



Katedra Inżynierii Oprogramowania
kierunek studiów: Informatyka
specjalność: Inżynieria Oprogramowania

Praca dyplomowa inżynierska

**IMPLEMENTACJA APLIKACJI DO WSPÓŁTWORZENIA
NOTATEK Z WYKORZYSTANIEM TECHNOLOGII PEER TO
PEER**

**IMPLEMENTATION OF A COLLABORATIVE NOTE-TAKING APPLICATION
USING PEER-TO-PEER TECHNOLOGY**

Oskar Marcin Chybowski

nr albumu: **54941**

Opiekun:

dr inż. Mirosław Mościcki

Katedra Inżynierii Oprogramowania

Szczecin, 2026

Spis treści

Wstęp	5
1 Zakres teoretyczny	6
1.1 Systemy współdzielenia dokumentów w czasie rzeczywistym	6
1.2 Algorytmy zapewniające silną ewentualną zbieżność	7
1.3 Architektura Peer-To-Peer w środowiskach mobilnych	8
1.4 Architektura Peer-To-Peer w systemach iOS, iPadOS, macOS, tvOS.	8
1.5 Istniejące rozwiązania i ich problemy	10
2 Wymagania systemowe	11
2.1 Wymagania funkcjonalne	11
2.2 Wymagania нефункционалне	12
3 Implementacja	13
3.1 Projekt architektury systemu	13
3.2 Model danych	14
3.3 Warstwa sieciowa i komunikacja P2P	16
3.4 Odkrywanie innych urządzeń	16
3.5 Transportowanie danych	20
3.6 Algorytm rozwiązywania konfliktów	22
3.7 Środowisko developerskie i stack technologiczny	22
3.8 Interfejs użytkownika	23
3.9 Napotkane wyzwania implementacyjne i rozwiązania	23
3.10 Ograniczenia środowisk iOS/macOS	23
4 Testowanie i weryfikacja	24
4.1 Metodologia testowania	24
4.2 Scenariusze testowe	24
4.3 Analiza wydajności i zużycia zasobów	24
5 Podsumowanie i kierunki rozwoju	25
5.1 Osiągnięte cele	25
5.2 Ocena spełnienia wymagań	25
5.3 Możliwości dalszej rozbudowy	25
Spis literatury	26
Artykuły	26
Źródła internetowe i inne	26

Wstęp

Tu będzie dopiero treść :)

1 Zakres teoretyczny

1.1 Systemy współdzielenia dokumentów w czasie rzeczywistym

Budując rozwiązania związane z równoczesnym tworzeniem i modyfikacją tekstu przez więcej niż jednego użytkownika, musimy rozważyć wyzwania napotykane w aktualizacji tworzonego dokumentu, gdzie każdy klient posiada lokalną kopię i nanosi na nie własne zmiany, ale też w międzyczasie musimy nanieść zmiany od pozostałych klientów. W takim systemie mówimy wtedy o zbieżności danych[1] - czyli zapewnieniu tego samego stanu między każdym klientem. W przypadku edycji tekstu skupię się na ewentualnej zbieżności, która uwzględnia posiadanie rozbieżnych kopii tego samego źródła danych u każdego z klientów przez pewien czas. Dopiero gdy zostanie zakończona edycja tekstu, zmiany zostają propagowane i nanoszone do pozostałych klientów. Finalnie każdy klient po czasie posiada identyczną kopię dokumentu. Ze strony doświadczeń użytkownika jest to skuteczna strategia ze względu na możliwość zapewnienia płynności interfejsu graficznego oraz z pomocą złożonych mechanizmów umożliwia rozwiązywanie konfliktów między kopiami.

Wspomniany model nie jest bez wad. Największym problemem jest istnienie konfliktów, których rozwiązanie klienci muszą ustalić za pomocą dodatkowych strategii. Najczęściej wykorzystywaną jest Last-Write-Wins (LWW). Rozstrzyga ona konflikty poprzez nanoszenie tylko tej zmiany, która jest uznawana jako ostatnia w kolejności zbioru konfliktujących operacji. Ustalanie kolejności nie jest jasno tutaj zdefiniowane. W systemach baz danych takich jak Cassandra[9] oraz SQL Server P2P[10] każdy zapis otrzymuje własny znacznik czasowy, na podstawie którego wybierany jest najmłodszy wpis i nim nadpisywane są zmiany w źródle danych. Zmiany ze starszymi znacznikami są porzucane. Zauważalną wadą LWW jest wysokie ryzyko utraty danych w czasie nanoszenia zmian, ponieważ wszystkie konfliktujące starsze zmiany nie są brane pod uwagę.

W przypadku tekstu jako typu danych, istnieje specjalny wariant ewentualnej zbieżności - silna ewentualna zbieżność. Ten model wykorzystuje specjalne struk-

tury danych, które zapewniają bezkonfliktowe nanoszenie zmian, a ich skuteczność opiera się na matematycznych dowodach[2].

Istnieje również model silnej zbieżności, gdzie każdy z klientów musi mieć tą samą kopię danych przez cały czas pracy systemu. Ze względu opóźnienia i potrzebę mechanizmu blokady klientów przed wprowadzaniem zmian, ten model został porzucony w dzisiejszych systemach, ponieważ wspomniane problemy skutkowały gorszą użytecznością w porównaniu do systemów wykorzystujących ewentualną zbieżność. Istnieją jednak prace, które wskazują na istnienie systemów opartych o model silnej zbieżności, które osiągają bardzo niskie czasy opóźnienia, co redukuje problem używalności takiego rozwiązania w rzeczywistych zastosowaniach.[11]

1.2 Algorytmy zapewniające silną ewentualną zbieżność

Podejście do rozwiązania problemu synchronizacji stanów między każdym klientem wymaga zaprojektowania algorytmu od podstaw skupionego na tym zagadnieniu. Przedstawię dwa najczęściej wykorzystywane algorytmy rozwiązujące opisany problem.

