Systemy rozproszone

Modele spójności nastawione na klienta

Cezary Sobaniec
Jerzy Brzeziński

UCZELNIA ONLINE

Zwielokrotnianie

- Zwielokrotnianie (replikacja, ang. *replication*) – utrzymywanie wielu kopii danych na niezależnych serwerach
- Cele stosowania zwielokrotniania (replikacji)
  - zwiększenie **niezawodności** (w tym dostępności)
  - zwiększenie **efektywności**
- Problemy
  - wysokie koszty implementacji
  - spójność replik
  - mobilność

Jak wiadomo zwielokrotnianie (czyli replikacja) jest stosowane dla zwiększenia ogólnej niezawodności systemu i poprawy jego efektywności. Prace nad rozwojem systemów rozproszonej pamięci współdzielonej (DSM) zaowocowały powstaniem nowych modeli spójności, które były później efektywnie realizowane przez odpowiednie protokoły spójności. Systemy DSM są jednak dość szczególnym zastosowaniem koncepcji zwielokrotniania danych. W większości realizowane są bowiem jako systemy homogeniczne, pracujące w sieci szybkich i niezawodnych połączeń. Głównym powodem stosowania replikacji w tych systemach było dążenie do zwiększenia efektywności przetwarzania, umożliwiające wykorzystanie multikomputera tak jak systemu wieloprocesorowego. W podejściu tym zakłada się jednak, że poszczególne procesy odwołujące się do wspólnej pamięci są statycznie zlokalizowane w wybranych węzłach i nie zmieniają swojej lokalizacji. Coraz częściej jednak aplikacje w systemach rozproszonych stają się mobilne, głównie z powodu rozwoju nowoczesnych technologii telekomunikacyjnych. Modele spójności nastawione na klienta uwzględniają ten aspekt zarządzania spójnością danych.

Replikacja pesymistyczna

- Blokowanie dostępu do replik modyfikowanych obiektów
- Spójność kosztem dostępności
- Wymaga dobrej jakości, niezawodnych połączeń sieciowych (stała koordynacja pracy serwerów)
  - LAN → WAN → sieci mobilne (również *ad hoc*)
- Długie czasy odpowiedzi systemu
- Problem z podziałem sieci i pracą w trybie odłączonym (ang. *disconnected*)

Większość protokołów spójności omawianych na poprzednim wykładzie zakładała stałą łączność z wszystkimi pozostałymi węzłami w sieci i to łączność o dobrej jakości. Założenie to wynika pośrednio ze stosowanego często blokowania wykonywania operacji. Niedostępność węzłów lub opóźnienia komunikacyjne będą się w takim przypadku przekładały na blokowanie aplikacji odwołującej się do wspólnych danych. Blokowanie jest konieczne aby zagwarantować własności opisane w modelu spójności. W zależności od siły modelu spójności blokowanie aplikacji wynikające z realizowanych uzgodnień między serwerami może być częstsze lub rzadsze. W modelu sekwencyjnym niemal każda operacja będzie wymagała komunikacji z innymi węzłami, podczas gdy np. w spójności PRAM można tego blokowania prawie całkowicie uniknąć. Aplikacje rozproszone są jednak uruchamiane nie tylko w sieciach lokalnych, gdzie można zapewnić wysoką i stałą jakość połączeń sieciowych, ale i również w sieciach rozległych, a ostatnio również w sieciach mobilnych (korzystając z takich technologii jak GPRS, UMTS czy WiFi). Cechą charakterystyczną sieci mobilnych jest oprócz zmieniającego się punktu dostępu do sieci, również zmienna jakość połączeń, a nawet ich czasowy brak. Dla wielu aplikacji przestoje wynikające z braku łączności są nieakceptowalne; dużo bardziej pożądane byłoby odwołanie się do (prawdopodobnie) nieaktualnych danych niż wymuszanie oczekiwania. Modele spójności nastawione na dane dbają o zachowanie określonych uporządkowań operacji, ale nie gwarantują uzyskania wysokiej dostępności danych. Skrajnym przypadkiem złej jakości połączeń jest czasowe odłączenie od sieci, które w przypadku urządzeń mobilnych jest bardzo prawdopodobne. Użytkownicy oczekują, że pomimo braku łączności pewne operacje będą mogły być nadal wykonywane w systemie. Niestety protokoły oparte na blokowaniu są *pesymistyczne* w tym sensie, że spodziewając się powstawania niespójności w systemie blokują współbieżne przetwarzanie, co wymaga stałej łączności. Brak łączności oznacza niemożliwość kontynuacji przetwarzania.

Replikacja optymistyczna

- Brak synchronizacji *a priori*
- Możliwość wprowadzania współbieżnych modyfikacji tych samych obiektów
- Akceptacja modyfikacji na jednym węźle i późniejsza propagacja zmian
  - algorytmy epidemiczne
- Mechanizm wykrywania i rozwiązywania konfliktów
- Sieci mobilne: połączenia przerywane
- Gwarantuje jedynie spójność *ostateczną* (ang. *eventual consistency*)

Problemy związane z dostępnością danych można rozwiązywać stosując replikację optymistyczną. W podejściu tym nie stosuje się obligatoryjnego blokowania dostępu do danych przed wykonaniem operacji. W przeciwieństwie do replikacji pesymistycznej, zakłada się, że współbieżnie realizowane operacje nie będą w większości przypadków powodowały powstawania problemu spójności. W przypadku wielu aplikacji jest to założenie jak najbardziej prawdzie. Np. w rozproszonym systemie plików stosującym zwielokrotnianie użytkownicy modyfikują pliki, ale z reguły modyfikacje te dotyczą różnych plików. W efekcie tego partycjonowania danych współbieżnie wprowadzane zmiany mogą być przesłane do innych węzłów i tam wprowadzone. Aktualizacja taka może nastąpić z opóźnieniem, w momencie gdy łączność zostanie przywrócona. Strategia taka może oczywiście prowadzić do powstawania konfliktowych modyfikacji, jeżeli dwaj użytkownicy dokonają współbieżnie modyfikacji tego samego obiektu. Konflikty muszą być wykryte i obsłużone. Obsługa może polegać na automatycznym wyborze jednej z wersji lub na zgłoszeniu problemu wyższej warstwie (być może bezpośrednio użytkownikowi). Zakłada się, że konflikty będą rzadkie, a jeżeli wystąpią, to są akceptowalne przez użytkowników.

Stosując replikację optymistyczną nie można uzyskać żadnych twardych gwarancji w odniesieniu do uporządkowania operacji. Mówi się jedynie o spójności **ostatecznej** (ang. *eventual consistency*), która gwarantuje, że *ostatecznie* dane staną się spójne, ale bez żadnych wskazówek co do czasu kiedy to nastąpi.

Cechy replikacji optymistycznej

Zalety

- duża dostępność zasobów
- większa adaptacyjność do zmieniających się warunków pracy sieci
- większa skalowalność
- dla wielu aplikacji konflikty występują rzadko lub są akceptowalne przez użytkowników

Wady

- brak silnej spójności danych
- możliwość wystąpienia konfliktów

Podstawową zaletą replikacji optymistycznej jest duża dostępność zasobów. Dane są bowiem udostępniane bez względu na ich stopień aktualności. Takie ustawienie priorytetów jest jednak bardzo często oczekiwane przez aplikacje.

Drugą ważną cechą replikacji optymistycznej jest jej adaptacyjność do zmieniających się warunków: konfiguracji sieci, jakości połączeń itp. Adaptacyjność ta wynika w dużej mierze z elastycznych mechanizmów propagacji zmian, które próbują rozsyłać dane tam, gdzie jest to możliwe, nie podejmując próby komunikacji z wszystkimi węzłami w sieci. Brak łączności powoduje czasowe wstrzymanie propagacji, ale nie blokuje dostępu do danych. Efektem takiego podejścia jest też większa skalowalność tych protokołów. Wykorzystanie dużej liczby węzłów w sieci nie pogarsza istotnie jakości pracy algorytmu ponieważ konflikty występują rzadko.

Podstawową wadą replikacji optymistycznej jest brak mechanizmów zapewniania silnej spójności danych, które mogą być konieczne dla niektórych aplikacji. Drugą wadą są konflikty występujące przy dostępie do danych, które, przy pewnym ich natężeniu, mogą stać się uciążliwe dla użytkowników.

