Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Instytut Informatyki

Praca dyplomowa inżynierska

na kierunku Informatyka
w specjalności Inżynieria Oprogramowania

Narzędzie do synchronizacji zawartości katalogów

Bartłomiej Moroz
Numer albumu 304076

promotor
dr inż. Rafał Biedrzycki

WARSZAWA 2023

Narzędzie do synchronizacji zawartości katalogów

Streszczenie. Świat cyfrowy ciągle się rozwija, a wraz z nim kumulowane i wymieniane dane. W konsekwencji, utrzymywanie replik na różnych maszynach pozostaje problemem pomimo ulepszeń warstwy sprzętowej. Dlatego też, choć problem synchronizacji plików jest rozważany od lat 90. XX wieku, nadal powstają nowe rozwiązania. W ramach tej pracy inżynierskiej stworzyłem własne narzędzie do synchronizacji plików pomiędzy maszynami przeznaczone na komputery z systemem Ubuntu, pozwalające na bezpieczną propagację zmian kontrolowaną przez użytkownika oraz możliwość rozwiązywania konfliktów. Praca opisuje proces projektowania, implementacji i testowania własnego programu na bazie analizy cech dostępnych narzędzi do synchronizacji online.

Słowa kluczowe: synchronizacja plików, aplikacja okienkowa, C++

Tool for directory contents synchronization

Abstract. The digital world is constantly growing, and with it, stored and exchanged data. As a result, maintaining replicas on different machines remains an issue despite hardware upgrades. New solutions continue to appear, even though the topic of file synchronization has existed since the 1990s. As part of my work, I created a remote file synchronization tool for machines running Ubuntu OS that allows users to propagate changes in a safe, curated manner and to resolve conflicts. This paper describes the design, development and testing process of my tool based on an analysis of existing online file synchronization software.

Keywords: file synchronization, desktop application, C++

Spis treści

1. Wstęp

1.1. Cel i zakres pracy

W ramach pracy inżynierskiej stworzyłem narzędzie na komputery z systemem operacyjnym Ubuntu pozwalające na synchronizację zawartości wybranych katalogów. Celem programu jest skrócenie czasu potrzebnego do wymiany plików między maszynami, jednocześnie dając użytkownikowi kontrolę nad tym, które pliki będą uczestniczyły w procesie. Narzędzie pozwala m.in. na:

- Tworzenie i wybieranie konfiguracji (profili);
- Uwierzytelnianie użytkownika poprzez protokół SSH;
- Synchronizację tylko wybranych plików;
- Ignorowanie plików na podstawie nazwy;
- Uruchamianie zewnętrznych narzędzi do rozwiązywania konfliktów;

Oprócz tego, praca obejmuje przegląd istniejących narzędzi, identyfikujący ich rodzaj, wady, zalety, możliwości.

1.2. Zawartość pracy

Praca składa się ze wstępu, trzech rozdziałów i podsumowania. Rozdział pierwszy (numer 2) zawiera opis problemu synchronizacji plików, analizę najważniejszych cech narzędzi oraz przegląd wybranych dostępnych narzędzi. Następny rozdział (numer 3) opisuje implementacje własnego narzędzia, zaczynając od określenia wymagań i założeń projektowych. Na podstawie założeń wybrałem narzędzia programistyczne (język, biblioteki, inne) za pomocą których program będzie tworzony. Opisana jest architektura projektu, użyte algorytmy, sposób działania, interakcja z użytkownikiem. Kolejny rozdział (numer 4) zawiera badania poprawności i wydajności (warunki badań oraz wyniki) stworzonego narzędzia i perspektywy dalszego rozwoju. Praca zakończona jest podsumowaniem zawierającym końcowe wnioski z pracy.

2. Synchronizacja plików

2.1. Opis problemu

Problem synchronizacji plików można zdefiniować w następujący sposób: Istnieje lokalizacja źródłowa A i lokalizacja docelowa B (lub wiele lokalizacji). Należy doprowadzić lokalizacje do wspólnego, identycznego stanu tak, aby znajdowały się w nim tylko najnowsze wersje danych (gdzie najnowsze wersje nie muszą być tylko w lokalizacji źródłowej A)[1].

Rozwiązanie naiwne, tj. przesłanie kopii wszystkich plików z A do B i *vice versa* nie jest rozwiązaniem akceptowalnym ze względu na koszt przesyłu dużej ilości danych [2][3] oraz ryzyko nadpisania zawartości plików, które wprawdzie znajdują się na obu maszynach, lecz nie zawierają w całości tych samych danych. Istnieje więc potrzeba odkrycia zmian, czyli porównania zawartości replik. Ręczne decydowanie, które pliki należy przesłać, jest czasochłonne dla większych zestawów plików (lub zupełnie niepraktyczne, jeśli liczba plików jest rzędu setek lub więcej) i obarczone ryzykiem popełnienia błędu ludzkiego. Jest to proces, który można zautomatyzować.

Żeby stwierdzić, które dane są najnowsze, musimy je porównać z wcześniejszą wersją. Można do tego wykorzystać czas modyfikacji dostępny w ramach metadanych plików[1]. Najprostsze narzędzie mogłoby porównywać czasy modyfikacji, lecz taki sposób wykrywania można łatwo „oszukać" umyślnie lub nieumyślnie[4] i może przestać działać, jeżeli istnieją problemy z reprezentacją czasu w systemie, np. "problem roku 2038"[5]. Co więcej, w przypadku, gdyby plik się zmienił w inny sposób w obu lokalizacjach, narzędzie nie zauważyłoby konfliktu — sytuacji, w której nie da się bez użytkownika stwierdzić, która wersja jest pożądana. Precyzyjniejszą, lecz bardziej czasochłonną metodą wykrywania zmian jest porównywanie sumy kontrolnej lub wyniku funkcji skrótu dla zawartości pliku[1]. W takiej sytuacji potrzebujemy odnieść się do stanu pliku po poprzedniej synchronizacji, który narzędzie musi wewnętrznie zapisywać (w przypadku porównywania czasów modyfikacji też jest to robione, aby sprawdzić, czy nie występuje wspomniany konflikt)[1].

Ustalenie, które pliki trzeba zaktualizować, a które można pominąć, już znacznie zmniejsza nam ilość pracy do wykonania (jeżeli, na przykład, musimy przesłać tylko jeden na dziesięć plików). Możemy jednak pójść o krok dalej, redukując ilość przesyłanych danych potrzebnych do otrzymania nowej wersji pliku. Jednym z takich sposobów jest kompresja danych przed wysłaniem, dzięki której możemy zmniejszyć rozmiar pliku typowo od 2 do 4 razy[2][6]. Innym sposobem jest obliczenie i przesłanie samej *różnicy* w zawartości plików[2], tzw. *delta copying*[3]. Praktycznym przykładem takiego zachowania jest algorytm *rsync*[7][2].

Podsumowując, proces synchronizacji można więc podzielić na siedem zadań:

[1] Teoretycznie też nie jest nieomylna, ponieważ może wystąpić kolizja wyników, lecz dla dobrze dobranego algorytmu prawdopodobieństwo wystąpienia jest znikome.

[2] Występuje pewien narzut związany z obranym protokołem i metodą obliczania różnicy, ale jest on zazwyczaj nieistotny w porównaniu do rozmiaru całego pliku.

[3] Nazywane także *delta compression, delta sync, delta encoding.*

1. Wykrywanie zmian;
2. Ustalanie najnowszej wersji danych;
3. Transformacja danych (opcjonalne);
4. Przesył danych;
5. Transformacja otrzymanych danych do oryginalnej postaci (opcjonalne);
6. Zastosowanie zmian;
7. Zapamiętanie wykonanych zmian;

2.2. Główne cechy narzędzi

Synchronizacja offline/online O synchronizacji danych zazwyczaj mówi się w kontekście więcej niż jednej maszyny i komunikacji przez sieć lokalną lub Internet — większość dostępnych narzędzi zakłada taki model problemu. Takie założenie nie jest bezpodstawne, ponieważ przy przesyle danych z jednego punktu do drugiego największym wąskim gardłem (ang. *„bottleneck"*) jest właśnie łącze internetowe[3], dopiero potem czas odczytu i zapisu danych na dysku[4].

Nie mniej jednak istnieją narzędzia, które wspierają synchronizację w ramach jednej maszyny, czy to pomiędzy dyskami lub systemami plików, czy pomiędzy katalogami. Mogą to być osobne programy lub moduły narzędzi przeprowadzających synchronizację online. Z technicznego punktu widzenia, działanie wyłącznie wewnątrz jednej maszyny znacznie ułatwia pracę - nie trzeba się martwić o prywatność, protokoły komunikacji, niestabilne połączenia.

Topologia i kierunek synchronizacji Każde narzędzie do synchronizacji z inną maszyną zakłada pewną topolgię logicznej sieci[5] i w jakim kierunku zachodzi proces synchronizacji. W przypadku synchronizacji pomiędzy dwoma maszynami rozróżniamy synchronizację jednostronną i dwustronną. W synchronizacji jednostronnej, w ramach jednej sesji maszyna A przesyła swoje pliki maszynie B. Aby zagwarantować identyczny stan obu maszyn, musi nastąpić kolejna synchronizacja, tym razem z B do A. Może to być niewygodne dla użytkownika, jeśli narzędzie nie umożliwia zdalnego rozpoczęcia synchronizacji B do A z poziomu maszyny A. W synchronizacji dwustronnej nie ma takiego problemu — obie maszyny wymieniają się plikami i osiągają taki sam stan końcowy w ramach jednej sesji. Synchronizacja dwustronna wymaga jednak bardziej skomplikowanych narzędzi.

W przypadku większej liczby maszyn biorących udział w synchronizacji zazwyczaj mamy do czynienia z topologią gwiazdy i topologią siatki. Dla topologii gwiazdy typowy jest model klient-serwer, w którym jedna maszyna to serwer zarządzający procesem synchronizacji, a reszta to klienci po kolei biorący w niej udział. Taki tryb działania zamienia synchronizację wielostronną na wiele synchronizacji dwustronnych, w której serwer jest stałym członkiem[8][4] i szczególnie się sprawdza, gdy używamy go do tworzenia/wykorzystywania kopii zapasowych lub repozytoriów plików do odczytu. Topologia siatki zakłada połączenie wiele do wielu na równych prawach, tzw. model *peer-to-peer* (z ang. *„równy*

[4] Dzięki dyskom SSD, operacje wejścia/wyjścia mają obecnie jeszcze mniejsze znaczenie, niż kiedyś.
[5] tj. bierze pod uwagę, które maszyny się ze sobą komunikują i na jakich prawach, a nie fizyczną budowę sieci LAN/WAN.

z równym"). Eliminuje ona potrzebę serwera „nadzorcy" i może przyspieszyć komunikację zwłaszcza, gdy maszyny są ze sobą zsynchronizowane przez większość czasu, lub gdy wykrywanie zmian odbywa się szybko i okresowo, np. na bazie księgowania w systemie plików (ang. *„file system journaling"*).

Bezpieczeństwo i prywatność Komunikacja pomiędzy maszynami poza zamkniętą siecią lokalną jest narażona na ingerencję osób trzecich. W przypadku narzędzi synchronizujących pliki głównymi zagrożeniami są: nieuprawnione uzyskanie dostępu do przesyłanych danych, przesył własnych plików do nieświadomego odbiorcy lub uzyskanie dostępu do systemu plików przez złośliwego aktora podszywającego się pod członka sesji, zazwyczaj realizowane jako tzw. atak *Man-in-the-Middle* (z ang. *„człowiek pośrodku"*). Każda z tych trzech sytuacji może mieć katastrofalne skutki: podsłuchiwane dane mogą być prywatne lub, co gorsza, tajne, wysyłane dane mogą zawierać złośliwe oprogramowanie, a dostęp do systemu pliku pozwala na masową kradzież lub usuwanie danych.

Aby nie dopuścić do takich sytuacji, należy zwracać uwagę, czy narzędzia do synchronizacji mają ograniczone prawa dostępu, szyfrują ruch pomiędzy maszynami i stosują bezpieczne protokoły komunikacji, na przykład SSH2. Do tego, w przypadku rozwiązań opierających się na synchronizacji względem maszyny zewnętrznego usługodawcy (chmurze), kluczowe są jego prawa dostępu i rozporządzania danymi użytkownika — im bardziej ograniczone, tym lepiej dla bezpieczeństwa i prywatności użytkownika.

Cel, dostępność i wieloplatformowość Problem synchronizacji jest uniwersalny — populacja użytkowników takich narzędzi jest zróżnicowana m.in. pod względem skali sieci (komputery osobiste vs wiele maszyn roboczych vs całe klastry), rozmiaru i rodzaju danych (np. dokumenty tekstowe, kody źródłowe, multimedia) i rodzaju klienta (użytkownicy indywidualni vs przedsiębiorstwa i instytucje). Niektóre istniejące narzędzia starają się być do użytku ogólnego, inne skupiają się na specyficznym profilu, tracąc użyteczność w innych (na przykład narzędzie dla masowego przesyłu danych dla firm może nawet nie być wdrażalne/dostępne dla osób prywatnych, a program do synchronizacji repozytoriów kodu nie będzie efektywny dla dołączanych plików wideo). Z powyższych względów programy posiadają bardzo zróżnicowane licencje: bezpłatne (otwartoźródłowe lub własnościowe), komercyjne (pojedynczy zakup lub subskrypcje, szczególnie w przypadku usług chmurowych) i pośrednie, takie jak *trialware*, *shareware* czy *adware*[6].

Sama potrzeba synchronizacji nie zależy od rodzaju maszyny (tak długo, jak posiada system plików i połączenie sieciowe) czy systemu operacyjnego, dlatego należy wziąć pod uwagę różnorodność używanych platform systemowych (Microsoft Windows, GNU/Linux, BSD, macOS, iOS, Android, itp.). Chociaż zakres tej pracy obejmuje dystrybucję Ubuntu systemu Linux, wieloplatformowość pozostaje pożądaną cechą podczas analizy dostępnych rozwiązań.

Niezawodność Proces wymiany danych jest wrażliwy na awarie, zarówno uczestniczących maszyn, jak i samej sieci, co w najlepszym przypadku może skutkować potrzebą

[6] Użyto tutaj terminu pośrednie, ponieważ wprawdzie są dostępne bezpłatnie, ale oferują pełne możliwości (lub brak przeszkód, w postaci np. reklam) za opłatą.

retransmisji, a w najgorszym utratą danych. Odporność na awarie jest więc specjalnie istotna w przypadku operacji na dużych zestawach danych lub na krytycznych danych[7] [8]. Do tematu można podejść na dwa sposoby: zwiększając odporność na awarie lub umożliwiając podjęcie czynności zmniejszających konsekwencje awarii. Przykładami pierwszego sposobu jest np. stosowanie zmian w plikach w sposób atomowy (gwarantując integralność danych w każdym momencie) [4], obliczanie i porównywanie sumy kontrolnej transmitowanych danych[3] czy projektowanie narzędzi w taki sposób, by ich wewnętrzny stan był cały czas spójny. Drugi sposób skupia się na redukowaniu straconego czasu (np. umożliwiając kontynuowanie przerwanych synchronizacji, szczególnie dla plików o dużym rozmiarze) lub utraconej pracy w wyniku błędnego/niekompletnego stosowania zmian (np. poprzez cofnięcie stanu katalogu do ostatniej wykonanej kopii zapasowej).

