Niezawodność przetwarzania

Plan wykładu

Celem wykładu jest wprowadzenie do tematyki przetwarzania rozproszonego w środowisku zawodnym. Obejmie on omówienie podstawowych definicji związanych z niezawodnością, w tym pojęcia awarii, błędu, wady oraz klasy awarii. Następnie wprowadzone zostaną precyzyjne definicje mechanizmów (abstrakcji) kanałów komunikacyjnych o różnych właściwościach. W kontekście niezawodności scharakteryzowane zostaną też podstawowe modele systemów rozproszonych: model asynchroniczny, synchroniczny i częściowo synchroniczny. W kolejnym punkcie zaprezentowany będzie mechanizm detektora awarii. Przedstawione zostaną podstawowe klasy detektorów i relacje pomiędzy poszczególnymi klasami, wybrane metryki służące do oceny detektorów awarii, oraz implementacje detektorów awarii klas P, ◊P. Na zakończenie omówiony będzie problem elekcji w środowisku zawodnym i jego związek z detektorami awarii.

Proces poprawny i niepoprawny

Proces będziemy nazywać **poprawnym**, jeżeli działa zgodnie z algorytmem (specyfikacją). Przyjmiemy przy tym, że w przypadku takiego procesu zachodzi predykat $correct(P_i)$. W przeciwnym razie, tzn. gdy proces nie działa zgodnie z algorytmem (a więc również z oczekiwaniami) mówimy, że proces jest **niepoprawny** (jego zachowanie odbiega od oczekiwanego). W szczególności niepoprawny proces może nie działać w ogóle. Niepoprawne działanie procesu oznaczać też będzie, że $\neg correct(P_i)=True$.

Definicje błędu, awarii, wady

Pod pojęciem **awarii** (ang. *failure*) rozumieć będziemy działanie procesu niezgodne z algorytmem czy specyfikacją. **Błąd** (ang. e*rror*) będziemy traktować jako część stanu procesu odpowiedzialną za będącą jego następstwem awarię. Może on być symptomem awarii, która ujawnia się natychmiast lub dopiero w przyszłości. Natomiast jako **wadę** (ang. f*ault*) będziemy traktować stwierdzoną lub hipotetyczną przyczynę wystąpienia błędu.

Klasy awarii procesów

Wyróżnia się cztery podstawowe klasy awarii procesów: załamania, zaniechania, awarie powtarzalne i awarie dowolne.

- **Załamanie** (ang. *crashes*) oznacza, że proces działa poprawnie do pewnej chwili t, po której zaprzestaje wysyłać i odbierać wiadomości, pomimo, że wynika to z algorytmu procesu. Specjalnym przypadkiem jest sytuacja, w której proces ulega awarii przed wykonaniem jakiegokolwiek kroku algorytmu (ang. *initially-dead*).
- **Zaniechanie** (ang. *ommissions*) oznacza, że proces nie wysyła (lub nie dostarcza) wiadomości, pomimo, że powinien to zrobić zgodnie z wykonywanym algorytmem. Awarie zaniechania mogą być konsekwencją przepełnienia bufora lub przeciążenia sieci komunikacyjnej.
- **Awaria powtarzalna** (ang. *crashes and recoveries*) oznacza, że proces może być odtworzony (wznowiony) po awarii. Wystąpienie awarii tego typu wstrzymuje proces na pewien czas, po upływie którego proces rozpoczyna ponownie wykonywanie algorytmu. Proces jest uznawany za poprawny, jeżeli istnieje pewna chwila t, po której proces wykonuje algorytm i nigdy nie podlega awariom. Innymi słowy, proces poprawny to taki, który ulega skończonej liczbie awarii, przy czym po każdej z nich możliwe jest odtworzenie jego stanu. Proces błędny w tym modelu to taki, który nie jest wznowiony po kolejnej (w szczególności, pierwszej) awarii, lub taki, który nieskończenie wiele razy ulega awarii.
- **Awaria dowolna** (ang. *arbitrary*) oznacza, że zachowanie procesu odbiega w sposób dowolny od algorytmu, który powinien wykonywać. Na przykład, proces może wysyłać w dowolnej chwili różne wiadomości. Szczególnym przypadkiem awarii dowolnych są awarie bizantyńskie.

Jak łatwo zauważyć, najbardziej ogólną klasą awarii jest klasa z awariami dowolnymi. Zaniechania są specjalnym przypadkiem awarii powtarzalnych, natomiast załamania szczególnym przypadkiem zaniechań.

W synchronicznych systemach rozproszonych pojawia się dodatkowa klasa awarii związana z wykonywaniem kroków we właściwej kolejności, ale w niewłaściwym czasie. Są to awarie dotyczące czasu (ang. *timing errors*).

Modele kanałów komunikacyjnych

Przypomnijmy, że kanał jest obiektem (abstrakcją, mechanizmem) skojarzoną z uporządkowaną parą procesów $<P_i, P_j>$, modelującym jednokierunkowe łącza transmisyjne. Dotychczas zakładaliśmy, że kanały są niezawodne. Oznaczało to, że każda wiadomość wysłana kanałem ostatecznie (w końcu) docierała do adresata. Takie założenie nie zawsze jest realizowalne. Przeanalizujemy więc dalej różne modele kanałów uwzględniając możliwe ich awarie. W szczególności omówimy kanały rzetelne (ang. *fair-loss links*), wytrwałe (ang. *stubborn links*) i niezawodne (ang. *perfect links*), które będą podstawą konstrukcji wybranych algorytmów rozproszonych w środowisku zawodnym.

Kanały rzetelne

Kanał między dwoma procesami nazywać będziemy **kanałem rzetelnym** (ang. *fair loss point-to-point link*) jeżeli posiada on własności rzetelnego dostarczania, ograniczonego powielania i braku samogeneracji:

- Własność **rzetelnego dostarczania** (ang. *fair loss delivery*) oznacza. że jeżeli wiadomość M wysyłana jest nieskończoną liczbę razy przez proces P_i do procesu P_j i żaden z tych procesów nie ulega awarii (oba są poprawne), to wiadomość M jest dostarczona nieskończoną liczbę razy do P_j. Należy zauważyć, że własność ta dopuszcza zagubienia wiadomości M, nawet nieskończoną liczbę razy.
- Własność **ograniczonego powielania** (ang. *finite duplication*) oznacza, że jeżeli wiadomość M wysyłana jest skończoną liczbę razy przez proces P_i do procesu P_j, to wiadomość ta nie może być dostarczona nieskończoną liczbę razy do procesu P_j.
- Własność **braku samogeneracji** (ang. *no creation*) oznacza, że jeżeli wiadomość M została dostarczona do procesu P_j, to została ona wcześniej wysłana do tego procesu przez jakiś inny proces P_i. Innymi słowy, kanał nie tworzy samorzutnie wiadomości.

