Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Instytut Automatyki i Informatyki Stosowanej

Praca dyplomowa inżynierska

na kierunku Informatyka
w specjalności Systemy Informacyjno-Decyzyjne

Problem konsensusu w systemach rozproszonych

Tomasz Bartłomiej Mazur
Numer albumu 293152

promotor
dr inż. Tomasz Jordan Kruk

Warszawa 2020

Streszczenie

Problem konsensusu w systemach rozproszonych

Słowa kluczowe:

Abstract

Thesis title

Keywords:

Politechnika Warszawska

Warszawa, dd.mm.rrrr
miejscowość i data

imię i nazwisko studenta

numer albumu

kierunek studiów

OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4. lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz. U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w systemie iSOD są identyczne.

czytelny podpis studenta

Spis treści

1. Wstęp

1.1. Cel pracy inżynierskiej

Celem pracy inżynierskiej jest zaprojektowanie zbioru narzędzi wspomagających tworzenie i administrowanie klastrem serwerów. W celu zapewnienia wysokiej dostępności i odporności na awarie klaster będzie wykorzystywał protokół konsensusu RAFT. Zaprojektowane narzędzia umożliwią m.in. zarządzanie konfiguracją klastra, zapisywanie i replikację danych do wszystkich serwerów oraz tworzenie i zapisywanie kopii zapasowych. W pracy zostaną zaprezentowane sposoby użycia i możliwe zastosowania poszczególnych narzędzi. Na potrzeby pracy inżynierskiej wykorzystana zostanie implementacja protokołu RAFT stworzona przez firmę HashiCorp.

2. Systemy rozproszone

2.1. Definicja i własności systemu rozproszonego

System rozproszony możemy zdefiniować jako pewną strukturę, składającą się z grupy niezależnych, pracujących współbieżnie komponentów (komputerów, procesów), które realizują wspólny cel. Wykorzystują one różne formy komunikacji sieciowej w celu wymiany informacji i koordynacji działania. Systemy rozproszone projektowane są z myślą o zapewnieniu połączenia pomiędzy różnymi zasobami i udostępnianiu ich wielu użytkownikom. Przyjmuje się, że powinny one spełniać pewne wymagania, do których należą:

- przezroczystość,
- otwartość,
- skalowalność,
- odporność na awarie.

Przezroczystość systemu rozproszonego oznacza, że wywiera on na użytkowniku wrażenie jednej, spójnej całości. Przed użytkownikiem powinna być ukryta struktura wewnętrzna i złożony sposób działania. Cecha ta może być obecna w systemie na różnych poziomach. Przykładowo, przezroczystość dostępu oznacza, że ujednolicony jest sposób dostępu do danych i ich reprezentacji. Przezroczystość położenia polega na ukryciu rzeczywistego miejsca przechowywania zasobów. Możemy też mówić m.in. o ukrywaniu awarii, współbieżności lub replikacji.

Otwartość systemu rozproszonego oznacza, że oferowane przez niego usługi są zgodne z pewnymi wcześniej określonymi zasadami, regułami dotyczącymi tych usług. Specyfikacje usług powinny być kompletne, czyli zawierać obejmować wszystkie elementy niezbędne przy tworzeniu ich implementacji. Jednocześnie przyjmuje się, że specyfikacje są neutralne, a więc nie narzucają, jak dokładnie powinna wyglądać implementacja. Otwartość jest niezbędna do zapewnienia możliwości swobodnej rozbudowy systemu rozproszonego.

Skalowalność to własność systemu, która określa, w jakim stopniu jest on zdolny poradzić sobie z rosnącą ilością zasobów. System rozproszony może być skalowalny pod różnymi względami. Przede wszystkim istnieje możliwość dodawania do niego nowych zasobów, które mogą być od siebie znacznie oddalone pod względem geograficznym. Jednocześnie cecha ta oznacza, że rozszerzanie systemu nie powoduje zwiększonej trudności w jego zarządzaniu.

Odporność na awarie to zdolność do kontynuowania działania pomimo wystąpienia awarii części systemu. W systemach rozproszonych każdy komponent może ulec awarii niezależnie od pozostałych. Jednakże, nieprawidłowe działanie niewielkiej liczby komponentów nie powinno powodować braku możliwości świadczenia usług i powinno być ukrywane przed użytkownikiem. Awarie możemy podzielić na dwie kategorie. Do pierwszej, ogólnej, należą sytuacje, w których działanie komponentu jest nieprzewidywalne, np. celowo wysyła on nieprawidłowe lub sprzeczne informacje, może też np. spowalniać swoje działanie. Są to tzw. awarie bizantyjskie. Druga kategoria, awarie zatrzymania, obejmuje każde niespodziewane zatrzymanie działania danego komputera, które nie jest już wznawiane. Wykrywanie awarii bizantyjskich jest zdecydowanie trudniejsze od wykrycia awarii zatrzymania i wymaga zastosowania specjalnych mechanizmów.

2.1.1. Komunikacja w systemach rozproszonych

Szczególnie ważną rolę w systemie rozproszonym odgrywa komunikacja. Wymiana informacji między komponentami jest konieczna w celu koordynacji działania całego systemu. Przy projektowaniu systemu komunikacyjnego niezbędne jest uzgodnienie szczegółów jej funkcjonowania, m.in. dotyczących sposobu reprezentacji i długości poszczególnych typów danych oraz metod wykrywania błędów w przesyłaniu wiadomości. Systemy rozproszone mogą wykorzystywać różne mechanizmy wymiany wiadomości. Do najprostszych należą różne protokoły sieciowe (np. HTTP). Przykładem bardziej złożonej metody może być np. zdalne wywoływanie procedur (remote procedure control, w skrócie RPC).

Mechanizm RPC polega na wywołaniu procedury znajdującej się na innej maszynie. Jego działanie rozpoczyna się w momencie, w którym proces zażąda wykonania zdalnej procedury. Żądanie to jest przetwarzane na wiadomość sieciową, w której zawarty jest identyfikator wywołania wraz z przekazywanymi parametrami. Po odebraniu wiadomości proces wykonujący przekształca ją na odpowiednie wywołanie lokalne, które jest następnie wykonywane. Po zakończeniu działania wynik jest odsyłany procesowi wywołującemu jako odpowiedź na żądanie.

Metoda zdalnego wywoływania procedur została opracowana w celu zapewnienia większego poziomu przezroczystości w systemach rozproszonych. Przyjęto, że z punktu widzenia zarówno procesu wywołującego, jak i procesu wykonującego, zdalne wywołanie procedury powinno przypominać wywołanie lokalne. W tym celu stworzono koncepcję tzw. pieńka, czyli specjalnej procedury pośredniczącej, która pomaga przy obsłudze żądania RPC. Obie maszyny uczestniczące w zdalnym wywołaniu posiadają własny pieniek. Proces zdalnego wywołania procedury przy użyciu pieńków prezentuje się następująco:

1. Proces po stronie żądającego wywołuje procedurę pieńka z odpowiednimi parametrami.
2. Pieniek żądającego przetwarza wywołanie na wiadomość sieciową, w której umieszcza identyfikator procedury oraz parametry.
3. Wygenerowana wiadomość jest przesyłana do odpowiedniego odbiorcy,
4. Odbiorca odbiera wiadomość i przekazuje ją własnemu pieńkowi, który odczytuje z niej identyfikator i parametry wywołania.
5. Pieniek odbiorcy wywołuje odpowiednią procedurę, a następnie generuje wiadomość sieciową, w której umieszcza wynik wykonania.
6. Wiadomość sieciowa zawierająca odpowiedź na żądanie jest przesyłana do procesu żądającego.