Operational Transformation (OT) polega na zamianie każdej wykonanej operacji na kodowalny obiekt, który może być propagowany i nanoszony na kopie w innych replikach. Większość znanych implementacji zakłada też, że istnieje scentralizowany serwer określający kolejność każdej operacji zgłaszanej przez wszystkich klientów i dystrybuujący wspomniane zmiany do pozostałych replik. W przypadku wystąpienia konfliktu, operacje są sortowane przez serwer, a następnie każda z nich jest transformowana tak, by uwzględnić zmianę poprzedniej w kolejności operacji. Przykładowymi propozycjami algorytmów OT, których skuteczność nie została obalona dowodami, są: Jupiter[3], SOCT3/4[4] oraz TTF[5]. Jupiter oraz SOCT3/4 wymagają wcześniej wspomnianego scentralizowanego serwera. Google Docs opiera się na Operational Transformation[12]. Ze względu na złożoność implementacji większość systemów nie korzysta dziś z tych algorytmów do synchronizacji danych, gdzie przykładem takiej decyzji jest Figma[13].

Następcą Operational Transformation są bezkonfliktowe replikowane typy danych (Conflict-free replicated data types - CRDT). Pozwalają one na nanoszenie zmian na własne kopie przez klientów, bez potrzeby uzgadniania tego z innymi klientami. Tak jak każdy algorytm ewentualnej zbieżności, pozwala na tymczasową rozbieżność między replikami, ale po otrzymaniu wszystkich zmian przez każdego klienta, dane zawsze pozostają zbieżne. Przykładowymi algorytmami są RGA[6], WOOT[7] oraz TreeDoc[8].

1.3 Architektura Peer-To-Peer w środowiskach mobilnych

Współcześnie smartfony i podobne urządzenia mobilne posiadają moduły wspierające różne standardy komunikacji bezprzewodowej. Najpopularniejszymi z nich są Bluetooth, Bluetooth Low Energy (BLE), Wi-Fi (IEEE 802.11) oraz GSM (Global System for Mobile Communications). Część z nich, np. Bluetooth, jest zaprojektowana tylko pod komunikację na niewielkie odległości (poniżej 100 metrów). Wi-Fi w nowoczesnych systemach jest rozszerzane o wsparcie protokołu Wi-Fi Direct, który umożliwia komunikację Peer-To-Peer z pomocą Wi-Fi, które najczęściej samo w sobie służy do komunikacji z innymi urządzeniami, ale wykorzystując do tego urządzenia infrastruktury sieciowej - bezprzewodowe punkty dostępu (wireless access points).

1.4 Architektura Peer-To-Peer w systemach iOS, iPadOS, macOS, tvOS.

Systemy operacyjne iOS, iPadOS, tvOS oraz macOS są rozwijane przez firmę Apple Incorporated i instalowane wyłącznie na urządzeniach przez nią produkowanych - iPhone'y, iPady, Apple TV, Macintoshe. Współcześnie wszystkie sprzedawane modele zawierają moduły oferujące Bluetooth i Wi-Fi. Dla programistów przygotowane są specjalne biblioteki do bezpośredniego wykorzystania tych technologii, jednak ich użycie wymaga specjalnych certyfikatów wraz z wyjaśnieniem ich wykorzystania, które jest weryfikowane w czasie przygotowania do dystrybucji oprogramowania wykonywanego wspomniane technologie. Ze względu na wyraźną potrzebę zapewnienia bezpieczeństwa transportowanych danych, zalecanym przez Apple jest korzystanie nie z bezpośrednich narzędzi do komunikacji z wykorzystaniem Bluetooth czy Wi-Fi, ale z bibliotek, które oferują transport danych do pobliskich urządzeń. Są one abstrakcją na wcześniej wspomniane protokoły komunikacji bezprzewodowej jak i przewodowej, gdzie po stronie programisty pozostaje jedynie obsłużyć parowanie się z pobliskimi urządzeniami i przygotować kodowalne obiekty do transportu. Dziś możemy wyróżnić dwie takie biblioteki oferowane przez Apple - Multipeer Connectivity oraz Network.

Multipeer Connectivity to framework zapewniający komunikację oraz odkrywanie pobliskich urządzeń. Do komunikacji wykorzystuje protokoły komunikacji bezprzewodowej - w tym dostępne sieci Wi-Fi, Bluetooth oraz w przypadku Macintoshy oraz Apple TV - protokół komunikacji przewodowej - Ethernet. Interfejs dostarczany przez bibliotekę umożliwia transport danych w postaci niewielkich wiadomości, strumieniowania danych oraz transportu plików.

Architektura Multipeer Connectivity opiera się na sesjach - połączeniach między użytkownikami w pobliżu. Sesje są reprezentowane przez obiekty typu MCTSession. Takie obiekty są tworzone przez programistę w dowolnym momencie oraz u nowo połączonych klienta od razu po dołączeniu do sesji do której został zaproszony,

jednocześnie każda z kopii sesji zawiera informację o wszystkich połączonych klientach. Do odkrywania innych użytkowników używany jest protokół Bonjour.