Modele spójności

Modele nastawione na dane

- modele spójności przy dostępie ogólnym
  - spójność atomowa, sekwencyjna, przyczynowa, PRAM, podręczna
- modele spójności przy dostępie synchronizowanym
  - spójność zwalniania, wejścia, zakresu

Modele nastawione na klienta

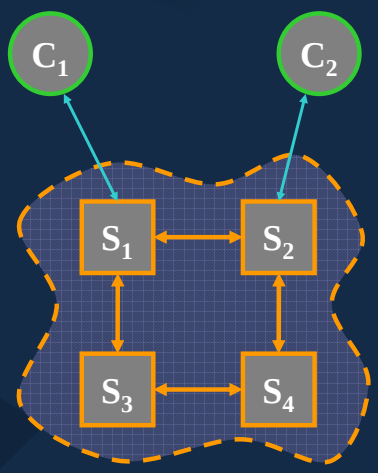
- uwzględniają możliwość migracji klienta
- obraz klienta i obraz serwera

Generalnie modele spójności zostały podzielone na dwie klasy. Modele spójności nastawione na dane (omówione na poprzednim wykładzie) zakładają, że procesy odwołujące się do rozproszonych, zwielokrotnionych danych są związane stale z jednym serwerem i nie przemieszczają się podczas przetwarzania. W niektórych realizacjach rozproszonej pamięci współdzielonej zarządzanie spójnością jest wręcz realizowane w samym procesie aplikacyjnym. Jeżeli jednak dopuścimy separację procesu aplikacyjnego od podsystemu zarządzania pamięcią i umożliwimy przemieszczanie się aplikacji, to modele spójności nastawione na dane (te przy dostępie ogólnym) przestaną spełniać swoją rolę. Dokładnie rzecz biorąc będzie tak dla wszystkich modeli oprócz atomowego, co ze względów efektywnościowych dyskwalifikuje tego typu rozwiązanie. Nieadekwatność modeli spójności nastawionych na dane wynika z tego, że przemieszczanie się klienta jest realizowane w sposób całkowicie niekontrolowany przez protokół spójności. Zlecenie zgłaszane do innego serwera jest interpretowane jako całkowicie niezależne od poprzednich zgłoszeń na innym serwerze, co oznacza, że wymagania dotyczące spójności danych mogą być inne.

Wprowadzenie mobilności klienta powoduje też, że mówiąc o spójności danych należy niezależnie rozpatrywać obraz historii przetwarzania serwerów i klientów.

Model systemu

- Zbiór niezawodnych serwerów, replikujących przestrzeń współdzielonych danych
- Mobilni klienci wykonują dostępy do współdzielonych danych łącząc się w ramach jednej **sesji** z wieloma serwerami
- Metody dzielą się na odczytujące i modyfikujące
- Ostateczna propagacja wszystkich modyfikacji



Dla potrzeb prezentacji modeli spójności nastawionych na klienta będziemy rozpatrywać system składający się ze zbioru niezawodnych serwerów, replikujących w pełni przestrzeń współdzielonych danych (obiektów). Klienci są oddzieleni od serwerów i mogą się miedzy nimi przełączać, co w praktyce oznacza, że są mobilni. Klient dołącza się do jednego serwera, zleca wykonanie serii operacji, a następnie przełącza się na inny serwer i kontynuuje przetwarzanie. Ciąg operacji zleconych różnym serwerom jest określany jako *sesja*. Od pojęcia sesji wywodzi się również oryginalna nazwa modeli spójności nastawionych na klienta: **gwarancje sesji** (ang. *session guarantees*).

Zlecenia klientów realizowane są w formie zdalnych wywołań metod na replikach współdzielonych obiektów. Metody obiektów dzielą się na dwie klasy: metody nie modyfikujące stanu obiektów (odczyty) i metody modyfikujące stan (zapisy).

Serwery kontaktują się ze sobą w celu propagacji zmian, które wprowadzają. Sposób organizacji propagacji zmian oraz ich uporządkowanie nie mają tutaj znaczenia. Ważne jest, że system gwarantuje **ostateczną propagację** wszystkich modyfikacji zleconych na dowolnym z serwerów do wszystkich pozostałych serwerów. Propagacja taka nie musi oznaczać jednak spójności ostatecznej (zobacz nast. slajd).

Spójność ostateczna

Spójność ostateczna (ang. *eventual consistency*) — o ile do systemu nie są zgłaszane nowe żądania modyfikujące, dane *ostatecznie* stają się spójne

W systemach stosujących zwielokrotnianie

- ostateczna propagacja wszystkich zapisów
- globalne uporządkowanie operacji nieprzemiennych
- determinizm przetwarzania

Przykład

- system DNS i unieważnianie pamięci podręcznych

W systemach rozproszonych stosujących zwielokrotnianie, w których zarządzanie spójnością nie jest tak rygorystyczne jak w przypadku systemów rozproszonej pamięci współdzielonej, możemy rozpatrywać **spójność ostateczną** jako najsłabszą gwarancję systemu oferującą podstawowy poziom spójności danych. W spójności ostatecznej gwarantuje się jedynie *ostateczną* spójność poszczególnych replik danych i to w sytuacji, gdy nie są już zgłaszane nowe modyfikacje. Ostatecznie czyli w zasadzie nie wiadomo kiedy – gdy upłynie dostatecznie dużo czasu na rozpropagowanie wszystkich modyfikacji do wszystkich serwerów.

Do osiągnięcia stanu spójnego może nie być wystarczające rozpropagowanie wszystkich modyfikacji w systemie. Jeżeli modyfikacje mają charakter przyrostowy (wykonywane są na obiektach), to trzeba dodatkowo zapewnić identyczne uporządkowanie wykonywanych operacji na wszystkich kopiach przy zachowaniu determinizmu przetwarzania. Problem ten nie istnieje w systemach, w których propagacji podlega *stan* obiektu po aktualizacji. W takich systemach wystarczające jest globalne ustalenie, która modyfikacja jest traktowana jako ostatnia.

Dobrym przykładem zastosowania spójności ostatecznej jest system DNS. W systemie tym stosuje się przechowywanie podręczne w pamięciach poszczególnych serwerów. Zawartość tych pamięci podręcznych ulega przedawnieniu po przekroczeniu zadanego przez administratora czasu. Po upływie tego czasu, po modyfikacji danych na serwerze głównym można więc przyjąć, że dane zostały rozpropagowane i są w stanie spójnym. W przypadku systemu DNS modyfikacje są dodatkowo wprowadzane zawsze w jednym miejscu, stąd nie istnieje problem konfliktowych zapisów.

Definicja nieformalna gwarancji sesji

Gwarancje sesji – modele spójności zorientowane na klienta, określające własności systemu, które zostaną zagwarantowane w odniesieniu do uporządkowania operacji zapisu zlecanych przez *pojedynczego migrującego* klienta

Modele spójności nastawione na klienta, zwane również gwarancjami sesji (ang. *session guarantees*), pozwalają na kontrolowanie spójności danych postrzeganych przez *pojedynczego*, *migrującego* klienta. „Pojedynczego”, ponieważ nie są udzielane *żadne* gwarancje w odniesieniu do uporządkowania operacji zlecanych przez różnych klientów. Uwzględniany natomiast jest fakt migracji, a więc przełączania się klienta pomiędzy serwerami. Zlecenie zapisu na jednym serwerze może być obserwowane na innym o ile zachowane będą wymagane przez klienta gwarancje sesji. Podejście to jest więc w pewien sposób ortogonalne w stosunku do modeli spójności nastawionych na dane. Z jednej strony bowiem uwzględnia fakt migracji klienta, a z drugiej nie umożliwia kooperacji klientów, jak to miało miejsce w przypadku modeli spójności nastawionych na dane.

Oznaczenia

r	operacja odczytu
	$r\|_{S_j}$ – odczyt wykonany przez serwer S_j
w	operacja zapisu
	$w\|_{S_j}$ – zapis wykonany przez serwer S_j
$\xrightarrow{C_i}$	porządek operacji zlecanych przez klienta C_i
$\xrightarrow{S_j}$	porządek operacji wykonywanych przez serwer S_j

Prezentacja gwarancji sesji będzie się odbywać z wykorzystaniem przedstawionej symboliki. Operacje typu odczyt czy zapis zapisywane z wykorzystaniem prostych oznaczeń *r* i *w*. W niektórych sytuacjach ważne będzie gdzie jest wykonywana dana operacja. Zostanie wtedy użyty dodatkowy indeks wskazujący na serwer realizujący operację.

Zakłada się, że zarówno klienci jak i serwery realizują operacje w sposób sekwencyjny: operacja po operacji. Historia przetwarzania serwerów czy ciągi zleceń poszczególnych klientów są więc porządkami liniowymi.

Read Your Writes (RYW)

- Użytkownik chce obserwować efekty swoich poprzednich modyfikacji

Definicja

$$\forall C_i \forall S_j \left(w \xrightarrow{C_i} r \,|_{S_j} \Rightarrow w \xrightarrow{S_j} r \right)$$

Przykład: rozproszony system plików — użytkownik chce widzieć wcześniejsze modyfikacje pliku, nawet jeżeli były wykonane na innym serwerze

Pierwszą gwarancją sesji, która zostanie zaprezentowana jest **odczyt własnych zapisów** (ang. *Read Your Writes — RYW*). Jest to najbardziej intuicyjna z gwarancji sesji. Opisując RYW nieformalnie: klient spodziewa się obserwować efekty swoich poprzednich zapisów po przełączeniu się do innego serwera. Oznacza to oczywiście, że modyfikacje, których dokonał na jednym z serwerów muszą zostać przesłane do nowego serwera przed zleceniem kolejnego odczytu. Na slajdzie znajduje się bardziej formalny zapis tego nieformalnego opisu. Dla każdego klienta i dla każdego serwera, jeżeli odczyt klienta C_i był poprzedzony zapisem, to przed zrealizowaniem tego odczytu na serwerze S_j, serwer ten powinien wykonać ten zapis. Warunek oczywiście aplikuje się do wszystkich zapisów, które wcześniej zlecił klient. Odczyt jest w tym przypadku oznaczony jako wykonywany konkretnie na serwerze S_j, ponieważ odczyty są realizowane tylko na jednym serwerze. W tym przypadku jest istotne, że jest to właśnie serwer S_j.