2.1.2. Systemy synchroniczne i asynchroniczne

Poszczególne komponenty systemy rozproszonego są niezależne i pracują współbieżnie. Wynika z tego, że wiele wydarzeń może zachodzić w tym samym czasie. Do zapewnienia poprawności i koordynacji działania komponentów często niezbędne jest ustalenie, w jakiej kolejności następowały te wydarzenia. Zagadnienie to nazywamy synchronizacją.

Systemy komputerowe do odmierzania czasu wykorzystują specjalne zegary, które wykazują pewną niedokładność. Zjawisko to możemy zauważyć, jeżeli ustawimy dwa niezależne zegary komputerowe na ten sam czas. Po pewnym okresie wskazania zegarów będą się różnić, ponieważ dokładność odmierzania czasu jest różna. Z podobną sytuacją mamy do czynienia w systemie rozproszonym, w którym każdy komponent wykorzystuje swój własny, lokalny zegar. Rzeczywista kolejność wystąpienia zdarzeń w systemie może być inna

niż wynikająca z czasu zarejestrowanego przez poszczególne komputery. W typowym systemie rozproszonym zazwyczaj nie istnieje globalny zegar, który pozwalałby ja ustalić. Dlatego też do synchronizacji zegarów wykorzystywane są specjalne algorytmy np. znaczniki czasu Lamporta. Na podstawie tego sposobu synchronizacji oraz pewnych przyjętych założeń dotyczących czasu, systemy rozproszone możemy podzielić na synchroniczne i asynchroniczne.

Systemy synchroniczne charakteryzują się tym, że możliwe jest ustalenie górnego limitu czasu potrzebnego na dostarczenie dowolnej wiadomości. Mamy też wiedzę na temat szybkości działania poszczególnych komponentów, co pozwala nam na ustalenie maksymalnego czasu potrzebnego na wykonanie danego zadania. Zakładamy tez, że system posiada dostęp do zegara globalnego.

Systemy asynchroniczne charakteryzują się brakiem możliwości przyjęcia założeń dotyczących czasu wykonywania zadań i dostarczenia wiadomości. Zakładamy, że przy komunikacji mogą wystąpić dowolne opóźnienia. Dodatkowo, nie mamy dostępu do zegara globalnego.

Rzeczywiste systemy rozproszone przez większość czasu wykazują własności systemów synchronicznych. Przy poprawnym działaniu opóźnienie dostarczenia wiadomości nie przekracza pewnego możliwego do ustalenia czasu. Niestety, założenie to przestaje być prawdziwe w momencie wystąpienia różnych nieprzewidzianych sytuacji, np. opóźnień wynikających z problemów z siecią, zgubienia lub zduplikowania wiadomości, nieoczekiwanej awarii komponentu. Z tego też powodu zakłada się, że rzeczywiste systemy rozproszone są na ogół asynchroniczne. Niestety, utrudnia to konstruowanie algorytmów wykorzystywanych w systemach rozproszonych, np. protokołów konsensusu. Dlatego też w algorytmach tych przyjmuje się często pewne założenia dotyczące synchronizacji, które ułatwiają ich projektowanie.

2.2. Problem uzgadniania danych

W systemie rozproszonym często zachodzi konieczność uzgodnienia danych. Uzgodnienie polega on podjęciu przez komponenty wspólnej decyzji dotyczącej danej operacji. Osiągnięcie takiego porozumienia często jest niezbędne do zapewnienia prawidłowego działania systemu. Możemy to zobrazować na przykładzie problemu zatwierdzania rozproszonego. W problemie tym przyjmujemy, że pewną operację powinny zatwierdzić i wykonać wszystkie komponenty danego systemu, albo żaden z nich. Będzie to możliwe jedynie w sytuacji, kiedy zostanie osiągnięte porozumienie. Innym przykładem może być problem wyboru koordynatora, czyli procesu, który będzie nadzorował wykonanie danego zadania. Innym bardzo ważnym przykładem problemu uzgadniania danych jest problem konsensusu. Konsensus jest szeroko wykorzystywany w algorytmach, które wymagają uzgodnienia przez procesy w systemie wartości pewnej danej. Należy wspomnieć o tym, że podjęcie decyzji musi nastąpić w skończonym i możliwie krótkim czasie. W przeciwnym razie niemożliwe byłoby poprawne działanie systemu rozproszonego.

2.2.1. Problem bizantyjskich generałów

Problem bizantyjskich generałów został opisany przez Lesliego Lamporta za pomocą historii, w której grupa generałów próbuje uzgodnić, czy należy zaatakować miasto wroga, czy dokonać odwrotu. Zakładamy, że do podjęcia decyzji o ataku niezbędna jest zgoda większości generałów. Przyjmujemy też, że mogą oni porozumiewać się bez problemów. Uzgodnienie

następuje w wyniku wymiany wiadomości, w których generałowie informują siebie nawzajem o swoich preferencjach. Podstawowym problemem jest fakt, że część generałów jest zdrajcami i dąży do podjęcia błędnej decyzji, tzn. takiej, która jest niezgodna z opinią większości lojalnych generałów. W tym celu zdrajcy mogą np. przekazywać pozostałym sprzeczne wiadomości.

Zauważmy, że uzgodnienie ostatecznej decyzji nie jest możliwe, jeżeli w procesie bierze udział trzech generałów, z których jeden jest zdrajcą. Przykładowo, jeżeli jeden z lojalnych generałów opowiada się za atakiem, a drugi za odwrotem, zdrajca może wysłać do każdego z nich różną wiadomość, odpowiadającą ich preferencji. W takiej sytuacji jeden z wiernych generałów uzna, że podjęta została decyzja o ataku, a drugi, że o odwrocie. Zostało udowodnione, że jeżeli w uzgadnianiu decyzji bierze udział n generałów, z których f jest zdrajcami, to do osiągnięcia porozumienia niezbędne jest spełnienie zależności:

$$\boldsymbol{n \geq 3f + 1} \tag{2.1}$$

Opisana historia jest zobrazowaniem rzeczywistego problemu występującego w systemach rozproszonych. Jak wiemy, każdy z komponentów (generałów) może ulec awarii bizantyjskiej i zacząć działać w sposób nieprzewidziany (stać się zdrajcą). System jest odporny na awarie tego typu, jeżeli spełniona jest zależność opisana wzorem (2.1). Własność ta jest prawdziwa dla każdego przypadku związanego z uzgadnianiem danych pomiędzy komponentami w systemie rozproszonych, m.in. dla problemu konsensusu.

2.2.2. Problem elekcji lidera

Wiele algorytmów stosowanych w systemach rozproszonych wymaga obecności lidera, czyli specjalnego procesu, który koordynuje wykonywanie poszczególnych zadań. Przed rozpoczęciem wykonywania takiego algorytmu należy więc dokonać wyboru odpowiedniego procesu, który będzie spełniał tę rolę. Elekcja lidera należy do grupy problemów uzgadniania. Procesy w systemie muszą porozumieć się w sposób jednoznaczny, który z nich zostanie koordynatorem. Oznacza to, że procesy, które nie wygrały elekcji muszą się podporządkować tej decyzji. Zwycięzca zaś pozostaje liderem do czasu zakończenia wykonywania zadań lub do momentu, w którym ulegnie awarii. Jeżeli dany proces zauważy, że nastąpiła awaria koordynatora, rozpoczyna następną elekcję.

Aby możliwe było wybranie jednego lidera, niezbędne jest wprowadzenie pewnej formy jednoznacznej identyfikacji procesów. Dowolne dwa procesy muszą mieć bowiem możliwość ustalenia, który z nich powinien mieć pierwszeństwo przy wyborze. Przykładowo, każdy z nich może być identyfikowany przez pewną liczbę całkowitą. Proces z większym numerem miałby więc pierwszeństwo przy dokonywaniu elekcji.

