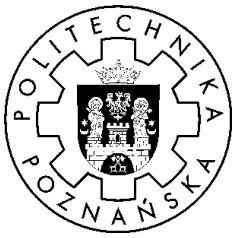
Politechnika Poznańska
Instytut Informatyki



Piotr Wysocki

Projekt i implementacja systemu komunikacji grupowej dla grup dynamicznych na bazie protokołu rozproszonego konsensusu w języku OCaml

Praca magisterska

Promotor: dr hab. inż. Paweł T. Wojciechowski

25 września 2009

Spis treści

Rozdział 1

Wstęp

1.1 Motywacja i cele pracy

Systemy komunikacji grupowej [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] dostarczają abstrakcje programistyczne, które umożliwiają budowę usług sieciowych o zwiększonej odporności na awarie. Zwiększona odporność na awarie jest osiągana przez *replikację* usługi sieciowej na kilku niezależnych serwerach połączonych siecią oraz koordynację interakcji klientów z tymi replikami. Gwarantuje się przy tym utrzymanie stanu spójnego wszystkich replik na serwerach, które nie uległy awarii i wciąż należą do grupy, pod warunkiem, że liczebność grupy nie spadła poniżej określonej liczby. Ogólny model takiej replikacji nazywany jest *zreplikowaną maszyną stanów*. Zastosowanie tego modelu sprawia, że klienci mogą korzystać z usługi sieciowej transparentnie i w sposób ciągły, pomimo awarii pojedynczych serwerów, które udostępniają tę usługę.

Podstawowe abstrakcje komunikacji grupowej, które umożliwiają implementację zreplikowanej maszyny stanów to mechanizmy *rozgłaszania komunikatów*, zapewniające dostarczanie komunikatów do wszystkich serwerów w grupie w sposób uporządkowany, oraz mechanizm *zarządzania przynależnością do grupy*, dostarczający spójny obraz grupy na każdym z serwerów, uwzględniając aktualne informacje na temat serwerów, które dołączyły lub zostały wykluczone z grupy. Niektóre systemy oferują też mechanizm *synchronizacji obrazów*, gwarantujący dostarczanie komunikatów do wszystkich replik w ramach tego samego spójnego obrazu grupy. Z uwagi na postulat braku centralnego koordynatora (stanowiącego niedopuszczalny pojedynczy punkt awarii), realizacja powyższych mechanizmów komunikacji grupowej musi opierać się o mechanizmy *rozproszonego uzgadniania*; zwykle implementuje się w tym celu oddzielną usługę, rozwiązującą tzw. problem *rozproszonego konsensusu* (ang. *distributed consensus*).

Naszym celem jest zaprojektowanie i implementacja *CamlGroups* – modelowego systemu komunikacji grupowej, oferującego minimalne usługi, ale wystarczające do implementacji zreplikowanej maszyny stanów. Architektura systemu jest wzorowana na nowoczesnej modularnej architekturze systemów komunikacji grupowej, zaproponowanej w [11]. Cechą tej architektury jest wyodrębnienie usługi konsensusu (tj. rozproszonego uzgadniania), która staje się podstawą do budowy kolejnych usług komunikacji grupowej. CamlGroups jest implementowany w języku funkcyjnym *Objective Caml* (w skrócie *OCaml*) [12]. Zaletą tego języka jest duża przejrzystość i zwięzłość kodu, co znacznie ułatwia weryfikację poprawności protokołów. Dzięki temu, CamlGroups będzie mógł w przyszłości stanowić wygodny system testowy do projektowania i prototypowania nowych usług i protokołów komunikacji grupowej.

1.2 Struktura pracy

Praca ma następującą strukturę.

Rozdział 2 przedstawia specyfikację podstawowych usług komunikacji grupowej. Rozdział 3 prezentuje architekturę systemu *CamlGroups*. W rozdziale 4 opisano implementację kluczowych części systemu *CamlGroups*, natomiast rozdział 5 opisuje sposób w jaki przeprowadzano testy systemu. W końcu, w rozdziale 6 przedstawiono wnioski wyciągnięte po napisaniu pracy oraz przedstawiono możliwości dalszego rozwoju systemu *CamlGroups*.

Rozdział 2

Semantyka komunikacji grupowej

2.1 Wstęp

Wstępną wersję tego rozdziału zawarto w raporcie technicznym TR-ITSOA-OB2-1-PR-09-02 w ramach projektu IT-SOA realizowanego na Politechnice Poznańskiej. Głównym autorem raportu jest autor tej pracy.

2.1.1 Cele rozdziału

W tym rozdziale przedstawiamy specyfikację podstawowych usług komunikacji grupowej, tj. rozgłaszania komunikatów z różnymi gwarancjami dostarczenia i uporządkowania komunikatów oraz własności detektorów awarii i usługi rozproszonego konsensusu (na podstawie [13, 14]). Następnie prezentujemy architekturę CamlGroups, jako fragment architektury opisanej w [11]. W dodatku pracy (na płycie CD-ROM) zawarto kompletny kod w OCaml protokołów, które zostały zaimplementowane. Wspierają one *grupy* dynamiczne, tj. procesy mogą zarówno opuszczać grupę (mogą ulegać awarii) jak i być dołączane do grupy w trakcie działania systemu.

Do implementacji CamlGroups wybrano protokoły zastosowane w takich systemach komunikacji grupowej, jak Fortika [7] i SAMOA [8]; odpowiednie referencje do publikacji, gdzie zaproponowano i opisano te protokoły, znajdują się w [14] i [15]. Systemy te, zaimplementowane w języku Java, mają zaawansowane cechy, takie jak: możliwość dynamicznej zamiany protokołów (SAMOA), wsparcie dla wielu frameworków protokołowych (Fortika) oraz wsparcie zarówno modelu awarii *crash-stop*, jak również *crash-recovery*. W odróżnieniu od tych systemów, funkcjonalność CamlGroups została ograniczona do minimum. Wspierany jest jedynie model awarii *crash-stop* (zwany też *crash-no-recovery*), tj. nie ma możliwości odtworzenia stanu procesów w grupie, które uległy awarii, a więc cechy charakterystycznej dla modelu awarii *crash-recovery*.

2.1.2 Struktura rozdziału

Rozdział ma następującą strukturę. Rozdział 2.2 zawiera podstawowe definicje i własności komunikacji grupowej. Wyjaśniono różnicę między modelem awarii *crash-stop* i *crash-recovery* oraz między modelem grup statycznych i dynamicznych. Opisano podstawowe kategorie kanałów komunikacyjnych oraz własności algorytmów rozgłaszania.

W rozdziale 2.3 przedstawiono specyfikację (własności) podstawowych konstrukcji komunikacji grupowej, tj. omówiono: podstawowe i jednolite rozgłaszanie niezawodne, rozgłaszanie niezawodne

z przyczynowym uporządkowaniem wiadomości, rozgłaszanie z globalnym uporządkowaniem wiadomości (tzw. rozgłaszanie atomowe), a także rozgłaszanie generyczne. Następnie krótko scharakteryzowano detektory awarii oraz zdefiniowano usługę rozproszonego konsensusu.

W rozdziale 3, przedstawiono modularną architekturę budowanego systemu, wymieniając znaczenie poszczególnych warstw protokołów w stosie protokołów oraz krótko wyjaśniając mechanizm komunikacji w stosie protokołów.

W dodatku znajduje się kod źródłowy protokołów do tej pory zaimplementowanych.

2.2 Podstawowe definicje

Poniżej podano definicje podstawowych pojęć i własności związanych z komunikacją grupową.

2.2.1 Modele awarii: *crash-stop* i *crash-recovery*

W naszym systemie przyjmujemy model awarii *crash-stop*. Oznacza to, że procesy mogą ulegać awarii. Możliwe jest wykrywanie awarii procesów. W zależności od zastosowanego detektora awarii oraz stopnia synchronizacji systemu, detekcja awarii jest mniej lub bardziej dokładna. Po awarii proces może wznowić przetwarzanie, ale do grupy może dołączyć – w przypadku grup dynamicznych – jedynie jako całkowicie nowy proces. Nie jest więc obsługiwane odtwarzanie stanu takich procesów po awarii.

Gdy w danym momencie jakiś proces p chce rozgłosić jakąś wiadomość m, mechanizm komunikacji grupowej musi znać inne procesy biorące udział w komunikacji grupowej. Możliwe są różne topologie sieci, jednak dla naszych potrzeb przyjmiemy topologię połączeniową *każdy z każdym*. Czyli każdy proces zna wszystkie inne procesy, tj. potrafi do nich wysłać wiadomość bezpośrednio.

Model awarii *crash-recovery* umożliwia ponowne przyłączenie się do grupy tych procesów, które wcześniej uległy awarii. W takim przypadku uruchamiana jest procedura *odtwarzania stanu przetwarzania*, która na podstawie zbiorów *punktów kontrolnych* (ang. *check-points*) przywraca spójny stan globalny dla wszystkich procesów w grupie.

2.2.2 Modele systemu: grupy statyczne i dynamiczne

W najprostszym przypadku zakłada się model *grup statycznych*, tj. grupa procesów biorących udział w komunikacji grupowej nie zwiększa się w trakcie działania systemu; procesy mogą być jedynie odłączane z grupy, np. jeśli ulegną awarii.

W przypadku modelu *grup dynamicznych*, procesy mogą być dynamicznie dołączane lub odłączane do/z grupy procesów. Dzięki temu procesy, które uległy awarii, mogą być zastępowane przez nowe procesy bez zatrzymywania całego systemu. Zachowane są przy tym wszystkie gwarancje poprawnego rozgłaszania komunikatów do wszystkich procesów, które były znane procesowi, który rozpoczął rozgłaszanie. W przypadku modelu awarii *crash-recovery*, przy dołączaniu procesu odtwarzanego po awarii, możliwe jest także uspójnienie stanu systemu do stanu sprzed awarii procesu.

Do zarządzania grupą, tj. dynamicznego dołączania i odłączania procesów, służy *usługa członkostwa* (ang. *membership service*). Usługa ta dostarcza każdemu z procesów w grupie spójny obraz tego, które procesy są członkami grupy. Uwzględniając są przy tym aktualne informacje na temat procesów, które dołączyły lub zostały wykluczone z grupy.

2.2.3 Kategorie kanałów komunikacyjnych

Zanim przejdziemy do opisu semantyki mechanizmów rozgłaszania, zdefiniujmy najpierw dwie podstawowe kategorie kanałów komunikacyjnych:

Kanały niezawodne
: Powiemy, że kanał komunikacyjny jest *kanałem niezawodnym* (ang. *reliable channel*), jeśli wysłanie wiadomości m przez proces p (poprawny lub nie) do poprawnego procesu q implikuje fakt, że proces q ostatecznie odbierze wiadomość m.

Kanały quasi-niezawodne
: Powiemy, że kanał jest *kanałem quasi-niezawodnym* (ang. *quasi-reliable channel*), jeśli wysłanie wiadomości m przez poprawny proces p do poprawnego procesu q implikuje fakt, że proces q ostatecznie odbierze wiadomość m.

gdzie proces *poprawny* (ang. *correct*) to proces, który nie ulega awarii w trakcie całego swojego życia. Natomiast o procesie, który ulega awarii mówi się, że jest to proces *niepoprawny* lub *awaryjny* (ang. *incorrect* lub *faulty*).