Kanały rzetelne: Operacje komunikacyjne

W dalszej części wykładu, podczas prezentacji algorytmów, by podkreślić użycie kanałów rzetelnych, będziemy używać zapisu $\text{send}^{\text{FL}}(P_i, P_j, M)$ by oznaczyć operację wysłania wiadomości M przez P_i do procesu P_j, a odpowiadające jej zdarzenia zapisywać będziemy jako $e\_sendFL\ (P_i, P_j, M)$. Analogicznie, przez $\text{deliver}^{\text{FL}}(P_j, P_i, M)$ oznaczać będziemy operację uaktywniającą zdarzenie $e\_receive^{FL}(P_j, P_i, M)$ i przekazującą wiadomość M (wysłaną przez proces P_j) do procesu aplikacyjnego P_i lub innego mechanizmu komunikacyjnego (innej warstwy komunikacyjnej).

Kanały wytrwałe

Kanał między dwoma procesami nazywać będziemy **kanałem wytrwałym** (upartym) (ang. *stubborn point-to-point link*), jeżeli posiada on własności wytrwałego dostarczania i braku samogeneracji.

- Własność **wytrwałego dostarczania** (ang. *stubborn delivery*) oznacza, że jeżeli wiadomość M wysyłana jest przez proces P_i do procesu P_j i oba te procesy są poprawne (nie ulegają awarii), to wiadomość M jest dostarczona nieskończoną liczbę razy do P_j.
- Własność **braku samogeneracji** (ang. *no creation*) oznacza, że jeżeli wiadomość została dostarczona do procesu P_j, to została ona wcześniej wysłana do tego procesu przez jakiś inny proces P_i. Innymi słowy, kanał nie tworzy samorzutnie wiadomości.

Kanały wytrwałe: Operacje komunikacyjne

W dalszej części wykładu, podczas prezentacji algorytmów, by podkreślić użycie kanałów wytrwałych, będziemy używać zapisu $\text{send}^{\text{SB}}(P_i, P_j, M)$ by oznaczyć operację wysłania wiadomości M przez P_i do

procesu P_j, a odpowiadające jej zdarzenia zapisywać będziemy jako $e\_send^{SB}(P_i, P_j, M)$. Analogicznie, przez $\text{deliver}^{SB}(P_j, P_i, M)$ oznaczać będziemy operację uaktywniającą zdarzenie $e\_receive^{SB}(P_j, P_i, M)$ i przekazującą wiadomość M (wysłaną przez proces P_j) do procesu aplikacyjnego P_i.

Algorytm implementujący kanały wytrwałe: Założenia

Przedstawiony algorytm zapewniający własności kanałów wytrwałych wykorzystuje mechanizm kanałów rzetelnych.

Kanały wytrwałe: Algorytm (1)

W używanym algorytmie wiadomości aplikacyjne są umieszczane w pakietach typu PACKET.

Pakiety *pcktOut* typu PACKET wysyłane przez monitory zawierają wiadomości aplikacyjne typu MESSAGE. Wiadomości *msgIn* z kolei to wiadomości aplikacyjne znajdujące się w odbieranych przez monitory pakietach. Wiadomości te są dalej przekazywane procesom aplikacyjnym z użyciem operacji $deliver^{SB}(P_i, P_j, M)$, uaktywniającej zdarzenie $e\_receive^{SB}(P_j, P_i, M)$ w odpowiednim procesie aplikacyjnym.

Kanały wytrwałe: Algorytm (2)

Po stronie nadawcy implementacja mechanizmu kanałów wytrwałych sprowadza się do wysyłania wiadomości nieskończoną liczbę razy. Po stronie odbiorcy wiadomości, każde odebrana wiadomość jest po prostu przekazywana procesowi aplikacyjnemu – adresatowi.

Kanały niezawodne

Kanał między dwoma procesami nazywany jest **kanałem niezawodnym** (ang. *perfect point-to-point link*, *reliable point-to-point link)* jeżeli posiada on własności niezawodnego dostarczania, braku powielania i braku samogeneracji.

- Własność **niezawodnego dostarczania** (ang. *reliable delivery*) oznacza, że jeżeli wiadomość M wysyłana jest przez proces P_i do procesu P_j i oba te procesy są poprawne, to wiadomość M jest ostatecznie dostarczona do P_j.
- Własność **braku powielania** (ang. no *duplication*) oznacza, że żadna wiadomość wysłana do procesu P_j nie może być dostarczona do procesu P_j więcej niż raz.
- Własność *braku* **samogeneracji** (ang. *no creation*) oznacza, że jeżeli wiadomość została dostarczona do procesu P_j, to została ona wcześniej wysłana do procesu P_j przez jakiś inny proces P_i. Innymi słowy, kanał nie tworzy samorzutnie wiadomości.

Czasami kanały tego typu nazywa się kanałami **ostatecznie niezawodnymi**.

Kanały niezawodne: Operacje komunikacyjne

W dalszej części wykładu, podczas prezentacji algorytmów, by podkreślić użycie kanałów niezawodnych, będziemy używać zapisu $\text{send}^{PL}(P_i, P_j, M)$ by oznaczyć operację wysłania wiadomości M przez P_i do procesu P_j, a odpowiadające jej zdarzenia zapisywać będziemy jako $e\_send^{PL}(P_i, P_j, M)$. Analogicznie, przez $\text{deliver}^{PL}(P_j, P_i, M)$ oznaczać będziemy operację uaktywniającą zdarzenie $e\_receive^{PL}(P_j, P_i, M)$ i przekazującą wiadomość M (wysłaną przez proces P_j) do procesu aplikacyjnego P_i.

Dla uproszczenia, jeżeli nie będzie to prowadzić do niejednoznaczności, w operacjach kanałów niezawodnych pomijać będziemy indeks górny. Tym samym, przyjmować będziemy, że:

$$\text{send}(P_i, P_j, M) = \text{send}^{PL}(P_i, P_j, M)$$

$$\text{receive}(P_j, P_i, M) = \text{receive}^{PL}(P_j, P_i, M)$$

Algorytm implementujący kanały niezawodne: Założenia

Przedstawiony algorytm zapewniający własności kanałów niezawodnych wykorzystuje mechanizm kanałów wytrwałych.

Kanały niezawodne: Algorytm (1)

Podobnie jak poprzednio, pakiety *pcktOut* typu PACKET wysyłane przez monitory zawierają wiadomości aplikacyjne typu MESSAGE wysyłane przez proces aplikacyjny. Wiadomości *msgIn* z kolei to wiadomości aplikacyjne znajdujące się w odbieranych przez monitory pakietach. Wiadomości te są dalej przekazywane procesom aplikacyjnym z użyciem operacji $deliver^{PL}(P_i, P_j, M)$, uaktywniającej zdarzenie $e\_receive^{PL}(P_j, P_i, M)$ w procesie aplikacyjnym P_i.

Kanały niezawodne: Algorytm (2)

Przedstawiona implementacja kanałów niezawodnych używa omówionej wcześniej abstrakcji kanałów wytrwałych (wiersz 2).

Jak widać, różnica między kanałami wytrwałymi i niezawodnymi polega tylko na sposobie dostarczania wiadomości. Wiadomość jest dostarczana przez monitor procesowi aplikacyjnemu tylko wtedy, gdy nie zawiera się on w zbiorze $delivered_i$. Po dostarczeniu procesowi wiadomość jest dodawana do zbioru $delivered_i$. Innymi słowy, wiadomość jest dostarczana tylko wtedy, gdy wcześniej nie była już dostarczona.

