Niezawodne systemy rozproszone

Cezary Sobaniec
Jerzy Brzeziński

Tolerowanie awarii

- Awaria częściowa (ang. *partial failure*)
  - automatyczne czynności naprawcze w przypadku systemu rozproszonego
- Uodpornienie procesów
- Niezawodna komunikacja grupowa
- Rozproszone zatwierdzanie – problem konsensusu
- Odtwarzanie

Systemy rozproszone potencjalnie mogą charakteryzować się wyższą odpornością na awarie. Wynika to z rozproszenia i być może zwielokrotnienia pewnych zasobów. Awaria częściowa (ang. *partial failure*) może w systemie rozproszonym spowodować uszkodzenie pojedynczej składowej systemu, która może być zastąpiona inną i nie musi powodować zatrzymania całego systemu. W systemach scentralizowanych awaria częściowa z reguły powoduje unieruchomienie całej aplikacji.

Większa odporność na awarie systemów rozproszonych jest tylko potencjalna i wymaga zastosowania różnych dodatkowych mechanizmów w celu automatycznego podejmowania czynności naprawczych w przypadku stwierdzeni awarii. Uodpornienie procesów może powodować, że awaria jednych nie będzie propagowała się na inne procesy. Problem ten pojawia się np. w kontekście komunikacji, szczególnie przy realizacji niezawodnego rozsyłania do grupy procesów.

Ważną cechą wielu aplikacji jest niepodzielność. Dotyczy to np. realizacji transakcji, której poszczególne kroki powinny zostać wykonane w całości. Realizacja niepodzielności w systemie rozproszonym wymaga zastosowania rozproszonego zatwierdzania, które jest równoważne problemowi konsensusu.

Jeżeli w systemie dochodzi już do awarii (węzła, procesu) to można podjąć próby automatycznego usunięcia skutków takiej awarii. Można w tym celu zastosować metody odtwarzania w celu przywrócenia stanu systemu rozproszonego sprzed awarii.

System wiarygodny

Wiarygodność — pewność działania systemu, która pozwala mieć uzasadnione zaufanie do usług, które ten system dostarcza

Cechy systemu wiarygodnego (ang. *dependable system*)

1. dostępność (ang. *availability*) – w każdym momencie
2. niezawodność (ang. *reliability*) – cały czas
3. pewność (ang. *safety*) – brak katastrofalnych skutków
4. pielęgnowalność (ang. *maintainability*) – naprawy

- bezpieczeństwo (ang. *security*)

Niezawodność systemu wiąże się z nieco ogólniejszą koncepcją systemów wiarygodnych (ang. *dependable systems*). Wiarygodność systemu oznacza w ogólności pewność działania systemu, która pozwala mieć uzasadnione zaufanie do usług, które ten system dostarcza. Konkretyzując tą definicję można wymienić następujące szczegółowe oczekiwania względem systemu: dostępność, niezawodność, pewność i pielęgnowalność.

Dostępność (ang. *availablility*) jest gotowością systemu do natychmiastowego działania. Jest opisywana prawdopodobieństwem, że w dowolnej chwili system jest w stanie spełniać swoje funkcje.

Niezawodność (ang. *reliability*) oznacza zdolność do ciągłego działania. W przeciwieństwie do dostępności, opis niezawodności odnosi się do przedziału czasu a nie do pojedynczego momentu. System, który wyłącza się na 1ms co godzinę ma dostępność na poziomie 99,9999%, ale jest bardzo zawodny. System, który jest niezawodny, ale jest wyłączany raz do roku na 2 tygodnie ma dostępność na poziomie 96%.

Pewność (ang. *safety*) oznacza, że chwilowa awaria pojedynczego komponentu nie powoduje katastrofalnych skutków.

Pielęgnowalność (ang. *maintainability*) dotyczy łatwości naprawiania uszkodzonego systemu. Wysoka pielęgnowalność ułatwia osiągnięcie wysokiej dostępności, ponieważ awarie mogą być automatycznie wykrywane i usuwane.

Jednocześnie od systemów wiarygodnych oczekuje się, że będą one **bezpieczne** (ang. *security*).

Podstawowe definicje

System **zawodzi** (ang. *fails*) gdy nie spełnia swoich obietnic; nie działa zgodnie ze specyfikacją

Błąd (ang. *error*) – część stanu systemu, wskutek którego może dojść do awarii (np. błąd transmisji danych)

Wada (ang. *fault*) – przyczyna błędu. Kontrolowanie wad: zapobieganie, usuwanie, przewidywanie

Awaria (ang. *failure*) – widoczna niezdolność do wykonywania wymaganej funkcji; jest skutkiem wystąpienia błędu powodowanego wadą

Tolerowanie awarii (ang. *fault tolerance*) – możliwość świadczenia usług pomimo występowania wad

System komputerowy **zawodzi** (ang. *fails*) gdy jego działanie przestaje być zgodne ze specyfikacją, a więc np. jedna z usług przestaje być dostępna.

Błąd (ang. *error*) jest częścią stanu systemu, wskutek którego może dojść do awarii. Błędy mogą się pojawiać np. podczas transmisji danych lub podczas wykonywania operacji arytmetycznych.

Wada (ang. *fault*) to przyczyna błędu. Znalezienie przyczyny błędu jest warunkiem koniecznym skutecznego usunięcia błędów w przyszłości.

Konsekwencją wystąpienia błędu jest **awaria** (ang. *failure*) systemu. Awaria oznacza widoczną niezdolność systemu lub jego składowej do wykonywania wymaganej funkcji w określonych granicach.

Ponieważ wiele wad jest efektem zewnętrznych przyczyn, niezależnych i niekontrolowanych przez system (np. pogoda), konieczne staje się ich kontrolowanie. Można to realizować poprzez zapobieganie ich występowaniu, usuwanie wad lub ich przewidywanie. W praktyce istotna jest możliwość **tolerowania awarii** (ang. *fault tolerance*), a więc możliwości świadczenia usług pomimo występowania wad.

Klasyfikacja wad

1. **Wady przejściowe** (ang. *transient faults*) – pojawiają się i znikają; powtórzenie operacji może spowodować zanik wady
   – np. pogoda przy transmisji bezprzewodowej
2. **Wady nieciągłe** (ang. *intermittent faults*) – pojawiają się i znikają samoistnie; trudne do wykrycia
   – np. luźny styk
3. **Wada trwała** (ang. *permanent fault*) – nie ustępuje aż do naprawy
   – np. przepalenie układu, błąd w oprogramowaniu

Wady, a więc przyczyny pojawiających się w systemie błędów, mogą być bardzo różne. W zależności od ich charakteru należy przedsięwziąć różne środki zaradcze stąd ważna jest klasyfikacja różnych typów wad.

Wady przejściowe (ang. *transient faults*) pojawiają się i znikają, mają charakter przejściowy. Właściwym sposobem postępowania z tego typu wadami jest powtórzenie operacji, ponieważ sytuacja taka może już nigdy nie wystąpić. Tego typu wady mogą się pojawić w wyniku przejściowych warunków pogodowych lub w wyniku pojawienia się przejściowych zakłóceń ze strony zewnętrznych urządzeń czy zwierząt.

Wady nieciągłe (ang. *intermittent faults*) również pojawiają się i znikają, ale ich cechą charakterystyczną są nawroty. Przykładem tego typu wady może być np. luźny styk. Wady tego typu są trudne do wykrycia, ponieważ mogą nie pojawiać się podczas monitorowania systemu w celu ich detekcji.

Wady trwałe (ang. *permanent faults*) nie znikają do momentu ich naprawy. Jest to wynikiem np. uszkodzenia jakiegoś podzespołu lub błędu w oprogramowaniu.

Modele awarii (1)

Załamanie (ang. *crash failures*) – trwałe zatrzymanie systemu, który dotąd pracował poprawnie

Błąd ominięcia (ang. *omission failures*) – brak odpowiedzi z serwera; serwer nie odbiera żądań lub nie wysyła odpowiedzi

Awaria odliczania czasu (ang. *timing failures*) – odpowiedź nie pojawia się w określonym przedziale czasowym

Błąd odpowiedzi (ang. *response failure*) – niepoprawna odpowiedź; błędna odpowiedź na żądanie lub **awaria zmiany stanu** (ang. *state transition failure*)

System, który ulega awarii przestaje świadczyć usługi. Nie zawsze jednak powodem awarii jest ten serwer, do którego kierowane jest bezpośrednio żądanie. Możliwe jest bowiem, że serwer ten jest blokowany brakiem dostępu do usług innych serwerów i z tego powodu nie daje odpowiedzi. Dla lepszego zrozumienia różnych typów awarii zaproponowano kilka kategorii. Slajd przedstawia jedną z takich klasyfikacji.

Awaria typu **załamanie** (ang. *crash failure*) oznacza trwałe załamanie systemu, który do tej pory działał poprawnie. W praktyce oznacza to np. zawieszenie systemu operacyjnego.

Błąd ominięcia (ang. *omission failure*) oznacza brak odpowiedzi ze strony serwera. Błąd taki może oznaczać, że żądanie nie dociera do serwera lub nie jest w nim obsługiwane. Tego typu sytuacja nie zakłóca pracy serwera, ponieważ brak żądania nie zmienia stanu serwera. Innym możliwym powodem braku odpowiedzi z serwera jest problem z dostarczeniem odpowiedzi na żądanie klienta. Serwer w takiej sytuacji odbiera żądanie, przetwarza je i próbuje wysłać odpowiedź, która jednak z różnych powodów nie dociera do klienta. Sytuacja taka wymaga od serwera, aby był przygotowany na powtórne przesłanie tego samego żądania przez klienta.

