach. Nazwijmy ją zdalnym interfejsem informacyjnym (REMOTE INFO INTERFACE). Jej zadaniem jest umiejętne przekazywanie żądań z procesu klienta poprzez sieć do właściwych węzłów systemu rozproszonego. Umiejscowienie komponentu REMOTE INFO INTERFACE prezentuje rysunek 3.3;
- dawać łatwo rozszerzalne narzędzie do obserwacji działania systemu rozproszonego w trakcie jego działania (VISUALIZER). Rozszerzalne, gdyż musi się dać łatwo zmodyfikować, aby umieć działać w trybie obserwacji żywego systemu, jak i odtwarzania jego działania na podstawie trwale zapisanych zrzutów stanu. Na pierwszy rzut oka może to się wydawać wymaganiem enigmatycznym i jednocześnie niepraktycznym, jednak jest to wymagane do działania całego mechanizmu. Zakres tych zmian opisuję dokładniej w 3.5.6. VISUALIZER również musi używać interfejsu informacyjnego do uzyskiwania informacji na temat poszczególnych węzłów;
- dopuszczalne są niewielkie różnice pomiędzy czasem wskazywanym przez zegary na poszczególnych węzłach, jednak generalnie nie może być gigantycznych różnic pomiędzy nimi. To wymaganie nie powinno również w wielkim stopniu ograniczać proponowanego rozwiązania, jako że istnieją obecnie protokoły synchronizacji czasu, które działają z dużą dokładnością. My zaś nie wymagamy dokładnej synchronizacji, błąd nawet rzędu jednej sekundy jest akceptowalny.

3.4. Architektura

Mechanizm w wersji podstawowej składa się z dwóch komponentów, o nazwie DUMPER i PLAYER. Jednakże zanim opiszę zasadę działania każdego z nich, przyjrzyjmy się wpierw



Rysunek 3.3: Schemat monitora systemu rozproszonego

elementom systemu, których obecności zażądaliśmy w rozdziale 3.3. Elementy te nie są zasadniczo częścią samego rozwiązania, jednak są wymagane, aby mechanizm mógł działać.

3.4.1. INFO INTERFACE

Aby komponent DUMPER miał szansę działać, system musi posiadać pewien interfejs, dzięki któremu DUMPER będzie mógł pobierać stan lokalnego węzła. Nazwaliśmy to interfejsem informacyjnym i oznaczyliśmy jako INFO INTERFACE. Implementacja tego interfejsu w każdym węźle systemu rozproszonego powinna potrafić przyjmować żądania klientów, wyciągać wymagane informacje i przekazywać je do klienta. Rozwiązanie nie narzuca ograniczeń na implementację interfejsu informacyjnego, jednakże jego projekt musi być na tyle ogólny, aby można było łatwo użyć jego implementacji do komunikacji komponentu VISUALIZER z węzłami systemu rozproszonego. Może to być interfejs programistyczny czy usługa sieciowa obsługująca ustalony protokół.

Aby przybliżyć ideę interfejsu informacyjnego przedstawię przykładową jego postać. Niech dla ustalenia uwagi będzie to asynchroniczny interfejs programistyczny, który wydaje się być tu naturalnym rozwiązaniem, jako że klient w oczekiwaniu na wynik zapytania może robić inne pożyteczne rzeczy. Jego szkielet można zawrzeć w przykładowym kodzie w języku C++ zamieszczonym na rysunku 3.4. Zauważmy, że oprócz metod definiujących żądania w interfejsie można znaleźć dodatkową funkcję `sleep`, której znaczenie opisuję w p. 3.5.4.

```
class InfoInterface
{
public:
    // user's callback as processing is asynchronous
    class UserRequestCallback
    {
        /**
         * Called when request submitted succeeded.
         */
        void requestCompleted(NodeInfo const & info) = 0;

        /**
         * Called when request submitted failed.
         */
        void requestFailed(ErrorInfo const & info) = 0;

        /**
         * Called when request submitted timed out.
         */
        void requestTimedOut() = 0;
    };

    // example method
    void submitRequestGetNodeInformation(
        NodeAddress targetNode,
        UserRequestCallback *callback,
        TimeDuration const & timeout
    ) = 0;
    // other methods...

    //special sleep function
    void sleep(TimeDuration const & period) = 0;
};
```

Rysunek 3.4: Przykładowy szkielet interfejsu informacyjnego

3.4.2. REMOTE INFO INTERFACE

Kolejnym wymaganiem względem systemu rozproszonego, w którym chcielibyśmy sprawnie użyć naszego mechanizmu wizualizacji w celu skutecznego tropienia problemów, jest posiadanie implementacji interfejsu informacyjnego do użytku na węzłach klienckich (nie należących bezpośrednio do samego systemu). Implementację tę nazwaliśmy zdalnym interfejsem informacyjnym i oznaczyliśmy jako REMOTE INFO INTERFACE. REMOTE INFO INTERFACE ma za zadanie umożliwić aplikacjom klienckim prostą metodę komunikacji z węzłami systemu rozproszonego. Mówiąc "aplikacje klienckie" można myśleć o wszelkim oprogramowaniu monitorującemu stan systemu rozproszonego. Wybór takiego rozwiązania kwestii komunikacji programów monitorujących z samym systemem wydaje się o tyle wygodny, że nie trzeba

wówczas za każdym razem implementować komunikacji sieciowej pomiędzy nimi. Łatwiej jest dostarczyć i wykorzystać bibliotekę, która będzie się umiała porozumiewać z samym systemem. Dodatkową zaletą takiego rozwiązania jest możliwość użycia istniejących narzędzi do obserwacji działania systemu rozproszonego w czasie jego działania w celu wizualizacji jego poczynań post mortem. Będzie to możliwe poprzez podmianę rzeczywistego komponentu REMOTE INFO INTERFACE na fałszywy, który zamiast przesyłać żądania do rzeczywistego węzła systemu, przekieruje je do komponentu PLAYER.

3.4.3. VISUALIZER

Aby stosunkowo niskim kosztem wdrożyć projektowany mechanizm skutecznego tropienia problemów w systemie rozproszonym z wizualizacją działania systemu już po awarii, potrzebujemy również samych narzędzi wizualizujących. Wymaganie stworzenia takiego narzędzia od zera tylko na potrzeby odgrywania działania systemu post mortem wydaje się być jednak nieakceptowalne. Pojawia się zatem pomysł wykorzystania istniejących narzędzi. Te z pewnością egzystują dla lwiej części rozwijanych systemów, gdyż trudno sobie wyobrazić system, który jest skutecznie rozwijany, a takowych nie posiada. Zwykle takie oprogramowanie działa w ten sposób, że co jakiś czas komunikuje się z węzłami systemu, odpytując je o interesujące dla siebie informacje, które następnie wizualizuje w specyficzny dla siebie sposób. Jednakże nasz mechanizm wymaga, aby to oprogramowanie monitorujące nie implementowało samodzielnie komunikacji z węzłami w sieci. Wymagamy, aby odbywało się to przez dostarczoną bibliotekę REMOTE INFO INTERFACE, która się o to troszczy. Zyskujemy wówczas możliwość nieskomplikowanego przystosowania takiego narzędzia do wizualizacji działania systemu po awarii.

3.4.4. FAKE REMOTE INFO INTERFACE

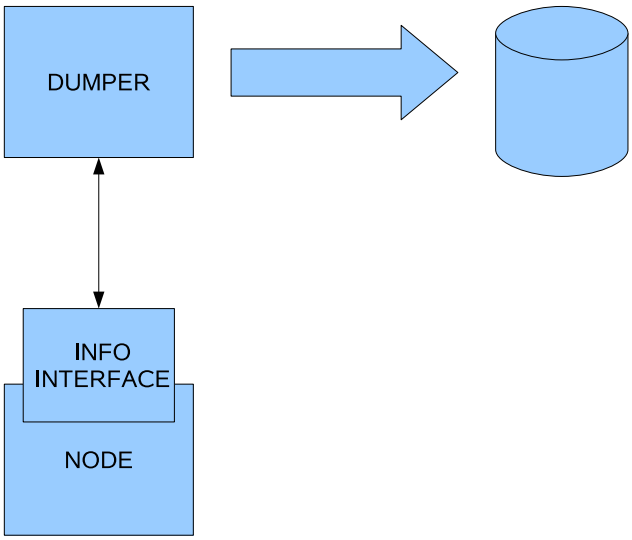
FAKE REMOTE INFO INTERFACE jest implementacją INFO INTERFACE, która zamiast wysyłać żądania komponentu VISUALIZER do rzeczywistych węzłów, będzie przekazywać je jednostce obsługującej odgrywanie. Będzie ona zastępowała REMOTE INFO INTERFACE podczas fazy odgrywania w naszym mechanizmie, aby umożliwić łatwe użycie istniejących narzędzi monitorujących i wizualizujących działanie systemu do wizualizacji systemu post mortem.

3.4.5. DUMPER

DUMPER jest komponentem odpowiedzialnym za nagrywanie migawek z aktywości systemu rozproszonego. Jest on uruchamiany na każdym jego węźle. Jednostka ta ma za zadanie periodycznie odpytywać lokalny węzeł o jego stan poprzez interfejs informacyjny (INFO INTERFACE). DUMPER umie następnie zamienić ten stan w ciąg bajtów i zapisać na urządzeniu pamięci stałej (używając metod serializacji otrzymanego obiektu). Informacje będą zapisane sekwencyjnie, każda pozycja będzie zawierać czas zapisu i binarną reprezentację zebranej informacji. Usytuowanie komponentu DUMPER prezentuje rys. 3.5.

3.4.6. PLAYER

PLAYER jest komponentem, który potrafi interpretować zebrane w jedno miejsce zrzuty z podzbioru węzłów systemu. Początkowo miał on również pozwalać użytkownikowi mechanizmu sterować czasem i szybkością odtwarzania. Jednak na etapie projektowania uznaliśmy,



Rysunek 3.5: Zasada działania komponentu DUMPER (widok jednego węzła)

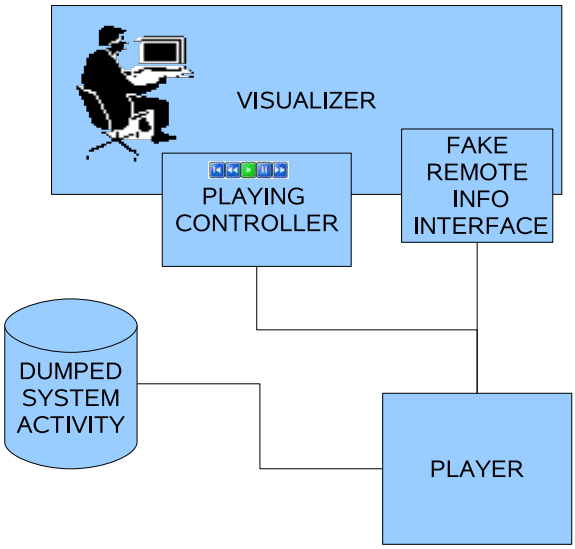
że ta funkcjonalność powinna zostać zawarta w oddzielnym module, nadbudowanym nad interfejsem komponentu PLAYER. Istotnie taka jednostka powstała i jest opisana szerzej w p. 3.4.7.