2.2.4 Własności algorytmów rozgłaszania

Przez *rozgłaszanie wiadomości* rozumiemy mechanizm komunikacyjny, za pomocą którego proces może wysłać wiadomość do innych procesów należących do tej samej grupy procesów. Rozróżnia się wiele takich mechanizmów, które oferują różne gwarancje jeśli chodzi o tolerancję awarii w systemie oraz uporządkowanie dostarczanych wiadomości.

Algorytmy zastosowane do implementacji mechanizmów rozgłaszania spełniają pierwsze dwie własności oraz jedną z dwóch ostatnich:

Ważność
: Powiemy, że rozgłaszanie spełnia własność *ważności* (ang. *validity*), jeśli proces wywołujący mechanizm rozpraszania dla danej wiadomości ostatecznie dostarczy tę wiadomość.

Integralność
: Powiemy, że rozgłaszanie spełnia własność *integralności* (ang. *integrity*), jeśli każdą wiadomość m, każdy poprawny proces dostarcza co najwyżej raz ale tylko i wyłącznie wtedy, gdy wiadomość m została wcześniej rozgłoszona.

Zgodność
: Powiemy, że rozgłaszanie spełnia własność *zgodności* (ang. *agreement*), jeśli dostarczenie wiadomości m pewnemu poprawnemu procesowi implikuje dostarczenie wiadomości m wszystkim poprawnym procesom.

Jednolita zgodność
: *Jednolicie zgodnym* (ang. *uniform agreement*) rozgłaszaniem nazywamy takie rozgłaszanie, które gwarantuje dostarczenie wszystkim poprawnym procesom każdej wiadomości, która została dostarczona do co najmniej jednego procesu (poprawnego lub nie).

Dodatkowo mogą być spełnione dowolne własności z poniższej listy:

Uporządkowanie wiadomości zgodne z porządkiem kolejkowym (FIFO)
: Powiemy, że rozgłaszanie *porządkuje wiadomości zgodnie z porządkiem kolejkowym (FIFO)*

(ang. *First-In-First-Out (FIFO) order*), jeśli rozgłoszenie wiadomości m_1 przed wiadomością m_2 przez proces P_i implikuje to, że żaden proces P_j nie odbierze wiadomości m_2, jeśli nie odebrał wcześniej wiadomości m_1.

Przyczynowe uporządkowanie wiadomości

Powiemy, że rozgłaszanie *porządkuje wiadomości przyczynowo* (ang. *causal order*), jeśli wiadomości przyczynowo zależne będą dostarczone do odbioru zgodnie z relacją poprzedzania (ang. *happens before*).

Globalne uporządkowanie wiadomości

Powiemy, że rozgłaszanie *porządkuje wiadomości globalnie* (ang. *total order*), jeśli zapewniona jest taka sama kolejność dostarczania wiadomości do wszystkich poprawnych procesów.

Ogólne uporządkowanie wiadomości

Niech będzie dana relacja konfliktów $C : M \times M \rightarrow Boolean$, gdzie M to zbiór rozgłaszanych wiadomości. Dwie wiadomości znajdą się w relacji C wtedy i tylko wtedy, gdy są w *konflikcie*. Znaczenie konfliktu jest następujące: Jeśli dwie wiadomości są w konflikcie, to muszą być dostarczone do wszystkich procesów w tej samej kolejności. W przeciwnym razie mogą być dostarczane w dowolnej kolejności.

Powiemy, że rozgłaszanie *porządkuje wiadomości generycznie* (ang. *generalized order*), jeśli dla wszystkich par wiadomości będących w konflikcie, wiadomości składające się na te pary zostaną dostarczone do wszystkich poprawnych procesów w tej samej kolejności.

2.3 Podstawowe konstrukcje komunikacji grupowej

W tym rozdziale zdefiniowano podstawowe konstrukcje komunikacji grupowej. Podano własności, jakie spełniają te konstrukcje.

2.3.1 Podstawowe rozgłaszanie niezawodne

Podstawowym rozgłaszaniem niezawodnymi (ang. best effort broadcast) nazywamy taki mechanizm rozgłaszania, który posiada następujące własności:

- ważność (ang. validity),
- integralność (ang. integrity).

2.3.2 Rozgłaszanie niezawodne

(Zgodnym) rozgłaszaniem (ang. *(regular) reliable broadcast*) nazywamy taki mechanizm rozgłaszania niezawodnego, który posiada następujące własności:

- ważność (ang. validity),
- integralność (ang. integrity),
- zgodność (ang. agreement).

2.3.3 Jednolite rozgłaszanie niezawodne

Jednolitym rozgłaszaniem (ang. *uniform reliable broadcast*) nazywamy taki mechanizm rozgłaszania niezawodnego, który posiada następujące własności:

- ważność (ang. validity),
- integralność (ang. integrity),
- jednolita zgodność (ang. uniform agreement).

2.3.4 Rozgłaszanie z przyczynowym uporządkowaniem wiadomości

Rozgłaszaniem z przyczynowym uporządkowaniem wiadomości (ang. *causal order broadcast*) nazywamy taki mechanizm rozgłaszania niezawodnego, który posiada następujące własności:

- ważność (ang. validity),
- integralność (ang. integrity),
- zgodność (ang. agreement),
- przyczynowe uporządkowanie wiadomości (ang. causal delivery).

2.3.5 Rozgłaszanie z globalnym uporządkowaniem wiadomości

Rozgłaszaniem z globalnym uporządkowaniem wiadomości lub w skrócie *rozgłaszaniem atomowym* (ang. *total order broadcast* lub *atomic broadcast*) nazywamy taki mechanizm rozgłaszania niezawodnego, który posiada następujące własności:

- ważność (ang. validity),
- integralność (ang. integrity),
- zgodność (ang. agreement),
- globalne uporządkowanie wiadomości (ang. total order).

2.3.6 Rozgłaszanie generyczne

Rozgłaszaniem generycznym (ang. *generic broadcast*) nazywamy taki mechanizm rozgłaszania niezawodnego, który posiada następujące własności:

- ważność (ang. validity),
- integralność (ang. integrity),
- zgodność (ang. agreement),
- generyczne uporządkowanie wiadomości (ang. generalized order).

Zauważmy, że gdy relacja konfliktów C (zdefiniowana w rozdziale 2.2.4) jest pusta, to rozgłaszanie generyczne sprowadza się do rozgłaszania niezawodnego. Z drugiej strony, gdy do relacji C należy każda para rozgłaszanych wiadomości, to rozgłaszanie generyczne jest równoważne rozgłaszaniu atomowemu.

2.3.7 Detektory awarii

Każdy proces p_i ma dostęp do lokalnego modułu *detektora awarii* (ang. *failure detector*), oznaczanego FD_i. Każdy lokalny moduł FD_i monitoruje procesy w systemie i zarządza listą procesów podejrzewanych o awarię. Moduł detektora błędu może zostać odpytany przez swój proces, zwracając w odpowiedzi listę procesów, które uznaje on za działające w danej chwili, przy czym:

- każdy lokalny detektor awarii może popełnić błąd poprzez dodanie do listy podejrzanych procesów proces, który nadal działa. Innymi słowy, detektory awarii nie są wiarygodne;
- każdy lokalny detektor awarii może zmienić zdanie i usunąć jakiś proces z listy podejrzanych procesów, jeśli uzna, że podejrzewanie tego procesu było błędem;
- w danym czasie detektory awarii dwóch różnych procesów mogą mieć różne listy podejrzewanych procesów.

Detektory awarii charakteryzowane są przez własność *kompletności* (ang. *completeness*), która definiuje ograniczenia w odniesieniu do procesów które uległy awarii, oraz *dokładność (ang. accuracy)*, która definiuje ograniczenia w odniesieniu do poprawnych procesów.

Wyróżnia się następujące typy własności *kompletności* (ang. *completeness*) detektorów błędów:

Silna kompletność (ang. *strong completeness*)
: oznacza, że ostatecznie każdy proces, który ulega (uległ?) awarii jest trwale podejrzewany przez *każdy* poprawny proces;

Słaba kompletność (ang. *weak completeness*)
: oznacza, że ostatecznie każdy poprawny, który ulega (uległ?) awarii jest trwale podejrzewany przez *pewien* poprawny proces.

Wyróżnia się następujące typy własności *dokładności* (ang. *accuracy*) detektorów błędów:

Silna dokładność (ang. *strong accuracy*)
: oznacza, że żaden proces nie jest podejrzewany zanim nie ulegnie awarii;

Słaba dokładność (ang. *weak accuracy*)
: oznacza, że pewien poprawny proces nigdy nie jest podejrzewany, jako ten który uległ awarii;

Ostatecznie silna dokładność (ang. *eventual strong accuracy*)
: oznacza, że istnieje czas, po którym poprawne procesy nie są podejrzewane przez jakikolwiek poprawny proces;

Ostatecznie słaba dokładność (ang. *eventual strong accuracy*)
: oznacza, że istnieje czas, po którym pewien poprawny proces nie jest nigdy podejrzewany, jako ten który uległ awarii;

Na podstawie powyższych własności, Chandra i Toueg [16]) wyróżnili klasy detektorów awarii; przykładowe klasy detektorów awarii to:

Perfekcyjny detektor awarii (P)
: spełnia własności silnej kompletności oraz silnej dokładności,

Ostatecznie perfekcyjny detektor awarii ($\Diamond P$)
: spełnia własności silnej kompletności oraz ostatecznie silnej dokładności,

Ostatecznie silny detektor awarii ($\Diamond S$)
: spełnia własności silnej kompletności oraz ostatecznie słabej dokładności.

2.3.8 Rozproszony konsensus

Usługa *rozproszonego konsensusu* (ang. *distributed consensus*), rozwiązuje problem rozproszonego konsensusu, zdefiniowany następująco:

Mamy dane n procesów, każdy proces p_i posiada swoją wartość początkową v_i. Cały zbiór procesów ma za zadanie *zgodzić się* (podjąć decyzję) na pewną wspólną wartość v, która należy do zbioru $V = \{v_i : 1 <= i <= n\}$, przy czym muszą być spełnione następujące trzy własności:

- **zakończenie** – każdy poprawny proces ostatecznie podejmie decyzję;
- **ważność** – jeśli jakiś proces podejmie decyzję v, to v należy do zbioru V;
- **jednolita zgodność** – dwa procesy nie mogą podjąć dwóch różnych decyzji.

W klasycznym środowisku asynchronicznym problem rozproszonego konsensusu jest nierozwiązywalny [17]. Jednakże rozszerzając środowisko asynchroniczne o detektory awarii, problem można rozwiązać przy użyciu detektora awarii klasy $\Diamond P$ [16].

Rozdział 3

Architektura systemu komunikacji grupowej

W tym rozdziale opisana zostanie architektura modularnego systemu komunikacji grupowej, tj. wymienione zostaną poszczególne protokoły w stosie protokołów oraz zdefiniowany zostanie interfejs komunikacyjny między protokołami. Zakłada się *model warstwowy* stosu protokołów, który nie tylko ułatwia zrozumienie zależności między protokołami, ale także ułatwia implementację systemu. W modelu tym protokoły tworzą stos protokołów w którym protokoły komunikują się miedzy sobą korzystając ze zunifikowanego interfejsu. Kolejne warstwy protokołów dodawane do stosu od dołu do góry, mogą korzystać jedynie z warstw niższych, przy czym implementacja warstw niższych nie musi być znana, a jedynie semantyka usługi dostarczanej przez protokół.