Jednym z algorytmów, które są wykorzystywane do wyboru lidera, jest algorytm tyrana. Rozpoczyna się on w momencie, w którym proces w systemie spostrzega, że nie ma kontaktu z liderem. Wysyła wtedy wiadomość do wszystkich procesów z wyższym identyfikatorem. Jeżeli żaden z nich nie udzieli odpowiedzi, proces wysyłający zostaje liderem i algorytm się kończy. Z kolei proces o wyższym numerze po otrzymaniu wiadomości udziela odpowiedzi potwierdzającej, że przejmuje kontrolę nad elekcją, a następnie wysyła kolejną wiadomość do procesów o wyższym numerze od siebie. W ten sposób po pewnym czasie jeden z uczestników elekcji nie otrzyma odpowiedzi na swoją wiadomość, co będzie oznaczało, że został liderem. Rozsyła on wtedy do wszystkich pozostałych wiadomość, że przejął rolę koordynatora.

Innym przykładem algorytmu wyboru lidera może być np. algorytm pierścieniowy. Zakładamy, że procesy połączone są pierścieniem i każdy z nich posiada dwóch sąsiadów.

Elekcja rozpoczyna się, gdy jednej z procesów zauważy, że nie ma kontaktu z liderem. Wysyła on wtedy do swojego najbliższego systemu wiadomość ze swoim numerem. Odbiorca wiadomości przesyła wiadomość dalej, dokładając do niej swój własny numer. Jeżeli jeden z procesów nie odbierze wiadomości, wiadomość jest przesyłana dalej, do następnego w pierścieniu. W końcu wiadomość dociera do inicjatora elekcji. Wybiera on z listy najwyższy numer i wysyła w obieg wiadomość z informacją o zwycięzcy elekcji.

2.3. Pojęcie konsensusu

Konsensus w powszechnym rozumieniu oznacza zgodę, wspólne stanowisko wypracowane w wyniku nieraz długich dyskusji nad daną sprawą. W przypadku systemów rozproszonych rozumienie tego pojęcia jest bardzo podobne. Konsensus oznacza zgodę pomiędzy procesami dotyczącą wartości pewnej danej. Zakładamy, że każdy proces w systemie przedstawia na początku swoją propozycję tej wartości. Następnie, w wyniku wymiany wiadomości między procesami, następuje uzgodnienie i przyjęcie jednej z przedstawionych wcześniej propozycji. Mechanizm konsensusu należy do grupy problemów uzgadniania danych. Oznacza to, że przy podejmowaniu decyzji należy wziąć pod uwagę, że w systemie mogły nastąpić awarie i pewne procesy mogą być niepoprawne lub niedostępne, co utrudnia znacząco konstrukcję i implementację protokołów konsensusu. Protokół konsensusu to specjalny algorytm, który definiuje sposób osiągania konsensusu w systemie. Istniejące protokoły wykorzystują wiele różnych sposobów uzgadniania. Często wykorzystywana jest koncepcja głosowania, zgodnie z którą wszystkie procesy w systemie decydują się na wartość, która uzyska poparcie większości z nich. Wiele protokołów korzysta też z koncepcji lidera, który jest odpowiedzialny za nadzór nad całym procesem uzgadniania. Pomimo zróżnicowanego sposobu działania przyjęło się, że protokoły konsensusu powinny umożliwić spełnienie kilku podstawowych warunków, do których należą:

- Zakończenie (ang. termination) – decyzja dotycząca wartości musi zostać podjęta w skończonym czasie przez wszystkie poprawne procesy.
- Zgodność (ang. agreement) – wszystkie poprawnie działające procesy decydują się na tę samą wartość.
- Integralność (ang. integrity) – jeżeli wszystkie poprawne procesy zaproponowały tę samą wartość, to każdy z nich musi zdecydować się na przyjęcie tej wartości.

Protokoły konsensusu projektowane są tak, aby umożliwić tolerowanie awarii systemu. Część z nich, np. Paxos i Raft, umożliwia tolerowanie jedynie awarii zatrzymania, a więc nieprzewidzianego zatrzymania działania procesu. Mają one jednak wiele zalet, do których należą: mniejszy poziom skomplikowania, szybkość działania i korzystna złożoność komunikacyjna (określa ona, jak wiele wiadomości musi być wymienione przy podejmowaniu decyzji). Zalety te wynikają z faktu, że wykrywanie i tolerowanie awarii zatrzymania jest stosunkowo proste. W przypadku takich protokołów do osiągnięcia konsensusu zazwyczaj wystarczające jest poprawne funkcjonowanie większości procesów, tzn. musi być spełniony warunek:

$$n \geq 2f + 1 \tag{2.2}$$

gdzie f – liczba procesów, które uległy awarii zatrzymania, n – liczba wszystkich procesów.

Protokoły konsensusu zaprojektowane do tolerowania awarii bizantyjskich są zazwyczaj bardziej skomplikowane, ponieważ proces uzgadniania w przypadku wystąpienia takiej awarii jest zdecydowanie trudniejszy. Wynika z tego też, że protokoły tego typu są wolniejsze i mają większą złożoność komunikacyjną, zaś do osiągnięcia konsensusu niezbędne jest spełnienie warunku opisanego wzorem (2.1). W przypadku niektórych protokołów warunek ten może być jeszcze bardziej zaostrzony, np. dla algorytmu Phase King wymagane jest spełnienie zależności $n \geq 4f$.

2.3.1. Twierdzenie FLP

Jak to już zostało wspomniane wcześniej, rzeczywiste systemy rozproszone są asynchroniczne. Przesyłane w nich wiadomości mogą zaginąć, zostać zniekształcone lub zduplikowane. Nie jest więc możliwe określenie górnego limitu czasu potrzebnego na przesyłanie wiadomości, który byłby zawsze prawdziwy. Nie możemy też założyć, że poszczególne procesy będą zawsze przetwarzały dane z określoną szybkością. Wynika z tego podstawowa trudność w projektowaniu protokołów konsensusu. Po raz pierwszy została ona określona w 1985 roku przez Michaela Fishera, Nancy Lynch i Michaela Patersona. Udowodnili oni, że w przypadku systemu asynchronicznego nie jest możliwe stworzenie algorytmu, który gwarantowałby osiągnięcie konsensusu, jeżeli istnieje możliwość awarii zatrzymania jednego z procesów. Własność ta jest nazywana twierdzeniem FLP (od pierwszych liter nazwisk naukowców). W dowodzie tego twierdzenia założono, że w systemie nie mogą zajść awarie bizantyjskie, które są znacznie trudniejsze do wykrycia i naprawy. Przyjęto też, że wiadomości dostarczane są poprawnie i nie są nigdy duplikowane. Naukowcy udowodnili, że, w przypadku, kiedy jeden procesów przestanie odpowiadać, możliwa jest sytuacja, w której uzgadnianie decyzji będzie trwało nieskończenie długo. Oznacza to, że nie istnieje protokół konsensusu, w którym możliwe jest zagwarantowanie, że zawsze spełniony będzie warunek zakończenia. Jednakże sytuacja, w której warunek zakończenia nie będzie nigdy spełniony, ma bardzo małe prawdopodobieństwo zaistnienia. Wobec tego możliwe jest przyjęcie pewnym założeń dotyczących systemu, które pozwalają na ominięcie twierdzenia FLP. Przykładowo, część protokołów konsensusu, przyjmuje górny limit czasu (ang. timeout), w którym decyzja musi zostać podjęta. Jeżeli czas minie, a wartość nie zostanie uzgodniona, wówczas proces uzgadniania rozpoczyna się od nowa. Rozwiązanie to wykorzystuje m.in. protokół konsensusu Raft.