Podczas wizualizacji VISUALIZER będzie myślał, że rozmawia za pomocą interfejsu informacyjnego z rzeczywistymi węzłami. W rzeczywistości zaś VISUALIZER będzie oszukiwany, gdyż będzie komunikować się z komponentem PLAYER (przezroczyście, gdyż FAKE REMOTE INFO INTERFACE będzie mieć taki sam interfejs jak REMOTE INFO INTERFACE). Ten zaś otrzymując żądania o stan systemu będzie przekazywał odzyskane informacje na podstawie zachowanych zrzutów. Proszę zauważyć, że VISUALIZER musi zostać zmodyfikowany, aby umieć poprawnie dostarczać swoją funkcjonalność wizualizującą zarówno w trakcie jego działania, jak i w czasie komunikacji z fałszywą jego inkarnacją pod postacią komponentu FAKE REMOTE INFO INTERFACE. Naturę i zakres tych zmian opisuję precyzyjniej w p. 3.5.6. Komponent PLAYER będzie musiał w locie interpretować zapisane migawki z wielu węzłów. Będzie musiał efektywnie scalać zapisane zrzuty w ciąg chronologicznie ustawionych zdarzeń. Usytuowanie komponentu PLAYER prezentuje rys. 3.6.

Sam PLAYER będzie zatem dostarczał operacje takie jak "przejdź do następnej migawki", "idź do zdarzenia, które wystąpiło najbliżej czasu t" itd. Sterowanie tempem wizualizacji nie będzie więc w zakresie funkcjonalności komponentu PLAYER.

3.4.7. PLAYING CONTROLLER

Sterowanie czasem i szybkością odtwarzania to zadanie komponentu PLAYING CONTROLLER. Ma on pozwolić użytkownikowi zacząć, bądź kontynuować wizualizację systemu rozproszonego w dowolnie wybranym punkcie na osi czasu oraz pozwolić wybrać prędkość, z jaką kolejne zdarzenia mają być prezentowane. Będzie on nadbudowany nad interfejsem komponentu PLAYER i w zależności od woli użytkownika, będzie odpowiednio często żądał przejścia do następnej zapisanej migawki, wywoływał operacje zatrzymania odtwarzania, czy też



Rysunek 3.6: Zasada działania komponentu PLAYER

przeniesienia się we wskazane przez użytkownika miejsce na osi czasowej. Umiejscowienie komponentu PLAYING CONTROLLER prezentuje rys. 3.6.

3.5. Projekt

Oprócz nieuchronnie czekających nas wyzwań implementacyjnych, wpierw należało rozwiązać kilka kluczowych kwestii projektowych. W rozdziale tym skoncentruję się na bardziej szczegółowym opisie każdego z komponentów rozwiązania oraz zasadach ich działania. Opiszę również, w jaki sposób zaplanowaliśmy integrację mechanizmu wizualizacji systemu rozproszonego z istniejącymi narzędziami do graficznej prezentacji funkcjonowania systemu oraz jakie problemy należało tu rozwiązać.

3.5.1. INFO INTERFACE

W naszym systemie interfejs informacyjny został zrealizowany jako asynchroniczny interfejs programistyczny według schematu zaprezentowanego na rysunku 3.4. Tak jak zamieszczony szkielet interfejsu wskazuje jego użytkownik może zlecić żądanie uzyskania informacji na temat węzła z obiektem funkcji zwrotnej (ang. *callback*), którego metody zostaną wywołane, gdy system uzyska tę informację lub gdy żądanie nie powiedzie się np. z powodu podania złych parametrów wejściowych lub też gdy okaże się, że nie można jej uzyskać we wskazanym przez użytkownika okresie czasu.

Dla potrzeb sterowania czasem i prędkością odtwarzania potrzebowaliśmy dodać do interfejsu jedną funkcję `sleep(time)`, której semantykę i celowość opisuję w 3.5.6. Funkcja ta będzie wołana przez komponent VISUALIZER, aby warstwa uzyskiwania informacji o węzłach działała niezależnie od trybu wizualizacji (działającego systemu, czy post mortem).

3.5.2. DUMPER

Nośnik zapisywanych migawek

Pamięcią stałą, z której korzysta DUMPER są zwykłe pliki na dysku. Pierwszy plik zawiera metadane zachowywanego zrzutu (w oddzielnym pliku), tak aby w połączeniu z nim samym były samoopisującą się całością. W naszym przypadku te metadane zawierały adres sieciowy węzła naszego systemu, ale może to być równie dobrze dowolna inna informacja potrzebna do identyfikacji węzła, jego inkarnacji itd. Pliki zaś z samym zrzutem mają konfigurowalną wielkość zarówno pojedynczego pliku, jak i całości. W przypadku przekroczenia zarezerwowanego miejsca DUMPER usuwa pliki z najstarszymi zrzutami. Poprzez ograniczenie wymagań przestrzeni dyskowej częściowo zrealizowany został postulat używania akceptowalnych zasobów na działanie samego mechanizmu. Ustalany limit miejsca na dysku nie zmniejsza użyteczności samego mechanizmu, gdyż podczas tropienia błędów w systemie rozproszonym po jego awarii zwykle interesuje nas tylko ostatnie kilka godzin, minut z jego działania, nie zaś zapis jego działania od samego początku.

Szczegółowy projekt nagrywania

Na początku działania komponentu DUMPER zapisywane są metadane zrzutu. Następnie zaś podczas funkcjonowania samego węzła systemu rozproszonego zapisywane są migawki z jego działania opierając się na cyklicznej interakcji z interfejsem informacyjnym INFO INTERFACE. Konkretnie, każde zakończone powodzeniem żądanie uzyskania informacji o węźle powoduje zapis jej binarnej reprezentacji wraz z jej długością oraz czasem zapisu do pliku na dysku. Żądanie zakończone niepowodzeniem nie powoduje żadnej akcji. W fazie projektowania uznaliśmy, że taka procedura zrzutu stanu węzła będzie wystarczająca i żadne optymalizacje na tym poziomie nie są potrzebne tzn. komponentowi PLAYER wystarczy sekwencyjne czytanie zrzutów nawet przy dalekich skokach na osi czasu. Jednakże, jak pokazały późniejsze testy, było to zbyt optymistyczne założenie i należało, modyfikując procedurę zarówno zapisu, jak i interpretacji migawek, wyraźnie przyspieszyć zaprojektowany mechanizm. Jednak dokładniej pomysł optymalizacji i szczegóły jej realizacji zostaną zaprezentowane w rozdziale 3.6.1.

Realizacja wymagania wykorzystania mechanizmu na żądanie

Skoro jednym z wymagań systemu było wykorzystanie mechanizmu na żądanie, postanowiliśmy funkcjonalność komponentu DUMPER zawrzeć w bibliotece dynamicznej. W rozwijanym przez nas systemie mieliśmy zaimplementowane wsparcie dla dynamicznie ładowanych bibliotek o ustalonym interfejsie między nią a węzłem systemu, więc integracja komponentu DUMPER z zachowaniem wymagania wykorzystania mechanizmu wizualizacji na żądanie była dosyć łatwa. Jednakże w innych systemach można by było osiągnąć ten sam efekt poprzez np. warunkową kompilację i linkowanie, tzn. aby możliwe było skorzystanie z mechanizmu należałoby wlinkować kod komponentu DUMPER w kod wizualizowanego systemu. Inną metodą jest na przykład konsolidacja kodu komponentu DUMPER na stałe z binariami systemu i aktywowanie oferowanej funkcjonalności poprzez dodatkowy argument z wiersza poleceń, czy to przez operację na interfejsie informacyjnym INFO INTERFACE.

```
class Player
{
public:
    class Position
    {
    public:
        Sample const * getSample();

        //other methods
    };

    /**
     * Sets the current position to the one given by the parameter.
     */
    void setPosition(Position const & argPosition);

    /**
     * Returns current position.
     */
    Position getCurrentPosition();

    /**
     * Looks up for the closest sample to the time specified.
     * @note Does not change the current position.
     */
    Position find(Time const & time);

    /**
     * Advances current position i.e. goes to the next sample.
     */
    void advanceCurrentPosition();

    //other methods
};
```

Rysunek 3.7: Szkielet interfejsu komponentu PLAYER

3.5.3. PLAYER

Interfejs

Funkcjonalność komponentu PLAYER zawiera:

- manipulowanie aktualną pozycją na osi czasowej,
- wyszukiwanie migawki najbliższej określonego czasu t,
- pobieranie wartości migawki znajdującej się na aktualnej pozycji,
- informowanie FAKE REMOTE INFO INTERFACE o zmianach aktualnej pozycji.

Rys. 3.7 prezentuje szkielet interfejsu komponentu PLAYER. Najważniejsze operacje to:

- `advanceCurrentPosition` – wywołuje przesunięcie do kolejnej migawki. Częstotliwość wywoływania tej operacji definiuje szybkość wizualizacji;

- `find` i `setPosition` – pierwsza operacja znajduje pozycję w ciągu migawek leżącą najbliżej danego punktu w czasie, druga zaś ustawia bieżącą pozycję. Użycie tych operacji jest niezbędne do implementacji skoków po osi czasowej;

- `Position::getSample` – przekazuje migawkę odpowiadającą danej pozycji. Migawka jest zawsze przeznaczona tylko do odczytu.