Przykładem zastosowania gwarancji RYW może być działanie wykonywane w rozproszonym, replikowanym systemie plików. Użytkownik dokonuje modyfikacji wybranych plików przy jednym komputerze, a następnie próbuje odczytywać te pliki pracując przy innym.

Monotonic Writes (MW)

- Użytkownik chce aby zapisy zlecane przez niego wszędzie były aplikowane w tej samej kolejności

Definicja

$$\forall C_i \forall S_j \left(w_1 \xrightarrow{C_i} w_2 \,|_{S_j} \Rightarrow w_1 \xrightarrow{S_j} w_2 \right)$$

Przykład: obiekt typu licznik z metodami dodaj(n) i ustaw(n). Wynik końcowy zależy od kolejności wywołań metod

`ustaw(10) → dodaj(2) → dodaj(1)`

Kolejna gwarancja to **monotoniczne zapisy** (ang. *Monotonic Writes — MW*). Nieformalnie można opisać tą gwarancję w ten sposób, że użytkownik chce wymusić globalne uporządkowanie *swoich* zapisów w systemie. Definicja formalna odwołuje się do dwóch zapisów w_1 i w_2, zleconych do wykonania właśnie w tej kolejności, przy czym drugi zapis jest wykonywany na serwerze S_j. Zanim jednak ten zapis zostanie wykonany, serwer S_j musi wykonać wcześniejszy zapis w_1. Warunek ten musi być spełniony dla każdego zapisu klienta C_i.

Przykładem zastosowania gwarancji MW może być seria odwołań do obiektu *licznik*, posiadającego dwie modyfikujące metody: *dodaj(n)* i *ustaw(n)*. Zakładając, że tylko jeden klient wywołuje metody tego obiektu, stan końcowy można osiągnąć porządkując globalnie wszystkie modyfikujące wywołania tego obiektu, co może być zapewnione poprzez żądanie zachowania gwarancji MW przy każdym wywołaniu metody.

Gwarancje sesji – zapisy odnośne

Definicja *Relevant Writes*

- *RW*(*r*) oznacza zbiór **zapisów odnośnych**, których wykonanie miało wpływ na wynik operacji odczytu *r*

Przykład interpretacji

- Dla obiektu typu licznik: *RW*(*r*) to sekwencja poprzednich wywołań modyfikujących stan:

`ustaw(10) → dodaj(2) → dodaj(1)`

Prezentacja pozostałych dwóch gwarancji sesji wymaga wprowadzenia dodatkowego, pomocniczego pojęcia **zapisów odnośnych** (ang. *relevant writes*). Zbiór zapisów odnośnych jest wyznaczany dla zadanej operacji odczytu *r* i zawiera tylko te operacje zapisu, które wpłynęły bezpośrednio na wynik operacji odczytu *r*.

W przypadku obiektu licznik przedstawionego wcześniej zbiorem zapisów odnośnych dla zadanej operacji *r* będzie zbiór wszystkich wcześniejszych operacji zapisu wykonanych na serwerze, który realizuje operację *r*. Ponieważ zbiór *RW(r)* ma być minimalny, to można dokonać tu pewnego odcięcia operacji zapisu, które poprzedzają ostatnie wykonanie metody *ustaw(n)*. Metoda ta jest metodą inicjującą, a więc bez względu na to, jakie operacje były wykonywane wcześniej stan obiektu po wywołaniu tej metody będzie zależał tylko i wyłącznie od jej argumentu.

Monotonic Reads (MR)

- Użytkownik chce obserwować stan systemu co najmniej tak aktualny jak podczas ostatniej interakcji

Definicja

$$\forall C_i \forall S_j \left[r_1 \xrightarrow{C_i} r_2 \,|_{S_j} \Rightarrow \forall w_k \in RW(r_1) \left(w_k \xrightarrow{S_j} r_2 \right) \right]$$

Przykład: skrzynka pocztowa mobilnego użytkownika. Zawartość skrzynki powinna zawierać przynajmniej te listy, które były widoczne przy poprzednim dostępie.

Kolejna gwarancja to **monotoniczność odczytów** (ang. *Monotonic Reads — MR*). Nieformalnie rzecz ujmując: użytkownik chce obserwować monotonicznie przyrastającą informację o stanie systemu. Innymi słowy: kolejne odczyty nie powinny „cofać się w czasie” i zwracać informację starszą niż ta, którą już zaobserwował klient podczas swoich poprzednich odczytów.

Definicja formalna rozpatruje dwie operacje odczytu r_1 i r_2. Przed wykonaniem odczytu r_2 serwer musi zagwarantować, że wykonał wszystkie zapisy, które są zapisami odnośnymi w stosunku do r_1.

Jako przykład zastosowania można rozważyć skrzynkę pocztową użytkownika. Każde kolejne otworzenie skrzynki, bez względu na to poprzez jaki serwer realizowane, powinno zwrócić co najmniej te listy, które były widoczne poprzednio. Gwarancja ta nie wymusza aktualizowania danych, jej rola jest bardziej minimalistyczna – użytkownik nie chce tracić dostępu do informacji, z którą już się zapoznał.

Writes Follow Reads (WFR)

- Użytkownik chce respektowania przyczynowej zależności wprowadzanych modyfikacji względem poprzednich odczytów

Definicja

$$\forall C_i \forall S_j \left[r \xrightarrow{C_i} w|_{S_j} \Rightarrow \forall w_k \in RW(r) \left(w_k \xrightarrow{Sj} w \right) \right]$$

Przykład: lista dyskusyjna. Użytkownik wysyła odpowiedź i chce, aby wszyscy widzieli łącznie z jego odpowiedzią również oryginalny list

Ostatnia gwarancja sesji to **zapisy po odczytach** (ang. *writes follow reads* — *WFR*). Wykonanie zapisu po wykonaniu operacji odczytu może oznaczać, że zapis ten zależy przyczynowo od tego odczytu (do zapisu wykorzystano wynik operacji odczytu). Użytkownik może, poprzez zastosowanie gwarancji WFR, wymusić respektowanie tej zależności przyczynowej przez inne serwery. Oznacza to, że wykonanie zapisu będzie musiało być poprzedzone wykonaniem tych zapisów, które wpłynęły na taki, a nie inny wynik operacji odczytu.

Przykładem zastosowania może być lista dyskusyjna. Użytkownik wysyłając odpowiedź na inny list zakłada, że wszędzie gdzie jego odpowiedź się pojawi będzie też dostępny oryginalny list, na który odpowiada. List oryginalny będzie dostępny, gdy każdy serwer realizujący zapis nowego listu zrealizuje wcześniej dodanie listu, do którego odnosi się ten bieżący.

Definicje – podsumowanie

poprzednia operacja	bieżąca operacja	gwarancja sesji
write	read	RYW
write	write	MW
read	read	MR
read	write	WFR

Podsumowując definicje gwarancji sesji można zauważyć, że odnoszą się one zawsze do bieżącej operacji i do pewnych operacji ją poprzedzających. W zależności od tego, które uporządkowanie użytkownik (klient) chce zachować, może wskazać właściwą gwarancję. Jest też oczywiście możliwe łączenie wielu gwarancji w celu uzyskania skumulowanych własności uporządkowania operacji. W szczególności operacje mogą być zlecane bez precyzowania żadnych gwarancji, a więc bez żadnej kontroli co do uporządkowania.

Deklarowanie wielu gwarancji sesji nie zawsze wpływa na działanie systemu. Jeżeli przykładowo użytkownik żąda RYW a zleca wykonanie operacji zapisu, to ta gwarancja nie wpłynie na działanie systemu.

Protokół logiczny – oznaczenia

O_{S_j}	zbiór operacji zapisu wykonanych przez serwer *Sj*
W_{C_i}	zbiór operacji zapisu wykonanych przez klienta C_i
R_{C_i}	zbiór operacji zapisu, których efekty zostały (częściowo) odczytane przez klienta C_i
⟨***op, SG***⟩	żądanie wykonania operacji ***op*** z uwzględnieniem gwarancji sesji ze zbioru *SG*
⟨***op, W***⟩	żądanie wykonania operacji ***op*** wysyłane do serwera ***W*** – zbiór operacji wymaganych do wykonania operacji ***op***
⟨***op, res, W***⟩	odpowiedź serwera: ***res*** – wynik operacji ***op*** ***W*** – zbiór operacji będących wynikiem wykonania operacji ***op***

Podobnie jak przy okazji modeli spójności nastawionych na dane również i w przypadku gwarancji sesji zostaną przedstawione niektóre protokoły spójności realizujące te modele. Prezentacja ta w pierwszej kolejności dotyczyć będzie protokołu „logicznego”, który nie jest praktycznym, gotowym do wykorzystania protokołem a raczej szkieletem, który wymaga dalszego uzupełnienia. Jego prezentacja jest jednak bardzo użyteczna, ponieważ znane protokoły spójności gwarancji sesji powielają go.