Modele systemów rozproszonych w kontekście niezawodności

Jak wcześniej wspomniano, ważnym aspektem charakteryzującym system rozproszony są przyjęte założenia dotyczące zachowania procesów i kanałów komunikacyjnych uwzględniające upływający czas. W zależności od tego czy istnieją w danym modelu ograniczenia dotyczące czasu komunikacji i szybkości względnej procesów możemy wyróżnić systemy asynchroniczne, częściowo synchroniczne i synchroniczne. Szczegóły dotyczące każdej z tych klas zostaną przedstawione na kolejnych slajdach.

System asynchroniczny

W modelu asynchronicznego systemu nie przyjęto żadnych dodatkowych założeń dotyczących czasu, w tym ograniczeń dotyczących czasu transmisji wiadomości i czasu przetwarzania. Ponadto procesy nie mają dostępu do wspólnego, globalnego zegara, np. fizycznego zegara czasu rzeczywistego.

Jak wiadomo, pomimo braku dostępu do zegara globalnego, nadal możliwy jest pomiar czasu. Będzie to jednak tylko czas logiczny związany z wymianą komunikatów. Implementacja odpowiednich zegarów logicznych była przedstawiona na poprzednich wykładach.

Dla wielu problemów konstrukcja algorytmów rozproszonych w zawodnych systemach asynchronicznych jest trudna, a często nawet niemożliwa bez dodatkowych założeń dotyczących synchronizmu lub dostępności specyficznych mechanizmów. Przykładem może być tutaj problem konsensusu, o którym będziemy mówić później.

System synchroniczny

Na drugim końcu klasyfikacji modeli w stosunku do modelu asynchronicznego, znajduje się model synchroniczny.

Systemy synchroniczne charakteryzują się synchronicznością przetwarzania i komunikacji oraz dostępnością zegara globalnego.

W przetwarzaniu synchronicznym istnieje i jest znane górne ograniczenie dotyczące czasu przetwarzania poszczególnych procesów, lub inaczej, względnych prędkości wykonywania procesów. Można zatem łatwo określić czas, po którym dane zadanie powinno zostać wykonane przez proces. Takimi zadaniami może być wysłanie lub odebranie wiadomości lub też przetwarzanie lokalne.

W komunikacji synchronicznej istnieje znane górne ograniczenia czasu transmisji wiadomości przez kanały komunikacyjne. Jest to innymi słowy czas potrzebny aby komunikat wysłany przez nadawcę dotarł od adresata w najgorszym przypadku.

Dostępność zegara globalnego oznacza możliwość lokalnego odczytu przez procesy aktualnego stanu wspólnego zegara globalnego. Odczyt ten jest precyzyjny bądź obarczony znanym maksymalnym błędem odczytu lokalnego w stosunku do wartości zegara globalnego.

Niestety, z uwagi na silne założenia dotyczące czasu, konstrukcja systemów synchronicznych jest bardzo trudna. Ponadto, w systemach tego typu trzeba zakładać czasy maksymalne, na ogół znacznie

większe niż czasy średnie. Takie założenie prowadzi zwykle w praktyce do istotnego zmniejszenie efektywności aplikacji.

Systemy częściowo synchroniczne (1)

Rzeczywiste systemy rozproszone *zwykle* zachowują się jak systemy synchroniczne, tzn. *zazwyczaj* znane są pewne ograniczenia czasowe na czas dostarczenia wiadomości. System tego typu można by więc nazwać systemem *zazwyczaj* synchronicznym.

Istnieją jednak odcinku czasu, w których te znane ograniczenia czasowe przestają obowiązywać i wtedy system taki zachowuje się jak system asynchroniczny, np. w chwili przeciążenia sieci komunikacyjnej, przepełnienia buforów itp.

Z powyższych powodów wprowadzono pojęcie systemów **częściowo synchronicznych** (ang. *partially synchronous*). Systemy takie można zdefiniować na kilka sposobów:

- Mogą to być systemy, w których po pewnym, nieznanym czasie, zaczynają obowiązywać przyjęte założenia na górne ograniczenia na czas dostarczenia wiadomości. Innymi słowy, systemy takie *ostatecznie* zaczynają się zachowywać jak systemy synchroniczne, lecz nie wiadomo, od jakiego momentu.
- Można także przyjąć, że w systemach częściowo synchronicznych istnieją odcinku czasu, w których istnieją ograniczenia na czasy dostarczenia wiadomości i odcinku te są wystarczająco długie, by umożliwić zakończenie algorytmów.

Zaprezentowane definicje nie znaczą oczywiście, że system musi być początkowo asynchroniczny, czy też, że po pewnym znanym czasie system taki stanie się systemem synchronicznym na zawsze.

Systemy częściowo synchroniczne (2)

Można także dokonać klasyfikacji systemów częściowo synchronicznych na podstawie parametrów dotyczących właściwości procesów i kanałów komunikacyjnych.

- Parametr pierwszy określa, czy procesy są synchroniczne, czy nie. Synchroniczność procesów oznacza, że istnieje stała k_0 taka, że jeżeli pewien proces P_i wykona k_0+1 kroków, to każdy inny proces wykona co najmniej 1 krok.
- Parametr drugi określa, czy istnieją ograniczenia na czas przesyłania wiadomości (opóźnienie komunikacyjne).
- Trzeci parametr określa porządek wiadomości. Porządek synchroniczny oznacza, że jeżeli proces P_i wysyła wiadomość M do procesu P_j w czasie τ_i oraz proces P_k wysyła wiadomość M' do procesu P_j w czasie $\tau_j > \tau_i$, to wiadomość M zostanie dostarczona do P_j przed wiadomością M'.
- Czwarty parametr określa, czy komunikacja odbywa się za pomocą niepodzielnej komunikacji rozgłaszania czy też za pomocą komunikacji punkt-punkt.
- Ostatni z parametrów, to parametr mówiący, czy operacje *send/receive* są operacjami atomowymi (tzn., czy operacja wysłania wiadomości M kończy się dopiero po zakończeniu odbioru tej wiadomości przez adresata), czy też są rozłączne. Innymi słowy, czy komunikacja między procesami jest synchroniczna, czy też nie.

Korzystając z tych pięciu parametrów można wyodrębnić 32 klasy systemów. Podział ten jednakże nie zdobył powszechnego uznania, głównie z uwagi na to, że w rzeczywistych systemach niektóre parametry są do pewnego stopnia równoważne innym. Na przykład, niepodzielne operacje rozgłaszania i receive/send wymagają istnienia ograniczeń na czas przesyłania wiadomości, by można je było zaimplementować.

Detektory awarii

Z praktycznego punktu widzenia, zawodny rozproszony detektor awarii (ang. *failure detector*) można definiować jako zbiór „wyroczni", które są powiązane z poszczególnymi procesami biorącymi udział w przetwarzaniu rozproszonym. Zadaniem tych wyroczni jest dostarczanie listy procesów podejrzewanych o niepoprawne działanie (czyli takich, które uległy awarii).