2.3.2. Problem replikacji danych

Rozważmy przykład systemu zbudowanego w tradycyjnej architekturze klient-serwer. Klient wysyła zapytanie do serwera, który następnie łączy się z bazą danych i wykonuje zadane operacje. Po zakończeniu przetwarzania do klienta wysyłana jest odpowiedź. Architektura takiego systemu często wykorzystywana jest do przechowywania ważnych danych. Jednakże, gdy do dyspozycji mamy tylko jeden serwer, odporność całego systemu na awarie jest bardzo ograniczona. Zatrzymanie, nieprawidłowe działanie lub brak możliwości połączenia się z serwerem powoduje, że nie mamy dostępu do danych do czasu naprawienia awarii. Z tego też powodu systemy, które służą do przechowywania ważnych danych często są projektowane jako rozproszone – składają się z kilku serwerów. W przypadku, kiedy nastąpi awaria jednego z nich, klient może zwrócić się z zapytaniem do innego, który udzieli odpowiedzi. O ile awarii ulegnie odpowiednio mało komponentów, dostęp do danych nie jest ograniczony.

Aby tak opisany system był użyteczny, niezbędne jest zaprojektowanie skutecznego mechanizmu replikacji danych. Odpowiedzi udzielane przez poszczególne serwery będą poprawne, jeżeli każdy z nich będzie miał dostęp do aktualnej repliki wszystkich zapisanych danych. Gdy więc klient zażąda dodania do systemu jakichś danych, to powinny one zostać zapisane w pamięci każdego serwera. Analogicznie, jeżeli klient zażąda usunięcia lub modyfikacji danych, operacje te powinny zostać wykonane na wszystkich serwerach. Dzięki temu będziemy mieć pewność, że system zawsze udzieli klientowi poprawnej odpowiedzi na zapytanie.

2.3.3. Koncepcja automatu skończonego

W przedstawionym w poprzednim punkcie systemie każdy z serwerów oczekuje na pojawienie się zapytania klienta. Wysłanie żądania do serwera powoduje, że wykonuje on zadane operacje, a następnie udziela odpowiedzi, która jest zależna od typu zapytania. Proces ten może spowodować dodanie lub modyfikację przechowywanych danych. Mówimy wtedy, że zmianie uległ wewnętrzny stan serwera. Serwer pozostaje w stanie wynikowym do czasu otrzymania kolejnego zapytania. Wynika z tego, że serwer może zostać zdefiniowany za pomocą pewnego automatu, w którym sygnały wejściowe odpowiadają komendom wywoływanym przez użytkownika, a sygnały wyjściowe – odpowiedziom generowanym przez serwer. Wykonanie serii operacji powoduje przejścia do kolejnych stanów. Przedstawiony automat możemy zdefiniować w sposób następujący:

- $\boldsymbol{S}$ - zbiór wszystkich stanów,
- $\boldsymbol{C}$ - zbiór komend,
- $\boldsymbol{R}$ – zbiór odpowiedzi,
- $\boldsymbol{e}$: $\boldsymbol{C} \times \boldsymbol{S} \rightarrow \boldsymbol{R} \times \boldsymbol{S}$ – funkcja odpowiedzi systemu na wywołanie danej komendy.

Relacja $\boldsymbol{e}(C, S) = (R, S')$ oznacza, że wywołanie zapytania C w stanie S, spowodowało przejście do stanu S' i udzielenie odpowiedzi R. Tak zdefiniowany automat określany jest w języku angielskim jako maszyna stanu (z ang. *state machine*). W systemach rozproszonych często przyjmuje się, że zbiór wszystkich możliwych stanów jest skończony. W takim przypadku automat nazywamy skończoną maszyną stanu (z ang. *finite-state machine)*.

Zakładamy, że przedstawiony automat musi być deterministyczny. W przypadku wywołania w danym stanie S komendy C możliwe powinno być przejście tylko do jednego określonego stanu S'. Tak więc wykonanie w danym stanie automatu serii komend powoduje przejście przez jednoznacznie określoną ścieżkę stanów do jednego możliwego stanu końcowego. W przypadku opisywanego systemu rozproszonego oznacza to, że wynik poprawnego wykonania serii operacji na danych zawsze będzie taki sam. Każdy z serwerów wykona w te same działania na danych w tej samej kolejności, co spowoduje wygenerowanie tej samej serii odpowiedzi. Jeżeli stan początkowy każdego z serwerów był identyczny, to po wykonaniu operacji każdy z nich przejdzie do tego samego stanu.

Lista wykonanych przez automat operacji przechowywana jest w specjalnym logu, określanym w języku angielskim jako *replicated state log*. Każdy z serwerów w systemie zawiera lokalną kopię tego logu. W przypadku wystąpienia różnic w aktualnym stanie pomiędzy dwoma komponentami, możemy przypuszczać, że jeden z nich uległ awarii. Naprawa polega na analizie zawartości logu i przywróceniu zepsutego automatu do stanu, w którym wystąpiły różnice, a następnie odtworzeniu na nim kolejnych operacji. Po zakończeniu tego procesu automat znajdzie się w poprawnym stanie, co wynika z własności determinizmu.

Zapewnienie poprawności replikacji automatu skończonego jest więc możliwe, jeżeli zapewniona zostanie poprawność replikacji logu. W tym celu wykorzystywane są protokoły konsensusu. Przed wykonaniem danej operacji komponenty systemu rozproszonego porozumiewają się ze sobą na temat nowego wpisu do logu. Jeżeli dana operacja zostanie uzgodniona i zapisana w logu przez wszystkie poprawne serwery, wówczas jest ona wykonana przez system. W przypadku zastosowania mechanizmu konsensusu do zapewnienia poprawności replikacji, odporność systemu na jest zależna od zastosowanego protokołu. Tak więc, aby działanie systemu było poprawne, wymagane jest spełnienie warunku (2.1) albo (2.2).

Proces replikacji automatu skończonego z wykorzystaniem mechanizmu konsensusu jest podstawą wielu współcześnie tworzonych systemów rozproszonych, w tym systemów chmurowych i systemów służących do przechowywania ważnych danych.

3. Raft jako protokół konsensusu

3.1. Motywacja powstania

Protokół Raft został zaprojektowany w 2013 roku przez naukowców z uniwersytetu w Stanfordzie. Jego założenia i sposób działania opisali oni w artykule „W poszukiwaniu zrozumiałego algorytmu konsensusu" (z ang. *In Search of an Understandable Consensus Algorithm*). Główną motywacją autorów do stworzenia nowego protokołu był znaczny stopień skomplikowania dotychczas istniejących protokołów, w szczególności stosowanego w wielu systemach protokołu Paxos. Założenia Paxos powodują, że protokół ten jest trudny do zaimplementowania, a wymagana przez niego architektura jest niepraktyczna do zastosowań w rzeczywistych systemach rozproszonych. Dodatkowo, naukowcy wskazywali na problemy związane ze zrozumieniem jego działania, wykazywane przez studentów i początkujących informatyków. Przy pracy skupili się więc na tym, by nowy protokół Raft miał jasne i zrozumiałe założenia oraz by był przydatny i łatwy do zastosowania w rzeczywistych systemach.