Synchronizacja na żądanie i periodyczna Niektóre narzędzia oferują automatyczną synchronizację co określony kwant czasu lub tuż po wykryciu zmian[1]. W zależności od potrzeb użytkownika, może to być zaletą lub wadą (albo nawet niebezpieczeństwem). Synchronizacja periodyczna zwalnia użytkownika z obowiązku pamiętania o ręcznym zleceniu wykonania kosztem zmniejszania kontroli nad narzędziem, własnymi danymi i zasobami systemu. W najgorszym wypadku, jeżeli narzędzie nie jest odpowiednio zabezpieczone, może propagować niechciane tymczasowe zmiany. Dodatkowo, synchronizacja periodyczna jest niewygodna w kontekście przeprowadzania badań wydajności — wyniki otrzymamy dopiero, gdy program „zadecyduje", że chce przeprowadzić wymianę plików.

Wydajność i skalowalność Wydajność aplikacji synchronizujących może być oceniana pod względem czasu potrzebnego do przeprowadzenia synchronizacji, wykorzystywanego miejsca i zasobów systemu lub rozmiaru danych przesyłanych przez sieć. Pierwsza metryka, czas, bezpośrednio wynika ze zdefiniowanego problemu (wymiana wszystkich danych pomiędzy maszynami jest czasochłonna), szczególnie dla danych o dużym rozmiarze lub rozległej sieci maszyn. W specyficznych scenariuszach decydujące mogą być pozostałe względy, na przykład gdy wykorzystujemy systemy wbudowane (albo urządzenia mobilne) o ograniczonych parametrach lub jeśli łącze internetowe jest wyjątkowo kosztowne w użyciu, a nie zależy nam na jak najszybszym zakończeniu procesu. Jeżeli mamy jednak do czynienia z sytuacją typową, chcemy dążyć do balansu pomiędzy czasem pracy wewnętrznych algorytmów a czasem przesyłu danych. Ze wzrostem zaawansowania algorytmów wrasta ich czas pracy, ale spada ilość danych potrzebnych do przesłania, a więc i czas spędzony na ich transmisji.

Z wydajnością związana jest skalowalność, w tym kontekście rozumiana jako umiejętność zachowania swoich osiągów przy zwiększonej skali działania. Cecha ta jest najczęściej spotykana w rozwiązaniach przeznaczonych dla przedsiębiorstw czy instytucji, ponieważ tacy użytkownicy zwykle operują na znacznie większych danych.[3]

[7] Dane mogą być krytyczne zarówno w kontekście poprawnego funkcjonowania systemu komputerowego, jak i w sensie ich wartości lub wagi, np. dokumenty medyczne, umowy handlowe, chronione lub tajne informacje itp.

Łatwość i wygoda użycia Ciężko obiektywnie porównywać narzędzia pod względem wygody i prostoty, ponieważ zależą one od preferencji i poziomu wiedzy i umiejętności informatycznych użytkownika końcowego. Można jednak wyszczególnić pewne elementy narzędzi, które pozytywnie wpływają na odbiór końcowy, takie jak: możliwość stosowania skrótów klawiszowych, napisana w zrozumiały, lecz szczegółowy sposób instrukcja obsługi lub interaktywny moduł pomocy, krótki lub automatyczny proces konfiguracji, interfejs użytkownika zaprojektowany zgodnie z zasadami UX (User Experience, z ang.*„doświadczenie użytkownika”*) i ułatwieniami dostępności.

2.3. Wybrane dostępne narzędzia

Istnieją dziesiątki narzędzi publicznie dostępnych (zarówno bezpłatnie jak i odpłatnie) dla użytkowników indywidualnych. Oferowane przez te narzędzia funkcjonalności mogą być takie same lub pokrywać się w dużym stopniu, przez co można je grupować za pomocą zbioru określonych wcześniej cech. Należy zaznaczyć, że omawiane narzędzia zostały wybrane jako przykłady reprezentujące te grupy (lub w przypadku pierwszej pozycji, ze względu na historyczny wpływ) i *nie należy ich traktować jako rekomendacji.*

2.3.1. Rsync

Dostępne od 1996 roku, to narzędzie można znaleźć w ramach niektórych dystrybucji systemów *nix (dostępne obecnie także na systemy Windows). Oferuje jednokierunkową synchronizację pomiędzy dwiema maszynami, wykorzystując protokół SSH, narzędzie *rsh* lub bezpośrednią, niezabezpieczoną komunikację. Narzędzie jest *open source*, a sposób działania i wykorzystywany (także w późniejszych narzędziach) algorytm bardzo dobrze udokumentowany w raporcie technicznym[2] oraz pracy doktorskiej twórcy, Andrew Tridgell'a[7]. Algorytm *rsync* jest jedną z metod pozwalających na przesył praktycznie tylko zmienionych danych w pliku. W specyficznych przypadkach (dla małej liczby plików o dużych rozmiarach) narzędzie może osiągnąć wielokrotną redukcję przesłanych danych.[9]

Pomimo swojego pionierskiego statusu i względnej popularności, *rsync* w obecnych czasach ma ograniczone zastosowanie ze względu na model działania i przyjęte założenia (synchronizacja jednostronna, brak sprawdzania stanu plików przed synchronizacją) i nie jest w stanie wykorzystać w pełni dostępnych zasobów[3].

2.3.2. Unison

Pierwszy raz opublikowane w 1998 roku, *Unison* jest bardziej zaawansowanym narzędziem z interfejsem graficznym pozwalającym na dwukierunkową synchronizację pomiędzy dwiema maszynami lub w ramach jednej maszyny. To narzędzie *open source* wykorzystuje algorytm oparty o algorytm *rsync* do właściwej transmisji danych, ale uzupełnia go o dodatkową warstwę tworzenia profili, wykrywania różnic, rozwiązywania konfliktów, zapewnienia integralności danych, czy tworzenia kopii zapasowych. Dodatkowo, *Unison* jest wieloplatformowe (Windows, Unix, macOS) i pozwala na synchronizację pomiędzy maszynami z różnymi systemami operacyjnymi.[4]

Trzeba jednak zaznaczyć, że mimo tego jest mało przenośne - do poprawnego funkcjonowania oczekuje, że na obu maszynach nie tylko znajdują się dokładnie te same wersje programu, ale że także zgadzają się wersje użytego kompilatora języka OCaml. Narzędzie nie jest rozwijane od 2012 roku.[4]

Podobne narzędzia: FreeFileSync[8].

2.3.3. Syncthing

Nowocześniejsze rozwiązanie *open source* rozwijane od 2013 roku i oferujące synchronizację wielu maszyn jednocześnie. Syncthing korzysta z własnych protokołów i oferuje wysoki poziom bezpieczeństwa poprzez wykorzystywanie nowoczesnych metod szyfrowania i uwierzytelniania maszyn. W przeciwieństwie do poprzednich narzędzi, wykorzystywana jest architektura *peer-to-peer*, gdzie każda skonfigurowana i dodana do wewnętrznej sieci maszyna (wspierane Windows, Unix, macOS, Android) może komunikować się z inną.[10]

Kolejną różnicą jest stosowanie periodycznej synchronizacji (odstępy czasu definiowane przez użytkownika) w ramach chwilowo dostępnych maszyn, podczas gdy poprzednie narzędzia oferowały tylko synchronizację na żądanie.[10]

Podobne narzędzia: Resilio Sync/Resilio Connect[9].

2.3.4. Dropbox

Jedna z wielu usług oferujących przechowywanie danych w chmurze, która oferuje dodatkowe funkcje związane z synchronizacją przesyłanych plików. *Dropbox* pozwala na synchronizowanie maszyny z repliką znajdującą się na zewnętrznym serwerze, który propaguje te zmiany dalej (wspierane systemy Windows, Linux, macOS, Android, iOS). Zaletą takiego rozwiązania jest możliwość tworzenia kopii zapasowych na tym serwerze lub udostępniania w bezpieczny sposób wybranym osobom.[11] Usługodawca zapewnia, że przechowywane dane są szyfrowane i nie ma do nich wglądu, chociaż w przeszłości miały miejsce incydenty związane z bezpieczeństwem i prywatnością.

W przeciwieństwie do innych rozwiązań nie ma możliwości wybrania źródłowego lub docelowego katalogu. Co więcej, bezpłatny plan oferuje tylko 2 GB dostępnego miejsca oraz połączenie do 3 maszyn (gdzie płatny plan oferuje 3 TB miejsca i nielimitowaną liczbę maszyn).[11]

Podobne narzędzia: MEGA[10].

[8] `https://freefilesync.org`
[9] `https://www.resilio.com` - Wcześniej znane jako BitTorrent Sync.
[10] `https://mega.io`

3. Implementacja własnego narzędzia

3.1. Założenia projektowe

3.1.1. Wstępny opis projektu

Pierwotna treść zadania, przed sprecyzowaniem wymagań, warunków działania i interakcji wyglądała następująco:

> „Załóżmy, że czasem pracujemy na maszynie X, a innym razem na Y. Istnieje potrzeba synchronizacji zawartości katalogów przez Internet (setki tysięcy plików, dziesiątki GB). Na maszynie, z którą chcemy się synchronizować, użytkownik ma konto, na które może zalogować się przez SSH.
> Synchronizacja odbywa się na żądanie użytkownika. Narzędzie powinno szybko wykrywać zmiany zawartości plików i proponować wymianę zmodyfikowanych plików. W przypadku zmiany tego samego pliku na obu maszynach konflikt rozstrzyga użytkownik. Narzędzie posiada interfejs graficzny ze skrótami klawiszowymi."

3.1.2. Wymagania

Połączenie pomiędzy maszynami

1. Synchronizacja dwustronna — po synchronizacji obie maszyny mają mieć identyczne stany.
2. Komunikacja pomiędzy maszynami odbywa się poprzez protokół SSH[11].
3. Brak wyznaczonego serwera — Na maszynie obecnej program włączany jest przez użytkownika, przy synchronizacji na maszynie zdalnej program jest automatycznie wywoływany.
4. Maszyna może brać udział w wielu synchronizacjach jednocześnie, o ile nie dotyczą tych samych obszarów.

Porównywanie zawartości katalogów

1. Zmiany wykrywane są poprzez porównywanie wyniku funkcji skrótu zawartości pliku.
2. Identyfikowane jest powstanie nowego pliku lub zniknięcie starego.
3. Informacje o stanie każdego obserwowanego pliku po synchronizacji są zapisywane do bazy danych.
4. Narzędzie ogłasza konflikt, gdy na obu maszynach są wersje o skrócie innym niż zapamiętany podczas poprzedniej synchronizacji.

Interakcja użytkownika

1. Narzędzie posiada interfejs graficzny oraz możliwość nawigacji wyłącznie za pomocą skrótów klawiszowych.
2. Użytkownik może tworzyć konfiguracje, podając lokalny katalog, drugą maszynę i zdalny katalog.
3. Użytkownik wybiera konfigurację, która będzie synchronizowana.

[11] Specyfikacja: RFC4250-4256 i inne.

4. Rozpoczynając synchronizację, użytkownik uwierzytelnia się metodami typowymi dla protokołu SSH.
5. Narzędzie prezentuje użytkownikowi listę zmian, konfliktów i proponowanych akcji.
6. Użytkownik może uzyskać dodatkowe informacje o danym pliku (rozmiar, data modyfikacji).
7. Użytkownik może dodać zasady wyłączające niektóre pliki z synchronizacji (odrzucanie po nazwie lub rozszerzeniu).
8. Użytkownik może dokonać synchronizacji tylko wybranych plików.
9. Użytkownik może zmienić proponowaną przez program akcję.
10. Użytkownik może wybrać, którą wersję zachować podczas konfliktu, lub użyć zewnętrznego narzędzia.

Przesył danych

1. Pliki są przesyłane za pomocą protokołu SFTP[12].
2. Pliki są kompresowane przed przesyłaniem.
3. Pliki, które nie udało się w pełni zsynchronizować, są oznaczane do ponownej synchronizacji, a przesłane fragmenty są tymczasowo zapisywane.
4. Zmiany w plikach są aplikowane w sposób atomowy.

3.1.3. Ograniczenia techniczne

Narzędzie ma działać ma maszynach z systemami operacyjnymi Ubuntu (wersja 20.04 lub nowsza). Narzędzie wymaga kompilatora C++ wspierającego co najmniej standard C++17. Narzędzie ma obsługiwać protokół SSH1 i SSH2.

3.2. Wybór narzędzi

3.2.1. Język

Wybór języka C++ jako głównego języka projektu był silnie sugerowany podczas określania tematyki pracy. Wybór ten jest poparty następującymi argumentami:

Szybkość działania Język C++ jest językiem kompilowanym bezpośrednio do kodu wykonywalnego. Dzięki temu programy napisane w tym języku są szybsze od języków *stricte* interpretowanych (np. Python, JavaScript).

Ogólnodostępność Język C++ ma wieloletnią historię i jego kompilatory są dostępne na większość platform sprzętowych i systemów operacyjnych — w tym tych, które ma wspierać pisane narzędzie. Sam język posiada szczegółową specyfikację w ramach standardu ISO C++, co gwarantuje spójność działania na różnych platformach. Dodatkowo nie ma problemów ze znalezieniem materiałów odnośnie używania C++, czy to w formie książek i publikacji, czy blogów, poradników, forów i stron internetowych.

Kompatybilność z językiem C Język C++ jest w większości kompatybilny wstecznie z językiem C. Jest to ważna cecha, ponieważ biblioteki systemowe Linux, które są wykorzystywane w projekcie, są napisane właśnie w C.

Dostępność bibliotek Ten punkt jest konsekwencją dwóch wyżej wymienionych cech. Istnieje wiele bibliotek *open source* oferujących funkcje, które są potrzebne w projekcie. Szczegóły odnośnie wybranych bibliotek i ich specyfiki znajdują się w następnej sekcji, 3.2.2.

Nie wybrano pokrewnego języka C (który także jest szybki i ogólnodostępny), ponieważ jest językiem niskopoziomowym, narzucającym paradygmat programowania proceduralnego i wymagającym zagłębiania się w szczegóły implementacyjne elementów, które nie są istotne z punktu widzenia tematyki tej pracy. Język C++ oferuje wyższy poziom abstrakcji, bezpieczeństwa działania, znacznie bardziej rozbudowaną bibliotekę standardową i możliwość stosowania paradygmatu programowania obiektowego.

3.2.2. Biblioteki

Biblioteka standardowa C++ Oferuje większość funkcji związanych z odczytem i zapisem danych, generyczne kontenery danych (w ramach STL — Standard Template Library, z ang. *„standardowa biblioteka szablonów”*), komunikację z systemem plików niezależną od systemu operacyjnego.

Biblioteki systemowe systemu Linux Wykorzystywane są do interakcji z systemem operacyjnym i systemem plików, które nie są możliwe używając biblioteki standardowej języka (ponieważ ich nie ma lub są specyficzne dla systemów Linux). Biblioteki są napisane w języku C, więc w konsekwencji fragmenty kodu korzystające z nich też muszą być napisane w stylu C.

GUI Głównymi kandydatami były *Qt*[12] i *wxWidgets*[13]. Obie biblioteki, pozwalają tworzyć rozbudowane aplikacje okienkowe które są wieloplatformowe, wspierają skróty klawiszowe, posiadają rozbudowaną dokumentację i są dostępne w ramach konkretnych dystrybucji systemu Linux. Ostatecznie wybrano *wxWidgets* z trzech powodów: jest mniejsza, wykorzystuje natywne elementy GUI systemu operacyjnego (podczas gdy *Qt* oferuje tylko własne implementacje) oraz przyjaźniejszej licencji, która w przypadku *wxWidgets* jest bardziej swobodną wersją Library GPL[14] (jest to przodek obecnie znanej LGPL - GNU Lesser General Public License).

