

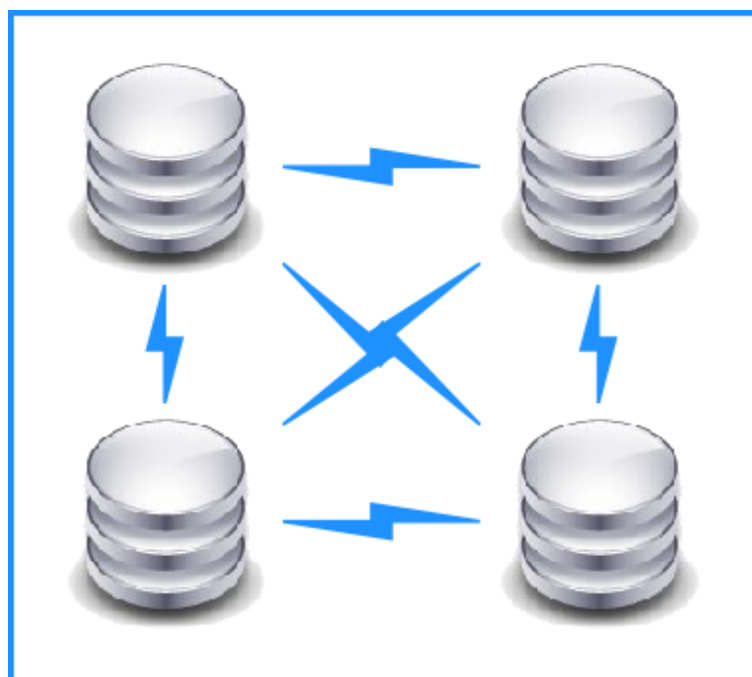
---

# Rozproszona baza danych

## Dokumentacja projektu

---

Piotr Kalański, Marcin Kubacki, Marek Kurdej, Adrian Wiśniewski



# Spis treści

<b>1</b>	<b>Wprowadzenie</b>	<b>2</b>
<b>2</b>	<b>Architektura</b>	<b>3</b>
2.1	Warstwa komunikacyjna . . . . .	3
2.2	Warstwa logiki aplikacyjnej . . . . .	3
2.3	Warstwa dostępu do danych . . . . .	3
<b>3</b>	<b>Trójfazowe zatwierdzanie transakcji</b>	<b>6</b>
3.1	Model automatowy . . . . .	6
3.2	Opis słowny . . . . .	7
<b>4</b>	<b>Dynamiczne dodawanie nowych węzłów</b>	<b>8</b>
<b>5</b>	<b>Diagramy sekwencji</b>	<b>13</b>
5.1	Wiadomości . . . . .	13

# Rozdział 1

## Wprowadzenie

Obecnie redundancja w systemach bazodanowych jest koniecznością. Zabezpiecza ona przed utratą danych, gdyby z pewnych powodów pojedynczy węzeł odmówił posłuszeństwa. Rozwiązaniem typowym w środowiskach produkcyjnych jest klastr wysokiej dostępności (ang. *emphHigh Availability Cluster*), który powiela transakcje na poszczególnych węzłach.

Tematem projektu było stworzenie rozproszonej bazy danych, a właściwie klastra wysokiej dostępności, która umożliwiałaby wykonywanie zapytań na dowolnym węźle, a następnie automatyczną synchronizację takiej operacji na pozostałe węzły bazy. Dodatkowo w przypadku uruchomienia nowego węzła, węzeł ten powinien móc zsynchronizować się z pozostałymi węzłami.

Ze względu na ograniczenia czasowe rozwiązanie zostanie przedstawione na 2 implementacjach zamiast 4, pierwszej napisanej w języku Java oraz drugiej w .NET. Trzecia implementacja w Qt nie funkcjonowała poprawnie, więc zostanie wykluczona z dalszych rozważań.

Kluczowymi algorytmami rozproszonymi dla tego projektu były:

- trójfazowe zatwierdzanie transakcji (ang. *Three Phase Commit*)
- autorski pomysł synchronizacji dynamicznie dołączanych węzłów

Oba algorytmy zostaną szczegółowo omówione w dalszej części dokumentacji.

## Rozdział 2

# Architektura

Całe rozwiązanie składa się z kilku części. Ogólnie podzielić projekt można na następujące warstwy:

- komunikacyjną
- logiki aplikacyjnej
- dostępu do danych

### 2.1 Warstwa komunikacyjna

W skład warstwy komunikacyjnej wchodzi klasy służące do nasłuchiwania na wybranym porcie TCP i UDP. Nasłuchiwanie na porcie TCP potrzebne jest do odbierania dowolnych komunikatów obsługujących transakcje, natomiast UDP używany jest w celu odbierania tzw. "heartbeats", czyli komunikatów wysyłanych w regularnych odstępach przez wszystkie węzły w celu ogłaszania wszystkim zainteresowanym, że dany węzeł jest dostępny. Po odebraniu żądania połączenia tworzona jest obiekt klasy roboczej - TCP Worker, który zajmuje się odbieraniem pakietów i składaniem wiadomości w całość. Następnie każdy z wątków zajmujący się takimi połączeniem przekazuje wiadomość do Dispatcher'a.

Ostatecznie po przetworzeniu wiadomości musi nastąpić pewna reakcja objawiająca się wysłaniem konkretnego pakietu. I tu używane są klasy TCP i UDP sender, które wysyłają wcześniej przygotowaną wiadomość. Tu warto jeszcze wspomnieć o Hello generatorze, który okresowo wysyła heartbeats (i tylko on korzysta z protokołu UDP). Ważny jest tu TCP sender, który przechowuje pulę połączeń, ewentualnie nawiązuje nowe.

