

Uniwersytet Jagielloński w Krakowie
Wydział Fizyki, Astronomii i Informatyki Stosowanej

Michał Ziobro

Nr albumu: 1089437

Biblioteka zdalnego sterowania komputera za pomocą smartfona

Praca licencjacka/
na kierunku Informatyka

Praca wykonana pod kierunkiem
dr Wiesław Chmielnicki
Zakład Technologii Gier

Kraków 2017

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Kraków, dnia

Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Kraków, dnia

Podpis kierującego pracą

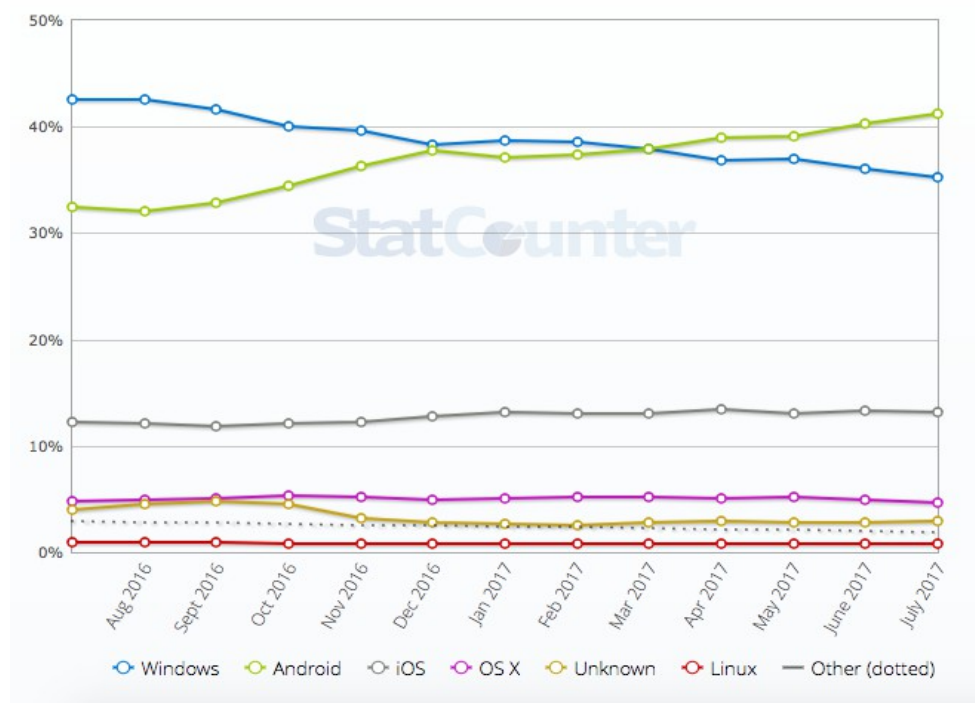
SPIS TREŚCI

1. WPROWADZENIE	4
1.1 Cel pracy	5
1.2 Zakres pracy	6
2. POSTAWY TEORETYCZNE PROGRAMOWANIA SIECIOWEGO	6
2.1 Model ISO OSI	6
2.2 Model TCP/IP	9
2.3 Protokół Internetowy IP	10
2.4 Protokół Transportowy TCP i UDP	11
2.5 Modele programów sieciowych	15
2.6 Interfejs programowania gniazd sieciowych	16
2.7 Typy serwerów	21
3. BONJOUR JAKO REALIZACJA ZERO-CONFIGURATION NETWORKING	26
3.1 Idea Zero-configuration Networking	26
3.2 Adresy link-local	28
3.3 Multicast DNS	29
3.4 DNS Service Discovery	31
3.5 Testowanie DNS SD z wiersza poleceń	32
3.6 DNS Service Discovery API	33
4. PODSTAWY TEORETYCZNE PROGRAMOWANIA DLA SYSTEMU ANDROID	34
4.1 Android SDK i architektura aplikacji	34
4.2 Komponenty Activity oraz Fragment	36
4.3 Komponenty Service oraz IntentService	42
4.4 Komponenty Intent i IntentFilter	44
4.5 Komponent BroadcastReceiver	45
4.6 Komponent ContentProvider i Loader	46
4.7 Tworzenie interfejsu użytkownika	47
4.8 Android NDK i JNI	48
5. IMPLEMENTACJA BIBLIOTEKI ZDALNEGO STEROWANIA SMARTFONEM KOMPUTERA	49
5.1 Struktura biblioteki RemoteController API	50
5.2 Implementacja Serwera	53
5.3 Pula wątków i kolejka zadań	56
5.4 Implementacja Klienta	58
5.5 Biblioteka zdarzeń klawiatury i myszy dla systemu macOS	60
5.6 Inne zdarzenia systemu macOS	63
5.7 Biblioteka zdarzeń klawiatury i myszy dla systemu Windows	63
6. ARCHITEKTURA APLIKACJI GRAFICZNYCH DLA SYSTEMU macOS	65
6.1 Tworzenie GUI w aplikacjach Cocoa	66
6.2 MVC w aplikacjach Cocoa	67

7. ARCHITEKTURA APLIKACJI DLA SYSTEMU ANDROID	69
7.1 Dodanie biblioteki natywnej zdalnego sterowania	69
7.2 Implementacja głównego Activity	69
7.3 Wyszukiwanie zdalnych urządzeń	71
7.4 Serwis klienta zdalnego sterowania	71
7.5 Interfejs kontrolera myszy	72
7.6 Interfejs kontrolera klawiatury	73
7.7 Interfejs odtwarzacza i innych zdarzeń systemowych	75
8. PODSUMOWANIE	75
8.1 Realizacja celów	75
8.2 Perspektywy rozwoju produktów w oparciu o bibliotekę zdalnego sterowania	75
BIBLIOGRAFIA	77

1. WPROWADZENIE

Współczesne urządzenia mobilne jak telefony komórkowe, tablety czy zegarki już dawno przestały być wykorzystywane jedynie do pierwotnie przypisanych im funkcji. Telefony oprócz prostego wykonywania połączeń głosowych i wysyłania wiadomości tekstowych są wszechstronnymi narzędziami służącym w codziennej pracy i rozrywce. Telefony przejmują kolejne funkcjonalności niegdyś typowe jedynie dla komputerów stacjonarnych czy laptopów. Zegarki nie służą już jedynie do sprawdzania godziny, ale pozwalają odbierać maile czy dokonywać pomiarów naszego tętna. Tablety są odpowiedzią na coraz większą potrzebę mobilności wśród użytkowników końcowych.



Rys.1 Udział w rynku systemów operacyjnych w roku 2017 [1]

Smartfony zostały spopularyzowane w końcówce lat 2000. Pod koniec 2012 roku na świecie było już ponad miliard urządzeń tego typu. Urządzenia te wyposażone są w dotykowy ekran, jedną lub więcej kamer wysokiej rozdzielczości, sensory takie jak akcelerometr, żyroskop, czy czujnik pola magnetycznego (ang. magnetic field sensor), anteny sieci WiFi, Bluetooth i 4G. W połączeniu z nowoczesnymi systemami operacyjnymi dedykowanymi urządzeniom mobilnym jak Android i iOS dają programistom niemal nieskończone możliwości tworzenia i rozwijania kolejnych aplikacji. Popularność mobilnych systemów operacyjnych jest już obecnie większa niż systemów desktopowych takich jak Windows czy Linux.

1.1 Cel pracy

W związku z ciągle rosnącą popularnością urządzeń mobilnych istnieje ciągła potrzeba poszukiwania dla nich nowych zastosowań. Potencjalnym takim polem zastosowań współczesnych telefonów jest praca zdalna z komputerami stacjonarnymi. W niniejszej pracy zostanie przeanalizowana możliwość wykorzystania smartfonu do zdalnego sterowania komputerem.

Systemy Android czy iOS udostępniając biblioteki do obsługi ekranów dotykowych w tym detekcji gestów, czujniki ruchu wydają się być wręcz idealnym środowiskiem do tworzenia tego typu aplikacji. Z drugiej strony systemy operacyjne zarówno Windows jak i macOS posiadają biblioteki umożliwiające symulowanie zdarzeń myszy czy klawiatury w stosunkowo prosty sposób.

W połączeniu z wykorzystaniem skrótów klawiszowych czy innych wywołań systemowych daje to programiście możliwość stworzenia nie tylko narzędzi zdalnej klawiatury i myszy, ale także bardziej specyficznych kontrolerów jak: pilot do odtwarzaczy wideo czy prezentacji slajdów. Istnieje również możliwość przesyłania obrazu w czasie rzeczywistym i stworzenia z telefonu czy tabletu wirtualnego ekranu dotykowego dla komputera stacjonarnego. W końcu można przerobić nasz smartfon w kontroler gier wideo. Możliwości wykorzystania telefonu, zegarka czy tabletu wydają się tutaj ograniczone jedynie wyobraźnią dewelopera.

1.2 Zakres pracy

Praca obejmuje trzy główne zagadnienia. Pierwszym jest omówienie podstaw teoretycznych związanych z funkcjonowaniem sieci lokalnej LAN, protokołów sieciowych TCP, UDP, typów architektury klient-serwer w programowaniu sieciowym, architektury aplikacji systemu Android, architektury aplikacji systemu macOS czy tworzenia, kompilacji i linkowania bibliotek w języku C. Przedstawione zostaną również podstawowe informacje na temat technologii DNS-SD oraz Bonjour.

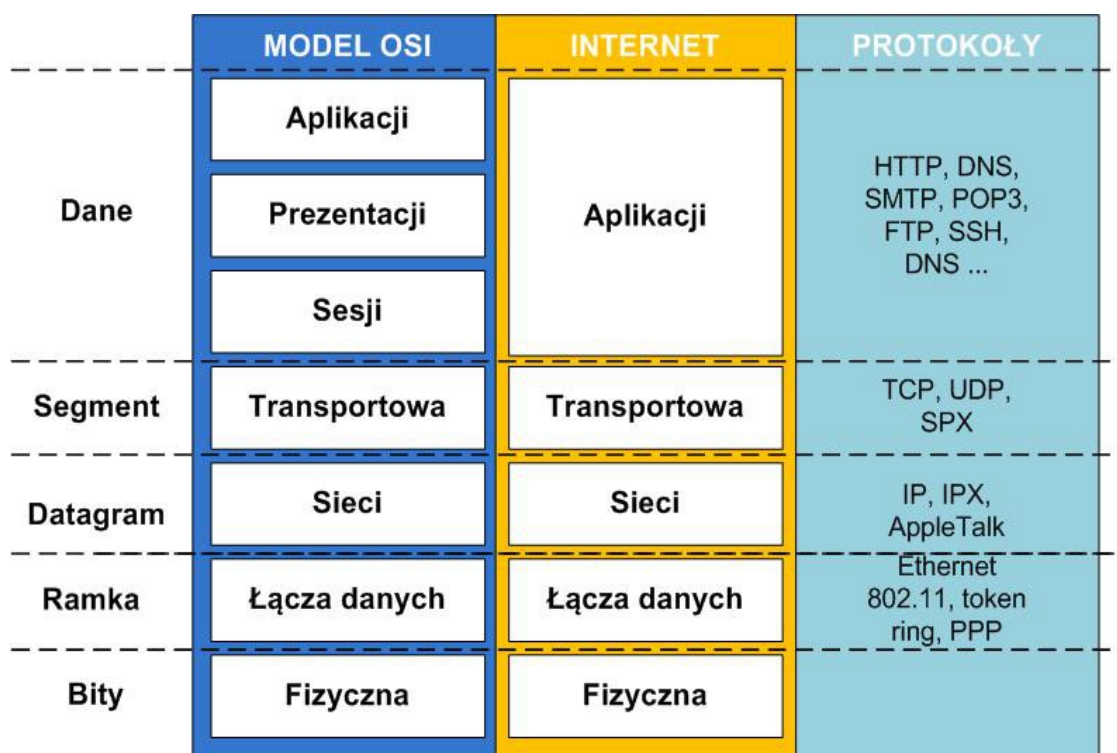
Drugim zagadnieniem będzie omówienie istotnych fragmentów implementacji biblioteki zdalnego sterowania smartfonem komputera. Ze szczególnym uwzględnieniem implementacji serwera TCP, puli wątków czy kolejki zadań w języku C. Przedstawienie wywołań systemowych z bibliotek Carbon na macOS i Win32 na system Windows, które pozwalają na symulowanie w tych systemach zdarzeń myszy i klawiatury. Omówienie możliwości wykorzystania tworzonej biblioteki do zbudowania aplikacji GUI dla systemu macOS z wykorzystaniem narzędzi takich jak Xcode i framework Cocoa. Z drugiej strony zaprezentowane zostanie wykorzystanie tworzonej biblioteki po stronie klienta w aplikacji na system Android. Poruszone zostaną tutaj zagadnienia interoperacyjności języka Java i Android SDK z kodem natywnym napisanym w języku C oraz przedstawione możliwości detekcji gestów czy ruchów żyroskopu.

Ostatnim zagadnieniem poruszonym w tym opracowaniu będzie zaprezentowanie działania aplikacji klienta na Androida i programu serwera na macOS oraz przedstawienie możliwości dalszego rozwoju produktów w oparciu o utworzoną bibliotekę zdalnego sterowania.

2. PODSTAWY TEORETYCZNE PROGRAMOWANIA SIECIOWEGO

2.1 Model ISO OSI

W odpowiedzi na potrzebę ustandaryzowania urządzeń i rozwiązań sieciowych na początku lat 80 XX wieku stworzono model OSI sieci komputerowych. Model ten obejmuje siedem warstw (Rys. 2). W każdej z warstw na komputerze źródłowym do przesyłanych danych dodawane są pewne metadane.



Rys. 2 Model ISO OSI RM [2]

Warstwa fizyczna specyfikuje typy interfejsów sieciowych, kabli czy częstotliwość sygnałów radiowych np. 2.4 GHz. Odbywa się tu fizyczna transmisja niestrukturyzowanych danych jako strumieni bitów w oparciu o teorię sygnałów. Urządzenia warstwy fizycznej to koncentratory (ang. hub), wzmacniacze sygnału (ang. repeater), okablowanie.

Warstwa łącza danych przesyła dane w postaci ramek między bezpośrednio ze sobą połączonymi węzłami sieci. Kontroluje transmisję w medium fizycznym umożliwiając naprawę błędów. Specyfikacja IEEE 802 dzieli warstwę łącza danych na dwie podwarstwy: warstwę kontroli dostępu do medium (ang. Media access control, MAC) oraz warstwę kontroli połączenia logicznego (ang. Logical link control, LLC). Pakiet warstwy sieciowej uzupełniany jest tutaj o adres fizyczny MAC. Urządzenia warstwy łącza danych to przełączniki (ang. switch), mosty (ang. bridge) oraz karty sieciowe (ang. Network Interface Card, NIC).

Warstwa sieci posiada informacje o topologii sieci i trasowaniu danych. Urządzenia pracujące w tej warstwie to routery. Najpopularniejszymi protokołami warstwy sieci są IPv4 oraz IPv6. Odpowiadają za adresowanie logiczne pakietów danych (ang. datagram). Inne protokoły warstwy sieci to IPX czy AppleTalk.

Warstwa transportowa dokonuje podziału danych na segmenty przesyłane przy użyciu protokołów transmisyjnych TCP (ang. Transmission Control Protocol) lub UDP (ang. User Datagram Protocol). Segmenty są uzupełniane o numer portu przypisujący je do konkretnej aplikacji w systemie operacyjnym.

Warstwa sesji odpowiada za synchronizację danych między nadawcą i odbiorcą. W tym celu rozpoczyna, zarządza i kończy sesję danych. Odpowiada za realizację połączeń typu: full-duplex, half-duplex czy simplex. Przykładowo warstwa sesji znajduje zastosowanie w protokole TCP to synchronizowanego zamykania połączenia czy w protokołach zdalnego wywoływania procedur (ang. RPC).

Warstwa prezentacji dokonuje mapowania różnych sieciowych formatów danych na formaty zrozumiałe przez warstwę aplikacji. Zapewnia niezależność warstwy aplikacji od reprezentacji danych występujących w różnych systemach komputerowych. Na urządzeniu źródłowym dokonuje odpowiedniego sformatowania (ang. encoding), zaszyfrowania (ang. encryption) i kompresji (ang. compression) wysyłanych danych.

Warstwa aplikacji jest najbliższą użytkownikowi końcowemu. Obejmuje protokoły komunikacyjne poziomu aplikacji, np. HTTP, SMTP, POP3. W architekturze klient-serwer w warstwie aplikacji po jednej stronie komunikacji mamy program klienta, natomiast po drugiej program serwera.

2.2 Model TCP/IP

Model TCP/IP jest modelem warstwowej struktury protokołów komunikacyjnych w sieciach komputerowych. Posiada pewne podobieństwa do modelu OSI (Rys. 3). Model ten został opracowany przez DARPA w latach 70, XX wieku. Stos TCP/IP stanowi podstawę funkcjonowania obecnego internetu. W skład modelu wchodzi cztery warstwy: aplikacji, transportowa, internetu oraz dostępu do sieci.

Warstwa dostępu do sieci (ang. link layer) obejmuje warstwę łącza danych i fizyczną z modelu OSI. Tworzone są tutaj ramki danych (ang. frame) opatrzone adresem fizycznym MAC urządzenia docelowego. Ramki są następnie przesyłane za pośrednictwem mediów transmisyjnych i kart sieciowych między interfejsami warstwy internetowej dwóch hostów znajdujących się w tej samej sieci.



Rys. 3 Porównanie modelu OSI i modelu TCP/IP [3]

Warstwa internetu (ang. Internet layer) umożliwia przesyłanie pakietów danych (ang. datagram) między potencjalnie wieloma sieciami. Odpowiada warstwie sieciowej modelu OSI. W warstwie tej mają miejsce dwie główne czynności: przypisywanie adresów logicznych IP hostom i trasowanie pakietów danych opatrzonych docelowym adresem IP z urządzenia źródłowego do urządzenia docelowego.

Warstwa transportowa (ang. transport layer) realizuje połączenie między procesami na obu końcach komunikacji. Obejmuje takie czynności jak kontrola błędów transmisji, dzielenie danych na segmenty, czy adresowanie ich numerami portów. Wyróżnia dwa typy protokołów TCP oraz UDP.

Warstwa aplikacji (ang. application layer) odpowiada trzem najwyższym warstwom modelu OSI: aplikacji, prezentacji i sesji. Definiowane są tutaj protokoły komunikacyjne usług oferowanych przez aplikacje, np: HTTP, FTP, SMTP czy DHCP.

2.3 Protokół Internetowy IP

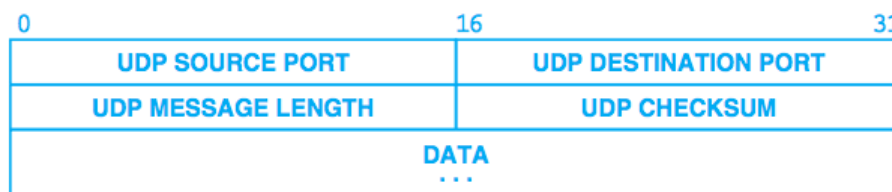
Protokół internetowy IP (ang. internet protocol) odpowiada za przesyłanie pakietów danych (ang. datagram) z węzła źródłowego do węzła docelowego z wykorzystaniem adresów logicznych IP oraz trasowania pakietów przy pomocy tablic routingu na routerach podłączonych do sieci. Protokół IP nie gwarantuje ani odpowiedniej kolejności dostarczanych pakietów ani nawet dostarczenia pakietu do odbiorcy. Niezawodność transmisji danych musi być zapewniona przez protokoły wyższych warstw stosu sieciowego. Możemy wyróżnić dwa rodzaje adresów IP i odpowiadających im nagłówek tj. IPv4 oraz IPv6.

2.4 Protokoły transportowe TCP i UDP

Protokół TCP jest protokołem połączeniowym podczas gdy UDP jest protokołem bezpołączeniowym. Poprzez wprowadzenie pojęcia portów (ang. protocol ports) umożliwiają komunikację nie tylko między hostami w sieci, ale konkretnymi aplikacjami lub procesami systemów komputerowych działających na tych hostach.

Z portami związane są gniazda sieciowe będące abstrakcyjnymi końcami komunikacji. Komunikaty (ang. messages) wysyłane za pośrednictwem protokołu TCP i UDP zawierają dwa numery portów: portu docelowego i portu źródłowego. W ten sposób aplikacja na urządzeniu docelowym ma możliwość odesłania odpowiedzi do nadawcy wiadomości.

Protokół UDP (ang. User Datagram Protocol) zapewnia komunikację między dwoma hostami na zasadach bezpołączeniowych. Protokół UDP nie gwarantuje ani dotarcia wiadomości do adresata ani poprawnej kolejności odbieranych datagramów. Nie pozwala również na kontrolę tempa przepływu pakietów między hostami. Datagram może być zgubiony, wysłany wielokrotnie (zdublowany) lub w niepoprawnej kolejności. W związku z tym odpowiednie rozwiązanie powyższych problemów spoczywa na aplikacji korzystającej z tego protokołu.



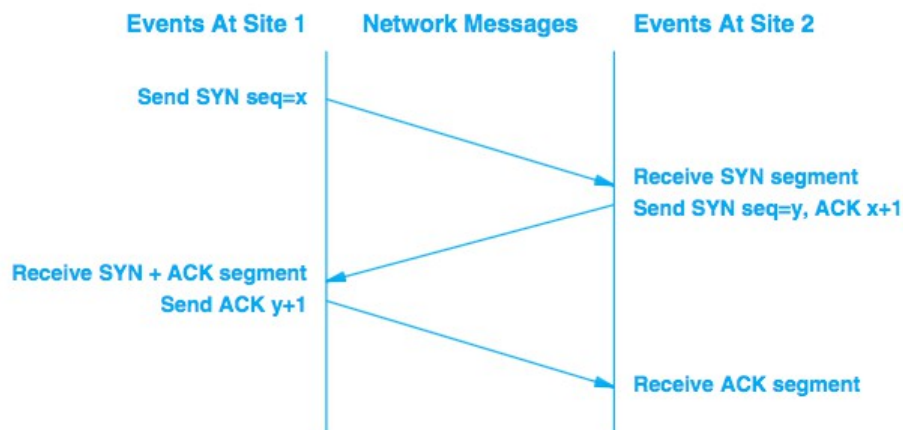
Rys. 4 Nagłówek datagramu UDP [6]

Proces tworzenia datagramu UDP w warstwie transportowej i przekazywania go do protokołu IP warstwy sieci nazywany jest multiplexingiem, proces odwrotny na hoście docelowym nazywany jest natomiast demultiplexingiem. Podczas demultiplexingu system operacyjny rozpakowuje pakiet IP i odczytuje nagłówek UDP. Na podstawie odczytanego numeru portu docelowego następuje dopasowanie wiadomości do odpowiedniej kolejki zaalokowanej w systemie (z takim samym numerem portu) na której nasłuchuje aplikacja użytkownika.