SSH/SFTP Rozważałem biblioteki *libssh*[15], *libssh2*[16], *libassh*[17] i *QSsh*[18]. Ostatnia z nich jest powiązana z *Qt* i została odrzucona w momencie wyboru biblioteki GUI. Biblioteki *libssh2* i *libassh* zostały odrzucone, ponieważ wspierają tylko protokół SSH2 (który nie jest kompatybilny z wersją SSH1, a zależy nam na kompatybilności ze starszymi systemami). W konsekwencji wybrana została biblioteka *libssh*, która wspiera obie wersję oraz korzystanie z protokołu SFTP. Biblioteka jest napisana w większości w języku C, z czym wiążą się opisane wcześniej niedogodności. Wykorzystywana na zasadach licencji LGPL.

[12] `https://www.qt.io`
[13] `https://www.wxwidgets.org`
[14] `https://opensource.org/licenses/wxwindows.php`
[15] `https://libssh.org`
[16] `https://www.libssh2.org`
[17] `https://www.nongnu.org/libassh/`
[18] `https://github.com/sandsmark/QSsh`

Kompresja Wybrałem algorytm ZStandard[19] i jego implementację dostępną w języku C, *zstd*[20]. Jest to algorytm ogólnego zastosowania, tak jak powszechnie znany DEFLATE, lecz nowszy (w konsekwencji, bazujący na nowocześniejszych, bardziej wydajnych metodach) i szybszy[6]. Zstandard jest wykorzystywany w wielu projektach komercyjnych i *open source*[13], co jest znakiem wysokiej jakości i efektywności. Implementacja algorytmu ma porównywalny współczynnik kompresji z biblioteką *zlib* (którą wykorzysuje np. biblioteka *ZipLib*[21] w C++), lecz wielokrotnie krótszy czas kompresji (ok. 4 razy szybciej) i dekompresji (ok. 5 razy szybciej)[13], co ma duże znaczenie, gdy chcemy często i na dużą skalę kompresować pliki. Implementacja algorytmu posiada dwie licencje (do wyboru): GPLv2 i BSD.

Funkcja skrótu Wybrałem algorytm xxHash[22] i jego oficjalną implementację o tej samej nazwie dostępną w języku C[23]. Algorytm jest dostępny w wariancie 32-, 64- i 128-bitowym i oferuje zadowalający kompromis pomiędzy niskim prawdopodobieństwem kolizji, a szybkością działania, pokonując pod względem prędkości przetwarzania danych (29.6 GB/s) inne znane algorytmy, takie jak MD5 (0.6 GB/s), SHA1 (0.8 GB/s) czy Murmur3 (3.9 GB/s) i inne[14], dla których możemy znaleźć implementację w C++[24]. Szybkość działania jest tutaj kluczowa, ponieważ funkcja skrótu jest używana dla całego, potencjalnie dużego, drzewa plików, a zależy nam na możliwie najkrótszym czasie wykrywania zmian. Wybrano implementację w C zamiast w C++, ponieważ ta druga wymaga dość nowego standardu C++20, podczas gdy projekt ograniczony jest do C++17. Implementacja posiada licencję BSD 2-Clause.

Baza danych Jako bazę danych wybrano *SQLite*[25] oraz bibliotekę *SQLiteC++*[26]. *SQLite* operuje na bazach danych w formie zwykłego pliku, dzięki czemu nie trzeba instalować żadnego programu na maszynie docelowej — wystarczy, że program będzie zawierał bibliotekę. Dzięki temu projekt jest bardziej przenośny. Biblioteka *SQLite++* oferuje warstwę pośrednią w C++ pomiędzy własnym kodem, a API *SQLite* w języku C. *SQLite* znajduje się w domenie publicznej, a *SQLite++* posiada licencję MIT.

Automatyzacja testów Następujące biblioteki do testów jednostkowych były rozważane: *GoogleTest*[27], *Catch2*[28], *CppUnit*[29] i *Boost.Test*[30]. *CppUnit* jako pierwsze została odrzucona w poszukiwaniu bardziej nowoczesnego rozwiązania. Następnie zrezygnowno z *Boost.Test*, ponieważ preferowalny był sposób pisania testów w *GoogleTest* i *Catch2* (o bardzo zbliżonej

[19] Opisany w RFC 8878: `https://datatracker.ietf.org/doc/html/rfc8878`
[20] `https://facebook.github.io/zstd/`
[21] `https://bitbucket.org/wbenny/ziplib/wiki/Home`
[22] Specyfikacja: `https://github.com/Cyan4973/xxHash/blob/dev/doc/xxhash_spec.md`
[23] `https://cyan4973.github.io/xxHash/`
[24] Na przykład: `https://github.com/stbrumme/hash-library`
[25] `https://www.sqlite.org/index.html`
[26] `http://srombauts.github.io/SQLiteCpp/`
[27] `https://github.com/google/googletest`
[28] `https://github.com/catchorg/Catch2`
[29] `https://cppunit.sourceforge.net/doc/1.8.0/index.html`
[30] `https://www.boost.org/doc/libs/1_80_0/libs/test/doc/html/index.html`

postaci, w konwencji *xUnit*). Ostatecznie wybrano *GoogleTest* i oferowane automatyczne wykrywanie testów, chociaż *Catch2* jest równie dobrym wyborem.

Inne Biblioteki opisane w tym fragmencie nie mają kluczowego wpływu na projekt — zostały dodane, żeby nie musieć samemu implementować niektórych funkcji. Biblioteka *{fmt}*[31] oferuje formatowanie tekstu i prezentacje danych w formie tekstowej w wygodny i elastyczny sposób. Biblioteka *Easylogging++*[32] pozwala na sformatowane generowanie logów opisujących stan programu - informacje szczególnie przydatne w przypadku napotkania błędów. Obie biblioteki są napisane w nowoczesnym C++ i udostępniane na licencji MIT.

3.2.3. Środowisko programistyczne i inne narzędzia

Platforma systemowa Poniżej przedstawione są informacje odnośnie środowiska, w którym projekt był tworzony.

System operacyjny gospodarza: Windows 10 z Windows Subsystem for Linux 1.
System operacyjny gościa: Ubuntu 20.04 LTS
Kompilator: GCC 9.4.0

Dodatkowo, projekt był testowany w następującym środowisku:
System operacyjny: Ubuntu 22.04 LTS
Kompilator: g++ 11.3

System budowania *CMake*[33] to narzędzie służące do konfigurowania kompilacji oraz testowania programów napisanych w C/C++, generujących skrypty kompilacji natywne dla maszyny i kompilatora, na której działa. Dzięki temu otrzymywanie plików wykonywalnych wymaga minimalnej aktywności użytkownika, a projekt mniej zależny od używanego przez programistę kompilatora. Dodatkowo, odpowiednio skonfigurowany *CMake* jest w stanie automatycznie pobierać i kompilować potrzebne zależności. Narzędzie jest *open source* i dostępne na wiele platform sprzętowych i systemowych.

Aby poprawnie zbudować projekt, *CMake* jest wymagany w wersji co najmniej 3.14.

Analiza statyczna *Cppcheck*[34] to narzędzie do statycznej analizy kodu źródłowego C/C++, wykrywające błędy w kodzie, niezdefiniowane zachowanie i inne potencjalnie niechciane konstrukcje.

Cppcheck nie jest wymagane do zbudowania projektu.

Analiza dynamiczna *Valgrind*[35] to zestaw narzędzi do dynamicznej analizy programów napisanych w C/C++, w szczególności profilowania i wykrywania błędów w zarządzaniu pamięcią lub wątkami.

Valgrind nie jest wymagane do zbudowania projektu.

[31] `https://github.com/fmtlib/fmt`
[32] `https://github.com/amrayn/easyloggingpp`
[33] `https://cmake.org`
[34] `https://cppcheck.sourceforge.io`
[35] `https://valgrind.org`

Projektowanie GUI Narzędzie *wxFormBuilder*[36] jest przeznaczone do prototypowania i tworzenia GUI dla biblioteki *wxWidgets*. Jest w stanie symulować zachowanie tworzonego interfejsu oraz generować kod klas C++ lub Python oraz pliki w specjalnym formacie XRC, który jest wykorzystywany w projekcie.

wxFormBuilder nie jest wymagane do zbudowania projektu.

3.3. Architektura aplikacji

Komentarz wstępny Z punktu widzenia projektowania systemów, projekt nie jest skomplikowany — modułów jest niewiele i mają znacznie różne odpowiedzialności, a zależności są ograniczone. Przy podejmowaniu decyzji architektoniczych kierowałem się bardziej pragmatycznym minimalizmem niż dogmatycznym stosowaniem paradygmatu programowania obiektowego, np. jeżeli działanie zestawu powiązanych logicznie funkcji nie zależy od stanu, zamiast klasy mogą być przestrzenią nazw.

Zawarte w tym rozdziale diagramy UML są poglądowe; nie zawierają informacji o metodach i atrybutach, a jedynie ilustrują zależności. Jest to świadoma decyzja, którą podjąłem, aby nie zaburzać czytelności i nie pokazywać szczegółów implementacyjnych, które są nieistotne dla czytelnika na poziomie omawiania zadań i koncepcyjnego działania modułów.

3.3.1. Słownik pojęć

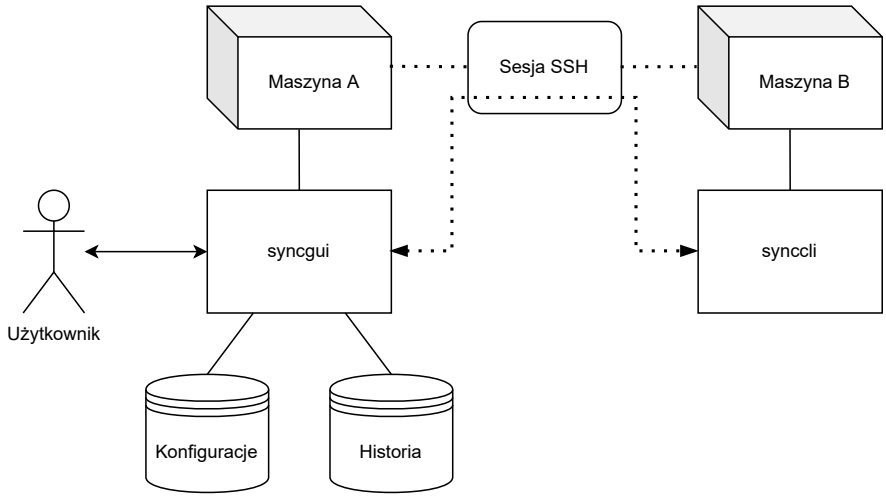
- Sygnatura, sygnatura pliku — wynik funkcji skrótu, której wejściem była zawartość pliku;
- Historia — ogół informacji o wszystkie plikach po ostatniej synchronizacji, utrzymywana pomiędzy sesjami aplikacji;
- Historia pliku — informacje o danym pliku po ostatniej synchronizacji, utrzymywane pomiędzy sesjami aplikacji;
- Główny synchronizowany katalog — katalog, którego zawartość ma być synchronizowana; ścieżki wszystkich plików są podawane względem tego katalogu;
- Konfiguracja — podawany przez użytkownika zbiór informacji odnośnie głównych synchronizowanych katalogów i maszyny zdalnej, inaczej profil działania aplikacji;
- Konflikt — sytuacja, w której stan pliku jest inny na maszynie lokalnej, maszynie zdalnej i w historii;
- Parowanie, parowanie plików — proces powiązywania ze sobą informacji o pliku z maszyny lokalnej, maszyny zdalnej i historii pliku oraz ustalania domyślnej akcji;
- Replika, replika pliku — wersja pliku na jednej z maszyn;
- Skanowanie — proces przeszukiwania głównych synchronizowanych katalogów na obu maszynach w poszukiwaniu zmian i wczytywania historii; w GUI oznacza także następujące po skanowaniu parowanie wyników;

3.3.2. Zarys aplikacji

Narzędzie jest podzielone na dwa komunikujące się ze sobą programy: **syncgui** i **synccli**. **Syncgui** jest programem głównym, który posiada GUI, zarządza bazami danych, sesją

[36] `https://github.com/wxFormBuilder/wxFormBuilder`

SSH, wykonuje większość operacji w ramach synchronizacji oraz uruchamia **synccli** na maszynie docelowej. **Synccli** jest niewielkim programem pomocniczym, który zawiera wyłącznie funkcje potrzebne do asystowania **syncgui** przy przeprowadzaniu synchronizacji — nie jest przeznaczony do interakcji z użytkownikiem. Model ilustruje Rysunek 3.1.



Rysunek 3.1. Uproszczony model działania aplikacji.

Typowa sesja użytkownika składa się z następujących akcji:

1. Użytkownik wybiera (lub tworzy) konfigurację;
2. Użytkownik rozpoczyna skanowanie;
3. Użytkownik uwierzytelnia się na maszynie zdalnej;
4. Narzędzie przeszukuje główne synchronizowane katalogi obu maszyn i wczytuje historię;
5. Narzędzie paruje wyniki i prezentuje je użytkownikowi;
6. Użytkownik wybiera akcje i/lub rozwiązuje konflikty;
7. Użytkownik rozpoczyna synchronizację;
8. Narzędzie propaguje zmiany i prezentuje rezultat użytkownikowi;

3.3.3. Obiekty domenowe

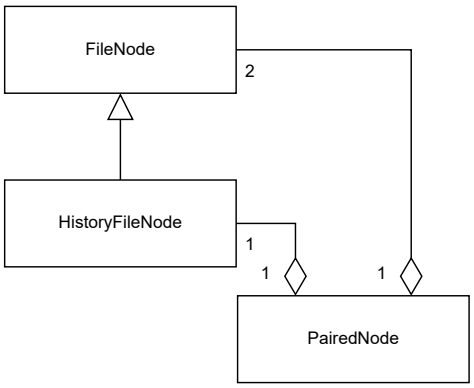
Program operuje na pięciu typach obiektów domenowych:

- *Configuration* — konfiguracja utworzona przez użytkownika, zawierająca informacje o głównych synchronizowanych katalogach, maszynie zdalnej i wygenerowany identyfikator UUID (Universally Unique IDentifier, z ang. *„uniwersalnie unikatowy identyfikator”*)[37], który jest unikatowy niezależnie od czasu i przestrzeni;
- *ConflictRule* — zasada rozwiązywania konfliktu, konflikty opisuje Paragraf 3.3.8;
- *FileNode* — informacje o pliku na jednej z maszyn: ścieżka, numer urządzenia (*device number*) i numer i-węzła (*inode number*)[38], czas modyfikacji, rozmiar, 128-bitowa sygnatura;

[37] Standard UUID opisany w RFC 4122: `https://www.rfc-editor.org/rfc/rfc4122`

[38] W systemach plików pochodnych po UFS taka para pozwala na jednoznaczną identyfikacje pliku w obrębie maszyny.

