FELADATKIÍRÁS

Az elektronikusan beadott változatban ez az oldal törlendő. A nyomtatott változatban ennek az oldalnak a helyére a diplomaterv portálról letöltött, jóváhagyott feladatkiírást kell befűzni.

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Sándor Dávid

BIOMETRIKUS AUTENTIKÁCIÓS MEGOLDÁS FEJLESZTÉSE

KONZULENS

Dr. Kővári Bence

Golda Bence

BUDAPEST, 2022

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Sándor Dávid**, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2022. 04. 21

...…….
Sándor Dávid

Összefoglaló

A szakdolgozat, vagy diplomaterv elkészítése minden egyetemi hallgató életében egy fontos mérföldkő. Lehetőséget ad arra, hogy az egyetemi évei során megtanultakat kamatoztassa és eredményeit szélesebb közönség előtt bemutassa, s mérnöki rátermettségét bizonyítsa. Fontos azonban, hogy a dolgozat elkészítésének folyamata számos csapdát is rejt magában. Rossz időgazdálkodás, hiányos szövegszerkesztési ismeretek, illetve a dolgozat készítéséhez nélkülözhetetlen „műfaji" szabályok ismeretének hiánya könnyen oda vezethetnek, hogy egy egyébként jelentős időbefektetéssel készült kiemelkedő szoftver is csak gyengébb minősítést kapjon a gyenge minőségű dolgozat miatt.

E dokumentum – amellett, hogy egy általános szerkesztési keretet ad a dolgozatodnak – összefoglalja a szakdolgozat/diplomaterv írás írott és íratlan szabályait. Összeszedjük a Word kezelésének legfontosabb részeit (címsorok, ábrák, irodalomjegyzék stb.), a dolgozat felépítésének általános tartalmi és szerkezeti irányelveit. Bár mindenkire igazítható sablon természetesen nem létezik, megadjuk azokat az általános arányokat, oldalszámokat, amelyek betartásával jó eséllyel készíthetsz egy színvonalas dolgozatot. A részletes és pontokba szedett elvárás-lista nem csupán a dolgozat írásakor, de akár más dolgozatok értékelésekor is kiváló támpontként szolgálhat.

Az itt átadott ismeretek és szemléletmód nem csupán az aktuális feladatod leküzdésében segíthet, de hosszútávon is számos praktikus fogással bővítheti a szövegszerkesztési és dokumentumkészítési eszköztáradat.

Abstract

Ide jön a ½-1 oldalas angol nyelvű összefoglaló, amelynek szövege a Diplomaterv Portálra külön is feltöltésre kerül. Ez a magyar nyelvű összefoglaló angolra fordított változata.

1 Bevezetés

A feladat bemutatása.

1.1 Motiváció

A motiváció ismertetése.

1.2 A dolgozat szerkezete

A dolgozat szerkezetének bemutatása.

2 Háttérismeretek

Az alábbi fejezetben olyan témakörök, fogalmak, technológiák kerülnek bemutatásra, amelyek a diplomaterv értelmezését segítik. A fejezetnek nem célja az adott témakörök részletes dokumentációja, a hangsúly minden esetben a diplomaterv megértéséhez elengedhetetlenül fontos fogalmak bemutatásán van. Amennyiben az Olvasó egy adott témakörhöz kapcsolódó további szakirodalmat keres, ajánlom az irodalomjegyzékben összegyűjtött források, hivatkozások tanulmányozását.

2.1 Biometrikus authentikáció

A biometrikus authentikáció alatt a hitelesítési mechanizmusoknak egy olyan családját értjük, amelyek alapjául valamilyen biológiai megkülönböztető jel, vagy viselkedésbeli karakterisztika szolgál. Általánosságban a biometrikus authentikációs megoldásokról elmondható, hogy az azonosítás alapjául hasznát információt nehezebb ellopni vagy hamisítani, mint a klasszikus jelszó alapú megoldások esetén.

Ez részben annak köszönhető, hogy a klasszikus hitelesítési megoldások determinisztikusak, azaz például egy jelszó esetén 100%-os karakteregyezés szükséges a sikeres hitelesítéshez. Ezzel ellentétben a biometrikus megoldások probabillisztikusak (valószínűségi alapon működnek). Egy jelszavas hitelesítés esetén a rendszer *mindenkit* beenged, aki a helyes jelszót adja meg, tehát tulajdonképpen bárki, aki el tudja lopni, vagy fel tudja törni a jelszót kérdés nélkül hozzáfér a rendszerhez. Ezzel szemben egy biometrikus authentikációs eredménye egy százalékos érték, azaz mennyire valószínű az, hogy a felhasználó tényleg az, akinek mondja magát.