Protokół TCP (ang. Transmission Control Protocol) realizuje połączeniową komunikację między dwoma hostami w sieci zapewniając niezawodność przesyłu wiadomości. Zdejmuje z programisty aplikacji konieczność zapewnienia poprawności odbieranych danych. Protokół TCP cechują:

- przesyłanie danych jako strumieni bajtów

- tworzenie i utrzymywanie wirtualnego połączenia między hostami, uzgadnianie trójfazowe (Rys. 5)
- buforowanie danych, ich agregacja i podział na numerowane pakiety celem zwiększenia efektywności transmisji, zarządza przepływem (ang. flow control) w taki sposób by nie dochodziło do przepełnienia bufora odbiorcy



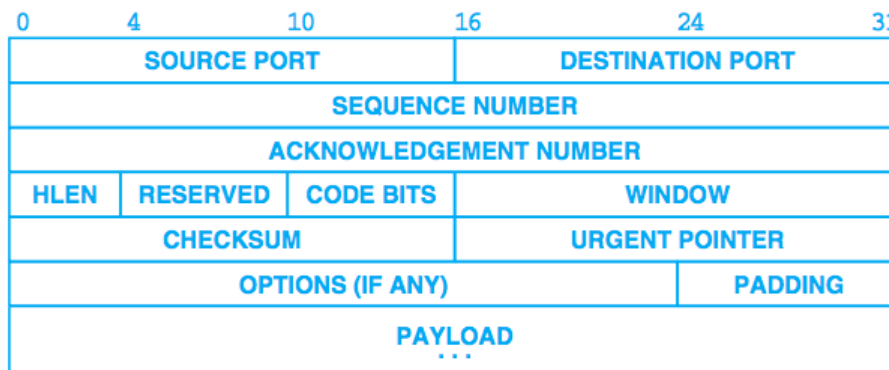
Rys. 5 Uzgadnianie trójfazowe w protokole TCP [6]

- jednoczesna komunikacja w obu kierunkach (ang. full-duplex) oraz możliwość zamknięcia połączenia w jednym z nich (ang. half-duplex)
- niezawodność przesyłu danych,
- stosowanie techniki okna przesuwającego (ang. sliding window), która zapobiega opóźnieniom wynikającym z oczekiwania na potwierdzenia od adresata.

Komunikacja TCP oparta jest na abstrakcyjnym połączeniu między dwoma aplikacjami, którego końce definiowane są przez parę liczb (host, port). Pozwala to programistom na tworzenie aplikacji współbieżnie obsługujących wielu klientów na jednym porcie. Utworzenie połączenia TCP obejmuje tzw. otwarcie pasywne i aktywne. Pasywne otwarcie polega na dowiązaniu jednej ze stron komunikacji do konkretnego numeru portu, na którym aplikacja oczekuje na przychodzące połączenia. Druga strona komunikacji nawiązując połączenie z tak utworzonym portem biernym dokonuje aktywnego otwarcia.

Protokół TCP przesyła dane w postaci segmentów. Segmenty TCP są wykorzystywane zarówno do ustanowienia połączenia, transferu danych, wysyłania potwierdzeń ACK, ogłaszania rozmiaru okna przesuwającego jak i zamykania połączenia. Segment TCP składa się z nagłówka (Rys. 6) z metadanymi oraz rzeczywistych danych. Ze względu na zmienną długość segmentów danych potwierdzenia ACK odwołują się nie do numerów segmentów, a numerów oktetów w strumieniu

bajtów.



Rys. 6 Nagłówek segmentu TCP [6]

2.5 Modele programów sieciowych

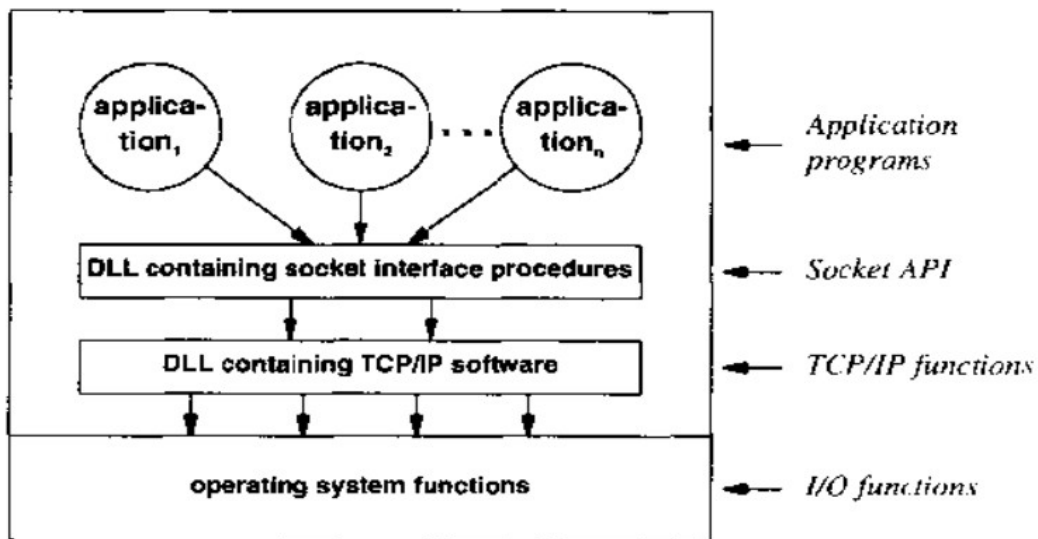
Programy sieciowe ze względu na ich architekturę możemy podzielić na dwa typy: Peer-to-Peer (P2P) oraz Klient-Serwer. Wybór architektury aplikacji sieciowej dokonywany jest przez programistę na etapie projektowania w zależności od konkretnych zastosowań.

W architekturze peer-to-peer każda ze stron komunikacji pełni równorzędną rolę. Przykładem tego typu aplikacji sieciowych są np. programy do wideokonferencji.

W architekturze klient-serwer z jednej strony komunikacji mamy program klienta, a z drugiej program serwera. Jest to najpowszechniej stosowany paradygmat programów sieciowych. Zadaniem programu serwera jest z reguły udostępnianie pewnych usług z których może skorzystać program klienta. W tego typu architekturze serwer dokonuje tzw. pasywnego otwarcia i oczekuje na przychodzące połączenia lub zapytania od klientów. To klient dokonuje aktywnego otwarcia połączenia lub wysyła zapytanie w sytuacji gdy chce skorzystać z usług udostępnianych przez serwer. Zadaniem serwera jest zaakceptowanie przychodzącego zapytania, przetworzenie go i zwrócenie stosownej odpowiedzi do klienta. Architektura klient-serwer może zostać zaimplementowana zarówno z wykorzystaniem protokołu UDP jak i protokołu TCP.

2.6 Interfejs programowania gniazd sieciowych

Implementacja programów klienta i serwera komunikujących się między sobą z wykorzystaniem protokołów TCP lub UDP odbywa się najczęściej poprzez interfejs gniazd sieciowych (ang. socket API) (Rys. 7).



Rys. 7 Dostęp do protokołów TCP/IP poprzez interfejs gniazd sieciowych [7]

Koncepcja gniazd sieciowych jest spotykana w większości języków programowania. Występujący w języku C interfejs gniazd sieciowych został historycznie zaprojektowany jako rozszerzenie funkcjonalności bibliotek wejścia/wyjścia (ang input/output). W ten sposób deskryptory plików zostały rozszerzone na komunikację sieciową. Deskryptory gniazd sieciowych pozwalają wysyłać i odbierać komunikaty w sposób analogiczny do pisanania i czytania z plików. Deskryptor pliku to liczba całkowita która identyfikuje otwarty plik lub w przypadku sieci otwarte gniazdo sieciowe.

Utworzenie gniazda sieciowego i uzyskanie powiązanego z nim deskryptora jest możliwe przy użyciu wywołania systemowego **int socket(int domain, int type, int protocol)**.

W programie serwera gniazdo sieciowe musi zostać powiązane z adresem hosta i numerem portu na którym będzie świadczona usługa. Wykorzystywana jest do tego celu funkcja **int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)**. Adres i numer portu są zdefiniowane poprzez strukturę adresową **struct sockaddr** (Rys. 8).

```
struct sockaddr {
    unsigned short    sa_family;    // address family, AF_XXX
    char              sa_data[14];  // 14 bytes of protocol address
};
```

Rys. 8 Struktura adresowa struct sockaddr [5]

W miejsce powyższej struktury stosuje się zwykle strukturę adresową odpowiadającą danej rodzinie adresowej. W przypadku protokołu IPv4 jest to struktura **struct sockaddr_in** (Rys. 9). W przypadku protokołu IPv6 jest to struktura **struct sockaddr_in6** (Rys. 10).

```
struct sockaddr_in {
    short int     sin_family;   // Address family, AF_INET
    unsigned short int sin_port; // Port number
    struct in_addr sin_addr;    // Internet address
    unsigned char  sin_zero[8]; // Same size as struct sockaddr
};
```

Rys. 9 Struktura adresowa struct sockaddr_in [5]

```
struct sockaddr_in6 {
    u_int16_t sin6_family; // address family, AF_INET6
    u_int16_t sin6_port;   // port number, Network Byte Order
    u_int32_t sin6_flowinfo; // IPv6 flow information
    struct in6_addr sin6_addr; // IPv6 address
    u_int32_t sin6_scope_id; // Scope ID
};
```

Rys. 10 Struktura adresowa struct sockaddr_in6 [5]

Istnieje również specjalna struktura **struct sockaddr_storage** będąca w stanie pomieścić zarówno struktury IPv4 jak i IPv6. Struktury adresową dla procedury bind() można ponadto uzyskać poprzez wywołanie **getaddrinfo()**. Funkcja ta zwraca listę struktur adresowych **struct addrinfo** (Rys. 11), które opakowują wymagane struktury typu **struct sockaddr**.

```
struct addrinfo {
    int ai_flags; // AI_PASSIVE, AI_CANONNAME, etc.
    int ai_family; // AF_INET, AF_INET6, AF_UNSPEC
    int ai_socktype; // SOCK_STREAM, SOCK_DGRAM
    int ai_protocol; // use 0 for "any"
    size_t ai_addrlen; // size of ai_addr in bytes
    struct sockaddr *ai_addr; // struct sockaddr_in or _in6
    char *ai_canonname; // full canonical hostname

    struct addrinfo *ai_next; // linked list, next node
};
```

Rys. 11 Struktura adresowa struct addrinfo [5]

Następnie serwer może rozpocząć nasłuchiwanie zapytań lub połączeń od klientów. Nasłuchiwanie zapytań ma miejsce w przypadku serwerów stosujących protokół bezpołączeniowy UDP, natomiast nasłuchiwanie połączeń klientów w przypadku stosowania protokołu połączeniowego TCP.

W przypadku serwerów TCP, aby rozpocząć nasłuchiwanie wymagane jest użycie dodatkowego wywołania systemowego na gnieździe sieciowym **int listen(int sockfd, int backlog)**.

Parametr **backlog** pozwala zdefiniować maksymalną długość kolejki oczekujących połączeń klientów. W przypadku przepełnienia kolejki klient może otrzymać informację o błędzie ECONNREFUSED lub podjąć próbę retransmisji zapytania. W przedstawiony powyżej sposób w programie serwera następuje otwarcie pasywne połączenia poprzez utworzenie biernego gniazda sieciowego.

Mając utworzone bierne gniazdo sieciowe, serwer TCP może rozpocząć akceptowanie przychodzących połączeń klientów wykorzystując do tego celu procedurę **int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)**. Funkcja **accept()** zdejmuje z kolejki oczekujące połączenie i tworzy dla niego aktywne gniazdo sieciowe. Wynikiem działania tej funkcji jest deskryptor nowego gniazda połączeniowego. Parametry funkcji pozwalają uzyskać informacje o danych adresowych klienta z którym nawiązano połączenie. Domyślnie funkcja **accept()** jest blokująca co oznacza, że w przypadku pustej kolejki oczekujących połączeń program serwera zostanie na niej zatrzymany do czasu pojawienia się w kolejce nowego zapytania od klienta. Bierne gniazdo sieciowe może oczywiście zostać oznaczone jako nieblokujące, w takim przypadku wywołanie funkcji **accept()** zakończone niepowodzeniem zwraca natychmiastowo błąd EAGAIN lub EWOULDBLOCK.

W przypadku serwera UDP otrzymane gniazdo serwera, a w przypadku serwera TCP otrzymane gniazdo połączenia umożliwia komunikację między klientem i serwerem. Wysyłanie zapytań i odpowiedzi możliwe jest poprzez parę funkcji **recvfrom()** i **sendto()** (Rys. 12). Wysyłanie i odbieranie wiadomości odbywa się poprzez bufor bajtów.

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
           const struct sockaddr *to, socklen_t tolen);