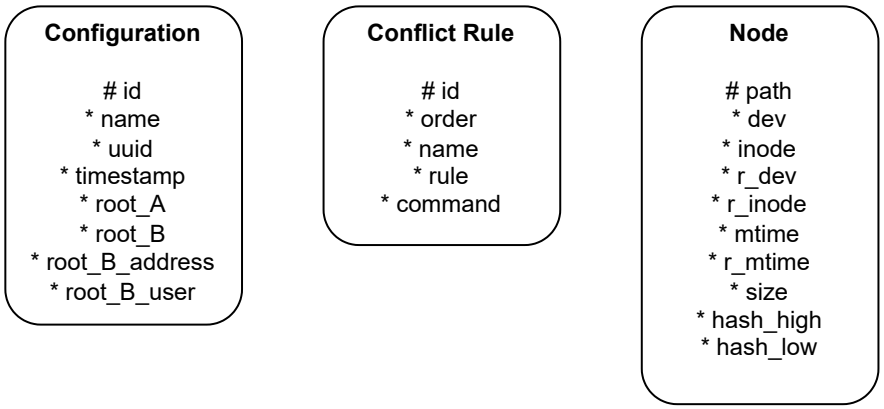
- *HistoryFileNode* — informacje o pliku w historii;
- *PairedNode* — skojarzone ze sobą informacje o pliku na obu maszynach i w historii (Rysunek 3.2), rozszerzone o akcję do wykonania na pliku i status wykonywania akcji;



Rysunek 3.2. Diagram UML klas przechowujących informacje o pliku.

3.3.4. Moduł bazy danych

Trzy typy obiektów domenowych wymagają persystencji: *Configuration*, *ConflictRule* i *HistoryFileNode*. Rysunek 3.3 przedstawia ich reprezentację w bazie danych.

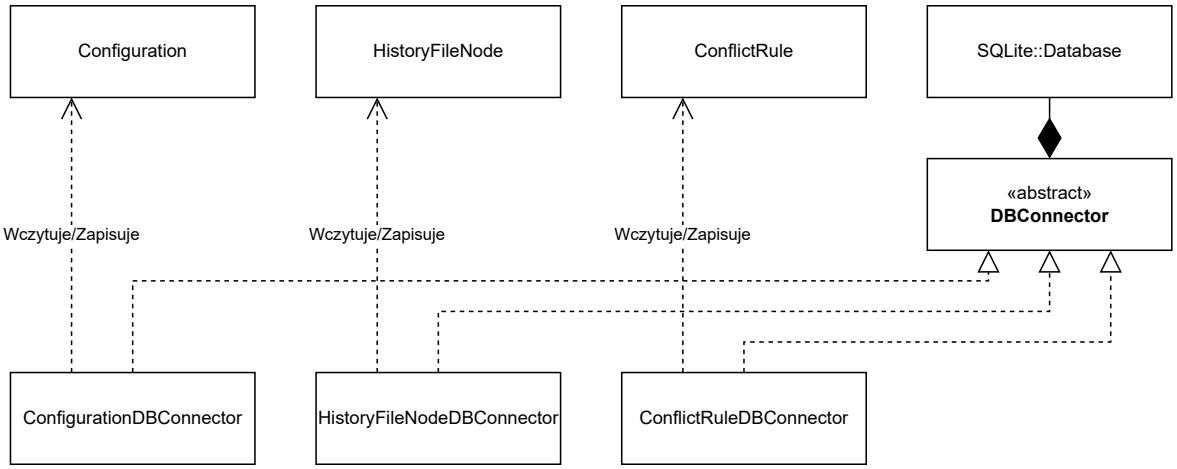


Rysunek 3.3. Diagramy ERD (notacja Barkera) dla wykorzystywanych tabeli baz danych. Prefiks `r_` w tabeli *Node* oznacza zapamiętane informacje o replice zdalnej. Sygnatura pliku podzielona jest na dwie 64-bitowe części.

Konfiguracje zapisywane są do głównej bazy danych (nazwanej `main.db3`). Historia plików i reguły rozwiązywania konfliktów związane z daną konfiguracją utrzymywane są w oddzielnej bazie danych, której nazwą jest UUID konfiguracji. Dzięki temu otrzymujemy lepszą separację danych między konfiguracjami oraz, korzystając z faktu, że bazy danych SQLite zawiera się w plikach, możliwość łatwego utworzenia kopii zapasowej danych przez użytkownika. Wszystkie bazy danych znajdują się w katalogu `~/.sync/db`.

Na poziomie aplikacji, dostęp do danych odbywa się przez klasę abstrakcyjną *DBConnector*, która jest rozszerzana przez DAO (Data Access Object, z ang. *„obiekt dostępu do danych”*) odpowiedzialne za poszczególne typy danych. Ze względu na prostą strukturę danych i brak różnorodnych operacji na obiektach domenowych, DTO (Data Transfer

Object, z ang. *„obiekt transferu danych”*) nie są wykorzystywane w projekcie. Rysunek 3.4 ilustruje sposób implementacji.



Rysunek 3.4. Diagram UML klasy DBConnector.

3.3.5. Algorytm przeszukiwania katalogu

Aby ocenić, które pliki wymagają synchronizacji, narzędzie musi posiadać komplet informacji o zawartości synchronizowanego katalogu — jakie pliki istnieją, są dostępne oraz jaka jest ich zawartość. Zadanie te jest realizowane przez algorytm opisany poniższym schematem blokowym (Rysunek 3.5, Rysunek 3.6). Algorytm implementuje klasa *Creeper* (z ang. *„pełzacz”*). Przed rozpoczęciem iterowania po zawartości katalogu, sprawdzane jest, czy znajdują się w nim pliki `.SyncBlackList` i `.SyncWhiteList`. Pliki zawierają „reguły ignorowania”, czyli zbiór wyrażeń opisujących, które ścieżki (w formie ścieżki względem głównego synchronizowanego katalogu) mają być pomijane (w `.SyncBlackList`) lub wyjątki od tych pierwszych (w `.SyncWhiteList`).

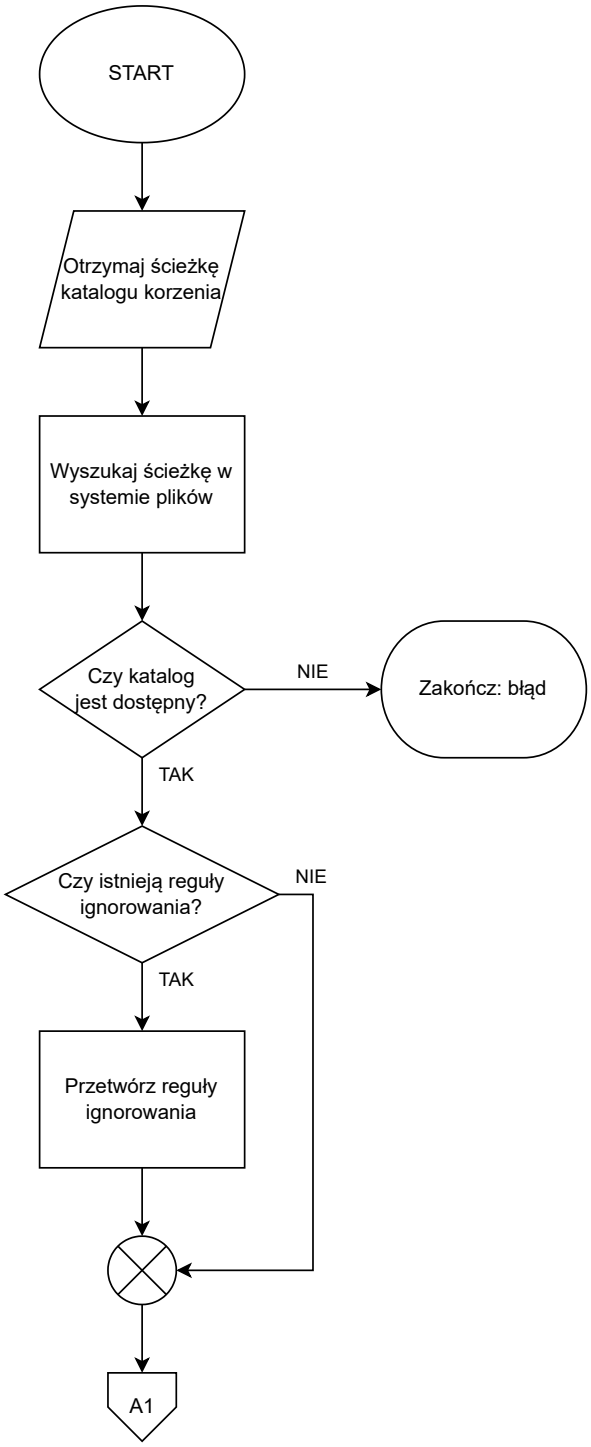
Pominięte (ignorowane) pliki nie będą brały udziału w dalszych procesach — z punktu widzenia narzędzia nie będą w ogóle istnieć. Wyrażenia korzystają ze składni wyrażeń regularnych. Ścieżka musi być dopasowana *w całości*, aby reguła była do niej zastosowana. Forma jest minimalnie przetwarzana przed parsowaniem. Wykonywane są dwie czynności:

- Zamiana `.` na `\.`;
- Zamiana `*` na `.*`;

Listing 1. Przykładowa zawartość pliku .SyncBlackList.

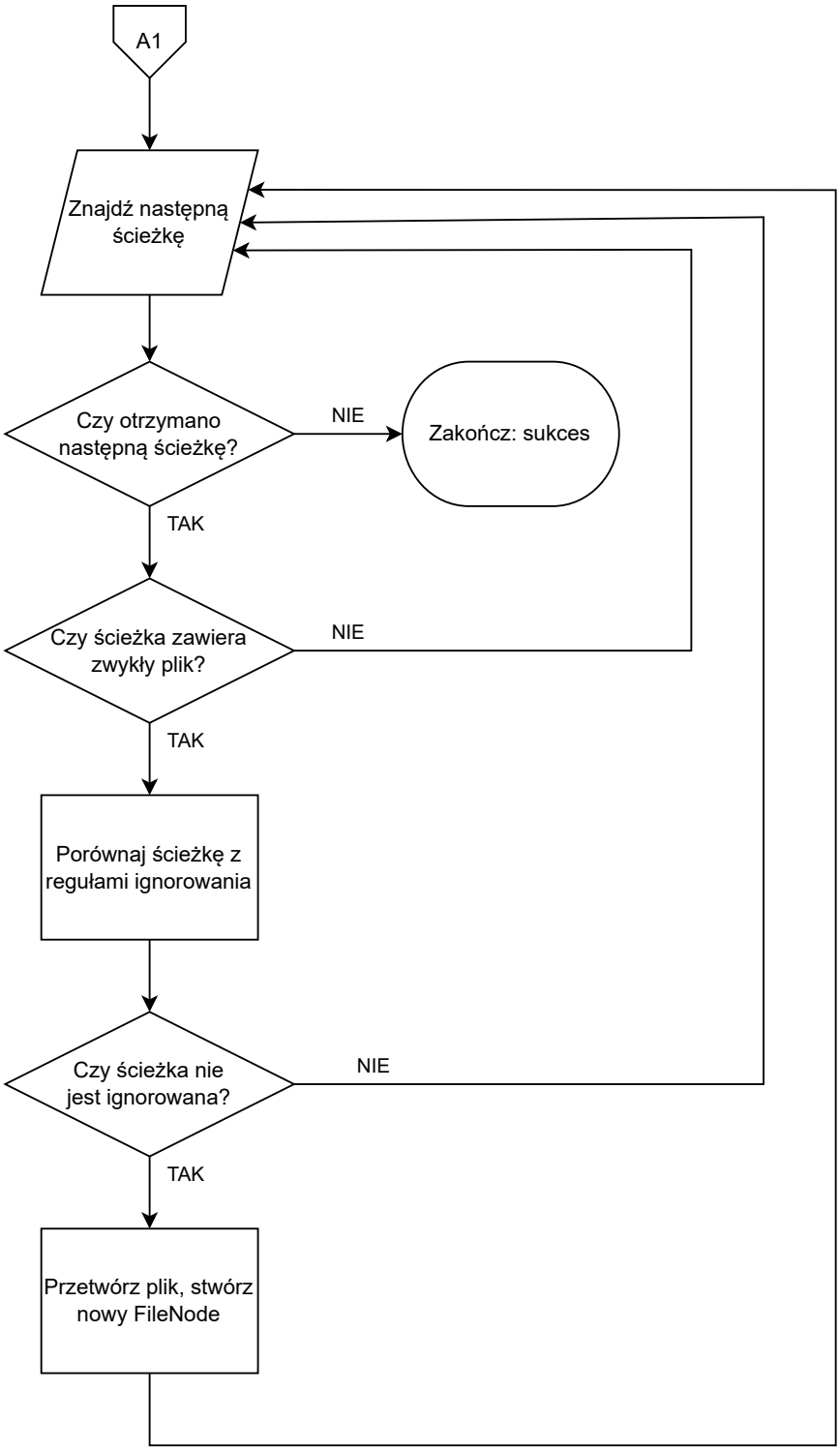
```
1 *.tmp
2 debug/*
3 ignore.me
4 */log/*
```

Dla każdej znalezionej ścieżki zawierającej zwykły plik[39] tworzony jest obiekt *FileNode* zawierający informacje o rozmiarze, czasie modyfikacji i obliczonej 128-bitowym sygnaturze pliku.



Rysunek 3.5. Schemat blokowy algorytmu przeszukiwania katalogu, cz. 1.

[39] inne rodzaje plików występujące w systemach Unix, takie jak potoki FIFO, gniazda, dowiązania symboliczne czy urządzenia, są ignorowane.

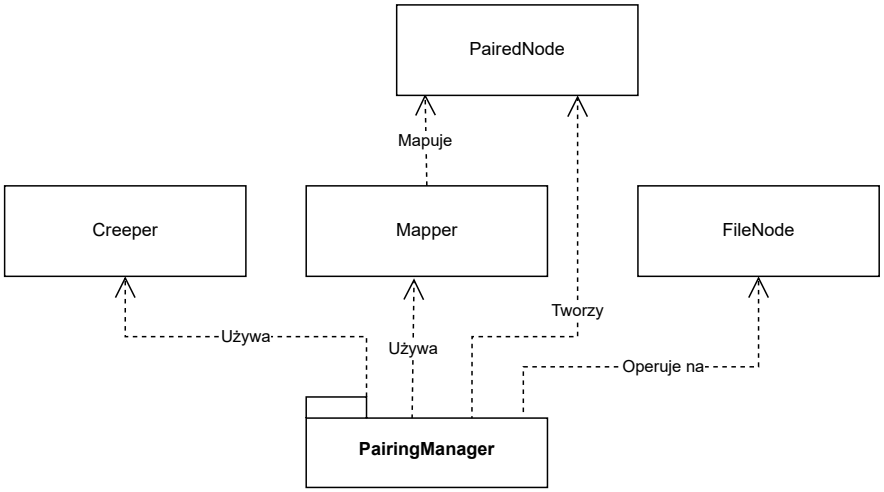


Rysunek 3.6. Schemat blokowy algorytmu przeszukiwania katalogu, cz. 2.

3.3.6. Moduł parowania

Moduł parowania (Rysunek 3.7) wykorzystuje dostarczone z obu maszyn listy obiektów *FileNode* i wczytaną z bazy danych konfiguracji listę obiektów *HistoryFileNode*. Jego zadaniem jest takie skojarzenie ze sobą tych trzech zbiorów, by otrzymać jeden, zawierający informacje o stanie obu replik (w porównaniu do ostatniej synchronizacji). Tworzenie takiego zbioru, czyli listy obiektów *PairedNode*, odbywa się w sposób inkrementalny, nakładając na siebie kolejne zbiory. Parowanie odbywa się w czterech fazach:

1. Konstruowanie obiektów *PairedNode* na bazie lokalnych *FileNode* i mapowanie ich;
2. Parowanie ścieżek w historii z ścieżkami lokalnymi (Rysunek 3.8);
3. Parowanie ścieżek zdalnych z resztą (Rysunek 3.9);
4. Wybieranie domyślnych akcji dla otrzymanych par (Tabela 3.1);



Rysunek 3.7. Diagram UML przestrzeni PairingManager.