Każdy serwer dla potrzeb zarządzania danymi rejestruje wszystkie operacje które sam realizuje. Powstaje w ten sposób liniowo uporządkowany zbiór operacji zapisu (odczyty nie muszą być pamiętane, ponieważ nie zmieniają stanu systemu). Każdy klient przechowuje dwa zbiory operacji zapisu. Zbiór *W* przechowuje wszystkie zapisy bezpośrednio zlecone przez klienta. Zbiór *R* przechowuje wszystkie zapisy, których efekty klient zaobserwował (być może tylko częściowo). Zbiór ten zawiera więc zbiór zapisów odnośnych w stosunku do wszystkich odczytów zleconych przez klienta.

Zlecenie wykonania operacji przekazywane przez aplikację ma postać komunikatu zawierającego kod operacji do wykonania i zbiór *SG* gwarancji sesji, które mają być przestrzegane przy wykonywaniu tej operacji. Zbiór gwarancji sesji może być inny przy każdej kolejnej operacji. Żądanie jest przetwarzane lokalnie po stronie klienta i do serwera jest wysyłany komunikat zawierający kod operacji oraz zbiór *W* zapisów, które muszą być sprawdzone przed wykonaniem bieżącej operacji. Odpowiedź serwera to komunikat zawierający kod operacji, jej wynik oraz zbiór *W* zapisów, które są „efektem” wykonania operacji.

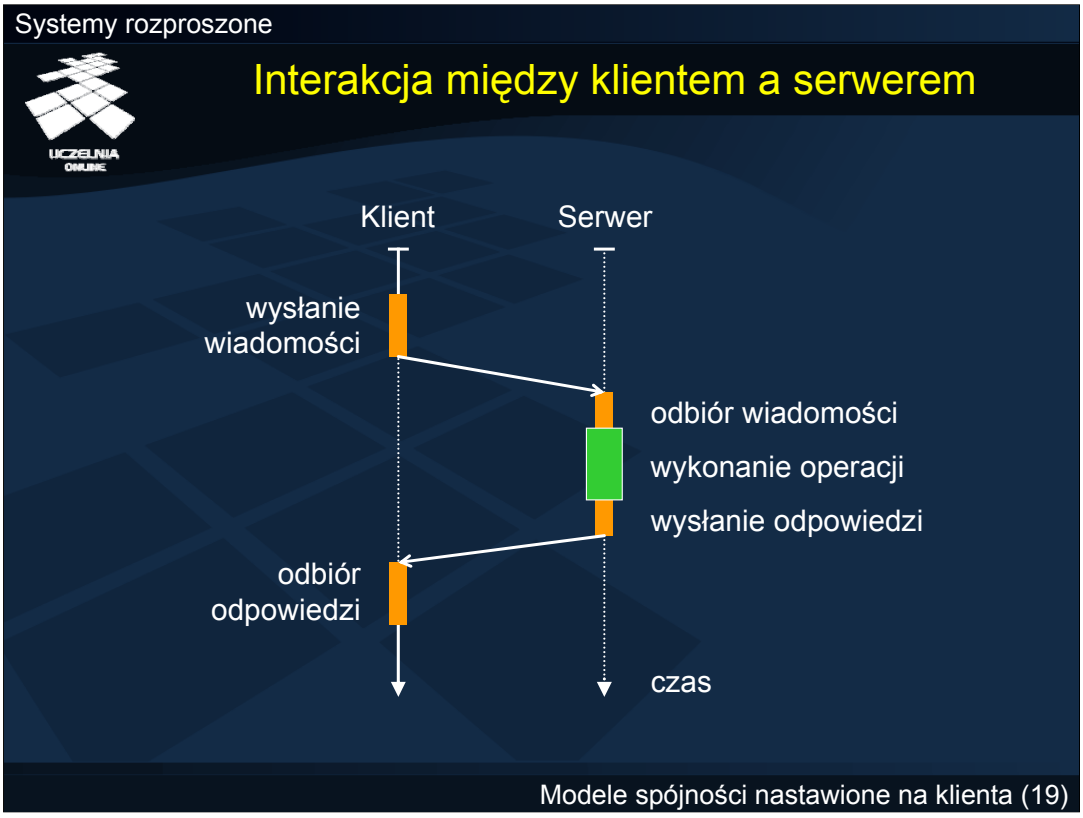
Historia przetwarzania

- **Historia H_{S_j}** jest liniowo uporządkowanym zbiorem $(O_{S_j}, \xrightarrow{S_j})$, gdzie O_{S_j} jest zbiorem zapisów wykonanych przez serwer S_j, a $\xrightarrow{S_j}$ relacją porządkującą te zapisy
- **Konkatenacja historii** H_{S_j} i H_{S_k} polega na dołączeniu na końcu H_{S_j} nowych operacji z H_{S_k} przy zachowaniu odpowiednich uporządkowań

$$H^*_{S_j} = H_{S_j} \oplus H_{S_k}$$

Każdy serwer rejestruje zapisy przez siebie wykonywane. Operacje te tworzą zbiór liniowo uporządkowany zwany **historią**. Historia przetwarzania jest potrzebna do tego, żeby zarejestrować zlecenia i przekazać je później innym węzłom w celu uspójnienia stanu replik. Ponieważ w większości przypadków serwery nie posiadają wiedz o stanie replik pozostałych serwerów, muszą przechowywać dane z historii dostatecznie długo. Odpowiednia część protokołu spójności może zajmować się czyszczeniem fragmentów historii przetwarzania, która została już zrealizowana przez wszystkie węzły, a więc może być usunięta. Jest to jednak zagadnienie nie związane bezpośrednio z problemem spójności i nie będzie tu analizowane.

Serwery synchronizują się poprzez bezpośrednią wymianę informacji punkt-punkt. Serwer *X* przekazuje do serwera *Y* fragment historii przetwarzania, który jest nieznany *Y*, a *Y* przekazuje odpowiedni fragment do *X*. Podczas przekazywania historii następuje wykonanie nieznanych operacji zapisu oraz realizowana jest **konkatenacja** historii, polegająca na doklejeniu na końcu bieżącej historii nowych operacji z zachowaniem kolejności.



Interakcja pomiędzy klientem a serwerem ma postać zdalnych wywołań procedur. Zlecenie operacji na serwerze powoduje przesłanie do niego komunikatu. Po stronie serwera, po wstępnym przetworzeniu komunikatu, następuje wykonanie operacji i następnie wysłanie odpowiedzi. Komunikat jest odbierany po stronie klienta i wyniki są przekazywane do aplikacji. Podczas realizacji zdalnej operacji klient jest blokowany.

Protokół spójności ingeruje w mechanizm wymiany danych pomiędzy klientem a serwerem w celu wykonania dodatkowych operacji zarządzających, mających na celu realizację wymaganych gwarancji sesji. Ingerencja ta ma miejsce przed wysłaniem komunikatu do serwera, po jego odebraniu po stronie serwera, przed wysłaniem odpowiedzi do klienta i po jej odebraniu. Prezentacja protokołów spójności opisuje działania wykonywanie właśnie w tych czterech wymienionych etapach przetwarzania żądań klienta.

Protokół logiczny – po stronie klienta C_i (I)

Wysyłanie żądania $\langle op, SG \rangle$ **do serwera** S_j

1. $W \leftarrow \emptyset$
2. **if** (`iswrite`(*op*) **and** $MW \in SG$)
   or (**not** `iswrite`(*op*) **and** $RYW \in SG$) **then**
3. $\quad W \leftarrow W \cup W_{C_i}$
4. **end if**
5. **if** (`iswrite`(*op*) **and** $WFR \in SG$)
   or (**not** `iswrite`(*op*) **and** $MR \in SG$) **then**
6. $\quad W \leftarrow W \cup R_{C_i}$
7. **end if**
8. send $\langle op, W \rangle$ to S_j

Obsługa żądania przechwytywana jest jeszcze po stronie klienta. Na podstawie zbiorów W_{Ci} i R_{Ci} przygotowywany jest zbiór *W* zawierający operacje, które muszą być zweryfikowane po stronie serwera. Zawartość tego zbioru zależy od tego jaką operację zleca klient oraz od tego jakich gwarancji sesji żąda. Funkcja *iswrite(op)* sprawdza czy wykonywana operacja jest zapisem. Zbiór wynikowy *W* może być pusty, może zawierać zapisy ze zbioru W_{Ci} lub ze zbioru R_{Ci} lub ich sumę mnogościową. Ostatecznie zbiór *W* jest wysyłany razem z kodem operacji do serwera.