Na rysunkach 3.1 i 3.2 przedstawiono stosy protokołów komunikacji grupowej, odpowiednio dla modelu grup statycznych i modelu grup dynamicznych. W oparciu o tę architekturę zamierzamy budować nasz system komunikacji grupowej *CamlGroups*, implementowany w języku OCaml. Jak widać, obie wersje stosu różnią się jedynie tym, że w modelu grup dynamicznych mamy dodatkowo warstwę protokołów *Group Membership* i *Generic Broadcast*. Jest to uproszczona wersja architektury zaproponowanej w [11] i zaimplementowanej w systemach Fortika [7, 14] oraz SAMOA [8, 15].

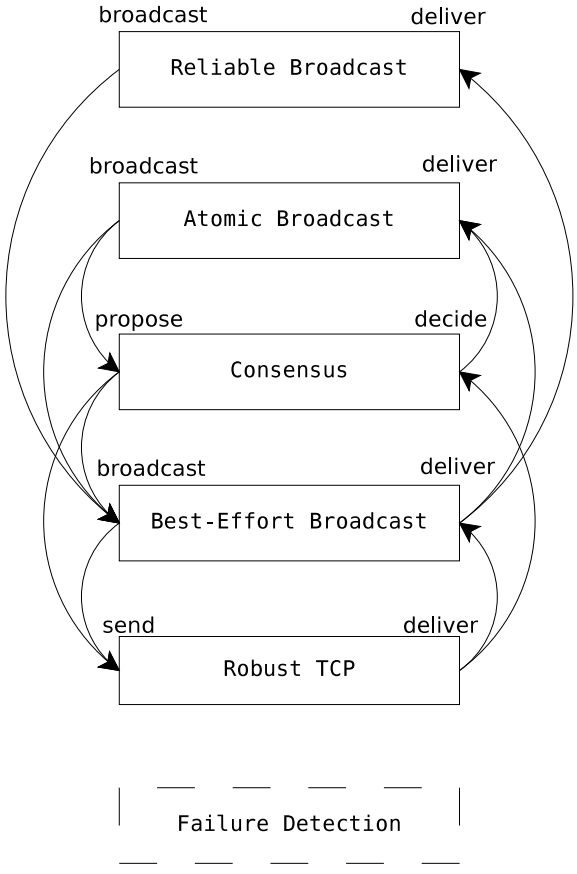
Poniżej krótko omówiono kolejne warstwy tej architektury, zaczynając od najniższej warstwy komunikacyjnej powyżej warstwy transportowej.

3.1 Fault-Tolerant TCP i Failure Detector

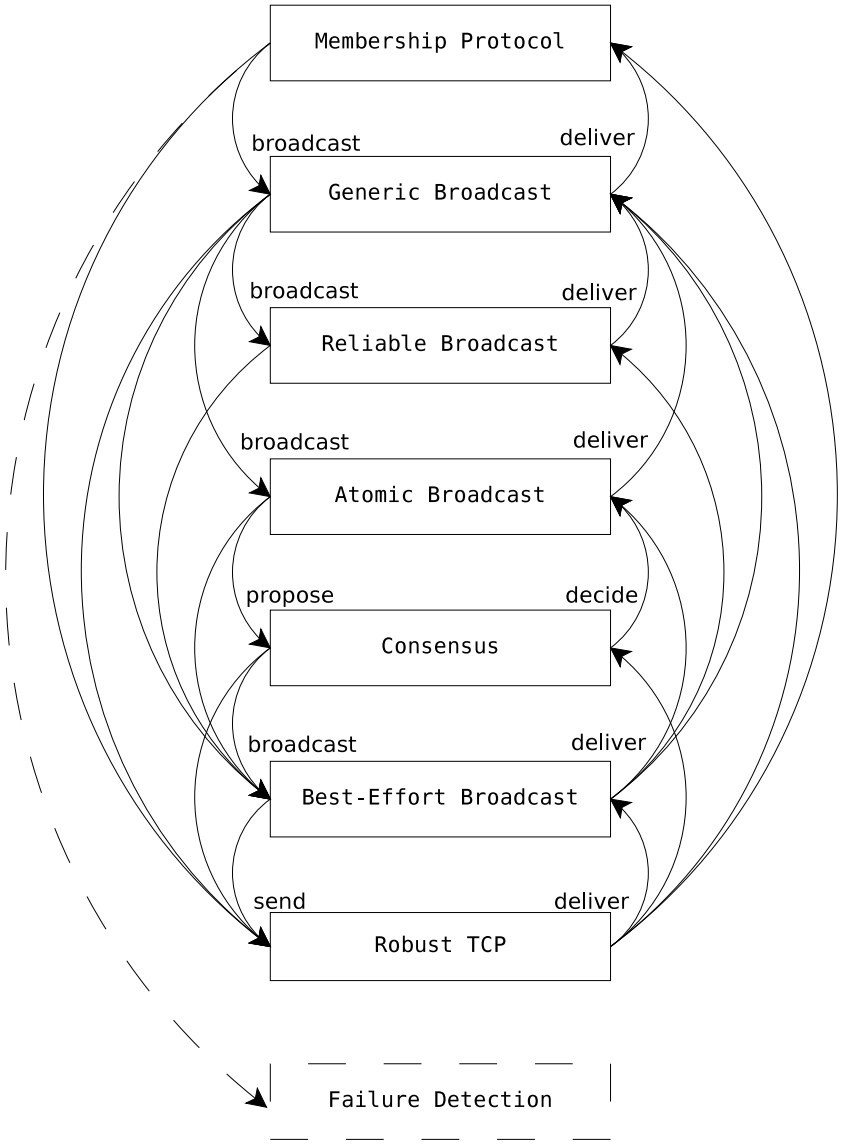
Powyżej standardowych protokołów TCP lub UDP konieczny jest protokół, który zajmie się komunikacją między dwoma procesami z dodatkowymi gwarancjami odporności na błędy łącz komunikacyjnych, których nie zapewnia ani protokół TCP, ani tym bardziej UDP. Przykładowo, UDP może gubić pakiety, zaś protokół TCP nie dostarczy komunikatu, gdy nie uda się nawiązać połączenia bądź już nawiązane połączenie zostaje rozłączone z uwagi na awarię łącz. Protokół *Fault-Tolerant TCP* radzi sobie z przejściowymi awariami łączy.

Protokół Fault-tolerant TCP oraz wiele innych protokołów komunikacji grupowej korzysta z protokołu *Failure Detector*, która implementuje usługę detektora błędów.

Rysunek 3.1: Architektura dla grup statycznych



Rysunek 3.2: Architektura dla grup dynamicznych



3.2 Distributed Consensus

Kolejną warstwą w stosie protokołów komunikacji grupowej jest usługa rozproszonego konsensusu. Do jej implementacji zastosowano protokół *Distributed Consensus*, opracowany na podstawie algorytmu zaproponowanego przez Chandra i Toueg'a [16]. Protokół Distributed Consensus umożliwia realizację szeregu dalszych usług komunikacji grupowej, w szczególności rozgłaszania atomowego i generycznego.

Protokół korzysta z Reliable Broadcast, Fault Tolerant TCP i detektora błędów – każda wiadomość wysłana przez Distributed Consensus do jakiegoś procesu, np. proponowana wartość lub decyzja, trafia najpierw do Fault-Tolerant TCP lub Reliable Broadcast, które to protokoły wysyłają lub rozgłaszają tę wiadomość dalej. Dzięki temu mamy pewność, że wiadomość dojdzie do wszystkich poprawnych procesów, zgodnie z semantyką i gwarancjami tych usług komunikacyjnych.

Przy wysyłaniu wiadomości do Fault-Tolerant TCP lub Reliable Broadcast przekazywana jest również informacja o tym, w jaki sposób Distributed Consensus ma odebrać wiadomość zwrotną od tych protokołów. Model tej komunikacji, który łącznie z interfejsem programisty definiuje tzw. *framework protokołowy*, zostanie opisany w rozdziale *komunikacja*.

3.3 Atomic Broadcast i Generic Broadcast

Protokół *Atomic Broadcast* implementuje usługę rozgłaszania z globalnym uporządkowanie wiadomości, na podstawie algorytmu w [16]. Aby zagwarantować identyczną kolejność dostarczania rozgłoszonych wiadomości do wszystkich procesów poprawnych, protokół korzysta z usługi rozproszonego konsensusu w celu uzgodnienia spójnej kolejności komunikatów.

W celu implementacji usługi przynależności, niezbędnej w przypadku implementacji wsparcia dla grup dynamicznych, zastosowana architektura zakłada wykorzystanie protokołu *Generic Broadcast*, który implementuje rozgłaszanie generyczne. Umożliwia ono niezawodne dostarczanie wiadomości w grupie procesów z własnością globalnego uporządkowania wiadomości, za wyłączeniem wiadomości (np. kontrolnych), które mają być dostarczone do tych samych procesów, ale nie powinny być porządkowane.

3.4 Reliable Broadcast

Protokół *Reliable Broadcast* implementuje usługę rozgłaszania niezawodnego, korzystając z protokołu Fault-Tolerant TCP, stanowiącego warstwę niższą w stosie protokołów. Protokół Reliable Broadcast ma następujące własności: ważność (ang. validity), integralność (ang. integrity), oraz zgodność (ang. agreement), co łącznie gwarantuje pewność dostarczenia wiadomości do wszystkich poprawnych procesów, o ile do jakiegoś poprawnego procesu rozgłaszana wiadomość dojdzie (własność zgodności). Protokół nie gwarantuje żadnej kolejności dostarczania rozgłoszonych wiadomości do procesów. Dlatego więc potrzebne są kolejne warstwy protokołów.

Chcąc wysłać wiadomość, protokół Reliable Broadcast korzysta z protokołu Fault-Tolerant TCP, tj. zarówno komunikaty wychodzące jak i przychodzące do procesu, są przetwarzane przez obie warstwy.

3.5 Group Membership

Protokół *Group Membership* implementuje usługę przynależności do grup, która jest odpowiedzialna za ciągłe dostarczanie spójnego obrazu grupy na każdym z serwerów, uwzględniając aktualne informacje na temat serwerów, które dołączyły lub zostały wykluczone z grupy. Do implementacji tego protokołu wymagana jest usługa rozgłaszania atomowego lub usługa rozgłaszania generycznego.

3.6 Komunikacja w stosie protokołów

Protokoły w stosie protokołów komunikują się korzystając z ujednoliconego *interfejsu programisty*, który dostarcza funkcje wysyłania wiadomości do niższej warstwy w stosie protokołów oraz odbierania wiadomości zwrotnych z niższej warstwy. Wiadomości składają się z listy nagłówków oraz obiektu wiadomości, gdzie *lista nagłówków* odpowiada liście usług, które mają przetworzyć tę wiadomość. Protokół przetwarzający daną wiadomość (komunikat) ma dostęp jedynie do informacji zawartych w nagłówku, który znajduje się w głowie listy. *Obiekt wiadomości* zawiera natomiast wyłącznie dane wysyłane i odbierane przez aplikację powyżej systemu komunikacji grupowej.