Interpretacja plików z migawkami

Jako że zapisywanie migawek przez komponent DUMPER odbywa się lokalnie na każdym węźle, PLAYER ma do dyspozycji zbiór ciągów migawek zapisanych w oddzielnych plikach. Wymagania funkcjonalne komponentu PLAYER wymagają poruszania się po linearnej strukturze migawek posortowanych po czasie jej zapisu. Ważne jest również wymaganie wydajnościowe, gdyż wizualizacja ma być praktycznie wykonywalna nawet wtedy, gdy pliki ze zrzutami są bardzo duże. Potrzebny jest zatem sposób efektywnego poruszania się po zbiorze ciągów migawek. Aby to osiągnąć, zaplanowaliśmy, że komponent PLAYER będzie przechowywał wskaźniki do aktualnych pozycji w każdym zapisanym zrzucie, z których jeden będzie jednocześnie aktualną pozycją w ciągu wszystkich migawek.

Najprostsze rozwiązanie scalenia wszystkich ciągów migawek na początku fazy odtwarzania nie zostało wybrane, gdyż wprowadzałoby to często niepotrzebny narzut na inicjowanie wizualizacji. Postanowiliśmy zatem, że pliki ze zrzutami reprezentujące zapisane migawki będą wirtualnie scalane w locie na żądanie użytkownika. Zauważmy, że to powoduje amortyzację kosztu scalenia ciągów migawek poprzez rozłożenie go na czas całej wizualizacji. Dzięki temu uzyskujemy również natychmiastową inicjację samej wizualizacji. Dla przykładu więc implementacja metody `advanceCurrentPosition` będzie zgodnie z oczekiwaniami znajdować najmłodszą migawkę spośród następujących po migawkach wskazywanych przez aktualne pozycje w każdym z ciągów.

Aby zredukować liczbę dostępów do dysku zaplanowaliśmy, że każdy podmoduł komponentu PLAYER, zajmujący się zarządzaniem jednym ciągiem migawek, będzie utrzymywał pamięć podręczną złożoną z kilku migawek przeczytanych z wyprzedzeniem. Pamięć ta będzie zawierać migawki bezpośrednio następujące po aktualnej i mieć stały rozmiar (np. 5). Istotnie zyskujemy przy takim podejściu podczas wizualizacji bez zbyt częstych skoków po osi czasowej (każdy skok unieważnia pamięć podręczną), gdyż kilkakrotnie zmniejszamy liczbę dostępów do dysku.

W pierwszej próbie postawiliśmy na prostotę zrzutu i interpretacji zapisanych migawek (por. 3.5.2). Skoro pliki ze zrzutem zawierały jedynie sekwencję trójek (stempel czasowy migawki, rozmiar migawki, binarna reprezentacja migawki), PLAYER był skazany na niepotrzebne liniowe przeglądanie wszystkich elementów tego ciągu trójek w celu lokalizacji poszukiwanego elementu w trakcie operacji `find` i `setPosition`. Rzeczywiście nasze nadzieje, że takie podejście będzie wystarczające było zbyt optymistyczne. Należało zoptymalizować interpretację zapisanych zrzutów również poprzez zmiany w procedurze zapisu migawek w fazie nagrywania. Jest to przedmiotem p. 3.6.1.

Współpraca z modułami klienckimi

Komponent PLAYER współpracuje bezpośrednio z komponentem FAKE REMOTE INFO INTERFACE w zakresie dostarczania danych migawek oraz z komponentem PLAYING CONTROLLER, który używa operacji interfejsowych w celu sterowania czasem i szybkością wizualizacji. Jednakże interakcja w każdym z przypadków została zaprojektowana w inny sposób.

Interakcja z FAKE REMOTE INFO INTERFACE Fałszywa implementacja interfejsu INFO INTERFACE, której projekt jest opisany szerzej w p. 3.5.4, ma za zadanie, zamiast dostarczać komponentowi VISUALIZER informacji o rzeczywiście istniejących węzłach w systemie, "karmić" go informacjami indukowanymi z migawek otrzymanych od komponentu PLAYER. Jednakże nasz projekt organizuje to nieco inaczej, gdyż postanowiliśmy, że implementacja FAKE REMOTE INFO INTERFACE nie będzie za każdym razem, gdy przyjdzie zapytanie od komponentu VISUALIZER, prosić komponent PLAYER o informacje na temat aktualnego stanu danego węzła. Zamiast tego implementacja FAKE REMOTE INFO INTERFACE będzie składować w sobie aktualne informacje o wszystkich węzłach i aktualizując sobie je tylko wtedy, gdy zmieni się bieżąca pozycja w komponencie PLAYER (używając metody `Position::getSample`). FAKE REMOTE INFO INTERFACE będzie wówczas informowany o zdarzeniu zmiany aktualnej pozycji w ciągu migawek poprzez specjalne funkcje (w języku angielskim tzw. *upcalls*). W konsekwencji każde wywołanie `setPosition` czy `advanceCurrentPosition` oprócz zmiany pozycji będzie informować wszystkich klientów zainteresowanych zmianą pozycji o takim zdarzeniu. Tak zresztą jest zobowiązany postępować dowolny inny potencjalny klient komponentu PLAYER zainteresowany zmianami pozycji. Musi on wtedy zaimplementować metody interfejsu zaprezentowane na rys. 3.8.

```
class PositionChangedInterface
{
public:
    void setPositionCalled(Position const & newPos) = 0;

    void advancePositionCalled(Position const & newPos) = 0;
};
```

Rysunek 3.8: Interfejs do powiadamiania o zmianie aktualnej pozycji w ciągu migawek

Interakcja z komponentem PLAYING CONTROLLER Współpraca polega tu na tym, że PLAYING CONTROLLER wywołuje interfejsowe operacje komponentu PLAYER w celu manipulowania aktualną pozycją wizualizacji. Komunikacja ta jest więc jednostronna.

3.5.4. FAKE REMOTE INFO INTERFACE

Komponent FAKE REMOTE INFO INTERFACE implementuje INFO INTERFACE. Musi zatem w sposób asynchroniczny odpowiadać na żądania klientów interfejsu (np. komponentu VISUALIZER) dotyczące stanu danego węzła. Źródłem tych informacji jest jednak PLAYER, który zarządza zapisanymi migawkami. Jednak zgodnie z ogólnym schematem współpracy modułów klienckich komponentu PLAYER (por. p. 3.5.3) FAKE REMOTE

INFO INTERFACE nie komunikuje się z nim przy każdym żądaniu komponentu VISUALIZER. Zamiast tego musi implementować dodatkowe funkcje, wywołując które PLAYER będzie informował go o każdej zmianie pozycji. Metody te muszą więc pojawić się w FAKE REMOTE INFO INTERFACE jako nowe operacje, które są implementacją interfejsu `PositionChangedInterface`. Implementacja tych metod musi z grubsza uaktualnić informacje komponentu FAKE REMOTE INFO INTERFACE o stanie węzłów. Dla przykładu implementacja `setPositionCalled` musi pobrać od komponentu PLAYER aktualną migawkę (za pomocą funkcji `PlayerInterface::getCurrentPosition()` i potem `Postion::getSample()`), odnaleźć w swoich strukturach węzeł, którego ta migawka dotyczy i zaktualizować przechowywane informacje na jego temat. Szkielet interfejsu FAKE REMOTE INFO INTERFACE prezentuje rys. 3.9.

```
class FakeRemoteInfoInterface:
    public InfoInterface,
    public PositionChangedInterface
{
public:
    void sleep(TimeDuration const & period);

    //InfoInterface methods
    void submitRequestGetNodeInformation(
        NodeAddress targetNode,
        UserRequestCallback *callback,
        TimeDuration const & timeout
    );

    //others
private:
    friend class Player;

    //PLAYER upcalls, can be called by player only
    void setPositionCalled(Position const & newPos);

    void advancePositionCalled(Position const & newPos);
};
```

Rysunek 3.9: Szkielet FAKE REMOTE INFO INTERFACE

Realizacja żądania komponentu VISUALIZER

Komponent VISUALIZER zleca żądania w celu uzyskania informacji o stanie danego węzła. Algorytm jego realizacji wygląda następująco:

1. FAKE REMOTE INFO INTERFACE otrzymuje asynchroniczne żądanie;

2. FAKE REMOTE INFO INTERFACE markuje to żądanie do wykonania we własnym wątku i zwraca sterowanie wywołującemu;

3. zakolejkowane żądanie jest wykonywane: FAKE REMOTE INFO INTERFACE sprawdza, czy posiada wymaganą informację; jeśli tak to wywołuje metodę `requestCompleted` w obiekcie funkcji zwrotnej komponentu VISUALIZER, w przeciwnym przypadku wywoływana jest funkcja `requestTimedOut` w celu zasymulowania przekroczenia dozwolonego czasu przeznaczonego na wykonanie żądania.

Podkreślmy raz jeszcze, że FAKE REMOTE INFO INTERFACE podczas realizacji żądania użytkownika nie komunikuje się bezpośrednio z komponentem PLAYER, gdyż mechanizm powiadamiania o każdej zmianie pozycji gwarantuje, że FAKE REMOTE INFO INTERFACE ma zawsze najbardziej aktualną informację o stanie węzłów.

Implementacja funkcji sleep

Interfejs INFO INTERFACE zawiera specjalną funkcję `sleep`. Zostanie ona użyta do możliwości zachowania jednolitej implementacji części komponentu VISUALIZER, która służy do uzyskiwania informacji na temat węzłów. Jednolitej, gdyż chcemy, aby VISUALIZER działał poprawnie bez żadnych modyfikacji w warstwie używania implementacji interfejsu informacyjnego zarówno podczas obserwacji działającego systemu, jak i w trybie wizualizacji post mortem.

```
while (this->stopRequested() == false)
{
    //send the requests to obtain nodes info

    //then invoke sleep() method
    this->infoInterface->sleep(this->retrievalInterval);
}
```

Rysunek 3.10: Szkielet programu wątku odpytującego węzły o ich stan (komponent VISUALIZER)

Będziemy zakładać, że w komponencie VISUALIZER będzie tylko jeden wątek odpytujący węzły poprzez interfejs INFO INTERFACE. Będzie on w pętli (patrz rys 3.10) najpierw zlecać żądania do węzłów, następnie zaś wywoływać funkcję `sleep` z parametrem, który definiuje częstotliwość odpytywania. Implementacja metody `sleep` na węzłach systemu oraz w REMOTE INFO INTERFACE będzie powodować bezwarunkowe zablokowanie wątku wywołującego na czas określony przez parametr `time`. Implementacja tej funkcji w FAKE REMOTE INFO INTERFACE będzie powodować zablokowanie wątku dopóki nie zostanie zasygnalizowana zmiana aktualnej pozycji, a parametr `time` jest tu ignorowany.