Kolejność obiektów *FileNode* w zbiorach nie musi być taka sama, więc nie możemy polegać na pozycji w zbiorze. Dlatego przy tworzeniu par musimy wyszukiwać istniejące pary za pomocą ścieżki. Mapowanie nazw ścieżek (realizowane przez klasę *Mapper*) polega na dodawaniu asocjatywnych par ścieżka-*PairedNode*[40] do obiektu klasy *Mapper*. Do wyszukiwania ścieżek, klasa używa kontener C++ typu *std::map* o złożoności wyszukiwania $O(logn)$[41].

Podczas parowania ustalany jest status każdego pliku obu replik:

- *New* - istnieje plik, którego nie było podczas ostatniej synchronizacji;
- *Absent* - brak pliku;
- *Clean* - brak zmian zawartości od ostatniej synchronizacji;
- *Dirty* - zmiany zawartości od ostatniej synchronizacji;
- *Changed* - stan pliku zmienił się pomiędzy skanowaniem a synchronizacją; ten stan nie jest osiągalny podczas parowania;

[40] Dokładniej na uchwyt do obiektu.

[41] Kontener *std::map* wewnętrznie tworzy drzewo czerwono-czarne (binarne): `https://en.cppreference.com/w/cpp/container/map`

Po ustaleniu statusów, program może zaproponować domyślną akcję do wykonania podczas synchronizacji. Spis wszystkich dostępnych akcji zawiera Paragraf 3.3.7. Możliwe domyślne akcje to:

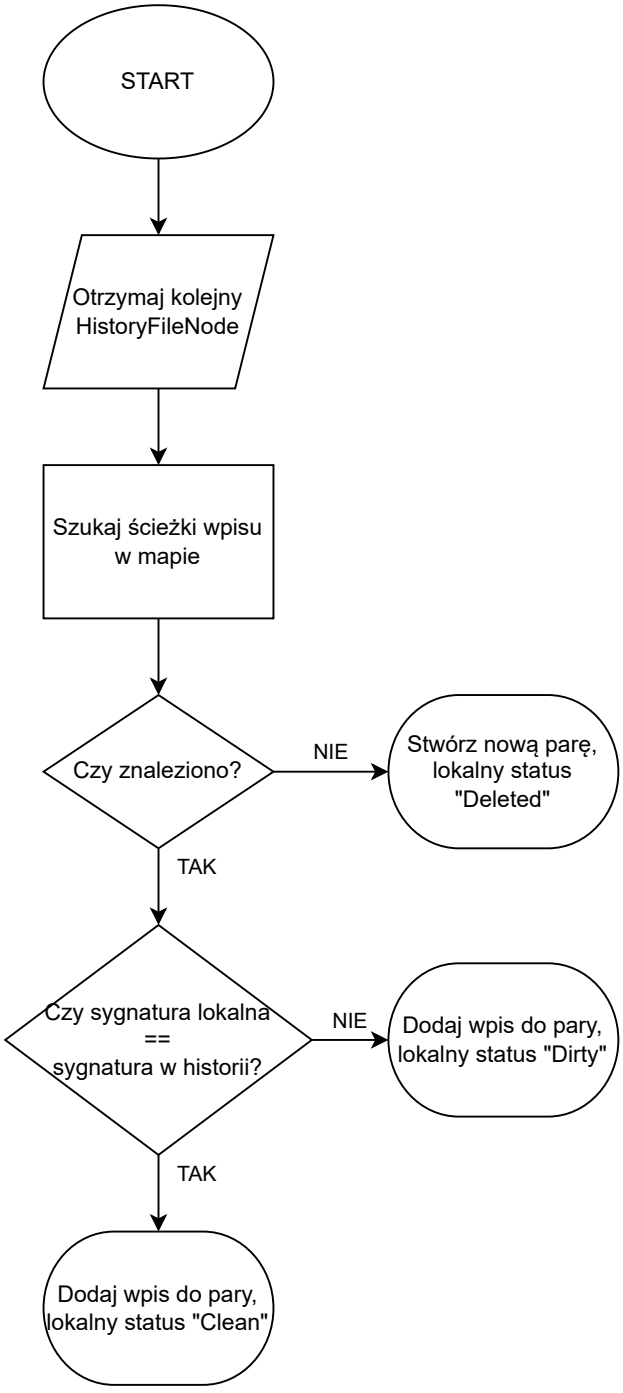
- *Local to Remote* (*L to R*) - propaguj zmiany z maszyny lokalnej na zdalną;
- *Remote to Local* (*R to L*) - propaguj zmiany z maszyny zdalnej na lokalną;
- *No Op* - nie rób nic;
- *Fast Forward* - zaktualizuj sam wpis w historii pliku;
- *Conflict* - wystąpił konflikt, żadna akcja nie może zostać automatycznie podjęta;

Sytuacje oznaczone w Tabeli 3.1 znakiem '-' są niemożliwe do osiągnięcia. Wynika to z faktu, że status *New* występuje, gdy dla danej ścieżki nie istnieje historia pliku, podczas gdy *Clean* i *Dirty* występują tylko jeśli historia istnieje. Sytuacje *Dirty-Dirty* i *New-New*, oznaczone w Tabeli 3.1 znakami '???', są niejednoznaczne. Na podstawie samego statusu program nie jest w stanie określić, czy repliki pliku są identyczne, a więc czy ma do czynienia z akcją *Fast Forward*, czy *Conflict*. Taki problem rozwiązywany jest wcześniej, w przedostatniej fazie parowania[42].

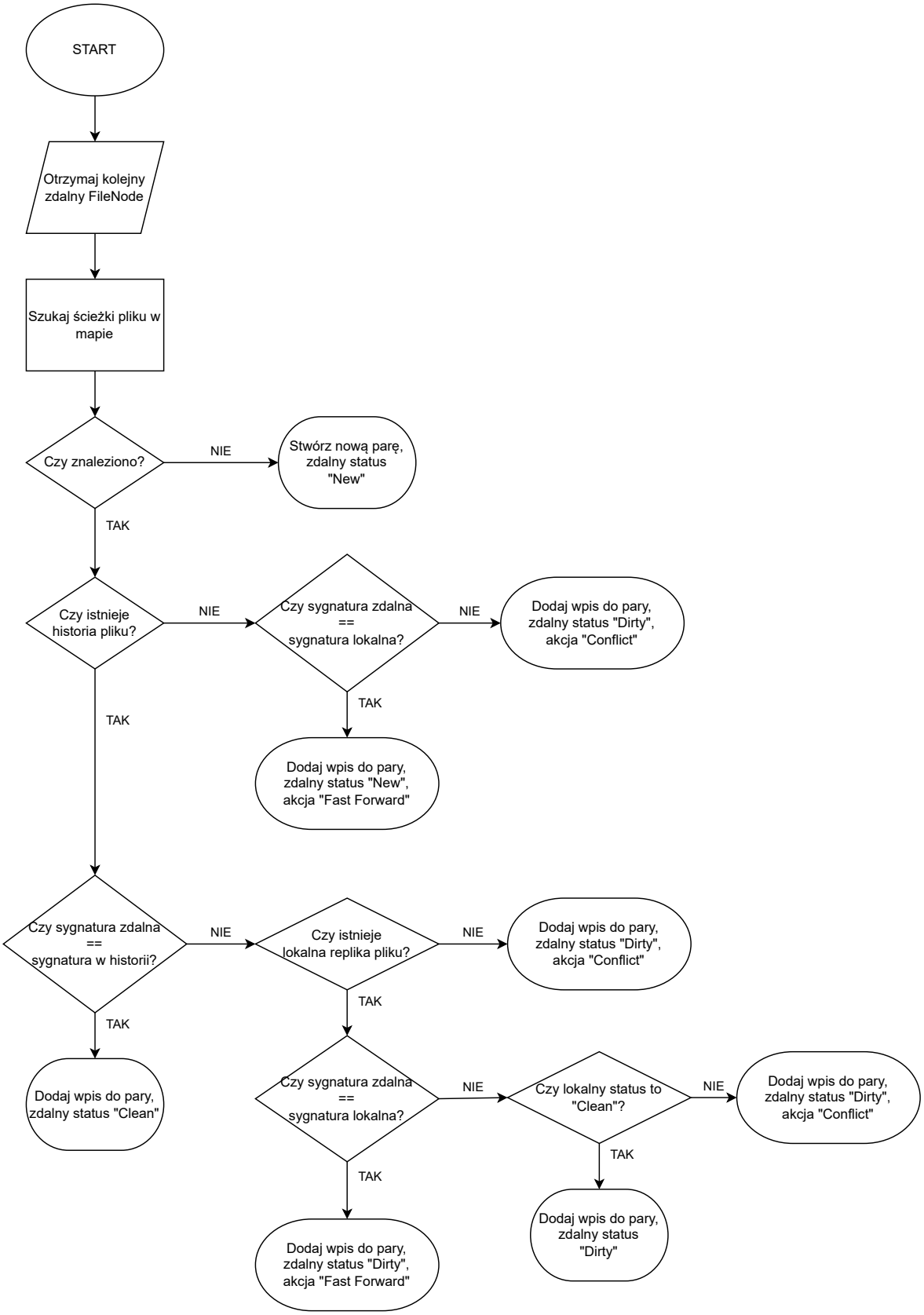
Tabela 3.1. Tabela wyboru akcji na podstawie stanów replik.

		Replika zdalna			
		Clean	Dirty	New	Absent
Replika lokalna	Clean	No Op	R to L	-	R to L
	Dirty	L to R	???	-	Conflict
	New	-	-	???	L to R
	Absent	L to R	Conflict	R to L	Fast Forward

[42] Rysunek 3.9, punkty decyzyjne „Czy sygnatura zdalna == sygnatura lokalna?".



Rysunek 3.8. Schemat blokowy algorytmu parowania z historią repliki.

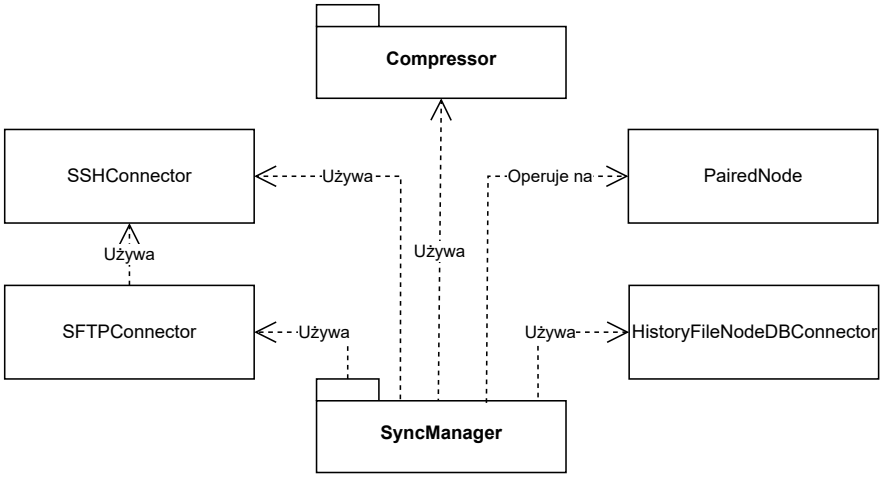


Rysunek 3.9. Schemat blokowy algorytmu parowania z repliką zdalną.

3.3.7. Moduł synchronizowania

Moduł (Rysunek 3.10) jest odpowiedzialny za propagowanie wybranych przez użytkownika zmian (akcji skojarzonych z daną parą plików). Dostępne akcje to:

- *Local to Remote* — propaguj zmiany z maszyny lokalnej na zdalną;
- *Remote to Local* — propaguj zmiany z maszyny zdalnej na lokalną;
- *No Op* — nie rób nic (brak zmian na obu maszynach);
- *Ignore* — nie rób nic (decyzja użytkownika);
- *Fast Forward* — zaktualizuj sam wpis w historii pliku;
- *Conflict* — wystąpił konflikt, żadna akcja nie może zostać automatycznie podjęta;
- *Resolve* — propaguj zmiany po rozwiązanym konflikcie do obu maszyn (Paragraf 3.3.8);



Rysunek 3.10. Diagram UML przestrzeni SyncManager.

Przed rozpoczęciem synchronizacji następuje próba "zablokowania"głównego synchronizowanego katalogu na obu maszynach. Odbywa się to poprzez dodanie wpisu do pliku `.SyncBLOCKED` znajdującego się w katalogu głównym narzędzia. Synchronizacja jest anulowana, jeżeli podczas blokowania (przykładowo `/home/userxyz/files`) już istnieje wpis zawierający:

- taką samą ścieżkę;
- podrzędny katalog ścieżki, np. `/home/userxyz/files/pictures/2021`;
- nadrzędny katalog ścieżki, np. `/home`;

Po zakończeniu synchronizacji, główny katalog synchronizowany jest w analogiczny sposób "odblokowywany". Jeżeli synchronizacja z jakiegoś powodu nie zakończyła się poprawnie (np. w przypadku awarii systemu, straty połączenia lub natychmiastowego zakończenia procesu przez użytkownika), katalogi nie mogą być automatycznie odblokowane.

Synchronizacja każdej pary plików z dopuszczalną akcją (nie może być *No Op*, *Ignore* lub *Conflict*) jest podzielona na następujące kroki:

1. Weryfikacja stanu pliku na obu maszynach. Jeżeli plik się zmienił od czasu wykonania skanu, synchronizacja pary jest anulowana, stan zmienionego pliku jest ustawiany na *Changed*, a stan postępu całej pary na *Canceled*. Aby skrócić czas sprawdzania,

program porównuje tylko rozmiar i czas modyfikacji (ten sposób wykorzystuje także wcześniej omawiany program Unison[4]).

2. Właściwa propagacja zmian. Wykonywane jest jedno z poniższych:
   - Jeżeli propagowany jest stan *Absent* — usunięcie pliku na jednej z maszyn.
   - Jeżeli propagowany jest stan *Clean*, *Dirty* lub *New* — przesłanie zawartości pliku z jednej maszyny do drugiej.
   - W przypadku akcji *Resolve* (rozwiązanie konfliktu) — zarówno zastosowanie zmian lokalnie, jak i przesłanie ich do drugiej maszyny.
3. Jeżeli podczas propagacji zmian wystąpił błąd, stan postępu pary ustawiony jest na *Failed*, a następne kroki pominięte.
4. Zaktualizowanie informacji o stanie pliku na obu maszynach (w szczególności czas modyfikacji).
5. Zapisanie wyników do bazy danych historii synchronizacji.
6. Ustawienie stanu postępu pary na *Success*.

Jeżeli wybrana akcja to *Fast Forward*, kroki 2, 3 i 4 są pomijane. Zmiany propagowane są szeregowo[43], zgodnie z kolejnością alfabetyczną ścieżek. Dzięki temu, w przypadku awarii lub błędu, nie ma potrzeby ponawiania synchronizacji dla tych zmian, które zostały poprawnie zastosowane. Aby uniknąć potencjalnej straty danych przy niedokończonym zapisie nowej zawartości, operacja zapisu jest dwufazowa. Najpierw wykonywane jest zapisywanie danych do pliku tymczasowego, następnie zamiana pliku z docelowym, która w większości przypadków może być wykonana atomowo[44]. Moduł obsługuje kontynuację przerwanego przesyłu pliku - jeżeli wykryje, że istnieje fragment odpowiedniego pliku tymczasowego na docelowej maszynie.