W przypadku wiadomości wychodzących od aplikacji do zdalnego procesu, kolejne warstwy protokołów doklejają swoje nagłówki do listy nagłówków wiadomości. Każdy kolejny nagłówek na liście nagłówków wiadomości, doklejony przez protokół P_i, zawiera informacje konieczne do przetworzenia tej wiadomości przez protokół P_i w zdalnym stosie protokołów. Aby móc odebrać wiadomość zwrotną, tj. odpowiedź na wysyłaną wiadomość, należy umieścić w nagłówku referencję do protokołu, który ma przetworzyć dany nagłówek wiadomości.

Oprócz komunikacji między procesami (na odpowiednim poziomie abstrakcji) oraz między warstwami zależnych protokołów realizowanej poprzez nagłówki przesyłanych wiadomości, dopuszcza się także bezpośrednią komunikację między usługami na zasadzie "sprzężenia zwrotnego", np. protokół Fault-Tolerant TCP może zaprzestać wysyłania wiadomości do procesu p, jeśli uzyska on informację z usługi Group Membership, z których wynika, że p nie jest już członkiem grupy. Tego typu komunikację w stosie protokołów należy jednak zminimalizować, gdyż może ona utrudnić propozycję ujednoliconego interfejsu.

Do implementacji systemu używany jest język OCaml z rodziny języków ML, których cechy takie jak: typy abstrakcyjne, konstrukcja *porównania do wzorca* (ang. *pattern matching*), interfejsy, moduły, *funkcje wysokiego poziomu* (ang. *higher-order functions*) oraz możliwość komunikacji przez sieć *domknięć funkcji* (ang. *function closures*) znacznie ułatwiają implementację modularnych stosów protokołów (zob. także [18]). Jednym z drugorzędnych celów autora niniejszej pracy była chęć sprawdzenia się w programowaniu w nowym środowisku jakim dla autora jest język OCaml.

Rozdział 4

Implementacja

W tym rozdziale opisana zostanie implementacja systemu komunikacji grupowej dla grup dynamicznych w języku OCaml.

Podstawą implementacji systemu była istniejąca biblioteka komunikacji grupowej dla grup statycznych w języku OCaml. W sferze implementacji zadaniem autora tej pracy było:

- uporządkowanie kodu źródłowego biblioteki
- naprawienie istniejących błędów w bibliotece
- zaprojektowanie nowej architektury systemu komunikacji grupowej uwzględniającego implementację biblioteki komunikacji grupowej dla grup statycznych
- implementacja nowej architektury systemu komunikacji grupowej zakładającej dynamiczne grupy procesów
- integracja biblioteki komunikacji grupowej dla grup statycznych z nową architekturą systemu
- implementacja nowych warstw systemu:
  - Generic Broadcast
  - Membership Protocol

W kolejnych podrozdziałach zostaną opisane:

- wykorzystana biblioteka komunikacji grupowej dla grup statycznych w języku OCaml
- nowa architektura systemu komunikacji grupowej w postaci *framework'u* protokołowego
- implementacja warstw Generic Broadcast oraz Membership Protocol

4.1 Biblioteka komunikacji grupowej dla grup statycznych w języku OCaml

Autor niniejszej pracy w celu konstrukcji systemu komunikacji grupowej dla grup dynamicznych wykorzystał kod źródłowy istniejącej biblioteki komunikacji grupowej dla grup statycznych.

Biblioteka składała się z następujących modułów głównych:

- Robust TCP (rtcp)

- Distributed Consensus (consensus)
- Atomic Broadcast (abcast)

oraz z kilku modułów pomocniczych i testowych.

Moduły główne realizują odpowiednie zadania opisane w rozdziale 2. Były jednak ze sobą dosyć ściśle powiązane, a interfejs pośredniczący między modułami nie był jednolity.

Jednym z modułów pomocniczych jest moduł *main*. Moduł ten odpowiednio inicjalizował pozostałe moduły oraz wykonywał testowe rozgłaszanie. Moduły *rtcp* oraz *abcast* były inicjalizowane z poziomu modułu *main*, natomiast moduł *consensus* był inicjalizowany z poziomu modułu *abcast*. Tylko moduł *abcast* miał dostęp do modułu *consensus*. W systemie komunikacji grupowej dla grup dynamicznych, który w założeniu ma być modularny, takie rozwiązanie nie wchodzi w grę. Stąd wynikła potrzeba zmiany powiązań między modułami i budowa nowego *framework'u*, o którym będzie mowa w następnym podrozdziale.

4.2 *Framework* protokołowy

Głównym i najważniejszym elementem skonstruowanego systemu jest *framework* protokołowy. Jest to moduł OCaml'owy enkapsulujący system komunikacji grupowej. Moduł ten udostępnia interfejs umożliwiający niezawodne rozgłaszanie wiadomości poprzez mechanizmy *Reliable Broadcast*, *Atomic Broadcast* oraz *Generic Broadcast* (patrz: rozdział 2). Za pomocą interfejsu do tego modułu można również dołączyć proces do grupy (usługa *Membership Service*).

4.2.1 OCaml'owy interfejs modułu

Kod źródłowy interfejsu modułu znajduje się na stronie 19. Interfejs modułu *framework* wygląda następująco:

Został zdefiniowany typ `'a Ar_framework.t_instance`[1], który jest typem `Ar_framework.t_instance` przyjmującym dowolny argument (*'a*). Zmienna tego typu reprezentuje pojedynczą instancję *framework'u*.

Ponadto zdefiniowano następujące funkcje:

initialize — inicjalizuje *framework*, zwraca zmienną typu `'a Ar_framework.t_instance`. Argumentami tej funkcji są: numer portu nasłuchu warstwy połączeń P2P, funkcja typu *callback* wywoływana w momencie dostarczenia wiadomości oraz relacja konfliktów (patrz rozdział 2.2.4 na stronie 8).

start — rozpoczyna działanie instancji *framework'u*.

bcast — rozgłasza daną wiadomość używając podanej konkretnej instancji *framework'u*.

mp_join — dołącza proces wywołujący tę funkcję do podanej grupy procesów.

W dalszej części tej pracy za *moduł framework* będzie się uważać *instancję typu* `Ar_framework.t_instance`.

4.2.1.1 *Workflow* rozgłaszania wiadomości

Zakładając poprawne zainicjalizowanie i wystartowanie instancji modułu można jej użyć do rozgłaszania wiadomości. Funkcja *bcast* jako argumenty przyjmuje instancję modułu *framework* oraz

[1] Przedrostek `Ar_framework` oznacza nazwę modułu, do którego należy zdefiniowana nazwa.

Rysunek 4.1: Interfejs modułu: ar_framework.mli

```
open Globals

type 'a t_instance

(** Initializes the framework
  @param udp_port : int - port for listening
  @param callback : ('a -> unit) - callback for incoming messages
  @param conflict_relation : ('a -> 'a -> bool) - messages' conflict relation
  **)
val initialize: int -> ('a -> unit) -> ('a -> 'a -> bool) -> 'a t_instance

(** Broadcasts a message.
  @param inst : 'a t_instance - framework instance
  @param obj : 'a - object containing the message
  **)
val bcast: 'a t_instance -> 'a -> unit

(** Starts the framework
  @param inst : 'a t_instance - framework instance
  **)
val start: 'a t_instance -> unit

(** Joins current process to the group identified by given process
  @param inst : 'a t_instance - framework instance
  @param process : process_id - process which is a member of the process group
  **)
val mp_join: 'a t_instance -> process_id -> unit

val connections_string : 'a t_instance -> string
```

wiadomość odpowiedniego typu (dla danego typu instancji). Przykładowo, dla instancji *framework'u* typu `int Ar_framework.t_instance` odpowiednim typem wiadomości będzie wartość typu `int`.

Moduł *framework* zbudowany jest podobnie jak moduły odpowiedzialne za poszczególne warstwy systemu komunikacji grupowej. Podobieństwo polega na tym, że podobnie jak w modułach systemu komunikacji grupowej, w module *framework* istnieje funkcja inicjalizująca moduł (*initialize*) oraz funkcja rozgłaszająca wiadomość danego typu (*bcast*). Ponadto moduł *framework* kontaktuje się ze światem zewnętrznym (tj. pozostałymi modułami) poprzez system *callback'ów*, analogiczny do systemu *callback'ów* użytego w systemie komunikacji grupowej.

Odpowiednie wywołanie funkcji *bcast* powoduje uruchomienie mechanizmu komunikacji grupowej. W wyniku jego działania dana wiadomość jest rozgłaszana do wszystkich poprawnych procesów. Zapewniona jest odpowiednia kolejność dostarczania wiadomości — zależy ona od podanej przy inicjalizacji modułu relacji konfliktów.

Mechanizm komunikacji grupowej ostatecznie dostarcza poprawnym procesom rozgłaszaną wiadomość. Dostarczenie wiadomości z warstwy *Generic Broadcast* do modułu *framework* odbywa się przez wywołanie odpowiedniej funkcji *callback'owej*, podanej modułowi *Generic Broadcast* przy inicjalizacji. Następnie wiadomość jest dostarczana aplikacji, czyli wywoływana jest funkcja *callback'owa* podana modułowi *framework* przy inicjalizacji. Argumentami funkcji *callback'owych* jest dostarczana wiadomość. Warstwa aplikacyjna w momencie otrzymywania wiadomości wykonuje pewne akcje związane już z samą logiką aplikacji. W szczególnym przypadku może również rozgłosić nową wiadomość, wywołując funkcję *bcast.*

4.2.1.2 Relacja konfliktów

Framework do prawidłowego działania wymaga podania relacji konfliktów $C : M \times M \rightarrow bool$. W rzeczywistości jest to zwykła funkcja przyjmująca jako argumenty dwie wiadomości i zwracająca wartość logiczną. Wartość *prawda* oznacza, że wiadomości są *skonfliktowane.* W takim przypadku mechanizm rozgłaszający wiadomości dostarczy tę parę wiadomości w tej samej kolejności wszystkim procesom. Wartość *fałsz* oznacza, że wiadomości nie są *skonfliktowane* i mechanizm rozgłaszający wiadomości dostarczy tę parę wiadomości wszystkim procesom w dowolnej kolejności. W szczególnym przypadku mechanizm rozgłaszający wiadomości może dostarczyć wszystkim procesom *nieskonfliktowaną parę* wiadomości w tej samej kolejności.

Wartość logiczna relacji C determinuje wybór niższej warstwy do rozgłaszania: może nią być *Atomic Broadcast* lub *Reliable Broadcast.* Wybór pierwszej z nich charakteryzuje się większym kosztem komunikacyjnym w porównaniu z drugim wyborem.

Na relację konfliktów C nałożone jest następujące ograniczenie: Wymaga się, by dla danej pary wiadomości funkcja ta zwracała zawsze tę samą wartość. Uzasadnienie: każdy z procesów w mechanizmie komunikacji grupowej będzie osobno sprawdzał relację konfliktów. Algorytmy użyte w implementacji systemu komunikacji grupowej będą działać prawidłowo tylko w przypadku, gdy powyższe ograniczenie będzie spełnione.