Wyrocznie te mogą być omylne, co oznacza, że mogą podejrzewać o awarię procesy poprawne, lub też odwrotnie – nie podejrzewać procesów niepoprawnych. Błędne podejrzenia są oczywiście niepożądane, ale w praktyce są akceptowalne, o ile nie powstrzymują poprawnych procesów przed zachowaniem się zgodnie z ich specyfikacją. Dlatego też mówi się, że detektor podejrzewa, a nie wykrywa proces niepoprawny.

Ponadto, w momencie stwierdzenia przez wyrocznię awarii procesu, proces ten powinien znaleźć się na liście procesów podejrzanych. Natomiast w przypadku odwrotnym, gdy wyrocznia stwierdzi, że proces jest poprawny, proces ten powinien być usunięty z listy procesów podejrzewanych.

Informacje dostarczane przez detektor są tym dokładniejsze, im większy synchronizm systemu rozproszonego. Właściwości detektorów możliwych do zaimplementowania w systemie określają stopień jego synchroniczności.

Model systemu

System składa się ze zbioru procesów, $\mathcal{P} = \{P_1, P_2, P_3, ..., P_n\}$. Kanały komunikacyjne łączące wybrane, stosownie do topologii przetwarzania, pary procesów są niezawodne. Zegar globalny jest dyskretny a jego dziedzina $\mathcal{Y}$ jest zbiorem liczb naturalnych $\mathbb{N}$.

Zakłada się też ponadto, że czasy komunikacji i przetwarzania są ograniczone lecz nieznane.

Wzorzec awarii

Chandra i Toueg wprowadzili pojęcie zawodnego detektora awarii, jako dostępnego dla programisty mechanizmu, który udziela informacji na temat stanu poszczególnych procesów. Niezwykle ważne jest tu określenie „zawodny", oznaczające tu, że nie spodziewamy się, iż detektor w każdej chwili udzieli odpowiedzi prawidłowej. Może on wskazywać procesy poprawne jako wadliwe (uległe awarii, błędne, niepoprawne), bądź na odwrót, przy czym może popełniać pomyłki nieskończenie wiele razy. Nie może to jednak spowodować naruszenia warunków poprawności bądź zakończenia algorytmu.

Niech **wzorzec awarii** $\boldsymbol{F}$ będzie funkcją $F: \mathcal{Y} \rightarrow 2^{\mathcal{P}}$, gdzie $F(\tau)$ oznacza zbiór procesów, które uległy awarii do momentu τ. Po awarii proces nie powraca do pracy, czyli prawdziwe jest wyrażenie

$$\forall \tau : F(\tau) \subseteq F(\tau+1) \qquad (11.9).$$

Wzorzec awarii opisuje więc faktyczny stan wszystkich procesów w różnych odcinkach czasu.

Ponadto zbiór procesów podejrzanych oraz poprawnych jest opisany odpowiednio przez równości

$$crashed(F) = \bigcup \tau \in \mathcal{T}\ F(\tau) \qquad (11.10)$$

i

$$correct(F) = \mathcal{P} \setminus crashed(F) \qquad (11.11).$$

Jeżeli P należy do zbioru $crashed(F)$, to mówimy że P uległ awarii zgodnie z F, a jeżeli proces P należy do zbioru $correct(F)$, mówimy, że P jest poprawny zgodnie z F.

Historia detektora awarii

Rozważamy tylko wzorce awarii F, w których co najmniej jeden proces jest poprawny, czyli $correct(F) \neq \varnothing$. Każdy detektor utrzymuje listę (zbiór) procesów, które podejrzewa.

Historia detektora awarii to funkcja odwzorowująca iloczyn kartezjański zbioru procesów i pewnego okresu czasu w rodzinę wszystkich podzbiorów zbioru $\mathcal{P}$. Wartość $H^{FD}(P_i, \tau)$ oznacza zbiór procesów podejrzewanych przez P_i w chwili τ.

Formalna definicja detektora awarii

Zbiór możliwych historii $\boldsymbol{\mathcal{D}(F)}$ oznacza sumę mnogościową wszystkich możliwych historii detektora awarii, które mogły się przydarzyć z wzorcem awarii F i detektorem awarii FD.

Detektor awarii FD jest to funkcja odwzorowująca wzorzec awarii F w zbiór możliwych historii detektora awarii $\mathcal{D}(F)$.

Własności detektorów

Detektory awarii charakteryzowane są przez dwie własności:

- kompletność (ang. *completeness*)
- dokładność (ang. *accuracy*)

Kompletność detektora związana jest z możliwością popełniania pomyłek pierwszego rodzaju – tzn., nie podejrzewania procesów niepoprawnych.

Dokładność detektora związana jest natomiast z możliwością pomyłki drugiego rodzaju – tj., podejrzewania przez detektor procesów, które są poprawne.

Zauważmy, że branie pod uwagę tylko jednej z tych własności nie jest wystarczające. W przypadku kompletności trywialny w implementacji detektor podejrzewający zawsze wszystkie procesy spełniałby zawsze własność silnej kompletności – podejrzewając wszystkie procesy, gwarantowałby, że podejrzewa również wszystkie procesy niepoprawne. Analogicznie trywialny przykład detektora nie podejrzewającego żadnego procesu można by przedstawić dla drugiej z własności. Na tej podstawie jasno widać konieczność uwzględniania obu własności.

Własność kompletności

Silna kompletność (ang. *strong completeness*) oznacza, że ostatecznie (w końcu) każdy proces, który jest niepoprawny, będzie trwale podejrzewany przez każdy poprawny proces.

Słaba kompletność (ang. weak completeness) oznacza, że ostatecznie (w końcu) każdy proces, który jest niepoprawny, będzie trwale podejrzewany przez niektóre poprawne procesy.

Własność dokładności

Silna dokładność (ang. *strong accuracy*) oznacza, że żaden proces nie jest podejrzewany, dopóki nie stanie się niepoprawny.

Słaba dokładność (ang. *weak accuracy*) oznacza, że niektóre poprawne procesy nie są nigdy podejrzewane.

Można zdefiniować dwie dodatkowe własności związane z dokładnością mówiące, że detektor awarii zacznie działać jak detektor gwarantujący słabą lub silną dokładność, ale dopiero po pewnym nieznanym ale ograniczonym czasie. Definicje tych własności są następujące:

Ostatecznie silna dokładność (ang. *eventual strong accuracy*) oznacza, że po upływie pewnego skończonego czasu żadne poprawne procesy nie są podejrzewane przez żadne poprawne procesy.

Ostatecznie słaba dokładność (ang. *eventual weak accuracy*) oznacza, że po upływie pewnego skończonego czasu niektóre poprawne procesy nie są podejrzewane przez żaden poprawny proces.

Klasyfikacja detektorów

		Dokładność			
		Silna	**Słaba**	**Ostatecznie silna**	**Ostatecznie słaba**
Kompletność	**Silna**	**P**	**S**	$\Diamond$**P**	$\Diamond$**S**
	Słaba	**Q**	**W**	$\Diamond$**Q**	$\Diamond$**W**

Kombinacja omówionych własności daje nam w sumie 8 klas detektorów, które zostały zaprezentowane w tabeli. Istnieje także wiele innych klas detektorów awarii, różniących się wymaganiami odnośnie dokładności i kompletności.