Jeżeli przesyłany plik ma odpowiednio duży rozmiar (>= 50 MB), plik jest kompresowany przed wysłaniem. Pozwala to zmniejszyć jego rozmiar od dwóch do czterech razy kosztem niewielkiego czasu kompresji i jeszcze mniejszego czasu dekompresji na maszynie zdalnej[2][6][13].

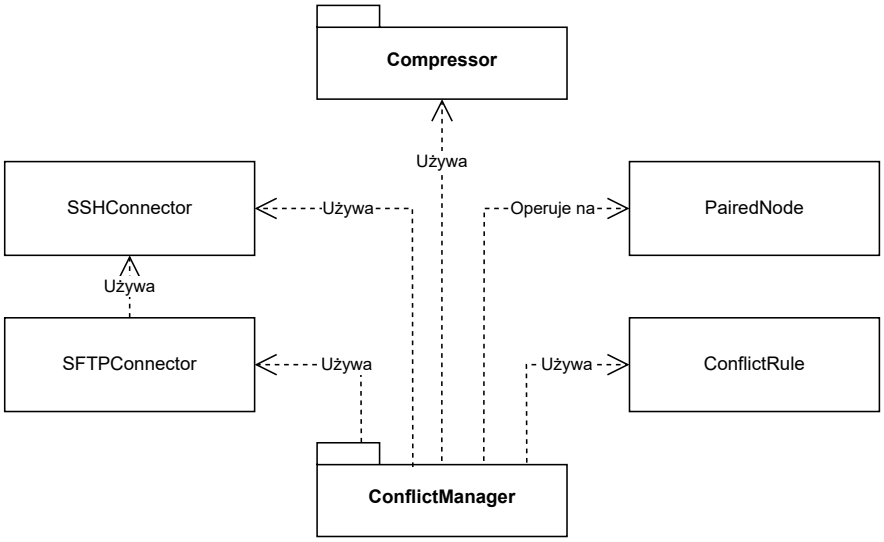
3.3.8. Moduł rozwiązywania konfliktów

Moduł (Rysunek 3.11) jest odpowiedzialny za przygotowanie tymczasowych plików zawierających konfliktujące wersje oraz umożliwianie użytkownikowi uruchamiania zewnętrznych narzędzi. Moduł może być uruchamiany przez użytkownika dla wybranych konfliktujących plików.

Wybór narzędzia dokonywany jest poprzez wybór zasady rozwiązywania konfliktów (instancji klasy *ConflictRule*), na którą składa się nadana przez użytkownika nazwa, wzorzec nazwy pliku (analogiczny do wykorzystywanego w Paragrafie 3.3.5) i polecenie do wykonania. Program oferuje dwa tryby wyboru narzędzi: ręczny i automatyczny. W trybie ręcznym użytkownik sam wybiera zasadę z listy zdefiniowanych zasad. W trybie automatycznym, moduł próbuje dopasować wzorce nazw kolejnych zasad do pliku — jeżeli znajdzie dopa-

[43] tj. tylko jedna zmiana może być jednocześnie aplikowana.

[44] Standard POSIX nie dopuszcza na przykład do atomowego przenoszenia plików pomiędzy różnymi punktami domontowań systemów plików: `https://man7.org/linux/man-pages/man2/rename.2.html`



Rysunek 3.11. Diagram UML przestrzeni ConflictManager.

sowanie, to wybrana będzie pasująca zasada, jeżeli żadna z zasad nie zostanie wybrana, moduł nie podejmie dalszych akcji.

Aby użytkownik mógł rozwiązać konflikt, obie wersje pliku muszą znajdować się na maszynie, na której działa program. Moduł automatycznie pobiera wersję z maszyny zdalnej do folderu tymczasowego (`~/.sync/tmp`), do której trafia też kopia wersji lokalnej. Pobieranie zdalnej wersji pliku jest optymalizowane tak jak w przypadku modułu synchronizacji (Paragraf 3.3.7). Tymczasowe kopie nie są usuwane do czasu wykonania synchronizacji konfliktującego pliku, dzięki czemu użytkownik nie musi rozwiązać konfliktu w ramach pojedynczej sesji programu.

Po zapewnieniu, że tymczasowe kopie konfliktujących wersji plików są dostępne, wykonywane jest polecenie przypisane do wybranej zasady rozwiązywania konfliktu. Polecenie może mieć formę dowolnego polecenia powłoki systemowej. Użytkownik może odnosić się do tymczasowych kopii plików za pomocą znaczników `$LOCAL` i `$REMOTE`, które program zastępuje odpowiednią ścieżką. Przykładowo, polecenie `meld $LOCAL $REMOTE` uruchomi narzędzie *Meld*[45] podając jako argumenty obie wersje, a `cp $LOCAL $REMOTE` nadpisze wersję zdalną wersją lokalną. Po zakończeniu wykonania polecenia, moduł sprawdza, czy oba pliki mają identyczną zawartość i nie dopuści do propagacji nowych wersji, jeżeli nie będą zgodne.

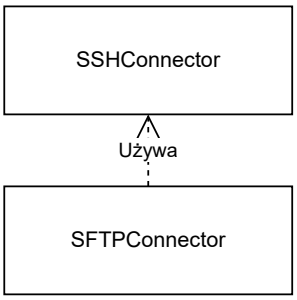
3.3.9. Moduł komunikacji SSH/SFTP

Wszystkie operacje związane z wykorzystywaniem protokołów SSH i SFTP są realizowane przez klasy *SSHConnector* i *SFTPConnector* (Rysunek 3.12).

SSHConnector pozwala połączyć się z maszyną zdalną, uwierzytelnia serwer i użytkownika i obsługuje komunikację z narzędziem na maszynie zdalnej. Narzędzie wspiera trzy główne metody uwierzytelniania, które mogą wystąpić w dowolnej kombinacji: hasło (*password*)[15], klucz publiczny (*pubkey*)[15] i wyzwanie (*keyboard-interactive*)[16].

[45] `https://meldmerge.org`

SFTPConnector wykorzystuje sesję SSH, by poprzez protokół SFTP realizować podstawowe operacje wejścia/wyjścia na plikach na maszynie zdalnej.[12]



Rysunek 3.12. Diagram UML klas SSHConnector i SFTPConnector.

3.4. Interfejs użytkownika

Jednym z głównych założeń projektowych było wymaganie, żeby narzędzie posiadało graficzny interfejs użytkownika. Na graficzny interfejs składają się przede wszystkim okno widoku głównego i okna widoków zarządzania, a także pomocnicze okna formularzy i informacyjne, które ze względu na prostotę i liczbę nie będą opisywane. Interfejs w całości napisany jest w języku angielskim.

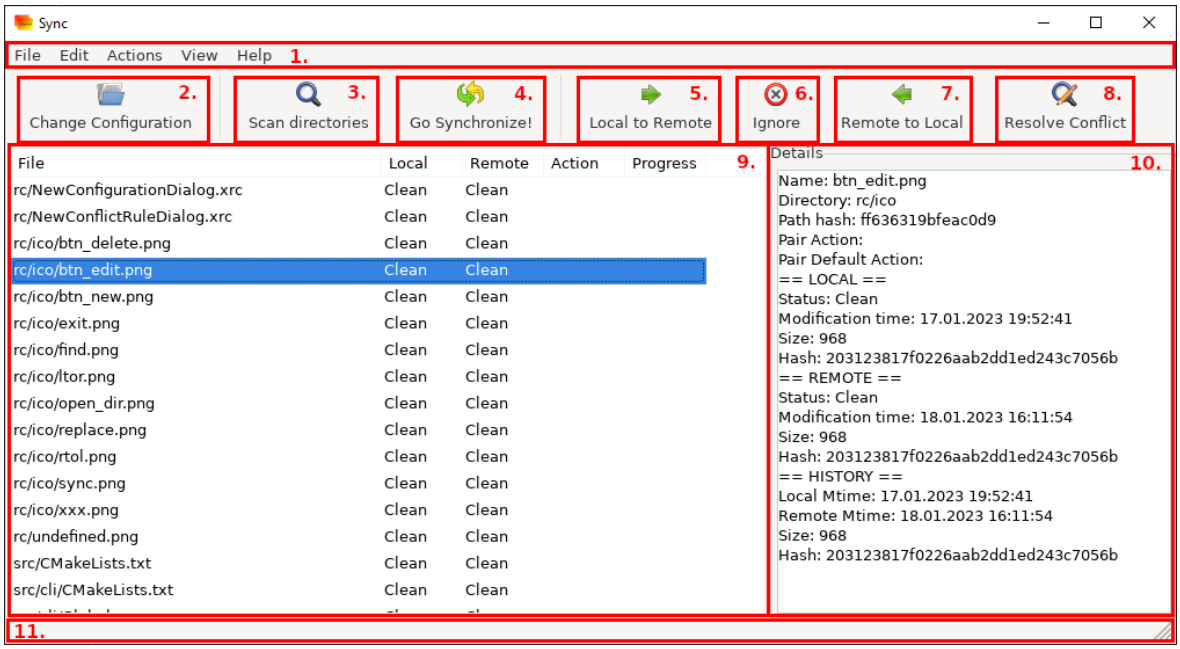
Widok główny Podczas używania narzędzia, użytkownik spędzi większość czasu w tym widoku (Rysunek 3.13). Z jego poziomu wykonuje się wszystkie akcje dotyczące synchronizacji. Nawigacja po widoku głównym może być w całości dokonana za pomocą skrótów klawiszowych — w szczególności, klawisze strzałek w połączeniu z Ctrl, Shift lub Enter pozwalają na szybkie ustawianie akcji poszczególnych plików i przeprowadzanie częściowych synchronizacji.

Lista oznaczonych na Rysunku 3.13 elementów widoku głównego:

- (1) Pasek menu - zawiera prawie wszystkie[46] dostępne akcje, jakie użytkownik może wykonać;
- (2) Przycisk otwierający okno zarządzania konfiguracjami;
- (3-4) Przyciski rozpoczynające kolejno wykrywanie zmian i synchronizowanie;
- (5-7) Przyciski wybierania typowych akcji;
- (8) Przycisk otwierający okno zarządzania zasadami rozwiązywania konfliktów;
- (9) Widok katalogu - lista wszystkich plików, ich statusów i wybranych akcji;
- (10) Widok szczegółowych informacji o zaznaczonym pliku;
- (11) Pasek stanu - przy zaznaczaniu interaktywnych elementów (lub najeżdżaniu na nie kursorem myszy) wyświetla podpowiedzi tłumaczące, co ten element robi;

Widok katalogu prezentuje zawartość wybranego w ramach konfiguracji katalogu w formie listy plików. Kolumna File zawiera ścieżkę pliku względem głównego synchronizowanego katalogu. Kolumny Local i Remote zawierają status wersji pliku (Paragraf 3.3.6

[46] Wyłączając zaznaczanie poszczególnych plików, co można robić tylko w widoku katalogu.

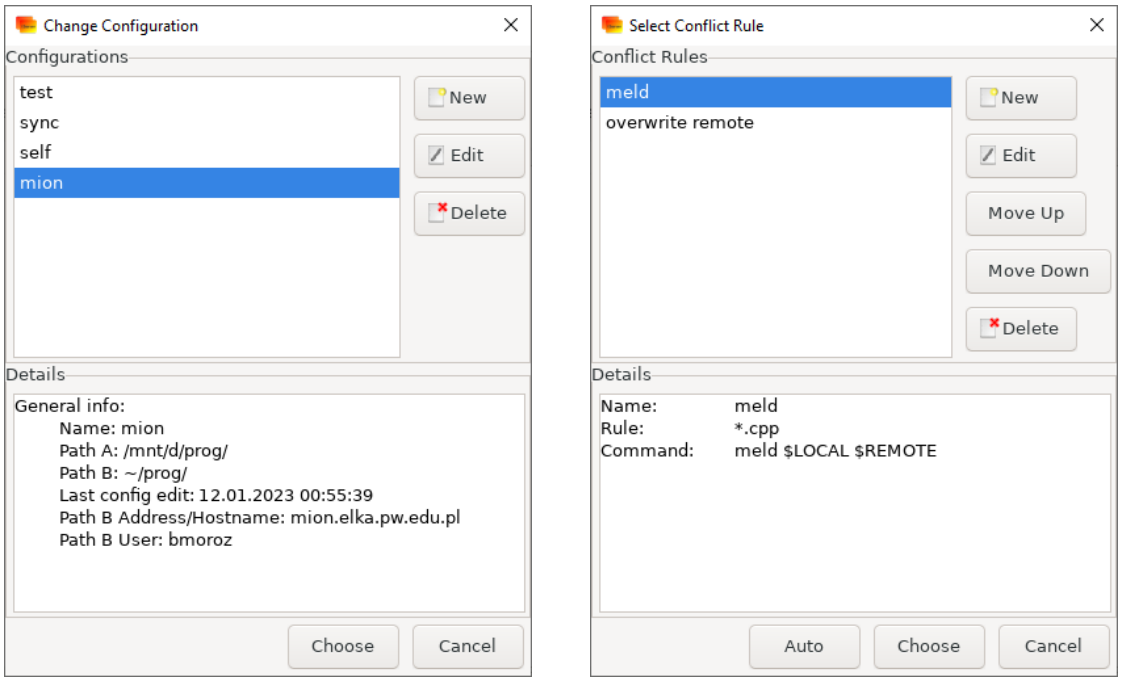


Rysunek 3.13. Główny widok aplikacji z oznaczonymi elementami.

zawiera listę możliwych statusów) na maszynie lokalnej i zdalnej. Kolumna Action zawiera wybraną dla pliku akcję do wykonania, jeżeli jego repliki nie są identyczne. Akcje propagacji zmian z jednej maszyny na drugą oznaczone są strzałkami z grotem po lewej stronie jeżeli aktualizujemy replikę lokalną, a w prawą, jeżeli replikę zdalną. Akcje wybrane przez użytkownika mają wyższy priorytet wyświetlania niż akcje automatycznie zaproponowane przez narzędzie. Kolumna Progress pokazuje rezultat synchronizacji danego pliku — *Success* jeżeli zakończyła się sukcesem, *Canceled* jeżeli została anulowana (taką sytuację opisuje Paragraf 3.3.7), *Failed* jeżeli wystąpił błąd lub awaria.

Pliki w widoku katalogu można filtrować za pomocą ustawień w menu View. Dostępne są dwa ustawienia: *Show up to date files* (pokazuj pliki bez zmian; domyślnie wyłączone) i *Show fast forwards* (pokazuj pliki o identycznej zawartości, ale różne od zapamiętanej; domyślnie włączone).

Inne widoki Rysunek 3.14 przedstawia widoki wyboru i zarządzania stworzonymi przez użytkownika konfiguracjami i zasadami rozwiązywania konfliktów. Składają się z listy istniejących obiektów, widoku szczegółowych informacji oraz przycisków inicjujących odpowiednie operacje CRUD (Create, Read, Update, Delete, z ang. *"twórz, czytaj, aktualizuj, usuwaj"*) i wyświetlających potrzebne formularze.



(a) Konfiguracje.

(b) Zasady rozwiązywania konfliktów.

Rysunek 3.14. Widoki okien zarządzania.

