Systemy Wbudowane

Projekt:

Sterowanie kursorem w systemie Linux
za pomocą aplikacji na telefon z systemem Android

Jacek Wiślak
Michał Mirowski
PK FMI, Informatyka

1.Założenia projektu

Projekt zakłada sterowanie wskaźnikiem (myszką) za pomocą aplikacji napisanej na telefon komórkowy z systemem Android w wersji 2.3.x i wyższej.
Aplikacja odbierająca i wykonująca koordynaty, oraz zdarzenia takie jak: kliknięcie lewym klawiszem myszy, kliknięcie prawym klawiszem myszy, będzie napisana jako moduł jądra systemu Linux.

Mogło by się wydawać, że tak niska warstwa w architekturze systemu, dla tego typu aplikacji jest złym miejscem, ponieważ może ona wpłynąć na stabilność systemu, oraz wywołać jakiś błąd krytyczny, z drugiej strony jednak ma to swoje plusy, taka aplikacja nie jest uzależniona od konkretnego systemu okien, system traktuje ją w podobny sposób jak każde fizycznie podłączone urządzenie wskazujące.

Zważywszy na to że ostatnio dyskutuje się dużo, jakoby w najpopularniejszej dystrybucji linuxowej jaką jest Ubuntu, odejść od popularnego xorg na rzecz mira, sądzę że przy prawidłowo napisanej aplikacji, przestrzeń jądra (linux kernel space) jest wbrew pozorom dobrym miejscem na tego typu aplikacje, zapewniającą kompatybilność z przyszłymi środowiskami graficznymi czy różnymi serwerami okien.

Pozostaje jeszcze kwestia łączności aplikacji mobilnej z komputerem, można by to zrealizować na kilka sposobów np. łącząc komputer i telefon przez bluetooth, jednak bardziej uniwersalnym rozwiązaniem wydaję się połączenie tych urządzeń przy użyciu sieci. W tym celu skorzystaliśmy z protokołu UDP, który w przeciwieństwie do TCP, nie ma narzutu na nawiązywanie połączenia i śledzenie sesji, jednak dzięki uproszczeniu jego budowy zapewnia on szybszą transmisję, oraz brak konieczności wykonywania dodatkowych zadań, co przy zastosowaniu protokołu TCP mogło by wpłynąć na płynną prace wirtualnej myszki.
Zdecydowaliśmy również, że telefon będzie łączył się z komputerem, co oznacza konieczność utworzenia gniazda (socketu) UDP z poziomu modułu jądra.

2.Aplikacja serwera (moduł jądra)

Co to jest moduł jądra?

Najprościej będzie przytoczyć fragment zaczerpnięty z artykułu Piotra Sobolewskiego pt. „Zrób własny moduł jądra Linuksa”

> *Chcesz, żeby Winamp odtwarzał muzykę w jakimś egzotycznym formacie? Możesz napisać wtyczkę, która rozszerzy jego funkcjonalność. Będzie to osobny plik, który Winamp ładuje przy starcie. W rezultacie w menu pojawiają się dodatkowe opcje, aplikacja zaczyna "rozumieć" formaty dotychczas dla niej nieczytelne albo podejmuje zupełnie nowe działania, np. wysyła pliki z muzyką przez Gadu-Gadu, pobiera teksty piosenek itp.*

> *W podobny mechanizm jest wyposażony Linux. Jeśli chcesz dodać do niego nową funkcję, możesz napisać moduł jądra. W ten sposób można dodać obsługę nowego sprzętu, systemu plików - niemal wszystko, co przyjdzie ci do głowy. W artykule powiemy, na czym to polega i od czego zacząć.*

Zakładam, że Ty czytelniku posiadasz minimalny zakres wiedzy o modułach jądra, wobec czego nie będę wyjaśniał wszystkich elementarnych pojęć oraz zasada działania takiego modułu.
Jednak jeżeli nie miałeś wcześniej do czynienia z modułami jądra, przed przystąpieniem do dalszej lektury tego poradnika/dokumentacji zachęcam cię do przeczytania wyżej zacytowanego artykułu, który znajdziesz na stronie: http://www.pcworld.pl/artykuly/57347/Zrob.wlasny.modul.jadra.Linuksa.html

Postaram się, wyjaśnić tobie każdą część działania modułu, zacznę od pliku modułu jądra.