Kolejną kategorię stanowią **awarie odliczania czasu** (ang. *timing failures*) oznaczające brak odpowiedzi ze strony serwera w określonym przedziale czasowym. Dotyczy to zarówno odpowiedzi przychodzących zbyt szybko (klient może być niegotowy na odbiór wiadomości) lub – co jest sytuacją bardziej typową – odpowiedź przychodzi zbyt późno. W drugim przypadku mówimy również o *awariach efektywności*.

Kolejną kategorię stanowią **błędy odpowiedzi** (ang. *response failures*), oznaczające błędną odpowiedź serwera. Błędna odpowiedź może być generowana w odpowiedzi na faktyczne żądanie klienta. Inna sytuacja może wystąpić w przypadku przesłania niepoprawnego czy niezrozumiałego przez serwer żądania. Może wtedy nastąpić **awaria zmiany stanu** (ang. *state transition failure*). Polega ona na wykonaniu przez serwer niestandardowych działań, powodujących zmianę stanu serwera, która nie powinna nastąpić.

Modele awarii (2)

Awarie bizantyjskie (dowolne) (ang. *Byzantine failures*, *arbitrary failures*) – najogólniejsza kategoria; serwer odpowiada w sposób całkowicie nieprzewidywalny

Zatrzymanie (ang. *stopping failures*) – serwer przestaje zwracać wyniki w sposób dający się stwierdzić w innych procesach

Awarie uciszające (ang. *fail-silent*) – awarie przejściowe, związane np. z niedostatkami efektywności

Awaria bezpieczna (ang. *fail-safe*) – awaria, której fakt zaistnienia może być łatwo stwierdzony przez inne procesy

Najogólniejszą kategorią awarii są **awarie bizantyjskie**, zwane również **dowolnymi** (ang. *Byzantine failures*, *arbitrary failures*). Awaria bizantyjska oznacza, że serwer może wygenerować całkowicie dowolną i niespodziewaną odpowiedź. Jego działanie może być nawet złośliwe i może angażować inne serwery w celu zafałszowywania odpowiedzi.

Zatrzymanie (ang. *stopping failure*) oznacza, że serwer przestaje zwracać wyniki w sposób dający się stwierdzić przez inne procesy. Najłagodniejszą formą zatrzymania jest omówione wcześniej załamanie. Inną formą jest np. błąd ominięcia.

Serwery z reguły nie uprzedzają innych, że mają zamiar ulec awarii. Jest to szczególnie problematyczne w systemach z **awariami uciszającymi** (ang. *fail-silent systems*). Procesy dokonując oceny pracy serwerów mogą dojść do przekonania, że uległy one awarii, pomimo że awaria jest tylko przejściowa i polega jedynie na zwolnieniu pracy, a więc na niedostatku efektywności.

Generowanie całkowicie losowych odpowiedzi przez serwer może być w stosunkowo prosty sposób wykryte przez procesy. Tego typu awaria jest awarią dowolną, ale o łagodnej formie. Mówimy, że jest to **awaria bezpieczna** (ang. *fail-safe*).

Maskowanie awarii

- Tolerowanie awarii → ukrywanie ich występowania
- Maskowanie przez nadmiarowość
- Nadmiarowość
  - **informacji** – dodatkowe bity (np. kod Hamminga) w celu maskowania szumów transmisji
  - **czasu** – powtórzenie wykonywania operacji (np. transakcja) – pomocne w przypadku awarii przejściowych lub nieciągłych
  - **fizyczna** – realizowana sprzętowo lub programowo (zwielokrotnianie procesów)

Tolerowanie awarii w praktyce oznacza ukrywanie (maskowanie) faktu ich występowania przed innymi procesami czy użytkownikami końcowymi. Podstawową metodą maskowania awarii jest zastosowanie nadmiarowości. Nadmiarowość może przyjmować różne formy: nadmiarowości informacji, czasu lub nadmiarowości fizycznej. **Nadmiarowość informacji** oznacza, że wymiana informacji realizowana jest z wykorzystaniem dodatkowych bitów umożliwiających odtworzenie informacji, która została zafałszowana. Przykładowo kod Hamminga umożliwia maskowanie szumów linii przesyłowej poprzez przesyłanie dodatkowych bitów korekcyjnych.

Nadmiarowość czasu polega na powieleniu wykonywania pewnych czynności w przypadku zajścia awarii. Przykładem mogą tu być transakcje. Wycofaną transakcję można ponownie zlecić systemowi do wykonania, licząc, że awaria, która spowodowała wcześniejsze wycofanie miała charakter przejściowy.

Ostatecznie **nadmiarowość** może być realizowana poprzez **fizyczne** zwielokrotnienie jednostek przetwarzających. Zwielokrotnienie takie może być realizowane zarówno sprzętowo jak i programowo. W przypadku zwielokrotnienia programowego mamy do czynienia *de facto* ze zwielokrotnieniem procesów. Zwielokrotnienie sprzętowe spotykane jest często w układach elektronicznych, które realizują przetwarzanie równolegle na tych samych danych i porównują wyniki w celu oszacowania poprawności działania układu.

Odporność procesów – grupy procesów

- Zwielokrotnianie procesów do postaci grup procesów
- Niezawodna komunikacja grupowa
- Grupy są dynamiczne
- Grupy
  - równouprawnione
    - brak pojedynczego punktu awarii
    - skomplikowane podejmowanie decyzji
  - hierarchiczne
    - wrażliwość na utratę koordynatora
    - decyzje podejmowane przez koordynatora

Tolerowanie wadliwego procesu jest możliwe poprzez zorganizowanie grupy procesów, które będą realizowały te same zadania i wzajemnie się kontrolowały. Zlecenia wysyłane do systemu będą przekazywane przez warstwę komunikacji grupowej realizującej niezawodne rozsyłanie wiadomości do wszystkich członków grupy. Awaria pojedynczego procesu nie wstrzyma w takiej sytuacji transmisji wiadomości do pozostałych procesów, co umożliwi przejęcie odpowiedzialności przez proces, który nadal jest poprawny.

Grupy są dynamiczne, co oznacza, że procesy mogą dołączać się do nich i odłączać podczas pracy systemu. Co więcej: procesy mogą w tym samym czasie należeć do kilku grup. Potrzebny jest więc mechanizm zarządzania grupami i przynależnością do nich.

Przynależność do grup

- Serwer grup (ang. *group server*)
  - prosta koncepcja i realizacja
  - rozwiązanie scentralizowane – wrażliwość na awarie
- Dołączanie do grupy i jej opuszczanie
- Synchronizacja dołączania/opuszczania z komunikacją
  - dołączanie/odłączanie jako ciąg komunikatów wysyłanych do grupy
- Odbudowa grupy
  - inicjowana przez wiele węzłów jednocześnie

Organizacja procesów w grupy oznacza, że trzeba zorganizować system zarządzania nimi, umożliwiający dołączanie nowych procesów do grupy, odłączanie od grupy i rozsyłanie komunikatów do grupy. Najprostszym podejściem jest zastosowanie **serwera grup** (ang. *group server*), który centralnie przechowywałby pełną informację o utworzonych w systemie grupach i przynależności poszczególnych procesów do grup. Rozwiązanie to jakkolwiek koncepcyjnie proste i łatwe do zrealizowania, obarczone jest typowymi wadami podejścia scentralizowanego, a więc m.in. wrażliwością na awarie centralnego serwera. W rozwiązaniu rozproszonym dołączanie do grupy może być zrealizowane poprzez rozesłanie do wszystkich procesów odpowiedniego komunikatu informującego o dołączeniu do grupy. Trudniejsza jest jednakże realizacja odłączenia od grupy, szczególnie jeżeli jest ono następstwem zatrzymania procesu. W takiej sytuacji pozostałe procesy muszą po pierwsze stwierdzić fakt awarii jednego z członków grupy, a po drugie muszą wspólnie podjąć decyzję o usunięciu takiego procesu z grupy.

Oddzielną kwestią jest synchronizacja operacji dołączania i odłączania procesów do/z grup z jednocześnie zachodzącą wymianą komunikatów. Proces, który dołącza się do grupy powinien zacząć odbierać wszystkie komunikaty kierowane do niego, a proces, który odłącza się od grupy powinien przestać otrzymywać tego typu komunikaty. Osiągnięcie tej synchronizacji jest możliwe poprzez wykorzystanie mechanizmów komunikacji grupowej do samego zarządzania członkowstwem.

Rozproszone zarządzanie grupą może doprowadzić do sytuacji, kiedy procesy stwierdzą, że grupa uległa tak rozlicznym awariom, że dalsze jej funkcjonowanie wymaga rekonstrukcji. Działanie takie może być zainicjowane przez wiele procesów jednocześnie, co wymaga dodatkowej koordynacji.

Systemy rozproszone

Maskowanie awarii i zwielokrotnianie

- Protokoły oparte na kopii podstawowej, szczególnie protokół podstawa-zapas (ang. *primary-backup*)
  - hierarchiczna grupa procesów
  - modyfikacje wykonywane przez koordynatora
  - awaria: elekcja nowego koordynatora
- Protokoły zwielokrotnionych zapisów
  - równoprawna grupa procesów
  - aktywne zwielokrotnianie lub algorytmy kworum
- Tolerowanie *k* uszkodzeń
  - awarie uciszające: wystarczy *k* + 1 węzłów
  - awarie bizantyjskie: potrzeba 2*k* + 1 węzłów

Niezawodne systemy rozproszone (11)

Maskowanie awarii odbywa się poprzez organizowanie grup procesów i rozsyłanie do nich wszystkich w sposób niezawodny komunikatów. Wymaga to zastosowania zwielokrotnienia procesów, co może być realizowane na dwa podstawowe sposoby: poprzez protokoły oparte na kopii podstawowej lub poprzez protokoły zwielokrotnionych zapisów.