Dzięki tak zdefiniowanej semantyce funkcji `sleep` podczas wizualizacji działającego systemu dedykowany wątek komponentu VISUALIZER będzie periodycznie odpytywał węzły o ich stan, a pozostały czas spędzał "śpiąc" w funkcji `sleep`. Jednocześnie zapewni to możliwość kontroli wątku odpytującego w trybie wizualizacji post mortem. Istotnie FAKE REMOTE INFO INTERFACE będzie "budzić" wątek odpytujący zgodnie z polityką wizualizacji określoną przez interakcję z użytkownikiem. Dla przykładu, kiedy użytkownik chce zastopować wizualizację, wtedy PLAYING CONTROLLER nie spowoduje przesunięcia aktualnej pozycji w komponencie PLAYER, więc w konsekwencji wątek odpytujący "śpiący" w funkcji `sleep` nie zostanie obudzony. Realizacja sterowania szybkością wizualizacji będzie oparta na odpowiednio częstym "budzeniu" wątku odpytującego pośrednio przez komponent PLAYING

CONTROLLER. Pośrednio, gdyż bezpośredniej interakcji pomiędzy komponentem PLAYING CONTROLLER a komponentem VISUALIZER nie będzie. Jest to jednak wymuszona reakcja, gdyż kiedy PLAYING CONTROLLER wywoła metodę komponentu PLAYER, która zmienia aktualną pozycję "głowicy na taśmie", ten poinformuje komponent FAKE REMOTE INFO INTERFACE o zaistniałej zmianie. Reakcją na tę informację w komponencie FAKE REMOTE INFO INTERFACE będzie zaś obudzenie wątku odpytującego komponentu VISUALIZER. Szczegółów synchronizacji dostarcza p. 3.5.7.

3.5.5. PLAYING CONTROLLER

Projekt komponentu PLAYING CONTROLLER zawiera w sobie plan implementacji elementów graficznych (przycisków tj. `PLAY`, `PAUSE`, `FAST FORWARD` oraz suwaka pozwalającego poruszanie się po osi czasowej). Poza tym należy związać z nimi poprawną obsługę zdarzeń z nimi związanych. Ta zaś polega na interakcji z komponentem PLAYER w celu przesunięcia aktualnej pozycji w plikach ze zrzutami i poprzez to pośrednio kontroli wątku odpytującego w komponencie VISUALIZER (por. 3.5.6). Dla przykładu naciśnięcie przycisku `PLAY` powinno poskutkować cyklicznym wywołaniem metody `advanceCurrentPosition` z interfejsu komponentu PLAYER w celu przesunięcia się do kolejnej migawki zrzutu oraz odblokowaniem wątku wysyłającego żądania o stan węzłów w komponencie VISUALIZER.

3.5.6. VISUALIZER

W naszym systemie mieliśmy już gotowe narzędzie wizualizujące rozwijany przez nas system rozproszony w trakcie jego działania. Zasada jego działania opierała się na periodycznym odpytywaniu podzbioru węzłów systemu o ich stan, gdzie każde zakończone żądanie zwykle wpływało na odświeżenie widoku.

Jednakże nie było ono przystosowane do pracy w trybie wizualizacji systemu po awarii, gdyż w szczególności przyjmowało bardzo słuszne jak na zwykłe narzędzie do wizualizacji założenia, że czas się nie cofa lub na swój własny sposób prowadziło politykę częstotliwości odpytywania węzłów systemu o ich stan. To musiało się zmienić, aby VISUALIZER był użyteczny w kontekście realizacji mechanizmu wizualizacji post mortem.

Należało więc dokonać następujących modyfikacji w komponencie VISUALIZER:

- usunąć możliwie najwięcej założeń o linearności czasu i stanowości monitorowania. Ideałem byłoby tu bezstanowe narzędzie monitorujące odporne na zmianę stempli czasowych otrzymywanych zdarzeń. Jednakże w naszym przypadku takich założeń było niewiele, więc rzeczywiście koszt tych modyfikacji był niewielki;

- dodać elementy widoku pozwalające na sterowanie czasem podczas wizualizacji post mortem. Elementy te (w naszym przypadku jest to suwak z osią czasu oraz przyciski uruchamiające, pauzujące i przyspieszające wizualizację) są bezpośrednio związane z akcjami komponentu PLAYING CONTROLLER;

- wpłynąć na politykę częstotliwości odpytywania węzłów o ich stan poprzez zapewnienie, że schemat odpytywania węzłów o ich stan jest zgodny z założeniami przedstawionymi w 3.5.4. W naszym przypadku obyło się bez żadnych zmian poprzez wykorzystania różnych semantyk metody `sleep` w obydwu prawdziwych oraz fałszywej implementacji interfejsu FAKE REMOTE INFO INTERFACE.

Zauważmy, że modyfikacje te nie dotykały warstwy samej wizualizacji systemu rozproszonego. Oczywiście zmiany uodporniające komponent VISUALIZER na możliwość otrzymania

zdarzeń z przeszłości wpłynęły na warstwę interpretacji samych zdarzeń. Również należało dodać suwak czasowy i przyciski wraz z implementacją obsługi zdarzeń na tych elementach widoku.

3.5.7. Synchronizacja wizualizacji

Delikatną kwestią, na która należało zwrócić uwagę była synchronizacja komponentów PLAYER, PLAYING CONTROLLER, FAKE REMOTE INFO INTERFACE i VISUALIZER podczas wizualizacji. Zwróćmy uwagę, że ich współpraca musiała zapewniać, że:

- żadne żądanie komponentu VISUALIZER nie jest obsługiwane, gdy FAKE REMOTE INFO INTERFACE aktualizuje swoje informacje na temat węzłów po zmianie pozycje głowicy na taśmie,
- FAKE REMOTE INFO INTERFACE nie może zaktualizować informacji o stanie węzłów w wyniku zmiany aktualnej pozycji głowicy, gdy jakiekolwiek żądanie komponentu VISUALIZER jest obsługiwane.

Spełnienie tych warunków było kluczem do zapobiegnięcia wyścigów pomiędzy wątkiem obsługującym żądania komponentu VISUALIZER wewnątrz FAKE REMOTE INFO INTERFACE a wątkiem komponentu PLAYING CONTROLLER wywołującego metody przesunięcia w komponencie PLAYER. Przykładowy scenariusz prowadzący do sytuacji wyścigu (z ang. *race condition*) w przypadku braku zapewnienia powyższych asercji jest następujący:

1. VISUALIZER zleca żądanie uzyskania informacji o węźle X,
2. PLAYING CONTROLLER wywołuje metodę `advanceCurrentPosition` w celu przejścia do następnego zdarzenia
3. PLAYER przesuwa głowicę o jedną migawkę do przodu i informuje o zmianie pozycji komponent FAKE REMOTE INFO INTERFACE (w sposób synchroniczny w kontekście komponentu PLAYING CONTROLLER),
4. FAKE REMOTE INFO INTERFACE aktualizuje sobie dane o węźle X na podstawie aktualnej migawki (ale ciągle w kontekście komponentu PLAYING CONTROLLER), podczas gdy w tym samym czasie wątek FAKE REMOTE INFO INTERFACE obsługuje żądanie zlecone przez komponent VISUALIZER!

Aby spełnić powyższe warunki wprowadziliśmy następującą synchronizację:

- implementacja funkcji `sleep` w komponencie FAKE REMOTE INFO INTERFACE zapewnia, że wszystkie żądania komponentu VISUALIZER są zakończone zanim wątek rzeczywiście zostanie uśpiony (sytuacja zagłodzenia nie jest możliwa, gdyż zakładamy, że tylko wątek odpytujący komponentu VISUALIZER, w którego kontekście metoda `sleep` jest wywoływana, zleca żądania wywołując funkcje interfejsowe komponentu FAKE REMOTE INFO INTERFACE);
- implementacja metod interfejsu `PositionChangedInterface` zapewnia, że aktualizacja danych o węzłach w związku ze zmianą aktualnej pozycji poczeka, aż wątek odpytujący komponentu VISUALIZER zaśnie (czyli również wszystkie żądania zostaną zakończone).

3.6. Implementacja

W rozdziale tym zaprezentuję kilka ciekawych aspektów implementacji zaprojektowanego mechanizmu wizualizacji systemu rozproszonego post mortem.

3.6.1. Optymalizacja interpretacji zapisanych migawek

Jak udowodniły testy prototypu naszego mechanizmu, odczuwalne stały się kłopoty z wydajnością dalekich skoków na osi czasowej podczas wizualizacji. Brak dodatkowej struktury nad sekwencyjnym ciągiem migawek (por. 3.5.3) powodował niemożność implementacji dalekich skoków w inny sposób niż liniowe przechodzenie wszystkich migawek aż do napotkania tej z pasującym kluczem.

Postanowiliśmy więc zbudować strukturę indeksu migawek nad dotychczasowym ciągiem migawek. Jego idea jest następująca:

- indeks migawek będzie ciągiem par (klucz (stempel czasowy), przesunięcie w pliku ze zrzutem). Każda para ma stały rozmiar, więc nawet zapisując ten ciąg w pliku, który ma linearną strukturę, można znaleźć każdy klucz używając logarytmicznej liczby przesunięć w pliku (operacji `seek`). Proszę zauważyć, że nie można było tego samego zrobić dla oryginalnych plików ze zrzutem, gdyż migawki nie miały stałego rozmiaru;

- indeks będzie zapisywany już w trakcie fazy nagrywania. Rozwiązanie polegające na utworzeniu go na początku samej wizualizacji byłoby zbyt kosztowne w typowym przypadku;

- dla każdego pliku z migawkami mamy oddzielny plik indeksowy, gdyż przechowuje on przesunięcia w oryginalnym pliku ze zrzutem;

- indeks nie jest uaktualniany przy zapisie każdej migawki. Zamiast tego jest on akumulowany w pamięci i co każde n zapisów migawek do oryginalnego pliku ze zrzutem jest on wyrzucany z pamięci (z ang. *flush*) na dysk;

Zwróćmy uwagę, że narzut czasowy utrzymywania takiej struktury jest niewielki, dodatkowa pamięć zużyta na indeks jest również znikoma w porównaniu z rozmiarem plików z samymi migawkami.

Dzięki strukturze indeksowej implementacja skoków na osi czasowej mogła być przyspieszona, gdyż zamiast liniowego poszukiwania migawki z odpowiednim kluczem, mogliśmy wyszukać przesunięcie odpowiadające danemu stemplowi czasowemu za pomocą wyszukiwania binarnego w pliku indeksowym. Następnie używając jednej operacji przesunięcia wskaźnika otrzymywaliśmy żądaną migawkę.