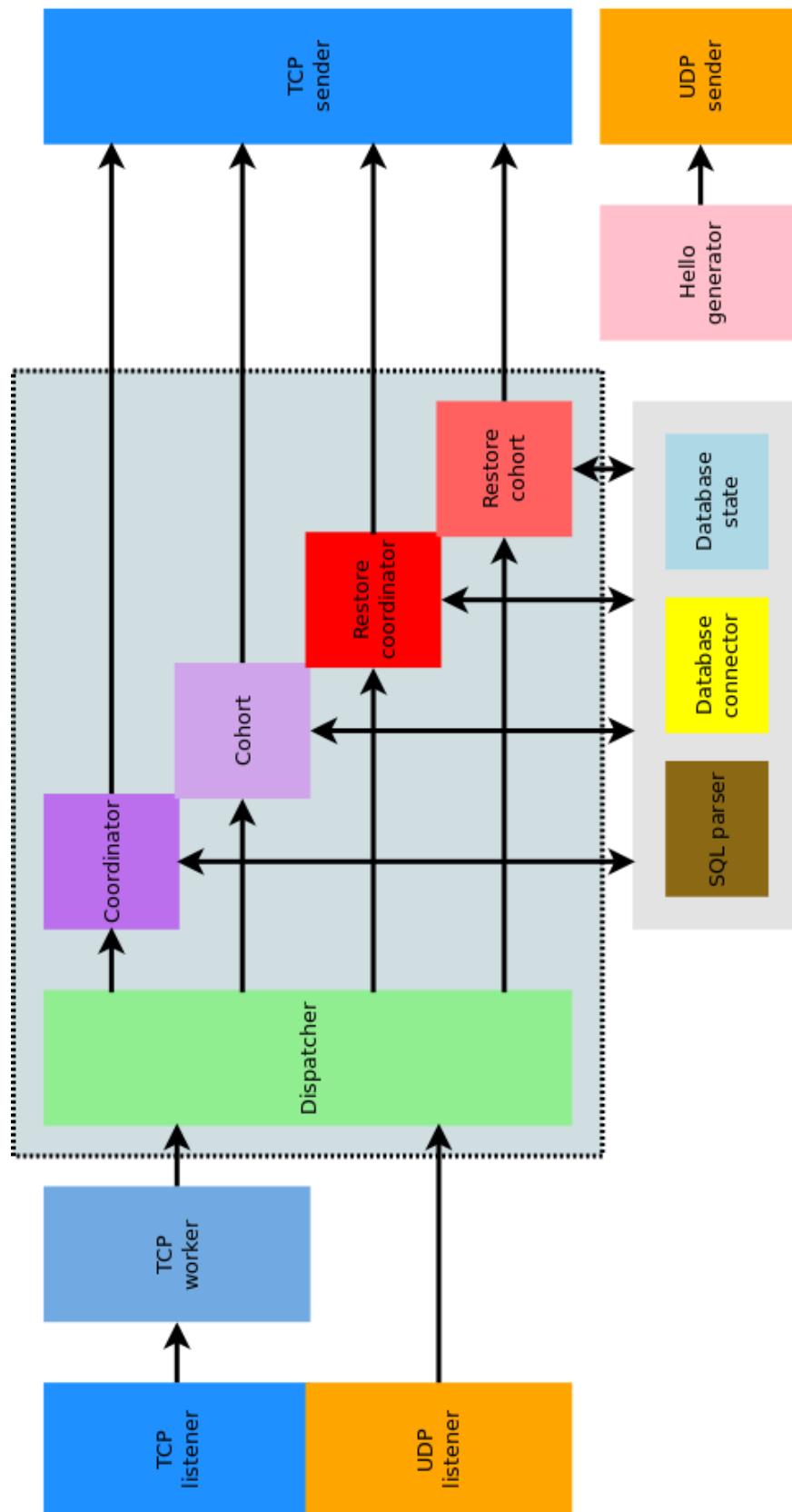
### 2.2 Warstwa logiki aplikacyjnej

Dispatcher odpowiada za klasyfikację wiadomości i odpowiednią reakcję. Reakcja ta objawia się utworzeniem odpowiedniej klasy roboczej, która będzie odpowiadała za przechodzenie po grafie stanów automatu trójfazowego zatwierdzania transakcji, lub synchronizacji węzłów. W zależności od tego, czy węzeł w danym momencie koordynuje transakcję, czy realizuje polecenia z innych węzłów tworzone są instancje obiektów Cohort lub Coordinator - podobnie dla synchronizacji (przedrostek Restore).

### 2.3 Warstwa dostępu do danych

W tej warstwie znajdują się wszelkie klasy, które mają na celu manipulować właściwymi danymi przechowywanymi w bazie. Klasą, którą po części jest logiką aplikacyjną jest Database

state. Ma ona na celu przechowywać aktualne blokady i stan bazy danych. Oprócz tego występują dwie inne pomocnicze klasy - Database connector i SQL parser, które mają na celu parsować i zapytania i umożliwić dialog z właściwą bazą danych.



Rysunek 2.1: Architektura rozwiązania

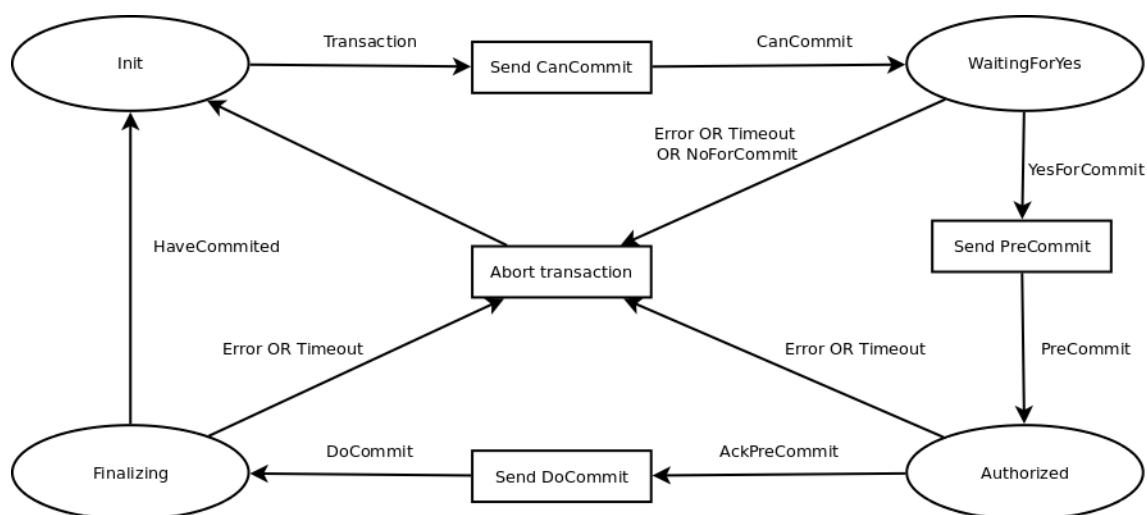
## Rozdział 3

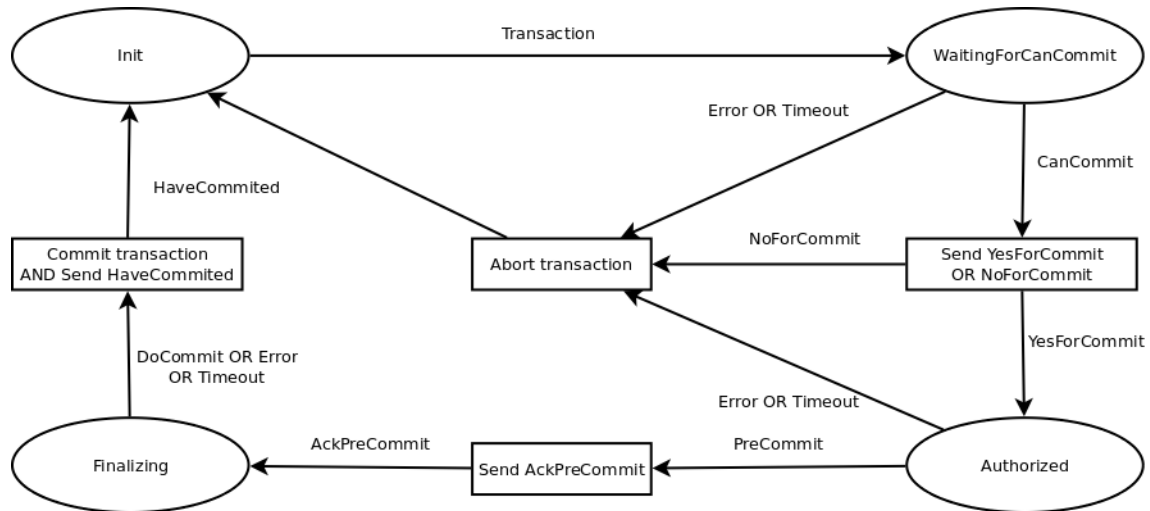
# Trójfazowe zatwierdzanie transakcji

Główną częścią logiki systemu będzie trójfazowe zatwierdzanie transakcji, które umożliwi wykonanie pojedynczej transakcji na wszystkich lub na żadnym węźle. Poniżej opis protokołu.

### 3.1 Model automatowy

Serwer (Coordinator)



**Klient (Cohort)****3.2 Opis słowny**

Zatwierdzenie trójfazowe jest rozproszonym algorytmem, który nakazuje węzła w rozproszonym systemie uzgodnić fakt zatwierdzenia transakcji. W przeciwieństwie do swojego dwufazowego poprzednika, wersja trójfazowa jest nieblokująca.

**Serwer (Coordinator)**

1. Pewien klient inicjuje sekwencję trójfazowego zatwierdzenia za pomocą pakietu **Transaction**. W razie problemu transakcja jest anulowana. Koordynator wysyła zapytanie do klientów, czy chcą zatwierdzić (**CanCommit**) i przechodzi do stanu oczekiwania.
2. W przypadku odpowiedzi pozytywnej wysyła do wszystkich wiadomość **PreCommit**, który nakazuje stacjom klienckim przygotować się do zatwierdzenia. W razie błędu, upłynięcia limitu czasu dla którejś ze stacji lub odebrania od dowolnego klienta **NoForCommit** - transakcja jest anulowana. W przeciwnym wypadku następuje trzecia faza
3. W trzeciej fazie koordynator wysyła **DoCommit**, który ostatecznie nakazuje klientom zatwierdzić lub odrzucić daną transakcję. Gdyby w którejś fazie nie dotarł pakiet - klient lub serwer odczekują określony czas i ewentualnie anulują transakcję.