Az ilyen megoldások alapjául szolgáló biometriákat alapvetően két nagyobb csoportra lehet osztani:

- **Fiziológiai (statikus)**

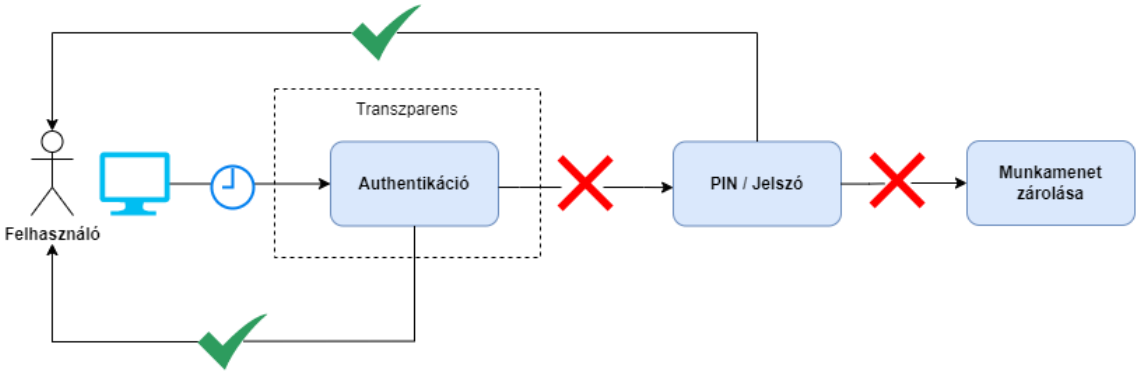
  A felhasználó fizikai karakterisztikáit használják, például újlenyomat, retina, arc formája, vénák szerkezete.

- **Viselkedés alapú (dinamikus)**

  A felhasználó viselkedési karakterisztikáit használják, például kézírás, beszéd, mozgás, billentyű leütések ritmikája, **kurzormozgás.**

2.2 Folyamatos authentikáció

A folyamatos authentikáció egy olyan hitelesítési módszer, ahol a felhasználó személyazonosságát valós időben lehet megerősíteni. A klasszikus statikus authentikációtól (például egy jelszó megadása) abban tér el, hogy a felhasználót a teljes munkamenet során, folyamatosan ellenőrzi. A folyamatos authentikációs módszerek alapja leggyakrabban viselkedési minták vagy biometriák szoktak lenni. Előnye, hogy a munkafolyamat megszakítása nélkül tud működni, a felhasználó számára láthatatlan módon.