Relacja redukcji detektorów

Pomiędzy różnymi klasami detektorów awarii można wyznaczyć pewną hierarchię pozwalającą wykorzystywać detektory (określane nieformalnie jako **silniejsze**) w celu emulowania własności innych (które określa się wtedy jako detektory *słabsze*).

Mówimy, że istnieje algorytm redukcji $A^{R}_{FD \to FD'}$, jeżeli dla każdego przebiegu tego algorytmu potrafi on przekształcić wyjście detektora awarii FD w możliwe wyjście detektora FD'.

Relacja redukcji detektorów (2)

Jeżeli istnieje algorytm redukcji $A^{R}_{FD \to FD'}$ przekształcający FD w FD', to mówi się, że FD' jest **redukowalny** do FD lub *słabszy* niż FD, co zapisuje się jako $FD \succeq FD'$. Oznacza to, ze problem rozwiązywalny za pomocą FD' da się także rozwiązać za pomocą detektora FD.

Jeżeli FD redukuje się do FD' oraz FD' redukuje się do FD, to mówi się, że FD oraz FD' są **równoważne**. Wreszcie, jeżeli FD redukuje się do FD' ale FD' nie można zredukować do FD, to mówi się, że FD' jest *ściśle słabszy* niż FD.

Jeżeli natomiast dwa detektory nie redukują się do siebie wzajemnie, to detektory takie nazywamy nieporównywalnymi.

Relacje między detektorami

Doskonały detektor awarii może być scharakteryzowany własnością silnej kompletności (SC) i własnością silnej dokładności (SA). Może się jednak zdarzyć, że nie będziemy potrzebować detektora z tak silnymi własnościami, które mogą powodować zwiększenie wymagań dotyczących synchronizmu systemu. W takim przypadku możemy osłabić jedną lub obie własności w zależności od potrzeb.

Na rysunku zaprezentowano wszystkie możliwości związane z „osłabieniem" doskonałego detektora awarii.

Do wyboru mamy trzy scenariusze:

- Osłabiona zostanie silna kompletność do słabej kompletności (SC → WC). W wyniku tego działania powstanie detektor klasy Q.
- Osłabiona zostanie silna dokładność do słabej dokładności (SA → WA). W wyniku tego działania powstanie detektor klasy S.
- Osłabiona zostanie silna dokładność do ostatecznie silnej dokładności (SA → ESA). W wyniku tego działania powstanie detektor klasy ◊P.
-

Hierarchia detektorów

Ostatecznie możemy zobrazować wszystkie zależności pomiędzy poszczególnymi klasami detektorów awarii przedstawiając sześcian, którego wierzchołkami są właśnie poszczególne klasy detektorów.

Każda z krawędzi tego sześcianu będzie reprezentowała pewną relację pomiędzy wierzchołkami (klasami detektorów), które łączy.

Zacznijmy może od detektorów równoważnych. Są to klasy P≅Q, ◊P≅◊Q, S≅W, ◊S≅◊W. Krawędzie łączące odpowiednie pary zostaną zastąpione symbolem dwukierunkowej strzałki rysowanej linią przerywaną.

Inne pary detektorów połączone wspólną krawędzią znajdują się w zdefiniowanej wcześniej relacji redukcji. Dla przykładu można wymienić tutaj pary S≽◊S (◊S jest ściśle słabszy od S), P≽S, ◊P≽◊S czy też Q≽◊Q.

I na koniec wierzchołki znajdujące się po przekątnych czy to sześcianu czy też kwadratu (boku) tworzą pary nieporównywalnych klas. Przykładowo mogą to być pary (P, ◊Q) czy też (P, ◊W).

Jakość detekcji

Detektory awarii można oceniać na podstawie jakości usług. Właściwości kompletności i dokładności, choć o bardzo dużym znaczeniu teoretycznym, mają niewielkie znaczenie praktyczne. Osoby korzystające z systemów z detektorami awarii nie oczekują, że w nieskończonym czasie ich programy będą działać poprawnie – interesują ich raczej bardziej realistyczne kryteria. Dwoma interesującymi własnościami mogłyby być **prędkość** wykrywania awarii (liczba jednostek czasu wymaganych do wykrycia błędnego procesu) oraz **dokładność**, opisująca, jak dobrze detektorowi udaje się unikać pomyłek.

Metryki

Jedna z propozycji oceny jakości detektorów awarii wprowadziła trzy główne oraz cztery dodatkowe metryki jakości. Są to czas wykrycia pomyłki, czas trwania pomyłki, czas pomiędzy kolejnymi pomyłkami, prawdopodobieństwo trafności zapytania, średni współczynnik pomyłek, przyszły okres pracy bezbłędnej.

- **Czas wykrycia pomyłki** (ang. *detection time*) definiuje się jako czas, który upływa od awarii pewnego procesu do chwili, gdy zacznie on być permanentnie podejrzewany.
- **Czas trwania pomyłki** (ang. *mistake duration*) definiuje się jako czas potrzebny detektorowi, by zaprzestał podejrzewać poprawny proces.
- **Czas pomiędzy kolejnymi pomyłkami** (ang. *mistake recurrence time*) określa się jako czas między dwiema kolejnymi pomyłkami.
- **Prawdopodobieństwo trafności zapytania** (ang. *query accuracy probability*) oznacza prawdopodobieństwo prawdziwości osądu detektora.
- **Średni współczynnik pomyłek** (ang. *average mistake rate*) czyli średnia liczba pomyłek w jednostce czasu.
- **Okres pracy bezbłędnej** (ang. *good period duration*) określona jako długość okresów czasu, podczas których detektor nie popełnia pomyłek.
- **Przyszły okres pracy bezbłędnej** (ang. *forward good period duration*) określona jako oczekiwana długość okresu czasu, który mija między losowo wybranym momentem czasu a chwilą, w której detektor popełnia kolejną pomyłkę.

Wszystkie metryki zdefiniowane są dla pary procesów P_i i P_j, gdzie P_i monitoruje P_j, i P_i nie ulega awarii (założenia te nie wpływają na ogólność wyników). Na kolejnych slajdach zaprezentowane zostaną wybrane metryki wraz z przykładami.

W zależności od wymagań aplikacyjnych przedstawione metryki mogą okazać się niewystarczające. Algorytmy detekcji są bardzo różnorodne, ze względu na wymagania dotyczące warstwy komunikacyjnej, złożoności komunikacyjnej i czasowej oraz stopnia zawodności. Wiedząc, iż dla programisty każde z tych kryteriów może okazać ważne, wskazane wydaje się stworzenie metryk uwzględniających te kryteria. Pamiętając o tym zaproponowano ocenę nowego algorytmu, biorąc pod uwagę pesymistyczne zapotrzebowanie na pasmo. Znajomość tego parametru jest niezwykle cenna gdy należy liczyć się z ograniczonymi zasobami transmisyjnymi. Określa więc on skalowalność algorytmu – cechę niezwykle pożądaną nie tylko w przypadku detektorów, ale każdego algorytmu rozproszonego. Natomiast w innej pracy podczas pomiarów uwzględniono obciążenie procesora i zużycie pamięci przez algorytm detekcji. Należy tu jednak zauważyć, iż metryka ta odzwierciedla w rzeczywistości jakość implementacji algorytmu, a nie jakość samego algorytmu.