Urządzenie ogłaszające u pobliskich urządzeń dostępną sesję jest nazywany odkrywcą. Jego rolę reprezentuje obiekt `MCNearbyServiceBrowser`. Możemy również wykorzystać obiekt `MCBrowserViewController` który zapewni interfejs graficzny, dzięki któremu użytkownik będzie mógł zaprosić klientów do swojej sesji. Urządzenie poszukujące dostępnych sesji jest nazywany nadawcą. Jego reprezentantem jest obiekt `MCNearbyServiceAdvertiser`.

Obiektem identyfikującym urządzenie między sesjami jest `MCPeerID`, zawierający dane unikalne dla klienta w zakresie danej sesji.

W czasie odkrywania urządzeń w pobliżu, mamy jedynie dostęp do ograniczonej ilości informacji, które są dystrybuowane przez nadawców. Dopiero po wysłaniu zaproszenia przez odkrywcę, a następnie akceptacji przez nadawcę, możemy wykorzystać w pełni interfejs Multipeer Connectivity do komunikacji.

Network jest frameworkiem skupionym na ogólnej interakcji z połączeniami sieciowymi. Oprócz lokalnego dostępu do innych urządzeń, daje możliwość wykorzystania całej dostępnej infrastruktury sieciowej. Apple w przypadku tworzenia aplikacji w oparciu o połączenia peer to peer, zaleca łączenie Network z biblioteką `DeviceDiscoveryUI`, która zapewnia dodatkowy interfejs graficzny dla użytkownika aplikacji implementujący proces parowania. W ramach tego procesu Apple pozwala na wybór protokołu do komunikacji w czasie parowania. Mamy do wyboru Bonjour - starszy, należący do Apple protokół - oraz Wi-Fi Aware - otwarty, międzyplatformowy standard. `DeviceDiscoveryUI` jest nową biblioteką, dostępną od systemów operacyjnych wydanych w 2025 roku - iOS 26, iPadOS 26, macOS 26. Wyjątkiem jest wsparcie dla tvOS, tutaj Apple oferuje wsparcie od wersji 16, wydanej w 2022 roku. Wi-Fi Aware jest również dostępny dopiero od systemów wydanych w 2025 roku. Network wraz z własnym protokołem ramkowania jest dostępny od wersji systemów operacyjnych wydanych w 2019 roku.

W przypadku oparcia aplikacji komunikującej się peer to peer z innymi urządzeniami o Network, musimy przygotować własny protokół ramkowania `NWProtocolFramerImplementation`. W nim powinniśmy obsłużyć wszystkie wydarzenia związane z poprawnym zarządzaniem połączeniem - inicjację, powrót z uśpienia, zatrzymanie, proces czyszczenia przez dealokację. Do nasłuchiwania na przychodzące połączenia wykorzystalibyśmy obiekt `NWListener`. Połączenie z innymi urządzeniami byłoby reprezentowane przez obiekt `NWConnection`, a jeśli chcielibyśmy zaimplementować własny proces odkrywania i negocjacji połączenia z pobliskimi urządzeniami, musielibyśmy dodatkowo wykorzystać obiekt `NWBrowser`.

1.5 Istniejące rozwiązania i ich problemy

Dwoma najpopularniejszymi aplikacjami do współtworzenia notatek w czasie rzeczywistym są Google Docs i Microsoft Word Online. Ich zakres funkcjonalności jest bardzo podobny - złożone formatowanie tekstu; możliwość dodawania załączników, tabel; historia wersji dokumentu; obecność publicznego API; wykorzystanie nowoczesnych standardów szyfrowania komunikacji; eksport dokumentu w postaci innego rodzaju pliku oraz wymagają konta do możliwości ich użycia. Google Docs korzysta ze swojego algorytmu Operational Transformation do synchronizacji zmian między użytkownikami, sposób przechowywania dokumentów nie jest jasny, nie wiemy w jakiej postaci są one przechowywane na serwerach Google, a dostęp do aplikacji jest darmowy. Microsoft Word Online nie udostępnia publicznie informacji o stosowanym podejściu do synchronizacji danych, dokumenty są przechowywane w postaci plików docx, a dostęp do aplikacji jest również darmowy.

Głównym problemem większości dzisiejszych narzędzi do tworzenia dokumentów jest uzależnienie od dostawcy. Każda z wymienionych aplikacji, jak i wiele innych dostępnych na rynku do skorzystania ze swoich usług wymaga założenia konta na platformie dostawcy, a w czasie współpracy wymagany jest ciągły dostęp do Internetu. W momencie gdy pracujemy nad jednym dokumentem z innymi użytkownikami w tym samym pomieszczeniu i nasze urządzenia są podłączone do tej samej sieci, opóźnienie w nanoszeniu zmian między użytkownikami zawsze uwzględnia dostęp do scentralizowanych serwerów. W momencie gdy zostaniemy odcięci od dostępu do Internetu, ale nadal będąc w tej samej sieci lokalnej, tracimy możliwość dalszej pracy. Szczególnie widać to w przypadku Google Docs, którego algorytm synchronizujący narzuca obecność serwera wybierającego kolejność nanoszenia zmian w dokumentach.

2 Wymagania systemowe

W poniższym rozdziale opiszę specyfikację wymagań dla projektowanej aplikacji do współtworzenia notatek w architekturze peer to peer. Celem jest określenie zakresu funkcjonalności oraz ograniczeń technologicznych, których będę się trzymać w ramach proponowanej implementacji. Wyodrębnię dwóch aktorów:

- Użytkownika - osoba zarządzająca notatkami przez interfejs graficzny zaimplementowany w ramach implementacji,
- Klient - instancja implementowanej aplikacji na urządzeniu fizycznym.