Należało jeszcze zadbać o jeden istotny szczegół. Mogło się bowiem zdarzyć, że węzeł uległ awarii po zapisaniu pewnej liczby migawek, ale przez zapisem indeksu lub co gorsza w czasie zapisu pliku indeksowego. Jeszcze inny negatywny scenariusz, który należało obsłużyć to sytuacja, gdy mamy dostępny plik z migawkami a plik z indeksem jest niedostępny (np. został nieopatrznie usunięty czy uszkodzony). We wszystkich tych przypadkach postanowiliśmy, że PLAYER będzie odbudowywać indkeks. W przypadku braku pliku z indeksem odbudowa ta będzie bardzo bolesna, jednakże uznaliśmy, że jest to raczej przypadek rzadki. Natomiast w sytuacji, gdy nie zawiera on informacji o kilku ostatnich migawkach, odbudowa przebiega raczej szybko, gdyż jedyne co należy zrobić to uzupełnić indeks informacjami o k ostatnich migawkach, które zostały zapisane w pliku ze zrzutem, a nie zostały zapisane w pliku indeksowym.

3.6.2. Poprawność wyłączania komponentu DUMPER w trakcie działania systemu

Jednym z wymagań naszego mechanizmu było wykorzystania go tylko na żądanie. My rozszerzyliśmy definicję tego wymagania o możliwość wyłączenia mechanizmu w trakcie działania systemu. W naszym systemie oznaczało to deaktywację kodu komponentu DUMPER wraz z odładowaniem kodu samej biblioteki dynamicznej. Był to ciekawy problem z punktu widzenia implementacji w kontekście używanego przez komponent DUMPER asynchronicznego interfejsu INFO INTERFACE. Kod biblioteki dynamicznej DUMPER mógł być bowiem usunięty tylko wtedy, gdy wszystkie zlecone żądania przez komponent DUMPER zostały już zakończone przez implementację interfejsu INFO INTERFACE. W przeciwnym wypadku mogłyby wystąpić wołania do kodu biblioteki, który już został usunięty z pamięci procesu węzła systemu.

Pomysł na spełnienie tego warunku jest następujący:

- kiedy DUMPER otrzymuje żądanie, aby zakończyć pracę (bo np. węzeł również kończy pracę), to po pierwsze nie zleca żadnych nowych żądań;
- każdy obiekt funkcji zwrotnej przekazywany do żądania do implementacji INFO INTERFACE otrzymuje specjalny inteligentny wskaźnik, który potrafi liczyć liczbę referencji do niego samego. Jedną referencję do tego wskaźnika trzyma sam DUMPER, pozostałe są utrzymywane przez obiekty funkcji zwrotnej komponentu DUMPER. Przy takich założeniach kod komponent DUMPER może być odładowany, gdy licznik referencji tego wskaźnika spadnie do jedynki. Wtedy mamy pewność, że jego jedynym posiadaczem jest sam DUMPER.

3.7. Zakres modyfikacji samego systemu w celu wspierania mechanizmu wizualizacji post mortem

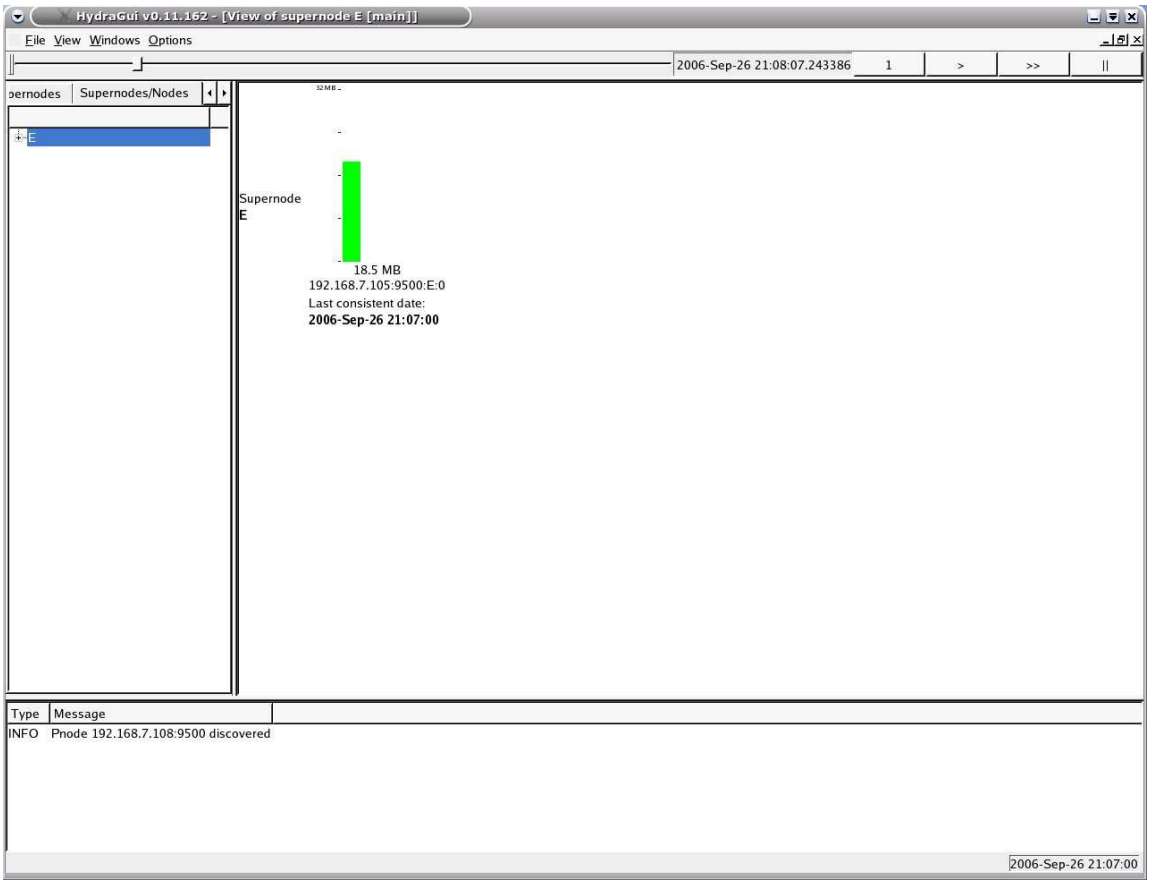
Istotnym wymaganiem względem projektowanego mechanizmu wizualizacji post mortem była relatywna prostota jego integracji zarówno z samym systemem, jak i istniejącymi wizualizatorami systemu. Zauważmy, że w kontekście modyfikacji kodu samego systemu (implementacja zapisu migawek), cel ten został w pełni osiągnięty, gdyż zrzucanie informacji o stanie systemu zostało zamknięte w jednym module, nie jest on porozrzucany po liczącym kilkaset tysięcy wierszy kodzie systemu. W kontekście samej wizualizacji należało w niewielki sposób tylko zmodyfikować narzędzie prezentujące graficznie działanie systemu, aby mogło być również użyteczne w czasie wizualizacji post mortem (por. p. 3.5.6), napisać komponent interpretujący zapisane zrzuty (co znalazło się również w oddzielnym module) oraz zaimplementować fałszywą inkarnację interfejsu INFO INTERFACE.

3.8. Narzut na działanie samego mechanizmu

Nie zaobserwowaliśmy spadku wydajności naszego systemu, gdy załadowaliśmy komponenty DUMPER na każdym z jego węzłów. Nie było to dla nas zaskoczeniem, gdyż komponent DUMPER nie wykonuje żadnych skomplikowanych obliczeniowo działań. Jest tak dlatego, że jego projekt stawia na maksymalną prostotę zapisu migawek (wprowadzenie indeksu jest tu koniecznym kompromisem). Praktyka pokazała również, że wydajność samej wizualizacji jest wystarczająca do skutecznego użytku naszego mechanizmu w celu graficznej obserwacji funkcjonowania systemu rozproszonego po awarii.

3.9. Prezentacja mechanizmu i jego osiągnięcia

Zaimplementowany przez nas mechanizm wizualizacji systemu rozproszonego post mortem przyczynił się do znalezienia wielu błędów. Inni członkowie naszego zespołu dostrzegli i szczególnie docenili jego wartość przy tropieniu konkretnych rodzajów błędów. Dla przykładu w naszym systemie węzły przetrzymują pewne mobilne komponenty, które oprócz migrowania na inne węzły mogą również się dzielić. Migracja lub podział mogą zajść przy jasno zdefiniowanych okolicznościach. Wizualizacja post mortem okazała się nieoceniona przy poszukiwaniu błędów, których przyczyną był np. niepotrzebny transfer komponentu z węzła na inny węzeł, czy błędna decyzja dotycząca podziału komponentu. Istotnie o wiele łatwiej zauważyć, że takie zdarzenie nieoczekiwanie zaszło obserwując graficzne umiejscowienie komponentów na węzłach niż przeszukując tej samej informacji w logach systemu. Graficzna reprezentacja położenia mobilnego komponentu na węźle systemu rozproszonego jest widoczna na rysunku 3.11.