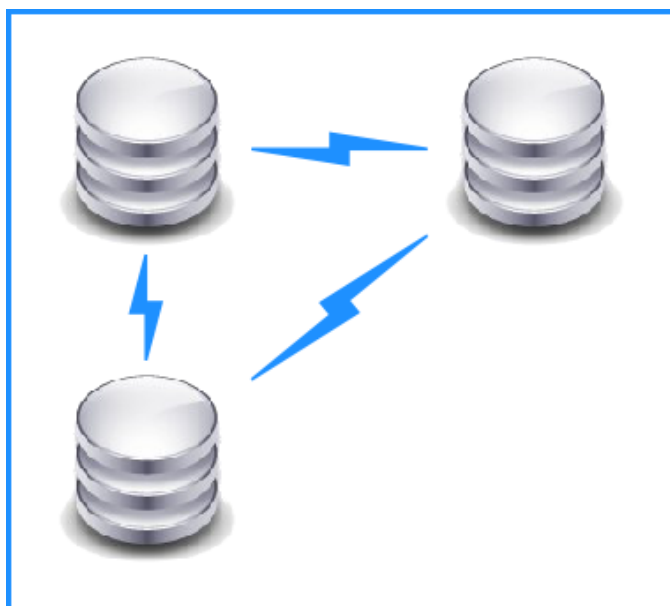
**Klient (Cohort)**

1. Klient dostaje zapytanie, czy chce zatwierdzić transakcję i odpowiada twierdząco (komunikat **YesForCommit**) i przechodzi do stanu przygotowania, ewentualnie odpowiada, że chce transakcję odrzucić (komunikat **NoForCommit**).
2. W stanie przygotowania, jeśli dostanie komunikat o anulowaniu lub minie timeout, transakcja jest anulowana. Jeśli otrzyma komunikat **PreCommit** potwierdza go komunikatem **AckPreCommit**.
3. Po odebraniu komunikatu **DoCommit** transakcja jest zatwierdzana (i wysyłany jest komunikat **HaveCommitted**), w przeciwnym wypadku (błąd lub upłynięcie timeoutu transakcja jest anulowana).

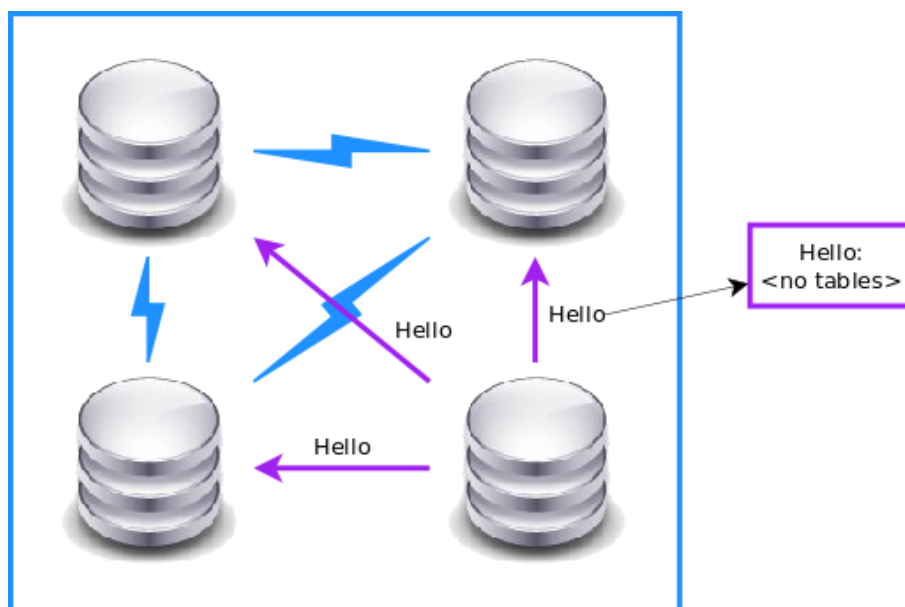


## Rozdział 4

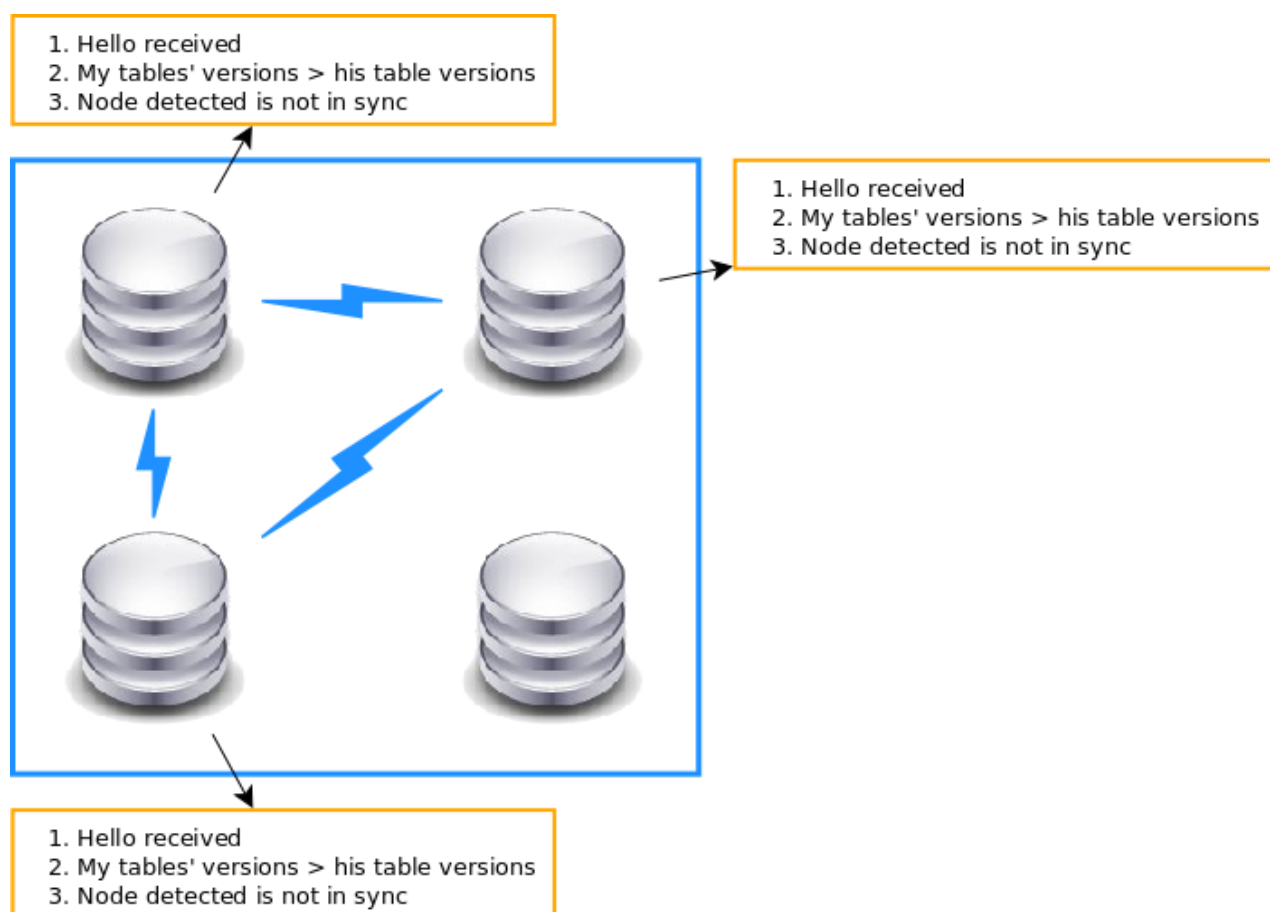
# Dynamiczne dodawanie nowych węzłów



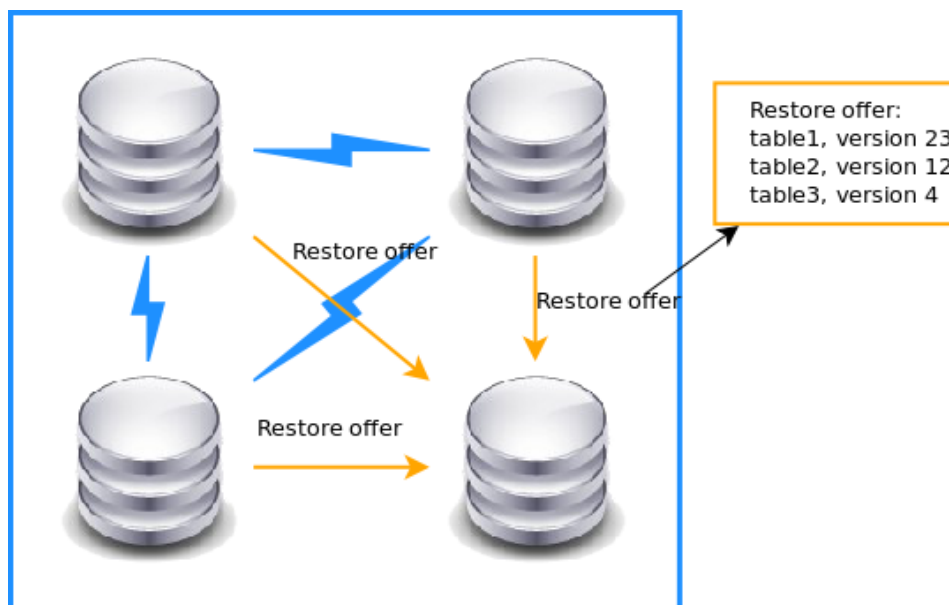
Rysunek 4.1: Początkowa faza - 3 węzły działające