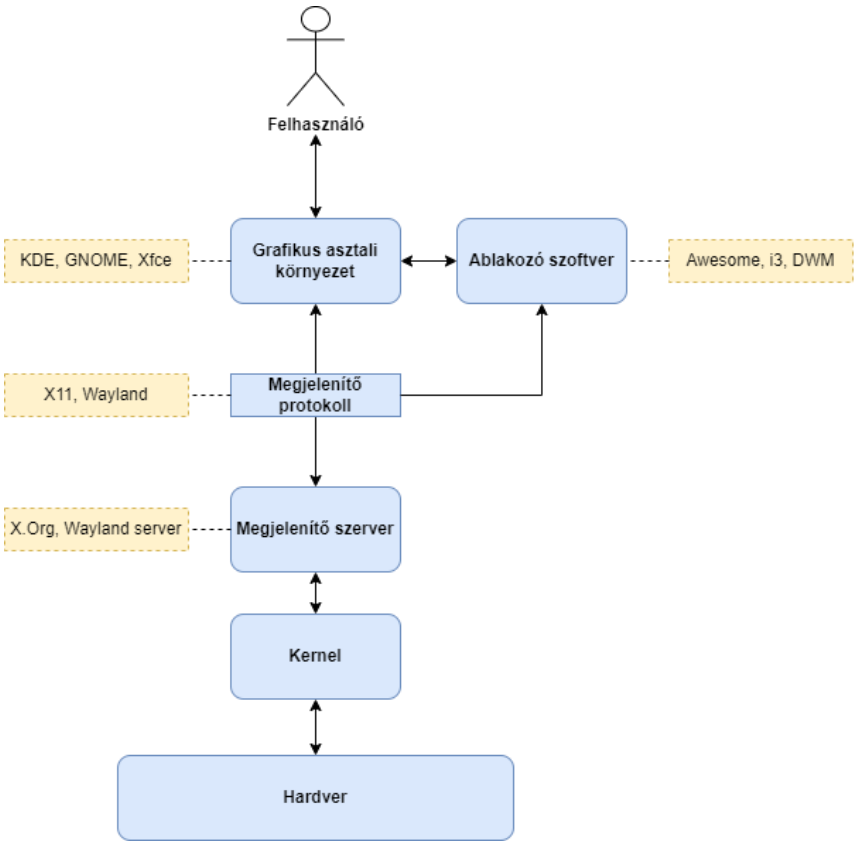


1. ábra A folyamatos authentikáció

A folyamatos authentikáció működéséről ad egy áttekintő képet az *1. ábra* **A folyamatos authentikáció**. A felhasználó valamilyen munkamenetet végez (például banki tranzakció). Bizonyos időközönként a háttérben a felhasználó személye ellenőrzésre kerül, a folyamatos authentikáció alapjául szolgáló módszer (például viselkedési biometriák) alapján. Amennyiben az ellenőrzés sikeres, úgy a felhasználó megszakítás nélkül folytathatja a munkamenetet, nem érzékel semmit a háttérben futó authentikációból. Amennyiben az ellenőrzés eredménye nem meggyőző, ilyenkor általában egy klasszikus egy faktoros azonosítási módszert szoktak alkalmazni (például PIN kód, vagy jelszó megadása). Ha így sikerült azonosítania magát a felhasználónak, akkor visszatérhet a munkamenethez, ha nem, akkor pedig a munkamenet zárolásra kerül (például tranzakció megszakítása, kijelentkeztetés a banki felületről).

2.3 Unix-szerű grafikus felhasználói felületek

A Unix-szerű operációs rendszerek esetén a grafikus felhasználói felület (amennyiben a rendszer egyáltalán rendelkezik GUI-val (Graphical User Interface)) meglehetősen összetett. Egy GUI általában több különböző komponensből áll, például (a teljeség igénye nélkül): asztali környezet, ablakozó szoftver, widget könyvtárak, bemeneti / kimeneti eszközök, megjelenítő szerver, stb. A következő ábra betekintést nyújt egy tipikus GUI felépítésébe:



2. ábra A GUI felépítése

A felhasználó elsősorban az **asztali környezettel** (desktop environment) lép interakcióba. A legtöbb grafikus interfésszel rendelkező Linux disztribúció valamilyen asztali környezetet használ. Jelenleg a legelterjedtebb asztali környezetek közé tartozik a GNOME, a KDE illetve a Xfce. Ezek a szoftverek különböző GUI elemeket biztosítanak (ikonok, widget-ek, háttérképek), interfészeket nyújtanak grafikus felületek programozására. Az asztali környezetek gyakran nem csak felületet és külalakot nyújtanak, sokszor tartalmaznak saját fájlkezelőt, beállítóprogramot, levelezőklienst és egyéb felhasználói programokat. Architekturálisan egy magasabb absztrakciós szinten

helyezkednek el, mint a megjelenítő szerver, nem kommunikálnak közvetlenül a kernellel.

Az **ablakozó szoftver** (window manager) olyan szoftver, amely egy ablakrendszerben az ablakok elhelyezését és megjelenését vezérli egy grafikus felhasználói felületen. Ez lehet egy asztali környezet része vagy önállóan is használható.

Azt a szoftvert, ami a különböző GUI komponenseket összefogja és lehetővé teszi, hogy ezek hatékonyan együttműködjenek **megjelenítő szervernek** (display server) hívják. A megjelenítő szerver kezeli az alsóbb szintű funkciókat, közvetlenül kommunikál a kernellel (ezen keresztül pedig a hardver erőforrásokkal). A képernyőre való rajzolást és a bemeneti / kimeneti eszközök adatainak továbbítását a grafikus alkalmazások felé szintén a megjelenítő szerver végzi. A többi felsőbb szintű komponenst egymással integrálja, interfészeket biztosít, amelyeken keresztül az alsóbb szintű funkciók elérhetővé válnak.

A megjelenítő szerver kliensének tekintünk általában minden grafikus felülettel rendelkező alkalmazást, de természetesen GUI nélküli programok is lehetnek kliens alkalmazások. A megjelenítő szerver a klienseivel a **megjelenítő protokollon** (display protocol) keresztül kommunikál.