Protokół logiczny – po stronie serwera S_j

Odbiór żądania $\langle op, W \rangle$ **od klienta** C_i

1. **while not** $O_{S_j} \supseteq W$ **do**
2. `wait()`
3. **end while**
4. wykonaj operację *op* i zapisz wyniki w *res*
5. **if** `iswrite`(*op*) **then**
6. $H_{S_j} \leftarrow H_{S_j} \oplus \{op\}$
7. $W \leftarrow \{op\}$
8. **else**
9. $W \leftarrow RW(op)$
10. **end if**
11. send $\langle op, res, W \rangle$ to C_i

Najważniejsza część protokołu realizowana jest po stronie serwera. W linii 1 następuje sprawdzenie czy zbiór operacji zapisu wykonanych przez serwer zawiera wszystkie operacje wymagane przez klienta. Jeżeli tak, przetwarzanie może być kontynuowane. Łatwo zauważyć, że przetwarzanie nigdy nie będzie wstrzymywane jeżeli klient nie żąda zachowania żadnych gwarancji sesji (zbiór *W* będzie w tym przypadku pusty). Jeżeli serwer nie jest dostatecznie zaktualizowany (nie wykonał wszystkich operacji wymaganych przez klienta), bieżąca operacja musi być wstrzymana (linia 2). Wznowienie jej nastąpi po aktualizacji serwera w wyniku wymiany informacji z innym serwerem. O ile wymagania klienta są spełnione, można przejść do jej wykonania (linia 4). W przypadku operacji zapisu, po jego wykonaniu, jest on dodawany do lokalnej historii H_{Sj}, a więc od zbioru O_{Sj} z zachowaniem uporządkowania (linia 6). Jednocześnie kod operacji jest zwracany do klienta (linia 7). W przypadku odczytów natomiast obliczany jest zbiór zapisów odnośnych dla bieżącej operacji (linia 9). Ostatecznie wynik operacji jest odsyłany do klienta (linia 11).

Protokół logiczny – po stronie klienta C_i (II)

Odbiór odpowiedzi $\langle op, res, W \rangle$ **z serwera** S_j

1. **if** `iswrite`(*op*) **then**
2. $W_{c_i} \leftarrow W_{C_i} \cup W$
3. **else**
4. $R_{c_i} \leftarrow R_{C_i} \cup W$
5. **end if**
6. deliver *res*

Odebranie wiadomości po stronie klienta powoduje aktualizację lokalnie przechowywanych zbiorów W_{ci} i R_{ci} na podstawie zbioru operacji przesłanych przez serwer. W przypadku zapisu nastąpi dodanie tego zapisu do zbioru W_{ci} (linia 2), a w przypadku odczytu zbiór R_{ci} zostanie uzupełniony o zbiór operacji zapisu, które miały wpływ na taki, a nie inny wynik zrealizowanej właśnie operacji odczytu (linia 4). Ostatecznie wyniki zdalnie realizowanej operacji dostarczane są do aplikacji (linia 6).

Protokoły oparte na wektorach wersji

Protokół logiczny

- nierealistyczny ze względu na monotonicznie rozrastające się zbiory operacji

Wektory wersji

- wektor wersji: sekwencja liczb $V=[v_1\ v_2 \ldots v_n]$
- koncepcja oparta na zegarach wektorowych
- aktualizacja wektorów wersji przez poszczególne serwery
- etykietowanie zapisów aktualną wartością wektora wersji z serwera wykonującego zapis po raz pierwszy
- efektywna reprezentacja zbiorów operacji

Podstawową wadą protokołu logicznego jest operowanie na jawnych zbiorach operacji zapisu. Zbiory te w trakcie pracy systemu monotonicznie przyrastają, powodując monotoniczny rozrost przesyłanych komunikatów sieciowych. Bardzo szybko działanie takiego protokołu stawałoby się kompletnie nieefektywne. Stąd praktyczna realizacja protokołu spójności gwarancji sesji wymaga zaproponowania efektywnej metody reprezentacji zbiorów operacji zapisu.

Metoda taka została zaproponowana już przez twórców samego modelu i oparta jest na wykorzystaniu **wektorów wersji** (ang. *version vectors*). Wektory wersji koncepcyjnie oparte są na mechanizmie zegarów wektorowych (ang. *vector clocks*), wykorzystywanych w systemach rozproszonych do reprezentacji czasu logicznego i do śledzenia zależności przyczynowych pomiędzy zdarzeniami. Wektor wersji to sekwencja liczb, które w klasycznym podejściu interpretowane są jako liczniki zdarzeń zarejestrowanych na poszczególnych węzłach (w tym przypadku zapisów). Każdy zapis, który jest zgłaszany po raz pierwszy w systemie, a więc taki, który trafia do serwera w wyniku bezpośredniej interakcji z klientem, jest etykietowany bieżącą wartością wektora wersji przechowywanego na serwerze. Mechanizm ten zapewnia unikalną identyfikację wszystkich operacji zapisu.

Reprezentacja zbiorów zapisów

- Niech $T : O \rightarrow V$ będzie funkcją odwzorowującą operację zapisu w wektor wersji przypisany temu zapisowi

Definicja

- Dla danego wektora wersji *V*, zbiór operacji zapisu *WS*(*V*) reprezentowanych przez ten wektor wersji jest definiowany następująco:

$$WS(V) = \bigcup_{j=1}^{N_S} \left\{ w \in O_{S_j} : T(w) \leq V \right\}$$

- Relacja dominacji dla wektorów wersji:

$$\forall i : V_1[i] \geq V_2[i]$$

Wektory wersji umożliwiają efektywną reprezentację zbiorów operacji. Slajd pokazuje w jaki sposób realizowana jest reprezentacja zbiorów zapisów przez wektory wersji. Załóżmy, że funkcja *T:O→V* będzie wydobywać z kodu operacji jej etykietę wektorową. Zbiór operacji reprezentowanych wektorem wersji *V* będzie oznaczany jako *WS(V)*. Zbiór ten obejmuje wszystkie zapisy z wszystkich serwerów o ile etykieta wektorowa dla tych zapisów jest *zdominowana* przez zadany wektor wersji. Relacja dominacji jest przedstawiona na slajdzie. Wektor V_1 dominuje nad V_2 jeżeli wszystkie pozycje tego wektora są nie mniejsze od odpowiednich pozycji wektora V_2.

Interpretacja zbiorów *WS(V)* jest dość intuicyjna w przypadku klasycznych wektorów wersji, gdzie każda pozycja wektora oznacza liczbę operacji zapisu przyjętych do realizacji przez dany serwer.

Reprezentacja poprzez wektory wersji

Zalety

- stały rozmiar wektora wersji
- efektywne przetwarzanie

Wady

- zbiory reprezentowane przez wektory wersji zawierają więcej operacji niż wynika to z definicji gwarancji sesji
  - → warunek dostateczny zapewniania gwarancji sesji
  - → dokładność reprezentacji zbiorów

Reprezentacja zbiorów operacji z wykorzystaniem wektorów wersji ma podstawową zaletę w postaci stałego i stosunkowo niskiego rozmiaru wektora wersji. Oznacza to, że odpowiednie komunikaty sieciowe będą obarczone tym samym kosztem i ich rozmiar nie będzie się zmieniać w związku z wykonywaniem coraz to nowszych operacji zapisu. Wektory wersji są też prostymi strukturami danych, w efekcie czego ich przetwarzanie jest bardzo efektywne.

Podstawowa wada stosowania wektorów wersji to mniejsza precyzja reprezentacji zbiorów zapisu. Wektor wersji odcina w czasie pewien moment przetwarzania obejmując swoim zasięgiem wszystko co było wcześniej. Nie można więc stosując wektor wersji wskazać na pojedynczy zapis, którego wykonanie chcielibyśmy wymusić na innym serwerze. Każda wartość wektora wersji obejmująca swoim zasięgiem zapis *w* obejmuje również wszystkie inne zapisy, który zostały przyjęte do realizacji przez dany serwer przed zapisem *w*. Zbiory zapisów reprezentowanych przez wektory wersji są więc nadzbiorami zbiorów dokładnych (które występowały w protokole logicznym). Zastosowanie wektorów wersji w protokołach spójności będzie więc powodowało wymuszanie wykonywania dodatkowych operacji zapisu, których wykonanie nie wynika bezpośrednio z definicji samych gwarancji sesji. W efekcie, protokoły stosujące wektory wersji spełniają warunek dostateczny zapewnienia odpowiednich gwarancji sesji. Są jednak w pewien sposób nadmiarowe, wymuszając dodatkowy synchronizm przetwarzania. Nadmiarowość ta będzie tym większa im mniej dokłada jest reprezentacja zbiorów zapisów. Z tego powodu konstrukcja wektorów wersji staje się istotnym elementem rzutującym na efektywność pracy protokołu spójności.