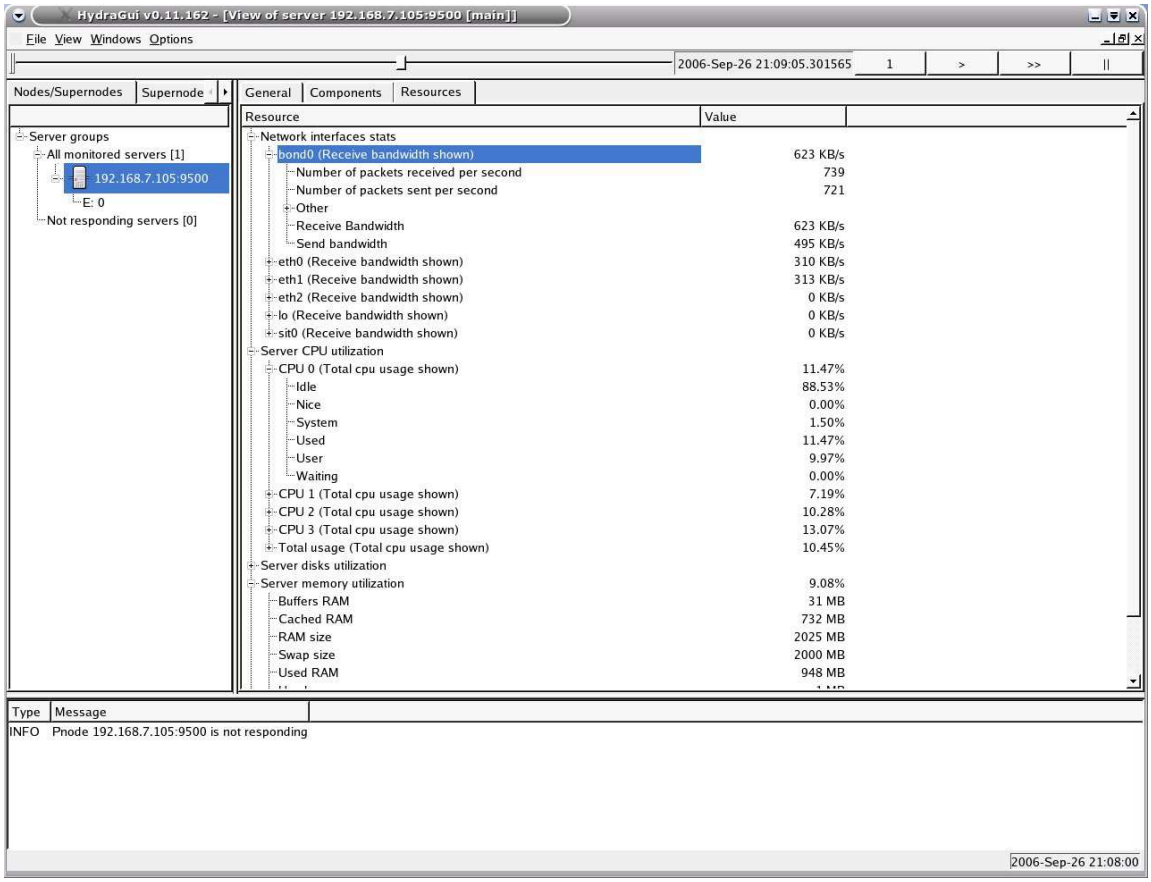


Rysunek 3.11: Widok mobilnego komponentu na węźle systemu rozproszonego

Wizualizacja naszego systemu po awarii okazała się również pożyteczna podczas implementacji maszyny stanowej na każdym węźle. Dla potrzeb monitorowania kondycji węzłów, oprócz tysięcy cząstkowych statystyk, musieliśmy wprowadzić kilka wysokopoziomowych stanów opisujących w jakiej fazie wykonywania swojej pracy węzeł się znajduje. Przejścia pomiędzy stanami są powodowane przez operacje wykonywane na węźle systemu. Dzięki mechanizmowi wizualizacji post mortem błyskawicznie udało się zauważyć, że węzeł znalazł się

w niepożądanym stanie.

Innym dowodem na użyteczność naszego mechanizmu jest fakt, że dzięki niemu w łatwy sposób można obserwować zużycie zasobów systemowych w czasie działania węzłów już po zakończeniu ich działania. Możemy również obserwować i śledzić zmiany naszych własnych statystyk dotyczących rozwijanego systemu. W konsekwencji udało nam się wykryć wiele błędów logicznych, jak również kłopoty wydajnościowe pewnych podjednostek węzła. Widok statystyk zużycia zasobów systemowych węzła obrazuje rysunek 3.12.



Rysunek 3.12: Widok statystyk wykorzystania zasobów systemowych węzła systemu rozproszonego

Na obydwu rysunkach (3.11 i 3.12) można dostrzec elementy widoku, które są związane ze sterowaniem czasem i szybkością wizualizacji post mortem. Istotnie pod paskiem menu głównego umiejscowiony jest suwak, który służy do manewrowania momentem rozpoczęcia lub kontynuowania wizualizacji (oczywiście skoki wstecz są również możliwe). Z prawej strony suwaka można zobaczyć przyciski umożliwiające, kolejno od lewej, przesunięcie się do kolejnej migawki, rozpoczęcie wizualizacji w zwykłym tempie (`PLAY`) lub w przyspieszonym (`FAST FORWARD`) i zatrzymanie wizualizacji (`PAUSE`). Między suwakiem a przyciskami można również dostrzec stempel czasowy migawki znajdującej się na aktualnej pozycji odgrywania.

Rozdział 4

Rozszerzenia mechanizmu wizualizacji post mortem

Pokazaliśmy już, jak uzyskać mechanizm wizualizacji funkcjonowania systemu rozproszonego po awarii. Uzasadniliśmy użyteczność zaproponowanego rozwiązania w tropieniu błędów w systemach rozproszonym. Okazuje się, że bazując na funkcjonalności (nie koniecznie całej) stworzonego mechanizmu można zrealizować inne pomysły, których celem jest jeszcze skuteczniejsze znajdowanie problemów w systemach rozproszonych. W tym rozdziale zaprezentuję jeden z nich, którego realizacją zajął się zespół, w którym pracowałem. W zarysie polega on na stworzenie mechanizmu do wykrywania nietypowych sytuacji w systemie rozproszonych, którego ideę prezentuje rys. 4.1.

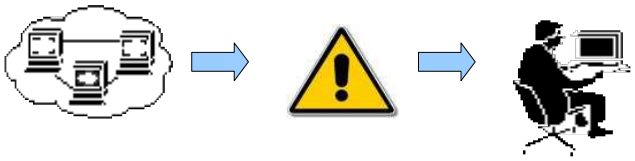
4.1. Wykrywanie anomalii systemu w czasie jego działania oraz po jego awarii

Tropiąc przyczynę błędu w systemie rozproszonym, badamy jego stan, w czasie gdy problem się pojawił, analizujemy przeróżne wskaźniki. I zwykle robimy to ręcznie, to znaczy oglądamy różne statystyki, analizujemy ich zmiany w czasie, próbując w ten sposób dociec, czy przypadkiem zachowanie systemu w którymś momencie nie odbiegało od normy. Pojawia się zatem pomysł, aby usystematyzować ten proces, aby wykrywać anomalie systemu automatycznie. Warto tu już podkreślić istnienie dwóch aspektów tego zagadnienia, mianowicie wykrywania dziwnych zachowań systemu w trakcie jego pracy oraz po jego awarii, kiedy system nie działa, a my chcielibyśmy się dowiedzieć, czy awaria ta nie była spowodowana wystąpieniem pewnych nietypowych zachowań systemu. Okazuje się, że można zaprojektować mechanizm, który, choć ortogonalny do pomysłu z nagrywaniem i odtwarzaniem działania systemu rozproszonego, daje się w prosty sposób z nim zintegrować, dając mechanizm działający dla obydwu aspektów wykrywania anomalii.

4.1.1. Wymagania względem projektowanego mechanizmu wykrywania anomalii

Przedstawię teraz wymagania funkcjonalne, jakie postawiliśmy sobie podczas projektowania systemu wykrywania odchyleń od normalnego działania. Uznaliśmy, że nasz mechanizm do wykrywania anomalii powinien:

- wykrywać nietypowe sytuacje w systemie rozproszonym w trakcie jego działania;



Rysunek 4.1: Idea wykrywania anomalii w systemie rozproszonym

- pozwalać na możliwość definiowania kryteriów wystąpienia nieprawidłowości. Wyznaczniki te nie mogą być zapisane trwale w kodzie źródłowym samego systemu;
- demaskować odchylenia od poprawnego działania systemu już po jego awarii. Dla nas oznacza to, że powinna być łatwo integrowalny z istniejącym systemem do odgrywania działania systemu rozproszonego;
- umożliwiać wielokrotne uruchomienie mechanizmu wizualizacji post mortem z różnymi kryteriami. Innymi słowy, to samo wykonanie może być kilkakrotnie testowane z innymi miarami "normalnego" działania (bez odchyleń).
- umożliwiać jego użycie na żądanie. Użytkownik powinien mieć łatwy sposób jego włączenia lub wyłączenia.

Nietrudno zauważyć, że oprócz samego mechanizmu, istotny jest tu również umiejętny projekt samych kryteriów dla praktycznego wykrywania odchyleń w systemie. Co nam bowiem po mechanizmie, który co prawda doskonale może służyć do ujawniania nietypowych zachowań systemu zarówno w czasie jego działania, jak i po awarii, ale kryteria są tak zdefiniowane, że system albo nic nie wykrywa, albo też generuje całą masę fałszywych alarmów. To spostrzeżenie każe dołożyć wszelkich starań, aby aplikowane miary "normalnego" działania systemu radziły sobie ze wskazanymi zagrożeniami.

4.1.2. Idea

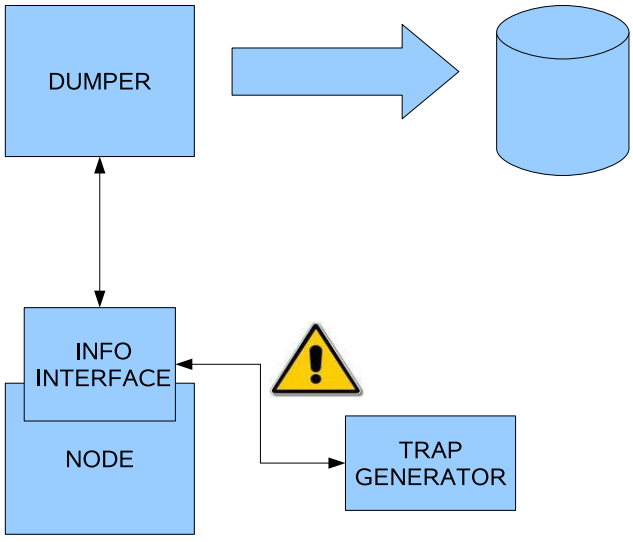
Wykrywaniem anomalii w naszym rozwiązaniu będzie zajmować się komponent nazwany TRAP GENERATOR. Jego zadaniem będzie analiza poprawności działania systemu na podstawie generowanych przez sam system statystyk pod kątem dostarczonych mu kryteriów "normalnego" funkcjonowania systemu. Wynikiem tej analizy ma być brak akcji, jeżeli kryteria sukcesu są spełnione oraz podniesienie alarmu (sprecyzujemy to pojęcie później) w przypadku naruszenia któregoś ze zdefiniowanych warunków. Schemat rozwiązania wydaje się prosty, gdyż komponent TRAP GENERATOR zostanie umieszczony na każdym węźle systemu rozproszonego i będzie odpytywał periodycznie lokalny dla siebie węzeł o zbiór statystyk dotyczących stanu węzła. Następnie TRAP GENERATOR będzie sprawdzać, jak otrzymane statystyki mają się do kryteriów sukcesu. W przypadku nie spełnienia któregoś z warunków TRAP GENERATOR będzie powiadamiał zainteresowanych, że oto w systemie zaistniała nietypowa sytuacja. Jednak ponownie, aby idea ta mogła znaleźć zastosowanie w praktyce, sam system musi spełnić dodatkowe warunki, aby projektowany mechanizm wykrywania anomalii miał szansę skutecznie zadziałać. System więc powinien:

- zbierać statystyki dotyczące swojego działania oraz umieć dostarczyć je żądającemu poprzez interfejs informacyjny (INFO INTERFACE);

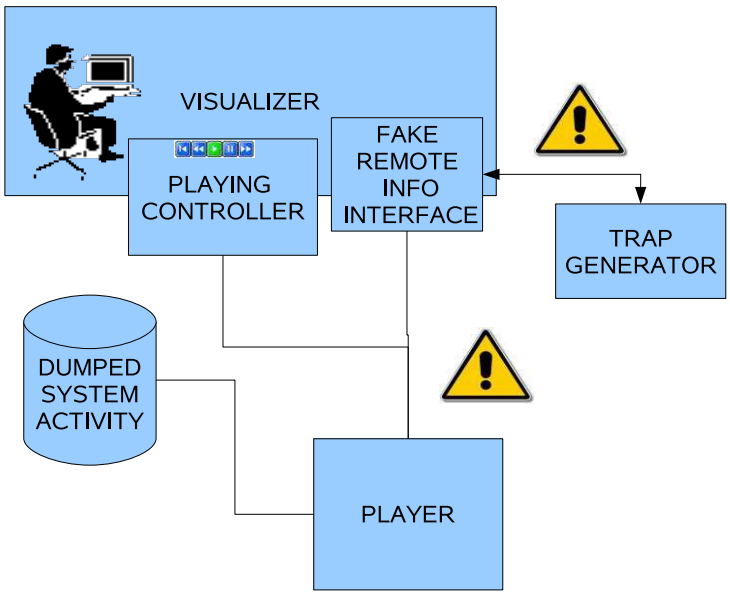
- posiadać mechanizm do zgłaszania zdarzeń w systemie oraz mechanizm rejestrowania się na powiadomienia dotyczące określonych zdarzeń np. przez oprogramowanie monitorujące. Możliwość zgłaszania zajścia zdarzenia jest wymagana, aby TRAP GENERATOR mógł zgłosić wystąpienie nietypowego zachowania systemu. Możliwość rejestrowania się na określone zdarzenia w systemie jest potrzebna, aby obserwator, który jest zainteresowany informacją o występowaniu pewnych zdarzeń, mógł zostać o tym powiadomiony. W naszym przypadku komponent VISUALIZER będzie rejestrował się na zdarzenia wystąpienie anomalii w systemie. W zależności od tego, czy obserwuje działający system, czy też pracuje w trybie odgrywania, będzie mógł uzyskiwać informacje na temat odchyleń systemu od normy.

4.1.3. Architektura

Niezależnie od rozważanego aspektu wykrywanie anomalii (w trakcie działania (rys. 4.2), czy już po awarii (rys. 4.3)) dominującym komponentem rozwiązania jest TRAP GENERATOR, gdyż to on będzie przeprowadzał analizę danych mu statystyk i ewentualnie zgłaszał zaistnienie nietypowych sytuacji w systemie. Jednakże, aby komponent ten miała szansę działać, musi on otrzymać wsparcie od innych komponentów wizualizacji systemu rozproszonego po awarii. Po kolei zatem opiszę, jak nowe komponenty w tym rozwiązaniu mają działać oraz jak istniejące muszą wspierać skuteczne wykrywanie anomalii.



Rysunek 4.2: Wykrywanie anomalii podczas rzeczywistego działania systemu (widok jednego węzła)



Rysunek 4.3: Wykrywanie anomalii podczas wizualizacji działania systemu post mortem

INFO INTERFACE

Interfejs informacyjny INFO INTERFACE będzie potrzebny komponentowi TRAP GENERATOR do dwóch celów. Po pierwsze musi udostępniać metody do uzyskania statystyk danego węzła. Tę funkcjonalność łatwo jednak wpleść w istniejący model interfejsu INFO INTERFACE poprzez dodanie nowej metody `submitRequestGetNodeStatistics`. Po drugie zaś INFO INTERFACE musi dać komponentowi TRAP GENERATOR możliwość zgłoszenia, że wystąpiło nietypowe zdarzenie w systemie. Równolegle system musi również dostarczać możliwość powiadamiania zainteresowanych, że takie zdarzenie nastąpiło. Zatem to, co należy jeszcze zrobić to dodać do INFO INTERFACE wsparcie dla zgłaszania zdarzeń w systemie oraz rejestrowania się na powiadomienia. Ponownie jak w przypadku możliwości wyciągania informacji z węzła systemu rozproszonego, tutaj również rozwiązanie nie wymusza drogi implementacji tej funkcjonalności. Jednak znów, aby przybliżyć możliwą metodę realizacji tych wymagań na rys. 4.4 zaprezentuję rozszerzenie interfejsu z rys. 3.4.

Przykładowo podana metoda `registerForTrapEvents()` ma być wołana przez komponenty VISUALIZER i DUMPER, aby móc dowiadywać się na temat odchyleń od normy w działaniu jednego węzła w systemie. Komponenty te będą robić inny użytek z uzyskanych w ten sposób informacjach o zajściu nietypowej sytuacji w systemie. Jest tak, gdyż DUMPER, będzie tę informację zapisywał jako jeden z rodzajów migawek, tak aby w czasie wizualizacji post mortem wystąpienie takiej aberracji zostało zasygnalizowane użytkownikowi. Komponent VISUALIZER będzie zaś wizualizował fakt, że węzeł zgłosił wystąpienie problemowej sytuacji. Metoda `trapEventOccurred()` będzie natomiast wołana przez komponenty TRAP GENERATOR i PLAYER w celu poinformowania implementacji interfejsu INFO INTERFACE, że nietypowa sytuacja w systemie nastąpiła i należy poinformować o tym zarejestrowanych na ten typ zdarzenia klientów.

```
class InfoInterface
{
public:
    // method to retrieve statistics of a node
    void submitRequestGetNodeStatistics(
        NodeAddress targetNode,
        UserRequestCallback *callback,
        TimeDuration const & timeout
    ) = 0;

    // other methods...

    // user's callback as processing is asynchronous
    class UserNotificationCallback
    {
        void eventOccurred(
            EventDescription const & eventDescription
            ) = 0;
    };

    // example method to register for certain events
    void registerForTrapEvents(
        NodeAddress targetNode,
        UserNotificationCallback *callback
    ) = 0;
    // other methods...

    //interface for components that report the events

    //example method for event creator
    void trapEventOccurred(
        EventDescription const & eventDescription
        ) = 0;
};
```

Rysunek 4.4: Rozszerzenie szkieletu interfejsu informacyjnego o mechanizm powiadomień

REMOTE INFO INTERFACE

Jedyne co należy zmodyfikować w komponencie REMOTE INFO INTERFACE to dodać implementację powiadomień i metody `submitRequestGetNodeStatistics`, która podobnie jak w przypadku samego mechanizmu wizualizacji post mortem może być dzielona z implementacją INFO INTERFACE na węzłach systemu rozproszonego.

DUMPER

Zauważmy, że jeżeli DUMPER i TRAP GENERATOR działają podczas rzeczywistej pracy systemu, to dobrze by było, aby DUMPER umiał oprócz stanu wyciąganego periodycznie

od węzła, zrzucać również informacje o wygenerowanych przez TRAP GENERATOR zdarzeniach wystąpienia nietypowej sytuacji w systemie. Wtedy podczas odgrywania PLAYER może raportować, że taka sytuacja miała miejsce.

VISUALIZER

VISUALIZER, chcąc wspierać wykrywanie anomalii, jedyne co musi zrobić, to zarejestrować się na zdarzenia ich wystąpienia i w przypadku ich pojawienia się, wizualizować ten fakt poprzez wyświetlenie jakiegoś ostrzeżenia, wysyłać list elektroniczny do administratora, czy zareagować w dowolny inny sposób. Zauważmy, że ponownie jak w przypadku naszej realizacji mechanizmu wizualizacji post mortem, dla komponentu VISUALIZER nie będzie wielkiej różnicy, czy działa w trybie współpracy z rzeczywistym, działającym systemem, czy też w trybie odgrywania. Wszystko ponownie załatwi podmiana biblioteki REMOTE INFO INTERFACE na FAKE REMOTE INFO INTERFACE, która zamiast obserwować "żywy" system, będzie "rozmawiać" z komponentem PLAYER.

FAKE REMOTE INFO INTERFACE

Komponent FAKE REMOTE INFO INTERFACE musi radzić sobie od teraz z rejestracją komponentu VISUALIZER na powiadomienia oraz sygnalizacją wystąpienia zdarzeń zarówno przez komponenty PLAYER, jak i TRAP GENERATOR. Można zapytać, dlaczego sygnały o zdarzeniach będą przychodzić z dwóch źródeł? Odpowiedź jest następująca: PLAYER będzie raportował o wystąpieniach nietypowych sytuacji na podstawie zapisanego przez komponent DUMPER zrzutu, natomiast TRAP GENERATOR będzie alarmować o napotkaniu anomalii sprawdzając statystyki, które otrzyma bezpośrednio od FAKE REMOTE INFO INTERFACE (a faktycznie od komponentu PLAYER) pod kątem spełnienia kryteriów dostarczonych w czasie odtwarzania.

PLAYER

Jedyne co nowego musi udostępniać PLAYER to umiejętność odczytania z zapisanego zrzutu zdarzeń o wystąpieniu odchylenia od normy (odchylenia te zostały stwierdzone na podstawie kryteriów ustalonych w trakcie działania rzeczywistego systemu, czyli fazy nagrywania) oraz zgłoszenie ich do komponentu FAKE REMOTE INFO INTERFACE.

TRAP GENERATOR

Kluczowym komponentem w naszym mechanizmie wykrywania aberracji w systemie rozproszonym jest TRAP GENERATOR. Jako, że zasada jego działania jest ortogonalna do aspektu wykrywania anomalii, nie będziemy rozgraniczać opisu na przypadek, gdy mamy do czynienia z rzeczywiście działającym systemem i przypadek, gdy współpracujemy z komponentem tylko odgrywającym jego działanie.