2.4 Megjelenítő protokollok [1]

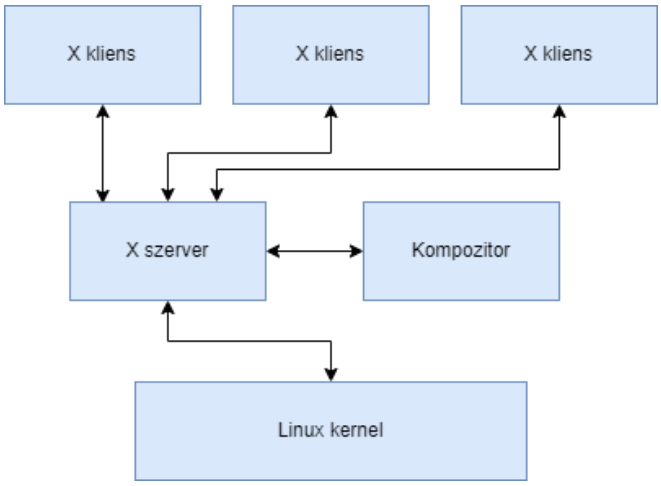
Linux rendszerek esetén többféle megjelenítő protokollal találkozhatunk és egy adott protokollhoz általában többféle implementáció is létezik. Az idők során két megjelenítő protokoll terjedt el nagyobb körben, az X Window System, illetve a Wayland protocoll. Mivel a legnépszerűbb grafikus interfésszel rendelkező Linux disztribúciók LTS (Long Term Support) verziója szinte kivétel nélkül a fent említett megjelenítő protokollok egyikét használja, ezért a diplomaterv során további megjelenítő protokollokkal nem foglalkoztam.

2.4.1 X Window System

Az X Window System (X11, helyenként csak X) egy nyílt forráskódú megjelenítő protokoll készlet, amely Unix-szerű rendszerekhez készült. A protokollt 1984-ben kezdték el kifejleszteni és jelenleg a 11-es verziónál tart (innen ered az X11 kifejezés). Maga a protokoll egy szöveges leírás, ami publikusan elérhető. A protokollhoz tartozik egy szerveroldali referencia implementáció, amit X.Org Server-

nek hívnak. A protokollt megvalósító elterjedtebb C-ben implementált kliensoldali könyvtárak az Xlib és az XCB.

Architektúráját tekintve az X Window System egy kliens-szerver architektúrát valósít meg. Az X szerver vezérli a fizikai megjelenítő készülékeket és feldolgozza a bemeneti eszközöktől érkező adatokat. Az X kliensek olyan alkalmazások, amelyek az X szerveren keresztül szeretnének interakcióba lépni a bemeneti / kimeneti eszközökkel.



3. ábra Az X Window System architektúrája

A rendszerhez tartozik még egy komponens, a kompozitor. A kompozitor feladata, hogy a különböző ablakok elrendezését vezérelje a képernyőn.

Bizonyos kifejezéseket az X Window System árnyaltabban használ a közbeszédhez képest, a legfontosabbak ezek közül a következők:

- **device (eszköz)** – Dedikált vagy alaplapra integrált videókártya.
- **monitor** – Fizikai megjelenítő eszköz.
- **screen (képernyő)** – Egy olyan terület, amelyre grafikus tartalmat lehet renderelni. Ez egyszerre több monitoron is megjelenhet (akár duplikálva, akár kiterjesztve).
- **display (kijelző)** – Képernyők gyűjteménye, amely gyakran több monitort foglal magába. A Linux-alapú számítógépek általában képesek arra, hogy több kijelzővel rendelkezzenek egyidejűleg. Ezek között a felhasználó egy speciális billentyűkombinációval, például a control-alt-funkcióbillentyűvel válthat, átkapcsolva az összes monitort az egyik kijelző képernyőinek megjelenítéséről a másik kijelző képernyőire.

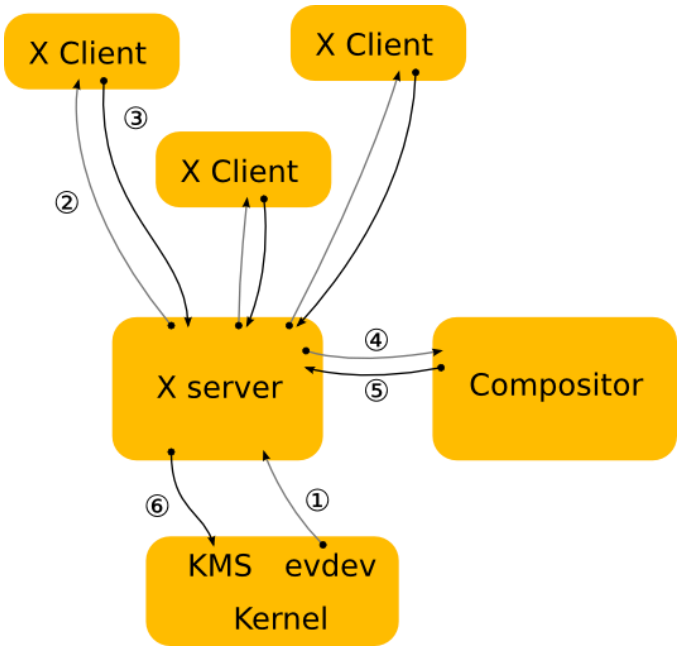
Az X protokoll négy különböző üzenet típust definiál, amelyeknek a szerver és a kliensek közti kommunikációban van szerepe. A protokoll a következő üzenet típusokat különbözteti meg:

- **request** – A kliens küldi a szervernek. Egy request sokféle információt tartalmazhat, mint például egy új ablak létehozását, vagy a kurzor pozíciójának lekérdezését.
- **reply** – A szerver küldi a kliensek. A reply üzenetek a request üzenetek hatására jönnek létre és a kliens által kért információt tartalmazzák.
- **event** – A szerver küldi a kliensnek. Az ilyen típusú üzeneteket általában nem közvetlenül a kliens váltja ki. Sokféle típusú event üzenet létezik, ilyen például a bemeneti eszközök (például billentyűzet vagy egér) által generált események.
- **error** – A szerver küldi a kliensnek. Hasonlóan működnek az event típusú üzenetekhez, valamilyen hiba fennállását jelzik.

2.4.2 Wayland [2]

A Wayland egy ingyenes, nyílt forráskódú megjelenítő protokoll. A Wayland projekt célja, hogy leváltsa az X Window System-et egy modernebb, egyszerűbb és biztonságosabb protokollra. A protokollt 2008-ban kezdték el fejleszteni és a mai napig aktívan dolgoznak rajta. Az X Window System-hez hasonlóan a Wayland protokoll is egy szerver-kliens architektúrát követ. Ellentétben az X-szel a Wayland esetében a megjelenítő szervert kompozitornak hívják. Ez abból az alapvető architektúrális különbségből ered, hogy a Wayland esetében a megjelenítő szerver és a kompozitor egy komponensként funkcionál. A protokollhoz tartozik egy szerveroldali referencia implementáció C-ben, amit Weston kompozitornak hívnak. A legnépszerűbb kliensoldali könyvtár a libwayland szintén C-ben lett implementálva.

A legjobb módja annak, hogy összehasonlítsuk a Wayland és az X Window System architektúráját és megértsük a különbségeket az, ha végig követjük egy bemeneti eszköz által generált esemény útját egészen addig, ameddig az esemény által kiváltott változás megjelenik a képernyőn. Az X esetében ez a következőképpen zajlik le:



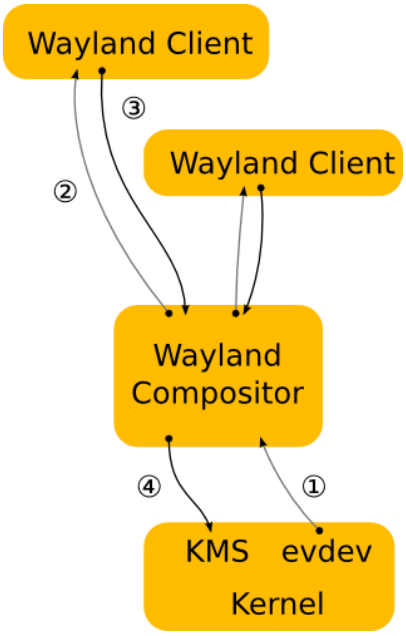
4. ábra Egy esemény feldolgozása az X-szen.

1. A kernel kap egy eseményt egy bemeneti eszköztől és elküldi az X szervernek a bemeneti vezérlőn (evdev driver) keresztül. A különböző eszközspecifikus eseményprotokollok lefordítását az evdev szabványra a kernel végzi el.
2. Az X szerver meghatározza, hogy melyik ablakot érintette az esemény és elküldi azoknak az X klienseknek, amelyek az adott ablakban a kérdéses eseményre feliratkoztak.
3. A kliensek feldolgozzák az eseményt és eldöntik, hogy mit tegyenek. Sokszor egy bemeneti esemény hatására a felhasználói felületnek meg kell változnia, például a felhasználó egy hivatkozás felé viszi a kurzort, vagy bepipál egy checkbox-ot. Az esemény feldolgozása után a kliens egy renderelési kérést (request típusú üzenet) küld a szervernek.
4. Az X szerver megkapja a renderelési kérést és egy illesztőprogramon keresztül szól a hardvernek, hogy az végezze el a renderelést. Az X szerver

továbbá kiszámítja a renderelés határoló régióját és elküldi ezt a kompozitornak egy káreseménynek (damage event) nevezett üzenetben. Erre azért van szükség, mert a kompozitornak a bemeneti esemény hatására lehet, hogy bizonyos effekteket kell alkalmaznia (forgatás, skálázás, stb).