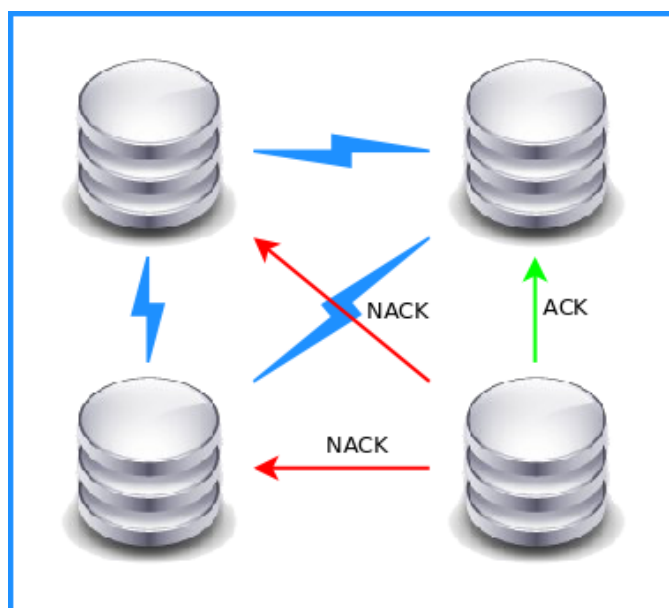
Rysunek 4.2: Nowo dodany węzeł zaczyna rozgłaszać swoje Hello



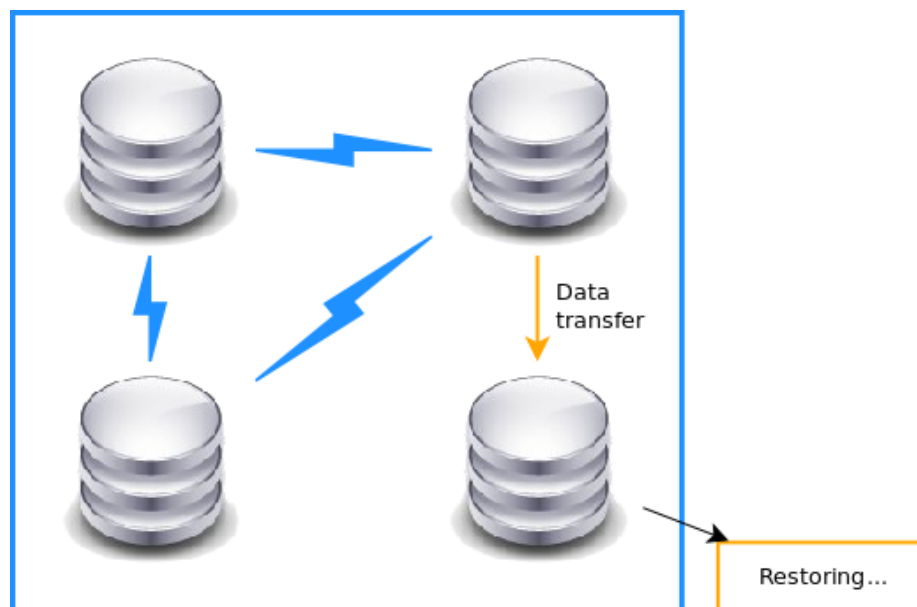
Rysunek 4.3: Obecne węzły porównują tabele w pakiecie Hello z własnymi wersjami i stwierdzają, że konieczna jest synchronizacja



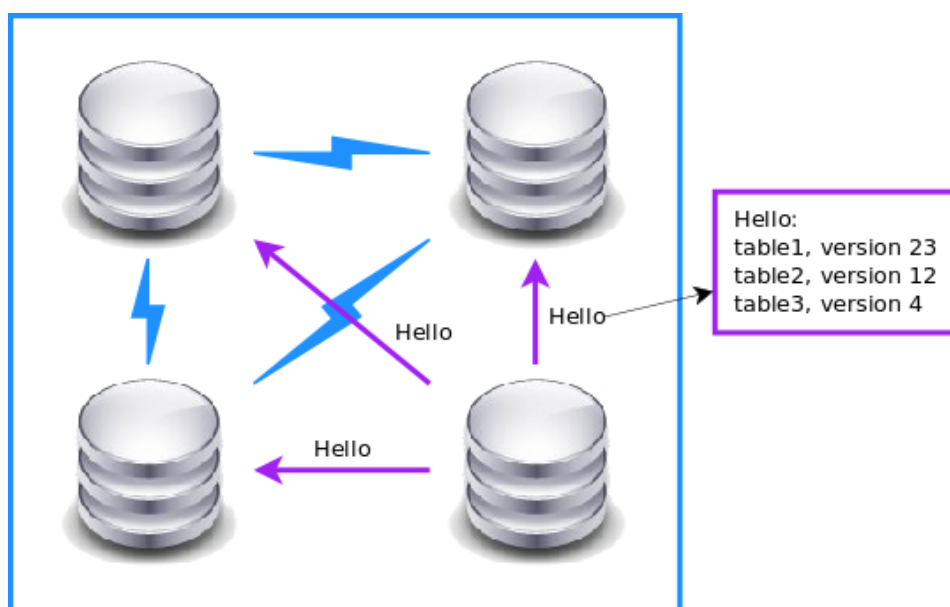
Rysunek 4.4: Węzły blokują odpowiednie obszary danych, a gdy ich listy prowadzonych sesji będą puste, wysyłana jest oferta synchronizacji



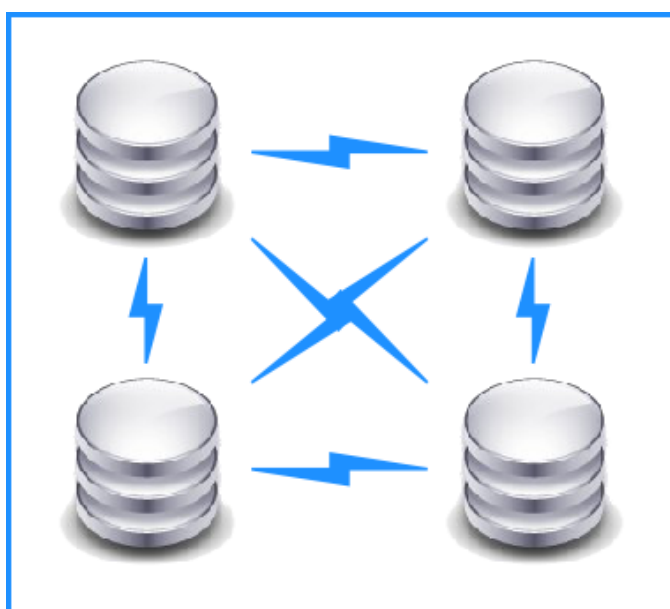
Rysunek 4.5: Nowy węzeł wybiera jeden z węzłów do synchronizacji, pozostałe odrzuca



Rysunek 4.6: Następuje transfer tabel i synchronizacja danych



Rysunek 4.7: Po synchronizacji węzeł będzie wysyłał już hello z poprawnymi wersjami tabel