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/in.h>
#include <net/sock.h>
#include <linux/skbuff.h>
#include <linux/delay.h>
#include <linux/inet.h>
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/pci.h>
#include <linux/input.h>
#include <linux/platform_device.h>

#define SERVER_PORT 5555

/* Struktura reprezentujaca urzedzenie wejsciowe */
struct input_dev *vms_input_dev;
/* Struktura reprezetujaca urzedzenie */
static struct platform_device *vms_dev;
/* Struktura reprezentująca gniazdo (socket) */
static struct socket *udpsocket=NULL;
/* Struktura reprezentująca kolejke danych odbieranych przez socket */
struct workqueue_struct *wq;
```

Jak widać poza listą niezbędnych includów, została również zadeklarowana stała **SERVER_PORT**, której wartość oznacza port przez który będzie identyfikowane gniazdo. Do obsługi socketu będzie potrzebna nam struktura **socket** oraz **workqueue_struct**, ta druga będzie odpowiedzialna za wykonywanie odpowiednich czynności, po odebraniu danych przez socket.

Pozostałe struktury są odpowiedzialne za emulacje urządzenia wskazującego.

Ze względu na obszerny kod funkcji inicjalizującej moduł omówię ją w dwu częściach, pierwszej dotyczącej socketu oraz work queue, drugiej urządzenia wejściowego.

Funkcja inicjalizująca moduł:

```
static int __init server_init( void )
{
    ////////////SOCKET//////////////////
    struct sockaddr_in server;
    int servererror;
    printk("INIT MODULE\n");
    ///////socket do odbierania danych///////
    if (sock_create(PF_INET, SOCK_DGRAM, IPPROTO_UDP, &udpsocket) < 0) {
        printk( KERN_ERR "server: Error creating udpsocket.n" );
        return -EIO;
    }
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons( (unsigned short)SERVER_PORT);
    servererror = udpsocket->ops->bind(udpsocket, (struct sockaddr *) &server,
sizeof(server ));
    if (servererror) {
        sock_release(udpsocket);
        printk( KERN_ERR "server: blad przy bindowaniu portu" );
        return -EIO;
    }
    udpsocket->sk->sk_data_ready = cb_data;

    /* utworzenie work queue */
    INIT_WORK(&wq_data.worker, send_answer);
    wq = create_singlethread_workqueue("myworkqueue");
    if (!wq){
        return -ENOMEM;
    }
```

Do utworzenia gniazda służy funkcja **sock_create**, której lista argumentów przedstawia się następująco: int sock_create(int family, int type, int protocol, struct socket **res) potrzebujemy jeszcze określić adres lokalny, lub zdalny punkt końcowy do podłączenia gniazda. W tym celu należy zdefiniować strukturę **sockaddr_in** która będzie odpowiedzialna za reprezentacje tego adresu. Gdy wszystko jest już gotowe możemy, spróbować podłączyć socket do danego adresu korzystając z funkcji **bind**, aby moduł nie musiał monitorować ciągle stanu gniazda, należy ustawić wskaźnik **sk_data_ready** na funkcję, która

będzie wykonywana w przypadku odebrania danych przez socket.
Na końcu jest tworzony work queue, który będzie wykonywany w przypadku odbioru danych, za uruchomienie wykonywania tego work queue odpowiedzialna jest funkcja **cb_data**, natomiast utworzony work queue ma za zadanie wykonanie funkcji **send_answer**

```
    ////////////////////MYSZKA//////////////
    /* Zarejestruj urządzenie w systemie */
    vms_dev = platform_device_register_simple("vms", -1, NULL, 0);
    if (IS_ERR(vms_dev)){
    printk ("vms_init: error\n");
    return PTR_ERR(vms_dev);
    }

    /* Zainicjuj strukture input_dev */
    vms_input_dev = input_allocate_device();
    if (!vms_input_dev) {
        printk("Bad input_allocate_device()\n"); return -ENOMEM;
    }