5. A káreseményből a kompozitor megtudja, hogy valami megváltozott az ablakban és az képernyőnek azt a részét újra kell komponálnia, ahol az ablak megváltozott. Miután ezt megtette, a kompozitor küld egy renderelési kérést az X szervernek.
6. Az X szerver megkapja a renderelési kérést a kompozitortól és végrehajtja azt.

A Wayland protokoll esetén a kompozitor és a megjlenítő szerver egy és ugyanaz a komponens. Ez lehetővé teszi, hogy a kompozitor közvetlenül a klienseknek küldje a bemeneti eseményeket és fordítva a kliensek közvetlenül a kompozitornak küldik a káreseményeket. Ugyanez a folyamat a Wayland esetében a következőképpen zajlik le:



5. ábra Egy esemény feldolgozása Wayland-en.

1. A kernel kap egy eseményt egy bemeneti eszköztől és elküldi a Wayland kompozitornak a bemeneti vezérlőn keresztül.
2. A kompozitor meghatározza, hogy melyik ablakot érintette az adottt esemény és tájékoztatja erről az érintett klienseket. A kompozitor érti a

különböző effekteket, transzformációkat, amikkel az egyes elemek rendelkezhetnek, így képes az ablak-lokális – képernyő-lokális koordináták fordítására. Ezáltal a kompozitor pontosan meg tudja határozni, hogy melyik ablakot érintette az adott esemény, feleslegessé válik az X-es acrcitektúrában a 4-es és az 5-ös lépés.

3. Az X-es architektúrához hasonlóan a kliens megkapja az eseményt és feldolgozza azt. Egy újabb különbség a protokollok között, hogy a Wayland esetén a renderelés kliens oldalon történik (gyakorlatban a kliens a kompozitorral megosztott közös videómemória pufferbe renderel). Végezetül a kliens értesíti a kompozitort, hogy jelezze, hogy a felhasználói felületen változás történt (káresemény).
4. A kompozitor összegyűjti a kliensektől a káreseményeket és újra összeállítja a képernyőt.

Az X Window System architektúrájában a kompozitor felelős azért, hogy mindent megjelenítsen a képernyőn, de ezt mégis az X szerveren keresztül kell tennie. Lényegében az X szerver egy közvetítő szerepet játszik a kliensek és a kompozitor, illetve a kompozitor és a hardver között. A Wayland protokollban azáltal, hogy a meglejelenítő szerver helyére lép a kompozitor lényegesen csökkent a rendszer komplexitása, illetve a kommunikációs többlet.

2.5 Rust [3]

A Rust egy viszonylag új általános célú programozási nyelv. Az első stabil verziója 2014-ben jelent meg. A nyelv megalkotásakor a hangsúly a teljesítményen, a konkurrencia támogatásán és a biztonságon volt. Leggyakrabban rendszerszintű programozásra szokták használni, de magasabb szinten is népszerűnek számít. Alapvetően a nyelv a C-re és a C++-ra épít, de más nyelvekből is vesz át jól bevált ötleteket. A Rust egy kompilált nyelv, ami betartatja memóriabiztonságot (tehát minden hivatkozás érvényes memória területre mutat) *garbage collector* mechanizmus használata nélkül. A memóriabiztonság betartására, illetve a versenykörülmények elkerülésére a Rust a tulajdonossági mechanizmust (ownership) használja, ami leegyszerűsítve az objektumok élettartamának követését és a változók hatókörének (scope) ellenőrzését jelenti fordítási időben.

A diplomaterv implementálása során az adatgyűjtő és munkamenet záróló klienst Rust nyelven készítettem el. A választásban több szempont is szerepet játszott. Egyrészt a kliens alkalmazás jellegét figyelembe véve a biztonságtechnikai szempontok meglehetősen fontosak. Ebből kifolyólag a kompilált nyelvek előnyt élveznek az interpretált nyelvekkel szemben, nehezebb a program működését futás közben módosítani. Másrészt a Rust nyelv teljesítményét tekintve nem sokkal marad el a C/C++-tól [4], ugyanakkor a használata (legalábbis számomra) meglehetősen könnyeb. Természetesen az is egy nyomós érv volt, hogy a nyelvhez elérhető X11 protokoll implementáció [5], amelyet az XML protokoll leírásból generáltak, nem pedig egy már más nyelven készült, meglévő implementációhoz tartalmaz burkoló kódot. Végső soron pedig az új, korszerű technológiával való megismerkedés is szerepet játszott a döntésben.