Typy wektorów wersji

Wektory wersji oparte na serwerach: $V_S = [\, v_1\ v_2 \ldots v_{N_S} \,]$

- v_i oznacza liczbę zapisów wykonanych przez serwer S_i
- N_S jest ogólną liczbą serwerów w systemie

Wektory wersji oparte na obiektach: $V_O = [\, v_1\ v_2 \ldots v_{N_O} \,]$

- v_i oznacza liczbę zapisów wykonanych na obiekcie x_i
- N_O jest ogólną liczbą współdzielonych obiektów w systemie

Wektory wersji oparte na klientach: $V_C = [\, v_1\ v_2 \ldots v_{N_C} \,]$

- v_i oznacza liczbę zapisów zleconych przez klienta C_i
- N_C jest ogólną liczbą klientów w systemie

Wektory wersji w klasycznej postaci mają tyle pozycji ile jest serwerów w systemie. Każda pozycja reprezentuje liczbę zapisów przyjętych do realizacji (i wykonanych) przez dany serwer. Dokładnie taką konstrukcję mają zegary wektorowe. Okazuje się jednak, że dla potrzeb konstrukcji protokołów spójności gwarancji sesji nie jest to jedyna konstrukcja, która może być rozważana.

Wektory wersji mogą być konstruowane w oparciu o obiekty, do których odwołują się użytkownicy w systemie. Wektor V_O będzie miał tyle pozycji ile jest obiektów dzielonych. Każda pozycja będzie reprezentować liczbę modyfikacji wykonanych na tym obiekcie przez dany serwer.

Trzecią propozycją konstrukcji wektora wersji jest wektor V_C oparty na klientach. Każda pozycja takiego wektora wersji oznacza liczbę operacji zleconych przez *i*-tego klienta. Wektor taki ma więc rozmiar równy liczbie klientów w systemie.

Przedstawione typy wektorów wersji oczywiście mogą się znacząco różnić rozmiarem w zależności od systemu i charakteru aplikacji. Różna też może być częstotliwość zmian takiego wektora wersji. Ten ostatni aspekt jest szczególnie istotny w przypadku implementacji realnie działających systemów, ponieważ ich konfiguracja może ulegać ciągłym zmianom.

Protokół VsSG – oznaczenia

V_{S_j}	wektor reprezentujący operacje zapisu wykonane przez serwer S_j
W_{C_i}	wektor reprezentujący operacje zapisu wykonane przez klienta C_i
R_{C_i}	wektor reprezentujący operacje zapisu, których efekty zostały odczytane przez klienta C_i
$\langle op, W \rangle$	żądanie wykonania operacji *op* wysyłane do serwera *W* – wektor reprezentujący zbiór operacji wymaganych do wykonania operacji *op*
$\langle op, res, W \rangle$	odpowiedź serwera: *res* – wynik operacji *op* *W* – wektor reprezentujący zbiór operacji będących wynikiem wykonania operacji *op*

Przedstawiony zostanie teraz protokół spójności VsSG korzystający z wektorów wersji opartych na serwerach. Jest to protokół zaproponowany przez twórców gwarancji sesji (nazwa została nadana później). Protokół ten wykorzystuje struktury danych analogiczne do struktur danych protokołu logicznego. Podstawowa zmiana polega na zastosowaniu wektorów wersji zamiast jawnych zbiorów operacji.

Protokół VsSG – po stronie klienta C_i (I)

Wysłanie żądania $\langle op, SG \rangle$ **do serwera** S_j

1. $W \leftarrow \mathbf{0}$
2. **if** (`iswrite`(*op*) **and** MW$\in SG$)
   or (**not** `iswrite`(*op*) **and** RYW$\in SG$) **then**
3. $\quad W \leftarrow \max(W, W_{C_i})$
4. **end if**
5. **if** (`iswrite`(*op*) **and** WFR$\in SG$)
   or (**not** `iswrite`(*op*) **and** MR$\in SG$) **then**
6. $\quad W \leftarrow \max(W, R_{C_i})$
7. **end if**
8. send $\langle op, W \rangle$ to S_j

Protokół spójności korzystający z wektorów wersji, zgodnie z tym co było powiedziane wcześniej, co do struktury jest bardzo podobny do protokołu logicznego. Ponieważ jednak operacje dotyczą wektorów wersji, to odpowiednie operatory mają zmienioną postać. I tak: w linii 1 następuje wyzerowanie wektora wersji *W*. W kolejnych liniach wartość tego wektora jest wypełniana pozycjami z wektorów R_{Ci} i W_{Ci} – zamiast sumy mnogościowej wykorzystywana jest maksymalizacja wartości wektorów realizowana jako maksymalizacja każdej z pozycji niezależnie.

Protokół VsSG – po stronie serwera S_j

Odbiór żądania $\langle op, W \rangle$ **od klienta** C_i

1. **while** (**not** $V_{S_j} \geq W$) **do**
2. `wait()`
3. **end while**
4. wykonaj operację *op* i zapisz wyniki w *res*
5. **if** `iswrite`(*op*) **then**
6. $V_{S_j}[j] \leftarrow V_{S_j}[j] + 1$
7. oznacz zapis *op* etykietą V_{S_j}
8. $H_{S_j} \leftarrow H_{S_j} \oplus \{op\}$
9. **end if**
10. send $\langle op, res, V_{S_j} \rangle$ to C_i

Po stronie serwera, w pierwszym kroku następuje sprawdzenie aktualności danych po stronie serwera realizowane przez porównanie wektora wersji serwera z wektorem wersji przesłanym przez klienta. Dominacja wektora wersji serwera oznacza, że zbiór zapisów reprezentowany tym wektorem wersji (a więc i wykonanych już przez serwer) jest nadzbiorem zbioru zapisów reprezentowanych wektorem wersji przesłanym przez klienta. Efektywnie oznacza to, że serwer wykonał wszystkie operacje zapisu, których oczekiwał klient. Ze względu na nadmiarowość reprezentacji zbiorów operacji, tak naprawdę serwer wykonał więcej operacji niż żądał klient, ale to nie narusza modelu spójności.

Serwer, po ewentualnej aktualizacji swoich danych, przechodzi do wykonania zlecenia (linia 4). W przypadku zapisu następuje inkrementacja wektora wersji przechowywanego przez serwer na pozycji reprezentującej ten serwer (linia 6). Operacja zapisu oznaczona bieżącą wartością wektora wersji trafia następnie do historii przetwarzania rejestrowanej przez serwer.

Bez względu na rodzaj operacji serwer kończy przetwarzanie odsyłając bieżącą wartość swojego wektora wersji (linia 10).

Protokół VsSG – po stronie klienta C_i (II)

Odbiór odpowiedzi ⟨*op, res, W*⟩ **z serwera** S_j

1. **if** `iswrite`(*op*) **then**
2. $W_{c_i} \leftarrow \max(W_{C_i}, W)$
3. **else**
4. $R_{c_i} \leftarrow \max(R_{C_i}, W)$
5. **end if**
6. deliver ⟨*res*⟩

Odbiór komunikatu z serwera powoduje aktualizację lokalnych wektorów wersji W_{Ci} i R_{Ci}, w zależności od rodzaju zlecanej operacji. Monotonicznie przyrastające wartości wektorów W_{Ci} i R_{Ci}, odzwierciedlają stan systemu, w którym był dokonywany ostatni zapis lub odczyt w systemie (wektory wersji na serwerach przechowują informację o czasie logicznym). Operacje zlecane później mogą powoływać się na ten stan i wymuszać aktualizację serwerów, gdy nie wszystkie modyfikacje został nałożone na serwer.

Synchronizacja serwerów

Co każde Δt na serwerze S_j

1. **foreach** $S_k \neq S_j$ **do**
2. send $\langle S_j, H_{S_j} \rangle$ to S_k
3. **end for**

Odbiór komunikatu aktualizacyjnego $\langle S_k, H \rangle$ na serwerze S_j

4. **foreach** $w_i \in H$ **do**
5. **if not** $V_{S_j} \geq T(w_i)$ **then**
6. wykonaj zapis w_i
7. $V_{S_j} \leftarrow \max(V_{S_j}, T(w_i))$
8. $H_{S_j} \leftarrow H_{S_j} \oplus \{w_i\}$
9. **end if**
10. **end for**
11. `signal()`

Fragmenty protokołów przedstawione do tej pory miały na celu zagwarantowanie *bezpieczeństwa* pracy protokołu, a więc niedopuszczenie do sytuacji, w której któraś z gwarancji sesji byłaby naruszona. Użytkownicy jednak spodziewają się, że wykonanie ich aplikacji będzie postępować, a więc, że będzie spełniony warunek *postępu* algorytmu rozproszonego. Postęp w przypadku omawianych protokołów będzie możliwy, gdy modyfikacje wprowadzane na jednym serwerze będą propagowane do pozostałych. Organizacja tej propagacji może być bardzo różna, począwszy od prostego rozgłaszania wiadomości aktualizacyjnych, a skończywszy na zastosowaniu algorytmów epidemicznych. Przedstawiony algorytm nie jest próbą efektywnego rozwiązania problemu propagacji, a jedynie uzupełnia w minimalny sposób kod przestawiony wcześniej, tak aby zaprezentować protokół w całości.