int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
             struct sockaddr *from, int *fromlen);
```

Rys. 12 Sygnatury funkcji **recvfrom()** i **sendto()** [5]

Podczas komunikacji połączeniowej argumenty struktur adresowych są ignorowane, w związku z czym często zamiast powyższych funkcji stosuje się ich uproszczone odpowiedniki **recv()** oraz **send()**. Dodatkowo istnieje również para funkcji **read()/write()**, które różnią się od funkcji **recv()/send()** brakiem parametru **flag**.

Wszystkie funkcje piszące do i czytające z gniazd sieciowych zwracają rzeczywistą ilość wysłanych lub odebranych bajtów danych. Funkcje te są domyślnie blokujące, co oznacza, że w przypadku braku danych w buforze lub przepełnienia bufora odpowiednio czekają na pojawienie się nowych danych do odczytu lub na zwolnienie miejsca w buforze i możliwość wysłania danych.

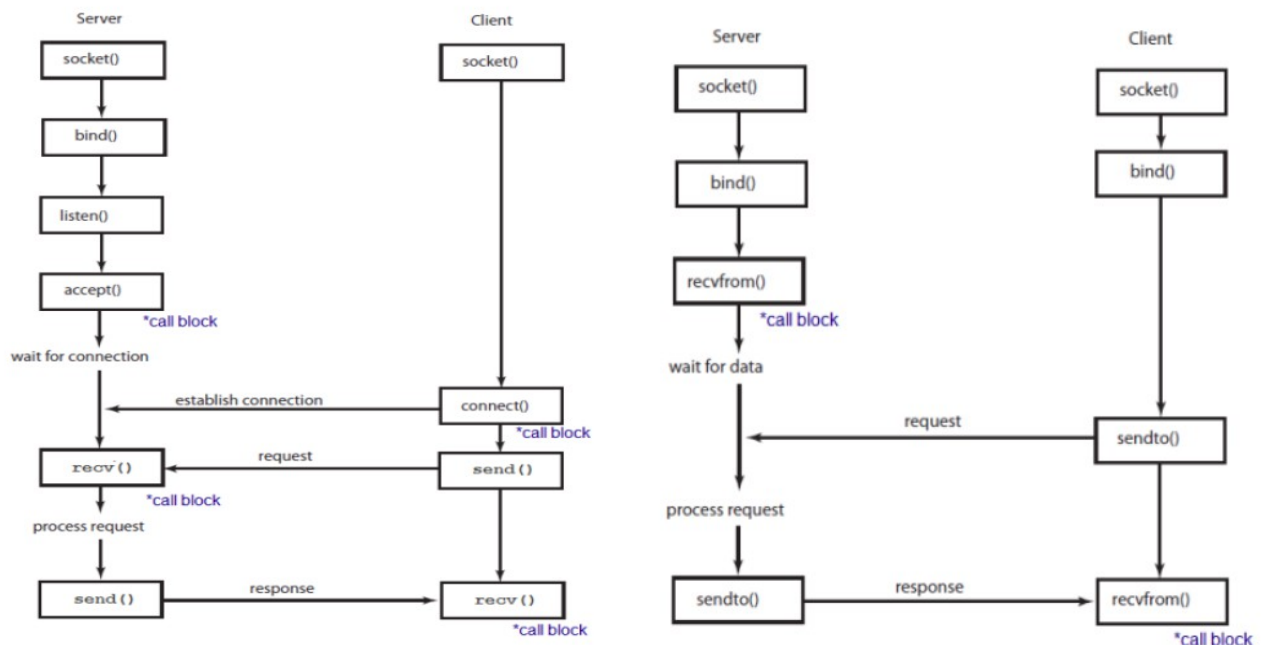
Wywołanie **fcntl(sock_fd, F_SETFL, O_NONBLOCK)** ustawia gniazdo sieciowego w

trybie nieblokującym. W przypadku gniazd nieblokujących funkcje czytające i piszące nie mają możliwości odebrania bądź wysłania danych zwracają błąd EAGAIN lub EWOULDBLOCK.

Aby komunikacja klient-serwer mogła się odbywać, program klienta musi nawiązać połączenie z serwerem przy pomocy wywołania systemowego **int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)**. Funkcja ta wiąże gniazdo sieciowe klienta z danymi adresowymi zdalnego hosta.

W przypadku klientów stosujących protokół UDP konieczne jest jawne dowiązanie gniazda sieciowego do lokalnego adresu i numeru portu przy pomocy funkcji bind().

Po zakończeniu komunikacji odbywającej się z wykorzystaniem gniazd sieciowych konieczne jest odpowiednie ich zamknięcie. W systemach unix odbywa się to w sposób analogiczny do bibliotek wejścia/wyjścia z wykorzystaniem funkcji **int close(int fd)**. Dodatkowo istnieje funkcja **int shutdown(int sockfd, int how)** dająca nieco więcej kontroli nad procesem zamykania połączenia. Funkcja ta dla wskazanego gniazda sieciowego umożliwia częściowe zamknięcie połączenia typu full-duplex, w wyniku czego otrzymujemy połączenie typu half-duplex. Parametr **how** pozwala określić następujące flagi: SHUT_RD (zamknięcie odczytu), SHUT_WR (zamknięcie zapisu), SHUT_RDWR (całkowite zamknięcie jak w przypadku close()).



Rys. 13 Model klient-serwer z użyciem protokołu TCP i UP [8]

2.7 Typy serwerów

Istnieją różne sposoby implementacji programu serwera w ramach architektury klient-serwer. Podstawowy podział wynika z wykorzystywanego protokołu warstwy transportowej. Serwer może zostać zrealizowany jako połączeniowy wykorzystujący protokół TCP lub

bezpołączeniowy stosujący protokół UDP.

Serwer połączeniowy tworzy bierne gniazdo sieciowe (ang. passive socket) powiązane z zadaniem adresem IP i numerem portu na którym nasłuchuje zgłoszeń połączeń przychodzących od klientów. Akceptując otrzymane zgłoszenie połączenia tworzy aktywne gniazdo sieciowe (ang. active socket) poprzez, które odbywa się komunikacja serwera z danym klientem. Po zakończeniu interakcji pomiędzy klientem i serwerem połączenie jest zamykane. Główne zalety tego rozwiązania to: niezawodność wynikająca z zastosowania protokołu TCP, łatwość implementacji. Do wad serwerów połączeniowych można zaliczyć zwiększone zużycie zasobów systemowych wynikające z konieczności tworzenia osobnego gniazda dla każdego z połączeń (klienta) oraz podatność na awarie aplikacji klienta.

Serwery bezpołączeniowy jak sama nazwa wskazuje umożliwia komunikację między klientem i serwerem bez potrzeby tworzenia i utrzymywania połączenia klient-serwer. Zarówno program serwera jak i klienta tworzy gniazdo sieciowe jawnie dowiązane do konkretnego adresu IP i numeru portu. W ten sposób możliwa jest komunikacja poprzez bezpośrednie wysyłanie zapytań i odpowiedzi na znany adres IP i numer portu. Do zalet serwera bezpołączeniowego można zaliczyć mniejsze zużycie zasobów systemowych, wydajniejszą transmisję danych czy możliwość rozgłaszania (ang. broadcast) komunikatów do wielu klientów. Istotną wadą takiej implementacji jest natomiast zawodność transmisji UDP, która wymaga większych nakładów pracy programisty celem zapewnienia odpowiedniej spójności danych w warstwie aplikacji. Często aplikacje, które poprawnie działają w sieciach lokalnych, stają się zawodne po rozszerzeniu działalności na sieć rozległą.

Każdy z powyższych typów serwerów ze względu na liczbę obsługiwanych klientów i wymagany czas odpowiedzi może być zaimplementowany jako: iteracyjny, współbieżny lub pseudowspółbieżny.

Serwer iteracyjny obsługuje zapytania klientów kolejno w ramach jednego wątku w pętli nieskończonej. Przepustowość takiego serwera jest ograniczona, zwłaszcza jeżeli zaistnieje potrzeba dłuższego przetwarzania zapytania, np. operacje wejścia/wyjścia z systemu plików. Może to wymagać od klientów długiego oczekiwania na obsługę zapytania i wysłanie odpowiedzi zwrotnej. Niewątpliwie do zalet serwerów iteracyjnych można zaliczyć ich prostotę.

Serwer współbieżny pozwala na jednoczesną obsługę wielu klientów czy to w osobnych wątkach czy procesach. Pomimo, że takie rozwiązanie jest trudniejsze w zaprojektowaniu i implementacji daje niekwestionowaną korzyść wynikającą z możliwości zwiększenia przepustowości serwera i skróceniu średniego czasu obsługi zapytań.

Algorytm iteracyjnego serwera połączeniowego wygląda następująco:

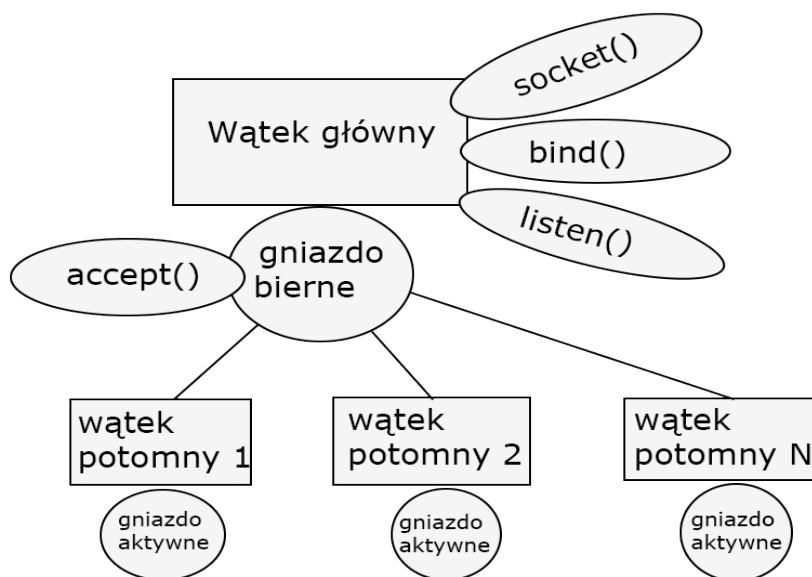
1. utworzenie biernego gniazda sieciowego powiązanego ze znanym adresem i numerem portu (**socket, bind**)
2. rozpoczęcie nasłuchiwanie zgłoszeń połączeń od klientów (**listen**)
3. w pętli nieskończonej:
 - a) akceptowanie kolejnego zgłoszenia połączenia z klientem (**accept**)
 - b) utworzenie aktywnego gniazda sieciowego dla połączenia z klientem
 - c) odbieranie zapytań od klienta i przetwarzanie ich zgodnie z protokołem oraz wysyłanie odpowiedzi do klienta (**read, write**)
 - d) zamknięcie gniazda połączenia (**close**) i powrót do pkt. 3.a)

Algorytm iteracyjnego serwera bezpołączeniowego wygląda następująco:

1. utworzenie gniazda serwera powiązanego ze znanym adresem i numerem portu (**socket, bind**)
2. w pętli nieskończonej:
 - a) odebranie zapytania od klienta, przetworzenie go zgodnie z protokołem oraz odesłanie odpowiedzi do klienta na adres podany w zapytaniu (**recvfrom, sendto**)
 - b) powrót do pkt. 2.a)

Algorytm współbieżnego serwera połączeniowego (Rys. 14) wygląda następująco:

1. utworzenie biernego gniazda sieciowego powiązanego ze znanym adresem i numerem portu (**socket, bind**)
2. rozpoczęcie nasłuchiwanie zgłoszeń połączeń od klientów (**listen**)
3. w pętli nieskończonej:
 - a) akceptowanie kolejnego zgłoszenia połączenia z klientem (**accept**)
 - b) **utworzenie nowego procesu potomnego do obsługi połączenia z klientem (fork)**
 1. utworzenie aktywnego gniazda sieciowego dla połączenia z klientem
 2. odbieranie zapytań od klienta i przetwarzanie ich zgodnie z protokołem oraz wysyłanie odpowiedzi do klienta (**read, write**)
 3. zamknięcie gniazda połączenia (**close**)
 4. **zakończenie procesu (exit)**
 - c) powrót do pkt. 3.a)



Rys. 14 Schemat współbieżnego serwera połączeniowego

Algorytm współbieżnego serwera bezpołączeniowego wygląda następująco:

1. utworzenie gniazda serwera powiązanego ze znanym adresem i numerem portu (**socket**, **bind**)
2. w pętli nieskończonej:
 - a) odebranie zapytania od klienta, (**recvfrom**)
 - b) **utworzenie nowego procesu potomnego do przetworzenia zapytania (fork)**
 1. przetworzenie zapytania np. czasochłonne operacje we/wyj
 2. sformułowanie i wysłanie odpowiedzi zgodnie z protokołem (**sendto**)
 3. **zakończenie procesu (exit)**
 - c) powrót do pkt. 2.a)

Serwer współbieżny do równoległej obsługi wielu klientów może zamiast procesów potomnych wykorzystywać wątki (ang. thread). W systemie operacyjnym liczba aktywnych procesów lub wątków jest ograniczona. W związku z tym w rzeczywistości nie jest możliwe tworzenie nowych procesów lub wątków w sposób nieskończony dla dowolnej liczby klientów. Problem ten może zostać rozwiązany poprzez wykorzystanie wzorca projektowego puli wątków (ang. thread pool).

Dodatkowo oprócz implementacji iteracyjnej i współbieżnej istnieje możliwość zaimplementowania tzw. pozornej współbieżności w ramach pojedynczego procesu. Pozorna współbieżność ma zastosowanie zwłaszcza tam gdzie koszty tworzenia nowego procesu czy wątku przewyższają wynikające z tego korzyści. Serwery tego typu sprawdzają się zwłaszcza wtedy gdy

poszczególne połączenia klientów współdzielą i wymieniają ze sobą wspólne dane. Algorytm serwera połączeniowego z pozorna współbieżnością:

1. utworzenie biernego gniazda sieciowego powiązanego ze znanym adresem i numerem portu (**socket, bind**)
2. rozpoczęcie nasłuchiwanie zgłoszeń połączeń od klientów (**listen**)
3. zainicjowanie pustego zbioru deskryptorów (**FD_ZERO**)
4. dodanie gniazda biernego do zbioru deskryptorów (**FD_SET**)
5. w pętli nieskończonej:
 - a) wybranie ze zbioru deskryptorów tych które są gotowe do odczytu (**select**)
 - b) jeżeli gniazdo bierne jest gotowe do odczytu to (**FD_ISSET**)
 1. akceptowanie nowego połączenia (**accept**)
 2. utworzenie gniazda aktywnego dla połączenia z klientem
 3. dodanie gniazda aktywnego do zbioru wszystkich deskryptorów (**FD_SET**)
 - c) dla każdego z gniazd aktywnych gotowych do odczytu:
 1. odebranie zapytania od klienta i przetworzenie go zgodnie z protokołem oraz wysyłanie odpowiedzi do klienta (**read, write**)
 2. jeżeli żądanie zamknięcia połączenie, to zamknij połączenie (**close**)
 3. powrót do podpunktu c.1)
 - d) powrót to punktu 5.a)

Zaimplementowanie serwera z pozorną współbieżnością wymaga użycia funkcji **int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)**. Funkcja ta pozwala na wybranie z zadanych zbiorów deskryptorów plików, podzbiorów gotowych odpowiednio do odczytu, zapisu lub tych dla których wystąpił błąd. Operacje na zbiorach deskryptorów można wykonywać przy użyciu makr zdefiniowanych w pliku nagłówkowym `sys/types.h`. Makro **FD_ZERO** pozwala wyzerować zbiór deskryptorów, makro **FD_SET** dodać nowy deskryptor do zbioru, makro **FD_CLR** usunąć deskryptor ze zbioru, a **FD_ISSET** sprawdzić obecność deskryptoru w zbiorze.

3. BONJOUR JAKO REALIZACJA ZERO-CONFIGURATION NETWORKING

Ręczne konfigurowanie połączenia z urządzeniami i programami sieciowymi celem dostępu do oferowanych przez nie usług sieciowych jest uciążliwe z punktu widzenia użytkownika końcowego. Dlatego wraz z rozwojem technologii sieciowych opartych o model TCP/IP pojawiły się pomysły przezwyciężenia tego problemu. Rozwiązania, które pozwalają w sposób automatyczny wyszukiwać usługi sieciowe często określa się mianem bezkonfiguracyjnych architektur sieciowych (ang. zero-configuration architecture). Przykładem takiej technologii jest Bonjour rozwijany przez firmę Apple. Pozwala on na publikowanie (rozgłaszanie) oraz odnajdowanie usług sieciowych bazujących na protokole TCP/IP zarówno w sieciach lokalnych (LAN), jak i w sieciach rozległych (WAN). Podpinając do sieci drukarkę nie jest wymagane ręczne przypisanie jej konkretnego adresu IP czy ręczne wprowadzanie adresu drukarki na każdym z komputerów. Z punktu widzenia użytkownika końcowego, widzi on dostępne w sieci drukarki w postaci listy i może wybrać przy pomocy której z nich chce wydrukować dokument. Aplikacje sieciowe wykorzystują technologię Bonjour do automatycznej detekcji innych instancji aplikacji w sieci.

3.1 Idea Zero-configuration Networking

Historycznie poszczególne platformy stosowały własne protokoły komunikacji sieciowej jak AppleTalk, IPX, NetBIOS. Z czasem większość z nich dokonała tranzykcji do protokołu internetowego IP, który coraz bardziej zyskiwał na znaczeniu i stawał się niejako standardem rozwijającego się Internetu.

W modelu TCP/IP każde z urządzeń chcących się ze sobą komunikować potrzebuje mieć przypisany unikalny adres IP. Adres IP może być przypisany statycznie lub w sposób dynamiczny poprzez serwer DHCP.

Numeryczne adresy IP są mało przyjazne z punktu widzenia użytkownika końcowego. Odpowiedzią na ten problem są serwery DNS, jednak wymagają one administratora sieci. Żmudna i czasochłonna konfiguracja rosnącej liczby urządzeń i aplikacji sieciowych stanowi niepotrzebny balast dla użytkowników końcowych, często niewielkich sieci domowych. Zeroconf jest właśnie takim brakującym ogniwem, które ma uczynić sieć TCP/IP łatwą i intuicyjną w użytkowaniu. W przypadku architektury bezkonfiguracyjnej nie jest istotne czy w sieci dostępny jest serwer DHCP, czy ktoś skonfigurował serwer DNS. Istotnym założeniem jest, że urządzenie Zeroconf ma być tak samo łatwe w użyciu jak urządzenie wpinane przez port USB, gdzie mamy do czynienia z technologią Plug&Play. Analogiczna sytuacja ma miejsce w przypadku aplikacji i usług sieciowych, które dzięki Zeroconf zyskują tą samą prostotę użytkowania.

Technologia Zeroconf składa się z trzech istotnych komponentów: adresów lokalnych dla

łącza (ang. link-local address), usługi mDNS (ang. multicast Domain Name System) oraz usługi DNS SD (ang. DNS Service Discovery). Adresy link-local to usługa, która gwarantuje przypisanie każdemu urządzeniu w sieci unikalnego adresu numerycznego. Usługa mDNS rozwiązuje z kolei problem nieintuicyjności numerycznych adresów IP. Podobnie jak w sieci Internet serwery DNS pozwalają nam odwoływać się do stron www poprzez opisowe nazwy domen, tak samo usługa mDNS pozwala odwoływać się do urządzeń sieciowych przy pomocy przyjaznych nazw. Ostatnią usługą tworzącą fundamenty architektury Zeroconf jest usługa DNS SD. DNS Service Discovery wywodzi się z dziedzictwa protokołu AppleTalk, który pozwalał na korzystanie z usług sieciowych bez potrzeby zapamiętywania enigmatycznych adresów czy nazw urządzeń.

Specyfikacja Zeroconf nie narzuca programistom stosowania żadnego konkretnego protokołu warstwy aplikacji, jedynym wymaganiem jest korzystanie z modelu TCP/IP. Adresy link-local oraz usługa mDNS powinny być dostarczane z poziomu systemu operacyjnego. Technologia Bonjour jest domyślnie wbudowana w systemach macOS, natomiast w systemach Microsoft Windows dostępna jest poprzez oprogramowanie Bonjour for Windows. Istnieją również implementacje dla platform Unixowych, wiele nowszych dystrybucji Linuxa posiada biblioteki Bonjour w standardzie. Z technologii Zeroconf można również korzystać w systemach operacyjnych na urządzenia mobilne jak Android czy iOS.

3.2 Adresy link-local

Urządzenia aby komunikować się przez sieć w modelu TCP/IP potrzebują mieć przypisane adresy IP. Adres IP może być przypisany statycznie przez administratora sieci lub przydzielony dynamicznie przez serwer DHCP. Coraz częściej mamy do czynienia z niewielkimi sieciami: urządzenia mobilne, drukarki sieciowe, kamery cyfrowe w których serwer DHCP nie jest obecny, a ręczna konfiguracja adresów jest niepraktyczna. Specyfikacja Zeroconf pozwala automatycznie uzyskać adres lokalny w przypadku braku serwera DHCP oraz bez interwencji administratora sieci.

Wybór adresu jest dokonywany w sposób rozproszony. Każde z urządzeń samo wybiera sobie losowo adres i następnie weryfikuje jego unikalność. W specyfikacji RFC 3927 dla adresów link-local zarezerwowany jest zakres 169.254.0.0 do 169.254.255.255. W systemach macOS i Windows adresy lokalne przydzielane są począwszy od roku 1998, w sytuacjach gdy komputer nie uzyskał adresu IP w żaden inny sposób. Adres link-local nie nadaje się do komunikacji w sieciach rozległych, aczkolwiek jest wystarczający do komunikacji lokalnej. Maksymalna liczba adresów link-local to 65024 kombinacje. Adresowanie link-local przewidziane jest raczej do niewielkich sieci kilkudziesięciu urządzeń bez działającego serwera DHCP lub jako zabezpieczenie w

przypadku błędów protokołu DHCP. Algorytm uzyskiwania adresu link-local wygląda następująco:

1. urządzenie losuje adres lokalny używając generatora liczb pseudolosowych
2. urządzenie wysyła zapytania sondujące ARP (ang. address resolution protocol) o adres fizyczny MAC dla wylosowanego adresu IP
3. jeżeli uda się uzyskać adres MAC dla wylosowanego adresu IP oznacza to, że jest on zajęty i należy wylosować nowy adres (pkt. 1)
4. jeżeli po kilku próbach ARP nie ujawniono konfliktu, urządzenie może zatwierdzić wylosowany adres IP jako swój adres link-local.
5. urządzenie wysyła ogłoszenie ARP o zatwierdzeniu nowego adresu link-local, które dokonuje aktualizacji informacji o tym adresie w pamięci podręcznej ARP poszczególnych urządzeń sieciowych
6. urządzenie broni adresu IP w przypadku pojawiających się zapytań sondujących ARP od nowych urządzeń

3.3 Multicast DNS

Usługa mDNS (ang. Multicast Domain Name System) to kolejny istotny komponent architektury Zeroconf. Jej działanie jest niezależne od metody pozyskania adresu IP urządzenia. Pozwala na uzyskanie lokalnie unikalnej nazwy hosta w przypadku braku serwera DNS. Istnieją następujące zalety płynące ze stosowania mDNS i opisowych nazw identyfikujących urządzenia w sieci:

- numeryczny adres IP jest tymczasowy i może ulegać zmianie wraz z upływem czasu,
- adres IP może zostać ponownie wykorzystany przez inne urządzenie w sieci, a próby odwołania się do zapamiętanego adresu IP mogą skutkować nawiązaniem połączenia z zupełnie innym urządzeniem niż oczekiwano,
- urządzenia mobilne mogą często zmieniać sieć, otrzymując w różnych sieciach zupełnie różne adresy IP. Pod tym względem nazwa hosta jest dużo bardziej stabilna niż adres numeryczny
- numeryczny adres IP jest dużo trudniejszy do zapamiętania przez człowieka niż opisowa nazwa urządzenia
- w przypadku listy urządzeń do wyboru, łatwiej jest przypisać im znaczenie na podstawie

nazwy niż adresu numerycznego.

Tradycyjne serwery DNS wymagają konfiguracji i zarządzania, są niezastąpione w sieciach rozległych i Internecie. W przypadku niewielkich sieci lokalnych stosowanie serwerów Unicast DNS jest nieopłacalne. W takich przypadkach dużo bardziej sensowne jest korzystanie z Multicast DNS, którego działanie ma charakter rozproszony, tj. nie ma centralnej jednostki zarządzającej nazwami urządzeń. Tradycyjne nazwy DNS mają hierarchiczną strukturę drzewiastą. Mapowanie nazwy www.google.com rozpoczyna się od domeny najwyższego poziomu (ang. top-level domain) com. Następnie mapowana jest nazwa domeny drugiego poziomu google, itd. W celu odróżnienia nazw lokalnych, usługa mDNS używa jako domeny najwyższego poziomu nazwy .local. Niewątpliwą zaletą stosowania nazw lokalnych jest brak organizacji rejestrującej i zarządzającej domenami, oraz wynikających z tego kosztów i komplikacji. Z drugiej strony nie ma gwarancji unikalności domeny lokalnej, pojawia się konieczność rozwiązywania konfliktów nazw w ramach jednej sieci lokalnej. Algorytm uzyskiwania lokalnie unikalnej nazwy wygląda następująco:

1. urządzenie tworzy rekord DNS z mapowaniem adresu IP na wybraną nazwę w domenie .local,
2. urządzenie wysyła zapytanie o wybraną nazwę w domenie .local na multicast'owy adres IP 224.0.0.251 (dla IPv4) lub FF02::FB (dla IPv6),
3. jeżeli po wysłaniu trzech zapytań sondujących w odstępach 250ms nie otrzymano odpowiedzi z informacją o konflikcie nazw przejdź do pkt. 6,
4. jeżeli podczas sondowania odebrano wiadomość z informacją o konflikcie nazw przejdź do pkt. 5,
5. urządzenie prosi użytkownika o wybranie nowej nazwy w domenie .local, aktualizuje rekord DNS i przechodzi do pkt. 2,
6. urządzenie kilkakrotnie wysyła ogłoszenie o zatwierdzeniu nazwy w domenie .local celem zaktualizowania pamięci podręcznej mDNS innych urządzeń w sieci lokalnej,
7. urządzenie broni swojej nazwy w domenie .local obserwując zapytania wysyłane przez inne urządzenia sieciowe, a w przypadku wykrycia konfliktu nazw odsyła stosowną odpowiedź.

Zapytania z rekordami mDNS są wysyłane przy użyciu protokołu bezpołączeniowego UDP. W związku z tym, że jest to protokół zawodny konieczne jest kilkukrotne wysłanie zapytania.

3.4 DNS Service Discovery

Mechanizm DNS SD (ang. DNS Service Discovery) korzystając z Multicast DNS zapewnia prosty i efektywny sposób na odnajdowanie usług danego typu dostępnych w sieci lokalnej. Użytkownik nie potrzebuje znać ani adresu IP ani nazwy lokalnej urządzenia świadczącego usługę. Nie potrzebuje nawet wiedzieć jak nazywają się poszczególne usługi danego typu. Technologia DNS SD bazuje na standardowych zapytaniach DNS i typach rekordów DNS. W celu wyszukiwania usług może korzystać zarówno z link-local Multicast DNS jak i global Unicast DNS. Rozgłaszanie usług w sieci odbywa się z wykorzystaniem Multicast DNS lub DNS Dynamic Update.

Oprogramowanie korzystające z interfejsu DNS Service Discovery umożliwia odnajdowanie i wyświetlanie listy usług danego typu oraz mapowanie wybranej usługi na konkretny adres IP i numer portu. Przykładowo dzięki DNS SD możliwe jest wyszukiwanie usług sieciowych które umożliwiają wydrukowanie dokumentu. Istotne jest tutaj podkreślenie, że wyszukiwane są usługi, a nie urządzenia. Jest to podejście zapewniające dużo większą elastyczność.

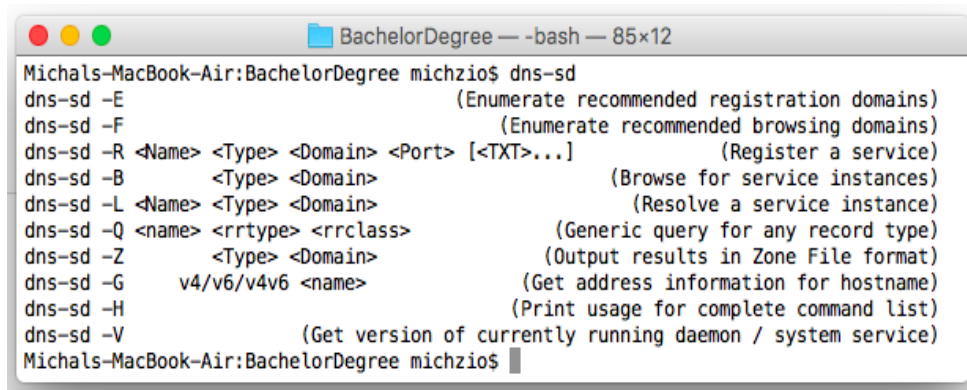
Typ usługi rozgłaszanej przy pomocy mechanizmu DNS SD składa się z reguły z dwóch części opisujących co dana usługa oferuje, oraz jaki protokół jest stosowany do realizacji tejże usługi. Przykładowo usługa może być typu: `_remotely-click._tcp`. Taka usługa oferuje funkcjonalność zdalnego sterowania komputerem stosując w tym celu protokół TCP. Z punktu widzenia wyszukiwania usług istotne jest aby typ usługi był jednoznaczny, tzn. dwie aplikacje świadczące kompletnie różne usługi nie powinny posługiwać się tym samym typem. W związku z tym został utworzony darmowy rejestr typów usług <http://www.dns-sd.org/ServiceTypes.html>. Nazwa typu usługi może składać się z maksymalnie 14 znaków: małe litery (a-z), cyfry (0-9) oraz pauza (-).

DNS Service Discovery wewnętrznie wykorzystuje protokół DNS. Wynika to z następujących cech protokołu DNS:

- protokół DNS jest protokołem zapytań
- protokół DNS posiada centralny serwer DNS potrzebny do realizacji wyszukiwania usług,
- DNS Dynamic Update umożliwia realizację protokołu rejestracji usług,
- protokół DNS posiada mechanizm bezpieczeństwa DNSSEC.
- wersja Multicast DNS protokołu pozwala na wyszukiwanie usług w środowisku sieci link-local.
- protokół DNS posiada zdefiniowany rekord typu SRV, możliwy do wykorzystania przy odnajdowaniu usług.

3.5 Testowanie DNS SD z wiersza poleceń

Najprostszym sposobem na przetestowanie działania DNS Service Discovery jest użycie narzędzia `dns-sd` z wiersza poleceń. Pozwala ono wykonać następujące operacje: publikowanie usług, przeglądanie usług oraz mapowanie usługi na dane adresowe.



```

Michals-MacBook-Air:~$ dns-sd
dns-sd -E                      (Enumerate recommended registration domains)
dns-sd -F                      (Enumerate recommended browsing domains)
dns-sd -R <Name> <Type> <Domain> <Port> [<TXT>...] (Register a service)
dns-sd -B <Type> <Domain>      (Browse for service instances)
dns-sd -L <Name> <Type> <Domain> (Resolve a service instance)
dns-sd -Q <name> <rrtype> <rrclass> (Generic query for any record type)
dns-sd -Z <Type> <Domain>      (Output results in Zone File format)
dns-sd -G v4/v6/v4v6 <name>    (Get address information for hostname)
dns-sd -H                      (Print usage for complete command list)
dns-sd -V                      (Get version of currently running daemon / system service)
Michals-MacBook-Air:~$

```

Rys. 15 Narzędzie `dns-sd` do testowania

Aby wyszukiwać usługi danego typu w zadanej domenie można użyć następującego polecenia linii komend: **`dns-sd -B <Type> <Domain>`**. Pominięcie argumentu `domain` oznacza wskazanie domyślnej domeny czyli `local`. Znaną usługę można zmapować na dane adresowe przy pomocy komendy: **`dns-sd -L <Name> <Type> <Domain>`**. Rejestracja usługi w celu jej rozgłaszania możliwa jest przy pomocy wywołania: **`dns-sd -R <Name> <Type> <Domain> <Port> [<TXT> ...]`**.

3.6 DNS Service Discovery API

Interfejs programistyczny usługi DNS Service Discovery dostępny jest w wielu językach programowania w tym między innymi: C, Java, Python, Objective-C, Swift. DNS SD API zostało zaprojektowane jako asynchroniczne tj. po rozpoczęciu nasłuchiwanie usług danego typu, biblioteka powiadamia aplikację o pojawieniu się/zniknięciu danej usługi poprzez funkcje callback'owe. Przykładowo w środowisku Cocoa podstawowe operacje protokołu DNS SD mogą zostać zainicjowane przy pomocy obiektów klas **`NetService`** (Rys. 16) oraz **`NetServiceBrowser`**. Natomiast asynchroniczne powiadomienie o zdarzeniach odbywa się poprzez implementację metod z delegat **`NetServiceDelegate`** oraz **`NetServiceBrowserDelegate`**.