4. Wyniki

4.1. Weryfikacja poprawności

Przez cały proces tworzenia narzędzie podlegało weryfikacji poprawności za pomocą testów automatycznych oraz testów manualnych, aby zagwarantować, że spełnia wszystkie założenia projektowe i postawione wymagania. Oprócz tego, do wykrywania potencjalnych błędów, wykorzystywane były narzędzia opisane w Paragrafie 3.2.3;

Testy automatyczne obejmowały kluczowe elementy skanowania i synchronizacji:

- Wykrywanie plików;
- Ignorowanie plików;
- Mapowanie ścieżek plików;
- Blokowanie i odblokowywanie konfiguracji do synchronizacji;
- Wykrywanie zmian plików;
- Wykrywanie konfliktów;
- Sugerowanie domyślnych akcji;

Testy manualne obejmowały wszystkie elementy, które z różnych powodów nie mogły być weryfikowane automatycznie:

- Uwierzytelnianie serwera;
- Uwierzytelnianie użytkownika przez różne metody;
- Transfer plików;
- Poprawne działanie GUI: elementów interaktywnych, zmieniania rozmiaru okien, wyświetlania informacji, walidacji danych;

- Moduł konfliktów i uruchamianie narzędzi;
- Integralność w przypadku symulowanych awarii (odłączenie od sieci, natychmiastowe zakończenie procesu);
- Integracja wszystkich modułów w przykładowych konfiguracjach;

4.2. Badania wydajności

Aby sprawdzić praktyczność narzędzia, przeprowadzono badania wydajności. Wybrane zostały dwie metody synchronizacji, dla których porównano czas wykrywania zmian (skanowania) i czas propagacji zmian (synchronizowania):

- Synchronizacja przy użyciu własnego narzędzia.
- Synchronizacja przy użyciu narzędzia *Unison* z włączoną kompresją oraz opcją `fastcheck=false`, która zmusza *Unison* do porównywania zawartości plików (domyślnie dla systemów Unix używa tylko rozmiaru i czasu modyfikacji).

Jako punkt odniesienia wybrano naiwną synchronizację przy użyciu narzędzia *scp*[47] służącego do przesyłu plików przez protokół SSH. Narzędzie uruchamiane jest dwukrotnie, przesyłając *wszystkie pliki i podkatalogi katalogu głównego synchronizacji* z lokalizacji A do lokalizacji B i odwrotnie, z włączoną kompresją (opcja `-C`). Czas "skanowania"jest zerowy, ponieważ żadne wykrywanie zmian nie zachodzi. W konsekwencji część danych jest utracona podczas synchronizacji.

Badania przeprowadzono w obrębie sieci lokalnej o dobrej jakości połączenia (przepustowość średnio 30MB/s, dopuszczalny ping do 10ms, strata pakietów do 1%) przy porównywalnych dostępnych zasobach systemu i dyskach HDD. Maszyny były uruchamianie ponownie przed każdym testem. Każdy test był wykonany cztery razy, a wyniki przedstawione poniżej są średnią arytmetyczną wyników poszczególnych testów.

Zestawy danych

- Zestaw 1: 20 plików o rozmiarze 500MB każdy, 3 katalogi. 6 plików jest zmodyfikowanych w lokalizacji A, 6 jest zmodyfikowanych w lokalizacji B, 8 pozostaje niezmienionych. Modyfikacja danych polegała na całkowitym nadpisaniu zawartości kopiami pewnego pliku binarnego (nadającego się do kompresji, w przeciwieństwie do szumu) o rozmiarze 500MB. Zestaw sprawdza wydajność dla bardzo małej liczby plików o bardzo dużym rozmiarze.
- Zestaw 2: 200 plików o rozmiarze 50MB każdy, 3 katalogi. 60 plików jest zmodyfikowanych w lokalizacji A, 60 jest zmodyfikowanych w lokalizacji B, 80 pozostaje niezmienionych. Modyfikacja danych polegała na całkowitym nadpisaniu zawartości kopiami pewnego pliku binarnego o rozmiarze 50MB. Zestaw sprawdza wydajność dla małej liczby plików o dużym rozmiarze.
- Zestaw 3: 1,000 plików o rozmiarze 10MB każdy, 3 katalogi. 300 plików jest zmodyfikowanych w lokalizacji A, 300 jest zmodyfikowanych w lokalizacji B, 400 pozostaje niezmienionych. Modyfikacja danych polegała na całkowitym nadpisaniu zawartości

[47] *OpenSSH secure file copy*: `https://man7.org/linux/man-pages/man1/scp.1.html`

losowymi danymi. Zestaw sprawdza wydajność dla średniej liczby plików o dużym rozmiarze.

- Zestaw 4: 5,000 plików o rozmiarze 2MB każdy, 3 katalogi. 1,500 plików jest zmodyfikowanych w lokalizacji A, 1,500 jest zmodyfikowanych w lokalizacji B, 2,000 pozostaje niezmienionych. Modyfikacja danych polegała na całkowitym nadpisaniu zawartości losowymi danymi. Zestaw sprawdza wydajność dla średniej liczby plików o małym rozmiarze.
- Zestaw 5: 10,000 plików o rozmiarze 1MB każdy, 3 katalogi. 3,000 plików jest zmodyfikowanych w lokalizacji A, 3,000 jest zmodyfikowanych w lokalizacji B, 4,000 pozostaje niezmienionych. Modyfikacja danych polegała na całkowitym nadpisaniu zawartości losowymi danymi. Zestaw sprawdza wydajność dla dużej liczby plików o małym rozmiarze.
- Zestaw 6: 50,000 plików o rozmiarze 200kB każdy, 3 katalogi. 15,000 plików jest zmodyfikowanych w lokalizacji A, 15,000 jest zmodyfikowanych w lokalizacji B, 20,000 pozostaje niezmienionych. Modyfikacja danych polegała na całkowitym nadpisaniu zawartości losowymi danymi. Zestaw sprawdza wydajność dla bardzo dużej liczby plików o bardzo małym rozmiarze.

Początkowa zawartość plików została wygenerowana przez odczyt losowych danych z `/dev/urandom`. W obu zestawach mamy łącznie po 10GB danych w każdej lokalizacji i stosunek 60%:40% plików zmienionych do niezmienionych. Taki stosunek może być wyzwaniem dla narzędzi, ponieważ im większa część plików jest zmieniona, tym potencjalnie krótszy czas jest zaoszczędzony.

4.3. Wyniki przeprowadzonych badań

Tabela 4.1. Wyniki badań wydajności — naiwna synchronizacja.

L. plików / rozmiar	Czas skanowania	Czas synchronizacji	Suma
20 / 500MB	N/A	21 min. 25 s	21 min. 25 s
200 / 50MB	N/A	20 min. 11 s	20 min. 11s
1000 / 10MB	N/A	23 min. 7 s	23 min. 7 s
5000 / 2MB	N/A	24 min. 44 s	24 min. 44 s
10000 / 1MB	N/A	32 min. 2 s	32 min. 2 s
50000 / 200kB	N/A	90 min. 43 s	90 min. 43 s

Tabela 4.2. Wyniki badań wydajności — własne narzędzie.

L. plików / rozmiar	Czas skanowania	Czas synchronizacji	Suma
20 / 500MB	6 min. 1 s	6 min. 18 s	12 min. 19 s
200 / 50MB	10 min. 13 s	4 min. 3 s	14 min. 16 s
1000 / 10MB	6 min. 19 s	8 min. 20 s	14 min. 39 s
5000 / 2MB	13 min. 30 s	7 min. 11 s	20 min. 41 s
10000 / 1MB	18 min. 45 s	11 min. 10 s	29 min. 55 s
50000 / 200kB	40 min. 52 s	39 min. 26 s	80 min. 18 s

Tabela 4.3. Wyniki badań wydajności — *Unison*.

L. plików / rozmiar	Czas skanowania	Czas synchronizacji	Suma
20 / 500MB	7 min. 56 s	5 min. 21 s	13 min. 17 s
200 / 50MB	7 min. 28 s	5 min. 46 s	13 min. 14 s
1000 / 10MB	6 min. 31 s	8 min. 27 s	14 min. 58 s
5000 / 2MB	16 min. 1 s	8 min. 31 s	24 min. 32 s
10000 / 1MB	22 min. 7 s	16 min. 26 s	38 min. 33 s
50000 / 200kB	55 min. 30 s	53 min. 12 s	108 min. 42 s

Dla małej liczby plików, oba narzędzia do synchronizacji oferują znaczną poprawę wydajności w porównaniu ze zwykłym kopiowaniem plików, zajmując ok. od 35% do 45% mniej czasu. W przypadku plików o dużych rozmiarach czas skanowania i synchronizacji jest porównywalny dla obu narzędzi. Dla dużej liczby plików o niewielkim rozmiarze widać duży spadek szybkości — skanowanie i synchronizacja zajmuje 50% dłużej dla 5000 plików w porównaniu do 1000, a łączny czas działania jest porównywalny z czasem potrzebnym do przesłania wszystkich plików naiwnym kopiowaniem. Dla jeszcze większej liczby plików szybkość zauważalnie maleje dla wszystkich metod. Stosunek czasu skanowania do czasu synchronizacji nie zmienia się znacząco wraz ze wzrostem liczby plików.

W przypadku plików o dużym rozmiarze głównym wąskim gardłem podczas badań były operacje wejścia/wyjścia na dysku, który podczas skanowania/synchronizowania był używany w 100%. Średnie użycie procesora było minimalne dla obu narzędzi dla każdego zestawu, sięgając maksymalnie 3-5% użycia CPU. Rozmiar użytej pamięci RAM był zależny od liczby skanowanych plików, lecz w każdym przypadku dwu- lub trzykrotnie większy dla *Unison*.

Po przeprowadzeniu badań można wyciągnąć następujące wnioski:

- *Dla badanego scenariusza* stworzone narzędzie jest lepsze od *Unison* (dla dużej liczby plików) lub porównywalne (dla małej liczby plików).[48]
- Przy małej liczbie plików (do 1000) wraz ze wzrostem plików w katalogu czas skanowania i synchronizacji nie wzrasta znacząco;
- Przy dużej liczbie plików (5000 i więcej) proces synchronizowania *znacznie* spowalnia. Zarówno stworzone narzędzie, jak i *Unison* szybciej skanuje i propaguje pliki o dużym rozmiarze niż dużo plików o małym rozmiarze;

[48] Należy jednak zaznaczyć, że *Unison* wykorzystuje algorytm *rsync*, który przy częściowych modyfikacjach plików może potencjalnie bardziej zmniejszyć czas transmisji.

- W związku z powyższym, łączny rozmiar synchronizowanego katalogu ma drugorzędne znaczenie przy ocenianiu „kosztowności” synchronizacji. Przy poprawie wydajności nacisk powinien być kładziony na redukcję stałej składowej czasu przetwarzania pliku;
- Czas skanowania i synchronizacji jest mocno zależny od operacji na dysku — w konsekwencji niskopoziomowe optymalizacje (wykorzystywanie cache procesora, rozmiary buforów, zarządzanie stronami pamięci przez system operacyjny), a przede wszystkim zajęcie dysku przez inne procesy mają duże znaczenie. Z tego powodu czas działania narzędzi może być nieoczekiwanie krótszy lub dłuższy;

4.4. Dalszy rozwój

Prezentowane narzędzie jest w wersji podstawowej, którą można rozwijać pod każdym względem. Interfejs użytkownika może być poszerzany o dodatkowe filtry i akcje, a widoki mogą zawierać jeszcze więcej informacji. Wydajność procesów wykrywania i stosowania zmian może być znacznie poprawiona poprzez dołączenie kolejnych algorytmów do obecnych kroków lub bardziej wydajny kod operacji dyskowych. Oprócz tego, warto rozważyć wspieranie innych dystrybucji systemu Linux, inne systemy uniksopodobne lub system Windows, jako że znaczna część operacji jest niezależna od systemu operacyjnego.

Wielowątkowość Narzędzie w obecnej formie korzysta z jednego wątku do obsługi interfejsu użytkownika, przez co nie odświeża w pełni widoku podczas wykonywania operacji obciążających procesor. Wykorzystanie oddzielnych wątków na poszczególne okna jest wspierane przez bibliotekę GUI i może zwiększyć płynność działania interfejsu. Wykorzystywanie wielu wątków podczas synchronizacji także jest możliwe, lecz może nie wprowadzić znacznej poprawy, jeśli synchronizacja jest ograniczana przez czas operacji wejścia/wyjścia na dysku (co można było zaobserwować podczas badań) lub połączeniu sieciowym (co często ma miejsce przy komunikacji pomiędzy odległymi od siebie maszynami).

Tryb "szybki" Narzędzie zawsze używa wyniku funkcji skrótu zawartości plików do wykrywania zmian. Alternatywą jest wykrywanie zmian na podstawie czasu modyfikacji, które jest znacznie szybsze[1]. Jest to jednak metoda, która może dawać niepoprawne wyniki, jeśli użytkownik lub inne programy dowolnie manipulują czasem modyfikacji[4] lub jeżeli istnieją problemy z reprezentacją czasu w systemie, np. "problem roku 2038"[5]. Wzorując się na *Unison*, narzędzie można rozszerzyć o tryb "szybki"przeznaczony dla użytkowników, którym zależy wyłącznie na szybkości działania. W takim trybie porównywane by były tylko rozmiary i czas modyfikacji plików, a funkcja skrótu nie była by w ogóle używana.

Obsługa dowiązań symbolicznych W obecnej formie narzędzie ignoruje dowiązania symboliczne aby uniknąć nieskończonych pętli i zapewnić, że drzewo katalogu będzie miało jeden korzeń. Dzięki temu można uprościć operacje na ścieżkach i blokowanie innych synchronizacji. Synchronizowanie dowiązań (na dwa sposoby — faktycznej zawartości dowiązania lub ścieżki, na którą wskazuje) pozwoliłoby na uwzględnianie wielu równoległych katalogów w ramach jednej konfiguracji.

Wsparcie innych metod uwierzytelniania Narzędzie pozwala uwierzytelniać się hasłem, kluczem oraz metodą interaktywną (Paragraf 3.3.9). Są to najczęściej spotykane metody, lecz istnieją też inne, np. GSSAPI (Generic Security Services Application Program Interface). Zwiększanie liczby wspieranych metod jest pożądane, ponieważ pozwala to na używanie narzędzia w większej liczbie środowisk. Rozszerzenie wsparcia wymaga jednak zmiany biblioteki obsługującej protokół SSH.

Wykorzystanie *delta copying* (algorytm *rsync*) Algorytm *rsync* (z którego korzysta badany *Unison*) pozwala na znaczną redukcję danych wysyłanych przy synchronizacji kosztem zwiększonej złożoności obliczeniowej. Algorytm najlepiej sprawdza się dla plików o dużym rozmiarze, które niewiele się zmieniły, a w najgorszym przypadku (przesyłanie całego pliku) tworzy niewielki narzut danych związany z używanym protokołem[7]. Narzędzie obecnie nie wykorzystuje takiej metody, ponieważ protokół SFTP (którego wykorzystanie było jednym z założeń) służy do przesyłu danych do zdalnego pliku, a nie procesu, jak zakłada *rsync*. Powoduje to, że narzędzie na maszynie zdalnej musi dodatkowo przetwarzać otrzymany tymczasowy plik, potrzebując więcej operacji wejścia/wyjścia, niż algorytm teoretycznie zakłada. Ten sam problem dotyczy kompresji, lecz w jej przypadku czas dekompresji jest minimalny.