2.6 Erlang

Az Erlang egy univerzális, konkurens, funkcionális programozási nyelv és futási időben *garbage collection* mechanizmussal ellátott környezet. Az Erlang és az Erlang/OTP (Open Telecom Platform) kifejezést sokszor felcserélhető módon használják. Az Erlang/OTP az Erlang környezetből, számos „off-the-shelf” Erlang könyvtárból és tervezési mintából (viselkedésleíró sablon) áll. A diplomamunkám során az alkalmazás szervert Erlang nyelven implementáltam. Az Erlangot a következő jellemvonásokkal rendelkező rendszerek megalkotására tervezték:

- **Elosztottság**: A nyelv magas szinten támogatja a moduláris és konkurrens programozást.
- **Hibatűrés**: Az Erlang számos eszközt és tervezési irányelvet („Let-it-crash”, felügyeleti fa) biztosít, amellyel a hibák előfordulása és a rendszerre gyakorolt hatása minimalizálható.
- **Magas rendelkezésre állás**: Az Erlang folyamatok izolációjából következően, ha egy folyamat hibába ütközik és leáll, az a rendszernek csak egy kisebb, elkülönített részében fog szolgáltatás kiesést okozni.
- **Laza valós idejűség** (Soft real-time): Az Erlangot eredetileg telekommunikációs rendszerek létrehozására alkották meg, így a valós idejű működés egy alapvető kritérium volt a kezdetektől fogva.

- **Kód cserélése futásidőben** (Hot swapping): Az Erlang/OTP-ben található tervezési minták generikus módon támogatják egy futó folyamat kódjának frissítését anélkül, hogy a folyamatot le kéne állítani.

2.6.1 Típusok

A következő leírásban az Erlang nyelv adattípusairól, illetve egyedi típusok specifikálásáról lesz szó. Az Erlang egy dinamikusan erősen típusos (dinamically strongly typed), egyszeri értékadást használó programozási nyelv [6]. Több alap adattípust definiál, ilyenek például az integer, binary vagy az atom. A beépített adattípusok felhasználásával lehetséges saját típusok specifikálása. A típus specifikációnak többek közt dokumentációs célja van, továbbá nagy mértékben növeli a kód olvashatóságát és plusz információval látja el a hiba detektáló eszközöket (például Dialyzer). Egy típus specifikálásának a következő a szintaxisa:

```
-type name() :: datatype()
```

A *-type* kulcsszó jelzi egy modulban, hogy típus specifikáció következik. A name a specifikált típus neve, ezzel a névvel lehet hivatkozni a típusra a modulon belül, a modulon kívül (amennyiben exportálásra kerül a típus) a modulnév:típusnév() szintaxissal lehet hivatkozni a specifikált típusra. A datatype a specifikált típus leírása, ami lehet egy beépített adat típus, egy másik specifikált típus, egy atom vagy integer típusú érték (pl.: „foo” vagy 21), vagy ezek tetszőlegesen vett uniója. Példa egy típus specifikációra:

```
-type mytype() :: boolean() | undefined
```

A *mytype* nevezetű típus ennek értelmében olyan adattípust definiál, amely vagy egy *boolean* értéket vesz fel, vagy az *undefined* atom értékét.

2.6.2 Modulok

Egy Erlang alkalmazásban a kód modulokra van osztva. Egy modulban található kódot két fő részre lehet bontani, a modullal kapcsolatos attribútumok deklarálására (például típus specifikációk vagy exportált függvények listája) és függvény deklarációkra. Gyakran előfordul, hogy két modul strukturálisan nagyon hasonlít egymásra, ugyanazokat a mintákat követik. Ilyenek például a felügyeleti modulok, amelyek általában csak abban térnek el, hogy milyen Erlang folyamatokat felügyelnek. Ezeknek a gyakori strukturális mintáknak a formalizálására létre lehet hozni

úgynevezett viselkedés leíró modulokat. Ezáltal szét lehet választani a kódot egy újrahasználható, generikus részre (viselkedés leíró modul) és egy specifikus részre (callback modul). Az Erlang/OTP-ben vannak előre definiált, beépített viselkedések (például gen_server), de van lehetőségünk saját viselkedés leíró modulokat definiálni [7]. Egy ilyen generikus modulban deklarálhatunk olyan függvényeket, amelyeknek az implementációja kötelező azoknak a callback moduloknak, amelyek megvalósítják ezt a viselkedést. Egy callback függvény deklarációja a következőképpen néz ki:

```
-callback FunctionName(Arg1, Arg2, ..., ArgN) -> Res.
```

A FunctionName a callback függvény neve, az ArgX az X-edik argumentum típusa, a Res pedig az eredmény típusa. Lehetőség van opcionális callback metódusok megadására is, ekkor az -optional_callback kulcsszót kell használni. A viselkedést megvalósító modulban, a következő modul attribútummal tudjuk jelezni az Erlang fordítóprogramjának, hogy ez egy callback modul:

```
-behaviour(Behaviour)
```

A Behaviour Erlang atom egy modul neve kell, hogy legyen. Így a fordítóprogram felismeri, hogy ez egy callback modul és jelezni fogja, ha valamelyik callback metódus nem került implementálásra.