    /* Przypisz input device unikalną nazwe */
    vms_input_dev->name="remote mouse";

    /* Ustawienie zdarzen generowanych przez urzadzenie */
    set_bit(EV_REL, vms_input_dev->evbit);
    set_bit(EV_KEY, vms_input_dev->evbit);
    set_bit(BTN_0, vms_input_dev->keybit);
    set_bit(BTN_2, vms_input_dev->keybit);
    set_bit(REL_X, vms_input_dev->relbit);
    set_bit(REL_Y, vms_input_dev->relbit);

    /* Zarejestruj urządzenie wejsciowe */
    input_register_device(vms_input_dev);

    printk("WIRTUALNA MYSZKA ZAINICJALIZOWANA.\n");

    return 0;
}
```

Myślę, że większość funkcji dotyczących emulacji urządzenie wskazującego, nie wymaga wyjaśnienia. Omówię natomiast ustawienie zdarzeń generowanych przez nasze urządzenie.
Podobnie jak prawdziwa myszka, nasz moduł będzie generował relatywne przesunięcia kursora oraz kliknięcia prawym lub lewym klawiszem myszy, związku z czym należy ustawić odpowiednie bity w strukturze odpowiadającej za obsługę urządzenia wskazującego (input device), za kliknięcia lewym przyciskiem myszy odpowiada bit przypisany pod makro **BTN_0**, prawym **BTN_2**, jak można się domyślić **BTN_1** odpowiada środkowemu przyciskowi myszki, który nie jest obsługiwany przez nasz moduł.
Za przesuwania kursora względem osi X odpowiedzialny jest bit **REL_X** natomiast dla osi Y jest to bit **REL_Y.**

Jak widać na końcu funkcji inicjalizującej, jest wypisywany komunikat, który możemy odczytać w logu systemowym.

Funkcje pomocnicze

```
short int bin_to_int(unsigned char *p, int index) {
    short int liczba=0;
    liczba=liczba|(p[index]<<8);
    liczba=liczba|p[index+1];
    return liczba;
}

void int_to_bin(short int liczba, unsigned char *tab) {
    tab[0]=liczba>>8;
    tab[1]=liczba;
}
```

Dane pomiędzy urządzeniami są wysyłane w postaci pakietów, aby przesłać wartości odpowiadające przesunięciom relatywnym kursora, musimy zamienić je na ciąg bajtów o określonej strukturze, tak aby zarówno program nadawczy i odbiorczy wysyłały je w określony sposób.
Przyjęliśmy, że będziemy przechowywać wartości odpowiadające przesunięciom kursora w zmiennych short int, ponieważ niezależnie od architektury komputera, w języku C zmienne short int zawsze są wielkości 16-bitowej, dodatkowy plus tego rozwiązania to odpowiednik short int w języku Java (short).

Struktura danych przesyłanych przez sieć przedstawia się następująco:

X (bity 15..8)	X (bity 7..0)	Y (bity 15..8)	Y (bity 7..0)

Aby wydobyć wartości przesunięcia relatywnego kursora z danych przesłanych przez telefon zdefiniowaliśmy funkcję **bin_to_int**, która dla podanego ciągu bajtów, począwszy od pozycji podanej argumentem index, wyodrębnia liczbę przesłaną przez urządzenie sterujące.

Czujny czytelnik zapewne zauważył brak zmiennej/rejestru odpowiedzialnego za obsługę kliknięć, aby zmniejszyć ilość przesyłanych danych, wydzieliliśmy specjalne wartości zmiennych X, Y, informujące moduł o danym zdarzeniu, w naszym przypadku będzie to kliknięcie prawym lub lewym przyciskiem myszy. Jeżeli wartość zmiennej **X będzie równa 32767** oznacza to kliknięcie **lewym** przyciskiem myszy, natomiast jeżeli **Y= 32767** oznacza to kliknięcie **prawym** przyciskiem myszy.

Odbiór danych z sieci

Jak wspomniałem wyżej po odebraniu danych przez socket wykonywana jest funkcja **send_answer**