3.1.1. Protokół Paxos

Paxos, opisany przez Lesliego Lamporta w 1989 roku, był jednym z pierwszych protokołów, które zapewniały bezpieczeństwo, poprawność i odporność na awarie w systemie rozproszonym. Jego implementacje z różnymi modyfikacjami są nadal wykorzystywane we współczesnych systemach rozproszonych, np. w technologii Chubby, stworzonej przez Google. Paxos został zaprojektowany do zastosowania w systemach asynchronicznych. Oznacza to, że, ze względu na twierdzenie FLP, nie gwarantuje, że osiągnięcie konsensusu będzie zawsze możliwe w skończonym czasie. Jednakże, prawdopodobieństwo zaistnienia takiej sytuacji jest bardzo niewielkie. Paxos zapewnia za to, że działanie systemu będzie poprawne, pomimo wystąpienia odpowiednio małej liczby awarii zatrzymania, problemów z siecią, duplikacją oraz utratą niektórych wiadomości. W swojej podstawowej wersji protokół ten nie zapewnia odporności na awarie bizantyjskie, wynikające z nieprzewidywalnego zachowania niektórych procesów.

Działanie protokołu Paxos opiera się na podziale procesów w systemie na role z określonymi obowiązkami i sposobem działania. Wyróżniamy następujące role procesów:

- Inicjator (ang. *proposer*) – oczekekuje na żądania klienta. Po nadejściu zapytania inicjuje działanie algorytmu konsensusu poprzez zgłoszenie nowej propozycji i koordynuje jego wykonanie. W systemie może znajdować się wiele procesów pełniących tę rolę.
- Głosujący, akceptujący (ang. *acceptor*, *voter*) – proces, który bierze udział w procesie uzgadniania. Każda propozycja przedstawiona przez inicjatora musi zostać zaakceptowana przez większość głosujących, aby mogła być przyjęta prze system
- Uczący się, uczeń (ang. *learner*) – proces, który wykonuje decyzję, która została wcześniej zaakceptowana przez większość akceptujących, a następnie zwraca wynik klientowi, który nadesłał żądanie. Rola uczącego jest często połączona z rolą inicjatora.
- Lider – najważniejszy z inicjatorów, odpowiedzialny za prawidłowe funkcjonowanie mechanizmu konsensusu, w danym momencie rolę lidera może pełnić tylko jeden proces

Proces uzgadniania danych w protokole Paxos jest podzielony na dwie fazy. Pierwsza z nich rozpoczyna się w momencie, w którym do inicjatora nadejdzie zapytanie pochodzące od klienta. Inicjator odbiera żądanie, a następnie przygotowuje wiadomość zawierającą pewną wartość n i wysyła ją do wszystkich akceptujących. Po otrzymaniu propozycji każdy z akceptujących sprawdza wartość n i w zależności od niej udziela odpowiedzi, w której informuje, czy akceptuje propozycję. Zgoda akceptującego na propozycję jest równoznaczna ze złożeniem tzw. obietnicy – zobowiązuje się on, że nie będzie więcej akceptował propozycji zawierających wartości n mniejsze niż ostatnio zgłoszona. Faza druga rozpoczyna się w momencie, w którym inicjator otrzyma wiadomości akceptujące od większości głosujących. Wysyła wtedy do nich drugą serię wiadomości, w których zawarty jest poprzednio przyjęty numer propozycji n i proponowana do uzgodnienia wartość v. Jeżeli akceptujący uzna, że wiadomość jest poprawna, wysyła do inicjatora i wszystkich uczących się informację, że wartość została uzgodniona. Po uzgodnieniu proces uczący się może wykonać zapytanie zgłoszone przez klienta i udzielić odpowiedzi. Opisany przebieg wiadomości w systemie został przedstawiony na poniższych rysunkach.

W przypadku, kiedy uzgadnianie zakończy się niepomyślnie, w celu przyjęcia propozycji inicjator musi rozpocząć algorytm od początku. Przyczyną niepomyślnego zakończenia działania algorytmu może być np. brak zgody większości akceptujących na przyjęcie propozycji w pierwszej lub drugiej fazie. Z konieczności zaakceptowania każdej propozycji przez kworum głosujących wynika, że protokół Paxos zapewnia odporność na awarię zatrzymania wtedy, kiedy spełniony jest warunek (2.2). Oznacz to, że liczba poprawnych procesów akceptujących musi być większa od liczby procesów, które uległy awarii, aby możliwe było osiągnięcie konsensusu. Dodatkowo, do pomyślnego zakończenia przebiegu całego procesu niezbędne jest poprawne działanie przynajmniej jednego inicjatora i ucznia.

Opisany przebieg uzgodnienia wartości według protokołu Paxos jest bardzo ogólny. W rzeczywistości wymieniane między procesami wiadomości zawierają więcej informacji. Widzimy jednak, że cały proces jest bardzo skomplikowany. Jak już wspominaliśmy twórcy protokołu Raft za główny cel postawili sobie stworzenie algorytmu, który byłby zdecydowanie łatwiejszy do zaprezentowania i zrozumienia.

3.1.2. Cechy protokołu Raft

W celu zapewnienia, że protokół Raft będzie łatwy do zrozumienia, autorzy zdecydowali się go zaprojektować tak, aby możliwa była dekompozycja problemu. Oznacza to, że stosowany algorytm możemy podzielić na kilka zagadnień, które mogą być rozważane niezależnie od innych. W protokole możemy więc wyróżnić następujące zagadnienia:

- Wybór lidera,
- Replikacja logu,
- Bezpieczeństwo.

Protokół Raft został zaprojektowany do zastosowania w systemach, w których konieczne jest zapewnienie poprawności replikacji danych. Osiągnięcie konsensusu opiera się więc na zapewnieniu, że każdy komponent w systemie będzie zawierał ten sam zestaw logów. Podobnie, jak w przypadku protokołu Paxos, Raft jest przeznaczony do zastosowania w systemach asynchronicznych, a więc teoretycznie możliwe są sytuacje, w których osiągnięcie konsensusu nie będzie możliwe. Raft nie zapewnia także odporności na awarie bizantyjskie. Opisywane założenia i algorytmy opierają się na założeniu, że procesy obecne w systemie działają poprawnie, o ile tylko są dostępne (nie uległy awarii zatrzymania i możemy się z nimi

połączyć). Komunikacja w systemie opiera się zaś na zastosowaniu zdalnego wywoływania procedur (RPC).

3.2. Wybór i rola lidera

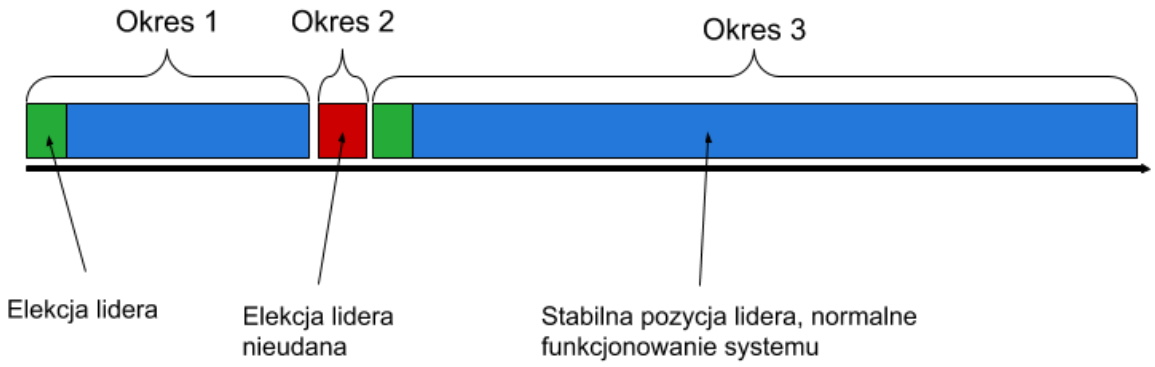
Liderem w protokole Raft nazywamy najważniejszy proces, który koordynuje działanie całego systemu. Lider posiada szczególne znaczenie w algorytmie. Od poprawnego działania i stabilności procesu, który pełni tę rolę, zależy prawidłowe działanie całego systemu. Wynika to z tego, że lider jako jedyny może rozpoczynać i koordynować proces replikacji logu do pozostałych procesów. Wszystkie zapytania przychodzące od klientów są więc przekierowywane do lidera, który jest odpowiedzialny za ich obsłużenie.