Protokoły oparte na kopii podstawowej stosują najczęściej model **podstawa-zapas** (ang. *primary-backup*), w którym wszystkie modyfikacje przetwarzane są najpierw przez wybrany węzeł koordynujący, a następnie propagowane do pozostałych węzłów. Odczyty mogą być realizowane współbieżnie przez dowolny węzeł będący kopią serwera podstawowego. Procesy są więc zorganizowane w postaci hierarchicznej grupy z wyróżnionym węzłem-koordynatorem. Wadą takiego rozwiązania jest oczywiście ryzyko awarii węzła-koordynatora. W takiej sytuacji należy wykonać algorytm elekcji i dokonać wyboru nowego węzła-koordynatora spośród kopii zapasowych.

Protokoły zwielokrotnionych zapisów działają na zasadzie **aktywnego zwielokrotniania** (ang. *active replication*) lub poprzez dobieranie **kworum**. W obu podejściach procesy są zorganizowane w równouprawnioną grupę. W aktywnym zwielokrotnianiu wszystkie żądania (również zapisy) przesyłane są w identycznej postaci do wszystkich węzłów i przetwarzane współbieżnie. Algorytmy oparte na kworum wymagają wykonania modyfikacji przez procesy tworzące kworum. Zaletą protokołów zwielokrotnionych zapisów jest brak elementu centralnego przetwarzania, ale pojawia się w nich dodatkowy koszt związany z rozproszoną koordynacją.

Istotną kwestią jest liczba replik procesów, jakie będą tworzone w systemie. Mówimy, że system jest w stanie tolerować *k* uszkodzeń, gdy jego funkcjonowanie będzie nadal poprawne po wystąpieniu *k* awarii. Osiągnięcie takiego poziomu odporności zależy od charakteru awarii występujących w systemie. Jeżeli awarie mają charakter uciszający, to do tolerowania *k* awarii wystarczy *k+1* procesów. Proces *k+1* będzie w stanie udzielić odpowiedzi po awarii *k* procesów. W przypadku awarii bizantyjskich, w najgorszym przypadku *k* błędnie działających procesów może generować złą, ale spójną odpowiedź. Stwierdzenie tego faktu jest możliwe pod warunkiem posiadania dostępu do *k+1* poprawnych węzłów. W efekcie do tolerowania *k* awarii bizantyjskich potrzeba *2k+1* węzłów.

Uzgodnienia w systemach wadliwych

Porozumienie (ang. *agreement*) — uzgodnienie uzyskane przez wszystkie sprawne procesy w skończonej liczbie kroków. Przykłady uzgodnień:

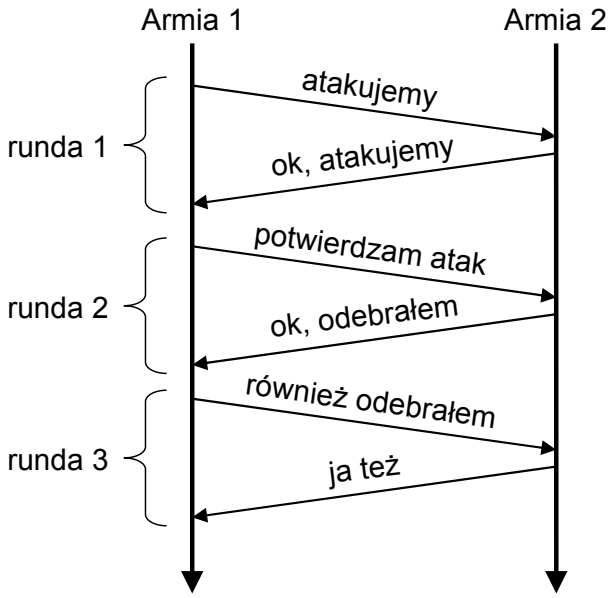
- wybór koordynatora
- zatwierdzenie transakcji
- synchronizacja

Przykład: problem dwu armii — koordynacja ataku dwóch armii niebieskich (po 3000 pododdziałów) na armię czerwoną (5000 pododdziałów). Armie niebieskie komunikują się poprzez posłańców.

Klient, podejmując decyzje na podstawie odpowiedzi z serwerów, może tolerować *k* awarii jeżeli ma dostęp do *2k+1* serwerów. Problem staje się jednak bardziej złożony gdy grupa procesów ma osiągnąć porozumienie. Porozumienie polega na podjęciu wspólnej decyzji i decyzja ta ma być podjęta w skończonej liczbie kroków. Z problemem uzgadniania mamy bardzo często do czynienia. Np. wybór koordynatora potrzebnego do pracy wielu algorytmów rozproszonych jest pewnego rodzaju porozumieniem. Zatwierdzenie bądź odrzucenie operacji atomowej wykonywanej w systemie rozproszonym (np. transakcji) również wymaga uzgodnienia. Innym przykładem może być ogólnie pojęta synchronizacja procesów, np. wzajemne wykluczanie – tu również mamy do czynienia z uzgadnianiem.

Problemy uzgadniania stają się bardzo trudne w systemach, w których mogą występować awarie. Klasycznym przykładem obrazującym możliwe trudności jest **problem dwu armii**. W problemie tym rozpatrywany jest atak dwóch armii niebieskich o sile 3000 pododdziałów każda, na armię czerwoną o sile 5000 pododdziałów. Armie niebieskie stacjonują w odległych miejscach i muszą wspólnie podjąć decyzję o ataku. Jedynie ich wspólny atak może zakończyć się zwycięstwem. Decyzja jest więc w tym przypadku binarna: atakować lub nie. Niestety komunikacja armii niebieskich wymaga wysyłania posłańców, którzy mogą być przechwyceni przez wroga – obrazuje to zawodny kanał komunikacyjny.

Problem dwu armii



Kanały komunikacyjne w problemie dwu armii są zawodne. Każdy komunikat może być więc utracony. Załóżmy, że generał pierwszej armii niebieskiej wysyła posłańca z wiadomością „atakujemy” do generała drugiej armii. Ten odebrawszy propozycję, odsyła posłańca z wiadomością „ok, atakujemy”. Generał 1 dostaje potwierdzenie, ale zauważa, że generał 2 nie wie, że on dostał potwierdzenie, a więc że może nie wiedzieć czy przeprowadzić atak czy nie. W związku z tym wysyła drugiego posłańca potwierdzającego decyzję o ataku. Generał 2 po odebraniu wiadomości stwierdza, że generał 1 nie wie, że jego posłaniec dotarł z potwierdzeniem, a skoro nie dotarł to generał 2 może zwątpić w plany ataku. Na wszelki wypadek wysyła więc kolejnego posłańca do generała 1, potwierdzającego odbiór wiadomości o potwierdzeniu ataku... Łatwo zauważyć, że bez względu na liczbę przesyłanych komunikatów (posłańców), generałowie nigdy nie uzyskają całkowitej pewności co do podejmowanej decyzji. Wynika to z tego, że zawsze ostatni przesyłany komunikat nie będzie potwierdzany, a jeżeli nie ma pewności, że komunikat ten dotarł, to nie ma również pewności, że cały protokół podejmowania decyzji został zakończony. Pomimo więc bardzo prostej sytuacji modelowej – mamy tu do czynienia z dwoma poprawnymi procesami (generałowie) – nie można podjąć decyzji, ponieważ komunikacja jest zawodna.

Problem bizantyjskich generałów

- Atak armii dowodzonej przez *n* generałów
- Komunikacja niezawodna punkt-punkt
- *m* generałów jest zdrajcami – usiłują nie dopuścić do porozumienia – nieprawdziwe i sprzeczne informacje
- Zadanie: porozumienie się lojalnych generałów co do liczebności oddziałów
- Założenie: zdrajcy nie współpracują
- Porozumienie: wektor liczebności oddziałów *L*:
  - *L*[i] liczebność oddziałów *i*-tego lojalnego generała
  - lub wartość nieokreślona w przypadku zdrajcy

Rozpatrzmy teraz przypadek, kiedy komunikacja jest niezawodna, ale procesy mogą ulegać dowolnym awariom. Klasycznym przykładem może być tutaj **problem bizantyjskich generałów**. W tym przykładzie również armia niebieska będzie atakować armię czerwoną, ale tym razem armia niebieska jest dowodzona przez *n* generałów, którzy muszą podjąć wspólnie decyzję. Do podjęcia decyzji jest potrzebna wiedza dotycząca liczebności podległych im oddziałów. Niestety *m* spośród wszystkich *n* generałów jest zdrajcami. Oznacza to, że ich odpowiedzi będą celowo nieprawdziwe i sprzeczne, próbując w ten sposób nie dopuścić do porozumienia się generałów lojalnych. Zakładamy tu dodatkowo, że generałowie zdrajcy nie współpracują.

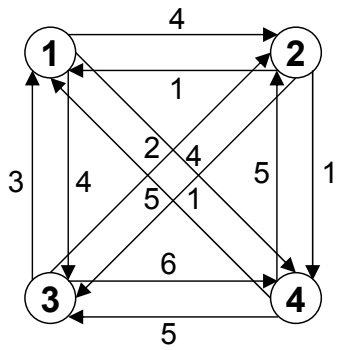
Ze względu na przewidywane dowolne awarie procesów, należy w tym przypadku inaczej zdefiniować porozumienie. Generałowie dążą do ustalenia liczebności oddziałów i wymieniają te informacje między sobą. Porozumieniem jest ustalenie wektora liczebności oddziałów przez poszczególnych generałów, w taki sposób, że jeżeli *i*-ty generał jest lojalny, to *i*-ta pozycja wektora zawiera liczebność jego oddziału. W przeciwnym razie pozycja ta będzie nieokreślona.