```
func registerService(domain: String, type: String, name: String, port: Int32) -> Void {  
  
    DispatchQueue.global(qos: .userInitiated).async {  
  
        self.netService = NetService(domain: domain, type: type, name: name, port: port);  
        self.netService?.delegate = self;  
        self.netService?.startMonitoring();  
        self.netService?.publish();  
  
        RunLoop.current.run();  
    }  
}  
  
func unregisterService() {  
    netService?.stop();  
    self.delegate?.serviceUnregistered?(netService!);  
}
```

Rys. 16 Publikowanie usługi w sieci lokalnej
z użyciem obiektu NetService w Cocoa/Swift

Dodatkowo na platformie Android dostępne jest kompatybilne z technologią Bonjour API: Network Service Discovery.

4. PODSTAWY TEORETYCZNE PROGRAMOWANIA DLA SYSTEMU ANDROID

Android to obecnie najpopularniejszy system operacyjny na urządzenia mobilne. Google szacuje, że z ich systemu aktywnie korzysta ponad 2 miliardy użytkowników miesięcznie na całym świecie. Początki Androida sięgają roku 2003 kiedy to powstała firma Android Inc. założona przez Andy Rubin'a i Rich Miner'a. Jej celem było stworzenie inteligentnego systemu operacyjnego na urządzenia przenośne takie jak kamery cyfrowe, a w późniejszym czasie również telefony komórkowe. Dynamiczny rozwój systemu Android nastąpił po przejęciu firmy w 2005 roku przez Google. Android oparty jest o jądro Linuxa (ang. Linux kernel) i zaprojektowany pod kątem urządzeń mobilnych oferując wsparcie dla ekranów dotykowych czy rozpoznawania mowy. Z czasem zasięg platformy Android rozszerzył się ze smartfonów i tabletów na urządzenia takie jak zegarki (Android Wear), telewizory (Android TV) czy obecnie samochody (Android Auto).

Aplikacje na system Android pisane są z wykorzystaniem narzędzi wchodzących w skład Android SDK (ang. Software Development Kit). Programiści mają do wyboru dwa języki programowania Java, oraz od niedawna Kotlin. Dodatkowo poprzez Android NDK istnieje wsparcie dla języków i bibliotek natywnych C/C++. Począwszy od grudnia 2014 roku do rozwoju aplikacji Google oficjalnie zaleca stosowanie zintegrowanego środowiska programistycznego Android Studio bazującego na IDE IntelliJ IDEA od firmy JetBrains. Istnieje również emulator AVD systemu Android pozwalający testować większość funkcjonalności systemu bez konieczności posiadania fizycznego urządzenia.

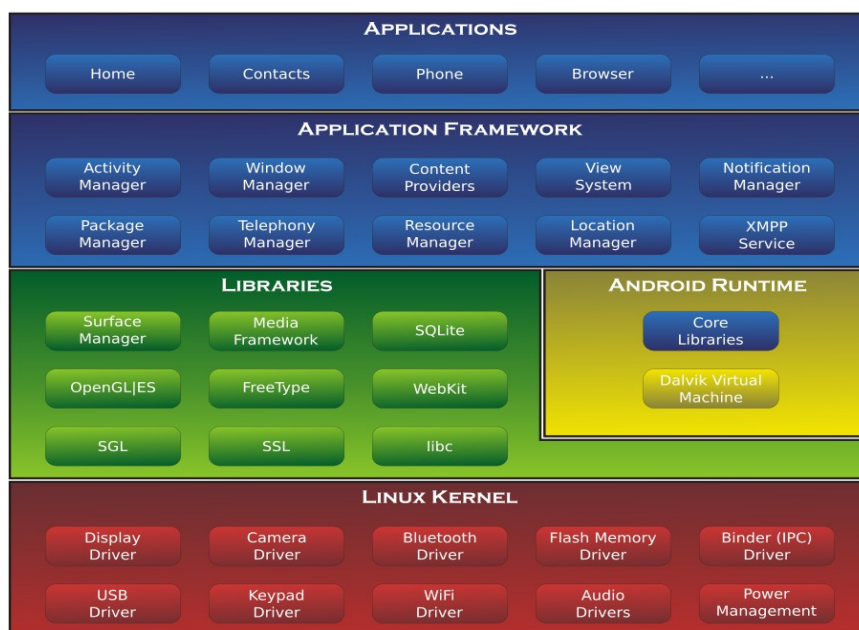
4.1 Android SDK i architektura aplikacji

Podstawę programowania aplikacji dla systemu Android stanowi zbiór narzędzi oferowanych przez Android SDK (Rys. 17). Jest to ogromna kolekcja bibliotek pozwalająca tworzyć w sposób wydajny bardzo zróżnicowane i zaawansowane aplikacje. W skład systemu Android wchodzi między innymi takie biblioteki jak: SQLite (do zarządzania bazami danych), OpenGL ES (do tworzenia grafiki 3D) czy SSL. Należą one do warstwy bibliotek napisanych w języku C ponad którymi istnieje interfejs programistyczny w języku Java wraz z innymi framework'ami w języku Java. Całość systemu bazuje na jądrze Linuxa.

Architektura aplikacji pisanych dla systemu Android opiera się o kilka kluczowych reużytkowalnych komponentów: Intent, Intent Filter, Activity, Fragment, Loader, Service, Content Provider, Broadcast Receiver. Wiele aplikacji stosuje ponadto jeden z dwóch popularnych wzorców projektowych: MVC (ang. Model View Controller) lub MVVM (ang. Model View ViewModel).

Wzorec MVC dzieli klasy aplikacji na trzy grupy. Model to klasy reprezentujące encje danych. View to klasy reprezentujące widok czyli wszystko co użytkownik może zobaczyć na

ekranie. W tym przypadku spora część graficznego interfejsu użytkownika jest projektowana przy pomocy deklaratywnie zdefiniowanego kodu XML w postaci plików w folderze res/layout. Android Studio umożliwia graficzne tworzenie interfejsów użytkownika przy pomocy predefiniowanych widgetów. Controller to grupa klas, która łączy ze sobą Model i View. Głównym zadaniem tych klas jest aktualizowanie widoków, pobieranie danych od użytkownika i modyfikowanie modelu. Komponentami Androida, które można by utożsamiać z kontrolerami są Activity oraz Fragment.



Rys. 17 Stos bibliotek systemu operacyjnego Android [11]

W przypadku wzorca projektowego MVVM spora część kodu kontrolerów zostaje zastąpiona bindowaniem widoków (View) do modelu (Model). Bindowanie umożliwia reaktywne aktualizowanie modelu na podstawie danych wprowadzanych przez użytkownika oraz z drugiej strony aktualizowanie widoków w sytuacji zmian danych w modelu. Dodatkowo dochodzi nam grupa klas ViewModel, które przechowują i zarządzają danymi bezpośrednio związanymi z konkretnym interfejsem użytkownika.

Większość zasobów (ang. resource) aplikacji jak: layouty, pliki graficzne (ang. drawables), animacje, kolory, menu, wartości tekstowe (ang. string), style, czcionki jest definiowanych w postaci plików XML i zapisywanych w katalogu /res aplikacji. Dostęp do tych zasobów jest możliwy poprzez dynamicznie generowaną podczas kompilacji klasę R.

Istotnym obiektem w aplikacjach androidowych jest Context. Context zapewnia interfejs do globalnych informacji dotyczących środowiska aplikacji. Umożliwia on dostęp do zasobów aplikacji oraz wykonywanie operacji takich jak uruchamianie Activity, czy rozgłaszanie (ang.

broadcast) i odbieranie (ang. receive) Intent'ów.

Dużo usług systemu operacyjnego dostępnych jest poprzez spójny interfejs tj. metodę **getSystemService()** wywoływana na obiekcie kontekstu. W ten sposób możliwe jest uzyskanie menadżerów między innymi następujących usług: **ActivityManager**, **AlarmManager**, **AudioManager**, **BatteryManager**, **BluetoothManager**, **CameraManager**, **ClipboardManager**, **ConnectivityManager**, **NotificationManager**, **LocationManager**, **NfcManager**, **NsdManager**, **SensorManager**, **TelephonyManager**, czy inne. Dystrybucja aplikacji odbywa się w postaci plików .apk (ang. application package).

4.2 Komponenty Activity oraz Fragment

Podstawowymi komponentami praktycznie każdej aplikacji androidowej są Activity i Fragment. Activity to komponent który reprezentuje pojedynczy ekran aplikacji. Służy jako punkt wejściowy dla wszelkiej interakcji użytkownika z konkretnym ekranem aplikacji, podobnie jak funkcja **main()** w przypadku programów konsolowych. Definiuje przepływ nawigacji wewnątrz aplikacji. Implementacja Activity odbywa się poprzez rozszerzenie klasy Activity dostępnej w Android SDK. Większość aplikacji posiada wiele ekranów z różnymi interfejsami użytkownika co wymaga tworzeniu wielu komponentów Activity w ramach pojedynczej aplikacji. Każde Activity, aby mogło być używane przez aplikację musi zostać zarejestrowane w pliku konfiguracyjnym **AndroidManifest.xml** (Rys. 18). Poszczególne Activity są z reguły niezależne od siebie (ang. decoupled).

```
<manifest ... >
  <application ... >
    <activity android:name=".ExampleActivity" />
    ...
  </application ... >
  ...
</manifest >
```

Rys. 18 Deklarowanie Activity w AndroidManifest.xml [14]

Każda instancja Activity przechodzi przez różne stany w czasie swojego życia. Na każdym etapie cyklu życiowego (Rys. 19) wywoływana jest odpowiednia metoda callback'owa pozwalająca programiście odpowiednio zmodyfikować zachowanie konkretnego Activity.

Metoda **onCreate()** jest wywoływana w momencie pierwszego utworzenia nowego obiektu Activity. Na tym etapie następuje powiązanie Activity z konkretnym widokiem przy pomocy wywołania **setContentView()**. Widok ten może być zdefiniowany programistycznie w kodzie lub

deklaratywnie w postaci pliku XML w katalogu res/layout. Na tym etapie następuje inicjalizacja wszystkich niezbędnych elementów interfejsu użytkownika oraz obiektów modelu danych wykorzystywanych do wyświetlenia bieżącego ekranu aplikacji. Do metody jako argument przekazywany jest obiekt **savedInstanceState** typu **Bundle**. Zawiera on dane na temat stanu ostatniej instancji bieżącego Activity i może być wykorzystany w celu odtworzenia dotychczasowego ekranu np. w przypadku zmiany jego orientacji.

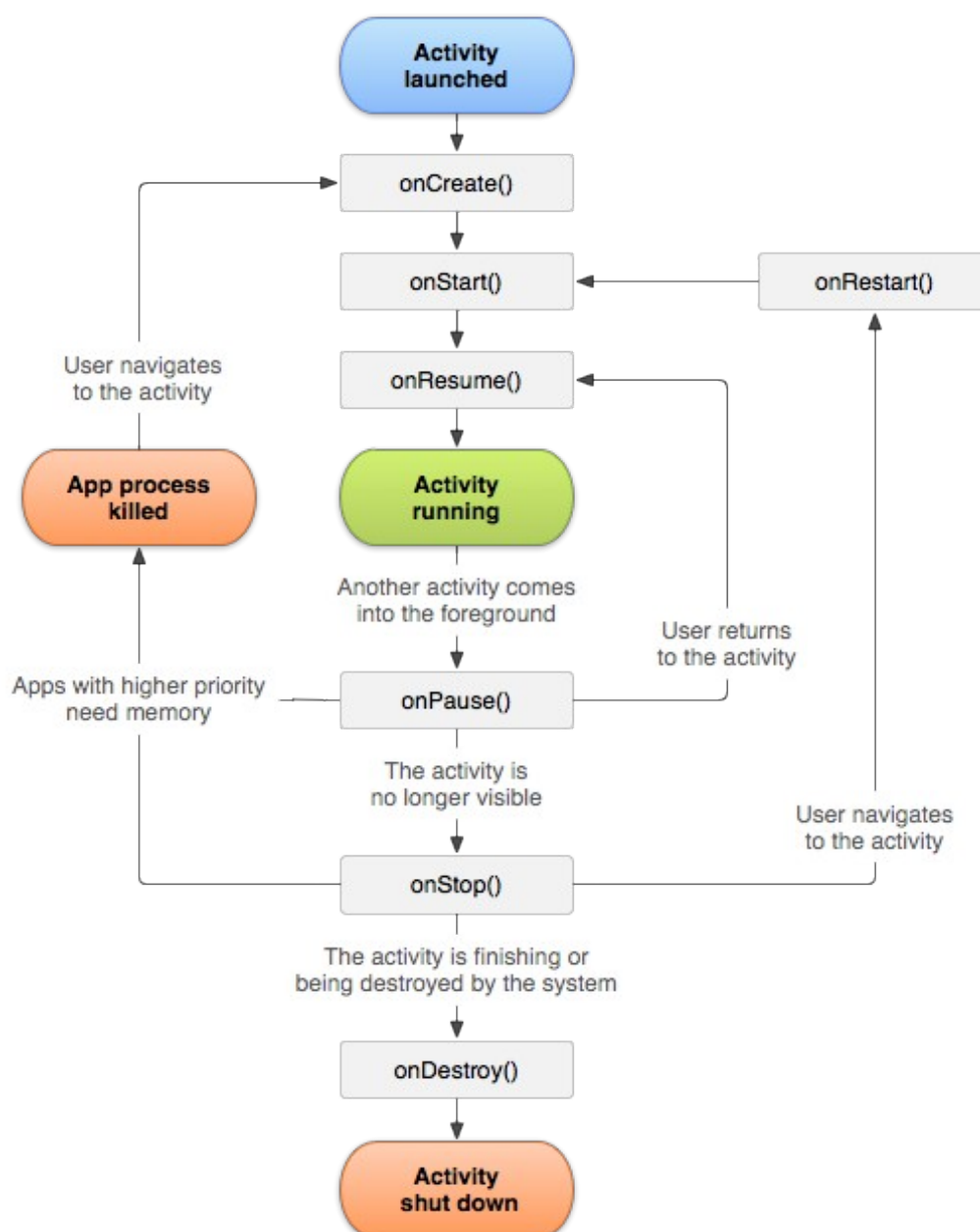
Metoda **onStart()** jest wywoływana w momencie gdy Activity staje się widoczne dla użytkownika. Na tym etapie zalecane jest uruchomienie animacji, plików audio, zarejestrowanie BroadcastReceiver'ów oraz dokonanie wszelkich czynności wymagających wyświetlenia zawartości na ekranie.

Metoda **onResume()** jest wywoływana w momencie gdy Activity trafia na pierwszy plan (ang. foreground), a użytkownik rozpoczyna z nim interakcję. Aplikacja pozostaje w tym stanie do czasu gdy wydarzy się coś co przerwie interakcję z użytkownikiem np. przychodząca rozmowa telefoniczna lub wygaszenie ekranu. Metoda ta jest dobrym miejscem na wznowienie wszelkich czynności wstrzymanych w metodzie **onPause()**, tj. restartowanie animacji, widoku kamery, wznowienie odtwarzania audio/wideo, etc.

Metoda **onPause()** jest wywoływana tuż przed przejściem Activity do drugiego planu (ang. background). W tej metodzie należy zawiesić czynności takie jak animacje, odtwarzanie muzyki, zwolnić zasoby systemowe jak broadcast receiver'y, uchwyty czujników (ang. sensor) oraz inne które nadmiernie zużywają baterię. Nie jest to natomiast miejsce w którym powinno wykonywać się operacje zapisywania danych użytkownika czy aplikacji lub wykonywać transakcje bazodanowe jako, że czas wykonania tej metody powinien być krótki. Wszystkie operacje tego typu powinny być wykonywane w metodzie **onStop()**. Ze stanu zawieszenia (ang. paused state) Activity może przejść albo do stanu wznowienia (ang. resumed state), albo zostać zatrzymane (ang. stopped state).

Metoda **onStop()** jest wywoływana zaraz po metodzie **onPause()** w sytuacji kiedy Activity przestaje być widoczne. Ma to miejsce w momencie gdy na ekranie pojawia się zupełnie nowe Activity w całości go zakrywające lub gdy dotychczasowe Activity będzie kończyć swoje działanie wywołaniem **onDestroy()**. Jest to dobre miejsce żeby zapisać stan aplikacji oraz dane użytkownika na dysku czy w bazie danych. Powinny zostać tutaj zwolnione wszystkie niepotrzebne zasoby (np. broadcast receiver'y) aby nie dopuścić do wycieku pamięci, jako że metoda **onDestroy()** niekiedy może zostać pominięta. Jeżeli działanie Activity będzie wznowione zostanie wywołana metoda **onRestart()**.

Metoda **onRestart()** jest wywoływana pomiędzy zatrzymaniem Activity (ang. stopped state), a ponownym jego uruchomieniem (ang. started state).



Rys. 19. Cykl życiowy Activity [14]

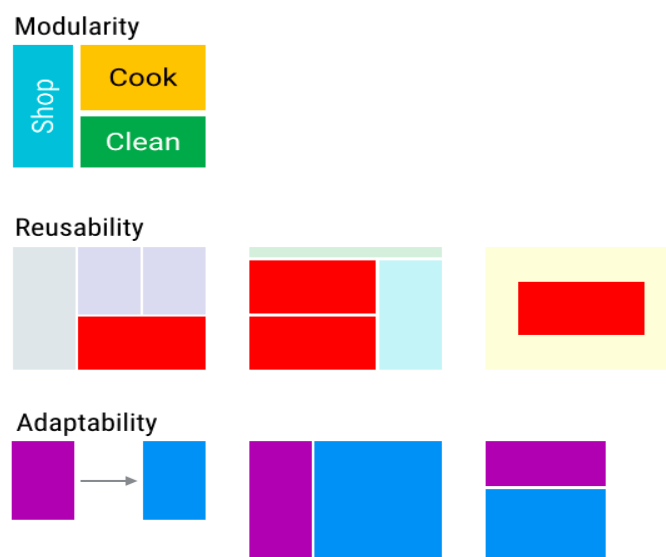
Metoda **onDestroy()** jest wywoływana tuż przed usunięciem obiektu Activity. Niszczenie obiektu może być zapoczątkowane przez system operacyjny w sytuacji niedoboru pamięci lub przez użytkownika wywołaniem metody **finish()**. Istnieje możliwość rozróżnienia pomiędzy tymi dwoma scenariuszami przy użyciu metody **isFinishing()**. W tej metodzie należy zwolnić wszystkie zasoby, które do tej pory nie zostały jeszcze zwolnione w pozostałych metodach jak **onStop()**.

Fragment to komponent, który jak sama nazwa wskazuje reprezentuje fragment interfejsu użytkownika w obrębie Activity lub implementuje pewną część jego zachowań. Z jednej strony wiele Fragment'ów może składać się na pojedyncze Activity, z drugiej strony pojedynczy Fragment

może być reużytkowany w wielu Activity. Fragment posiada swój własny cykl życia (Rys. 21) i może prowadzić niezależną interakcję z użytkownikiem. Jednocześnie nie może on istnieć w oderwaniu od Activity tj. musi być osadzony w którymś z nich. Cykl życia Activity wpływa na cykl życia zawartych w nim Fragment'ów. Kiedy Activity jest niszczone, podobnie dzieje się z jego Fragment'ami.

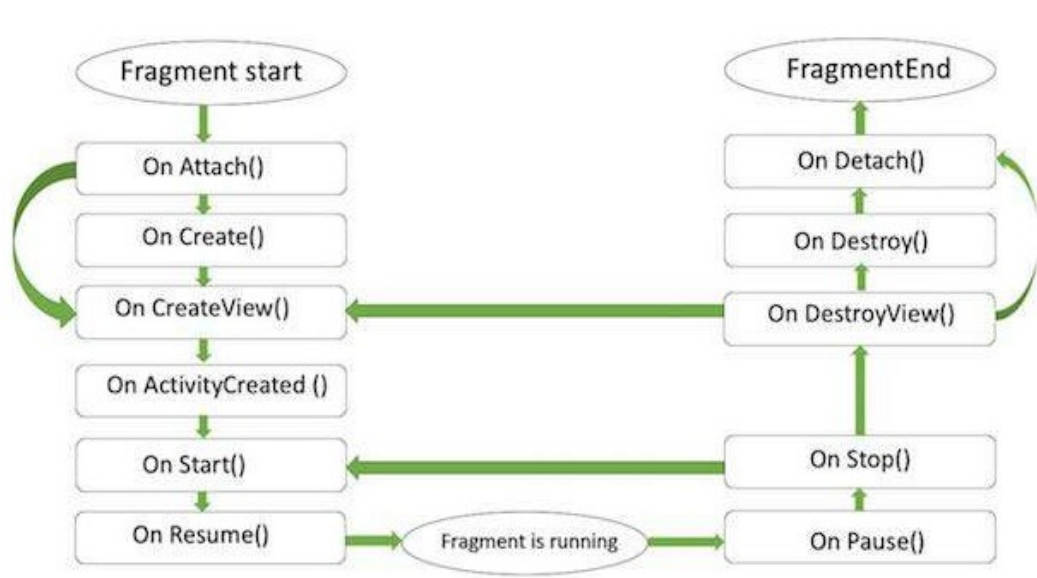
Głównym założeniem projektantów było umożliwienie podziału rozbudowanych i skomplikowanych Activity na mniejsze kawałki skupiające się na wykonywaniu konkretnego zdania. Używając Fragment'ów programista zyskuje (Rys. 20): modularność (lepszą organizację kodu), reużytkowalność (mniejsza ilość powielonego kodu), adaptowalność (dostosowywanie układu widoków na ekranie do jego rozmiarów i orientacji).

Fragment'y zostały dodane do Android SDK począwszy od wersji 3.0 Honeycomb. Google udostępnia również Fragment'y na wcześniejsze wersje Androida poprzez Support Library. Fragment może zostać dodany do Activity na dwa sposoby: statycznie, poprzez znacznik `<fragment>` w pliku XML layout'u oraz dynamicznie z użyciem transakcji **FragmentManager'a**. Zaleca się, aby komunikacja Fragment'u z Activity odbywała się poprzez interfejs listenera z funkcjami callback'owymi.



Rys. 20 Zalety stosowania Fragment'ów [15]

Cykl życia Fragment'u można podzielić na dwie części: sekwencja zdarzeń podczas dodawania Fragment'u do Activity oraz sekwencja zdarzeń podczas usuwania Fragment'u z Activity.



Rys. 21 Cykl życia Fragment'u [16]

Metoda **onAttach()** informuje o zdarzeniu dołączania Fragment'u do Activity. Metoda **onCreate()** jest miejscem inicjalizacji nowej instancji Fragment'u. W metodzie **onCreateView()** ma miejsce tworzenie widoku Fragment'u, który ma zostać osadzony w hierarchii widoków głównego Activity. Wraz z zakończeniem tworzenia Activity gospodarza we Fragment'cie wywoływana jest funkcja callbackowa **onActivityCreated()**. Począwszy od tego momentu kod Fragmentu może bezpiecznie odwoływać się do kodu Activity w którym został osadzony. Metoda **onStart()** jest wywoływana w momencie gdy widok Fragmentu pojawia się na ekranie. Metoda **onResume()** jest wywoływana po metodzie **onStart()** gdy Fragment staje się zdolny do interakcji z użytkownikiem.

Metoda **onPause()** jest wywoływana w momencie gdy Fragment przestaje być zdolny do interakcji z użytkownikiem. Kolejnym krokiem jest wywołanie funkcji **onStop()** gdy Fragment przestaje być widoczny na ekranie. W tym momencie następuje niszczenie widoku Fragment'u i związanych z nim zasobów, wywoływana jest metoda **onDestroyView()**. Na koniec fragment jest niszczone **onDestroy()** i odłączany od Activity gospodarza **onDetach()**.

4.3 Komponenty Service oraz IntentService

Service to komponent aplikacji przeznaczony do wykonywania czasochłonnych operacji w tle (ang. background), nie posiada interfejsu użytkownika. Interakcja z użytkownikiem nie jest wymagana, a działanie Service może trwać dłużej niż czas życia aplikacji. Inne komponenty aplikacji mogą komunikować się z Service'em albo poprzez interfejs IBinder'a, albo poprzez IPC (ang. interprocess communication). Typowe zastosowania Service'u to odtwarzanie muzyki w tle, wykonywanie transakcji sieciowych.

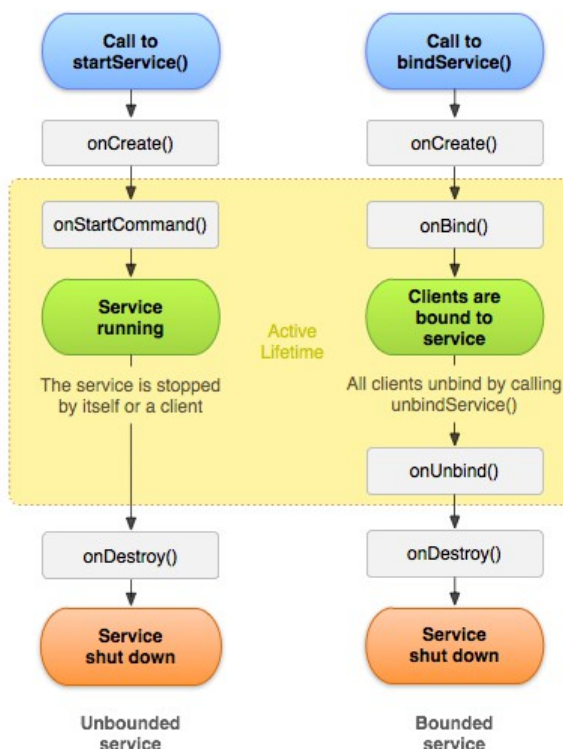
Android SDK wyróżnia trzy typy Service'ów: Foreground Started Service (dostrzegalny przez użytkownika poprzez notyfikację w pasku statusu, interakcja z aplikacją nie warunkuje działania tego typu serwisu), Background Started Service (niewidoczny dla użytkownika, wykonuje pewne operacje w tle), Bound Service (występuje interakcja typu klient-serwer przy użyciu interfejsu IBinder lub komunikacji międzyprocesowej IPC, serwis tego typu działa jedynie wtedy gdy istnieją powiązania z innymi komponentami). Istnieje możliwość utworzenia Service'u, który jest zarówno uruchamiany (ang. started) jak i dowiązany do innego komponentu (ang. bound). Service typu **Started** jest uruchamiany przy pomocy metody **startService()**. Taki serwis działa w nieskończoność nawet jeżeli komponent który go uruchomił został zwolniony. Service typu **Bound** jest uruchamiany w wyniku dowiązania go do innego komponentu przy pomocy funkcji **bindService()**. Każdy z typów serwisów wymaga zaimplementowania nieco innych metod podczas rozszerzania klasy Service. Cykl życia serwisu typu **Started** (Rys. 22, po lewej) wymaga zaimplementowania następujących metod:

- Metoda **onCreate()** jest wywoływana w momencie tworzenia serwisu po raz pierwszy, służy do jednorazowej inicjalizacji tego komponentu.
- Metoda **onStartCommand()** jest wywoływana tylko wtedy gdy serwis został uruchomiony przez inny komponent przy pomocy funkcji **startService()**.
- Zakończenie działania serwisu tego typu jest możliwe jedynie poprzez jawne wywołanie funkcji **stopSelf()** lub **stopService()**. W tym momencie następuje wywołanie callback'u **onDestroy()**. Pozwala on na zwolnienie niepotrzebnych zasobów i posprzątanie po serwisie.

Cykl życia serwisu typu **Bound** (Rys. 22, po lewej) wygląda następująco:

- Metoda **onCreate()** jest wywoływana w momencie tworzenia serwisu po raz pierwszy, służy do jednorazowej inicjalizacji tego komponentu.
- Metoda **onBind()** jest wywoływana gdy inny komponent aplikacji dokonał dowiązania do serwisu poprzez funkcję **bindService()**. Głównym zadaniem tej metody jest zwrócenie obiektu implementującego interfejs IBinder. Interfejs ten umożliwia interakcję komponentu klienta z komponentem serwisu.
- Metoda **onUnbind()** jest wywoływana w momencie odłączenia się od serwisu ostatniego klienta.
- Metoda **onRebind()** jest wywoływana w momencie ponownego dowiązania się klientów po wcześniejszym całkowitym ich odłączeniu.

- Serwis tego typu jest niszczony (wywołanie **onDestroy()**) w momencie odłączenia się od niego wszystkich klientów.

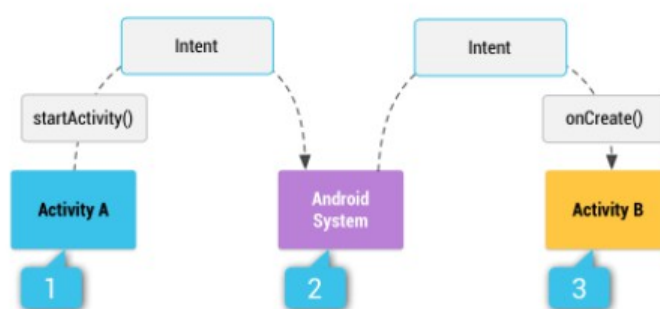


Rys. 22 Cykl życia komponentu Service [14]

Szczególnym typem serwisów jest tzw. Intent Service. Służy on do asynchronicznej obsługi zapytania klienta. Klient wysyła zapytanie przy pomocy wywołania **startService(Intent)**. Serwis obsługuje to zapytanie używając wątku (ang. worker thread). Po zakończeniu pracy kończy swoje działanie. Intent Service do poprawnego działania wymaga zaimplementowania metody **onHandleIntent(Intent)**. Wszystkie zapytania wysyłane to Intent Service'u są obsługiwane w ramach pojedynczego worker thread jedno po drugim.

4.4 Komponenty Intent i IntentFilter

Intent to specjalny komponent służący do wyrażenia pewnej intencji. Intencją jest wykonanie akcji przez któryś z komponentów dowolnej aplikacji w systemie. Typowymi przypadkami użycia Intent'ów są: uruchomienie nowego Activity poprzez wywołanie **startActivity(Intent)** lub **startActivityForResult(Intent)**, uruchomienie nowego Service'u poprzez wywołanie **startService(Intent)** lub dowiązanie do Service'u poprzez wywołanie **bindService(Intent)**, wysłanie wiadomości w trybie rozgłoszeniowym (ang. broadcast) przy pomocy **sendBroadcast(Intent)**.



Rys. 23 Uruchamianie Activity przy pomocy Intent'u [14]

W Android SDK wyróżnia się dwa typy intencji: jawne (ang. Explicit Intent) oraz niejawne (ang. Implicit Intent). Explicit Intent zawiera pełną kwalifikowaną nazwę klasy komponentu od którego żąda wykonania określonej akcji. Ten typ intencji jest używany głównie w celu uruchomienia komponentów z tej samej aplikacji. Implicit Intent nie definiuje nazwy konkretnego komponentu, podaje natomiast akcję, którą potrzebuje wykonać np. zrobienie zdjęcia, otwarcie strony www, etc. Na intencję tego typu może odpowiedzieć dowolna aplikacja będąca w stanie zrealizować zadaną akcję.

Wraz z Intent'em niezależnie od jego typu mogą zostać przesłane dodatkowe dane. Dane te przypisywane są do komponentu intencji przy pomocy metody **putExtra()**. Dowolny komponent aplikacji może pobrać Intent, który spowodował jego uruchomienie za pomocą metody **getIntent()**.

Komponenty aplikacji, które wykonują określone akcje mogą ogłaszać się w systemie poprzez zdefiniowanie filtrów intencji (ang. IntentFilter). W ten sposób komponent może odpowiadać na niejawną intencje (ang. Implicit Intent). Najczęściej Intent Filter jest definiowany poprzez znacznik **<intent-filter>** w pliku AndroidManifest.xml, aczkolwiek można go również zdefiniować programistycznie. Intent Filter wymaga podania nazwy akcji np. **android.intent.action.SEND**, oraz nazwy kategorii np. **android.intent.category.DEFAULT**. Dodatkowo można określić typ danych MIME.

4.5 Komponent Broadcast Receiver

W systemie Android istnieje możliwość wysyłania i odbierania wiadomości rozgłoszeniowych (ang. broadcast message) z informacją o wystąpieniu jakiegoś zdarzenia. Mechanizm ten realizuje wzorzec projektowy publish-subscribe. Broadcast Receiver jest komponentem, który nasłuchuje wiadomości określonego typu, jest więc subskrybentem. Nadawcą wiadomości może być system lub dowolna aplikacja. Przykładem zdarzeń systemowych, które mogą być nasłuchiwanie za pomocą Broadcast Receiver'a są np. rozpoczęcie ładowania baterii czy

włączenie trybu samolotowego. Komunikaty rozgłoszeniowe reprezentowane są przez Intent'y. Nazwa akcji Intent'u definiuje zdarzenie, którego dotyczy komunikat.

Android SDK wyróżnia dwa typy Broadcast Receiver'ów: deklarowane za pomocą znaczników **<receiver>** w pliku **AndroidManifest.xml** oraz rejestrowane za pomocą metody **registerReceiver()** dostępnej na obiekcie **Context**. Każdy Broadcast Receiver jest tworzony poprzez rozszerzenie klasy **BroadcastReceiver** i zaimplementowanie metody **onReceive()**. Do metody tej przekazywany jest Intent z informacją o rozgłaszanym zdarzeniu.

Oprócz subskrybowania komunikatów o zdarzeniach systemowych lub pochodzących z innych aplikacji, aplikacja może również publikować (rozgłaszać) własne komunikaty. W tym celu konieczne jest utworzenie customowego Intentu z informacją o zdarzeniu i rozgłoszenie go przy pomocy metody **sendBroadcast(Intent)** lub **sendOrderedBroadcast(Intent)**. Dodatkowo istnieje możliwość ograniczenia odbiorców rozgłoszenia tylko do lokalnej aplikacji. Wystarczy skorzystać z metody **sendBroadcast()** zdefiniowanej w klasie **LocalBroadcastManager**. Nasłuchiwanie niektórych rozgłoszeń systemowych wymaga uzyskania zezwolenia od użytkownika (ang. permission). Rozgłaszając własne komunikaty również można wymagać posiadania zezwoleń do ich odbioru.

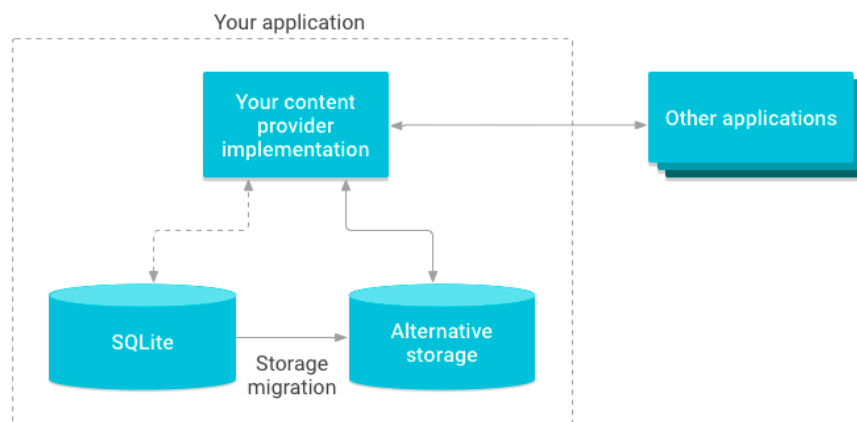
4.6 Komponent ContentProvider i Loader

Komponent Content Provider'a tworzy abstrakcyjną warstwę dostępu do danych. Pozwala na bezpieczne współdzielenie danych z innymi aplikacjami w systemie. Dostęp do danych udostępnianych przez Content Provider'a możliwy jest przy pomocy obiektu ContentResolvera i adresów URI. Content Provider jako element warstwy dostępu do danych może korzystać z różnych źródeł danych jak baza danych SQLite, sieć czy pliki.

Utworzenie Content Providera sprowadza się do rozszerzenia klasy ContentProvider i zaimplementowania jej metod. Adres URI powinien mieć format **content://<authority>/<entity>/<id>**. Lista metod implementowanych w ramach ContentProvider'a odpowiada typowym operacją CRUD. Metoda **query()** wykonuje kwerendę na źródle danych i zwraca rezultat w postaci obiektu **Cursor**. Metoda **insert()** wstawia nowy rekord do źródła danych. Metoda **delete()** usuwa istniejący rekord ze źródła danych. Metoda **update()** aktualizuje istniejący rekord w źródle danych. Metoda **getType()** zwraca typ MIME danych w źródle.

Aby skorzystać z danych udostępnianych przez Content Provider, można dokonać ich wczytania przy pomocy Loader API. Stosowanie Loader API niesie ze sobą wiele korzyści, tj. asynchroniczne wczytywanie danych stosując osobny wątek, stosowanie metod callback'owych do informowania o zdarzeniach wczytywania danych, zapisywanie wyników zapytań w pamięci

podręcznej pomiędzy zmianami konfiguracji urządzenia, zastosowanie ContentObserver'a w celu automatycznego przeładowywania danych w przypadku ich zmiany w źródle.



Rys. 24 Warstwa dostępu do danych
z wykorzystaniem komponentu Content Provider [14]

4.7 Tworzenie interfejsu użytkownika

Android pozwala na tworzenie graficznego interfejsu użytkownika zarówno w kodzie jak i deklaratywnie w postaci plików XML w katalogu `res/layout`. W Android Studio dzięki zastosowaniu nowego `ConstraintLayout`'u istnieje możliwość graficznego układania elementów (ang. widget) na ekranie. Interfejs użytkownika budowany jest z widoków (ang. View) oraz grup widoków (ang. View Group). Występują następujące grupy widoków : `LinearLayout`, `RelativeLayout`, `ListView`, `GridView`, `FrameLayout` , `ConstraintLayout`, etc. Każdemu z elementów widoku można przypisać unikalny identyfikator pozwalający odwoływać się do tego elementu w kodzie aplikacji przy pomocy `findViewById()` na obiekcie `Context`'u.

4.8 Android NDK i JNI

Tradycyjnie podstawowym językiem programowania dla platformy Android była Java. Obecnie Google coraz intensywniej zachęca programistów do korzystania z języka Kotlin. Często jednak istnieje konieczność skorzystania z dostępnego kodu natywnego, lub zaimplementowania kluczowych funkcjonalności aplikacji czy to ze względu na wydajność czy też przenośność w języku C/C++. Android oferuje taką opcję poprzez zbiór narzędzi i bibliotek skupionych w Android NDK. Dodatkowo w Javie interoperacyjność z językami natywnymi jest możliwa poprzez JNI (ang. Java Native Interface). Native Development Kit oprócz tego, że pozwala na współpracę z językami C/C++ dostarcza również bibliotek natywnych dla natywnych aktywności (ang. native Activity) czy

komponentów urządzeń takich jak czujniki (ang. sensor). Historycznie Android do budowania aplikacji z kodem natywnym korzystał z narzędzia ndk-build. Obecnie Android Studio wspiera kompilację natywnych bibliotek z wykorzystaniem narzędzia CMake.

W komponentach pisanych w Javie, w celu wskazania metod zaimplementowanych w kodzie natywnym należy użyć słowa kluczowego **native**. Biblioteki kodu natywnego mogą być podzielone na statyczne (.a) oraz współdzielone (.so). Interoperacyjność kodu napisanego w Javie z kodem napisanym w C/C++ jest możliwa dzięki Java Native Interface (JNI). Kod natywny umieszczany jest w katalogu /src/cpp w oddzielnych plikach z rozszerzeniem .c lub .cpp. W momencie wywoływania przez JVM (ang. Java Virtual Machine) funkcji natywnej do funkcji tej przekazywane są następujące parametry: wskaźnik na JNIEnv, wskaźnik na jobject oraz dowolne dodatkowe argumenty zdefiniowane w metodzie natywnej w Javie (Rys. 25).