Podobnie, jak w przypadku protokołu Paxos, szczególnie ważne znaczenie ma pojęcie kworum. Jest to minimalna liczba procesów niezbędna do zapewnienia prawidłowości działania całego algorytmu konsensusu. W protokole Raft przyjmujemy, że kworum to większość procesów. Zatwierdzenie i wykonanie większości operacji, np. wyboru lidera i replikacji wpisu do logu, wymaga zgody większości. Zatem, poprawne funkcjonowanie jest możliwe tylko wtedy, gdy liczba procesów poprawnych jest większa od liczby procesów, które uległy awarii, czyli spełniony jest warunek (2.2).

3.2.1. Pojęcie okresu

Czas w protokole Raft jest podzielony na okresy, czyli przedziały czasowe o dowolnej długości. Jako początek każdego okresu przyjmujemy chwilę, w której rozpoczęty zostaje mechanizm wyboru nowego lidera. Końcem jest zaś moment rozpoczęcia kolejnej elekcji. Okresy w systemie są identyfikowane kolejnymi liczbami całkowitymi. Każdy proces zapamiętuje i przechowuje lokalnie aktualny numer. Numer ten jest szeroko wykorzystywany w mechanizmie osiągania konsensusu w systemie. Przykładowo, każde żądanie wysyłane przez lidera do innych procesów zawiera aktualny identyfikator okresu. Jeżeli odbiorca żądania zauważy, że numer zawarty w żądaniu jest mniejszy od tego, który sam przechowuje, odmawia wykonania polecenia i udziela odpowiedzi negatywnej. Z drugiej strony, jeżeli odbiorca otrzyma w żądaniu numer większy niż lokalny, może go zaktualizować. Identyfikator okresu jest szczególnie ważny w procesie wyboru lidera i replikacji logu. Możemy powiedzieć, że jest to pewna forma zegara systemowego, która pozwala wykrywać i naprawiać wiele możliwych błędów.

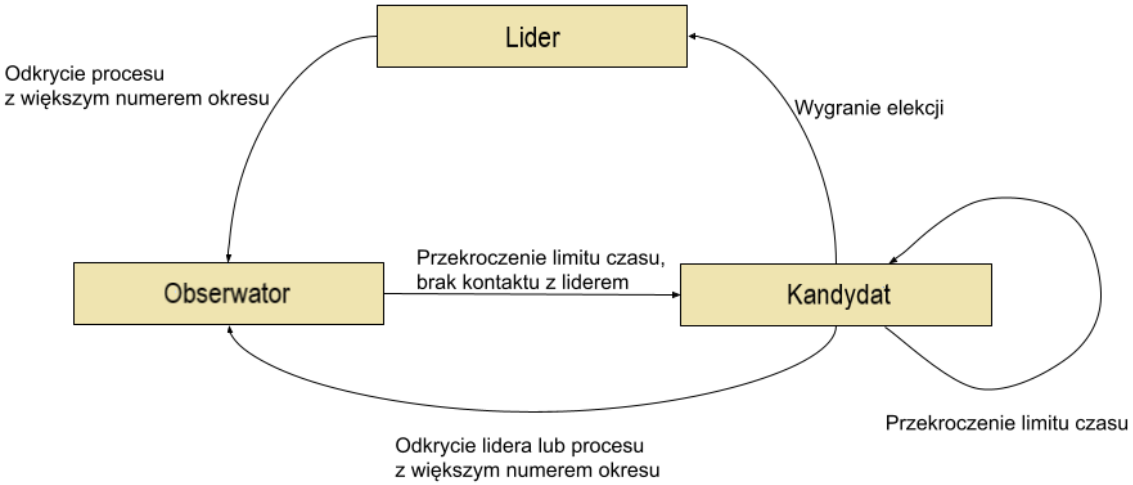
Przykładowy podział czasu w systemie na okresy został przedstawiony na Rys. 3.1. Pierwszy okres rozpoczyna się w momencie pierwszej elekcji. Zostaje wtedy wybrany lider i następuje normalne funkcjonowanie systemu. Końcem pierwszego okresu jest moment, w którym kontakt z liderem zostanie utracony przez większość procesów (np. z powodu awarii). Drugi okres rozpoczyna się próbą wyboru kolejnego lidera, która kończy się niepowodzeniem. W takim przypadku numer okresu jest zwiększony, a proces elekcji rozpoczyna się od nowa.



Rys. 3.2. Podział czasu na okresy w protokole Raft

3.2.2. Graf stanów procesu

Protokół Raft wyróżnia trzy możliwe stany, w których może znajdować się każdy proces. Przy poprawnym funkcjonowaniu systemu dokładnie jeden proces pełni role lidera, pozostałe zaś znajdują się w stanie obserwatora (ang. *follower).* Rola obserwatorów jest ograniczona. Ich głównym zadaniem jest branie udziału w głosowaniu nad wyborem lidera lub nowym wpisem do logu. Wszystkie żądania klienta przekazują oni do lidera. Każdy z obserwatorów oczekuje na komunikację pochodzącą od koordynatora. W sytuacji, kiedy przez dłuższy czas obserwator nie otrzyma żadnego sygnału od lidera, przechodzi do stanu kandydata. Kandydat, po znalezieniu się w tym stanie rozpoczyna elekcję nowego lidera. Jeżeli uda mu się ją wygrać, zostaje nowym liderem. Jeżeli zaś w trakcie procesu wyboru otrzyma informację, że w systemie jest już obecny lider, wraca do stanu obserwatora. Możliwa jest też sytuacja, w której lider lub kandydat dowie się, że przechowywany przez niego numer okresu jest mniejszy, niż ten zapisany przez inny proces. Przechodzi wtedy automatycznie do stanu obserwatora. Przejścia pomiędzy poszczególnymi stanami procesu zostały przedstawione na Rys. 3.2.