Obsługa przeniesień Zmiana nazwy pliku lub przeniesienie go do innego katalogu jest często wykonywaną czynnością podczas operacji na plikach. Stworzone narzędzie, tak jak inne narzędzia, wykrywa taką zmianę jako usunięcie pliku o starej ścieżce i stworzenie pliku o nowej ścieżce, co jest nieoptymalne.[3] Obsługa przeniesień była rozważana na wczesnych etapach rozwoju, lecz musiała zostać porzucana ze względu na znaczne skomplikowanie algorytmu parowania oraz problem z czytelną reprezentacją takiej sytuacji w interfejsie użytkownika.

5. Podsumowanie

W ramach tej pracy inżynierskiej zaprojektowano, stworzono i zbadano wydajność narzędzia do synchronizacji zawartości katalogów przez sieć. Opisane zostało zagadnienie synchronizacji plików i ogólny model działania. Na podstawie i analizy dostępnych narzędzi wyszczególniono możliwe sposoby działania i główne pożądane cechy. Przedstawione zostały przykładowe darmowe programy. Określono wymagania i ograniczenia stworzonego narzędzia.

Narzędzie zostało zaimplementowane jako zbiór modułów realizujących kolejne elementy procesu synchronizacji lub interakcję z użytkownikiem bądź systemem. Wykorzystano do tego język C++ oraz wydajne i sprawdzone komercyjnie biblioteki. Narzędzie jest przenośne pomiędzy maszynami z systemami operacyjnymi z rodziny Linux, dzięki wykorzystaniu wysoce ustandaryzowanego i dostępnego języka C++, funkcji systemowych opartych wyłącznie o standard POSIX i bibliotek przenośnych lub dostępnych jako prekompilowane, łatwe do zainstalowania pakiety. Razem z programem powstały także automatyczne testy sprawdzające poprawność zastosowanych algorytmów wykrywania zmian i sugerowania akcji oraz instrukcje instalacji i obsługi.

Powstałe narzędzie oferuje wszystkie wymagane funkcje i spełnia swoje główne zadanie, czyli szybką synchronizację dużych ilości danych. Dla badanych scenariuszy, szybkość synchronizacji jest porównywalna lub lepsza od istniejącego narzędzia *Unison*. Możliwe jest pełne lub wybiórcze propagowanie zmian i rozwiązywanie konfliktów przy pomocy zewnętrznych programów.

Narzędzie nie jest tak rozbudowane jak dostępne od lat i rozwijane przez profesjonalistów programy, ale może być dalej rozwijane pod każdym względem. Możliwe jest wprowadzenie opcjonalnego szybszego, lecz mniej niezawodnego sposobu wykrywania zmian, optymalnej obsługi przeniesień plików czy zastosowanie algorytmu *rsync* do szybszego transferu fragmentów plików.

Bibliografia

[1] Balmubramaniam S. i Pierce B. C., „What is a file synchronizer?" W *Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, 1998. adr.: https://dl.acm.org/doi/pdf/10.1145/288235.288261 (term. wiz. 27. 01. 2023).

[2] Mackerras P. i Tridgell A., „The rsync algorithm. Technical Report TR-CS-96-05," Department of Computer Science, Australian National University, raport tech. 1996.

[3] IBM Corporation, *IBM Aspera High-Speed Sync. Scalable, multi-directional synchronization of big data – over distance*, 2018. adr.: https://www.ibm.com/downloads/cas/9GDGKD0Q (term. wiz. 27. 01. 2023).

[4] B. C. Pierce, *Unison File Synchronizer. User Manual and Reference Guide*, 2016. adr.: https://www.cis.upenn.edu/~bcpierce/unison/download/releases/stable/unison-manual.html (term. wiz. 27. 01. 2023).

[5] Suzuki K., Kubota T. i Kono K., „Detecting and Analyzing Year 2038 Problem Bugs in User-Level Applications," w *2019 IEEE 24th Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2019, s. 65–6509.

[6] A. Furieri, *librasterlite2: benchmarks (2019 update)*, 2019. adr.: https://www.gaia-gis.it/fossil/librasterlite2/wiki?name=benchmarks+(2019+update) (term. wiz. 27. 01. 2023).

[7] A. Tridgell, „Efficient Algorithms for Sorting and Synchronization," prac. dokt., Australian National University, 1999.

[8] Jim T., Pierce B. C. i Vouillon J., „How to Build a File Synchronizer," 2002. adr.: http://web.mit.edu/6.033/2005/wwwdocs/papers/unisonimpl.pdf (term. wiz. 27. 01. 2023).

[9] M. P. Tridgell A., *rsync(1) manpage*, 2022. adr.: https://download.samba.org/pub/rsync/rsync.1 (term. wiz. 27. 01. 2023).

[10] The Syncthing Project, *Welcome to Syncthing's documentation! - Syncthing documentation*, 2022. adr.: https://docs.syncthing.net (term. wiz. 27. 01. 2023).

[11] Dropbox, Inc, *Porządek, bezpieczeństwo i elastyczny dostęp - Dropbox*, 2022. adr.: https://www.dropbox.com/dropbox (term. wiz. 27. 01. 2023).

[12] M. S., „SSH File Transfer Protocol," Internet Engineering Task Force, Internet-Draft, 2013. adr.: https://datatracker.ietf.org/doc/draft-moonesamy-secsh-filexfer/00/ (term. wiz. 27. 01. 2023).

[13] Meta Platforms, Inc., *Zstandard - Real-time data compression algorithm*, 2016. adr.: https://facebook.github.io/zstd/ (term. wiz. 27. 01. 2023).

[14] Y. Collet, *xxHash - Extremely fast non-cryptographic hash algorithm*, 2016. adr.: https://cyan4973.github.io/xxHash/ (term. wiz. 27. 01. 2023).

[15] Ylonen T. i Lonvick C., „The Secure Shell (SSH) Authentication Protocol," RFC 4252, 2006. adr.: https://www.rfc-editor.org/rfc/rfc4252 (term. wiz. 27. 01. 2023).

[16] Cusack F. i Forssen M., „Generic Message Exchange Authentication for the Secure Shell Protocol (SSH),” RFC 4256, 2006. adr.: https://www.rfc-editor.org/rfc/rfc4256 (term. wiz. 27. 01. 2023).

Wykaz symboli i skrótów

GUI – ang. *Graphical User Interface*, Graficzny interfejs użytkownika
SFTP – ang. *SSH File Transfer Protocol*
SSH – ang. *Secure Shell*

Spis rysunków

Spis tabel

Załącznik 1. Instrukcja używania narzędzia

Konfiguracje Tworzenie konfiguracji: skrót `Ctrl+N` lub menu *File->New Configuration*. Należy podać nazwę (Name), katalog lokalny (Root A), katalog zdalny (Root B), adres maszyny zdalnej (Address/Hostname) i użytkownik maszyny zdalnej (Username).

Wybieranie konfiguracji: skrót `Ctrl+O` lub przycisk *Change Configuration* lub menu *File->Change Configuration*.

Wykrywanie zmian / Skanowanie Uruchamianie skanowania: skrót `Ctrl+R` lub przycisk *Scan directories* lub menu *Edit->Scan directories*. Uwierzytelnianie użytkownika następuje automatycznie. Jeżeli wymagane jest hasło, wyzwanie, lub klucz jest chroniony hasłem, użytkownik zostanie poproszony o wpisanie. Po zakończeniu skanowania, pliki mogą znajdować się w następujących stanach:

- New - istnieje plik, którego nie było podczas ostatniej synchronizacji
- Absent - brak pliku
- Clean - brak zmian zawartości od ostatniej synchronizacji
- Dirty - zmiany zawartości od ostatniej synchronizacji
- Changed - stan pliku zmienił się pomiędzy skanowaniem a synchronizacją

Akcje Po zakończeniu skanowania, program automatycznie sugeruje domyślne akcje:

- Local to Remote (oznaczone ==>>) - propaguj zmiany z maszyny lokalnej na zdalną
- Remote to Local (oznaczone <<==) - propaguj zmiany z maszyny zdalnej na lokalną
- Ignore - nie rób nic
- Fast Forward - zaktualizuj sam wpis w historii pliku
- Conflict - wystąpił konflikt, żadna akcja nie może zostać podjęta (oprócz rozwiązania konfliktu)
- Resolve - propaguj zmiany po rozwiązanym konflikcie do obu maszyn

Użytkownik może ręcznie wybrać akcję, zaznaczając wiersze listy i wybierając akcję z menu *Actions* lub korzystając z przycisków *Left to Right*, *Ignore*, *Right to Left*, *Resolve Conflict* lub korzystając ze skrótów klawiszowych (w szczególności klawiszy strzałek).

Zasady ignorowania Użytkownik może zdefiniować zasady ignorowania ścieżek w plikach `.SyncBlackList` i `.SyncWhiteList` znajdujących się w głównym katalogu synchronizacji. Plik `.SyncBlackList` definiuje ścieżki, które będą zawsze pomijane podczas skanowania (w konsekwencji, program nie będzie świadom o ich istnieniu). Plik `.SyncWhiteList` definiuje wyjątki od poprzednich reguł.

Znak * oznacza "ciąg znaków dowolnej długości".

Listing 2. Przykładowe zasady ignorowania

```
1 node_modules/*
2 *.tmp
3 file.ignore.me
```

Synchronizacja Rozpoczynanie synchronizacji: skrót `Ctrl+G` lub `Enter` lub przycisk *Go Synchronize!* lub menu *Edit->Go Synchronize!*. Jeżeli użytkownik nie zaznaczył żadnych plików, synchronizowane są wszystkie, w przeciwnym wypadku tylko te zaznaczone. Pliki o dużym rozmiarze (>50MB) są kompresowane przed wysłaniem.

Program tuż przed propagacją zmian sprawdza szybko stan pliku (porównuje tylko rozmiar i czas modyfikacji). Jeżeli wykryto zmianę, synchronizacja tego pliku jest anulowana (kolumna Progress jest ustawiana na 'Canceled') i musi zostać przeskanowany ponownie.

Jeżeli z jakiegokolwiek powodu synchronizacja pliku nie zakończyła się sukcesem, kolumna *Progress* jest ustawiana na *Failed*. Bezpieczne jest ponawianie synchronizacji od razu, tzn. nie następuje utrata danych. Dla plików, których synchronizacja jest udana, kolumna *Progress* jest ustawiana na *Success* i plik na obu maszynach ma status *Clean*.

Blokowanie Aby zapewnić, że wybrany katalog główny nie jest używany przez inną instancję narzędzia, nakładana jest "blokada"w postaci wpisu do pliku `~/.sync/.SyncBLOCKED`.

Konflikty Za konflikt uważana jest sytuacja, gdy zawartość pliku jest różna na obu maszynach i w historii synchronizacji. Program domyślnie nie będzie synchronizował takich plików, chyba, że użytkownik ręcznie wybierze akcje lub zdecyduje się rozwiązać konflikt za pomocą przycisku *Resolve Conflict* (lub skrót `Shift+C`, lub menu *Actions->Resolve Conflict with a tool*). Użytkownik może wtedy wybrać jedną (lub stworzyć nową) ze zdefiniowanych zasad lub tryb Auto, który automatycznie dostosuje zasadę do pliku na podstawie nazwy.

Zasady składają się z nadanej przez użytkownika nazwy, reguły dopasowania ścieżki i polecenia do wykonania. Reguły dopasowania ścieżki są analogiczne do tych opisanych w sekcji Zasady ignorowania. Przykład: `*.txt` dopasuje się do każdego pliku z rozszerzeniem `.txt`. Polecenie może być dowolnym poleceniem powłoki, gdzie odwołujemy się do konfliktujących plików za pomocą symboli `$LOCAL` i `$REMOTE`. Przykład: `meld $LOCAL $REMOTE`.

Po wybraniu zasady, program pobierze plik z maszyny zdalnej, utworzy tymczasowe pliki w katalogu `~/.sync/tmp` (odpowiednio nazwane `xxx.SyncLOCAL` i `xxx.SyncREMOTE`, gdzie `xxx` to hash ścieżki) i wykona polecenie skojarzone z zasadą. Program będzie oczekiwał na wykonanie się polecenia (np. zamknięcie zewnętrznego narzędzia). Po pomyślnym rozwiązaniu konfliktu przez użytkownika, plik powinien zmienić przypisaną akcję na *Resolve*. Należy wtedy dokonać synchronizacji zmian (patrz sekcja Synchronizacja). Program pozwoli zsynchronizować zmiany tylko, gdy powstałe tymczasowe pliki mają identyczną zawartość.

Program zachowuje tymczasowe pliki pomiędzy sesjami, dzięki czemu użytkownik może nie musi rozwiązać wszystkich konfliktów za jednym razem. Z tego powodu program pobiera pliki z obu maszyn tylko, gdy nie znajdzie ich w katalogu `~/.sync/tmp`. W przypadku, gdy pliki zostały błędnie pobrane lub użytkownik chce z powrotem otrzymać oryginalną treść konfliktów, odpowiednie pliki `xxx.SyncLOCAL` i `xxx.SyncREMOTE` należy usunąć.

Załącznik 2. Instrukcja instalowania narzędzia

Instalowanie pakietów ogólnodostępnych

Listing 3. Skrypt powłoki instalujący potrzebne pakiety.

```
1 sudo apt-get install build-essential
2 sudo apt-get install libssh-dev
3 sudo apt-get install libgtk-3-dev
4 sudo apt-get install uuid-dev
5 sudo apt-get install cmake
```

Własne budowanie *wxWidgets* Tę sekcję można pominąć, jeżeli na maszynie głównej jest zainstalowane *wxWidgets* w wersji co najmniej 2.8.

Listing 4. Skrypt powłoki kompilujący i instalujący bibliotekę wxWidgets.

```
1 git clone --recurse-submodules\
2 https://github.com/wxWidgets/wxWidgets.git
3 cd wxWidgets
4 mkdir gtk-build
5 cd gtk-build
6 ../configure
7 make -j3    # use 3 cores. Set to the number of cores your have.
8 sudo make install
9 sudo ldconfig
```

Budowanie projektu Projekt można zbudować uruchamiając skrypt `build.sh` i podając jako jeden z argumentów `release` lub `debug`.

Przykładowe użycie: `./build.sh release -c -i`. Flaga `-c` wymusza (re)generację cache CMake (potrzebne przy pierwszym budowaniu), a flaga `-i` powoduje, że po zbudowaniu, potrzebne pliki zostaną skopiowane do `~/.sync`. Pełna instrukcja korzystania ze skryptu jest wyświetlana przy podaniu flagi `-h`.

Przygotowywanie programu do działania Na maszynie lokalnej (tej, na której będzie uruchamiane GUI), musi znajdować się plik `~/.sync/res/sync.xrs`. Zawiera on zasoby, bez których program się nie uruchomi. Plik `sync.xrs` generowany jest w ramach skryptu `build.sh` i znajduje się wraz z plikami wykonywalnymi w katalogu `release` lub `debug`, zależnie od wybranej konfiguracji. Pliki wykonywalne mogą znajdować się w dowolnym miejscu, chociaż zalecany jest katalog `~/.sync/bin`.

Do poprawnego działania, na maszynie zdalnej (tej, z którą chcemy się synchronizować) podprogram `synccli` musi być dodany do ścieżki tak, aby był wykrywalny dla sesji nieinteraktywnej (np. tworząc dowiązanie `/usr/local/bin/synccli` wskazujące na podprogram). Na maszynie zdalnej powinien działać serwer SSH.

Uruchomienie programu `syncgui` uruchomi narzędzie z GUI.