Algorytm Lamporta i innych

1. Wyślij do wszystkich generałów informację o liczebności swojego oddziału
2. Zbierz wyniki od wszystkich generałów
3. Przekaż wektor wyników (liczebności poszczególnych oddziałów) do wszystkich pozostałych generałów
4. Sprawdź pozycje *i*-te wszystkich wektorów
   - wartość będąca w większości staje się wynikiem
   - brak większości → pozycja nieokreślona

Jednym z algorytmów rozwiązujących problem bizantyjskich generałów jest algorytm Lamporta i innych. Jego idea polega na wstępnym rozesłaniu informacji o liczebności własnych oddziałów, a następnie na rozesłaniu informacji otrzymanych od innych generałów celem weryfikacji tak uzyskanych informacji.

Algorytm Lamporta i innych — przykład



Założenia:
n=4, *m*=1

Krok 2:
1. otrzymał (4, 1, 3, 5)
2. otrzymał (4, 1, 2, 5)
3. otrzymał (4, 1, 4, 5)
4. otrzymał (4, 1, 6, 5)

Krok 4:

1. otrzymał	2. otrzymał	4. otrzymał	
(4, 1, 2, 5)	(4, 1, 3, 5)	(4, 1, 3, 5)	
(3, 7, 1, 4)	(6, 6, 4, 4)	(4, 1, 2, 5)	
(4, 1, 6, 5)	(4, 1, 6, 5)	(3, 2, 1, 4)	
(4, 1, x, 5)	(4, 1, x, 5)	(4, 1, x, 5)	← wynik końcowy

Przykład na slajdzie pokazuje wykonanie algorytmu Lamporta dla 4 generałów, wśród których jeden jest zdrajcą, a więc *n*=4 i *m*=1. Warto zwrócić uwagę, że generał 3, który jest zdrajcą odpowiada za każdym razem inaczej, rozsyłając nieprawdziwe informacje. Pomimo tego pozostali generałowie są w stanie poprawnie ustalić liczebność swoich oddziałów, przegłosowując generała-zdrajcę.

Warunki osiągnięcia porozumienia

- W systemie z *m* wadliwymi procesami osiągnięcie porozumienia wymaga 2*m*+1 procesów poprawnych czyli 3*m*+1 procesów
- W systemie rozproszonym, w którym nie można zagwarantować dostarczania komunikatów w znanym, skończonym czasie, nie uda się osiągnąć żadnego porozumienia, gdy choć jeden proces jest wadliwy

Algorytm Lamporta gwarantuje poprawne określenie liczebności oddziałów lojalnych generałów, ale nie gwarantuje wykrycia generałów-zdrajców. Jeżeli bowiem generał-zdrajca będzie konsekwentnie udostępniał te same informacje wszystkim innym generałom, to nie zostanie wykryta żadna niezgodność i wartość, którą podał będzie uznana za poprawną.

Zmniejszenie liczby generałów do *n*=3 i pozostawienie jednego generała-zdrajcy spowoduje, że osiągnięcie porozumienia nie będzie możliwe. Udowodniono, że w systemie z *m* wadliwymi procesami porozumienie można osiągnąć wówczas, gdy istnieje 2*m*+1 procesów działających poprawnie, co daje ogólną liczbę 3*m*+1 generałów. Inaczej mówiąc, w systemie musi być co najmniej 2/3 procesów działających poprawnie.

Osiągnięcie porozumienia może wiązać się z wieloma innymi problemami. W asynchronicznym systemie rozproszonym, a więc w takim, w którym nie można zagwarantować dostarczenia komunikatów w znanym i skończonym czasie, osiągnięcie porozumienia jest niemożliwe, jeżeli choć jeden proces jest wadliwy (chodzi o awarie uciszające). Powodem jest niemożliwość rozróżnienia procesów, które uległy znacznemu spowolnieniu (ale nadal są poprawne) od procesów które uległy awarii typu załamanie.

Niezawodna komunikacja klient-serwer

- Komunikacja punkt-punkt — niezawodny protokół transportowy (np. TCP)
  - maskowanie błędów ominięcia (retransmisja)
  - załamanie – informacja dla klienta
- Komunikacja poprzez zdalne wywołania procedur (RPC)
  1. Klient nie potrafi odnaleźć serwera
  2. Zaginięcie żądania klienta
  3. Awaria serwera po odebraniu żądania
  4. Zaginięcie komunikatu z odpowiedzią
  5. Załamanie klienta po wysłaniu zamówienia

Komunikacja pomiędzy procesami w systemie rozproszonym najczęściej wykonywana jest za pośrednictwem niezawodnego, strumieniowego protokołu warstwy transportowej (np. TCP). Jest to wygodne, ponieważ protokół ten maskuje występowanie pewnych błędów, np. błędów ominięcia. Zagubienie komunikatu powoduje jego retransmisję, która jest całkowicie przezroczysta dla wyższej warstwy oprogramowania. Niektóre awarie jednak nie są maskowane. Z różnych powodów może dojść do załamania (zerwania) połączenia, co jest następnie sygnalizowane klientowi. Jedynym sposobem na maskowanie zerwanego połączenia mogłaby być automatyczna próba nawiązania go ponownie.

Na wyższym poziomie komunikacja może być realizowana z wykorzystanie zdalnych wywołań procedur (RPC) lub metod (RMI). Jedną z motywacji stosowania mechanizmu RPC było ukrywanie faktu rozproszenia wywołania procedury. Jeżeli w systemie występują awarie, to pojawiają się podczas przetwarzania sytuacje, które nie wystąpiłyby w przypadku lokalnego wywoływania procedur. Powoduje to, że zachowanie pełnej przezroczystości nie zawsze będzie możliwe. Błędy, które mogą się pojawić podczas realizacji zdalnego wywołania procedur można podzielić na 5 wymienionych klas.

RPC w sytuacjach awaryjnych (1)

1. Klient nie potrafi odnaleźć serwera
   - serwer jest wyłączony
   - serwer stosuje nowy interfejs (→ nowy pieniek)
   - obsługa błędu poprzez wyjątek lub sygnał
2. Zaginięcie żądania klienta
   - czasomierz
   - retransmisja żądania
   - środki do wykrywania powielonych żądań

Pierwsza kategoria błędów polega na niemożliwości odnalezienia serwera przez klienta. Niemożliwość taka może wynikać z problemów z lokalizacją serwera, jego wyłączeniem lub z zastosowaniem nowego interfejsu programowego. Serwer, który zmienia interfejs dostępu do usług dokonuje jednocześnie aktualizacji pieńka pośredniczącego w dostępie do niego. Klient stosujący stary pieniek nie będzie w takiej sytuacji w stanie dołączyć się do takiego serwera. Przedstawione typy błędów należy w jakiś sposób obsłużyć, najlepiej z zachowaniem przezroczystości wywołania metody. Jest to możliwe przy zastosowaniu mechanizmu wyjątków w nowszych językach programowania, takich jak język Java. W języku C można w tym samym celu zastosować np. sygnały.

Druga kategoria błędów występujących podczas realizacji zdalnych wywołań procedur to zaginięcie komunikatu z żądaniem klienta. Jest to kategoria, której poprawne obsłużenie jest najprostsze. Klient wysyłając komunikat z żądaniem ustawia czasomierz i po upływie ustalonego czasu uznaje, że komunikat nie dotarł do serwera, ponieważ nie odebrano odpowiedzi. W efekcie następuje retransmisja żądania. Jeżeli komunikat faktycznie został zagubiony, serwer obsłuży retransmitowane żądanie tak, jakby to był oryginał. Jeżeli komunikat jednak dotarł do serwera, to mechanizm wywołań zdalnych należy rozbudować o środki do wykrywania powielonych wywołań. W przypadku przeciążonego serwera klient może po kilku retransmisjach zrezygnować z wysyłania kolejnych. Następuje wtedy przejście do obsługi błędu typy „brak dostępu do serwera”.

RPC w sytuacjach awaryjnych (2)

3. Awaria serwera po odebraniu żądania
   - moment awarii: przed lub po wykonaniu operacji
   - możliwe sposoby obsługi:
     - semantyka co najmniej jednokrotna
     - semantyka najwyżej jednokrotna
     - brak gwarancji
     - semantyka dokładnie jednokrotna

Załamanie serwera może nastąpić po obsłużeniu żądania lub przed. W zależności od tego czy żądanie zostało obsłużone czy nie, reakcja klienta powinna być różna. Jeżeli serwer nie przystąpił do obsługi odebranego żądania, to żądanie takie powinno zostać retransmitowane przez klienta. Jeżeli serwer obsłużył żądanie i uległ awarii to powinno to zostać zgłoszone po stronie klienta. Problem polega jednak na tym, że klient nie jest w stanie tych dwóch sytuacji rozróżnić. Jedyne co stwierdza to fakt upłynięcia czasu od momentu wysłania żądania.

Istnieje kilka metod reakcji na awarię serwera. Pierwsza metoda polega na oczekiwaniu na wznowienie pracy serwera i przesłaniu ponownego żądania w celu uzyskania odpowiedzi. Metoda ta określana jest jako **semantyka co najmniej jednokrotna** (ang. *at least once semantics*). Gwarantuje ona, że każde wywołanie procedury zostanie wykonane przynajmniej jeden raz, ale być może, że również wiele razy. Drugie podejście polega na natychmiastowym przerwaniu prób komunikacji z serwerem po stwierdzeniu wystąpienia problemów. Jest to **semantyka co najwyżej jednokrotna** (ang. *at most once semantics*), która gwarantuje, że wywołanie procedury nastąpi najwyżej raz, ale być może, że nie nastąpi w ogóle. Najprostszym rozwiązaniem jest oczywiście nie gwarantowanie niczego. W takiej sytuacji wywołanie zdalne mogło nie nastąpić, mogło nastąpić raz lub wielokrotnie.