```
extern "C"
JNIEXPORT void JNICALL Java_ClassName_MethodName
(JNIEnv *env, jobject obj, jstring javaString)
{
    const char *nativeString = env->GetStringUTFChars(javaString, 0);

    //Do something with the nativeString

    env->ReleaseStringUTFChars(javaString, nativeString);
}
```

Rys. 25 Metoda natywna zgodna z JNI [17]

Każdy z typów natywnych ma swój odpowiednik w JNI. Przykładowo **const char *** odpowiada **jstring**, a **int** odpowiada typ **jint**. Dostęp do klasy Javy możliwy jest poprzez wywołanie metody **(*env)->GetObjectClass()**, która zwraca wartość typu **jclass**. Dostęp do poszczególnych metod w klasie możliwy jest przy pomocy funkcji **(*env)->GetMethodId()**. Wymaga ona podania nazwy funkcji i sygnatury jej parametrów, np. **()Ljava/lang/String;**. Zwraca wartość typu **jmethodID**, którą można wykorzystać do wywołania metody z klasy Java przy pomocy następującego wywołania w języku C **(*env)->CallObjectMethod()**. Konwersję pomiędzy typem **jstring**, a typem natywnym **const char *** umożliwia następująca procedura **(*env)->GetStringUTFChars()** (Rys. 25).

5. IMPLEMENTACJA BIBLIOTEKI ZDALNEGO STEROWANIA SMARTFONEM KOMPUTERA

Biblioteka zdalnego sterowania komputera smartfonem została napisana w języku C. Ma to za zadanie ułatwić jej użycie w różnych systemach operacyjnych. W większości systemów i języków programowanie istnieje bowiem wsparcie dla interoperacyjności z kodem natywnym w języku C/C++. Przykładowo aplikacje Cocoa na system macOS czy iOS mogą być pisane w języku Objective-C lub Swift. Język Objective-C stanowi nadzbiór (ang. superset) języka C. Również platforma Android zapewnia kompatybilność z bibliotekami napisanymi w języku C poprzez Android NDK oraz Java Native Interface. Podobnie sytuacja wygląda w innych systemach operacyjnych jak Windows czy Linux.

Oczywiście nie wszystkie wywołania systemowe stosowane na jednej z platform są bezpośrednio przenośne na inne platformy. Przykładowo pojawia się pewien problem ze stosowaniem unixowych wywołań systemowych i wątków POSIX w systemie Windows. Nawet wtedy istnieje jednak możliwość zastosowania narzędzi typu Cygwin lub MinGW, aby umożliwić uruchomienie tak napisanego kodu w systemie Windows.

5.1 Struktura biblioteki RemoteControllerAPI

Bibliotekę umożliwiającą zdalne sterowanie komputera przy pomocy smartfonu nazwano RemoteControllerAPI. Konceptyjnie bibliotekę tą można podzielić na dwie części: RemoteControllerAPI_Serwer oraz RemoteController API_Client. Pozwala to na realizację aplikacji sieciowych w modelu klient-serwer z wykorzystaniem tych bibliotek.

RemoteControllerAPI_Server jest przeznaczone dla aplikacji pisanych na komputery stacjonarne z systemem macOS, Windows czy Linux. Pozwala na zdalne kontrolowanie tych urządzeń i symulowanie w nich działania myszy czy klawiatury poprzez odpowiednie wywołania systemowe. Z drugiej strony RemoteControllerAPI_Client jest przeznaczony dla aplikacji pisanych na urządzenia mobilne z systemami Android czy iOS. Daje ono możliwość sterowania urządzeniem zdalnym z poziomu aplikacji mobilnych.

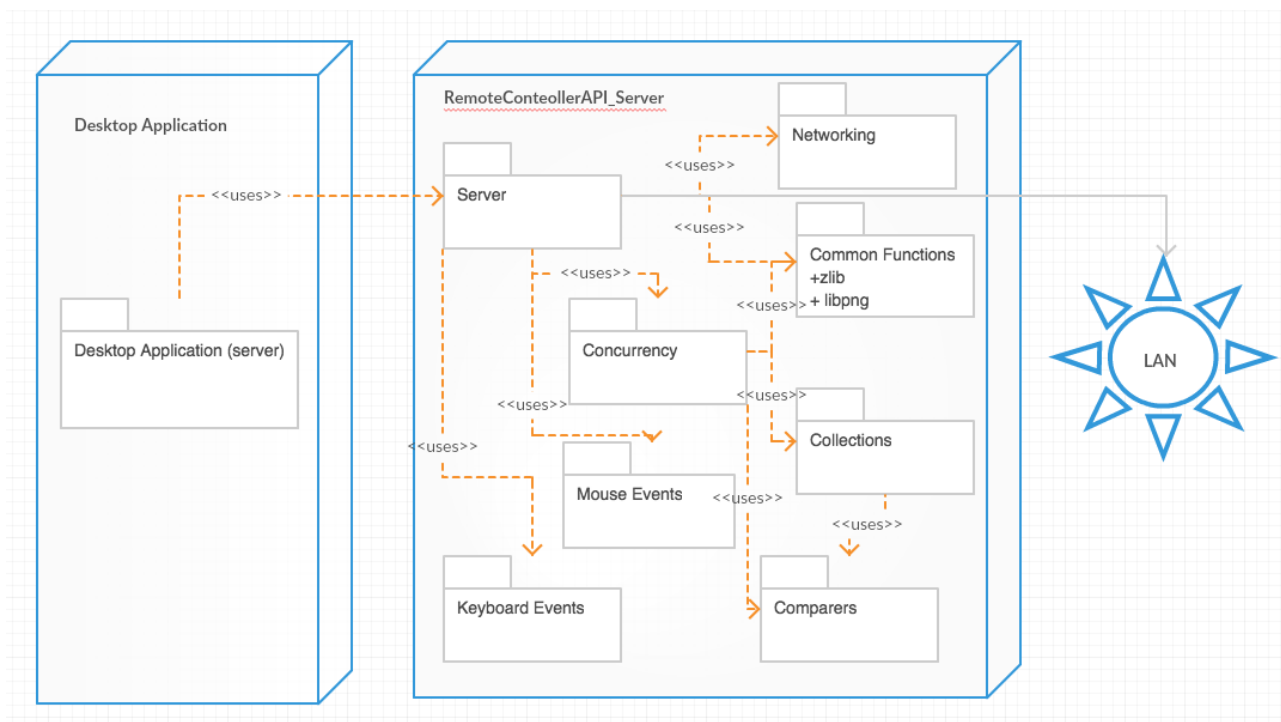
Cała biblioteka RemoteControllerAPI składa się z wielu mniejszych bibliotek (Rys. 31), które mogą być niezależnie reużytkowane. Z wielu tych mniejszych bibliotek korzysta zarówno RemoteControllerAPI_Server jak i RemoteControllerAPI_Client. Poszczególne biblioteki stanowią osobny projekt CMake, z oddzielnym plikiem CMakeLists.txt. Mogą być one załączane w ramach większych projektów CMake jako podprojekty. Służy do tego komenda **add_subdirectory()** w plikach konfiguracyjnych CMake. Dla każdego z podprojektów zdefiniowany jest jednocześnie plik CMakeLists.txt jak i plik make.txt umożliwiający kompilację i linkowanie bibliotek zarówno przy

użyciu CMake jak i Makefile.

Pliki konfiguracyjne pozwalają na kompilacje bibliotek statycznych (z rozszerzeniem .a) oraz bibliotek dynamicznych (z rozszerzeniem .so, .dylib lub .dll). Podstawowa różnica pomiędzy bibliotekami statycznymi i dynamicznymi dotyczy momentu w którym taka biblioteka jest łączona z programem. Biblioteki statyczne są dołączane do programu w momencie konsolidacji, natomiast biblioteki dynamiczne dopiero w momencie wykonywania programu. Biblioteka statyczna zwiększa rozmiar pliku wykonawczego, z drugiej strony biblioteki dynamiczne muszą zostać zainstalowane w systemie razem z plikiem wykonawczym, aby zapewnić poprawne działanie aplikacji. Do niewątpliwych zalet bibliotek dynamicznych należy możliwość ich współdzielenia przez większą liczbę programów.

Moduł **RemoteControllerAPI_Server** obejmuje następujące biblioteki języka C:

- **Server** – zawiera implementacje różnych typów serwerów (iteracyjne, współbieżne, połączeniowe, bezpołączeniowe), które mogą być uruchamiane przez zewnętrzne aplikacje GUI. Dodatkowo zawiera implementacje protokołu komunikacyjnego warstwy aplikacji umożliwiającego udostępnianie usług zdalnego sterowania komputerem: obsługa zdarzeń myszy, klawiatury i innych.
- **Networking** – biblioteka zawierająca kod wspólny wykorzystywany zarówno przez bibliotekę klienta, jak i bibliotekę serwera.
- **Concurrency** – biblioteka zawiera kod implementujący wzorzec projektowy puli wątków (ang. thread pool), który umożliwia tworzenie współbieżnych serwerów.
- **Common Functions** – to biblioteka zawierająca zestaw funkcji pomocniczych wykorzystywanych przez inne biblioteki z modułu, a których nie udało się przyporządkować do żadnego innego podprojektu. Część z funkcji tj. przetwarzanie bitmap i operacje na bitach stanowią podwaliny pod możliwość zaimplementowania streamingu ekranu komputera przez gniazda sieciowe.
- **Collections** – to biblioteka struktur danych takich jak lista pojedynczo wiązana (ang. linked list), lista podwójnie wiązana (ang. double linked list), tablica haszująca (ang. hash map) wykorzystywanych przez inne biblioteki z modułu.
- **Comparers** – to biblioteka implementująca funkcje pozwalające porównywać różne typy danych, wykorzystywana w testach jednostkowych i kolekcjach.



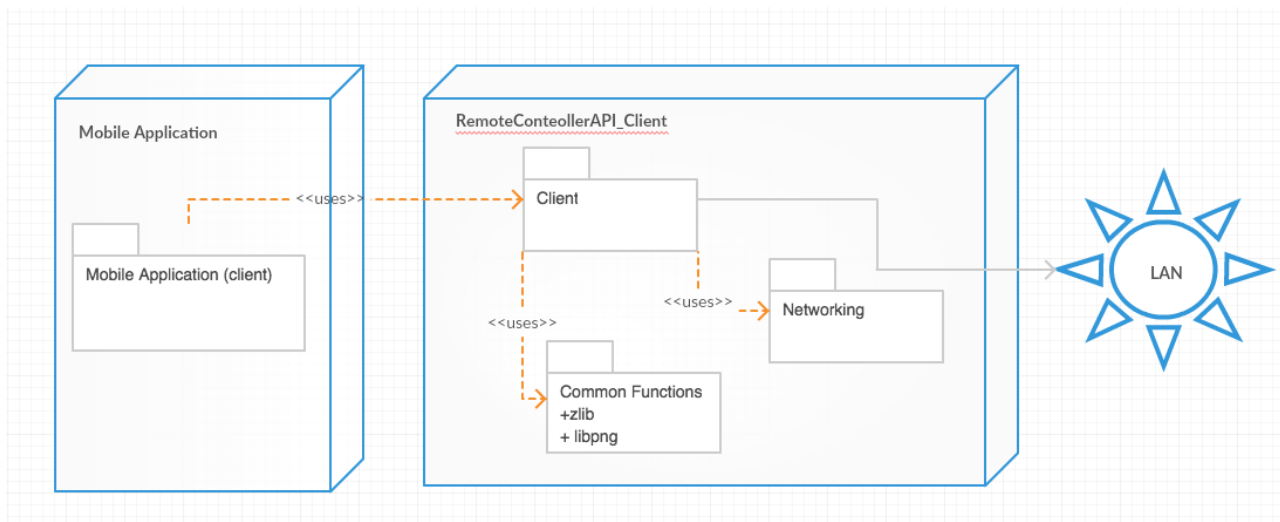
Rys. 26 Diagram pakietów biblioteki Serwera.

- **Keyboard Events** – to grupa bibliotek zapewniających jednolity interfejs symulowania zdarzeń klawiatury. Aktualnie dostępne są wersje biblioteki dla systemu macOS i Windows.
- **Mouse Events** – to grupa bibliotek zapewniających jednolity interfejs symulowania zdarzeń myszy. Aktualnie dostępne są wersje biblioteki dla systemu macOS i Windows.

Po drugiej stronie komunikacji sieciowej występuje moduł **RemoteController API_Client** obejmujący następujące biblioteki języka C:

- **Client** – zawiera implementacje programu klienta, który może być uruchamiany przez aplikacje GUI w celu komunikacji ze zdalnym serwerem. Dodatkowo zawiera implementację protokołu komunikacyjnego warstwy aplikacji umożliwiającego dostęp do usług zdalnego serwera.
- **Networking** – biblioteka zawierająca kod wspólny wykorzystywany zarówno przez bibliotekę klienta, jak i bibliotekę serwera.
- **Common Functions** – to biblioteka zawierająca zestaw funkcji pomocniczych wykorzystywanych przez różne biblioteki z modułu, a których nie udało się

przyporządkować do żadnego innego podprojektu. Część z funkcji tj. przetwarzanie bitmap i operacje na bitach stanowią podwaliny pod możliwość zaimplementowania streamingu ekranu komputera przez gniazda sieciowe.



Rys. 27 Diagram pakietów biblioteki Klienta.

5.2 Implementacja Serwera

Z punktu widzenia użytkownika biblioteki uruchomienie serwera sprowadza się do prostego wywołania funkcji **start_server(server_t, server_info_t)**. Funkcja ta pobiera procedurę z implementacją konkretnego typu serwera oraz strukturę ze szczegółowymi informacjami odnośnie parametrów serwera (Rys. 28).



Rys. 28 Struktura server_info_t

Struktura **server_info_t** posiada zestaw procedur, które pozwalają na jej zainicjalizowanie, wypełnienie danymi, dostęp do danych oraz usunięcie z pamięci. Poprzez tą strukturę aplikacja użytkownika zarządza adresem ip, numerem portu, gniazdem sieciowym czy hasłem bezpieczeństwa serwera. Struktura przechowuje również listę podłączonych klientów. Wypełniając odpowiednie pola struktury aplikacja korzystająca z biblioteki serwera ma możliwość obsługi zdarzeń serwera takich jak: uruchomienie serwera, zakończenie pracy serwera, wystąpienie błędów serwera, połączenie i odłączenie klienta, błędy połączenia czy błędy zapytania datagramowego.

Procedurę implementującą serwer można wybrać spośród dwóch grup: procedur dla serwerów połączeniowych i procedur dla serwerów bezpołączeniowych. Dodatkowo każda z tych procedur może w swojej implementacji posługiwać się innym typem serwera, np. iteracyjny, współbieżny oparty na puli wątków, pseudowspółbieżny.

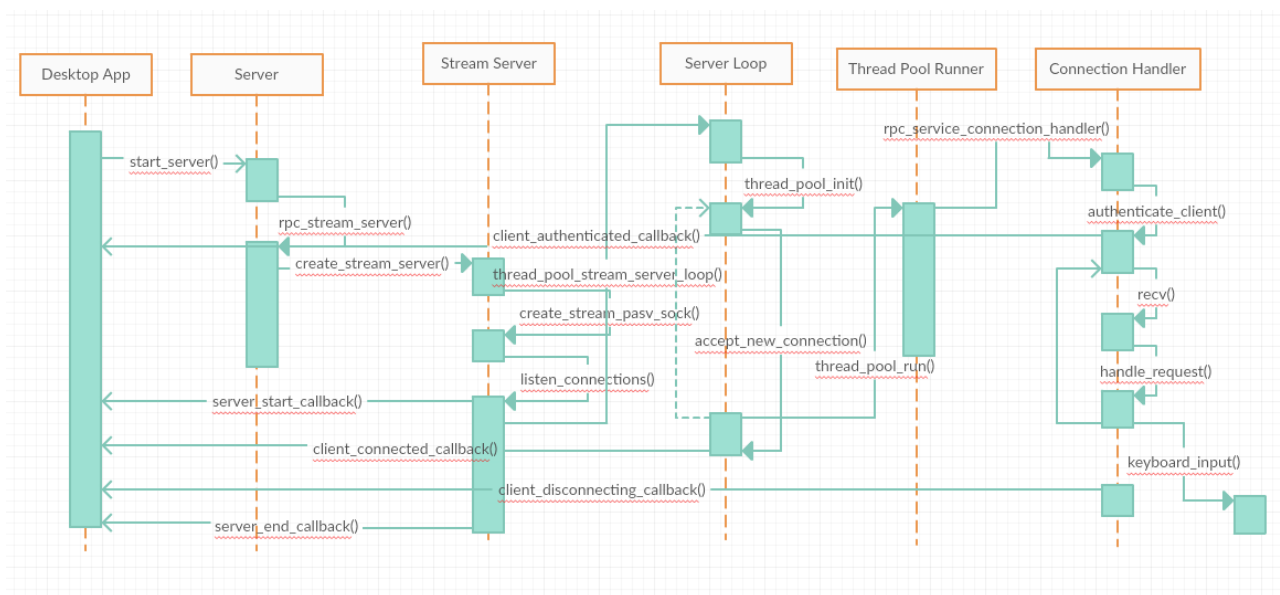
Serwer realizujący usługi wywoływania zdalnych procedur umożliwiających sterowanie komputerem na którym działa został nazwany **rpc_stream_server** i zaimplementowany jako połączeniowy, współbieżny z wykorzystaniem puli wątków.

Utworzenie serwera połączeniowego TCP odbywa się poprzez wywołanie funkcji **create_stream_server()** oraz podanie trzech argumentów: struktury **server_info_t**, procedury **server_loop_t** oraz procedury **connection_handler_t**.

Procedura **server_loop_t** jest jedną z predefiniowanych funkcji implementujących konkretny typ serwera. Biblioteka udostępnia następujące typy serwerów połączeniowych: iteracyjny, współbieżny, współbieżny z zarządzaną liczbą wątków, współbieżny oparty o pulę wątków, pseudowspółbieżny. Analogicznie biblioteka definiuje funkcje dla różnego typu serwerów bezpołączeniowych. Serwer stosowany domyślnie przez bibliotekę to **thread_pool_stream_server_loop()**.

Procedura **connection_handler_t** jest natomiast funkcją implementującą protokół obsługi połączenia z klientem. W przypadku obsługi zdalnego sterowania komputera przez klientów stosowana jest funkcja **rpc_service_connection_handler()**.

Tworzenie serwera TCP rozpoczyna się od utworzenia biernego gniazda sieciowego **create_stream_pasv_sock()**. Następnie serwer ustawiany jest w tryb nasłuchiwania połączeń klientów **listen_connections()**. W tym momencie następuje powiadomienie aplikacji użytkownika o pomyślnym utworzeniu serwera poprzez wywołanie funkcji callbackowej **server_start_callback_t**.



Rys. 29 Sekwencja wywołań serwera obsługi zdalnego sterowania

W ramach serwera wywoływana jest procedura pętli serwera **thread_pool_stream_server_loop()**. Pętla inicjalizuje nową pulę wątków o zadanym rozmiarze i rozpoczyna akceptowanie przychodzących połączeń klienta **accept_new_connection()**. Aplikacja użytkownika jest informowana o zdarzeniu nowego połączenia przy pomocy funkcji **client_connected_callback()**. Obsługa poszczególnych połączeń przekazywana jest do któregoś z wolnych wątków puli **thread_pool_run()**.

Wątek wywołuje funkcję obsługi połączenia **rpc_service_connection_handler()** realizującą protokół zdalnego sterowania komputera. Po uwierzytelnieniu klienta następuje obsługa jego zapytań **handle_request()**. Przetwarzanie zapytań skutkuje wywoływaniem odpowiednich akcji sterujących systemem operacyjnym komputera np. **keyboard_input()**, **mouse_move_by()**, itd. Zakończenie połączenia z klientem wiąże się z wywołaniem funkcji **client_disconnecting_callback()**, natomiast zakończenie pracy serwera z wywołaniem funkcji **server_end_callback()**.

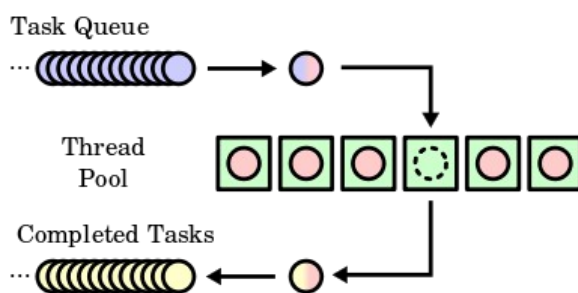
5.3 Pula wątków i kolejka zadań

Liczba wątków, które mogą jednocześnie istnieć w systemie operacyjnym jest ograniczona. Oznacza to, że nie można tworzyć nowych wątków w nieskończoność. Dodatkowo koszty tworzenia i usuwania wątków nie są bez znaczenia. Odpowiedzią na te problemy jest wzorec projektowy puli wątków (ang. thread pool). Pula wątków pozwala na utworzenie z góry zdefiniowanej liczby wątków (ang. worker threads), które będą oczekiwały w ciągłej gotowości na wykonanie powierzanych im zadań. Z koncepcją puli wątków nierozdzielnie wiąże się struktura

kolejki zadań (ang. task queue).

Mechanizm działania puli wątków w sposób ogólny opisuje Rys. 30. Każdy wątek spośród skończonego zbioru N wątków, gdy jest bezczynny pobiera kolejne zadanie z kolejki zadań. Operacja zdejmowania nowego zadania jest blokująca, tzn. wątek będzie czekał na kolejce zadań tak długo aż pojawi się nowe zadanie do wykonania. Synchronizacja operacji wstawiania i zdejmowania zadań z kolejki może odbywać się z wykorzystaniem zmiennych warunkowych (ang. conditional variable).

Pula wątków oraz kolejka zadań zostały zaimplementowane w bibliotece Concurrency. Pula wątków jest reprezentowana przez strukturę `thread_pool_t`. Natomiast kolejka zadań przez strukturę `task_queue_t`. Kolejka zadań bazuje na tradycyjnej kolejce FIFO (struktura `fifo_queue_t`). Różni się ona od kolejki FIFO dodaniem synchronizacji operacji wstawiania (ang. enqueue) i zdejmowania (ang. dequeue) zadań.



Rys. 30 Wzorzec projektowy Puli Wątków (ang. Thread Pool) [18]

Kolejka FIFO (ang. First In, First Out) reprezentowana jest przez strukturę `fifo_queue_t` zawierającą pojedyncze pole typu `doubly_linked_list_t`. Pole to jest listą dwukierunkową, której zadaniem jest przechowywanie elementów kolejki FIFO. Dla kolejki FIFO zdefiniowane są cztery operacje: inicjalizacji kolejki, wstawiania elementu do kolejki, zdejmowanie elementu z kolejki, zwalniania kolejki. Operacje wstawiania `fifo_enqueue()` umieszcza element na początku listy `dl_list_push_front()`. Operacja zdejmowania `fifo_dequeue()` pobiera element z końca listy `dl_list_back()` jednocześnie go usuwając `dl_list_pop_back()`.

Kolejka zadań `task_queue_t` opakowuje zwykłą kolejkę FIFO. Dodatkowo definiuje pola z liczbą zadań w kolejce `task_count` oraz obiekty służące synchronizacji operacji wstawiania i zdejmowania tj. `pthread_mutex_t` oraz `pthread_cond_t`. Każde zadanie umieszczane w kolejce zadań jest reprezentowane przez strukturę `task_t`. Zawiera ona pola do przechowywania wskaźnika na funkcję wykonującą zadanie (`runner_t`) oraz dodatkowe jej atrybuty. Podstawowe operacje kolejki zadań to wstawianie zadania do kolejki i zdejmowanie zadania z kolejki. Operacja

wstawiania **enqueue_task()** jest zaimplementowana jako producent z wzorca projektowego producent-konsument. Używa muteksu (ang. mutex) w celu uzyskania wyłącznego dostępu do kolejki. Sygnalizuje przy pomocy zmiennej warunkowej wątki czekające na dodanie nowego zadania do kolejki. Operacja zdejmowania **dequeue_task()** jest zaimplementowana jako konsument z wzorca projektowego producent-konsument. Również używa muteksu w celu uzyskania wyłącznego dostępu do kolejki. Jeżeli kolejka zadań jest pusta metoda ta blokuje wątek na zmiennej warunkowej aż do momentu pojawienia się nowego zadania. Po zasygnalizowaniu pojawienia się nowego zadania w kolejce metoda jest odblokowywana i wykonuje zdjęcie zadania z kolejki. Działanie kończy zwolnieniem muteksu.

Pula wątków **thread_pool_t** jest strukturą, która zawiera: współdzieloną kolejkę zadań do wykonania, tablicę wątków, aktualną liczbę wątków w puli, liczbę wątków wykonujących zadania, minimalny i maksymalny limit wątków, maksymalny czas oczekiwania wątku na pojawienie się nowego zadania, listę usuniętych wątków, muteks do synchronizacji dostępu. Podstawowymi operacjami puli wątków są **thread_pool_execute()** pozwalająca wykonać zadanie **task_t** oraz **thread_pool_run()** pozwalająca przekazać do wykonania dowolną funkcję **runner_t**. Obie te operacje sprowadzają się do umieszczenia nowego zadania w kolejce zadań tj. **enqueue_task()**. Algorytm wykonywany przez poszczególne wątki z puli (ang. worker threads) wygląda następująco:

1. w pętli nieskończonej czekaj na nowe zadanie do wykonania,
2. wykonaj metodę **dequeue_task_timed()** lub **dequeue_task()**,
3. sprawdź dostępność zadania, jeżeli czas oczekiwania jest zbyt długi zabij wątek
4. wykonaj zadanie
5. zwolnij zadanie i przejdź do pkt. 1

5.4 Implementacja klienta

Biblioteka **RemoteControllerAPI_Client** jest przeznaczona dla aplikacji które chcą umożliwić użytkownikowi zdalne sterowanie komputerem. Budowa programu klienta jest zdecydowanie prostsza niż budowa programu serwera. Dostępne są jedynie dwa typy klientów: połączeniowy i bezpołączeniowy. Program klienta może mieć injektowaną funkcję z dowolnym protokołem komunikacyjnym pod warunkiem, że jest on zgodny z protokołem komunikacyjnym serwera.

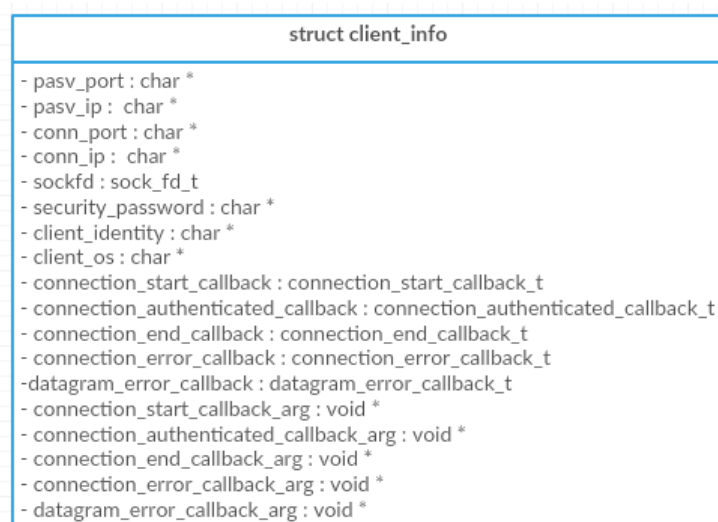
Aplikacja użytkownika może uruchomić program klienta za pomocą wywołania **start_client()**. Zakończenie działania klienta odbywa się natomiast poprzez wywołanie **end_client()**. Funkcja **start_client()** pobiera jako argumenty procedurę z implementacją kodu

klienta oraz strukturę **client_info_t** (Rys. 31).

Protokołu zdalnego sterowania komputerem zaimplementowany jest przy pomocy klienta połączeniowego **rpc_stream_client()**. Działanie tej procedury sprowadza się do utworzenia nowego połączenia TCP z serwerem **create_stream_conn()** i przekazania funkcji obsługi usługi zdalnego sterowania **rpc_service_handler()**.

Tworzenie połączenia TCP sprowadza się do utworzenia nowego gniazda sieciowego **create_conn_sock()** na podstawie danych adresowych zawartych w strukturze **client_info_t**. W tym momencie aplikacja użytkownika jest informowana o zdarzeniu nawiązania połączenia lub błędzie połączenia przez odpowiednie funkcje callback'owe.

Po poprawnym nawiązaniu połączenia z serwerem wywoływana jest funkcja obsługi połączenia **connection_handler_t**. W tym przypadku jest to funkcja obsługi protokołu zdalnego sterowania komputerem. Pierwszą czynnością wykonywaną w ramach protokołu jest uwierzytelnienie klienta na serwerze **authenticate_client_on_server()**. Po poprawnym uwierzytelnieniu klient przechodzi w stan nasłuchiwanie zdarzeń serwera **event_service_handler()**.



Rys. 31 Struktura **client_info_t**

Od teraz gniazdo klienta, może być wykorzystywane przez aplikację użytkownika do wykonywania asynchronicznych zapytań o zdalne wywołanie procedury sterującej komputerem na którym działa program serwera.

5.5 Biblioteka zdarzeń klawiatury i myszy dla systemu macOS

W systemie operacyjnym macOS symulowanie zdarzeń klawiatury i myszy jest możliwe przy pomocy biblioteki Quartz Event Services wchodzącej w skład framework'u Core Graphics. Biblioteka ta dostarcza narzędzi umożliwiających zarządzanie tzw. kranami zdarzeń (ang. event taps). Są to filtry pozwalające obserwować i zmieniać przepływ niskopoziomowych zdarzeń wejścia z urządzeń użytkownika (ang. user input events). Quartz Event Services zostały dodane w wersji 10.4 systemu OS X.

W celu zasymulowania zdarzenia klawiatury konieczne jest utworzenie źródła zdarzeń Quartz `CGEventSourceRef` `CGEventSourceCreate(CGEventSourceStateID stateID)`. Tworząc źródło zdarzeń należy zdefiniować odpowiedni stan źródła. Możliwymi wartościami są `kCGEventSourceStateCombinedSessionState` oraz `kCGEventSourceStateHIDSystemState`. Pierwsza flaga definiuje stan odnoszący się do wszystkich możliwych źródeł zdarzeń wysyłających zdarzenia do aktualnej sesji logowania użytkownika. Druga flaga definiuje natomiast stan odnoszący się do wszystkich hardware'owych źródeł zdarzeń pochodzących z systemu HID. W przypadku programów zdalnego sterowania możliwe jest używanie stanu źródła zdarzeń niezależnego od innych procesów. W takich przypadkach zalecane jest użycie `kCGEventSourceStatePrivate`.

Kolejnym krokiem jest utworzenie nowego zdarzenia klawiatury (ang. keyboard event) przy pomocy wywołania `CGEventCreateKeyboardEvent()`. Funkcja pobiera takie parametry jak utworzone wcześniej źródło zdarzeń, kod klawisza `CGKeyCode`, oraz flagę `keyDown` wskazującą na wciskanie/ puszczenie klawisza. Tworząc zdarzenia klawiatury warto wyczyścić ich maskę flag modyfikujących z maski Shift, a dopiero potem definiować nowe flagi. Do tego celu wykorzystywane są funkcje `CGEventGetFlags()` oraz `CGEventSetFlags()`.

Zdarzenia klawiatury są następnie wysyłane do strumienia zdarzeń określonego kranu zdarzeń `CGEventTapLocation` za pomocą wywołania `CGEventPost()`. Ostatnim krokiem jest zwolnienie utworzonego wcześniej zdarzenia klawiatury oraz źródła zdarzeń Quartz korzystając z `CFRelease()`. Maska flag modyfikujących jest liczbą całkowitą bez znaku `uint64_t`. Pozwala dołączyć do zdarzenia klawiatury klawisze modyfikujące jak Shift, Control, Command, etc. Kod klawisza jest natomiast liczbą całkowitą bez znaku `uint16_t`. Lista stałych dla wszystkich możliwych klawiszy została zdefiniowana w pliku nagłówkowym `virtual_key_code.h` biblioteki. Kody klawiszy identyfikują fizyczne klawisze na klawiaturze. Stałe klawiszy oznaczone ANSI odpowiadają pozycjom klawiszy na standardowej klawiaturze ANSI. Przykładowo `kVK_ANSI_A` wskazuje na kod klawisza dla litery A na klawiaturze z układem amerykańskim. Klawiatury z

innym layoutem mogą mieć umiejscowiony ten klawisz w innym fizycznym miejscu.

Biblioteka zdarzeń klawiatury (keyboard_events) dodatkowo implementuje symulowanie typowych skrótów klawiaturowych (ang. hot keys). Listy najpopularniejszych skrótów klawiaturowych definiowane są w zewnętrznych plikach XML z rozszerzeniem .plist (property list). Biblioteki dla systemu macOS posiadają narzędzia ułatwiające parsowanie takich plików.

Oddzielna biblioteka mouse_events pozwala z kolei symulować zdarzenia myszy. W tym celu konieczne jest utworzenie nowego zdarzenia myszy (ang. Quartz mouse event) przy pomocy funkcji **CGEventCreateMouseEvent()**. Tym razem źródło zdarzenia może zostać pominięte poprzez podanie wartości NULL. Natomiast wymagane jest określenie typu zdarzenia myszy CGEventType. Przykładowe wartości to: kCGEventLeftMouseDown, kCGEventMouseMoved czy kCGEventScrollWheel. Dodatkowo funkcja pobiera jako swoje parametry pozycję kursora myszy CGPoint(), oraz opcjonalną wartość modyfikującego przycisku myszy.

Tak utworzone zdarzenie myszy jest następnie wysyłane do odpowiedniego kranu zdarzeń CGEventTapLocation przy pomocy funkcji **CGEventPost()**. Typowo dla zdarzeń myszy stosowany jest kran kCGHIDEventTap. W przypadku zdarzeń kliknięcia myszy konieczne jest utworzenie dwóch zdarzeń, jednego dla zdarzenia wciśnięcia przycisku (kCGEventLeftMouseDown) oraz drugiego dla zdarzenia puszczenia przycisku (kCGEventLeftMouseUp). Dodatkowo w przypadku kliknięć wielokrotnych np. podwójne kliknięcie, przed wysłaniem zdarzeń konieczne jest ustawienia pola **kCGMouseEventClickState** wartością odpowiadającą liczbie kliknięć np. 2. Każde utworzone zdarzenie myszy podobnie jak było to w przypadku zdarzeń klawiatury po wysłaniu powinno zostać zdealokowane przy pomocy funkcji CFRelease(). W analogiczny sposób można tworzyć i wysyłać zdarzenia scrollowania. W bibliotece Quartz Event Services dostępna jest w tym celu specjalna funkcja **CGEventCreateScrollWheelEvent()**. Wymaga podania jednostki miary oraz wartości przesunięcia.