```
void send_answer(struct work_struct *data){
    struct  wq_wrapper * foo = container_of(data, struct  wq_wrapper, worker);
    int len = 0;
    int x,y;
    /* Dopuki sa jakies nie przetworzone pakiety petla jest wykonywana */
    while((len = skb_queue_len(&foo->sk->sk_receive_queue)) > 0){
        struct sk_buff *skb = NULL;

        /* odebranie pakietu */
        skb = skb_dequeue(&foo->sk->sk_receive_queue);
        x=bin_to_int(skb->data+8,0);
        y=bin_to_int(skb->data+8,2);

        ////////////////////SSTEROWANIE MYSZKA////////////////
        if (x==32767) {
            input_report_key(vms_input_dev, BTN_0, 1);
            input_report_key(vms_input_dev, BTN_0, 0);
        }
        else if (y==32767) {
            input_report_key(vms_input_dev, BTN_2, 1);
            input_report_key(vms_input_dev, BTN_2, 0);
        }
        else {
            input_report_rel(vms_input_dev, REL_X, x);
            input_report_rel(vms_input_dev, REL_Y, y);
        }

        input_sync(vms_input_dev);
        ////////////////////////////////////////////////////

        /* zwolnij pamiec zarezerwowana dla pakietu skb */
        kfree_skb(skb);
    }
}
```

Jak widać oprócz deklaracji zmiennych lokalnych, funkcja wykonuje kod zawarty w pętli while, dopóki nie zostaną przetworzone wszystkie pakiety, odebrane przez socket.

Pierwszym etapem jest **odebranie pakietu** i przypisanie go do tymczasowego bufora **skb**, z którego następnie jest **wyodrębniane x i y**, których wartość odpowiada relatywnemu przesunięciu wobec osi x i y.

Kolejny etap to sprawdzenie czy w owych wartościach nie znajdują się **wartości specjalne**, odpowiedzialne za **kliknięcie lewym** lub **prawym przyciskiem myszy**. Jeżeli warunek jest spełniony, wykonywana jest symulacja kliknięcia odpowiednim przyciskiem myszy, poprzez **ustawienie** odpowiedniego **bitu**, a **następnie jego wyzerowanie**, gdybyśmy go nie wyzerowali, system myślałby że przycisk jest cały czas wciśnięty.

Natomiast jeżeli w buforze nie znajduję się **żadna z wartości specjalnych**, zgłaszane jest do urządzenia **przesunięcie kursora** odpowiadające wartości odebranych x i y.

Funkcja zamykająca moduł

```
static void __exit server_exit( void )
{

    ////////////////////SOCKET////////////////
    if (udpsocket)
        sock_release(udpsocket);

    if (wq) {
                flush_workqueue(wq);
                destroy_workqueue(wq);
    }

    //////////////MYSZKA////////////////////
    /* Wyrejestruj urządzenie wejsciowe */
    input_unregister_device(vms_input_dev);

    /* Wyrejestruj urzadzenie */
    platform_device_unregister(vms_dev);