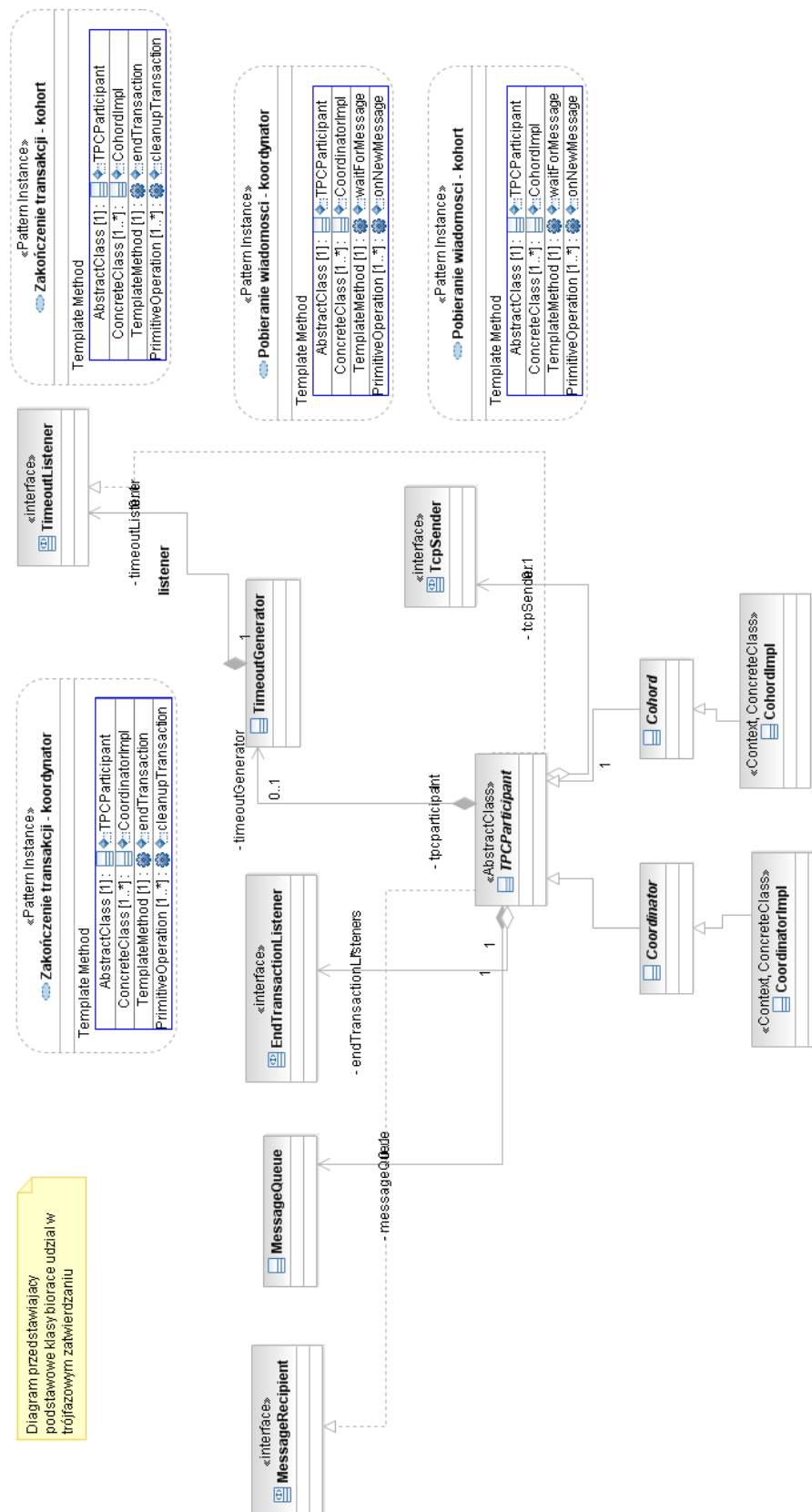
Rysunek 4.8: Ostatecznie zostanie uznany za zsynchronizowany i włączony do struktury

## Rozdział 5

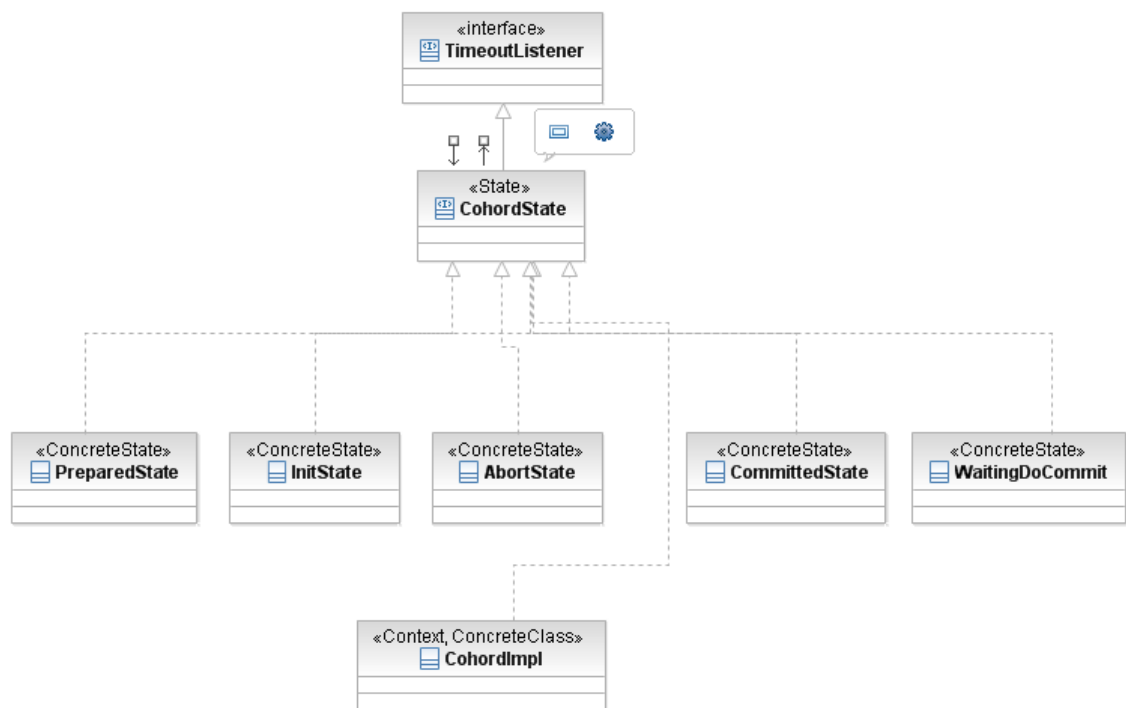
# Diagramy sekwencji

### 5.1 Wiadomości

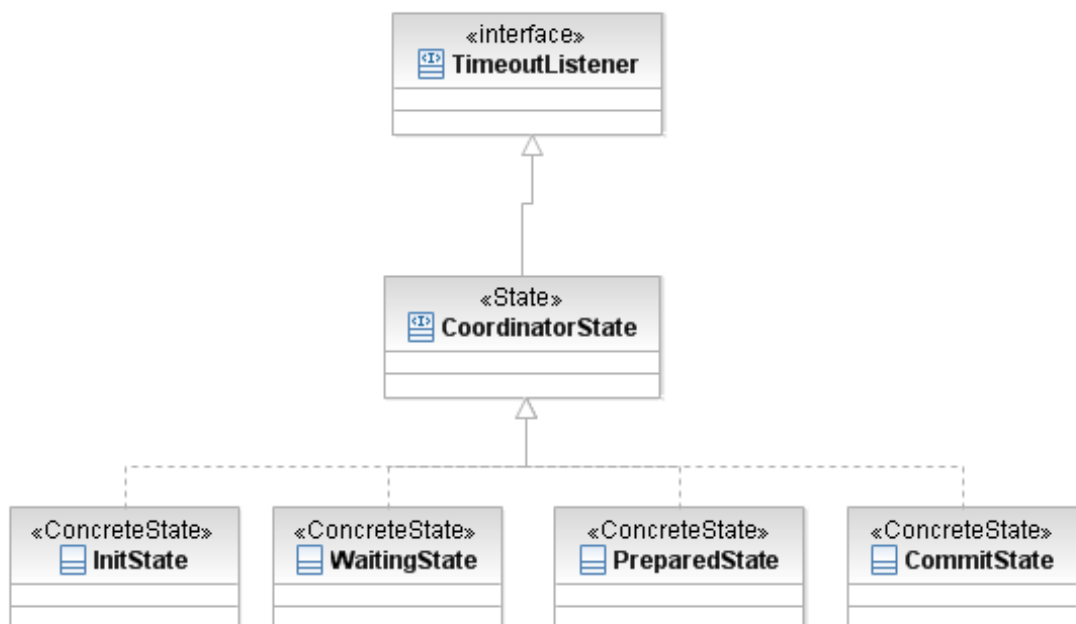
Poniżej zaprezentowane zostały diagramy sekwencji wywołań metod w klasach używanych do trójfazowego zatwierdzania transakcji. Aby wszystkie diagramy były zrozumiane na początku zaprezentowano hierarchię klas dla TPC oraz przedstawiające hierarchie klas stanów Coordinadora i Cohort'a.