4.2.2 Budowa wewnętrzna modułu

Moduł *framework* łączy poszczególne warstwy stosu systemu komunikacji grupowej (patrz rys. 3.2 na stronie 14) w jedną całość. Sam inicjalizuje odpowiednie warstwy oraz uruchamia komunikację między nimi. Programista korzystający z *framework'u* ma jedynie dostęp do udostępnionego

Rysunek 4.2: Definicje wiadomości w module *framework*

```
type gcs_init_data = Consensus.consensus_data * Abcast.abcast_data

type 'a mp_rtcp_req = gcs_init_data Mp.mp_rtcp_req
type 'a mp_gbcast_req = Mp.mp_gbcast_req

type 'a app_msg = 'a

type 'a gbcast_msg =
    GbcastAppMsg of 'a app_msg
  | GbcastMpMsg of 'a mp_gbcast_req
type 'a gbcast_phase_msg = 'a gbcast_msg Gbcast.t_msg
type 'a gbcast_bcast_msg = 'a gbcast_msg Gbcast.t_bcast_msg list

type 'a abcast_msg =
    AbcastGbcastMsg of 'a gbcast_bcast_msg
type 'a abcast_rtcp_msg = 'a abcast_msg Abcast.bcast_msg
type 'a abcast_proposal = 'a abcast_msg Abcast.proposal

type 'a rbcast_msg = RbcastGbcastMsg of 'a gbcast_bcast_msg
type 'a rbcast_rtcp_msg = 'a rbcast_msg Rbcast.bcast_msg

type 'a consensus_val = ConsensusAbcastProposal of 'a abcast_proposal
type 'a consensus_rtcp_msg = 'a consensus_val Consensus.bcast_msg

type 'a bebcast_msg =
    BebcastAbcastMsg of 'a abcast_rtcp_msg
  | BebcastConsensusMsg of 'a consensus_rtcp_msg
  | BebcastRbcastMsg of 'a rbcast_rtcp_msg
  | BebcastGbcastMsg of 'a gbcast_phase_msg
type 'a bebcast_rtcp_msg = 'a bebcast_msg Bebcast.bebcast_msg

type 'a rtcp_msg =
    RtcpBebcastMsg of 'a bebcast_rtcp_msg
  | RtcpConsensusMsg of 'a consensus_rtcp_msg
  | RtcpGbcastMsg of 'a gbcast_phase_msg
  | RtcpMpReq of 'a mp_rtcp_req

type 'a t_instance = {
  myaddr: (int * Unix.inet_addr);
  rtcp_udp_port: int;
  callback: ('a -> unit);
  conflict_relation: ('a gbcast_msg -> 'a gbcast_msg -> bool);
  mutable mp : gcs_init_data Mp.t_instance option;
  mutable gbcast : 'a gbcast_msg Gbcast.t_instance option;
  mutable abcast : 'a abcast_msg Abcast.abcast_instance option;
  mutable consensus : 'a consensus_val Consensus.consensus_instance option;
  mutable rbcast : 'a rbcast_msg Rbcast.t_instance option;
  mutable bebcast : 'a bebcast_msg Bebcast.bebcast_instance option;
  mutable rtcp : 'a rtcp_msg Rtcp.rtcp_instance option;
};;

let conflict_relation app_conflict_relation m1 m2 =
  match m1, m2 with
    GbcastAppMsg app_m1, GbcastAppMsg app_m2 -> app_conflict_relation app_m1 app_m2
  | GbcastMpMsg _, _ -> true
  | _, GbcastMpMsg _ -> true
;;
```

interfejsu.

Moduł *framework* składa się z zestawu definicji typów wiadomości, zestawu definicji funkcji typu *callback* oraz z implementacji funkcji interfejsu wymienionych w poprzednim rozdziale.

4.2.2.1 Typy wiadomości

Na rys. 4.2 znajduje się zestaw typów wiadomości przesyłanych przez poszczególne warstwy.

Funkcja *bcast* (patrz interfejs modułu na rys. 4.1) otrzymując wiadomość m do rozgłoszenia uruchamia mechanizm komunikacji grupowej. Odbywa się to poprzez wysłanie wiadomości m do modułu *gbcast (Generic Broadcast)*. Z modułu *gbcast* korzysta jednak jeszcze jeden moduł — *mp (Membership Protocol)*.

Modularna budowa systemu oraz jego elastyczność wymaga, by moduł *gbcast* działał zarówno dla wartości typu takiego jak m jak i dla wartości używanych przez warstwę *Membership Protocol*. Z tego powodu moduł *Generic Broadcast* zaprojektowano tak, że moduł ten nie wie jakiego typu wiadomości rozgłasza[2].

Zdefiniowano wspólny typ wariantowy dla rozgłaszanych wiadomości modułem *Generic Broadcast* o nazwie `'a gbcast_msg`. Na rys. 4.2 widać, że rozgłaszaną wiadomością modułem *Generic Broadcast (Gbcast)* może być wiadomość aplikacyjna (*GbcastAppMsg)* lub wiadomość pochodząca z modułu *Membership Protocol (GbcastMpMsg)*. Poprzez typ wariantowy dokonuje się enkapsulacji wiadomości i w ten sposób przedostaje się ona do kolejnej warstwy.

Ponadto dla modułu *Generic Broadcast* zdefiniowano dodatkowe typy wiadomości — są to typy `'a gbcast_phase_msg` oraz `'a gbcast_bcast_msg`. Są to typy wiadomości *wychodzących* z modułu *Generic Broadcast*. Zgodnie z rys. 3.2 moduł *Generic Broadcast* korzysta z usług modułów: *Atomic Broadcast, Reliable Broadcast, Best-Effort Broadcast oraz Robust TCP*. Z rys. 4.2 wynika, że pierwsze dwa moduły wysyłają wiadomości typu `'a gbcast_bcast_msg`, a ostatnie dwa moduły wysyłają wiadomości typu `'a gbcast_phase_msg`.

W analogiczny sposób zdefiniowano typy wiadomości dla wszystkich warstw.

4.2.2.2 Interfejs między-modułowy

Kolejne warstwy stosu protokołów połączone są systemem *callback'ów*. Moduł *framework* kolejnym warstwom podaje odpowiednie funkcje. Sterują one przepływem, "rozbierają" wiadomości. Sprawdzają jakiego są typu i na jego podstawie przekazują odpowiednim warstwom (modułom).

Niech za przykład posłuży warstwa *Reliable Broadcast* (rys. 4.3).

Funkcja *rbcast_bcast_callback* enkapsuluje wiadomość konstruktorem *BebcastRbcastMsg*. Tak "opakowana" wiadomość ma już typ `'a bebcast_msg` i jest wysłana "na dół" do modułu *Bebcast*.

Funkcja *rdeliver_callback* "rozbiera" wiadomość za pomocą *dopasowania do wzorca (*ang. pattern matching). Odpowiedni typ wiadomości implikuje wysłanie wiadomości do odpowiedniego modułu. W przypadku *Reliable Broadcast* w opracowanym *framework'u* wiadomość ta może zostać wysłana "*w górę"* tylko do modułu *Generic Broadcast*.

4.2.3 Wpływ Failure Detection na inne moduły

W przypadku odłączenia się jakiegoś procesu od grupy, bądź dołączenia się nowego (lub starego odłączonego z powodu awarii) procesu do grupy moduł *framework* wywołuje funkcję *group_change_callback*.

[2]Instancja Generic Broadcast'u ma typ `'a Gbcast.t_instance`, gdzie *'a* jest typem rozgłaszanej wiadomości. Funkcja *gbcast* w module *Gbcast* przyjmuje wiadomość typu *'a*.

Rysunek 4.3: Fragmenty modułu *framework* dot. *Reliable Broadcast*

```
(* ... *)

type 'a rbcast_msg = RbcastGbcastMsg of 'a gbcast_bcast_msg
type 'a rbcast_rtcp_msg = 'a rbcast_msg Rbcast.bcast_msg

(* ... *)

(* RBCAST *)
let string_of_rbcast_msg message =
  match message with
      RbcastGbcastMsg _ -> "rbcast (gbcast)"
;;

let rbcast_bcast_callback inst message group =
  logger_down (string_of_rbcast_msg (Rbcast.get_msg_from_bcast_msg message));
  Bebcast.bcast (optionValue inst.bebcast) group (BebcastRbcastMsg message)
;;

let rdeliver_callback inst message =
  logger_up (string_of_rbcast_msg message);
  match message with
      RbcastGbcastMsg m -> Gbcast.deliver (optionValue inst.gbcast) m
;;

(* ... *)

        inst.rbcast <- Some (Rbcast.initialize myaddr [myaddr] (rbcast_bcast_callback inst)
(rdeliver_callback inst));

(* ... *)
```

Funkcja ta przekazuje skład nowej grupy wszystkim warstwom. Wykonuje to przez wywołanie odpowiednich funkcji w odpowiednich modułach.

4.3 Implementacja warstwy Generic Broadcast

Implementacja warstwy *Generic Broadcast* (patrz rys. 3.2 na stronie 14) znajduje się w module *Gbcast*. Moduł ten udostępnia swój interfejs (rys. 4.5). Poprzez swój interfejs moduł *Gbcast* implementuje algorytm *Non-trivial Generic Broadcast*[19](nazywany później algorytmem *Generic Broadcast*) ukazany na rys. 4.4.

Celem warstwy *Gbcast* jest niezawodne rozgłaszanie wiadomości możliwie rzadko korzystając z wyroczni, jaką jest warstwa *Atomic Broadcast*. Warstwa ta wykorzystywana jest przez warstwę aplikacyjną oraz przez warstwę *Membership Protocol*.

4.3.1 Interfejs modułu

Moduł *Generic Broadcast* poprzez swój interfejs udostępnia:

- inicjalizator instancji `'a Gbcast.t_instance`
- funkcję *gbcast* rozgłaszającą wiadomość typu *'a*
- funkcję *r_deliver* dostarczającą modułowi *Gbcast* wiadomości wewnętrzne związane z rozgłaszaniem wiadomości
- funkcję *deliver* dostarczającą modułowi *Gbcast* wiadomości rozgłoszone przez niższe warstwy: *Abcast* lub *Rbcast*
- funkcje *suspend* i *set_new_group* odpowiednio zmieniające zbiór procesów w module *Gbcast*

4.3.2 Implementacja modułu

Postarano się, aby implementacja modułu *Generic Broadcast* wyglądała możliwie podobnie do pseudokodu. Użyty język programowania — język OCaml — pozwala na uzyskanie stosunkowo wysokiego poziomu abstrakcji (w porównaniu z innymi językami programowania).

Implementacja modułu Generic Broadcast znajduje się na rys. 4.6.

4.3.3 Moduł *Gbcast* we *framework'u*

Jak każdy inny moduł w opracowanym systemie komunikacji grupowej również ten moduł ma swoje funkcje typu *callback* we *framework'u*. Znajduje się w nim również relacja konfliktów. Część kodu źródłowego *framework'u* znajduje się na rys. 4.7. Z tego kodu widać, że takie pary wiadomości, z których jedna jest wiadomością modułu *Membership Protocol* są w konflikcie. Ta para wiadomości zostanie zatem dostarczona do wszystkich poprawnych procesów w tej samej kolejności.

Na rys. 4.7 można ponownie zobaczyć jak działają funkcje typu *callback*: funkcje wysyłające wiadomość "w dół" stosu protokołów po prostu enkapsulują wiadomość i wysyłają ją do odpowiedniego modułu. Funkcja *callback* dostarczająca wiadomość przez moduł *Gbcast* (czyli: wysyłająca wiadomość "w górę" stosu protokołów) "rozbiera" wiadomość i wysyła ją do odpowiedniego modułu w zależności od wnętrza wiadomości.