Oczywiście semantyki te nie są tym, czego oczekuje klient. Użytkownik oczekuje tak naprawdę **semantyki dokładnie jednokrotnej** (ang. *exactly once semantics*). Niestety semantyka taka jest niemożliwa do zrealizowania.

Podsumowując: możliwość załamania serwera wyraźnie różnicuje systemy scentralizowane i rozproszone w kontekście wykonywania procedur. W przypadku systemów scentralizowanych awaria serwera pociąga za sobą awarię klienta. W systemach rozproszonych awaria serwera wymaga obsłużenia tej sytuacji po stronie klienta.

RPC w sytuacjach awaryjnych (3)

4. Zaginięcie odpowiedzi
   - zaginięcie komunikatu czy spowolnienia serwera?
   - operacje idempotentne
   - numery kolejne zleceń + bit retransmisji
5. Załamanie klienta
   - pozostałe obliczenia – sieroty (ang. *orphan*)
   - obsługa sierot
     - eksterminacja – duży koszt
     - reinkarnacja (lub łagodna reinkarnacja)
     - wygaśnięcie

Klient nie odbierając potwierdzenia ze strony serwera nie ma możliwości poprawnego ocenienia zaistniałej sytuacji. Powodem braku odpowiedzi może bowiem być zaginięcie komunikatu z odpowiedzią potwierdzającą wykonanie operacji, ale równie dobrze źródłem braku odpowiedzi może być znaczące spowolnienie pracy przez serwer. Jedyną sensowną reakcją ze strony klienta jest w tym przypadku retransmisja żądania. Poprzednie zlecenie być może jednak dotarło do serwera i po retransmisji zostanie wykonane ponownie. W przypadku pewnych zleceń nie jest to problemem. Np. zlecenie odczytu fragmentu pliku od określonej pozycji, przy założeniu braku współbieżnego zapisu, zawsze da ten sam rezultat. Operacje takie są określane jako operacje **idempotentne** (ang. *idempotent*). Wiele operacji nie jest jednak idempotentnych. Np. usunięcie pliku przy ponownej próbie wykonania zwróci błąd, ponieważ pliku już nie będzie istniał. Podobnie wygląda sytuacja z tworzeniem nowego pliku. Rozwiązaniem jest numerowanie zleceń pochodzących od poszczególnych klientów. Serwer przed wykonaniem zlecenia sprawdza czy dane żądanie, pochodzące od konkretnego klienta, zostało już zrealizowane czy nie. Jeżeli było, to odsyła jedynie potwierdzenie, nie wykonując samej operacji. Dodatkowym zabezpieczeniem może być sygnalizowanie serwerowi, że dane żądanie jest retransmitowane, co powinno wzmóc czujność serwera. Rozwiązania te niestety wymuszają przechowywanie po stronie serwera informacji o klientach w postaci numerów ostatnio zrealizowanych zleceń. Informacje te będą niestety tracone przy ewentualnej awarii serwera.

Kolejnym problemem pojawiającym się podczas wykonywania zdalnych wywołań metod jest awaria klienta następująca po zleceniu zdalnego przetwarzania. Obliczenia zainicjowanie przez klienta mogą pozostać aktywne po stronie serwera niepotrzebnie go obciążając. Takie pozostałości nazywane są **sierotami** (ang. *orphan*). Sieroty mogą powodować problemy jeżeli np. powodowały zajmowanie zasobów. Również komunikaty zwrotne po zakończeniu przetwarzania mogą być mylące dla klienta, który wznowił swoje przetwarzania.

Istnieje kilka metod radzenia sobie z sierotami. Pierwsza, zwana **eksterminacją** (ang. *extermination*), polega na rejestrowaniu w pamięci trwałej (dysk) wszystkich zleceń przez namiastkę klienta i jawnym usuwaniu odpowiedzi-sierot. Wadą tego rozwiązania jest bardzo duży koszt związany z rejestrowaniem wywołań. Drugie rozwiązanie, zwane **reinkarnacją** (ang. *reincarnation*), polega na oznaczaniu każdego wywołania numerem epoki, oznaczającej logicznie upływający czas pomiędzy kolejnymi wznowieniami pracy klienta. Po wznowieniu pracy klient wysyła do wszystkich serwerów komunikat o nastaniu nowej epoki, który powoduje zatrzymanie wszystkich obliczeń tego klienta. Jeżeli jakaś sierota przeżyje na serwerze, do którego aktualnie nie ma dostępu, to i tak będzie mogła być łatwo wykryta poprzez stwierdzenie nieaktualnego numeru epoki. W łagodnej wersji reinkarnacji (ang. *gentle reincarnation*), serwer po odebraniu komunikatu o nastaniu nowej epoki próbuje ustalić właścicieli każdej z wykonywanych prac. Usuwane są tylko te zadania, dla których nie udało się ustalić właściciela. Ostatnią propozycją jest przydział ustalonego kwantu czasu zdalnej procedurze na jej wykonanie. Przekroczenie tego czasu jest możliwe tylko po uzyskaniu potwierdzenia od klienta. Jeżeli klient ulegnie awarii, potwierdzenie nie zostanie udzielone i automatycznie zadanie takie ulegnie anulowaniu. Metoda ta jest nazywana **wygaśnięciem** (ang. *expiration*).

Zatwierdzanie rozproszone

Zatwierdzanie rozproszone (ang. *distributed commit*) — wykonanie operacji przez wszystkich członków grupy (lub przez żadnego)

- Realizacja z wykorzystaniem koordynatora
- **Protokół zatwierdzania jednofazowego** (ang. *one-phase commit protocol*) – koordynator wskazuje rozproszonym procesom czy mają lokalnie wykonać (zatwierdzić) operację
  - lokalna operacja może nie być możliwa do zatwierdzenia

Zatwierdzanie rozproszone (ang. *distributed commit*) jest ogólnym problemem, w którym chodzi o wykonanie danej operacji przez każdego członka grupy procesów lub przez żadnego. Problem niezawodnego rozsyłania jest przykładem problemu zatwierdzania rozproszonego, w którym operacją, która ma być wykonana przez wszystkich (lub przez nikogo) jest dostarczenie konkretnej wiadomości.

Zatwierdzanie rozproszone b. często odbywa się przy udziale koordynatora. W najprostszym przypadku koordynator informuje procesy czy mają dokonać lokalnego zatwierdzenia wykonywanych operacji (np. transakcji). Podejście to określane jest jako **protokół zatwierdzania jednofazowego** (ang. *one-phase commit protocol*). W podejściu tym nie uwzględnia się sytuacji, w której lokalne zatwierdzenie operacji nie będzie możliwe, np. ze względu na zablokowane zasoby. W przypadku zatwierdzania rozproszonego potrzebne są więc bardziej rozbudowane schematy, np. **protokół zatwierdzania dwufazowego**.

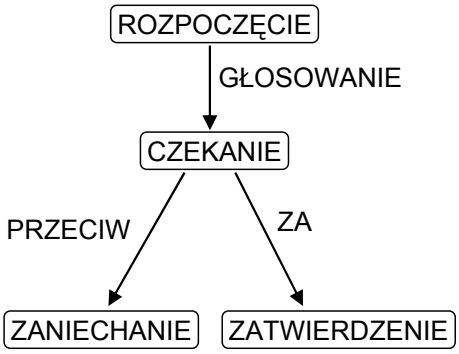
Protokół zatwierdzania dwufazowego – 2PC

1. Koordynator wysyła do wszystkich komunikat *GŁOSOWANIE*
2. Uczestnik głosowania odsyła komunikat *ZA* lub *PRZECIW*
3. Koordynator zbiera wyniki. Jeżeli wszyscy głosowali za zatwierdzeniem to wysyła komunikat *ZATWIERDŹ*. W przeciwnym wypadku wysyła komunikat *ZANIECHAJ*
4. Głosujący za zatwierdzeniem, jeżeli otrzymają komunikat *ZATWIERDŹ* zatwierdzają transakcję, w przeciwnym wypadku wycofują

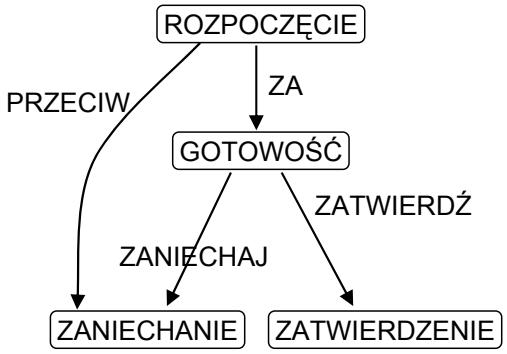
Protokół zatwierdzania dwufazowego (ang. *two-phase commit protocol*) uwzględnia fakt, że lokalne zatwierdzenie transakcji nie zawsze będzie możliwe. W związku z tym, przed zatwierdzeniem transakcji realizowane jest głosowanie mające na celu ustalenie czy wszyscy uczestnicy przetwarzania będą mogli transakcję zatwierdzić. Koordynator wysyła w tym celu komunikat *GŁOSOWANIE* do wszystkich uczestników przetwarzania, rozpoczynając w ten sposób głosownie. Uczestnik głosowania odpowiada komunikatem *ZA* lub *PRZECIW*, w zależności od tego czy może zatwierdzić operację lokalnie czy nie. Koordynator po odebraniu wyników głosowania podejmuje decyzję. Jeżeli wszyscy głosowali na tak, wysyła komunikat *ZATWIERDŹ*, powodujący globalne zatwierdzenie transakcji. Jeżeli pojawił się choć jeden głos na nie, wysyłany jest komunikat *ZANIECHAJ*. Uczestnicy przetwarzania, którzy głosowali na tak, po odebraniu komunikatu *ZATWIERDŹ* dokonują lokalnego zatwierdzenie transakcji. Jeżeli uczestnik głosował na nie, lub otrzymał komunikat *ZANIECHAJ*, dokonuje wycofania transakcji.