    printk("KONIEC PROGRAMU");
}
```

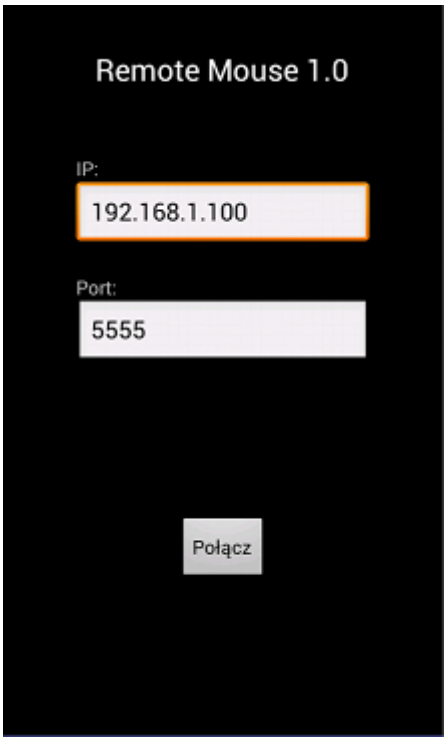
Jak widać w funkcji zamykającej moduł dokonywane jest czyszczenie utworzonego socketu, work queue, oraz wyrejestrowanie urządzeń związanych z urządzeniem wejściowym.

3.Aplikacja sterująca (Android)

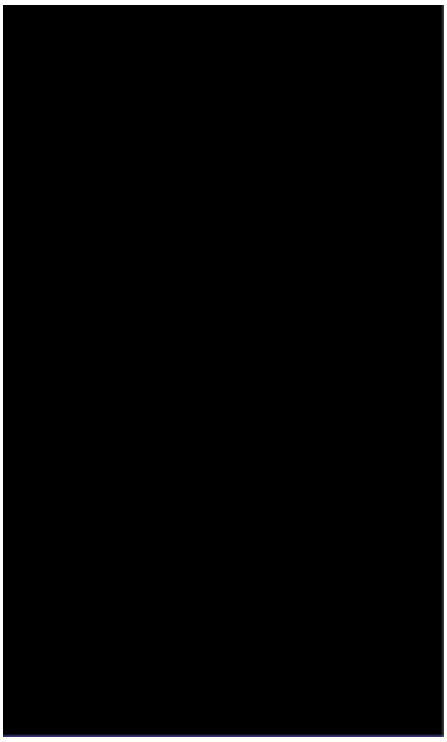
Aplikacja sterująca została napisana w języku **Java**, przy użyciu dostarczonego przez google **SDK**, które zostało połączone z kompilatorem **Eclipse** przy użyciu odpowiednej wtyczki.

Aplikacja składa się z **dwóch szablonów** (layoutów) pierwszy który odpowiedzialny jest za nawiązanie połączenia, oraz drugi który funkcjonalnością ma przypominać standardowy touchpad montowany w komputerach przenośnych, wobec czego usunięto wszystkie elementy, które mogłyby przeszkadzać czy też ograniczać powierzchnie „touchpada”

Layout startowy (pierwszy)



Layout touchpada (drugi)



Jeden z problemów, który można napotkać pisząc taką aplikacje, to problem z obsługą socketów, o ile socket można utworzyć w głównym wątku, to wysyłanie danych trzeba zrealizować w osobnym.
Jest to ograniczenie nałożone z góry przez Androida, taki mechanizm ma jednak swoje plusy, gdyby wysyłanie było realizowane w tym samym wątku co program główny, moglibyśmy odnieść wrażenie że program się zacina, ponadto jeśli wystąpiłby błąd podczas wysyłania pakietu, został by zamknięty cały program, a tak zostanie zakończony tylko wątek odpowiedzialny za wysłanie pakietu.

Aby nie musieć korzystać z wątków przy każdej małej operacji takiej jak np. wysyłanie pakietów, SDK udostępnia nam specjalną klasę **AsyncTask**, która jest typem generycznym, pozwalającym wykonywać nam określone czynności „w tle”

Poniżej przedstawiam kod funkcji **onCreate** wykonywanej po starcie aplikacji

```
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (!polaczony)
            setContentView(R.layout.activity_main);
        else
            setContentView(R.layout.touchpad);
        gestureScanner = new GestureDetector(this);
        Button btn1 = (Button)findViewById(R.id.button1);
        btn1.setOnClickListener(this);
```

```
    }
```

Aby obsłużyć zdarzenia związane z danym elementem GUI, musimy ustawić dla niego Listener, w naszym przypadku chcemy reagować na naciśnięcie guzika „Połącz” którego id=button1.

Jak widać funkcja w zależności od flagi **polaczony** ustawia widoczny odpowiedni layout. Flaga ta została wprowadzona, związku ze specyficznym zachowaniem androida przy obrocie ekranu, który po wykryciu takiego zdarzenia uruchamia od nowa to samo activity.
Wyobraźmy sobie sytuacje gdy połączenie jest już ustanowione, jako widoczny jest ustawiony layout touchpada, użytkownik dokonuje obrócenia ekranu telefonu i niestety ku jego zaskoczeniu, widzi layout startowy, zamiast layoutu touchpada, intuicyjnie naciska jeszcze raz przycisk połącz.
Następuje próba tworzenia socketu, który został już utworzony i oczywiście pojawia się komunikat o niespodziewanym zamknięciu aplikacji.
Aby uniknąć tego typu sytuacji zdefiniowaliśmy w polach klasy MainActivity flagę:

```
private boolean polaczony=false;
```

Dodatkowo zdefiniowaliśmy funkcje, która będzie wykonywana przy wykryciu zdarzenia obrotu ekranu:

```
    public void onConfigurationChanged(Configuration newConfig) {
        super.onConfigurationChanged(newConfig);
        if (soket!=null)
            this.polaczony=true;
    }
```

Jak widać do momentu w, którym socket nie zostanie utworzony, wartość flagi **poloczony** pozostaje bez zmian, w przeciwnym wypadku jest ona „podnoszona” co oznacza przypisanie wartości true.

Wróćmy do obsługi przycisku, za obsługę zdarzenia kliknięcia w Listenerze odpowiedzialna jest funkcja onClick

```
    public void onClick(View v) {
        switch(v.getId()){
            case R.id.button1:
                String ip;

                //pobranie ip i portu z formularza
                ip=((EditText) findViewById(R.id.editText1)).getText().toString();
                port = Integer.valueOf(
                        ((EditText)
findViewById(R.id.editText2)).getText().toString());

                //tworzenie gnizda
                try {
                    soket = new DatagramSocket(port);
                } catch (SocketException e) {
                    // TODO Auto-generated catch block
```

```
                    e.printStackTrace();
                }
                //pobranie ip serwera
                try {
                    adresip = InetAddress.getByName(ip);
                } catch (UnknownHostException e)
                    e.printStackTrace();
                }
                setContentView(R.layout.touchpad);
                polaczony=true;
                break;
        }
    }
```

W zależności od obiektu, który zgłosił zdarzenie wykonywana jest odpowiednia procedura, u nas jest ona zdefiniowana tylko dla przycisku „Połącz”
Ma ona za zadanie utworzenie socketu, oraz zmianę layoutu ze startowego na layout touchpada.

Zanim omówię obsługę zdarzeń ekranu dotykowego, przedstawię klasę **sendudp**, która rozszerza klasę **AsyncTask**, umożliwiając nam wysłanie pakietu w tle.

```
class sendudp extends AsyncTask<Void, Void, Void> {

    private DatagramPacket pakiet;
    private DatagramSocket soket;

    public sendudp(DatagramSocket socket, DatagramPacket packet) {
        this.pakiet=packet;
        this.soket=socket;
    }


    @Override
    protected Void doInBackground(Void... params) {
        try {
            soket.send(pakiet);
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return null;
    }

 }
```

Klasa w konstruktorze otrzymuje **pakiet** i **socket** na który chcemy wysłać przekazany w konstruktorze pakiet.
Posiada tylko jedną metodę **doInBackground**, która ma za zadanie wysłanie pakietu.

Jak obiecałem wyżej przechodzę do omówienia **obsługi zdarzeń wyświetlacza dotykowego**.

Na początek weźmy pod lupę metodę **onTouchEvent** odpowiedzialną za pobieranie współrzędnych z ekranu i przetwarzanie ich na relatywne przesunięcie kursora

```
    public boolean onTouchEvent(MotionEvent event) {
        short x=1,y=1,tmpx, tmpy;
        byte [] wspolrzedne=new byte[4];
        DatagramPacket pakiet;


        switch (event.getAction()) {

        case MotionEvent.ACTION_DOWN:
             prevx = (short) new Float(event.getX()).intValue();
             prevy = (short) new Float(event.getY()).intValue();
            break;
        case MotionEvent.ACTION_MOVE:
            //pobranie wsp□ednych dotyku
            x=(short) new Float(event.getX()).intValue();
            y=(short) new Float(event.getY()).intValue();
            tmpx=x;
            tmpy=y;

            ///odejmowanie wsp□ednych od startowych
            x-=prevx;
            y-=prevy;

            //przypisanie wsp□ednych jako poprzednie
            prevx=tmpx;
            prevy=tmpy;

            //konwersja do przyjetego formatu bufora
            wspolrzedne[0]=(byte) (x>>8);
            wspolrzedne[1]=(byte) x;
            wspolrzedne[2]=(byte) (y>>8);
            wspolrzedne[3]=(byte) y;