```
Michals-MacBook-Air:~ michzio$ osascript -e 'tell application "System Events"
> key code 40
> end tell '
Michals-MacBook-Air:~ michzio$ k
```

Rys. 32 Wywołanie AppleScript z wiersza poleceń

Symulowanie zdarzeń klawiatury jest również możliwe z wykorzystaniem języka skryptowego AppleScript do automatyzacji systemu macOS. AppleScript może być wywoływany zarówno z linii komend (Rys. 32) jak i z języka Objective-C czy Swift. Ponadto AppleScript pozwala sterować wieloma aplikacjami jak odtwarzacz wideo VLC:

- **tell application „VLC” to activate**
- **tell application „VLC” to volumeUp**
- **tell application „VLC” to fullscreen,**

przeglądarki internetowe, umożliwia zamykanie systemu operacyjnego, etc.

Skrypty AppleScript napisane w ramach projektu biblioteki zdalnego sterowania zostały skatalogowane w repozytorium <https://github.com/michzio/Apple-Script-System-Automation>.

```
tell application "VLC"
activate
try
  set currVideo to name of window 1
  display dialog currVideo
on error
  display dialog "nothing"
end try
end tell
```

Rys. 33 Pobranie nazwy odtwarzanego filmu w programie VLC

5.6 Inne zdarzenia systemu macOS

Zdarzenia systemu niezwiązane z symulowaniem zdarzeń klawiatury i myszy czy skrótami klawiszowymi zostały zaimplementowane w bibliotece `automation_scripts`. Przykładami takich zdarzeń są np. restartowanie systemu (`kAERestart`), zamykanie systemu (`kAEShutDown`), przechodzenie w stan uśpienia (`kAESleep`), czy wylogowywanie użytkownika (`kAEReallyLogOut`). Zdarzenia tego typu są tworzone przy pomocy funkcji `AECreatAppleEvent()` z frameworku Core Services, a następnie wysyłane za pomocą funkcji `AESend()`.

W ramach tego projektu zaimplementowano również zarządzanie wejściem/wyjściem dźwięku, czy jasnością ekranu. Ustawianie głośności wyjścia audio jest możliwe przy pomocy funkcji `AudioHardwareService SetPropertyData()`. Natomiast zmiana jasności ekranu poprzez wywołanie `IODisplaySetFloatParameter()`. Biblioteka `automation_scripts` zawiera również funkcjonalności wykonywania zrzutów ekranu (ang. screenshot) czy pobieranie nazwy programu dla okna pierwszoplanowego (ang. foreground window). Ta ostatnia funkcjonalność jest szczególnie istotna z punktu widzenia dopasowywania skrótów klawiszowych do aktualnie używanej aplikacji.

5.7 Biblioteka zdarzeń klawiatury i myszy dla systemu Windows

W systemie Windows symulowanie zdarzeń klawiatury i myszy jest możliwe przy pomocy biblioteki Windows USER (user32.dll) dostępnej w katalogu System32 . Komponent ten dostarcza takich funkcjonalności jak zarządzanie oknami, przekazywanie komunikatów czy właśnie przetwarzanie zdarzeń wejścia (ang. input processing).

Kluczową funkcją pozwalającą na syntezy zdarzeń klawiatury i myszy jest **SendInput()**. Funkcja ta jako argumenty pobiera tablicę zdarzeń wejścia (struktura **INPUT**) oraz jej rozmiar. Struktura **INPUT** reprezentuje pojedyncze zdarzenie, które ma zostać wysłane do strumienia wejścia z klawiatury czy myszy. Działanie tej funkcji jest ograniczone mechanizm UIPI (ang. User Interface Privilege Isolation) systemu Windows. Oznacza to, że zdarzenia wejścia mogą być wysyłane tylko do aplikacji, które mają ten sam lub niższy priorytet w systemie w stosunku do aplikacji inicjującej zdarzenie. Struktura **INPUT** ma za zadanie przechowywać informacje na temat syntezy zdarzenia wejściowego określonego typu. Dopuszczalne typy zdarzeń to **INPUT_MOUSE**, **INPUT_KEYBOARD** oraz **INPUT_HARDWARE**. Dla każdego z tych typów istnieje oddzielna struktura, specyficzna odpowiednio dla zdarzenia myszy **MOUSEINPUT** czy zdarzenia klawiatury **KEYBDINPUT**.

```
typedef struct tagMOUSEINPUT {
    LONG    dx;
    LONG    dy;
    DWORD   mouseData;
    DWORD   dwFlags;
    DWORD   time;
    ULONG_PTR dwExtraInfo;
} MOUSEINPUT, *PMOUSEINPUT;