Stany w protokole 2PC

Koordynator



Uczestnik głosowania



Podstawową wadą protokołu 2PC jest jego wrażliwość na awarie, zarówno koordynatora jak i poszczególnych uczestników. W przetwarzaniu zarówno po stronie koordynatora jak i uczestników występują stany oczekiwania na komunikat (CZEKANIE u koordynatora, ROZPOCZĘCIE i GOTOWOŚĆ u uczestnika). Awaria któregoś z procesów może powodować w tym przypadku zablokowanie pozostałych, a więc prowadzi do niepowodzenia protokołu jako całości. W celu rozwiązania tego problemu można zastosować odliczanie czasu celem wykrycia awarii.

Uczestnik głosowania czeka na początku w stanie ROZPOCZĘCIE na komunikat GŁOSOWANIE. Jeżeli komunikat nie nadchodzi, uznaje, że transakcję należy wycofać. Jeżeli komunikat jednak nadejdzie, odeśle odpowiedź PRZECIW.

Koordynator jest blokowany w stanie CZEKANIE. Jeżeli w określonym czasie nie otrzyma odpowiedzi od wszystkich uczestników, podejmuje decyzję o wycofaniu transakcji (wyśle komunikat *ZANIECHAJ*).

Uczestnik może być także blokowany w stanie GOTOWOŚĆ. Jeżeli nie nadejdzie komunikat od koordynatora, to uczestnik może podjąć decyzję o wycofaniu transakcji. Lepszym rozwiązaniem jest jednak próba skomunikowania się z innymi uczestnikami w celu uzyskania informacji o decyzji koordynatora. Jeżeli uczestnik *P* dowie się od innego uczestnika *Q*, że on przeszedł do stanu ZATWIERDZENIE, to *P* również może zatwierdzić transakcję, ponieważ jedynym sposobem na przejście do stanu ZATWIERDZENIE jest uzyskanie od koordynatora komunikatu *ZATWIERDŹ* wskazującego na taką decyzję koordynatora (komunikat *ZATWIERDŹ* nie dotarł po prostu do *P*). Podobnie jeżeli *Q* jest w stanie *ZANIECHANIE*, to *P* również może bez obaw wycofać swoją lokalną transakcję.

Sytuacja jest trudniejsza, gdy proces *Q* jest w stanie ROZPOCZĘCIE. Oznacza to, że *Q* nie dostał od koordynatora komunikatu *GŁOSOWANIE*. W takiej sytuacji bezpieczniej jest przejść w obu procesach do stanu ZANIECHANIE.

Najtrudniejsza sytuacja jest wtedy, gdy *Q* jest w stanie GOTOWOŚĆ, również czekając na decyzję od koordynatora. Mogło się zdarzyć, że koordynator uległ awarii po zainicjowaniu głosowania i wszystkie procesy są w stanie oczekiwania. Nawet jeżeli wszystkie są gotowe do zatwierdzenia, to i tak nie uda się podjąć żadnej decyzji – trzeba czekać na wznowienie pracy koordynatora.

Protokół 2PC z rekonstrukcją – koordynator

```
1.  zapisz w rejestrze GŁOSOWANIE
2.  roześlij do wszystkich uczestników komunikat GŁOSOWANIE
3.  dopóki nie zebrano wszystkich głosów
4.      czekaj na nadejście głosu
5.      jeśli upłynął czas
6.          zapisz w rejestrze ZANIECHAJ
7.          roześlij ZANIECHAJ
8.          wyjdź
9.      zapisz głos
10. jeśli wszyscy odpowiedzieli na tak
11.     zapisz w rejestrze ZATWIERDŹ
12.     roześlij ZATWIERDŹ
13. w przeciwnym wypadku
14.     zapisz w rejestrze ZANIECHAJ
15.     roześlij ZANIECHAJ
```


Slajd przedstawia działania protokołu 2PC zakładającego możliwość rekonstrukcji (odtwarzania stanu) procesów. Rekonstrukcja procesu wymaga zapisywania jego stanu do pamięci trwałej (np. dysk), dlatego wszystkie zmiany stanów w procesach są najpierw zapisywane do lokalnego rejestru, a następnie dopiero wykonywane.

Proces koordynatora rozsyła do wszystkich komunikat *GŁOSOWANIE* i oczekuje na odpowiedzi. Brak odpowiedzi (przekroczenie czasu oczekiwania) powoduje, że zapada decyzja o wycofaniu transakcji (linia 6). Jeżeli wszyscy odpowiedzą pozytywnie, transakcja jest zatwierdzana. W przeciwnym wypadku następuje jej wycofanie poprzez rozesłanie komunikatu *ZANIECHAJ*.

Protokół 2PC z rekonstrukcją – uczestnik

1. zapisz w rejestrze *ROZPOCZĘCIE*
2. czekaj na komunikat *GŁOSOWANIE*
3. jeśli upłynął czas
4. zapisz w rejestrze *PRZECIW* i wyjdź
5. jeśli głosujesz za zatwierdzaniem
6. zapisz w rejestrze *ZA*
7. wyślij koordynatorowi komunikat *ZA*
8. czekaj na komunikat z *DECYZJĄ* od koordynatora
9. jeśli upłynął czas
10. roześlij do innych uczestników *PYTANIE_O_DECYZJĘ*
11. czekaj na odebranie *DECYZJI*
12. zapisz *DECYZJĘ* w rejestrze
13. w przeciwny wypadku (głos przeciw)
14. zapisz w rejestrze *PRZECIW*
15. wyślij koordynatorowi *PRZECIW*

Jeżeli proces uczestnika w momencie zajścia awarii był w stanie ROZPOCZĘCIE i nie podjął jeszcze decyzji o zatwierdzeniu transakcji, to może ją po prostu wycofać po odtworzeniu stanu. Jeżeli zdążył przejść do stanu ZATWIERDZENIE lub ZANIECHANIE, to powinien zostać odtworzony właśnie do tego stanu. Problematyczne staje się załamanie w stanie GOTOWOŚCI. W tym przypadku proces nie jest w stanie sam zadecydować co powinien zrobić dalej. Musi skontaktować się z innymi uczestnikami, w celu zadecydowania o losie lokalnej transakcji. Sytuacja jest analogiczna do braku odpowiedzi od koordynatora po zainicjowaniu głosowania (linia 9).

Decyzja o zatwierdzeniu transakcji w normalnych warunkach przychodzi od koordynatora. Jeżeli jednak on nie odpowiada, można próbować uzyskać informacje o decyzji od innych uczestników – następuje wysłanie komunikatu do wszystkich uczestników w linii 10. Decyzja, która nadejdzie jest zapisywana i ostatecznie egzekwowana.

Jeżeli proces głosuje na nie w głosowaniu, lub zapada decyzja o odrzuceniu transakcji (komunikat *ZANIECHAJ* bezpośrednio od koordynatora lub pośrednio od innych uczestników), transakcja jest wycofywana.

Obsługa pytania o decyzję

Przetwarzanie realizuje oddzielny wątek po stronie klienta

```
1.  dopóki prawda
2.      czekaj na odebranie PYTANIA_O_DECYZJĘ
3.      czytaj ostatni STAN zapisany w rejestrze
4.      jeśli STAN==ZATWIERDZENIE
5.          wyślij ZATWIERDZENIE do pytającego
6.      w przeciwnym razie
7.          jeśli STAN==ROZPOCZĘCIE lub STAN==ZANIECHAJ
8.              wyślij ZANIECHAJ
9.          w przeciwnym razie
10.             pomiń
```