Same wymagania zostały podzielone odpowiednio na wymagania funkcjonalne i нефункционалне.

2.1 Wymagania funkcjonalne

- System musi umożliwiać rozgłaszanie swojej obecności w sieci lokalnej, by móc zostać wykrytym przez innych klientów.
- System musi umożliwiać wyszukiwanie innych aktywnych klientów znajdujących się w zakresie lokalnego otoczenia oraz lokalnej sieci.
- System musi pozwalać na wysyłanie zaproszeń do połączenia się z wykrytymi klientami.
- System musi na bieżąco wyświetlać listę aktualnie połączonych klientów.
- System musi umożliwiać utworzenie nowej notatki tekstowej.
- System musi umożliwiać edycję treści istniejącej notatki.
- System musi umożliwiać trwałe usunięcie notatki.
- System musi automatycznie tworzyć unikalny identyfikator oraz znacznik czasowy dla nowych notatek.
- System musi pozwalać na zdefiniowanie tytułu notatki, będącego niezależnym parametrem od treści notatki.
- System musi enkodować strukturę danych notatki do formatu umożliwiającego przesył notatki do połączonych klientów.
- System musi dekodować otrzymane zakodowane dane o notatce i przekształcić je w natywny obiekt reprezentujący notatkę w systemie.
- System musi automatycznie rozsyłać zaktualizowaną treść notatki do wszystkich aktualnie połączonych klientów w jak najkrótszym czasie od wykrycia zmian.

- System musi nadpisać istniejącą notatkę nowo otrzymaną kopią, gdy obie posiadają ten sam identyfikator, ale otrzymana kopia zawiera nowszy znacznik czasowy.
- System musi zapisywać wszystkie notatki w trwałej pamięci urządzenia.
- System musi odświeżać interfejs z listą notatek w czasie rzeczywistym.

2.2 Wymagania niefunkcjonalne

- System musi wspierać urządzenia mobilne firmy Apple z zainstalowanymi systemami operacyjnymi iOS lub iPadOS w wersji 18.0 lub wyższej.
- Kod źródłowy powinien być napisany w języku Swift z wykorzystaniem deklaratywnego frameworka do budowy interfejsów graficznych - SwiftUI.
- Operacje zapisu do plików oraz procesy sieciowe muszą być wykonywane poza głównym wątkiem - wątkiem obsługującym interfejs graficzny aplikacji.
- Czas propagacji zmian w notatce do innego połączonego klienta nie powinien wynosić więcej niż 1 sekunda.
- System musi obsłużyć przypadek zerwania połączenia, bez uszkodzenia notatki oraz rzucania wyjątków uniemożliwiających dalsze funkcjonowanie systemu.
- System musi zapewnić szyfrowaną komunikację między klientami.
- Interfejs użytkownika musi być responsywny i dostosowywać się do różnych rozmiarów ekranów i ich orientacji w zakresie urządzeń tworzonych przez Apple.
- Wygląd aplikacji powinien spełniać oficjalne wytyczne projektowe Apple Human Interface Guidelines.
- System powinien automatycznie dostosowywać paletę kolorów interfejsu graficznego do aktualnie ustawionego motywu systemowego.

3 Implementacja

3.1 Projekt architektury systemu

Przygotowana implementacja bazuje na architekturze Model-View. Jest to podejście, gdzie cała logika biznesowa jest zawarta w modelach, które bezpośrednio są przekazywane do warstwy prezentacji (View). Na tej warstwie jest wykonywane odpowiednie formatowanie danych, gdzie też brane pod uwagę są preferencje zapisu i językowe użytkownika. Model-View to uproszczony wariant popularnej w aplikacjach mobilnych architektury Model-View-ViewModel (MVVM), gdzie ViewModel jest warstwą zajmującą się przekształcaniem modeli biznesowych na gotowe do prezentacji obiekty. Warstwa View wtedy zajmuje się przede wszystkim definiowaniem struktury interfejsu użytkownika oraz logiką związaną z dostępnością (wsparcie dla funkcjonalności czytników ekranów, skalowaniem interfejsu). Przygotowana implementacja nie zawiera złożonej logiki prezentacji ani rozbudowanego graficznego interfejsu użytkownika, więc w celu uproszczenia kodu zdecydowałem się na porzucenie użycia warstwy ViewModelu.

Warstwa prezentacji aplikacji opisująca początkowy interfejs użytkownika jest zaimplementowana w `AllNotesScreen`. Ekran jest widoczny dla użytkownika, gdy ustawił swoją nazwę użytkownika w sieci peer-to-peer. Składa się z listy podzielonej na dwie sekcje - notatek których użytkownik jest autorem, oraz notatek do których użytkownik został zaproszony. Nad listą znajduje się przycisk stworzenia nowej notatki. Naciśnięcie na którąkolwiek z istniejących notatek prowadzi do otwarcia jej zawartości. W zależności od tego, czy użytkownik jest właścicielem danej notatki czy współtwórcą, interfejsem do edycji tej notatki są odpowiednio obiekty `NoteEditorScreen` oraz `SharedNoteEditor`. Z ekranu `NoteEditorScreen` możemy przejść do `ManageMembersScreen`, który jest listą użytkowników zaproszonych i możliwych do zaproszenia do edytowania aktualnej notatki.