Czas wykrycia pomyłki $\mathcal{T}_D$

Metryka ta określa, jak szybko detektor jest w stanie wykryć awarię węzła. Jest to jedyna i wystarczająca metryka określająca kompletność detektora. Jej wartością jest średni czas jaki upływa od awarii procesu P_i (na rysunku awaria ma miejsce w chwili τ_1), do momentu ostatecznego wykrycia awarii przez detektor FD (w przykładzie jest to chwila τ_2).

Czas trwania pomyłki $\mathcal{T}_M$

Metryka ta jest w rzeczywistości zmienną losową reprezentującą czas trwania pomyłki. W przykładzie widocznym na rysunku jest to czas pomiędzy chwilą τ_1, w której detektor popełnia pomyłkę niesłusznie podejrzewając proces P_i, a chwilą τ_2, w której koryguje swoją pomyłkę przestając podejrzewać proces P_i.

Okres między kolejnymi pomyłkami $\mathcal{T}_{MR}$

Określa okres czasu jaki upływa pomiędzy dwiema kolejnymi pomyłkami detektora. W przykładzie na rysunku w chwili τ_1 detektor niesłusznie zaczął podejrzewać proces P_i. Po 5 jednostkach czasu skorygował swoją pomyłkę, jednak w chwili τ_2 ponownie zaczął podejrzewać poprawy proces P_i. Tak więc w przykładzie tym $\mathcal{T}_{MR}$ będzie długością okresu czasu pomiędzy chwilą τ_1 i τ_2.

Prawdopodobieństwo dokładności zapytania Pr^A

Pr^A jest prawdopodobieństwem, że informacja dostarczona przez detektor w losowym momencie jest poprawna (ang. *accurate*). Jest to miara dotycząca dokładności detektora. Warto tutaj zwrócić uwagę, że dla detektorów spełniających warunek silnej dokładności, metryka ta zawsze przyjmuje wartość 1, jeżeli detektory pytany jest o stan procesu działającego poprawnie. Podobnie, jeżeli detektor spełnia warunek silnej kompletności, to ostatecznie metryka ta będzie przyjmować wartość 1, jeżeli detektor jest pytany o stan węzła wadliwego. Stąd, jeżeli detektor jest klasy P, to od pewnego momentu (a dokładnie od momentu określonego metryką $\mathcal{T}_D$) Pr^A będzie zawsze przyjmować wartość 1.

Metryka Pr^A – przykład

Okazuje się jednak, iż wspomniana metryka nie jest wystarczająca do stwierdzenia, który z detektorów jest lepszy, ze względu na dokładność detekcji. Na poniższym rysunku widać, iż dla obydwu detektorów metryka Pr^A przyjmuje taką samą wartość, czyli $0{,}75$ (w ciągu 3+1 jednostek czasu, przez jedną jednostkę detektor FD_1 się myli, stąd $Pr^A(FD_1) = 1/(3+1)=0{,}75$, a detektor FD_2 myli się przez 4 jednostki czasu na 12+4 jednostek czasu, stąd $Pr^A(FD_2) = 4/(12+4)=0{,}75$). Detektor FD_2 jednak popełnia pomyłki rzadziej (raz na 16 jednostek czasu), w związku z czym jest on lepszy, choć w świetle tej metryki obydwa detektory są równie dobre. Dla określenia dokładności detekcji konieczne jest więc zdefiniowanie dalszych metryk.

Średni współczynnik pomyłek A_M

Metryka ta określa średnią liczbę pomyłek jakie detektor popełnia w jednostce czasu. Ponownie, okazuje się, że jest to metryka nie zawsze wystarczająca do stwierdzenia, który z detektorów jest lepszy.

Metryki A_M i Pr^A – przykład

Na powyższym diagramie widać, iż detektor FD_1 jest lepszy od FD_2 ze względu na parametry A_M (dla FD_1 $AM(FD_1)=1/15$, dla FD_2 $A_M(FD_2)=1/10$) oraz Pr^A (dla FD_1 $Pr^A(FD_1)=10/15$, dla FD_2 $Pr^A(FD_2) =9/15$). FD_2 jednak szybciej koryguje swoje pomyłki, co dla niektórych aplikacji może mieć zasadnicze znaczenie. Widać stąd, iż nawet połączenie tych metryk nie jest w stanie odzwierciedlić jakości detekcji.

Implementacja detektora awarii: mechanizm pulsu

Implementacja detektora wykorzystująca mechanizm pulsu (ang. *heartbeat*) polega na tym, że proces monitorowany co pewien zdefiniowany czas T_j^P wysyła wiadomość typu „I'm alive". Odebranie tego typu wiadomości po drugiej stronie łącza komunikacyjnego przez detektor FD w czasie T_i^{to} od ostatnio odebranej wiadomości pozwala uznać proces P_i za poprawny. W przeciwnym wypadku detektor FD zaczyna podejrzewać proces P_i. Warto podkreślić fakt, że w tym scenariuszu aktywność leży po stronie procesu monitorowanego, który musi okresowo wysyłać wiadomości.

Implementacja detektora awarii: mechanizm odpytywania

Implementacja detektora wykorzystująca mechanizm odpytywania (ang. *polling, interrogation*) jest modyfikacją poprzedniego podejścia. Polega na tym, że proces monitorujący co pewien zdefiniowany okres czasu T_i^p wysyła pytanie o status typu „Are you alive?". Proces monitorowany po odebraniu tego typu komunikatu odpowiada na niego wysyłając wiadomość typu „I am alive". Jeśli odpowiedź nie dotrze do procesu monitorującego (detektora awarii FD) przez zadany czas T_i^{to}, to detektor zacznie podejrzewać P_i o awarię. Ten schemat monitorowania umożliwia lepszą kontrolę procesu monitorowania, gdyż aktywność leży w tym rozwiązaniu po stronie detektora awarii.

Doskonały detektor awarii

Detektor tej klasy ma własność silnej kompletność oraz silnej dokładności.

Algorytm implementujący doskonały detektor awarii z użyciem mechanizmu pulsu: Założenia

W przedstawionej dalej implementacji doskonałego detektora awarii zakładamy istnienie niezawodnych kanałów komunikacyjnych łączących procesy. Ponadto znany jest maksymalny czas transmisji komunikatu. W porównaniu do tej wartości czasy przetwarzania lokalnego i ewentualne różnice szybkości zegarów lokalnych są pomijalnie małe.

Algorytm implementujący doskonały detektor awarii z użyciem mechanizmu pulsu: Koncepcja

Algorytm wykorzystuje zegary lokalne procesów, które co stały odcinek czasu T_i^p inicjują operację wysłania wiadomości do wszystkich procesów. Długość T_i^{to} jest tak dobrana, by wszystkie wysłane wiadomości mogły dotrzeć do detektora. Brak wiadomości od pewnego procesu w danym okresie powoduje dodanie go do zbioru $suspected$ (o ile już się w tym zbiorze nie znajduje), co jest określane jako **wykrycie awarii** przez detektor.