Rysunek 4.4: Pseudokod implementacji *Non-trivial Generic Broadcast* (źródło: [19])

```
1: For every process p:
2:
3: procedure INIT
4:     seen ← ∅
5:     possibleRB ← ∅
6: end procedure
7:
8: upon RECEIVE(m, FIRST)from q:
9:     seen ← seen ∪ m
10:    good ← {r : received (m, GOOD, SECOND) from r}
11:    if seen has no messages conflicting with m then
12:        SEND(m, GOOD, SECOND) to all processes
13:    else
14:        SEND(m, BAD, SECOND) to all processes
15:    end if
16:
17: upon RECEIVE(m, ∗, SECOND)from n − f processes for the first time:
18:    seen ← seen ∪ m
19:    if |good| > n/2 and seen has no messages conflicting with m then
20:        possibleRB ← possibleRB ∪ m
21:    end if
22:    SEND(m, possibleRB ∩ C(m), THIRD) to sender(m)
23:
24: upon RECEIVE(m, ∗, THIRD)from n − f processes for the first time:
25:    R ← {r : received (m, ∗, THIRD) from r}
26:    for all r ∈ R do
27:        poss[r] ← M s.t. received (m, M, THIRD) from r
28:    end for
29:    if |r : m ∈ poss[r]| > n/2 then
30:        RBROADCAST(m)
31:    else
32:        ABROADCAST(m, ⋃_{r∈R} poss[r])
33:    end if
34:
35: upon RDELIVER(m):
36:    if m not gdelivered then
37:        gdeliver(m)
38:    end if
39:
40: upon ADELIVER(m, prec):
41:    for all m′ ∈ prec do
42:        if m′ not gdelivered then
43:            gdeliver(m′)
44:        end if
45:    end for
46:    if m not gdelivered then
47:        gdeliver(m)
48:    end if
49:
```

Rysunek 4.5: Interfejs modułu *Gbcast*

```
open Globals;;

(* 'a - message type *)
type 'a t_msg
type 'a t_instance
type 'a t_bcast_msg

(** Initializes gbcast layer.
  @param myself : process_id - own process ID
  @param group : process_id list - process group
  @param abcast_send : 'a -> unit - abcast (oracle) to use
  @param rbcast_send : 'a -> unit - of rbcast to use
  @param rtcp_send : 'a t_message -> process_id -> unit - a way to use rtcp
  @param gdeliver_callback : 'a -> unit - gdeliver callback
  @param conflict_relation : 'a -> 'a -> bool - conflict relation to use in gbcast
  *)
val initialize : process_id -> (process_id list)
-> ('a t_bcast_msg list -> unit)
-> ('a t_bcast_msg list -> unit)
-> ('a t_msg -> process_id list -> unit)
-> ('a t_msg -> process_id -> unit)
-> ('a -> unit) -> ('a -> 'a -> bool) -> 'a t_instance

(** Gbcasts message.
  @param instance : 'a gbcast_instance - gbcast instance
  @param message : 'a - message to send
  **)
val gbcast : 'a t_instance -> 'a -> unit

(** R_delivers messages to gbcast layer.
  @param instance : 'a gbcast_instance - gbcast instance
  @param message : 'a gbcast_message - message to deliver
  **)
val r_deliver : 'a t_instance -> 'a t_msg -> unit

(** Delivers messages to upper layers.
  * @param instance : 'a gbcast_instance - gbcast instance
  * @param message : 'a t_bcast_msg - message to deliver
  * *)
val deliver : 'a t_instance -> 'a t_bcast_msg list -> unit

val suspect : 'a t_instance -> process_id list -> unit

val set_new_group : 'a t_instance -> process_id list -> unit

val msg_of_bcast_msg : 'a t_bcast_msg list -> 'a list
val string_of_t_msg : 'a t_msg -> string
```

Rysunek 4.6: Implementacja modułu *Gbcast*

```
let gbcast instance message =
  bcast instance (First(message, generate_new_messageId instance, instance.myself))
;;

let r_deliver_first instance m =
  add_to_seen_set instance m;
  if no_conflicting_messages_in_seen instance m
  then bcast instance (Second(Good, m))
  else bcast instance (Second(Bad, m))
;;

let r_deliver_second instance m good_bad =
  let (msg, messageID, sender) = m in
    Hashtbl.add instance.secondMessages (messageID, sender) good_bad;
    let received_goodbads = Hashtbl.find_all instance.secondMessages (messageID, sender) in
      if List.length received_goodbads = List.length instance.group then
        begin
          add_to_seen_set instance m;
          ( let goods =
              let filter_fun = function Good -> true | Bad -> false
              in List.filter filter_fun received_goodbads
            in
              if List.length goods > List.length instance.group / 2 &&
                 no_conflicting_messages_in_seen instance m
              then add_to_possibleRB_set instance m );
          let poss =
            let filter_fun m =
              let (hd_msg, _, _) = m
              in hd = m || instance.conflict_relation hd_msg msg
            in List.filter filter_fun instance.possibleRB
          in instance.rtcp_send (Third(m, poss)) sender;
        end;
      ();;


let r_deliver_third instance m poss =
  let (msg, messageID, sender) = m in
    Hashtbl.add instance.thirdMessages (messageID, sender) poss;
    let received_posses = Hashtbl.find_all instance.thirdMessages (messageID, sender) in
      if List.length received_posses = List.length instance.group then
        begin
          let posses_having_m = List.filter (fun poss -> List.mem m poss) received_posses in
            if List.length posses_having_m > List.length instance.group / 2
            then rbroadcast instance m
            else abroadcast instance m received_posses
        end;
      ();;

let r_deliver instance message =
  match message with
      First (m) -> r_deliver_first instance m
    | Second (good_bad, m) -> r_deliver_second instance m good_bad
    | Third (m, poss) -> r_deliver_third instance m poss
;;
```

Rysunek 4.7: Kod źródłowy *framework'u* dotyczący modułu *Gbcast*

```
(* ... *)

let conflict_relation app_conflict_relation m1 m2 =
  match m1, m2 with
     GbcastAppMsg app_m1, GbcastAppMsg app_m2 -> app_conflict_relation app_m1 app_m2
    | GbcastMpMsg _, _ -> true
    | _, GbcastMpMsg _ -> true
;;

(* ... *)

let gbcast_abcast_callback inst message =
  Abcast.abcast (optionValue inst.abcast) (AbcastGbcastMsg message)
;;

let gbcast_rbcast_callback inst message =
  Rbcast.rbcast (optionValue inst.rbcast) (RbcastGbcastMsg message);
;;

let gbcast_bcast_callback inst message group =
  Bebcast.bcast (optionValue inst.bebcast) group (BebcastGbcastMsg message)
;;

let gbcast_rtcp_callback inst message process_id =
  rtcp_send inst process_id (RtcpGbcastMsg message)
;;

let gdeliver_callback inst message =
  match message with
      GbcastAppMsg m -> inst.callback m
    | GbcastMpMsg m -> Mp.g_deliver (optionValue inst.mp) m
;;

(* ... *)
```

Rysunek 4.8: Pseudokod implementacji *Membership Protocol* (źródło: opracowanie własne)

```
1: For every process p:
2:
3: upon JOIN(x):
4:     SEND(JoinRequest (myself)) to x
5:
6: upon RECEIVE(JOINREQUEST (j)):
7:     GBCAST(GbcastJoinRequest (j, myself, mygroup))
8:
9: upon RECEIVE(GBCASTJOINREQUEST (j, x, group)):
10:    new_group ← {j} ∪ group
11:    deliver(new_group, null)
12:    if x = myself then
13:        current_state ← GCS.GetCurrentState()
14:        SEND(ApprovedJoinRequest (new_group, current_state)) to j
15:    end if
16:
17: upon RECEIVE(APPROVEDJOINREQUEST (group, current_state)):
18:    deliver(group, current_state)
19:
```

4.4 Implementacja warstwy Membership Protocol

Implementacja warstwy *Membership Protocol* (patrz rys. 3.2 na stronie 14) znajduje się w module *Mp*. Moduł ten udostępnia swój interfejs (rys. 4.9). Poprzez swój interfejs moduł *Mp* implementuje algorytm *Membership Protocol* ukazany na rys. 4.8.

Celem warstwy *Mp* jest możliwość dołączenia procesów do grupy w taki sposób, aby powstała grupa była spójna. Powstaje pytanie: czym jest grupa spójna?

Definition 1. *Spójną grupą procesów* nazwiemy taką grupę procesów, w której każdy proces posiada identyczną listę procesów będących w grupie.

Warstwa ta wykorzystywana jest przez warstwę *framework* w momencie uruchomienia *framework'u* przez proces dołączający. Warstwa ta jest również używana przez procesy, które już dołączyły i działają w grupie — komunikacja między procesami w grupie zapewnia spójność grupy procesów po dołączeniu nowego procesu.

W ramach niniejszej pracy opracowano algorytm rozwiązujący problem *Membership Protocol*. Jego pseudokod znajduje się na rys. 4.8.

4.4.1 Interfejs modułu

Moduł *Membership Protocol (Mp)* poprzez swój interfejs udostępnia:

- inicjalizator instancji `'a Mp.t_instance`
- funkcję *join* dołączającą wywołujący proces do grupy reprezentowanej przez dany proces
- funkcje *r_deliver* oraz *g_deliver* dostarczające modułowi *Mp* wiadomości wewnętrzne związane z dołączaniem procesu do grupy. Funkcja *r_deliver* dostarcza wiadomości od jednego konkretnego procesu (są to wiadomości typu: żądanie dołączenia do grupy oraz potwierdzenie dołączenia do grupy). Funkcja *g_deliver* dostarcza rozgłoszoną po wszystkich procesach wiadomość o nowym dołączającym.

Rysunek 4.9: Interfejs modułu *Mp*

```
open Globals;;

type 'a mp_rtcp_req
type mp_gbcast_req
type 'a t_instance

val initialize :
  process_id ->
  (process_id list) ->
  ('a mp_rtcp_req -> process_id -> unit) ->
  (mp_gbcast_req -> unit) ->
  (process_id list -> 'a option -> unit) ->
  (unit -> 'a) ->
  'a t_instance
;;

val join : 'a t_instance -> process_id -> unit;;
val r_deliver : 'a t_instance -> 'a mp_rtcp_req -> unit;;
val g_deliver : 'a t_instance -> mp_gbcast_req -> unit;;
```

4.4.2 Implementacja modułu

Postarano się, aby implementacja modułu *Membership Protocol (Mp)* wyglądała równie zgrabnie jak pseudokod. Użyty język programowania — język OCaml — a szczególnie jego cecha — *dopasowanie do wzorca*, pozwolił na zwięzły zapis kodu źródłowego. Pokazano go na rys. 4.10.

4.4.3 Moduł *Membership Protocol* we *framework'u*

Moduł *Membership Protocol (Mp)* nie odbiega od normy i również ma swoje *callback'i* we *framework'u.* Funkcje te pełnią jednak trochę inną rolę w porównaniu z typowymi *callback'ami* dla modułów rozgłaszających wiadomości. Przypomnijmy, że moduł *Membership Protocol* odpowiada za dołączanie procesów do grupy procesów. Zatem moduł ten ma bardziej bezpośredni wpływ na pozostałe moduły. Wysyła im sygnały z nową listą procesów w grupie oraz odbiera od tych modułów ich stan. Stan innych modułów jest potrzebny do wysłania go nowo dołączanym procesom.