3.2 Model danych

Podstawowym obiektem reprezentującym notatkę w systemie plików jest `Note`. Składa się on z URL (Universal Resource Locator), czyli ścieżki prowadzącej do pełnej zawartości notatki użytkownika, która jest przechowywana w pliku tekstowym o rozszerzeniu `txt`. Dodatkowo przechowujemy w tym obiekcie jeszcze `name`, które pełni funkcję ułatwionego dostępu do przyjaznej nazwy, jednocześnie będąc nazwą pliku w lokalnym systemie plików oraz przyjazną nazwą prezentowaną w liście notatek do których użytkownik został zaproszony. Obiekt implementuje protokół `Identifiable`, który gwarantuje możliwość identyfikacji obiektu w zbiorze zawierającym wiele jego instancji, np. w tablicy. Ta cecha jest wymagana i wykorzystywana przez `SwiftUI`, by móc rozróżniać rodzaje zmian na komponentach interfejsów graficznych zawierających wiele kopii takiego obiektu, np. `List` lub `ForEach`. Umożliwia to dokonanie decyzji co do sposobu animacji zmian na ekranie użytkownika, ponieważ framework będzie mógł rozróżnić usunięcie i wstawienie nowego obiektu, od zmiany parametrów tej samej instancji.

```
struct Note: Identifiable {
    var id: URL { path }

    let name: String
    let path: URL
}
```

`NoteMessage` to obiekt zawierający zawartość notatki, którą wysyłamy do użytkowników, którzy przyjęli zaproszenie do edycji notatki. Znajdziemy w niej pola `SenderID`, które jest identyfikatorem właściciela notatki oraz `content`, czyli właściwą zawartość notatki. Jest ona również używana do wysyłania każdej aktualizacji do wszystkich użytkowników. Obiekt implementuje protokół `Codable`, który jest uniwersalnym interfejsem kodowania danych. Daje to nam wbudowane wsparcie kodowania do formatów JSON i XML. Mamy możliwość również pisania własnych koderów, które umożliwią zamianę obiektów implementujących `Codable` do wybranych przez nas, innych formatów danych.

```
struct NoteMessage: Codable {
    let senderID: String
    let content: String
}
```

Do reprezentacji zaproszeń stworzyłem obiekt `NoteInvitation`. Przechowuje on informacje potrzebne do obsługi całego procesu zaproszeń między użytkownikami systemu. Składa się z `invitatorID`, który jest wyspecjalizowanym obiektem frameworku `MultipeerConnectivity` i umożliwia identyfikację użytkownika w czasie komunikacji peer to peer. `note` to obiekt reprezentujący notatkę, którą

otrzymujemy w zaproszeniu. Składa się on z tytułu i treści, które były aktualne w momencie zapraszania użytkownika. Ostatnim i jednocześnie prywatnym parametrem jest `invitationHandler`, który jest typem funkcji przyjmującej wartość boolowską (prawda/fałsz). Jest on wykorzystywany do wstrzyknięcia logiki akceptacji notatki, która wymaga kilku operacji w logice klasy obsługującej przyjmowanie zaproszeń. Dzięki takiemu podejściu tworzymy luźną zależność do skomplikowanego obiektu na dalszych etapach systemu. `NoteInvitation` zawiera również metody `accept()` oraz `decline()`, które odpowiadają za wykonanie akcji zaproszenia, które pod spodem odpowiednio używają parametru `invitationHandler`, który można wykonać tylko raz, ze względu na specyfikę `Multipeer Connectivity`, a następnie usuwamy go z pamięci.

```
struct NoteInvitation: Identifiable {
  struct NoteContent: Codable {
    let title: String
    let noteSnapshot: String
  }

  var id: MCPeerID { invitorID }
  var noteName: String { note.title }
  let invitorID: MCPeerID
  let note: NoteContent
  private var invitationHandler: ((Bool) -> Void)?

  init(
    invitorID: MCPeerID,
    note: NoteContent,
    invitationHandler: ((Bool) -> Void)? = nil
  ) {
    self.invitorID = invitorID
    self.note = note
    self.invitationHandler = invitationHandler
  }

  mutating func accept() {
    invitationHandler?(true)
    invitationHandler = nil
  }

  mutating func decline() {
    invitationHandler?(false)
    invitationHandler = nil
  }
}
```

Ostatnim obiektem jest struktura `Peer`, która jest opisem aktualnego stanu połączenia wykrytego innej instancji systemu w pobliżu użytkownika. Składa się z wartości enumerowanej opisującej stan połączenia oraz identyfikatorem

użytkownika w sieci peer to peer. Implementują ona protokół `Identifiable`, by móc zostać poprawnie użyta do rysowania listy dostępnych klientów w pobliżu użytkownika.

```
struct Peer: Identifiable {
    enum ConnectionState {
        case available
        case joined
        case rejected
        case invitationPending
    }

    var id: String { mcPeer.displayName }
    let mcPeer: MCPeerID
    var state: ConnectionState
}
```

3.3 Warstwa sieciowa i komunikacja P2P

Całość komunikacji między urządzeniami odbywa się z wykorzystaniem frameworka Multipeer Connectivity. Klient twórcy notatki pełni rolę serwera, a pozostali użytkownicy, po uprzednim zaproszeniu, mogą dołączyć do edycji notatki, wysyłać swoje zmiany jak i odbierać zmiany, które dystrybuuje serwer.