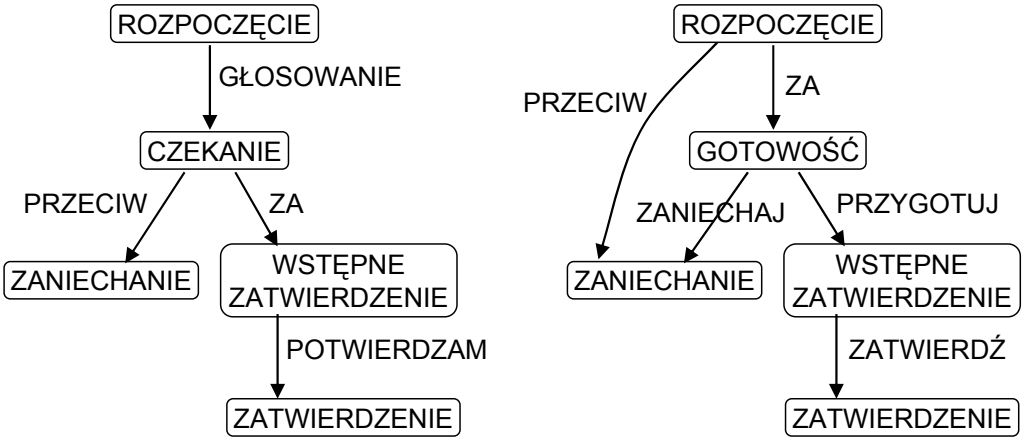

Każdy z procesów musi być gotowy na udzielenie odpowiedzi innym uczestnikom o decyzji w sprawie zatwierdzenia transakcji. Działanie takie może być realizowane przez oddzielny wątek, którego zadaniem jest tylko udzielanie odpowiedzi na komunikat *PYTANIE_O_DECYZJĘ*. Odpowiedź jest udzielana tylko wtedy, gdy jest znana, a więc gdy ostatni stan zapisany w rejestrze jest stanem końcowym *ZATWIERDZENIE* lub *ZANIECHANIE*. Odpowiedź negatywna jest udzielana również wtedy, gdy proces w momencie odbioru pytania o stan nadal pozostaje w stanie *ROZPOCZĘCIE*, co oznacza, że nie dostał jeszcze komunikatu *GŁOSOWANIE* od koordynatora, a to może oznaczać awarię koordynatora. Jeżeli odpytywany proces pozostaje nadal w stanie *GOTOWOŚĆI,* to ignoruje zapytanie i nie odpowiada, ponieważ sam nie zna jeszcze decyzji o losach transakcji. W skrajnym przypadku może się zdarzyć, że wszyscy uczestnicy nie dostaną komunikatu z decyzją od koordynatora, ponieważ ten ulegnie awarii. Wszyscy będą więc w stanie *GOTOWOŚCI*, a więc nie będą odpowiadać na pytania o decyzję. W efekcie nastąpi zablokowanie wszystkich uczestników do czasu rekonstrukcji procesu koordynatora, który po wznowieniu pracy podejmie decyzję. Z powodu tego blokowania protokół 2PC bywa również nazywany **blokującym protokołem zatwierdzania** (ang. *blocking commit protocol*).

Zatwierdzanie trzyfazowe

3. Koordynator zbiera wyniki. Jeżeli jakiś proces głosuje przeciw, koordynator wysyła komunikat *ZANIECHAJ*. Jeżeli wszyscy zagłosują za, koordynator wyśle komunikat *PRZYGOTUJ*.
4. Głosujący za zatwierdzeniem, jeżeli otrzymają komunikat *PRZYGOTUJ* wysyłają komunikat *POTWIERDZAM*
5. Koordynator po odebraniu od wszystkich komunikatu *POTWIERDZAM* wysyła *ZATWIERDŹ*
6. Po odebraniu komunikatu *ZATWIERDŹ* procesy zatwierdzają ostatecznie transakcję

Podstawową wadą zatwierdzania dwufazowego (2PC) jest niemożliwość dojścia do decyzji w przypadku awarii koordynatora. Uczestnicy przetwarzania muszą czekać blokująco na wznowienie pracy przez niego. Problem ten został rozwiązany w **protokole zatwierdzania trzyfazowego** (ang. *three-phase commit protocol* – 3PC). Protokół 3PC wprowadza dodatkową fazę przetwarzania, pomiędzy fazę głosowania i ostatecznego zatwierdzania. W fazie tej następuje częściowe, wstępne zatwierdzenie transakcji, które dopiero w następnej fazie powoduje ostateczne zatwierdzenie.

Stany w protokole 3PC



Slajd przedstawia maszyny stanowe dla protokołu 3PC. Istotą protokołu 3PC jest to, że:

1. Zarówno w przypadku koordynatora jak i uczestnika nie istnieje taki stan, z którego można by przejść bezpośrednio do stanu ZATWIERDZENIA albo ZANIECHANIA (np. stan CZEKANIE umożliwia przejście do ZANIECHANIE albo WSTĘPNE_ZATWIERDZENIE).
2. Nie istnieje stan, w którym nie można by podjąć ostatecznej decyzji i z którego można by przejść do stanu ZATWIERDZENIA.

Warunki te, jak wykazano, są konieczne i dostateczne do tego, żeby uzyskać protokół nieblokujący.

Jeżeli koordynator zablokuje się w stanie CZEKANIE, to zadecyduje o wycofaniu transakcji, ponieważ jeden z procesów uległ awarii. Jeżeli blokowanie nastąpi w stanie WSTĘPNE_ZATWIERDZENIE, to przekroczeniu czasu oczekiwania nastąpi zatwierdzenie transakcji, bo wiadomo, że proces, który uległ awarii głosował za zatwierdzeniem. Uczestnik, który uległ awarii po rekonstrukcji będzie mógł transakcję zatwierdzić.

Uczestnik może zablokować się w stanie GOTOWOŚCI lub WSTĘPNEGO_ZATWIERDZENIA. Po przekroczeniu czasu oczekiwania na odpowiedź od koordynatora przystąpi do próby uzyskania decyzji od innych uczestników. Jeżeli inny uczestnik jest w stanie ZATWIERDZENIA lub ZANIECHANIA, to stan ten powinien zostać przejęty przez zablokowanego uczestnika. Jeśli wszyscy uczestnicy są w stanie ZATWIERDZENIA_WSTĘPNEGO, to transakcja może być zatwierdzona. Uczestnik zablokowany w stanie ROZPOCZĘCIE może transakcję wycofać. Jeżeli jakiś uczestnik nadal jest w stanie ROZPOCZĘCIA, to znaczy, że żaden inny uczestnik nie jest w stanie WSTĘPNEGO_ZATWIERDZENIA.

Jeżeli uczestnik jest zablokowany w stanie GOTOWOŚCI i również większość innych uczestników jest w stanie GOTOWOŚCI, to transakcja może być zaniechana. Inny uczestnik, który mógł w tym czasie ulec awarii, nawet jeżeli dotarł do stanu WSTĘPNEGO_ZATWIERDZENIA może po rekonstrukcji wycofać swoją transakcję. W przypadku protokołu 2PC proces mógł po rekonstrukcji wrócić do stanu ZATWIERDZENIA, podczas gdy inni byli nadal w stanie GOTOWOŚCI. Do czasu zrekonstruowania procesu, pozostali uczestnicy protokołu 2PC nie mogli podjąć decyzji. W przypadku protokołu 3PC wiadomo, że rekonstrukcja będzie kończyła się stanem ROZPOCZĘCIE, ZANIECHANIE lub WSTĘPNE_ZATWIERDZENIE, dlatego pozostałe poprawne procesy mogą uzgodnić ostateczną decyzję.

Jeżeli uczestnik jest zablokowany w stanie WSTĘPNEGO_ZATWIERDZENIA, a większość pozostałych uczestników jest również w stanie WSTĘPNEGO_ZATWIERDZENIA, to transakcję można bezpiecznie zatwierdzić. Pozostałe procesy, które uległy awarii będą po rekonstrukcji w stanie GOTOWOŚCI, WSTĘPNEGO_ZATWIERDZENIA lub ZATWIERDZENIA.

Odtwarzanie (rekonstrukcja)

Odtwarzanie wsteczne (ang. *backward recovery*) — wycofanie systemu z aktualnego, błędnego stanu do wcześniejszego, poprawnego stanu

- okresowe zapisy stanu systemu (punkty kontrolne)
- przykład: retransmisja pakietu

Odtwarzanie postępowe (ang. *forward recovery*) — przeprowadzenie systemu z aktualnego, błędnego stanu do nowego, poprawnego stanu

- wcześniejsza znajomość możliwych błędów
- przykład: nadmiarowe kodowanie danych

Odtwarzanie (rekonstrukcja) ma na celu sprowadzenie procesu czy części systemu do stanu sprzed awarii. Generalnie istnieją dwie metody odtwarzania stanu: odtwarzanie wsteczne i odtwarzanie postępowe.

Odtwarzanie wsteczne (ang. *backward recovery*) polega na wycofaniu systemu z aktualnego, błędnego stanu do stanu wcześniejszego, który był poprawny. Wykonanie takiego odtwarzania wymaga okresowego zachowywania stanu systemu i w przypadku awarii jego odtworzenie. Stan jest utrwalany w postaci tzw. **punktu kontrolnego** (ang. *checkpoint*).

Odtwarzanie postępowe (ang. *forward recovery*) polega na przeprowadzeniu systemu z aktualnego, błędnego stanu do nowego, poprawnego stanu, od którego system będzie mógł kontynuować swoją pracę. Główną trudnością odtwarzania postępowego jest konieczność znajomości błędów jakie mogą wystąpić. Umożliwia to zaimplementowanie odpowiednich procedur korygujących te błędy.

Przykładem odtwarzania wstecznego jest retransmisja pakietu. Następuje w tym przypadku niejako wycofanie do sytuacji sprzed nadania komunikatu i powtórzenie tej operacji.

Przykładem odtwarzania postępowego jest kodowanie korekcyjne w protokołach transmisyjnych. Protokoły takie kodują dane nadmiarowo, dzięki czemu utraconą część danych (pojedynczy blok) można zrekonstruować z danych przesyłanych w otoczeniu felernego fragmentu.

Problemy odtwarzania

- Odtwarzanie jest kosztowne (efektywność)
- Możliwość nawrotu awarii po odtworzeniu
- Niektóre operacje są nieodwracalne
- Połączenie punktów kontrolnych i rejestrowania komunikatów (ang. *message logging*)
  - rejestrowanie u nadawcy (ang. *sender-based logging*)
  - rejestrowanie u odbiorcy (ang. *receiver-based logging*)
  - odtworzenie i powtórne dostarczenie komunikatów
  - determinizm przetwarzania

Odtworzenie poprzedniego stanu po błędzie jest operacją bardzo kosztowną, podobnie jak i samo okresowe rejestrowanie stanu przetwarzania. Co gorsze: poniesienie tych kosztów nie zawsze gwarantuje sukces, ponieważ ten sam błąd może wystąpić ponownie. Całkowite maskowanie pewnych błędów nie będzie więc możliwe, gdyż groziłoby zapętleniem operacji odtwarzania stanu.