Aby zapewnić jakikolwiek stopień niezależności modułów (ich rodzajów i liczby) od modułu *Membership Protocol*, do zbierania informacji o stanie modułów i aplikacji oraz do wysyłania nowej listy procesów do modułów posłużono się *callback'ami*.

Rysunek 4.10: Implementacja modułu *Membership Protocol*

```
type 'a mp_rtcp_req =
    JoinRequest of process_id
  | ApprovedJoinRequest of (process_id list * 'a)
type mp_gbcast_req =
    GbcastJoinRequest of (process_id * process_id * process_id list)

type 'a t_instance = {
  myself : process_id;
  group : process_id list;
  rtcp : ('a mp_rtcp_req -> process_id -> unit);
  gbcast : (mp_gbcast_req -> unit);
  deliver : (process_id list -> 'a option -> unit);
  get_gcs_init_data : unit -> 'a;
};;

let initialize myself group rtcp_send gbcast_send deliver get_gcs_init_data =
  let instance = {
    myself = myself;
    group = group;
    rtcp = rtcp_send;
    gbcast = gbcast_send;
    deliver = deliver;
    get_gcs_init_data = get_gcs_init_data
  }
  in instance;;

let join inst x =
  inst.rtcp (JoinRequest inst.myself) x
;;

let r_deliver inst msg =
  match msg with
      JoinRequest j ->
        let x = inst.myself
        and group = inst.group
        in let m = (j, x , group)
        in
          inst.gbcast (GbcastJoinRequest m)
    | ApprovedJoinRequest (group, gcs_init_data) ->
        inst.deliver group (Some gcs_init_data)
;;

let g_deliver inst msg =
  match msg with
      GbcastJoinRequest (j, x, group) ->
        inst.deliver (j::group) None;
        if x = inst.myself then
          inst.rtcp (ApprovedJoinRequest ((j::group), inst.get_gcs_init_data ())) j;
;;
```

Rysunek 4.11: Kod źródłowy *framework'u* dotyczący modułu *Membership Protocol*

```
(* ... *)
type gcs_init_data = Consensus.consensus_data * Abcast.abcast_data

type 'a mp_rtcp_req = gcs_init_data Mp.mp_rtcp_req
type 'a mp_gbcast_req = Mp.mp_gbcast_req

(* ... *)

let mp_rtcp_send_callback inst req process_id =
  ( let conns = rtcp_get_all_connections inst
    in if not (List.mem process_id conns)
    then ignore (Rtcp.open_client (optionValue inst.rtcp) (snd process_id) (fst process_id))
  );
  rtcp_send inst process_id (RtcpMpReq req)
;;

let mp_gbcast_callback inst message =
  Gbcast.gbcast (optionValue inst.gbcast) (GbcastMpMsg message)
;;

let mp_group_change_callback inst group gcs_init_data =
  let new_processes_to_connect =
    let connected_processes = rtcp_get_all_connections inst
    in let is_process_connected id = List.mem id connected_processes
    in let filter_fun id = id <> inst.myaddr && (not (is_process_connected id))
    in List.filter filter_fun group
  in
    ( if (List.length new_processes_to_connect) > 0
      then
        let connect process_id =
          ignore (Rtcp.open_client (optionValue inst.rtcp) (snd process_id) (fst process_id))
        in List.iter connect new_processes_to_connect
    );

    ( let sorted_group group =
        let comp a b = (process_num a) - (process_num b)
        in List.sort comp group
      in let group_with_myself group =
        if List.mem inst.myaddr group then group else inst.myaddr::group
      in let group = sorted_group (group_with_myself group)
      in
        Consensus.set_new_group (optionValue inst.consensus) group;
        Abcast.set_new_group (optionValue inst.abcast) group;
        Rbcast.set_new_group (optionValue inst.rbcast) group;
        Gbcast.set_new_group (optionValue inst.gbcast) group;
        Mp.set_new_group (optionValue inst.mp) group;
        connection_list_changed inst ();
    );

    ( match gcs_init_data with
        Some (consensus_data, abcast_data) ->
          Consensus.set_consensus_data (optionValue inst.consensus) consensus_data;
          Abcast.set_abcast_data (optionValue inst.abcast) abcast_data
      | None -> ()
    );
;;
```

Rozdział 5

Testy i walidacja

Zaprojektowany system komunikacji grupowej należało przetestować. Wykonywane były dwa rodzaje testów. Pierwszym rodzajem testów była kompilacja. System typów w języku OCaml pozwala na dokładną weryfikację użytych typów. Drugim rodzajem testów było testowanie zaimplementowanej specjalnie do testów aplikacji wykorzystującej stos protokołów komunikacji grupowej (*framework*). Aplikacja wykorzystuje system komunikacji grupowej do implementacji replikacji pasywnej. Zarządzanymi danymi przez aplikację jest książka telefoniczna.

Podczas testów wielokrotnie wyszły na jaw błędy. Były to błędy zarówno konstrukcyjne jak i programistyczne. Te pierwsze występowały w pierwszym rodzaju testów.

5.1 Testowanie przez kompilację w języku OCaml

Język OCaml jest językiem *statycznie typowanym* z *inferencją typów*. Cechy te pozwalają w pewnym stopniu stwierdzić poprawność programu w momencie jego kompilacji.

Statyczne typowanie Statyczne typowanie oznacza określanie typów wyrażeń w czasie kompilacji programu. Po określeniu typów wyrażeń sprawdzana jest ich zgodność w odpowiednich kontekstach, np. operator dodawania liczb całkowitych jako argumenty przyjmuje wyłącznie liczby całkowite. Brak zgodności typów implikuje błąd kompilacji.

Cecha ta jest popularną cechą wśród języków programowania.

Inferencja typów Inferencja typów, czyli wywodzenie typów to cecha języka programowania, która zwalnia programistę z obowiązku podawania typów zmiennych, parametrów oraz wyrażeń. Obowiązek identyfikacji owych typów zostaje przerzucony na kompilator.

Typy wyrażeń zostają przez kompilator automatycznie wyznaczone. W razie braku dopasowania typów następuje błąd kompilacji. Jest to pewnego rodzaju walidacja kodu źródłowego programu. Różni się ona od standardowego dopasowywania typów w językach bez inferencji typów tym, że w kodzie źródłowym w języku OCaml podajemy typy bardzo rzadko — lub wcale. Pozwala to programiście skupić się na logice programu.

Po zaprogramowaniu logiki programu następuje kompilacja, podczas której następuje (między innymi) walidacja typów wyrażeń czy zmiennych. Poprawna walidacja oznacza, że programista zaprogramował logikę programu spójnie — tj. wyrażenia są zgodne same ze sobą.

Wpływ systemu typów języka OCaml na poprawność implementacji Dzięki takim cechom języka OCaml jak inferencja typów czy statyczny system typów autor implementacji bardziej skupił się na wyższych warstwach abstrakcji programowania — na konstrukcji samego systemu i implementacji poszczególnych warstw. Walidacja typów przez kompilator języka OCaml następowała później, tj. po implementacji logiki aplikacji. Nie była też zbyt kosztowna. Miały również miejsce momenty zmiany typów pewnych wyrażeń, czy parametrów funkcji. Nie były one jednak problematyczne.

5.2 Testowanie aplikacją testową

Podstawowym narzędziem-aplikacją testową walidującą poprawność implementacji systemu komunikacji grupowej jest aplikacja testowa *ar_main*. Aplikacja ta po uruchomieniu łączy się z pozostałymi procesami w grupie lub tworzy nową grupę procesów (o liczebności równej 1). Każdy proces oczekuje na żądania od klienta przez protokół UDP. Żądania są następnie rozgłaszane do wszystkich procesów w grupie. Tę funkcję spełnia system komunikacji grupowej.

Warstwą aplikacyjną w aplikacji testowej *ar_main* jest prosta rozproszona książka telefoniczna. Żądania klienta dotyczą właśnie owej książki telefonicznej.

5.2.1 Aplikacja testowa

Aplikacja *ar_main* składa się z następujących modułów:

moduł *ar_app* implementuje prostą książkę telefoniczną

moduł *ar_framework* pełni funkcję modułu *framework* systemu komunikacji grupowej

moduł *ar_udp* odpowiada za komunikację procesu z klientem usługi

moduł *ar_main* łączy funkcjonalność powyższych modułów w całość

Procesy będące w grupie implementują pewien rodzaj replikacji pasywnej. Wygląda to tak, że klient usługi łączy się z dowolnym procesem przez połączenie UDP. Tym kanałem są wysyłane żądania systemowi. Dostępne żądania:

- dodanie numeru telefonu do danego rekordu (jeśli wpis dla rekordu o podanym nazwisku nie istnieje, to zostanie utworzony),
- usunięcie numeru telefonu z danego rekordu,
- edycja numeru telefonu w danym rekordzie,
- podgląd numerów telefonów w danym rekordzie.

Rekordy identyfikowane są po nazwisku (dowolny łańcuch znaków składający się z liter alfabetu angielskiego).

Proces otrzymujący żądanie wysyła go do pozostałych procesów w grupie. System komunikacji grupowej zapewnia odpowiednią kolejność dostarczania wiadomości do procesów w grupie. Nie musi to być zawsze jednakowa kolejność — patrz *Generic Broadcast* (rozdział 2.2.4 na stronie 8).

Rysunek 5.1: Relacja konfliktów w aplikacji testowej - kod źródłowy

```
let conflict_relation m1 m2 =
  let is_view m = match m with View -> true | _ -> false
  in
    if (is_view m1) || (is_view m2) then false
    else
      let name m = match m with
          Add (_, Name name) -> name
        | Del (_, Name name) -> name
        | Upd (_, _, Name name) -> name
        | View -> ""
      in if (name m1) <> (name m2) then false
      else
        match (m1, m2) with
            Add (_, Name n1), Add (_, Name n2) when n1 = n2 -> true

          | Add (_, Name n1), Del (_, Name n2) when n1 = n2 -> true
          | Del (_, Name n1), Add (_, Name n2) when n1 = n2 -> true

          | Del (_, Name n1), Upd (_, _, Name n2) when n1 = n2 -> true
          | Upd (_, _, Name n1), Del (_, Name n2) when n1 = n2 -> true

          | Upd (PhoneNo old_phone_no1, PhoneNo new_phone_no1, Name n1),
            Upd (PhoneNo old_phone_no2, PhoneNo new_phone_no2, Name n2)
                when (n1 = n2) && ((old_phone_no1 = old_phone_no2) || (new_phone_no1 =
new_phone_no2))
            -> true

          | _ -> false
;;
```

5.2.2 Proces testowania

Należało przetestować odłączanie i dołączanie procesów do grupy oraz samo rozgłaszanie wiadomości.

5.2.2.1 Testowanie rozgłaszania wiadomości

W aplikacji testowej wiadomości są rozgłaszane *generycznie*. Podano relację konfliktów dla rozgłaszanych wiadomości. Jej postać jest ukazana w postaci kodu źródłowego na rys. 5.1.