Algorytm implementujący doskonały detektor awarii z użyciem mechanizmu pulsu: Zdarzenie cykliczne

W przedstawionym dalej algorytmie używane są dodatkowe dwa oznaczenia: $e\_clock(P_i, period_i^{to})$ oraz $e\_clock(P_i, period_i^p)$. Są to zdarzenia, zachodzące niezależnie i regularnie, w procesie P_i, związane z upływem czasu, mierzonym przez lokalny zegar. Odstęp między zajściem dwóch kolejnych zdarzeń określany jest przez drugi parametr, odpowiednio $period_i^{to}$ lub $period_i^p$. Stąd,

- $e\_clock(P_i, period_i^{to})$ – cykliczne zdarzenie upływu czasu maksymalnego oczekiwania na otrzymanie pulsu.
- $e\_clock(P_i, period_i^p)$ – cykliczne zdarzenie upływu czasu wysłania pulsu.

Detektor klasy *P* : Algorytm (1)

Algorytm używa wiadomości $heartbeat$ typu HEARTBEAT. Zbiór $suspected_i$ zawiera identyfikatory procesów, które są podejrzewane przez detektor FD_i działający w monitorze Q_i. Zbiór ten początkowo jest pusty. Zbiór $correct_i$ zawiera identyfikatory procesów, które detektor uznaje za poprawne. Zbiór ten początkowo obejmuje wszystkie procesy. Zmienna $period_i^{to}$ określa długość maksymalnego oczekiwania na otrzymanie wiadomości typu HEARTBEAT (pulsu). Zmienna $period_i^p$ określa długość czasu pomiędzy wysłaniem dwóch kolejnych wiadomości HEARTBEAT (pulsu).

Detektor klasy P: Algorytm (2)

Co pewien czas algorytm sprawdza, które procesy należy zacząć podejrzewać. Jeżeli nie otrzymano w ostatnim czasie wiadomości typu HEARTBEAT od danego procesu (nie znajduje się on w zbiorze $correct_i$) i nie jest on jeszcze podejrzewany, to dołączany on jest do zbioru podejrzewanych procesów.

Co pewien czas rozsyłane są wiadomości typu HEARTBEAT do wszystkich pozostałych procesów. Otrzymanie takiej wiadomości powoduje dodanie procesu do zbioru $correct_i$.

Implementacja detektora ◊P

Ostatecznie doskonały detektor awarii posiada silną własność kompletności i ostatecznie silną własność dokładności.

Algorytm implementujący ostatecznie doskonały detektor awarii z użyciem mechanizmu pulsu: Założenia

W przypadku implementacji przedstawionej na kolejnych slajdach przyjęto istnienie niezawodnych łączy komunikacyjnych, założono częściowo synchroniczny model systemu, w którym zarówno czasy przetwarzania lokalnego jak i przesunięcie zegarów lokalnych mogą być pominięte. Implementacja będzie wykorzystywała technikę pulsu opartą na okresowym rozsyłaniu wiadomości przez proces monitorowany.

Algorytm implementujący ostatecznie doskonały detektor awarii z użyciem mechanizmu pulsu: Koncepcja

Wykorzystywany jest zegar lokalny procesu P_i. Proces ten co pewien stały okres czasu o długości T_i^p inicjuje operację wysłania wiadomości do wszystkich procesów.

Procesy, od których w pewnym okresie czasu o długości T_i^{to} nie otrzymano wiadomości (pulsu) dodawane są do zbioru procesów podejrzewanych o awarię przez detektor FD_i procesu P_i. Podstawowa różnica w stosunku do poprzednio omówionego algorytmu dla detektora klasy P polega na zwiększaniu długości okresu T_i^{to} w przypadku stwierdzenia pomyłki. Z założeń dotyczących środowiska wynika, że ostatecznie stała ta będzie większa niż maksymalne możliwe opóźnienie transmisji.

Detektor klasy ◊P: Algorytm (1)

Algorytm używa typów i zmiennych o identycznym znaczeniu jak poprzednio.

Detektor klasy ◊P: Algorytm (2)

Proces detekcji wywoływany jest co pewien czas T_i^{to}. Detektor dołącza wszystkie procesy, które nie należą do zbioru $correct_i$ czyli takie, od których nie otrzymano ostatnio wiadomości typu HEARTBEAT do zbioru $suspected_i$. Jest to równoznaczne z rozpoczęciem podejrzewania tych procesów.

Jeśli natomiast proces należy zarówno do zbioru $correct_i$ jak i do zbioru $suspected_i$ to oznacza to, że detektor popełnił pomyłkę, która powinna być skorygowana. W takim przypadku proces przestaje być podejrzewany (usunięty zostaje ze zbioru $suspected_i$, oraz zwiększany jest czas, po którym procesy zostają uznane za podejrzane, jeśli nie nadejdzie od niech żadna wiadomość typu HEARTBEAT. Zbiorowi $correct_i$ przypisana jest wartość zbioru pustego.

Co pewien czas proces wysyła też wiadomości typu HEARTBEAT do wszystkich pozostałych procesów. Symbol Δ w wierszu 7 algorytmu oznacza pewną stałą dobraną przez użytkownika.

Detektor klasy ◊P: Algorytm (3)

Po otrzymaniu wiadomości typu HEARTBEAT (pulsu) od innego procesu, proces ten dołączany jest do zbioru procesów uznanych za poprawne $correct_i$.

Problem wyboru lidera

W systemach rozproszonych bywają sytuacje, w których problemem nie jest wykrycie, które procesy uległy awarii, ale wspólne uzgodnienie wyboru jednego procesu, który wciąż jest poprawny i któremu wszystkie inne procesy **ufają**. Detektor wspomagający wybór jednego zaufanego procesu określany jest jako **detektor lidera Ω.** Taki zaufany proces może działać jako koordynator, niezbędny w pewnych rozproszonych algorytmach. Stąd też problem ten nazywa się problemem wyboru lidera (ang. *eventual leader election*).

Wprowadzimy teraz abstrakcję **detektora Ω**, inaczej **detektora ostatecznego lidera** (ang. *eventual leader detector*), wykonującego algorytm wyboru lidera. Mechanizm ten ma zapewniać, że wszystkie poprawne procesy zgodzą się ostatecznie na wybór tego samego procesu.

Mechanizm wyboru ostatecznego lidera posiada własności ostatecznej dokładności i ostatecznej zgodności.

- Własność **ostatecznej dokładności** (ang. *eventual accuracy*) oznacza, że istnieje pewna chwila czasu, po której wszystkie poprawne procesy ufają (nie podejrzewają) pewnemu poprawnego procesu.
- Własność **ostateczna zgodność** (ang. *eventual agreement*) oznacza, że istnieje pewna chwila czasu, po której nie ma dwóch takich poprawnych procesów, które by ufały dwóm różnym procesom.

Własności te oznaczają, że chociaż w czasie działania algorytmu lider może się zmieniać w dowolny sposób, jednakże ostatecznie wszystkie procesy zaczynają ufać jednemu liderowi i wyboru swojego już nie zmieniają – wybór lidera stabilizuje się.

Algorytm wyboru ostatecznego lidera: Założenia

Rozważać będziemy teraz rozwiązanie problemu wyboru lidera dla klasy awarii powtarzalnych przy założeniu, że procesy są odtwarzane, a kanały komunikacyjne posiadają własności kanałów rzetelnych.