W przedstawionym podejściu każdy serwer okresowo wysyła komunikat aktualizacyjny do każdego innego serwera. W komunikacie zawarta jest *cała* historia lokalnego przetwarzania danego serwera. Serwer odbierający komunikat aktualizacyjny iteruje po historii (jest ona liniowo uporządkowana) i dodaje do swojej historii wszystkie operacje, które są mu nieznane, a więc takie, których etykieta wektorowa nie jest zdominowana przez etykietę wektorową serwera (linia 5). Dołączanie operacji do lokalnej historii (linia 8) oznacza również jej wykonanie (linia 6). Zakończenie aktualizacji serwera powoduje obudzenie ewentualnych wątków realizujących żądania klientów, które zostały zablokowane ze względu na nieaktualność danych na serwerze. W wątkach tych następuje ponowne porównanie wartości wektorów wersji i ponowne zaśnięcie lub przejście do wykonania operacji.

Przedstawione rozwiązanie jest wysoce nieefektywne, ponieważ powoduje wielokrotny transfer tych samych informacji do tych samych serwerów. Pokazuje jednak istotę problemu i gwarantuje postęp.

Optymalizacja protokołu VsSG

Po stronie serwera

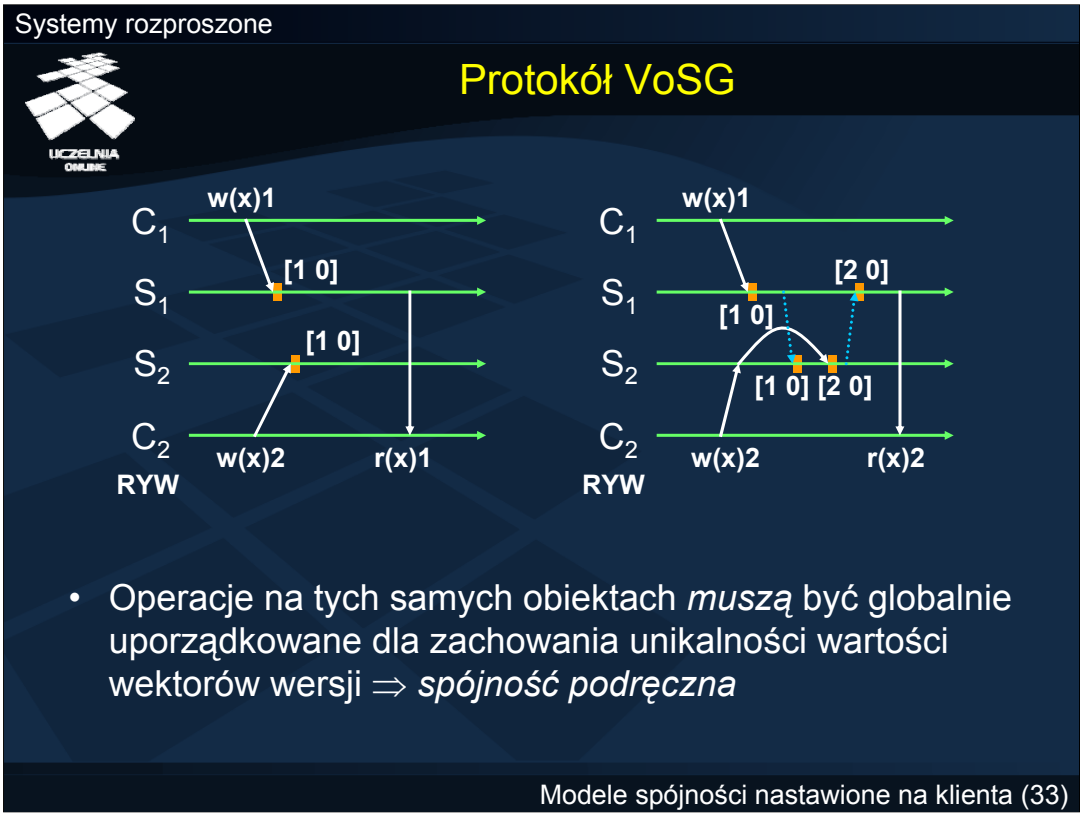
1. ...
2. **if** `iswrite`(*op*) **then**
3. send ⟨*op, res,* $V_{S_j}[j]$⟩ to C_i
4. **else**
5. send ⟨*op, res,* V_{S_j}⟩ to C_i
6. **end if**

Po stronie klienta

8. **if** `iswrite`(*op*) **then**
9. $W_{C_i}[j] \leftarrow W[j]$
10. **else**
11. $R_{C_i} \leftarrow \max(R_{C_i}, W)$
12. **end if**

Oryginalny protokół VsSG można nieco zoptymalizować ograniczając zasięg aktualizacji wektora wersji W_{Ci} po stronie klienta. Okazuje się bowiem, że wystarczające jest zaktualizowanie jedynie pozycji, która uległa zmianie podczas wykonywania zapisu, czyli pozycji reprezentującej serwer wykonujący zapis. W związku z tym serwer nie musi wysyłać całego wektora wersji przy zapisie, a jedynie pozycję, która podlega zmianie.

Przedstawiona modyfikacja ma dwie istotne zalety. Po pierwsze zmniejszeniu ulega rozmiar komunikatu potwierdzającego wykonanie operacji zapisu na serwerze. Po drugie wartość wektora wersji klienta W_{Ci} ulega zwiększeniu w znacząco mniejszym stopniu. Oznacza to, że zbiór zapisów reprezentowanych przez ten wektor będzie mniejszy (choć wciąż wystarczający) i w konsekwencji klient będzie wymuszał na kolejnych serwerach aktualizację mniejszej liczby zapisów, co może skutkować krótszym czasem oczekiwania na dostęp do serwera. Generalnie więc powinna wzrosnąć ogólna efektywność algorytmu.

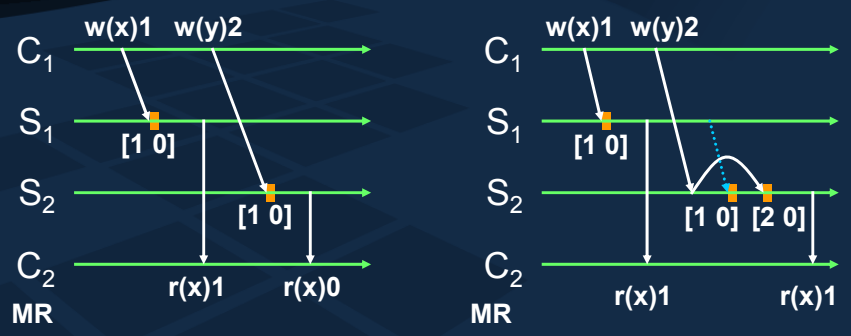


Protokoły spójności korzystające z wektorów wersji opartych na obiektach i klientach są w swojej konstrukcji bardzo zbliżone do protokołu VsSG. Inna konstrukcja wektora wersji oraz inny mechanizm jego aktualizacji wymuszają jednak pewne zmiany w protokole.

W przypadku protokołu VoSG korzystającego z wektorów wersji opartych na obiektach przetwarzanie wektorów wersji w identyczny sposób jak w przypadku wektorów wersji opartych na serwerach prowadziłoby do powstawania pewnych anomalii wynikających z braku unikalności wartości wektorów wersji po stronie serwera. Wektory wersji bazujące na serwerach były aktualizowane na pozycjach dedykowanych dla danego serwera, stąd każda nowa wartość wektora wersji powstająca w systemie, nawet przy niezależnej i współbieżnej pracy serwerów, była unikalna.

Przykład po lewej stronie pokazuje przykładowe przetwarzanie w systemie stosującym wektory wersji oparte na obiektach. W systemie występują dwaj klienci i dwa serwery. Obaj klienci zgłaszają zapis na tym samy obiekcie *x*, który załóżmy, że reprezentowany jest na pierwszej pozycji wektora wersji. Każdy z serwerów obsługujących klientów realizuje zapis współbieżnie, zwiększając wartość na odpowiedniej pozycji wektora wersji. W wyniku zapisu wektor wersji w obu serwerach jest zmieniany na [1 0]. Klient C_2 dokonuje następnie odczytu z serwera S_1, wymagając przy tym gwarancji RYW. Niestety wbrew żądanej gwarancji sesji, stan serwera nie odzwierciedla wyniku poprzedniego zapisu tego klienta. Dzieje się tak pomimo, że wektor wersji serwera S_1 dominuje nad przesyłanym przez klienta C_2 wektorem W_{C2}=[1 0]. Powodem zaistniałej anomalii jest nieunikalność wartości wektorów wersji powstających na serwerach, co uniemożliwia unikalne zaetykietowanie zapisów. Rozwiązaniem jest wymuszenie globalnego uporządkowania zapisów do poszczególnych obiektów, co oznacza wymuszenie *spójności podręcznej*. Przykład po prawej stronie realizuje odpowiednie porządkowanie operacji. W tym przypadku zapis klienta C_2 zostanie zaszeregowany po zapisie klienta C_1 i zostanie zaetykietowany unikalną wartością wektora wersji [2 0]. Dzięki temu odwołanie do serwera S_1 wymusi w przyszłości jego aktualizację przed wykonaniem odczytu. Niebieskie przerywane strzałki na rysunku oznaczają komunikaty aktualizacyjne.