Rysunek 5.1: Ogólna hierarchia klas dla TPC

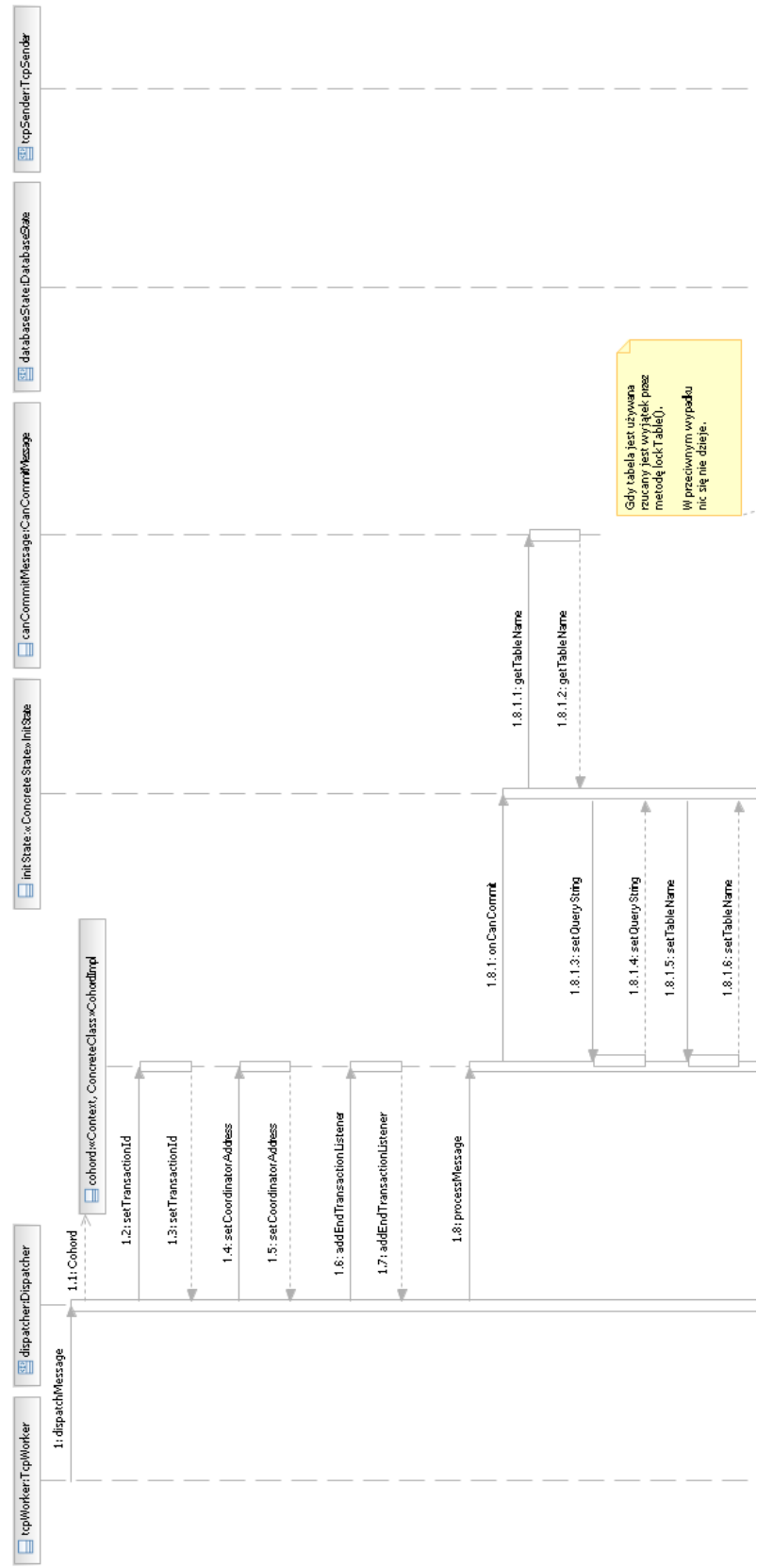


Rysunek 5.2: Hierarchia klas dla Cohort

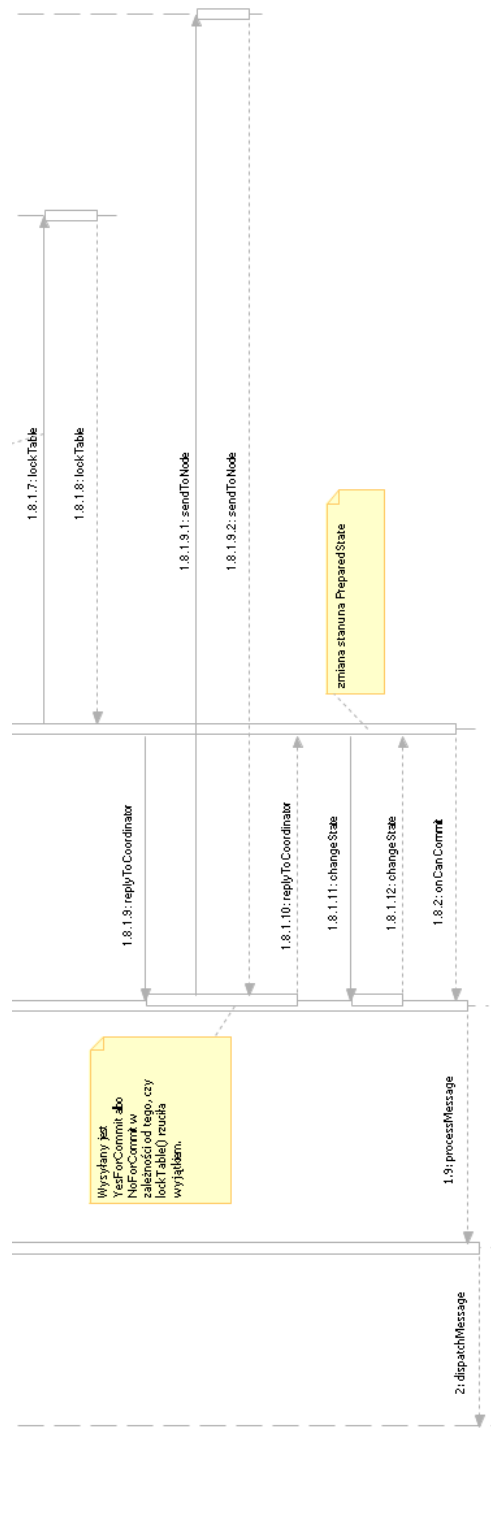


Rysunek 5.3: Hierarchia klas dla Coordinator

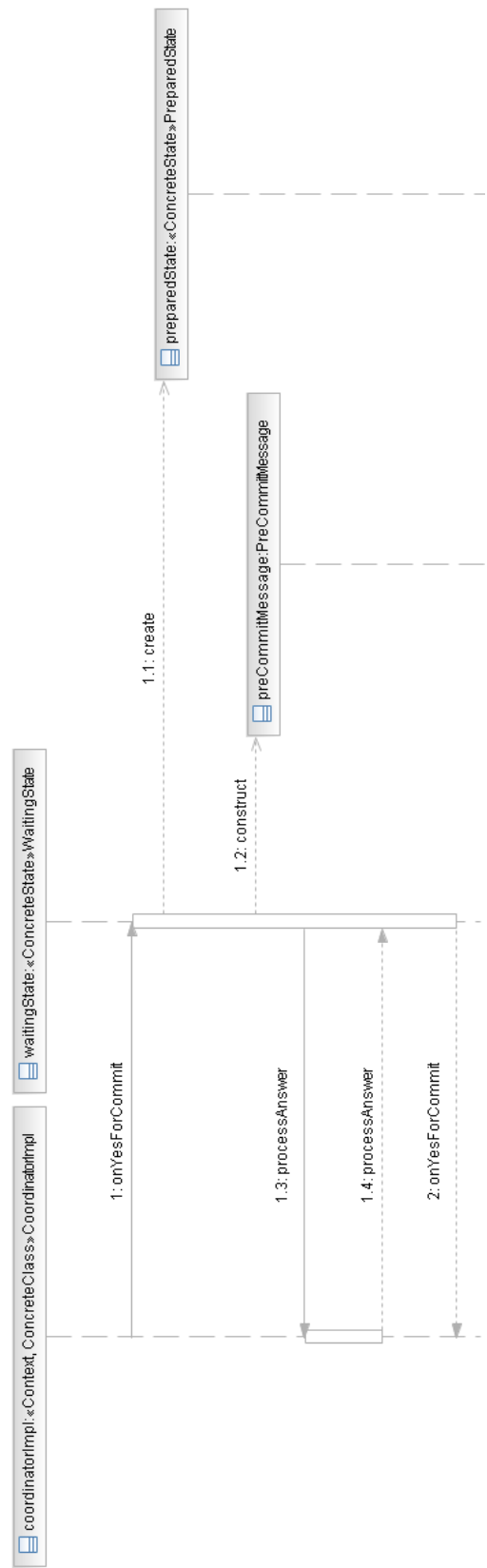