3.4 Odkrywanie innych urządzeń

Obiekt reprezentujący serwer został nazwany `NoteEditingSessionServer`, który dziedziczy właściwości po klasie `NSObject`, która jest uniwersalną implementacją wielu zachowań, które są wymagane od frameworków udostępnianych przez Apple, które zostały napisane w języku Objective-C. Jego konstruktor w przyjmowanych argumentach oczekuje tylko obiektu `OwnPeer`, który będzie wykorzystywany do identyfikacji instancji aplikacji u innych klientów. Sama implementacja konstruktora tworzy nową sesję `MCSession`; obiekt `MCNearbyServiceBrowser`, który odpowiada za wykrywanie pobliskich klientów. Finalnie przypisuje referencję do samego siebie jako parametr `delegate` dla utworzonych `MCSession` i `MCNearbyServiceBrowser`. Pozwala nam to zaimplementować metody, które będą wykorzystywane wewnątrz tych obiektów do komunikacji z innymi użytkownikami. Protokoły delegujące dla wspomnianych obiektów nazywają się odpowiednio `MCSessionDelegate` oraz `MCNearbyServiceBrowserDelegate`. Moja implementacja tych protokołów zostanie przedstawiona w dalszej części pracy.

```
init(peer: OwnPeer) {
    ownPeer = peer
    browser = .init(peer: peer.peer, serviceType: "peered")
}
```

```
    session = .init(peer: peer.peer, securityIdentity: nil,  
encryptionPreference: .required)  
    super.init() // wykonuje pozostałą część  
    browser.delegate = self //  
    session.delegate = self  
}
```

W momencie, gdy autor notatki otworzy ekran edycji, wykonuje się metoda `startServer()`, która wywołuje metodę `startBrowsingForPeers()` obiektu `MCNearbyServiceBrowser`. Opuszczenie ekranu edycji wywołuje metodę `stopServer()`, która wywołuje analogiczną metodę `stopBrowsingForPeers()` oraz zatrzymuje sesję poprzez wywołanie metody `disconnect()` obiektu `MCSession`.

```
func startServer() {  
    browser.startBrowsingForPeers()  
}  
  
func stopServer() {  
    browser.stopBrowsingForPeers()  
    session.disconnect()  
}
```

Obiekt `browser` w momencie wykrycia nowego użytkownika w pobliżu, wywołuje naszą metodę o nazwie `browser()`, która przyjmuje wszystkie potrzebne informacje o znalezionym użytkowniku. Implementacja mojego systemu następnie upewnia się czy odkryty użytkownik nie jest jednocześnie autorem notatki, co jest znanym błędem w Multipeer Connectivity. Następnie po udanej weryfikacji dodajemy nowy obiekt dostępnego użytkownika do tablicy na podstawie której jest budowany interfejs z listą dostępnych użytkowników.

```
func browser(  
    _ browser: MCNearbyServiceBrowser,  
    foundPeer peerID: MCPeerID,  
    withDiscoveryInfo info: [String: String]?  
    ) {  
    guard !visiblePeers.contains(where: { $0.mcPeer == peerID }) &&  
peerID.displayName != ownPeer.peer.displayName else { return }  
    DispatchQueue.main.async {  
        self.visiblePeers.append(Peer(mcPeer: peerID,  
state: .available))  
    }  
}
```

W sytuacji gdy użytkownik straci połączenie z połączonym klientem, następuje usunięcie jego identyfikatora z listy, poprzez wywołanie metody o takiej samej nazwie, ale z parametrami wskazującymi na scenariusz zgubienia klienta.

```

func browser(_ browser: MCNearbyServiceBrowser, lostPeer peerID:
MCPeerID) {
    DispatchQueue.main.async {
        guard let peerIdx = self.visiblePeers.firstIndex(where:
{ $0.mcPeer == peerID }) else { return }
        self.visiblePeers.remove(at: peerIdx)
    }
}

```

Każda modyfikacja obiektów klasy w tym wypadku musi zostać wywołana na tym samym wątku, ponieważ Multipeer Connectivity nie gwarantuje, że kod będzie się wykonał zawsze na tym samym wątku. Wybrałem wątek główny, ze względu na bezpośrednie użycie właściwości klasy wewnątrz obiektów odpowiedzialnych za budowę interfejsu użytkownika.

O stanie połączenia z innymi klientami mój system jest informowany przez wykonanie metody `session` z parametrami zawierającymi informację o stanie, który jest reprezentowany przez typ enumeracji, obiekt sesji oraz identyfikator klienta, których ten stan połączenia dotyczy. Na podstawie tych argumentów, aplikacja aktualizuje tablicę `visiblePeers`.

```

func session(
    _ session: MCSession,
    peer peerID: MCPeerID,
    didChange state: MCSessionState
) {
    DispatchQueue.main.async {
        guard let idx = self.visiblePeers.firstIndex(where: { $0.mcPeer ==
peerID }) else { return }
        switch state {
        case .connected:
            self.visiblePeers[idx].state = .joined
        case .notConnected:
            let currentState = self.visiblePeers[idx].state
            if currentState == .invitationPending || currentState == .joined
{
                self.visiblePeers[idx].state = .rejected
            }
        default:
            break
        }
    }
}