            //tworzenie pakietu
            pakiet=new DatagramPacket(
                    wspolrzedne, wspolrzedne.length,this.adresip,this.port);

            //wysylanie pakietu w odzielnym watku
            new sendudp(this.soket, pakiet).execute();
            break;

        }
        return gestureScanner.onTouchEvent(event);


    }
```

W tej metodzie Android umożliwia nam wykrycie zdarzeń takich jak: naciśnięcie palcem na ekran, przesunięcie palca po ekranie, odjęcie palca od ekranu.
My wykorzystamy tylko pierwsze dwie wymienione możliwości, a dokładnie **ACTION_DOWN** oraz **ACTION_MOVE.**
Gdy palec zostaje przyłożony do ekranu są zapamiętywane jego współrzędne, jako współrzędne poprzedniego zdarzenia, jeżeli kolejno nastąpi przesunięcie palca obliczane, jest relatywne przesunięcie, czyli różnica pomiędzy poprzednim przesunięciem a aktualnym, po obliczeniu tej różnicy aktualne współrzędne

przypisywane są jako współrzędne poprzedniego zdarzenia.
Następnie tworzony jest pakiet, który zawiera obliczone wartości i jest on wysyłany w osobnym wątku, poprzez utworzenie nowej instancji klasy sendudp i wykonaniu na niej metody execute.

Aby wykryć gesty takie jak kliknięcie w ekran, skorzystaliśmy z gotowej klasy **GestureDetector** która jest zawarta w SDK, dzięki czemu wystarczyło zaimplementować dwie metody, wykonywane odpowiednio przy kliknięciu na ekran, oraz przy dłuższym przytrzymaniu palca przy ekranie.

Dla zwykłego kliknięcia tzw. „tapnięcie”

```
    @Override
    public boolean onSingleTapUp(MotionEvent arg0) {
        byte [] wspolrzedne=new byte[4];
        DatagramPacket pakiet;

        //konwersja do przyjetego formatu bufora
        wspolrzedne[0]=(byte) (32767>>8);
        wspolrzedne[1]=(byte) 32767;
        wspolrzedne[2]=0;
        wspolrzedne[3]=0;

        //tworzenie pakietu
        pakiet=new DatagramPacket(
                wspolrzedne, wspolrzedne.length,this.adresip,this.port);

        //wysylanie pakietu w odzielnym watku
        new sendudp(this.soket, pakiet).execute();

        return false;
    }
```

Dla dłuższego przytrzymania palca

```
    @Override
    public void onLongPress(MotionEvent arg0) {
        byte [] wspolrzedne=new byte[4];
        DatagramPacket pakiet;

        //konwersja do przyjetego formatu bufora
        wspolrzedne[0]=0;
        wspolrzedne[1]=0;
        wspolrzedne[2]=(byte) (32767>>8);
        wspolrzedne[3]=(byte) 32767;

        //tworzenie pakietu
        pakiet=new DatagramPacket(
                wspolrzedne, wspolrzedne.length,this.adresip,this.port);

        //wysylanie pakietu w odzielnym watku
        new sendudp(this.soket, pakiet).execute();
    }
```

Mamy nadzieję, że udało nam się omówić wszystkie rzeczy mogące sprawić trudność przy pisaniu tego typu aplikacji.
Liczymy, że udało nam się przekonać Ciebie drogi czytelniku, że wbrew pozorom pisanie modułów jądra w systemach Linuxowych nie jest takie straszne, niemniej jednak jest to temat traktowany po macoszemu przez naszych rodowitych Informatyków, co wnioskuje choćby po brakuje polskiej literatury opisującej tworzenie tego typu modułów.

4. Bibliografia

1. **Sreekrishnan Venkateswaran**: *Essential Linux Device Drivers,* 2008. ISBN: 978-0132396554

2. **Piotr Sobolewski**: *Zrób własny moduł jądra Linuksa* Dostępny w World Wide Web:
http://www.pcworld.pl/artykuly/57347/Zrob.wlasny.modul.jadra.Linuksa.html