W ramach testowania rozgłaszania wiadomości sprawdzono, czy warstwa *Generic Broadcast* odpowiednio reaguje na żądania, tj. czy wywołuje odpowiednią warstwę niższą (*Reliable Broadcast* lub *Atomic Broadcast*). Tym samym sprawdzono, czy spełniona jest własność *ogólnego (generycznego) uporządkowania wiadomości* (patrz 2.2.4). Przy okazji sprawdzono pozostałe własności *rozgłaszania generycznego*.

Testowanie polegało na rozgłaszaniu różnych typów wiadomości w różnych kolejnościach w kolejnych instancjach grup procesów.

5.2.2.2 Testowanie dołączania procesów

Aplikacja testowa umożliwia dołączanie nowych procesów. W tym celu korzysta z *framework'u*. Aplikacja przy dołączaniu procesów oprócz wykorzystania odpowiednich funkcji *framework'u* ma zapewnić dostarczenie nowym procesom aktualnego stanu przetwarzania — stan *framework'u* oraz stan logiki samej aplikacji (zawartość książki telefonicznej).

Przeprowadzono proste testy dołączające nowe procesy oraz stare procesy (tj. takie, które ręcznie odłączono od grupy — przerywając działanie danego procesu). Przy nowo dołączonych procesach sprawdzono zawartość książki telefonicznej.

Poprawność przesyłania stanu *framework'u* wynikła z poprawności działania rozgłaszania wiadomości w grupie z nowo dołączonymi procesami. Poprawność zawartości książki telefonicznej sprawdzono wysyłając odpowiednie żądanie oraz sprawdzenie odpowiedzi na nie.

Testowanie polegało na odłączaniu danych procesów, dołączaniu nowych procesów i wysłaniu żądań implikujących rozgłaszanie wiadomości. Sprawdzono efekty tych działań.

Rozdział 6

Wnioski i krótki opis możliwych rozszerzeń

W tym rozdziale przedstawiono wnioski wyniesione podczas prac nad systemem komunikacji grupowej oraz podano możliwości rozszerzenia projektu.

6.1 Wnioski

6.1.1 Język programowania OCaml

Wybór języka programowania padł na OCaml. Uzasadnieniem były wspomniane już w tej pracy cechy tego języka:

- statyczny system typów z inferencją typów
- system modułów i interfejsów

Obie cechy przydały się autorowi tej pracy.

System typów z inferencją typów pozwolił autorowi na skupienie się na algorytmie — zwolnił autora z nadmiernego (w porównaniu z innymi językami programowania bez inferencji typów) myślenia nad typami wyrażeń. Z drugiej strony, ścisłe dopasowywanie typów przez kompilator pomogło autorowi w konstrukcji modularnego systemu. Zwiększa również szansę na to, że program jest poprawny.

System modułów i interfejsów wyraźnie dzieli kod źródłowy modułów. Moduły komunikują się między sobą wyłącznie za pomocą ich interfejsów, tj. żeby funkcje danego modułu były dostępne "na zewnątrz" (w innych modułach), muszą zostać uwzględnione w interfejsie danego modułu. Wyraźny podział kodu źródłowego modułów podczas konstrukcji systemu oraz implementowania algorytmów pozwolił autorowi na skupienie się wyłącznie na danym module.

Powyższe cechy pozwalają również stwierdzić autorowi, że język OCaml jest dobrym językiem do prototypowania rozwiązań.

6.1.2 Testowanie

Pomimo modularności, implementowany system komunikacji grupowej okazał się dosyć trudny w testowaniu. Trudność ta polegała na tym, że testując daną funkcjonalność trzeba było uruchomić

wiele modułów. Pojawienie się błędu w przetwarzaniu nie identyfikowało od razu modułu, w którym jest owy błąd. Dojście do tego, w którym module był błąd polegało na wstrzyknięciu wielu poleceń typu *print* lub użycie *debugger'a*. Było to jednak dosyć czasochłonne i wymagało dokładnej analizy logów tworzonych przez polecenia typu *print*.

6.1.3 Podstawa systemu

Autor stworzył system komunikacji grupowej na podstawie istniejącej biblioteki komunikacji grupowej w języku OCaml (patrz rozdział 4.1). Biblioteka ta posiadała implementację (między innymi) podstawowej warstwy komunikacji jaką jest *Robust TCP*, czyli warstwa komunikacji między procesami na najniższym poziomie. Autor przyznaje jednak, że nie została ona napisana zbyt starannie. Pojawiło się kilka błędów, które autor musiał rozwiązać. Zajęło to autorowi trochę czasu.

Lekcja jaką autor pracy wyniósł jest taka, że w przypadku przyjęcia obcego względem autora kodu do swojej bazy kodu źródłowego należy przeznaczyć pewną ilość czasu na jego przetestowanie i poprawki.

6.2 Możliwe rozszerzenia systemu

6.2.1 Środowisko testujące

Jednym z problemów, z którymi borykał się autor pracy był brak środowiska testującego rozwijane oprogramowanie. Powodował on liczne utrudnienia w szukaniu błędów programistycznych.

W trakcie rozwijania systemu komunikacji grupowej autor pracy znalazł ciekawe środowiska testujące oprogramowanie tworzone w języku OCaml: OUnit[1] oraz Kaputt [2].

Jednym z rozszerzeń systemu mogłoby być stworzenie środowiska testującego system na podstawie jednej z wymienionych wyżej platform testujących oprogramowanie tworzone w języku OCaml. Takie środowisko powinno testować każdy moduł osobno. Dla każdego modułu powinny istnieć różne scenariusze testowe. Jeśli dany moduł jest trudny do przetestowania w takim środowisku, można by rozważyć *refactoring* modułu, czyli zmianę sposobu implementacji modułu bez zmiany jego funkcjonalności, przy czym dopuszcza się tu zmianę interfejsu.

6.2.2 Implementacja warstwy Generic Broadcast algorytmem *thrifty*

W obecnym systemie komunikacji grupowej autor pracy zaimplementował warstwę Generic Broadcast korzystając z algorytmu *Non-trivial Generic Broadcast* ([19]). Algorytm *Thrifty Generic Broadcast* ([19]) rozwiązuje problem *rozgłaszania ogólnego (generycznego)* lepiej, tj. rzadziej korzysta z wyroczni (którą w przypadku zaimplementowanego systemu komunikacji grupowej jest warstwa *Atomic Broadcast*) niż algorytm *Non-trivial Generic Broadcast*. W tym sensie, ale również patrząc na sam algorytm ([19]) można stwierdzić, że algorytm *Thrifty Generic Broadcast* jest rozszerzeniem algorytmu *Non-trivial Generic Broadcast*.

Powodem wyboru *Non-trivial Generic Broadcast* przez autora pracy jako algorytmu warstwy *Generic Broadcast* w systemie był kończący się czas na rozwój systemu.

[1] http://www.xs4all.nl/~mmzeeman/ocaml/
[2] http://kaputt.x9c.fr/

Bibliografia

[1] K. P. Birman i R. V. R. (eds.). *Reliable distributed computing with the Isis toolkit*. IEEE Computer Society Press, 1994.

[2] R. V. Renesse, K. P. Birman i S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, Kwiecień 1996.

[3] D. Dolev i D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, Kwiecień 1996.

[4] Y. Amir i J. Stanton. The Spread wide area group communication system. Raport techniczny CNDS-98-4, Department of Computer Science, Johns Hopkins University, 1998.

[5] M. Hayden. The Ensemble system. Raport techniczny TR98-1662, Department of Computer Science, Cornell University, Styczeń 1998.

[6] H. Miranda, A. Pinto i L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. W *Proceedings of ICDCS '01: the 21st IEEE International Conference on Distributed Computing Systems*, Kwiecień 2001.

[7] EPFL. Fortika. http://lsrwww.epfl.ch/crystall/, 2006.

[8] EPFL. Samoa. http://lsrwww.epfl.ch/samoa/, 2008.

[9] Spread Concepts LLC. The Spread toolkit. http://www.spread.org/, 2006.

[10] B. B. . R. Hat. JGroups toolkit. http://www.jgroups.org/, 2009.

[11] S. Mena, A. Schiper i P. T. Wojciechowski. A step towards a new generation of group communication systems. W M. Endler i D. Schmidt, edytorzy, *Proceedings of Middleware '03: the 4th ACM/IFIP/USENIX International Middleware Conference (Rio de Janeiro, Brazil)*, volume 2672 of *LNCS*, strony 414–432. Springer, Czerwiec 2003.

[12] INRIA. Objective Caml. http://caml.inria.fr, 2009.

[13] R. Guerraoui i L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.

[14] S. Mena de la Cruz. *Protocol composition frameworks and modular group communication*. Praca doktorska, EPFL, EPFL, 2006.

[15] O. Rütti. *Concurrency and Dynamic Protocol Update for Group Communication Middleware*. Praca doktorska, EPFL, EPFL, 2009.

[16] T. D. Chandra i S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[17] M. J. Fischer, N. A. Lynch i M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, Kwiecień 1985.

[18] E. Biagioni, R. Harper i P. Lee. A network protocol stack in Standard ML. *Higher-Order and Symbolic Computation*, 14(4):309–356, Grudzień 2001.

[19] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier i S. Toueg. Thrifty generic broadcast. W *Proceedings of the DISC 2000: the 14th International Conference on Distributed Computing*, Październik 2000.

Dodatek A

Krótki manual jak korzystać z biblioteki

Aby skorzystać z biblioteki należy zaimplementować własny moduł *framework* łączący pozostałe moduły i udostępniający poprzez swój interfejs swoją funkcjonalność. Moduł *framework* powinien posiadać funkcję *init*, która inicjalizuje i uruchamia system komunikacji grupowej. W zależności od tego, czy korzystający z biblioteki ma zamiar wykorzystać moduł *Membership Protocol*, moduł *framework* powinien udostępnić również funkcję *join*, która dołącza proces do danej grupy procesów. Autor nowego modułu *framework* musi również skonstruować nowy stos protokołów oraz zaimplementować nowe funkcje typu *callback*, które połączą warstwy.

W celu uniknięcia tworzenia stosu protokołów na nowo autor pracy proponuje, by skopiować kod źródłowy modułu *ar_framework* (moduł *framework* aplikacji testowej) oraz odpowiadający temu modułowi interfejs. Następnie można nanieść zmiany w module, np. dodać drugą warstwę *Atomic Broadcast* oraz udostępnić dla niej interfejs dla warstwy aplikacyjnej. W tej chwili w module *ar_framework* nie można skorzystać z warstwy *Atomic Broadcast* bezpośrednio. Do rozgłaszania wiadomości można użyć wyłącznie warstwy *Generic Broadcast*.

Zmiany w module *ar_framework* nie są jednak potrzebne. Przymus rozgłaszania wiadomości jedynie za pomocą warstwy *Generic Broadcast* nie stwarza problemów. Moduł *ar_framework* jest inicjowany w module *ar_main*. Przy inicjalizacji należy podać: numer portu UDP, na którym proces ma nasłuchiwać na komunikaty od innych procesów, funkcję typu *callback* reagującą na dostarczane wiadomości oraz relację konfliktów dla warstwy *Generic Broadcast*.

Dodatek B

Płyta CD ze źródłami kodu wszystkich programów oraz źródłami dokumentów

Na załączonej płycie CD znajdują się kody źródłowe systemu *CamlGroups*, aplikacja testowa *ar_main* oraz zestaw testów do systemu *CamlGroups* i aplikacji testowej *ar_main*. Na płycie CD znajdują się również źródła niniejszej pracy.