Kluczowym zadaniem komponentu TRAP GENERATOR jest analiza statystyk pobranych od implementacji interfejsu informacyjnego w celu wykrycia odchyleń od normy, definiowanej przez kryteria, które są parametrem.

4.1.4. Projekt

W tym rozdziale skupię się na projekcie komponentu TRAP GENERATOR, jako że nowa funkcjonalność pozostałych komponentów jest albo rozszerzeniem istniejącej opisanej już w p. 3.5 albo zaś jest w miarę prosta do zrozumienia bez pogłębiania szczegółów jej realizacji.

TRAP GENERATOR

Aby zadośćuczynić wymaganiu wykorzystania mechanizmu na żądanie identycznie jak w przypadku komponentu DUMPER zamkniemy całą jego funkcjonalność w bibliotece dzielonej (z ang. *shared library, shared object*). Będzie ona ładowana, gdy użytkownik chce, aby uaktywnić mechanizm wykrywania anomalii, zaś deaktywowana i odładowywana, gdy użytkownik z jakichś powodów nie chce już niego korzystać.

Kryteria oraz dodatkowe ich opcje będą przekazywane do głównej funkcji biblioteki dzielonej jako parametry uruchomienia. W ten sposób nie będą one zaszyte w kodzie komponentu TRAP GENERATOR, przez co osiągniemy nasz cel, aby móc wielokrotnie uruchamiać mechanizm wizualizacji post mortem, za każdym razem z innymi regułami zgłaszania zajścia nietypowych sytuacji w systemie.

Przykładowe funkcje opisujące kryteria wystąpienia nietypowej sytuacji w systemie

Zaprezentuję teraz kilka przykładowych elementarnych funkcji, które zaaplikowane do konkretnych statystyk mogą służyć jako użyteczne jednostkowe kryteria "normalnego" funkcjonowania systemu.

Weryfikacja statystyk typu logicznego W naszym systemie są wartości typu logicznego, czyli przyjmujące wartość prawdy lub fałszu. Ich wartości pozwalają stwierdzić, czy w systemie przebiega jakieś zadanie w tle, czy jest przepełniony, czy jakiekolwiek inne sensowne informacje o działaniu węzła, które dają się reprezentować przez zmienną logiczną. Łatwo sobie zatem wyobrazić cząstkowe kryterium, które będzie testować tę wartość i raportować, kiedy niepożądana wartość została napotkana. Wtedy np. można automatycznie wykrywać, że system przeprowadza jakieś działania, które nie powinny być podejmowane.

Weryfikacja statystyk typu licznikowego Nasz system posiada również tysiące statystyk, które są zwykłymi licznikami. Używane są one do zliczania np. długości kolejek w podmodułach, liczby komunikatów wysłanych, obsłużonych itd. Można zatem natychmiast wykorzystać ten fakt do zdefiniowania kryteriów walidacji takich statystyk. Można na przykład sobie wyobrazić użyteczną heurystykę, która testuje różnicę pomiędzy liczbą komunikatów zakolejkowanych a liczbą komunikatów obsłużonych przez podmoduł węzła systemu rozproszonego. W ten sposób ustalając górny dopuszczalny limit takiej wartości można w łatwy sposób automatycznie wykrywać moduły, które mają kłopoty z wydajnością lub po prostu uległy zakleszczeniu (wtedy ta różnica np. rośnie nieograniczenie).

Widać, że przykłady użytecznych i niezbyt skomplikowanych kryteriów można mnożyć. Pokazałem tylko kilka, aby pokazać jak proste funkcje zaaplikowane do konkretnych statystyk mogą dać zupełnie przyzwoite efekty w kontekście automatycznego wykrywania anomalii.

4.1.5. Realizacja wymagań względem mechanizmu wykrywania anomalii

Zaprezentuję teraz w jaki sposób zaproponowane rozszerzenie mechanizmu wizualizacji systemu rozproszonego post mortem realizuje wymagania postawione w p 4.1.1:

- zaprojektowany mechanizm pozwala na wykrywanie anomalii w czasie działania systemu rozproszonego. Komponent VISUALIZER obserwując węzły systemu, może zarejestrować się na otrzymywanie zdarzeń zajścia nietypowych sytuacji na podstawie pewnych kryteriów decyzji, kiedy taka sytuacja ma miejsce;

- reguły decydujące, kiedy problemowa sytuacja występuje, są modyfikowalne poprzez dostarczenie odpowiednich parametrów do komponentu TRAP GENERATOR;
- przy każdym uruchomieniem mechanizmu wizualizacji post mortem z nakładką wykrywającą aberracje można podać inny zestaw norm "normalnego" funkcjonowania systemu;
- funkcjonalność wykrywająca anomalie jest zamknięta w module dostarczanym jako biblioteka dynamiczna, ładowana bądź odładowywana na życzenie użytkownika;

4.2. Implementacja

Implementacja tego rozszerzenia nie jest gotowa i została odsunięta na późniejszy termin w związku z bardziej krytycznymi zadaniami w planie implementacji całego systemu.

Rozdział 5

Podsumowanie

W pracy zaprezentowałem problematykę skutecznego tropienia problemów w systemach rozproszonych. Wskazałem różne ich rodzaje, które można napotkać podczas pracy nad systemem rozproszonym i różnicę w stopniu skomplikowania w ich rozwiązywaniu w porównaniu do klasycznych programów jednozadaniowych. Następnie przedstawiłem istniejące podejścia do ułatwienia tropienia błędów w takich systemach skupiając się na rozwiązaniach graficznie prezentujących funkcjonowanie programów rozproszonych. Bazując na badaniach istniejących narzędzi w tej gałęzi oprogramowania i doświadczeniach zdobytych podczas obserwacji dotychczasowych praktyk poszukiwania błędów wraz z częścią zespołu, w którym pracowałem, postanowiliśmy zaprojektować i zaimplementować autorski mechanizm wizualizacji systemów rozproszonych po awarii w celu tropienia błędów. Mechanizm ten miał być w założeniu gotowy do użycia w kontekście rozwijanego przez nas systemu rozproszonego dla zastosowań komercyjnych. Jego działanie opiera się na zapisie migawek z funkcjonowania systemu i wizualizacji jego działania na podstawie zapisanego zrzutu. Narzędzie to powstało wysiłkiem kilkuosobowego zespołu, w którym mi przypadł udział w jego projektowaniu, implementacja i pielęgnowanie komponentów: zbierającego informacje o węźle i zapisującego je w postaci migawek na dysk (DUMPER) oraz fałszywej implementacji interfejsu informacyjnego (FAKE REMOTE INFO INTERFACE).

Prezentuję również możliwości rozszerzenia naszego mechanizmu poprzez przykład nadbudowanego nad nim systemu wykrywania sytuacji nietypowych w systemach rozproszonych. Przedstawiłem tu jego projekt korzystający z możliwości działającego wizualizatora naszego systemu. Opiera się on na pomyśle analizowania statystyk generowanych przez sam system pod kątem dostarczonych kryteriów decydujących o fakcie zajścia nietypowej sytuacji w systemie.

Projekt implementacji mechanizmu wizualizacji systemu rozproszonego na podstawie migawek w celu tropienia błędów zakończył się sukcesem. Stworzone narzędzie zostało włączone do produkcyjnej wersji dużego systemu rozproszonego. Ma być jednym z głównych źródeł informacji, co działo się w systemie, zanim uległ on awarii w środowisku klienta. Choć już teraz, w fazie rozwijania systemu, zyskało ono sobie grono chętnych użytkowników wśród programistów naszego zespołu, gdyż przyczyniło się do diagnozy wielu błędów. Możliwość szybkiej wizualizacji systemu na podstawie zrzuconych migawek zachęca do sięgnięcia po stworzone w tym celu narzędzie podczas tropienia nietrywialnych błędów systemu.

Nie bez wpływu na sukces tego przedsięwzięcia, oprócz oczywiście samej idei, były powzięte założenia co do mechanizmu wizualizacji. Efektywność implementacji zarówno nagrywania, jak i odgrywania, minimalny wysiłek programisty wkładany w użycie mechanizmu, wymagania co do zasobów systemowych spowodowały, że istotnie stworzony mechanizm jest

praktycznym narzędziem ułatwiającym tropienie błędów w systemach rozproszonych.

Dodatek A

Zawartość płyty CD

Na załączonej płycie CD znajdują się:

- strzelczak.pdf – ta praca magisterska w formacie PDF,
- screenshots/ – zrzuty ekranu prezentujące działające narzędzie wizualizujące system rozproszony post mortem,
- interfaces/ – szkielety interfejsów komponentów mechanizmu wizualizacji.

Bibliografia

[1] Gvu: Phread Visualization Package - Gthread. `http://www-static.cc.gatech.edu/gvu/softviz/parviz/gthread/gthread.html%`.

[2] Steve Carr, Changpeng Fang, Tim Jozwowski, Jean Mayo, and Ching-Kuang Shene. ConcurrentMentor: A visualization system for distributed programming education.

[3] Steve Carr, Jean Mayo, and Ching-Kuang Shene. Threadmentor: a pedagogical tool for multithreaded programming. *ACM Journal of Educational Resources in Computing*, 3(1), 2003.

[4] Colin J. Fidge. Fundamentals of distributed system observation. *IEEE Software*, 13(6):77–83, 1996.

[5] Jorge Garcia and Richard Hughey. Scalable visualization of parallel systems. Technical Report UCSC-CRL-94-29, 1994.

[6] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proc. of USENIX*, 2006.

[7] Joel Huselius. Debugging Parallel Systems: A State of the Art Report. Technical Report 63, Mälardalen University, Department of Computer Science and Engineering, September 2002.

[8] Eileen Kraemer. The Animation Choreographer. `http://www-static.cc.gatech.edu/gvu/people/Phd/Eileen.Kraemer/chore.htm%l`.

[9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 12(7):558–565, 1978.

[10] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Computers*, 36(4):471–482, 1987.

[11] Robert H. B. Netzer and Barton P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *SC*, pages 502–511, 1992.

[12] John T. Stasko. The PARADE environment for visualizing parallel program executions: A progress report. Technical Report GIT-GVU-95-03, 1995.

[13] Brad Topol, John T. Stasko, and Vaidy S. Sunderam. Integrating visualization support into distributed computing systems. In *International Conference on Distributed Computing Systems*, pages 19–26, 1995.