Protokół VcSG



- Operacje pochodzące od poszczególnych klientów muszą być globalnie uporządkowane ⇒ wymuszenie *Monotonic Writes*

Protokół VcSG korzysta z wektorów wersji opartych na klientach. Podobnie jak w przypadku protokołu VoSG przetwarzanie wektorów wersji wymaga pewnych modyfikacji. Przeanalizujmy jednak najpierw przykład z rysunku po lewej stronie. Klient C_1 dokonuje dwóch zapisów do różnych zmiennych *x* i *y*, na różnych serwerach. Serwery, podczas wykonywania zapisów, dokonują aktualizacji swoich wektorów wersji na odpowiednich pozycjach reprezentujących zgłaszającego się klienta, niech to będzie pozycja pierwsza w przypadku klienta C_1. Klient C_2 realizuje odczyty zmiennej *x*: najpierw z serwera S_1, a później z serwera S_2. Po pierwszym odczycie aktualizowany jest jego wektor R_{C2} do wartości [1 0], którą przekazuje do serwera S_2 wykonując drugi odczyt. Niestety pomimo wymagania gwarancji MR serwer S_2 nie wykonał jeszcze zapisów, które miały wpływ na wynik zaobserwowanej wcześniej wartości zmiennej *x*, pomimo, że wektor wersji serwera dominuje nad wektorem wersji przekazanym przez klienta. Podobnie jak w przypadku protokołu VoSG problemem jest w tym przypadku nieunikalność wartości wektorów wersji wykorzystywanych do etykietowania zapisów. Unikalność taką można osiągnąć wymuszając globalne porządkowanie operacji zapisu zlecanych przez poszczególnych klientów. Działanie takie będzie równoważne wymuszeniu gwarancji MW dla każdego klienta. Rysunek po prawej przedstawia tą samą sytuację, ale po uporządkowaniu zapisów pochodzących od klientów. Dzięki temu realizacja drugiego zapisu klienta C_1 musi być poprzedzona aktualizacją serwera $S_{2,}$, co umożliwi poprawne zrealizowanie operacji odczytu przez klienta C_2.

Własności poszczególnych protokołów

VsSG Zalety: stabilna struktura wektora wersji. Zmiana następuje jedynie w momencie rekonfiguracji serwerów

Wady: mała dokładność

VoSG Zalety: małe komunikaty zwrotne z serwerów, duża dokładność

Wady: struktura wektora zmienia się często, wektor jest długi, wymusza *spójność podręczną*

VcSG Zalety: małe komunikaty w przypadku zapisów, średnia częstotliwość zmian struktury wektora wersji

Wady: wymusza *Monotonic Writes*

Protokoły spójności korzystające z różnych wektorów wersji charakteryzują się różnymi własnościami. Jedną z nich jest dokładność reprezentacji zbiorów operacji. Dokładność tą przebadano w eksperymentach symulacyjnych obserwując stosunek rozmiaru dokładnych zbiorów operacji wynikających ze stosowania odpowiednich gwarancji sesji do liczby zapisów, które muszą być zweryfikowane w wyniku operowania na wektorach wersji, reprezentujących te zbiory.

Standardowy protokół VsSG w typowej sytuacji może charakteryzować się dużą stabilnością struktury wektora wersji. W większości systemów rozproszonych to serwery są najstabilniejszym elementem systemu, ich rekonfiguracja zdarza się wiec stosunkowo rzadko. Wadą tego protokołu jest jednak dość niska dokładność reprezentacji zbiorów operacji zapisu. Zaproponowana optymalizacja pozwala na wyraźnie zwiększenie dokładności reprezentacji.

Protokół VoSG charakteryzuje się małymi komunikatami przesyłanymi pomiędzy klientami a serwerami. Są one małe ponieważ nie ma potrzeby przesyłania w nich całych wektorów wersji w komunikatach zwrotnych z serwera. Zastosowane w protokole wektory wersji oparte na obiektach gwarantują również dużą dokładność reprezentacji zbiorów operacji. Wadą jest konieczność zachowania spójności podręcznej w systemie, co wymaga dodatkowych środków. Wektory wersji mogą być też podatne na zmiany, jeżeli w systemie często powstają nowe obiekty a stare są usuwane.

Protokół VcSG charakteryzuje się małymi komunikatami w przypadku realizacji zapisów, ponieważ wektory W_{Ci} w tym protokole są zredukowane do pojedynczej pozycji. Dokładność odwzorowania zbiorów jest duża ze względu na dedykowane pozycje w wektorach wersji dla poszczególnych klientów. Częstotliwość zmian wektorów wersji z reguły będzie gdzieś pomiędzy częstotliwością zmian liczby serwerów i obiektów. Wadą jest konieczność wymuszania zachowania gwarancji *Monotonic Writes*, bez względu na to czy klienci jej wymagają czy nie.

Zależność pomiędzy modelami spójności

Obraz historii serwera

- Uporządkowanie operacji postrzeganych przez pojedynczy serwer
- Bezpośrednie odniesienie do modeli spójności nastawionych na dane

Obraz historii klienta

- Uporządkowanie operacji postrzeganych przez pojedynczego (migrującego) klienta
- Obraz różny od obrazów poszczególnych serwerów

Modele spójności nastawione na dane i te nastawione na klienta mają za zadanie opisanie reguł zarządzania spójnością danych. Istnieje więc jeden, wspólny cel dla którego opracowano te mechanizmy. Powstaje w związku z tym uzasadnione pytanie o zależności pomiędzy modelami spójności nastawionymi na dane a modelami spójności nastawionymi na klienta. Czy i kiedy własności modelu z jednej klasy mogą być dyskutowane w kontekście własności modelu z drugiej klasy.

Analiza zależności pomiędzy modelami spójności z różnych klas musi być poprzedzona wskazaniem pewnych istotnych różnic pomiędzy tymi klasami oraz ich konsekwencjami. Modele spójności nastawione na klienta dopuszczają możliwość przemieszczania się klientów. Przemieszczanie powoduje, że klient obserwuje częściowo stan jednego serwera a chwilę później stan innego serwera. W konsekwencji jego obraz historii przetwarzania może być różny od obrazów wszystkich serwerów. W związku z tym będziemy mówić o modelach spójności nastawionych na dane spełnianych przez obrazy pojedynczego klienta lub serwera.

Spójność PRAM i gwarancje sesji

PRAM po stronie serwera

- Obraz serwera zachowuje model spójności PRAM wtedy i tylko wtedy gdy wszystkie operacje zlecane przez wszystkich klientów wymagają gwarancji sesji RYW i MW

PRAM po stronie klienta

- Obraz klienta zachowuje model spójności PRAM jeżeli wszystkie operacje zlecane przez wszystkich klientów wymagają gwarancji sesji RYW, MW i MR

Generalnie rzecz ujmując wprowadzenie migracji znacząco osłabia modele spójności. Pamięć która był spójna sekwencyjnie z punktu widzenia migrującego klienta jest postrzegana jako kompletnie niespójna. Wyjątkiem jest model atomowy, który jest zachowywany pomimo przemieszczania się między serwerami, ale to wynika z zachowania zależności czasu rzeczywistego.

Przeprowadzone analizy pokazały, że model spójności PRAM można zachować w obrazach serwerów wtedy i tylko wtedy gdy wszystkie operacje zlecane przez wszystkich klientów będą wymagały co najmniej gwarancji sesji RYW i MW.

Można również rozważać zachowanie modelu spójności PRAM w obrazie przetwarzania klienta. Zachowanie modelu PRAM w tym przypadku oznacza, że zachowywany będzie lokalny porządek wykonywania operacji poszczególnych klientów. Zachowanie modelu spójności PRAM po stronie klienta jest trudniejsze i wymaga dodania jeszcze gwarancji MR do list wymagań klientów. O ile więc gwarancje RYW, MW i MR są zachowane dla wszystkich operacji wszystkich klientów, to ich obrazy przetwarzania będą spójne w sensie PRAM.

Spójność przyczynowa i gwarancje sesji

Spójność przyczynowa po stronie serwera

- Obraz serwera zachowuje model spójności przyczynowej wtedy i tylko wtedy gdy wszystkie operacje zlecane przez klientów wymagają gwarancji sesji RYW, MW, WFR i MR

Spójność przyczynowa po stronie klienta

- Obraz klienta zachowuje model spójności przyczynowej jeżeli wszystkie operacje zlecane przez wszystkich klientów wymagają gwarancji sesji RYW, MW, WFR i MR

Kolejne wnioski dotyczą modelu spójności przyczynowej.

Spójność przyczynową po stronie serwera można uzyskać wtedy i tylko wtedy, gdy wszystkie żądania będą zlecane wymagając zachowania wszystkich czterech gwarancji sesji: RYW, MW, WFR i MR.

Obraz klienta zachowuje spójność przyczynową jeżeli wszystkie operacje zlecane przez wszystkich klientów wymagają gwarancji sesji RYW, MW, WFR i MR.