```

Przechodząc do implementacji klienta, całość jest reprezentowana przez obiekt `NoteEditingSessionClient`. Jego konstruktor przyjmuje tylko identyfikator użytkownika, który jest typem `MCPeerID`, a implementacja obejmuje również stworzenie instancji `MCSession` do wykorzystania w trakcie połączenia z serwerem

oraz instancji `MCNearbyServiceAdvertiser`, która propaguje informacje o kliencie do wszystkich innych klientów w pobliżu. Do obu obiektów przypisujemy obiekt delegujący, który będzie właśnie utworzoną instancją `NoteEditingSessionClient`. Dla `MCNearbyServiceAdvertiser` obiekt delegującego musi implementować protokół `MCNearbyServiceAdvertiserDelegate`.

```
init(peer: MCPeerID) {
    ownPeer = peer
    session = MCSession(
        peer: peer,
        securityIdentity: nil,
        encryptionPreference: .required
    )
    advertiser = MCNearbyServiceAdvertiser(
        peer: peer,
        discoveryInfo: [:],
        serviceType: "peered"
    )
    super.init()
    advertiser.delegate = self
    session.delegate = self
}
```

Instancja `NoteEditingSessionClient` jest tworzona już przy pierwszym uruchomieniu aplikacji. Po utworzeniu przez użytkownika przyjaznej nazwy, która będzie używana do identyfikacji, w tle wywoływana jest metoda `startBrowsingForNotes()`, która wywołuje `startAdvertisingPeer()` obiektu `advertiser`. Przed rozpoczęciem nasłuchiwanie aplikacja wywołuje również `stopBrowsingForNotes()`, by zatrzymać nasłuchiwanie, jeśli wcześniej było ono rozpoczęte, oraz zatrzymuje działanie obiektu `MCSession`, by zamknąć ewentualnie istniejącą sesję edycji notatki.

```
func startBrowsingForNotes() {
    advertiser.startAdvertisingPeer()
}

func stopBrowsingForNotes() {
    advertiser.stopAdvertisingPeer()
    session.disconnect()
}
```

Jedyna metoda wymagana przez protokół `MCNearbyServiceAdvertiserDelegate` nazywa się `advertises()` i w argumentach przyjmuje informację o otrzymanym zaproszeniu i metadanych jakie to zaproszenie zawierało. Aplikacja próbuje zdekodować migawkę notatki, a następnie konstruuje obiekt `NoteInvitation` i umieszcza go w tablicy `invitations`. Umieszczenie w tablicy trzeba wykonać

na głównym wątku, ponieważ, analogicznie jak w wypadku implementacji `MCNearbyServiceBrowserDelegate`, nie mamy gwarancji na jakim wątku będzie wykonywała się ta metoda.

```
func advertiser(
    _ advertiser: MCNearbyServiceAdvertiser,
    didReceiveInvitationFromPeer peerID: MCPeerID,
    withContext context: Data?,
    invitationHandler: @escaping (Bool, MCSession?) -> Void
) {
    guard
        let context,
        let noteContent = try?
JSONDecoder().decode(NoteInvitation.NoteContent.self, from: context)
    else { return }

    DispatchQueue.main.async {
        self.invitations.append(
            .init(
                invitorID: peerID,
                note: noteContent,
                invitationHandler: { [weak self, invitationHandler] accepted
in
                    guard let self else { return }
                    invitationHandler(accepted, self.session)
                    DispatchQueue.main.async {
                        self.invitations.removeAll { $0.id == peerID }
                    }
                }
            )
        )
    }
}
```

3.5 Transportowanie danych

W wysłanym zaproszeniu wysyłamy w metadanych migawkę notatki zawierającą jej tytuł oraz ostatnio dostępną treść. Migawka jest kodowana w formacie JSON bazując na strukturze `NoteContent`. Przykładowym poprawnym zapisem JSON tej struktury jest poniższy przykład

```
{
  "title": "My new note",
  "noteSnapshot": "Lorem Ipsum."
}
```

Po stronie klienta, każda zmiana jest ogłaszana serwerowi poprzez wywołanie metody `send()` obiektu `NoteEditingSessionClient`, która przyjmuje identyfikator użytkownika do której ma wiadomość trafić oraz całą zawartość notatki. Jej implementacja zamienia notatkę wraz z identyfikatorem w typ `NoteMessage`, następnie koduje ją do formatu JSON, finalnie próbuje ją wysłać do określonego użytkownika.

```
func send(note: String, to peer: MCPeerID) {
    let message = NoteMessage(senderID: ownPeer.displayName, content:
note)
    guard let data = try? JSONEncoder().encode(message) else
{ return }
    try? session.send(data, toPeers: [peer], with: .reliable)
}
```

Przykładowy zapis instancji obiektu `NoteMessage` wygląda następująco:

```
{
  "senderID": "User2",
  "content": "Lorem Ipsum Test"
}
```

Po tym jak serwer odbierze wysłaną wiadomość, wywoływana jest metoda `session`, która w argumentach przekazuje zakodowane dane, sesję serwera oraz identyfikator użytkownika, który wysłał załączone dane. Po udanym zdekodowaniu danych, wybieramy wszystkich użytkowników, którzy dołączyli do sesji edycji notatki i wysyłamy do nich kopię otrzymanej wiadomości, a serwer dodatkowo wysyła identyczną kopię do warstwy prezentacji.

```
func session(_ session: MCSession, didReceive data: Data, fromPeer
peerID: MCPeerID) {
    guard let message = try? JSONDecoder().decode(NoteMessage.self,
from: data) else { return }

    let otherPeers = session.connectedPeers.filter { $0 != peerID }
    if !otherPeers.isEmpty {
        try? session.send(data, toPeers: otherPeers, with: .reliable)
    }

    DispatchQueue.main.async {
        self.noteChangesEmitter.send(message)
    }
}
```

3.6 Algorytm rozwiązywania konfliktów

Rozwiązywanie konfliktów jest dokonywane na podstawie strategii Last Writer Wins, gdzie ostatnio otrzymana wiadomość nadpisuje zawartość pozostałych kopii. W przypadku nanoszenia zmian na interfejs użytkownika, wykorzystałem natywną obsługę zmiany tekstu przez systemowy komponent `TextEditor`, który w większości wypadków potrafi sobie poradzić czy prostych podmianach zawartości tekstu.