Odtwarzalność procesu oznacza dostępność procedur STORE i RETRIEVE, które umożliwiają, odpowiednio, zapamiętanie a później odtworzenie po ewentualnej awarii wartości wybranych parametrów.

Algorytm wyboru ostatecznego lidera: Koncepcja

W celu implementacji mechanizmu detektora ostatecznego lidera można wykorzystać poprzednio poznaną implementację ostatecznie doskonałego detektora awarii. Istotnie, wystarczające do rozwiązania problemu jest wybieranie jako lidera procesu o najwyższym identyfikatorze spośród wszystkich nie podejrzewanych przez ostatecznie doskonały detektor awarii. Ostatecznie, o ile co najmniej jeden proces jest poprawny, wszystkie procesy będą ufać dokładnie jednemu, poprawnemu procesowi. Ponieważ jednak dalej będziemy rozważać awarie powtarzalne, konieczne jest wprowadzenie dodatkowego warunku, który musi zostać spełniony przez potencjalnego lidera – liczba awarii takiego procesu powinna być najmniejsza spośród wszystkich procesów.

Problem wyboru lidera: Algorytm (1)

Przedstawiony algorytm używa wiadomości HEARTBEAT. które zawierają numer epoki w polu *epochNo*. Epoką (ang. *epoch number*) określa się okres poprawnego działania procesu, który może zostać ewentualnie zakończony awarią procesu. Po awarii następuje odtwarzanie działania procesu (ang. *recovery*), przy czym każde odtworzenie oznacza początek nowej epoki. Dodatkowo zdefiniowano typ TRUSTED jako rekord zawierający pole identyfikatora procesu *pId* oraz pole *epochNo*, zawierające numer epoki procesu w chwili, gdy został wybrany jako kandydat na lidera.

Problem wyboru lidera: Algorytm (2)

Podobnie jak w poprzednich algorytmach, stosowany jest mechanizm pulsu: każdy proces wysyła cyklicznie wiadomości *heartbeat* (typu HEARTBEAT). Stosowane też są następujące zmienne: Zmienna $leader_i$ typu PROCESS_ID, zawierająca identyfikator bieżącego lidera. Analogicznie, zmienna $previousLeader_i$ zawiera identyfikator poprzedniego lidera. Elementy zbioru $candidateSet_i$ są parami składającymi się z identyfikatorów możliwych kandydatów na lidera i numerów epoki. Zmienna $epochNo_i$ zawiera numer epoki, odzwierciedlający liczbę awarii i wznowień pracy procesu P_i.

- Zmienne *result* oraz *candidate* –wykorzystywane są przez funkcję SELECTNEWLEADER, która zostanie przedstawiona na kolejnym slajdzie.
- Zmienna $period_i^p$ – określa długość czasu pomiędzy wysłaniem dwóch kolejnych wiadomości HEARTBEAT (pulsu). Jej wartość początkowa jest równa stałej T_i^p.
- Zmienna $period_i^{to}$ – określa długość okresu czasu między kolejnymi wywołaniami wyboru lidera. Jej początkowa wartość jest równa stałej T_i^{to}.

Problem wyboru lidera: Algorytm (3)

Procedura SELECTNEWLEADER wybiera dla danego zbioru (podanego jako argument) jako lidera proces o najmniejszej liczbie awarii, którego identyfikator jest najmniejszy.

Problem wyboru lidera: Algorytm (4)

Co jakiś czas procesy $period_i^{to}$ sprawdzają, czy bieżący lider może nim pozostać. W tym celu wywołują funkcję SELECTNEWLEADER, która wybiera jeden proces z zbioru $candidateSet_i$. Jeżeli w wyniku wykonanie tego kroku algorytmu lider się zmienił, zwiększane jest opóźnienie między kolejnymi próbami odnalezienia lidera. Jeżeli wybrano jakiegoś nowego lidera, proces zaczyna mu ufać.

Równolegle, co pewien czas $period_i^{p}$ proces wysyła do wszystkich pozostałych procesów swój bieżący numer epoki. Symbol Δ w wierszu 18 algorytmu oznacza pewną stałą dobraną przez użytkownika.

Problem wyboru lidera: Algorytm (5)

Po otrzymaniu wiadomości typu HEARTBEAT (pulsu) od innego procesu, następuje sprawdzenie, czy proces ten znajduje się już w zbiorze $candidateSet_i$. Jeżeli tak, to w zbiorze $possibleSet_i$ uaktualniany jest ewentualnie numer epoki. W przeciwnym wypadku, identyfikator tego procesu wraz z numerem epoki jest dołączany jest do zbioru $candidateSet_i$.

Po każdym wznowieniu (restarcie) procesu, modelowanym jako zajście zdarzenia $e\_recovery(P_i)$, wczytywany jest numer epoki za pomocą procedury RETRIVE. Wszystkie pozostałe zmienne wchodzące w skład stanu procesu są inicjowane wartościami początkowymi. Następnie numer ten jest zwiększany o jeden, zapisany do pamięci dyskowej za pomocą procedury STORE oraz rozsyłany do wszystkich monitorów.

Modele przetwarzania

Złożenie różnych założeń dotyczących procesów, rodzajów kanałów komunikacyjnych czy też klas detektorów awarii jest równoznaczne ze zdefiniowaniem specyficznych modeli przetwarzania.

Istnieje oczywiście bardzo wiele możliwych kombinacji przyjmowanych założeń, a stąd wiele modeli. W literaturze rozważane są na ogół modele: z jawnymi awariami, z ukrytymi awariami, z ostatecznie jawnymi awariami, ze wznowieniami po awariach oraz model losowy.

- W modelu z jawnymi awariami (ang. *fail-stop model, crash-stop*) procesy wykonują deterministyczne algorytmy, chyba że zaprzestaną działania w wyniku awarii. Kanały są niezawodne i jest dostępny doskonały detektor awarii. Rozwiązywanie problemów w tym modelu jest stosunkowo łatwe.
- W modelu z ukrytymi awariami (ang. *fail-silent model*) przyjmuje się, że procesy wykonują deterministyczne algorytmy, chyba że zaprzestaną działania w wyniku awarii, kanały są niezawodne, ale nie jest dostępny doskonały detektor awarii.
- W modelu z ostatecznie jawnymi awariami (ang. *fail-noisy model*) procesy wykonują deterministyczne algorytmy, chyba że zaprzestaną działania w wyniku awarii. Kanały są niezawodne, dostępny jest ostatecznie doskonały detektor awarii lub detektor typu Ω.
- W modelu ze wznowieniami po awariach (ang. *fail-recovery, crash-recovery*) procesy wykonują deterministyczne algorytmy, chyba że zaprzestaną działania w wyniku awarii. Po awarii działanie procesu jest wznawiane. Procesy są połączone kanałami wytrwałymi.
- W modelu losowym (ang. *randomized model*) procesom udostępniona jest losowa wyrocznia, dzięki której mogą wybierać jedną z kilku możliwych ścieżek przetwarzania – w tym przypadku algorytmy nie są deterministyczne.