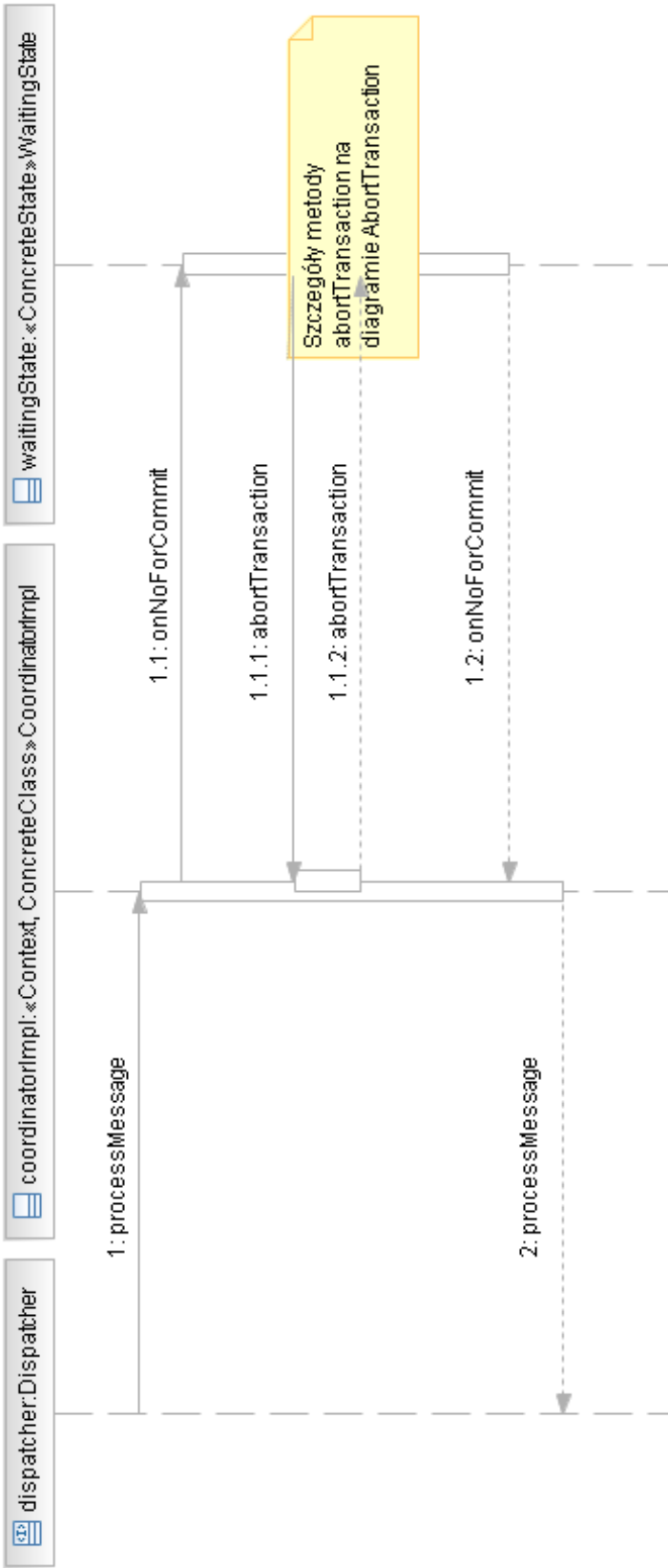
Rysunek 5.4: Obsługa wiadomości CanCommit - cz. 1



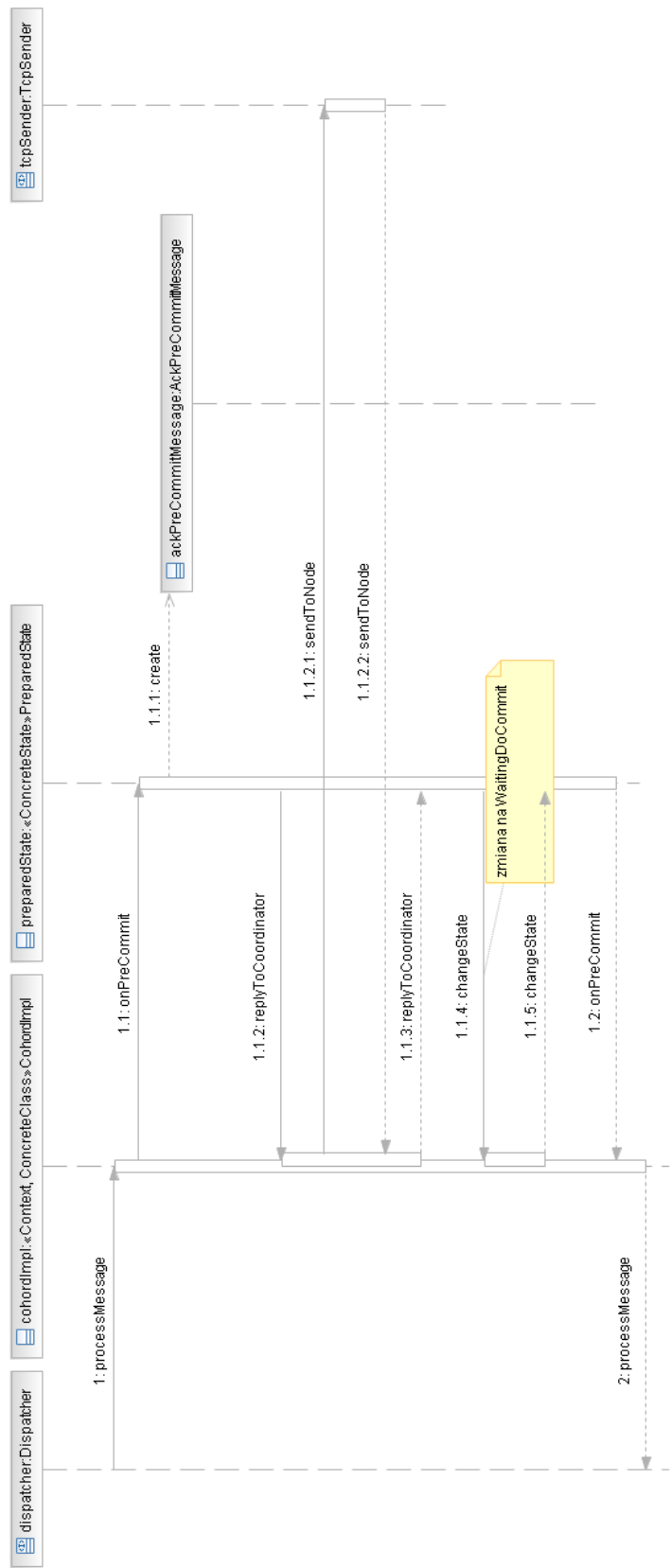
Rysunek 5.5: Obsługa wiadomości CanCommit - cz. 1



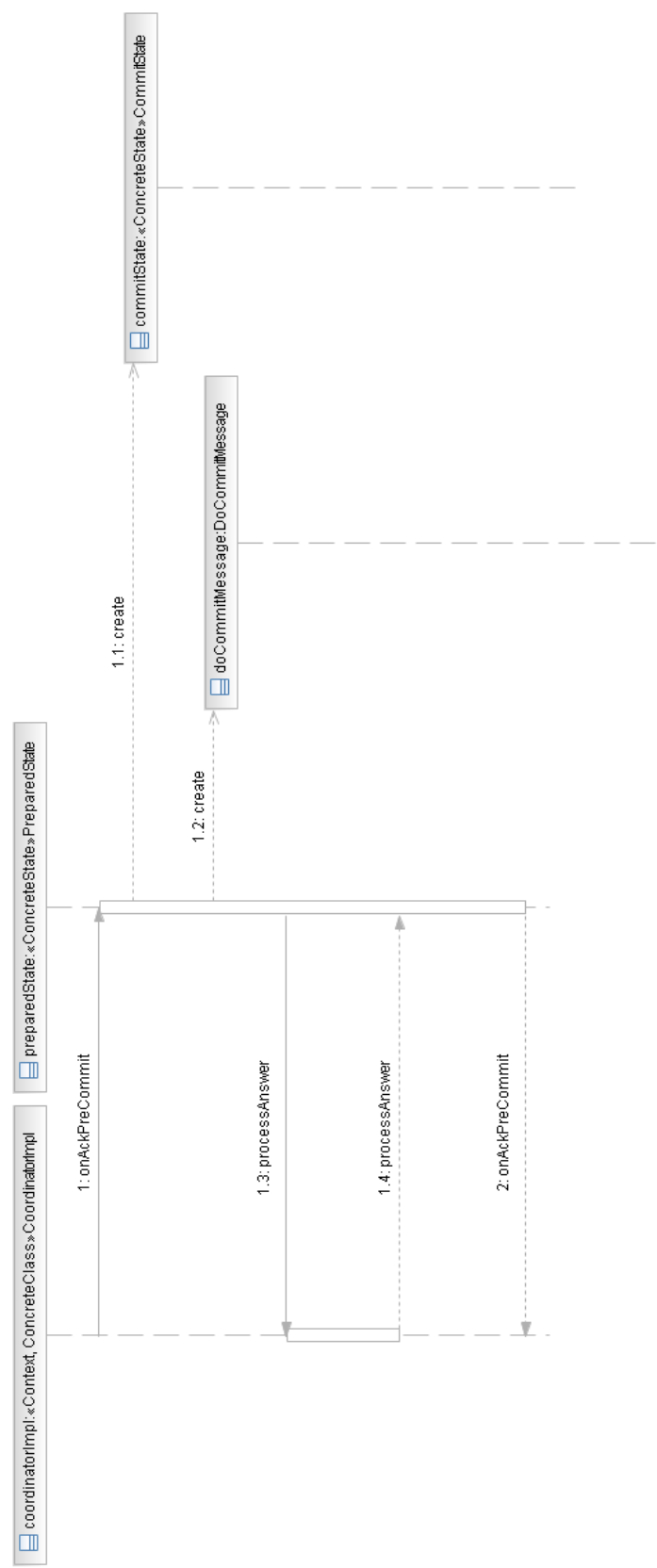
Rysunek 5.6: Obsługa wiadomości YesForCommit