typedef struct tagKEYBDINPUT {
    WORD    wVk;
    WORD    wScan;
    DWORD   dwFlags;
    DWORD   time;
    ULONG_PTR dwExtraInfo;
} KEYBDINPUT, *PKEYBDINPUT;
```

Rys. 34 Struktury zdarzeń wejścia myszy i klawiatury

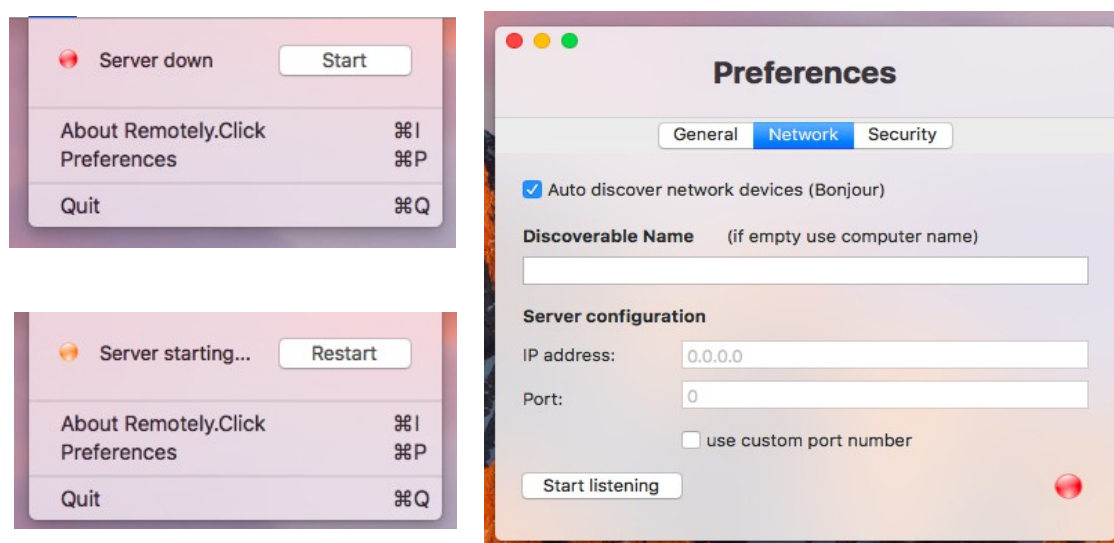
Zdarzenie myszy wymaga zdefiniowania pozycji kursora (dx, dy) oraz flagi **dwFlags** opisującej rodzaj zdarzenia np. kliknięcie, ruch myszy czy scrollowanie. W polu **mouseData** mogą być zdefiniowane dodatkowe dane jak np. wartość przesunięcia przy scrollowaniu.

Zdarzenie klawiatury wymaga natomiast zdefiniowania kodu klawisza **wVk** (ang. virtual key code). Wirtualny kod klawisza musi być liczbą całkowitą w zakresie od 1 do 254. Zamiast wirtualnych kodów klawiszy dopuszczalne jest ustawienie flagi **dwFlags** na wartość **KEYEVENTF_UNICODE** i stosowanie pola **wScan** (kody Unicode). Zdarzenia klawiatury można podzielić na dwa typy tj. wciśnięcie klawisza oraz puszczenie klawisza. W tym drugim przypadku konieczne jest dodanie do pola **dwFlags** flagi **KEYEVENTF_KEYUP**. Zdarzenie wejścia z klawiatury wymaga utworzenia obydwu zdarzeń i wysłanie ich funkcją **SendInput()** w

postaci tablicy dwuelementowej. Użycie klawiszy modyfikujących jak Shift, Windows, Control, Alt wiąże się z utworzeniem dodatkowych zdarzeń wejścia i wysłanie ich w odpowiedniej kolejności razem ze zdarzeniem zwykłego klawiszu.

6. ARCHITEKTURA APLIKACJI GRAFICZNYCH DLA SYSTEMU macOS

Moduł serwera biblioteki zdalnego sterowania komputerem przy pomocy smartfona RemoteControllerAPI_Server został wykorzystany do utworzenia aplikacji GUI na system macOS (Rys. 35). Aplikacja ta ma za zadanie ułatwić użytkownikowi końcowemu korzystanie z programu serwera zdalnego sterowania. Główne cechy tej aplikacji to: możliwość intuicyjnego uruchomienia i zatrzymania serwera, konfiguracja serwera, ustawienie hasła bezpieczeństwa, przeglądanie podłączonych urządzeń mobilnych, ustawienie aplikacji w tryb autostartu. Aplikacja powstała z wykorzystaniem framework'u Cocoa oraz języków Swift/Objective-C. Aplikacje tego typu mogą być tworzone w środowisku programistycznym Xcode.

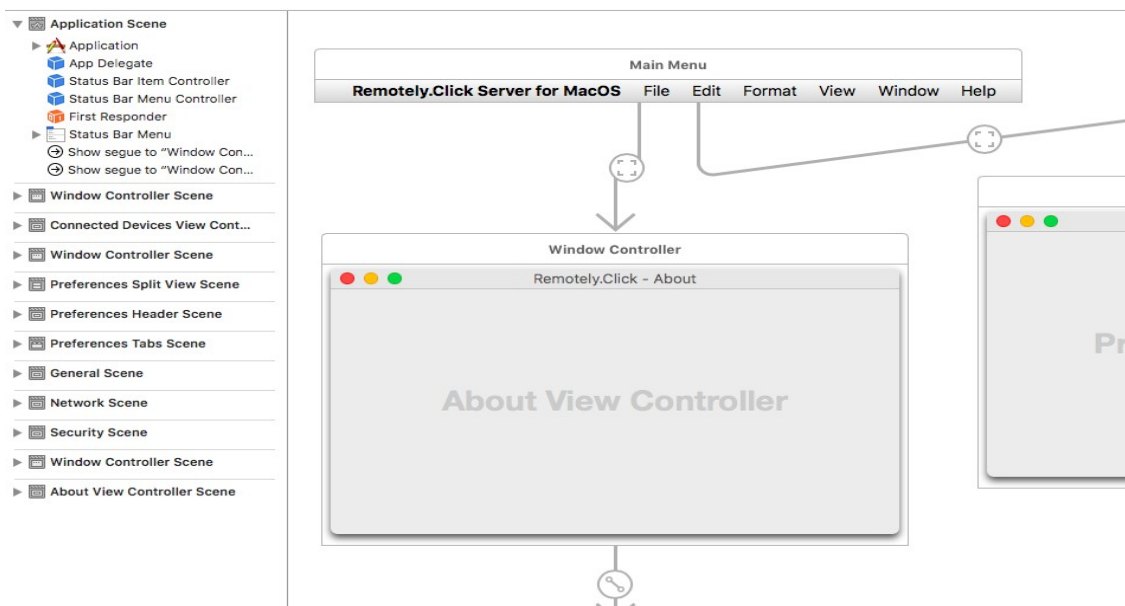


Rys. 35 Aplikacja serwera na system mac OS

Aplikacja RemoteClick Server dla macOS została utworzona jako aplikacja uruchamiana w pasku statusu w prawym górnym rogu systemu operacyjnego (ang. status bar app). Do stworzenia interfejsu graficznego aplikacji wykorzystano pliki .storyboard oraz .xib. Większość logiki biznesowej została napisana w języku Swift. Wyjątek stanowi kod wymagający współpracy z kodem natywnym, który napisano w języku Objective-C.

6.1 Tworzenie GUI w aplikacjach Cocoa

Projektowanie graficznych interfejsów użytkownika w aplikacjach Cocoa jest stosunkowo proste. Poszczególne widoki czy okna aplikacji tworzone są z wykorzystaniem koncepcji historyjek (Storyboard). Storyboard przedstawia nie tylko poszczególne okna aplikacji, ale również pozwala na zdefiniowanie interakcji i przejść (ang. segue) pomiędzy tymi oknami.



Rys. 36 Projektowanie GUI przy pomocy Storyboard

Tworzenie okienek aplikacji sprowadza się do przeciągania i upuszczania kolejnych predefiniowanych komponentów jak: **PushButton**, **Label**, **TextView**, **TextField**, **ImageView** czy **TableView**. Dla poszczególnych widoków lub okien mogą być definiowane klasy w języku Swift lub Objective-C. Klasy te rozszerzają odpowiednio typy **NSWindowController**, **NSViewController**, **NSView**, etc.

W przypadku klas kontrolerów związanych z danym oknem istnieje możliwość odwoływania się do poszczególnych jego komponentów poprzez atrybuty **@IBOutlet**. Atrybuty te można otrzymać w wyniku przeciągania komponentu ze Storyboard'u do kodu kontrolera. Obsługa zdarzeń takich jak kliknięcie przycisku możliwa jest za pomocą metod **@IBAction** w klasach kontrolerów. Podobnie jak w przypadku atrybutów **@IBOutlet** są one generowane w wyniku przeciągania.

Pliki **.xib** pozwalają natomiast projektować pojedyncze widoki, które następnie mogą być ładowane w kodzie w celu utworzenia własnych komponentów. Przykładowo w omawianej aplikacji zaimplementowano w ten sposób widok **ServerStatusMenuItemView**.

6.2 MVC w aplikacjach Cocoa

Aplikacje Cocoa tworzone są w oparciu o architekturę MVC (ang. model-view-controller) (Rys. 37). Widoki są to obiekty odpowiedzialne za wizualną reprezentację modelu. To komponenty projektowane w plikach **.storyboard** i **.xib** oraz przypisane do nich klasy widoków dziedziczące po **NSView**. Umożliwiają bezpośrednią interakcję z użytkownikiem aplikacji.

klasę kontrolera odpowiedniego interfejsu nazywanego w nomenklaturze Cocoa **protokołem** (ang. protocol). Widok poprzez obiekt **delegowania** (ang. delegate) wiedząc, że implementuje on uzgodnione w protokole metody może komunikować się z kontrolerem. W ten sposób odpytuje go jak ma zostać narysowany na ekranie.

Inny sposób komunikacji pośredniej pomiędzy widokiem, a kontrolerem to metody obsługi zdarzeń inicjowanych przez użytkownika aplikacji. W tym przypadku kontroler implementuje metody **@IBAction** wywoływane przez widok w sytuacji wystąpienia zdarzenia. Na podstawie powyższych informacji można stwierdzić, że komunikacja modelu i widoku z kontrolerem jest w Cocoa ściśle usystematyzowana.

7. ARCHITEKTURA APLIKACJI DLA SYSTEMU ANDROID

Moduł klienta biblioteki zdalnego sterowania komputerem **Remote ControllerAPI_Client** został wykorzystany do zaimplementowania aplikacji użytkownika dla systemu Android. W tym celu w środowisku programistycznym Android Studio został utworzony projekt **Remotely.Click for Android** ze wsparciem dla języków natywnych C/C++.

7.1 Dodanie biblioteki natywnej zdalnego sterowania

Dołączenie biblioteki zdalnego sterowania możliwe jest przy pomocy pliku konfiguracyjnego `/mobile/CMakeLists.txt`. Należy użyć komend **add_library()** w celu utworzenia nowej biblioteki z plików pośredniczących napisanych w języku C oraz **add_subdirectory()** w celu dodania biblioteki zdalnego sterowania jako podprojektu CMake. Aby umożliwić wywoływania funkcji z importowanej biblioteki należy załączyć pliki nagłówkowe komendą **include_directories()**. Ostatnim krokiem jest zlinkowanie wynikowej biblioteki z zaimportowanymi bibliotekami zdalnego sterowania korzystając z **target_link_libraries()**. Kod pośredniczący między kodem napisanym w Javie, a kodem bibliotek natywnych umieszczany jest w katalogu `src/main/cpp`.

7.2 Implementacja głównego Activity

Głównym Activity aplikacji jest **MainDrawerActivity** rozszerzające **DrawerActivity**. To w nim przechowywana jest referencja do serwisu klienta **RemoteControllerClientService**. Nawiązuje on połączenie z serwerem i zapewnia interfejs do zdalnego wywoływania procedur.

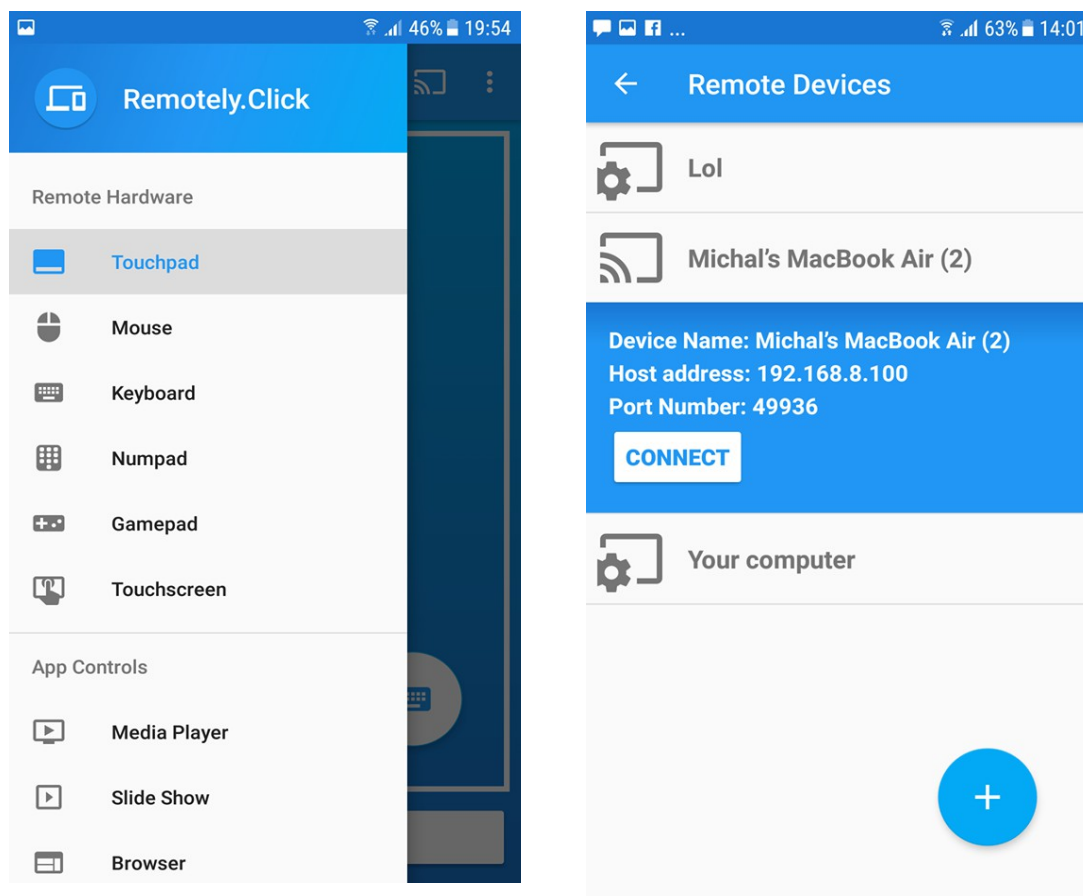
DrawerActivity to natomiast implementacja **AppCompatActivity**, która odpowiedzialna jest za dodawanie do aplikacji paska narzędziowego (ang. **Toolbar**), wysuwnego menu nawigacyjnego (ang. **Navigation Drawer**) oraz menu w pasku narzędziowym (ang. **Toolbar Overflow Menu**).

Navigation Drawer to specjalny wysuwny panel wyświetlający najważniejsze opcje aplikacji. Przez większość czasu jest ukryty i odsłaniany na żądanie użytkownika. Do jego implementacji wymagane jest użycie specjalnego układu widoków ekranu **DrawerLayout**. Główna zawartość ekranu wyświetlana jest jako odpowiedni **Fragment** wewnątrz placeholdera **FrameLayout**. Natomiast menu nawigacyjne tworzone jest przy pomocy **NavigationView** oraz odpowiedniego zasobu z katalogu `res/menu`.

Menu paska narzędziowego powstaje w wyniku nadpisania (ang. **override**) metody **onCreateOptionsMenu()**.

Podsumowując **MainDrawerActivity** odpowiada za zarządzanie serwisem klienta zdalnego

sterowania, podczas gdy `DrawerActivity` dostarcza nawigacji wewnątrz aplikacji.



Rys. 38 Navigation Drawer oraz wyszukiwanie urządzeń

7.3 Wyszukiwanie zdalnych urządzeń

Za wyszukiwanie zdalnych urządzeń, konfigurację nowych urządzeń i nawiązywanie połączenia odpowiada **RemoteDevicesActivity**. Wyświetla ono wszystkie wykryte w sieci lokalnej urządzenia posługujące się protokołem komunikacyjnym **_remotely-click._tcp**.

Lista urządzeń tworzona jest przy pomocy komponentu **RecyclerView**. **RecyclerView** wymaga implementacji **RecyclerView.Adapter**'a będącego obiektem dostarczającym i adaptującym dane dla poszczególnych rekordów listy. Adapter jako źródło danych wykorzystuje urządzenia znalezione w sieci lokalnej przy pomocy technologii Network Service Discovery i obiektu **NSDHelper**. Lista urządzeń pozwala na wyświetlenie szczegółowych informacji o konkretnym urządzeniu oraz nawiązanie z nim połączenia.

Network Service Discovery to biblioteka dla Androida, która implementuje mechanizm DNS-SD pozwalający wyszukiwać w sieci lokalnej usługi określonego typu. Odnajdowanie usług wymaga zaimplementowania obiektu zgodnego z interfejsem **NsdManager.DiscoveryListener** oraz wykonania pojedynczego wywołania asynchronicznego **discoverServices()**. Dla każdej

znajdzonej usługi można pobrać dane adresowe implementując interfejs **NsdManager.ResolveListener** i wywołując metodę **resolveService()**.

7.4 Serwis klienta zdalnego sterowania

Główna logika biznesowa aplikacji skupia się w obiekcie serwisu klienta zdalnego sterowania **RemoteControllerClientService**. Serwis jest uruchamiany w momencie nawiązania nowego połączenia ze zdalnym urządzeniem poprzez **RemoteDevicesActivity**. Serwis ten z jednej strony jest jawnie uruchamiany i kończony wywołaniami **startService()**, **stopService()**. Z drugiej strony poszczególne interfejsy kontrolerów robią dowiązanie do tego serwisu poprzez **MainDrawerActivity** i wywołania **bindService()**, **unbindService()**.

Metoda **onStartCommand()** wypełnia obiekt **ClientInfo**, a następnie nawiązuje połączenie korzystając z biblioteki natywnej i wywołania **rc_client.connect()**. Połączenie ze zdalnym urządzeniem odbywa się na osobnym wątku. Serwis implementuje również funkcje obsługujące zdarzenia programu klienta jak: rozpoczęcie połączenia, uwierzytelnienie połączenia, błąd połączenia czy zakończenie połączenia.

Po zakończonym sukcesem uwierzytelnieniu klienta na serwerze aplikacja wyświetla notyfikację tworzoną **NotificationCompat.Builder**em. Dzięki notyfikacji serwis przechodzi w tryb pierwszoplanowy **startForeground()**. Podczas dowiązywania się do serwisu wywoływana jest metoda **onBind()**, która zwraca obiekt implementujący interfejs **IBinder**. Głównym zadaniem tego obiektu jest umożliwienie dostępu do publicznych metod serwisu będących pomostem do zdalnego wywoływania akcji na serwerze np. **keyboardInput()**, **mouseMove()**, etc.

7.5 Interfejs kontrolera myszy

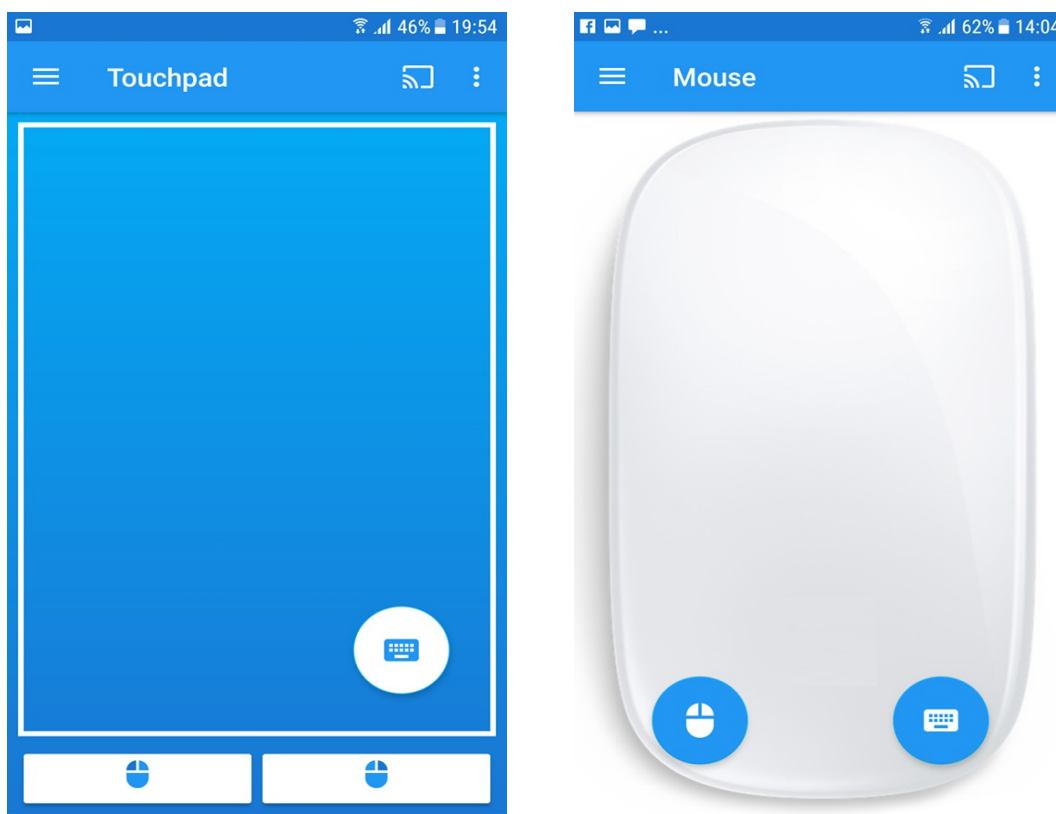
Zdalne wywoływanie zdarzeń myszy jest możliwe poprzez dwa interfejsy graficzne aplikacji: **TouchpadFragment** oraz **MouseFragment**. Widok touchpada został zaimplementowany w niestandardowej (ang. custom) klasie **TouchpadView**. Pozwala on z jednej strony na rysowanie ruchu palca na ekranie, a z drugiej strony na wykrywanie różnego typu gestów mapowanych na zdarzenia myszy.