Zrezygnowałem z implementacji struktur danych CRDT ze względu na bardzo wysoki koszt implementacji, ponieważ oprócz samych struktur, musiałbym napisać własną implementację obsługi podmiany tekstu i aktualizacji kursora w edytorze tekstowym, co znacznie wykracza poza zakres tej pracy.

3.7 Środowisko developerskie i stack technologiczny

Uruchomienie projektu wymaga posiadania komputera Macintosh z systemem operacyjnym macOS w wersji 26.0 (Tahoe) oraz środowisko programistyczne Xcode w wersji 26.0. Do uruchomienia aplikacji potrzebne jest założenie konta Apple ID i zaakceptowania warunków użytkownika konta deweloperskiego, dzięki któremu można wygenerować certyfikat, którego potrzebuje Xcode do zbudowania aplikacji.

Aplikacja wykorzystuje framework SwiftUI do budowania warstwy prezentacji. Jest to deklaratywny, wieloplatformowy framework między innymi dostarczający mechanizmy, które zostały wykorzystane w ramach tej pracy:

- automatycznego odświeżania interfejsu na podstawie nasłuchiwanie na zmiany modeli danych
- automatycznego zapisu prymitywnych danych na dysku
- wstrzykiwania zależności w głąb hierarchii interfejsu
- zaawansowanych wzorców stosowanych w tworzeniu dobrych doświadczeń użytkownika - np. wysuwalne panele (bottom sheets).

Kluczowym frameworkiem w budowaniu tej aplikacji jest Multipeer Connectivity, który zapewniał ułatwioną konfigurację całej komunikacji między pobliskimi użytkownikami. Pomocniczymi bibliotekami są `Foundation` oraz `Combine`.

`Foundation` daje dostęp do złożonych, ale bardzo często wykorzystywanych typów i funkcji w Swifcie - `URL`, `FileManager`, `JSONDecoder`, `JSONEncoder`.

`Combine` jest biblioteką dodającą programowanie reaktywne do Swifta, ale również jako pierwsza zapewniła mechanizmy komunikowania zmian w modelach danych do widoków implementowanych w SwiftUI. Moja praca wykorzystywała przede wszystkim obiekty `PassthroughSubjects`, które służyły za obiekty emitujące każde nowo otrzymane dane od innych użytkowników.

3.8 Interfejs użytkownika

3.9 Napotkane wyzwania implementacyjne i rozwiązania

3.10 Ograniczenia środowisk iOS/macOS

4 Testowanie i weryfikacja

4.1 Metodologia testowania

4.2 Scenariusze testowe

4.3 Analiza wydajności i zużycia zasobów

5 Podsumowanie i kierunki rozwoju

5.1 Osiągnięte cele

5.2 Ocena spełnienia wymagań

5.3 Możliwości dalszej rozbudowy

Spis literatury

Artykuły

- [1] Werner Vogels. „Eventually consistent”. W: 52.1 (2009-01), s. 40–44.
- [2] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan i Alastair R. Beresford. „Verifying strong eventual consistency in distributed systems”. W: 1.OOPSLA (2017-10).
- [3] David A. Nichols, Pavel Curtis, Michael Dixon i John Lamping. „High-latency, low-bandwidth windowing in the Jupiter collaboration system”. W: (1995), s. 111–120.
- [4] Nicolas Vidot, Michelle Cart, Jean Ferrié i Maher Suleiman. „Copies convergence in a distributed real-time collaborative environment”. W: (2000), s. 171–180.
- [5] Gérald Oster, Pascal Molli, Pascal Urso i Abdessamad Imine. „Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems”. W: (2006), s. 1–10.
- [6] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim i Joonwon Lee. „Replicated abstract data types: Building blocks for collaborative applications”. W: 71.3 (2011), s. 354–368.
- [7] Gérald Oster, Pascal Urso, Pascal Molli i Abdessamad Imine. „Data consistency for P2P collaborative editing”. W: (2006), s. 259–268.
- [8] Nuno Preguica, Joan Manuel Marques, Marc Shapiro i Mihai Letia. „A Commutative Replicated Data Type for Cooperative Editing”. W: (2009), s. 395–403.

Źródła internetowe i inne

- [9] Apache. *Data versioning - Apache Cassandra Documentation*. URL: <https://cassandra.apache.org/doc/latest/cassandra/architecture/dynamo.html#data-versioning>.
- [10] Microsoft. *Peer To Peer Transactional Replication - Microsoft SQL Server documentation*. URL: <https://learn.microsoft.com/en-us/sql/relational-databases/replication/transactional/peer-to-peer-transactional-replication?view=sql-server-ver17>.
- [11] Weixin Yu i Kaylee Xie. *A Paxos-Based Strongly Consistent Live Document Editor*. URL: https://www.scs.stanford.edu/26wi-cs244c/proj/live_document_editor.pdf.
- [12] John Day-Richter. *What's Different About New Google Docs*. URL: <https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html>.
- [13] Evan Wallace. *How Figma's Multiplayer Technology Works*. URL: <https://www.figma.com/blog/how-figmas-multiplayer-technology-works/>.