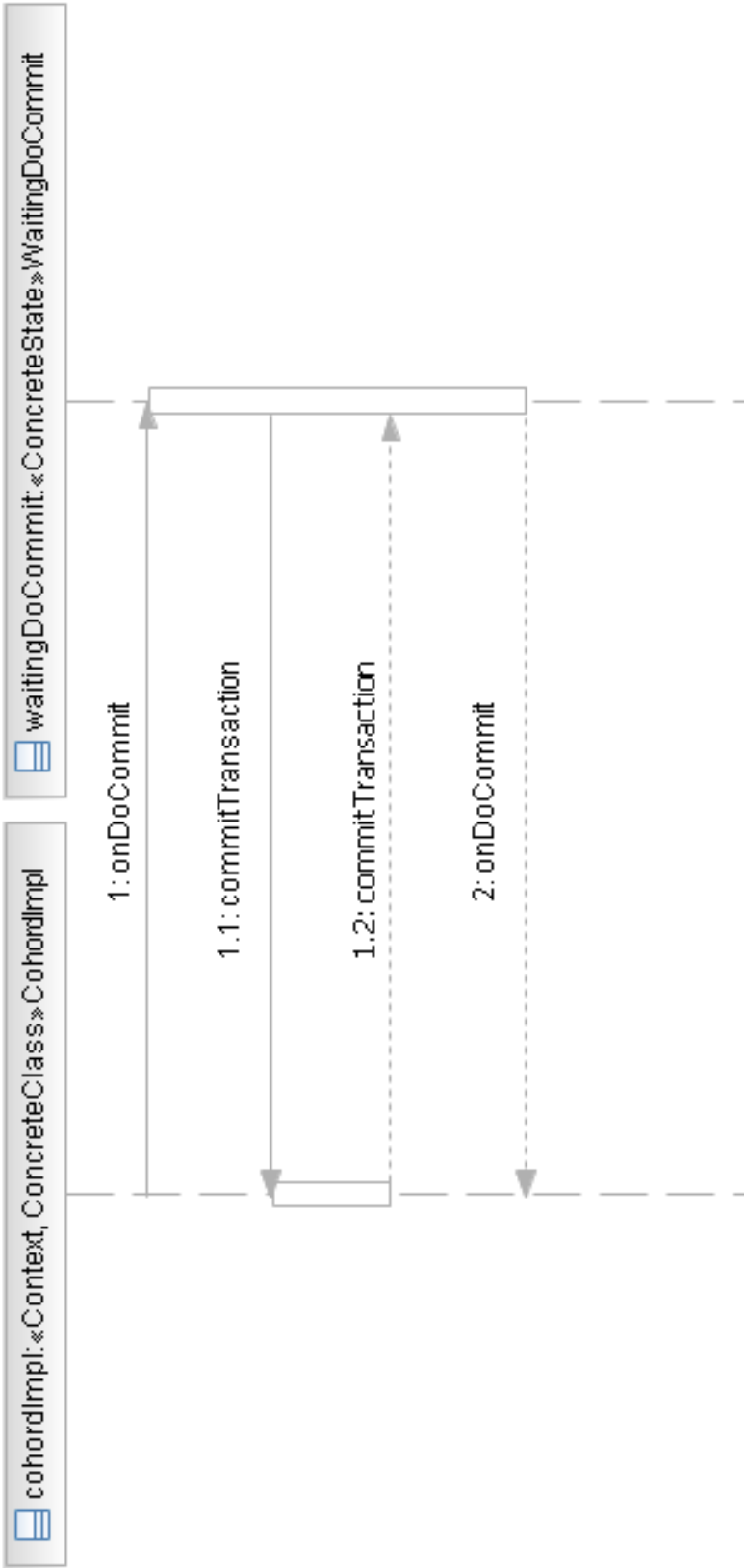
Rysunek 5.7: Obsługa wiadomości `NoForCommit`



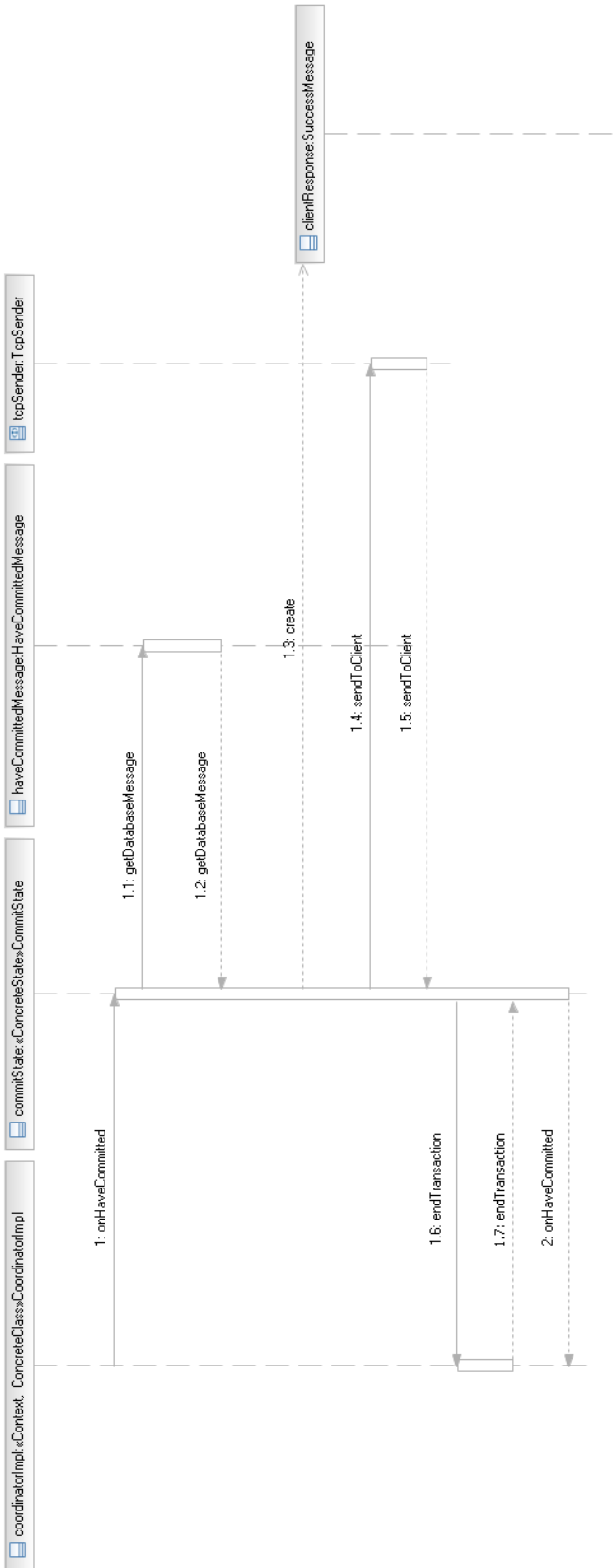
Rysunek 5.8: Obsługa wiadomości PreCommit



Rysunek 5.9: Obsługa wiadomości AckPreCommit



Rysunek 5.10: Obsługa wiadomości DoCommit



Rysunek 5.11: Obsługa wiadomości HaveCommitted