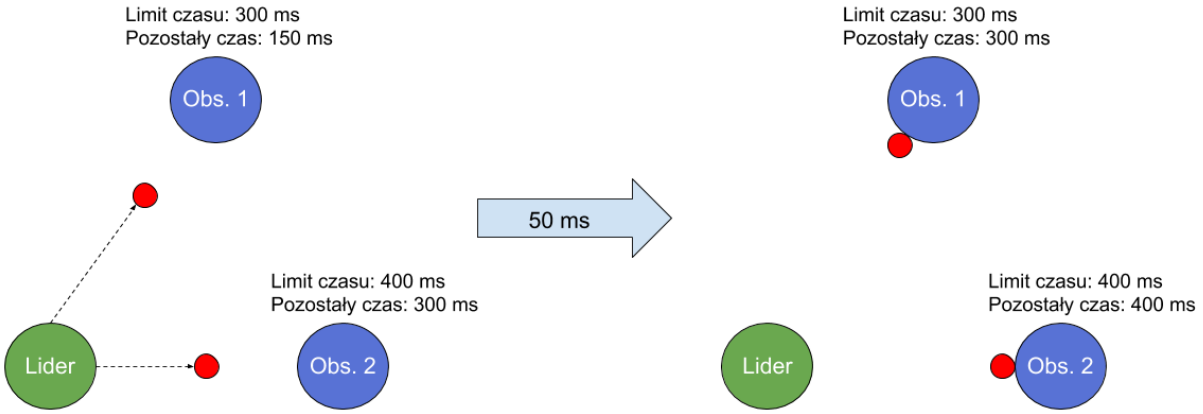


Rys. 3.2. Graf stanów procesu w protokole Raft

3.2.3. Mechanizm bicia serca

Efektywne działanie protokołu Raft jest zależne od dobrego działania i stabilności lidera. Każda kolejna elekcja powoduje przerwę w normalnym funkcjonowaniu systemu i może być przyczyną różnych nieprzewidzianych sytuacji. Z tego też powodu algorytm powinien zapewniać, że pozycja lidera będzie stabilna, a kolejne elekcje nie będą następowały w krótkich odstępach czasu. W tym celu stosowany jest mechanizm bicia serca. Każdy proces będący częścią systemu przechowuje wartość liczbową, która definiuje dopuszczalny limit czasu (ang. timeout), jaki może upłynąć pomiędzy kolejnymi komunikatami pochodzącymi od lidera. Jeżeli obserwator zauważy, że czas, jaki minął od otrzymania ostatniej wiadomości od koordynatora, przekracza dozwolony limit, rozpoczyna proces elekcji. Aby więc potwierdzić swoją pozycję, proces, który pełni rolę lidera, musi w określonych przedziałach czasowych wysyłać żądania do obserwatorów. W tym celu używa zdalnego wywołania procedury replikacji logu, która nie zawiera żadnego nowego wpisu do dodania. Obserwatorzy po otrzymaniu takiego żądania rozpoznają, że pochodzi ono od lidera, zapisują czas dostarczenia i resetują zegar elekcji. Przedstawiona procedura pozwala liderowi utrzymać swoją pozycję tak długo, dopóki będzie on w stanie dostarczać do obserwatorów wiadomości w odpowiednich przedziałach czasowych.

Przykład działania mechanizmu bicia serca został pokazany na Rys. 3.3. Przedstawiono na nim system składający się z trzech procesów: lidera i dwóch obserwatorów. Limit czasu obserwatora pierwszego został określony na 300 ms, a drugiego na 400 ms. Pozostały czas obserwatora pierwszego (150 ms) i drugiego (300 ms) oznacza, że tyle czasu będą oni jeszcze oczekiwać na kontakt. Proces lidera wysyła więc do każdego z nich wiadomość („bicie serca”), oznaczoną na rysunku czerwoną kropką. Po 50 ms wiadomości docierają do obu obserwatorów, którzy potwierdzają, że pochodzi ona od rzeczywistego lidera i resetują odliczanie czasu.



Rys. 3.3. Mechanizm bicia serca w protokole Raft

3.2.4. Proces elekcji

Proces wyboru lidera rozpoczyna się w momencie, kiedy jeden z obserwatorów przejdzie do stanu kandydata. Na początku kandydat zwiększa przechowywany przez siebie numer okresu o jeden. Następnie następuje proces głosowania. Kandydat głosuje na siebie, a następnie komunikuje się z pozostałymi procesami w systemie. Do każdego z nich wysyłane jest zapytanie, które ma formę zdalnego wywołania procedury. W parametrach kandydat przekazuje swój numer okresu oraz informacje o ostatnim posiadanym wpisie do logu. Na tej podstawie odbiorca decyduje o udzieleniu głosu. Przede wszystkim sprawdza, czy przekazany w procedurze numer okresu jest mniejszy niż ten, który sam przechowuje. Jeżeli tak, uznaje,

że kandydat nie posiada aktualnych informacji i nie udziela mu swojego głosu. Następnie odbiorca sprawdza, czy w aktualnym okresie brał już udział w elekcji. Obowiązuje bowiem zasada, że w danym okresie proces może udzielić głosu tylko jednemu kandydatowi. Odpowiedź negatywna udzielana jest również wtedy, kiedy log odbiorcy zawiera bardziej aktualne informacje niż log kandydata. Podsumowując, kandydat otrzymuje głos od danego odbiorcy, jeżeli spełnione są następujące warunki:

1. Numer okresu kandydata jest większy lub równy numerowi okresu odbiorcy.
2. Odbiorca nie poparł jeszcze żadnego kandydata w bieżącym okresie.
3. Log kandydata jest przynajmniej tak samo aktualny jak log odbiorcy.

Jeżeli kandydat otrzyma głosy większości procesów, wygrywa elekcję, zostaje nowym liderem i rozpoczyna działanie mechanizmu bicia serca. Przebieg typowej elekcji przedstawiony został na Rys. 3.4.

W czasie trwania elekcji może się zdarzyć, że kandydat otrzyma wiadomość od innego procesu, który uważa się za aktualnego lidera. Porównuje wtedy zapisany lokalnie numer okresu z tym przesłanym w wiadomości. Jeżeli numer kandydata jest taki sam albo mniejszy od numeru otrzymanego, wówczas kandydat automatycznie przechodzi do stanu obserwatora. W przeciwnym wypadku elekcja jest kontynuowana. Możliwa jest również sytuacja, w której dwa różne procesy znajdą się w stanie kandydata, rozpoczną elekcję i żaden z nich nie otrzyma wymaganej większości. Wówczas przyjmuje się, że doszło do tzw. podzielonej elekcji i nie został wyłoniony lider. Po pewnym czasie procesy zauważą ponownie, że nie mają kontaktu z liderem i rozpoczną kolejną elekcję. Aby taka sytuacja następowała jak najrzadziej, limity czasowe przechowywane przez poszczególne procesy są zróżnicowane. Dzięki temu w zdecydowanej większości przypadków jeden z nich zostaje kandydatem i wygrywa elekcję, zanim inne procesy zauważą brak kontaktu z liderem. Przykładowa podzielona elekcja została przedstawiona na rysunku (3.5).

3.3. Replikacja logu

Konsensus w systemie rozproszonym, opierającym się na replikacji automatu skończonego zostaje osiągnięty wtedy, gdy wszystkie procesy posiadają identyczne logi. Dwa logi są identyczne, jeżeli zawierają ten sam zestaw wpisów, ułożonych w tej samej kolejności. Zapewnienie poprawności replikacji kolejnych wpisów do logu jest najważniejszym zadaniem lidera w protokole Raft. Jak wiemy, lider odpowiada na wszystkie zapytania klientów, zatem jako jedyny może przyjmować od nich nowe wpisy do logu. Po akceptacji wpisu musi zapewnić, że zostanie on poprawnie zapisany również przez każdego z obserwatorów.

3.3.1. Mechanizm replikacji

Wpisy w logu są numerowane (indeksowane) przez kolejne liczby całkowite. Po otrzymaniu zapytania klienta lider zapisuje je do własnego logu wraz z odpowiednim numerem. Dodatkowo w logu zapamiętywany jest też okres, w którym nastąpił każdy zapis. Następnie lider komunikuje się z obserwatorami za pomocą zdalnego wywołania procedury. W parametrach wywołania umieszcza nowy wpis. Po otrzymaniu wiadomości odbiorca dodaje go do własnego logu i udziela pozytywnej odpowiedzi liderowi. Gdy lider otrzyma potwierdzenie zapisu od większości procesów, uznaje dany wpis za zatwierdzony. Wykonuje wtedy polecenie zawarte w zapytaniu i udziela odpowiedzi klientowi. Obserwatorzy zaś wykonują lokalnie dane

zapytanie po otrzymaniu informacji, że wpis został zatwierdzony przez lidera. Opisany mechanizm replikacji został przedstawiony na rysunku (3.6).

W sytuacji, w której lider nie otrzyma potwierdzenia dodania nowego wpisu od obserwatora, ponownie wywołuję procedurę zapisu, aż do momentu uzyskania pozytywnej odpowiedzi. Dzięki temu proces, który uległ awarii, będzie mógł po wznowieniu działania dodać wszystkie brakujące wpisy do własnego logu.