Za wykrywanie tych gestów odpowiada **TouchpadGestureDetector**. Dostępne gesty to: ruch myszy, pojedyncze kliknięcie, podwójne kliknięcie, potrójne kliknięcie, pojedyncze kliknięcie prawym palcem, podwójne kliknięcie prawym palcem, potrójne kliknięcie prawym palcem, scrollowanie dwoma palcami wertykalnie i horyzontalnie, gest zoomowania (ang. pinch gesture), gest rotacji. Wykrywanie gestów ma miejsce w metodzie **onTouchEvent()**. Przekazywany jest do niej obiekt **MotionEvent** z informacją o aktualnym położeniu palców w konkretnym widoku. **MotionEvent** pozwala rozpoznać następujące akcje: **ACTION_DOWN**, **ACTION_POINTER**

_DOWN, ACTION_POINTER_UP, ACTION_UP, ACTION_MOVE. Odpowiednie śledzenie sekwencji tych akcji pozwala na wykrywanie i obsługę powyższych gestów.

Oprócz obsługi gestów touchpada MouseFragment dodatkowo implementuje sterowanie kursorem myszy przy pomocy czujników ruchu (ang. motion sensor). Większość współczesnych urządzeń mobilnych posiada wbudowane różnego rodzaju czujniki takie jak: akcelerometr, żyroskop, etc. W przypadku interfejsu myszy zastosowano żyroskop (ang. gyroscope). Implementacja tego interfejsu została zawarta w klasie **SensorMouse**.

Dostęp do czujników urządzenia możliwy jest poprzez obiekt **SensorManager**. Aby uzyskać dostęp do żyroskopy wystarczy wywołać na menadżerze metodę getDefaultSensor() i podać odpowiedni typ czujnika Sensor.TYPE_GYROSCOPE. Klasa SensorMouse umożliwia wznawianie i zatrzymywanie odczytu danych żyroskopu. Implementuje ona funkcję nasłuchującą dane czujnika **onSensorChanged()**. Ruch myszy symulowany jest na podstawie wartości przyspieszenia kątownego żyroskopu względem osi Z oraz X.



Rys. 39 Interfejs touchpada oraz myszy

7.6 Interfejs kontrolera klawiatury

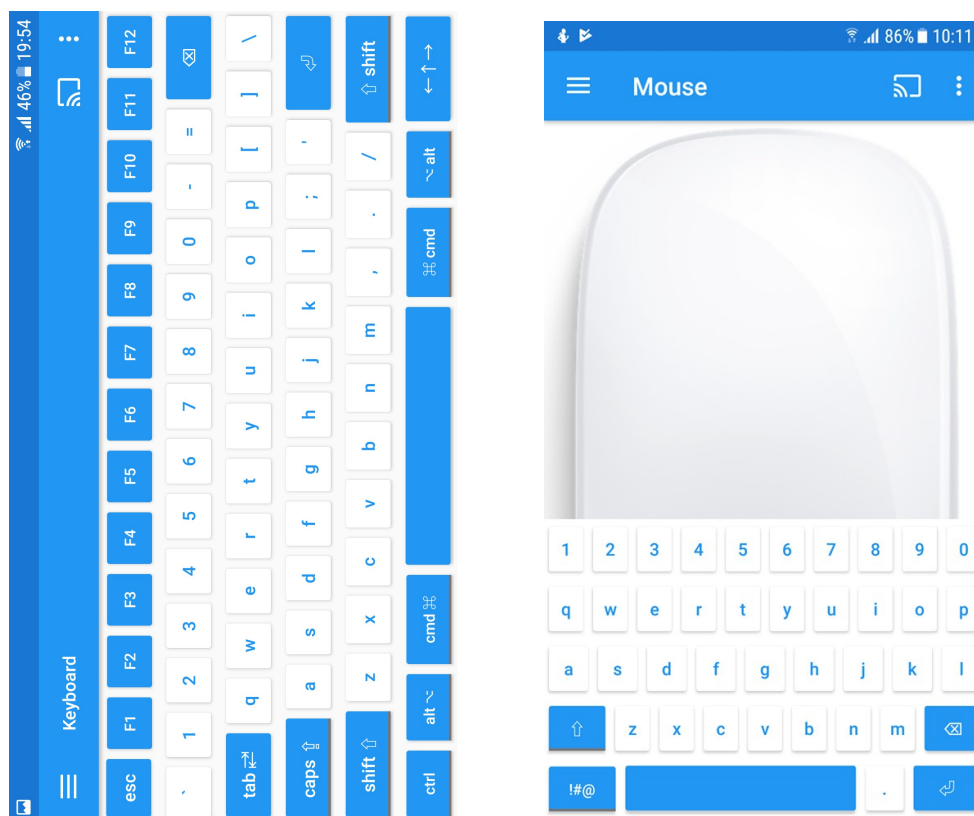
Zdalne wywoływanie zdarzeń klawiatury możliwe jest poprzez jeden z dwóch interfejsów graficznych. Pierwszym jest pełnowymiarowa klawiatura stosowana w KeyboardFragment. Drugim

uproszczona klawiatura stosowana do celów pomocniczych w pozostałych fragmentach, np. TouchpadFragment czy MouseFragment.

Pełnowymiarowa klawiatura została zaimplementowana w klasie **LandscapeKeyboard**. Natomiast uproszczona klawiatura w klasie **BottomSheetKeyboard**. Obie klawiatury dziedziczą ze wspólnej klasy **AbstractKeyboard** oraz pobierają w konstruktorze dwa argumenty: układ przycisków (ang. keyboard layout) oraz interfejs klienta RemoteControllerClientInterface.

Układ klawiatury jest odpowiednio dostosowywany, aby umożliwić przełączanie między znakami alfanumerycznymi i symbolami czy zmianę wielkości liter po naciśnięciu klawisza modyfikującego Shift.

Interfejs klienta pozwala na powiadamianie serwisu klienta o wystąpieniu zdarzenia wejścia z klawiatury. W wyniku zdarzenia następuje wywołanie metody keyboardInput() i przekazanie odpowiednio zszyntezowanych kodu klawisza i flag klawiszy modyfikujących.



Rys. 40 Klawiatura pełnowymiarowe i uproszczona

7.7 Interfejsy odtwarzacza i innych zdarzeń systemowych

Oprócz standardowych interfejsów umożliwiających zdalne sterowanie zdarzeniami klawiatury i myszy, aplikacja udostępnia kilka dodatkowych interfejsów ułatwiających zdalną pracę z komputerem.

Interfejs odtwarzacza umożliwia współpracę z popularnymi odtwarzaczami wideo jak VLC, odtwarzacze osadzone w stronach www np. YouTube czy bazujące na Silverlight. Interfejs odtwarzacza oparty jest o skróty klawiszowe (ang. hot keys).

Interfejs przeglądania prezentacji slajdów również bazuje na skrótach klawiszowych i pozwala na zdalne sterowanie pokazem slajdów w aplikacjach takich jak PowerPoint czy OpenOffice.

Interfejs klawiatury numerycznej stanowi uproszczoną wersję klawiatury syntezującej zdarzenia klawiszy numerycznych.

Interfejs przeglądarki internetowej definiuje pulpit z ikonami odnoszącymi się do popularnych skrótów klawiszowych przeglądarek internetowych. Z łatwością można utworzyć nową zakładkę, poruszać się pomiędzy otwartymi zakładkami, przechodzić w tryb pełnoekranowy czy otwierać okno w trybie prywatnym. Interfejs ten jest kompatybilny z najpopularniejszymi przeglądarkami internetowymi: Safari, Opera, Google Chrome czy Firefox.

Oprócz tego w aplikacji zdefiniowane są dwa dodatkowe interfejsy skrótów i zdarzeń systemowych. Pozwalają one na wywoływanie zdarzeń zamknięcia czy restartowania systemu oraz popularnych skrótów takich jak kopiuj, wytnij, wklej, drukuj, etc.

8. PODSUMOWANIE

8.1 Realizacja celów

W ramach projektu z powodzeniem utworzono przenośną bibliotekę zdalnego sterowania komputera smartfonem. W obecnej wersji moduł serwera biblioteki została przetestowana zarówno na systemie macOS jak i Windows. W przypadku systemu macOS bibliotekę wykorzystano do utworzenia aplikacji GUI Remotely.Click Server for macOS. Na systemie Windows wykonano testy biblioteki w środowisku Cygwin. Powstał również program Remotely.Click Server for Windows utworzony jako aplikacja WPF w środowisku .NET/C#. W przyszłości planowane jest wykorzystanie utworzonej biblioteki w ramach aplikacji WPF.

Moduł klienta biblioteki zdalnego sterowania został przewidziany do implementacji w aplikacjach na urządzenia mobilne. W celu przetestowania jego funkcjonalności utworzona została aplikacja dla systemu Android. W ramach tej aplikacji utworzono kilka interfejsów zdalnego sterowania komputerem tj. touchpad, myszkę (w tym myszkę sensoryczną), interfejs pełnej i uproszczonej klawiatury, klawiaturę numeryczną, interfejs kontroli odtwarzaczy multimedialnych, pilot do prezentacji slajdów, pulpit popularnych skrótów klawiszowych.

W ramach pracy przeanalizowano różne realizacje programów serwera. Z jednej strony rozważano zastosowanie serwera iteracyjnego lub współbieżnego. Z drugiej wykorzystanie protokołu połączeniowego TCP bądź bezpołączeniowego UDP. Ostatecznie zdecydowano się na użycie serwera współbieżnego połączeniowego opartego o pulę wątków. W ten sposób umożliwiono jednoczesne podłączanie się wielu urządzeń mobilnych do jednego komputera.

W module serwera zawarta została biblioteka implementująca wywołania zdarzeń myszy i klawiatury. Bibliotekę tą utworzono w dwóch wersjach. Dla systemu macOS wykorzystano framework Core Graphics i wchodzący w jego skład zestaw funkcji Quartz Event Services. Początkowo rozważano użycie AppleScript, jednak ze względu na brak obsługi zdarzeń myszy oraz ograniczenia wynikające z konieczności uzyskiwania zezwolenia do wykonywania zdarzeń klawiatury porzucono ten pomysł. W systemie Windows udało się z powodzeniem syntezywać zdarzenia myszy i klawiatury przy pomocy biblioteki Windows USER.

Niestety w przypadku obu systemów zestaw możliwych do zasymulowania zdarzeń myszy był ograniczony. Poprzez wywołania systemowe okazało się niemożliwe wywoływanie typowych gestów touchpada jak: zoomowanie, rotacja, przesuwanie pomiędzy pulpitemi itp. W związku z tym biblioteki uzupełniono o możliwość wykonywania popularnych skrótów klawiszowych, które zniwelowały powyższe problemy.

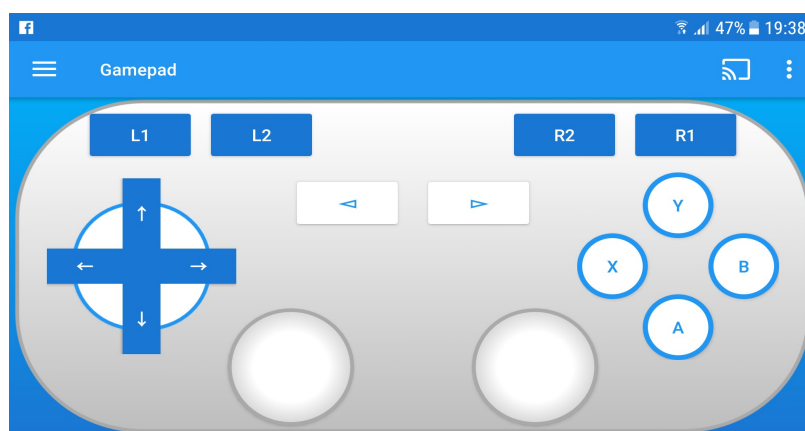
Zdalne wykonywanie skrótów klawiszowych pozwoliło na implementację uproszczonych interfejsów do kontroli wybranych aplikacji, np. odtwarzaczy wideo czy prezentacji slajdów.

Jednocześnie należy pamiętać, że skróty te w wielu przypadkach mogą ulegać zmianie, a użytkownicy mogą korzystać z bardzo różnych aplikacji do realizacji tych samych usług. Aby rozwiązać ten problem podjęto decyzję o przechowywaniu definicji skrótów klawiszowych w oddzielnych plikach .plist lub .xml. W ten sposób umożliwiono użytkownikom stosunkowo łatwą modyfikację i rozszerzanie bazy skrótów klawiszowych. Należałoby rozważyć implementację odpowiednich paneli konfiguracyjnych w aplikacji serwera, pozwalających na ustawianie przez użytkowników własnych skrótów klawiszowych.

8.2 Perspektywy rozwoju produktów w oparciu o bibliotekę zdalnego sterowania

Omawiana biblioteka zdalnego sterowania komputera smartfonem w wersji w której została zaimplementowana daje duże możliwości jej dalszego rozwoju. Rozwój ten może być dwojaki.

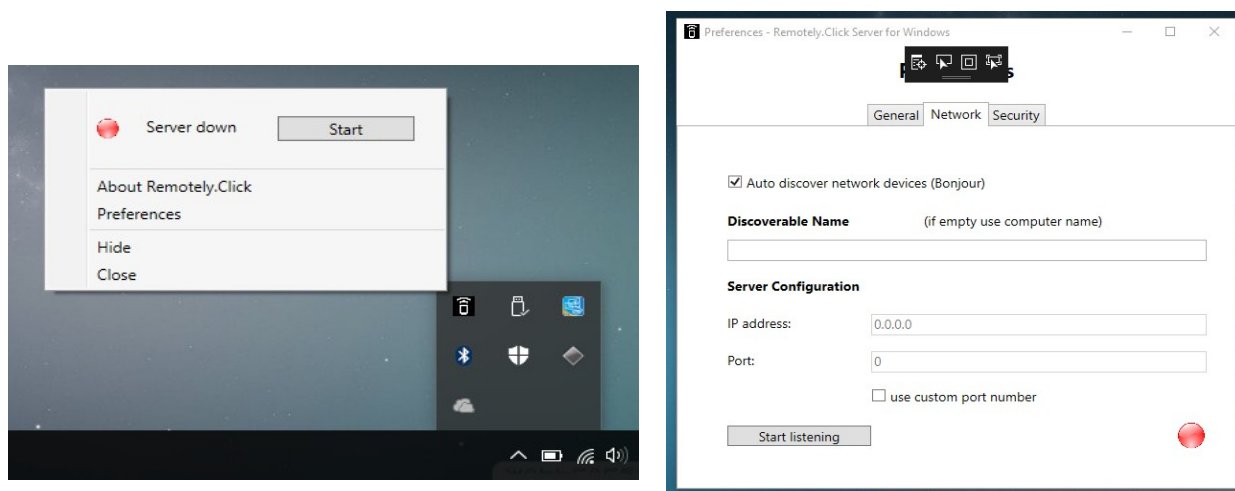
Z jednej strony możliwe jest poszerzanie funkcjonalności o obsługę dodatkowych zdalnych procedur. Przykładowo możliwe jest utworzenie nowych zdalnych interfejsów do kontrolowania komputera takich jak kontroler gier (ang. gamepad) (Rys. 50) czy ekran dotykowy (ang. touch screen). Biblioteka wywołań systemowych realizujących zdarzenia klawiatury i myszy może zostać zaimplementowana również na inne systemy operacyjne np. z rodziny Unix.



Rys. 41 Interfejs kontrolera gier (ang. gamepad)

Z drugiej strony biblioteka może być wykorzystana do tworzenia nowych aplikacji użytkownika dla większej liczby systemów operacyjnych. Moduł klienta z łatwością może być zaimplementowany w aplikacji na system iOS, natomiast moduł serwera po odpowiednich dostosowaniach w aplikacji na system Windows. Kod biblioteki zdalnego sterowania był testowany na systemie Windows z wykorzystaniem środowiska Cygwin. Przykładowa aplikacja paska zadań (ang. task bar app) dla systemu Windows napisana z wykorzystaniem frameworku Windows Presentation Foundation mogłaby wyglądać tak jak na Rys. 51.

Nowe produkty mogą swym zakresem obejmować również inne typy urządzeń mobilnych jak inteligentne zegarki (ang. smartwatch). Aplikacja androidowa może zostać rozszerzona o moduł przeznaczony dla platformy Android Wear. Analogicznie planowana aplikacja dla systemu iOS może zostać uzupełniona o moduł napisany dla platformy watchOS.



Rys. 47 Remotely.Click Server dla Windows

BIBLIOGRAFIA

1. <http://gs.statcounter.com/os-market-share>
2. <http://edu.pjwstk.edu.pl/wyklady/sko/scb/w2.html>
3. https://pl.wikipedia.org/wiki/Model_TCP/IP
4. https://pl.wikipedia.org/wiki/Model_OSI
5. Beej's Guide to Network Programming, Brian Beej Jorgensen
6. Internetworking with TCP/IP Vol I: Principles, Protocols, and Architecture, Douglas E. Comer
7. Internetworking with TCP/IP Vol 3: Client-Server Programming and Applications,
8. http://www.bogotobogo.com/cplusplus/sockets_server_client.php
9. Wykłady Programowanie Sieciowe, Michał Cieśla
10. Zero Configuration Networking: The Definitive Guide, O'Reilly, Stuart Cheshire, Daniel H. Steinberg
11. [https://en.wikipedia.org/wiki/Android_\(operating_system\)](https://en.wikipedia.org/wiki/Android_(operating_system))
12. Android Sensor Programming, Wrox, Greg Milette, Adam Stroud
13. Android Cookbook, O'Reilly, Ian F. Darwin
14. <https://developer.android.com>
15. <https://www.raywenderlich.com/149112/android-fragments-tutorial-introduction>
16. https://www.tutorialspoint.com/android/android_fragments.htm
17. https://en.wikipedia.org/wiki/Java_Native_Interface
18. https://en.wikipedia.org/wiki/Thread_pool
19. <https://developer.apple.com/documentation/coregraphics/>
20. Stanford University's Developing Applications for iOS Lecture CS193p