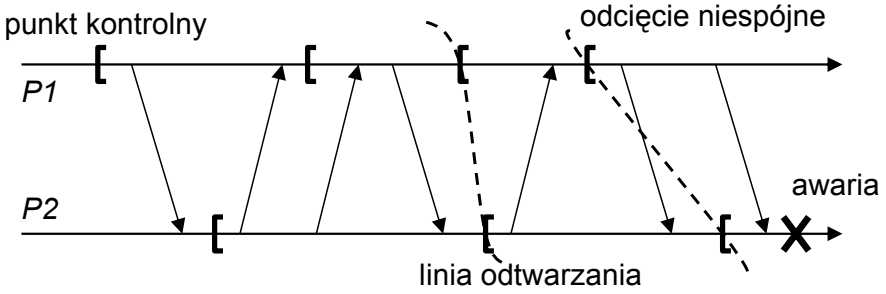
Innym problemem wstecznego odtwarzania jest niemożliwość wycofania pewnych operacji, szczególnie jeśli dotyczą one świata zewnętrznego.

Tworzenie punktów kontrolnych również jest dużym obciążeniem dla systemu. W przeciwieństwie do samej operacji odtwarzania, która wykonywana jest w momencie zajścia awarii, tworzenie punktów kontrolnych musi być realizowane nieustannie i to możliwie często. Z tego względu chętnie łączy się metodę odtwarzania wstecznego z rejestrowaniem komunikatów (ang. *message logging*). Rejestrowanie może się odbywać po stronie nadawcy (ang. *sender-based logging*) lub po stronie odbiorcy (ang. *receiver-based logging*). W przypadku wystąpienia załamania procesu można wycofać go do ostatniego punktu kontrolnego i dostarczyć aplikacji wiadomości, które były do niej dostarczone od momentu utworzenia tego punktu kontrolnego. W ten sposób można dokonać odtworzenia pojedynczego procesu bez konieczności wycofywania przetwarzania innych procesów systemu rozproszonego.

Rejestrowanie komunikatów ma dodatkową zaletę w postaci zachowania determinizmu przetwarzania. W systemie rozproszonym kolejność dostarczania komunikatów może być za każdym razem inna, co może powodować, że powtórne wykonanie tego samego programu da inny efekt. Wycofanie procesu i dostarczenie mu tych samych komunikatów pozwala na zachowanie spójności przetwarzania względem innych procesów.

Punkty kontrolne

- Węzły okresowo zapamiętują swój stan w lokalnej pamięci trwałej
- Odbudowa spójnego stanu globalnego
- **Linia odtwarzania** – najnowsze odcięcie spójne

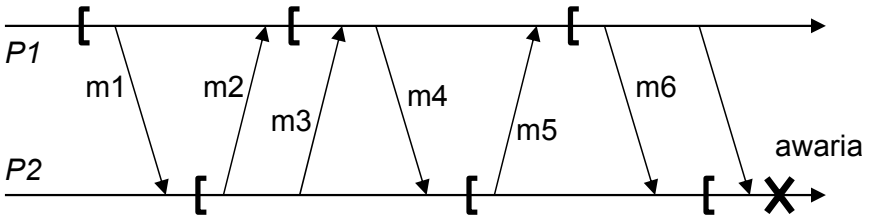


Wykonanie wstecznego odtwarzania jest możliwe pod warunkiem posiadania dostępu do punktów kontrolnych. System okresowo tworzy więc punkty kontrolne w poszczególnych węzłach i zapisuje je do lokalnej pamięci trwałej (ang. *stable storage*). Odtworzenie stanu po awarii wymaga skonstruowania spójnego stanu globalnego na podstawie stanów lokalnych (punktów kontrolnych). Spójność oznacza tu oczywiście, że fakt zarejestrowania odbioru wiadomości *m* powinien pociągać za sobą zarejestrowanie faktu wysłania wiadomości *m*. Dokonując odtwarzania systemu najlepiej posłużyć się najwcześniejszymi punktami kontrolnymi, takimi które będą dawały w sumie spójny obraz. Takie odcięcie nazywamy **linią odtwarzania** (ang. *recovery line*). Linia odtwarzania jest więc najnowszym odcięciem spójnym.

Na rysunku odcięcie stworzone przez ostatnie punkty kontrolne nie jest spójne, a więc nie może być wykorzystane do odtworzenia stanu. Wcześniejsza para punktów kontrolnych daje już spójne odcięcie i tworzy linię odtwarzania.

Niezależne punkty kontrolne

- Konieczność znalezienia linii rekonstrukcji
- Ryzyko wystąpienia **efektu domina** (ang. *domino effect*) – kaskadowe wycofywanie
- Konieczność okresowego odśmiecania pamięci lokalnej



Punkty kontrolne można zapisywać w sposób skoordynowany lub nie. **Niezależne rejestrowanie punktów kontrolnych** (ang. *independent checkpointing*) jest wygodne, ponieważ punkty kontrolne mogą być rejestrowane bez konieczności synchronizowania ze sobą rozproszonych procesów. Z drugiej jednak strony wymusza poszukiwanie linii rekonstrukcji w momencie zaistnienia awarii. Poszukiwanie takiego spójnego odcięcia wymaga rejestrowania zależności pomiędzy poszczególnymi punktami kontrolnymi w celu wykrycia niespójności. Samo znalezienie linii rekonstrukcji również nie jest trywialne. Co gorsze: poszukiwanie odcięcia spójnego może prowadzić do odrzucania kolejnych punktów kontrolnych. Sytuacja taka jest przedstawiona na rysunku. Pary punktów kontrolnych C13-C23 nie można wykorzystać, ponieważ punkt kontrolny C23 obejmuje zdarzenie odbioru wiadomości m6, a punkt kontrolny C13 nie obejmuje wysłania wiadomości m6. Cofając się w czasie, para C13-C22 również nie spełnia wymagań, ponieważ C13 obejmuje odbiór m5 a C22 nie obejmuje nadania m5. Podobnie pary C22-C12, C12-C21 i C21-C11. Ostatecznie więc stan początkowy przetwarzania pozwala uzyskać odcięcie spójne. Takie kaskadowe wycofywanie się nazywamy **efektem domina** (ang. *domino effect*). Okazuje się jednak, że podstawowym problemem odtwarzania niezależnego jest koszt przechowywania punktów kontrolnych. Każdy punkt kontrolny zajmuje dużą ilość pamięci a przechowywać trzeba wiele takich punktów. Pamięć taką można oczyszczać ze starych punktów kontrolnych, ale jest to znów operacja kosztowna, wymagająca zastosowania rozproszonego algorytmu odśmiecania.

Koordynowane punkty kontrolne

- Synchronizacja wszystkich procesów
- Automatyczne zapamiętanie stanu spójnego
- Algorytm rejestrowania stanu globalnego systemu
- Zastosowanie protokołu blokowania dwufazowego 2PC
  - faza I: zamówienie punktu kontrolnego i wstrzymanie komunikacji
  - faza II: odblokowanie komunikacji
- Ulepszone 2PC
  - wysyłanie zamówienia tylko do procesów zależnych od koordynatora (→ **migawka przyrostowa**)

Koordynowane rejestrowanie punktów kontrolnych (ang. *coordinated checkpointing*) polega na zsynchronizowaniu grupy procesów w celu zapamiętania swojego stanu w pamięci trwałej. Poniesienie kosztu synchronizacji jest opłacalne, ponieważ automatycznie tak wygenerowany zbiór punktów kontrolnych tworzy linię odtwarzania. Nie ma więc potrzeby przechowywania wielu punktów kontrolnych w węzłach, ponieważ zbiór ostatnich punktów kontrolnych automatycznie tworzy odcięcie spójne. Do skoordynowania procesów można wykorzystać algorytm rejestrowania stanu globalnego systemu. Innym, prostszym rozwiązaniem jest zastosowanie protokołu blokowania dwufazowego (2PC). W pierwszej fazie wykonywane jest zarejestrowanie punktów kontrolnych w poszczególnych węzłach po odebraniu zlecenia od koordynatora. Jednocześnie następuje wstrzymanie wysyłania nowych komunikatów. W drugiej fazie, po zebraniu potwierdzeń od wszystkich węzłów, koordynator wysyła zezwolenie na kontynuację przetwarzania. Algorytm pozwala na zarejestrowanie globalnie spójnego stanu, ponieważ po zarejestrowaniu stanu na węźle, węzeł ten przestaje nadawać nowe komunikaty do czasu zarejestrowania stanu w pozostałych węzłach. Eliminowane są więc automatycznie wiadomości, które potencjalnie mogłyby naruszyć spójność obrazu.

Podejście to można zoptymalizować poprzez rozsyłanie zlecenia na wykonanie punktu kontrolnego tylko do węzłów, które są *zależne* od koordynatora. Węzeł jest zależny od koordynatora, gdy odebrał wiadomość przyczynowo zależną od dowolnej wiadomości wysłanej przez koordynatora od czasu wykonania ostatniego punktu kontrolnego. Wykonanie **migawki przyrostowej** (ang. *incremental snaphot*) realizowane jest poprzez wysłanie przez koordynatora zlecenia wykonania punktu kontrolnego do wszystkich procesów, do których wysyłał wiadomość od ostatniego punktu kontrolnego. Każdy proces, który dostaje takie zlecenie propaguje je jednokrotnie do wszystkich procesów, do których sam wysyłał wiadomość. Po zidentyfikowaniu grupy procesów zależnych od koordynatora wykonywane jest zarejestrowanie punktów kontrolnych w tych procesach podobnie jak to miało miejsce w przypadku standardowego 2PC.