3.3.2. Własność zgodności logów

Przyjęty w protokole Raft sposób identyfikacji kolejnych wpisów do logu (indeks oraz numer serwera) pozwala na osiągnięcie dużego stopnia bezpieczeństwa i poprawności replikacji. Algorytm zapewnia bowiem, że spełniona jest własność zgodności logów, którą możemy zdefiniować następująco:

- Jeżeli dwa wpisy w różnych logach mają ten sam indeks i numer okresu, wówczas zawierają tę samą komendę,
- Jeżeli dwa różne logi zawierają wpis z tym samym indeksem i numerem okresu, to zawartość obu logów przed tym wpisem jest taka sama.

Jeżeli dwa wpisy zostały dodane do logu w tym samym okresie, to musiały zostać stworzone przez tego samego lidera. Wynika to z tego, że w danym okresie liderem może być tylko jeden proces. Lider zaś nie przypisuje unikalny numer indeksu do każdej nowej komendy użytkownika. Zatem, dwa wpisy w różnych logach identyfikowane przez te same numery indeksu i okresu na pewno zawierają tę samą komendę, co dowodzi, że prawdziwy jest punkt pierwszy własności zgodności logów. Punkt drugi wynika zaś ze sposobu działania procedury dodania nowych wpisów do logu obserwatora. W parametrach wywołania procedury lider umieszcza indeks i numer okresu wpisu, który znajduje się bezpośrednio przed wysyłaną obserwatorowi nową zawartością. Obserwator po otrzymaniu wiadomości sprawdza czy posiada we własnym logu zapis o numerach podanych w parametrach zdalnego wywołania procedury. Jeżeli tak, to otrzymana zawartość zostaje dodana do logu i zwracana jest odpowiedź pozytywna. W przeciwnym wypadku obserwator odmawia zapisu. Tak więc skuteczne dodanie każdego kolejnego wpisu oznacza, że log obserwatora zawiera również taką samą wcześniejszą zawartość jak log lidera.

3.3.3. Wykrywanie i usuwanie niespójności

Kiedy wszystkie procesy w systemie funkcjonują prawidłowo, problem niespójności logów nie występuje. Lider może bowiem wtedy skutecznie replikować kolejne wpisy do obserwatorów. W przypadku awarii systemu może się jednak zdarzyć, że poszczególne logi staną się niespójne. Przykładowo, lider może zapisać dany wpis do własnego logu, a następnie ulec awarii przed wywołaniem procedury dodania u obserwatorów. W takiej sytuacji zostanie wybrany nowy lider, który nie będzie jednak zawierał nowego zapisu, a więc logi staną się niespójne. Możliwe są sytuacje, w których log obserwatora będzie zawierał nadmiarowe wpisy lub będzie ich zawierał za mało. Niezależnie od tego, w algorytmie replikacji musi istnieć mechanizm zapewniający, że konflikty pomiędzy poszczególnymi logami zostaną wykryte i usunięte.

W celu wykrycia niespójności proces będący liderem przechowuje tablicę, w której dla każdego obserwatora zapisany jest numer wpisu, który powinien mu zostać dostarczony do

zapisania jako następny. Początkowo wartości te są o jeden większe od numeru indeksu ostatniego wpisu w logu lidera , a więc zakładamy, że logi są spójne (wszystkie komendy zostały już dostarczone). W sytuacji, kiedy obserwator odmówi dodania nowych wpisów, możemy przypuszczać, że jego log jest niespójny z logiem lidera.
Procedura przywrócenia spójności logów przebiega wtedy w sposób następujący:

1. Ustalenie indeksu i numeru okresu ostatniego wpisu, dla którego logi są spójne (wszystkie poprzednie wpisy są takie same),
2. Usunięcie z logu obserwatora wszystkich wpisów znajdujących się za ustalonym miejscem,
3. Przesłanie i zapisanie brakujących komend w logu obserwatora

Po otrzymaniu negatywnej odpowiedzi lider obniża wartość indeksu w tablicy dla danego obserwatora o jeden, a następnie ponownie używa zdalnego wywołania procedury dodania do logu. Obserwator ponownie sprawdza, czy w jego logu znajduje się wpis wskazany w parametrach wywołania procedury i udziela odpowiedzi. Proces ten jest ponawiany aż do momentu, w którym obserwator znajdzie wskazany indeks i numer okresu w swoim logu. Usuwa wtedy wszystko, co znajduje się w logu za znalezionym miejscem, a następnie dodaje wpisy przesłane przez lidera. Wywołanie procedury zapisu zakończy się sukcesem, a log obserwatora stanie się spójny z logiem lidera. Mechanizm usuwania niespójności przedstawiony został na rysunku (3.7).

3.4. Bezpieczeństwo

Metoda wyboru lidera i replikacji logu w protokole Raft zapewniają wysoki poziom bezpieczeństwa. Autorom algorytmu udało się udowodnić, że spełnia on następujące własności:

- Bezpieczeństwo elekcji,
- Własność zgodności logów,
- Kompletność lidera,
- Bezpieczeństwo automatu stanów.

Bezpieczeństwo elekcji oznacza, że w danym okresie rolę lidera może pełnić tylko jeden proces. Własność ta jest zapewniona za pomocą losowych wartości limitów czasowych dla poszczególnych procesów oraz konieczności uzyskania większości głosów w celu zdobycia pozycji lidera. Istnieją również mechanizmy pozwalające na rozwiązywanie konfliktu pomiędzy dwoma procesami uważającymi się za aktualnego lidera.

Zgodnie z własnością **kompletności lidera**, jeżeli w danym okresie zostanie zatwierdzony wpis do logu, to będzie on obecny w logu liderów we wszystkich kolejnych okresach. Dowód tej własności opiera się na przyjęciu założenia przeciwnego, a następnie wykazaniu, że jest ono sprzeczne. Dzięki temu możemy mieć pewność, że wszystkie komendy użytkownika, które udało się zatwierdzić będą wykonane przez wszystkie procesy w systemie.

Bezpieczeństwo automatu stanów to najważniejsza cecha, która zapewnia poprawność działania całego algorytmu. Mówi ona, że, jeżeli dany serwer wykonał komendę zapisaną w logu pod danym numerem, to żaden inny serwer nie wykona nigdy innej komendy dla tego numeru. Wynika z tego, że wszystkie procesy w systemie ostatecznie wykonają tę samą serię komend. Jest to równoznaczne z osiągnięciem konsensusu w całym systemie.

3.5. Tworzenie snapshotów

W rzeczywistym systemie rozproszonym użytkowanym przez wielu klientów, liczba wpisów w logu szybko wzrasta. Nieograniczony wzrost logu może być jednak procesem niekorzystnym, ze względu na większą zajętość pamięci i długi czas odtwarzania logu po wystąpieniu awarii procesu. Dlatego też w protokole Raft istnieje metoda skrócenia logu, która polega ona na zapisaniu aktualnego stanu systemu do tzw. snapshotu w pamięci. Snapshot obejmuje wszystkie wpisy, o których dany proces ma wiedzę, że zostały zatwierdzone. Zapisywane są również numer indeksu i okresu ostatniego wpisu. Następnie z logu usuwane są wszystkie wpisy, aż do miejsca, w którym został stworzony snapshot. Dzięki temu długość logu zostaje znacząco zredukowana. W protokole Raft przyjęto zasadę, że każdy proces, niezależnie od roli w systemie, może stworzyć snapshot samodzielnie. Mechanizm ten jest bezpieczny, ponieważ mamy pewność, że wszystkie wpisy wchodzące w skład snapshotu zostały już zreplikowane w systemie