2.7 React [8]

A React egy deklaratív, komponens alapú JavaScript könyvtár, amelyet felhasználói felületek létrehozásához készítettek. A kódot a Facebook és egy nyílt forráskódú fejlesztői közösség tartja karban. A könyvtár 2013-ban jelent meg először és mára az egyik legelterjedtebb frontend-könyvtár lett a webfejlesztésben. Főbb jellemzői, az újrahasznosítható komponensek, az egymásba ágyazott komponensek közti egyirányú adatáramlás (a szülő komponenstől a gyerek felé), illetve a virtuális DOM (Data Object Model) használata. A virtuális DOM összehasonlítja a komponensek korábbi állapotát és csak a megváltozott elemeket frissíti a valódi DOM-ban, ezzel növelve az oldal teljesítményét. A komponens könyvtár a népszerűségét többek között a meredek tanulási görbének és a széleskörű felhasználhatóságának köszönheti A React-et gyakran használják valamilyen ráépülő keretrendszerrel (pl. Next.js, Gatsby, CRA). Többek közt alkalmas SPA-k (Single Page Application), mobil alkalmazások vagy szerver oldali renderelést használó weboldalak elkészítéséhez. A diplomamunkám során

a webes vékonykliens alkalmazás felhasználói felületének és üzleti logikájának implementálása során React-et használtam.

3 Kapcsolódó munka

Ebben a fejezetben az elkészült megoldáshoz vezető kapcsolódó munkámat (irodalomkutatás, prototípusok készítése) fogom bemutatni és a felmerülő tervezői döntéseket megindokolni.

3.1 Wayland vagy X11

A natív Linux-on futó adatgyűjtő kliens alkalmazás tervezésekor az egyik alapvető kritérium a minél nagyobb felhasználói csoport támogatása volt. Tekintve a Linux disztribúciók változatos és sokszínű világát ennek a követelménynek korántsem triviális eleget tenni. Ahhoz, hogy minél több disztribúciót (és verziót) támogatni tudjon az alkalmazás, annál kernel-közelebbi szinten kell implementálni a klienst. Valószínűleg nagyban megkönnyítené az implementációt, ha az adatgyűjtő klienst az asztali környezetek szintjén készíteném el és például GNOME-specifikus lenne. Ugyanakkor ezzel a felhasználóknak egy jelentős hányadát kizárnám, például akik KDE-t vagy Xfce-t használnak. A logikus döntés tehát, hogy egy absztrakciós szinttel alacsonyabban, a megjelenítő protokollok szintjén készüljön el az alkalmazás.

A megjelenítő protokollok esetében már nem áll fent a „bőség zavara”, a Wayland protokoll és az X Window System között kell választani. Az X-szet 1984-ben kezdték el fejleszteni és a legújabb nagy verzió kiadása 1987-ben történt (kisebb verzió frissítések jelentek meg, jelenleg a legfrissebb az X11R7.7 2012-ben lett kiadva). Az idők során számos kritika érte az X Window System-et. Leggyakrabban az elavultság, a biztonságtechnikai hiányosságok és a teljesítmény azok a szempontok, amelyek mentén kritikákat fogalmaznak meg a protokollal szemben. Ugyanakkor sok Linux disztribúció a mai napig alapértelmezetten az X.Org-ot használja megjelenítő szervernek és mint opcionális választás szinte mindegyikben megtalálható.

A Wayland protokollt 2008-ban kezdték el fejleszteni (többen az X.org szerver fejlesztő csapatából) és a mai napig aktívan dolgoznak rajta. A projektre „következő generációs” megjelenítő szerverként hivatkoznak és célja, hogy leváltsa az X Window System-et egy modernebb, egyszerűbb és biztonságosabb protokollra. Több nagyobb felhasználóbázissal rendelkező Linux disztribúció (például: Debian, Ubuntu, Fedora) elkezdett átállni a Wayland-re, mint alapértelmezett megjelenítő protokoll (legalábbis a

GNOME asztali környezetet használó verziók). Mivel a Wayland protokoll modernebb és a jövőben minden bizonnyal át fogja venni az X Window System helyét és az adatgyűjtő kliens alkalmazást alapvetően időtálló módon terveztem implementálni, ezért a Wayland tűnt a jó választásnak.

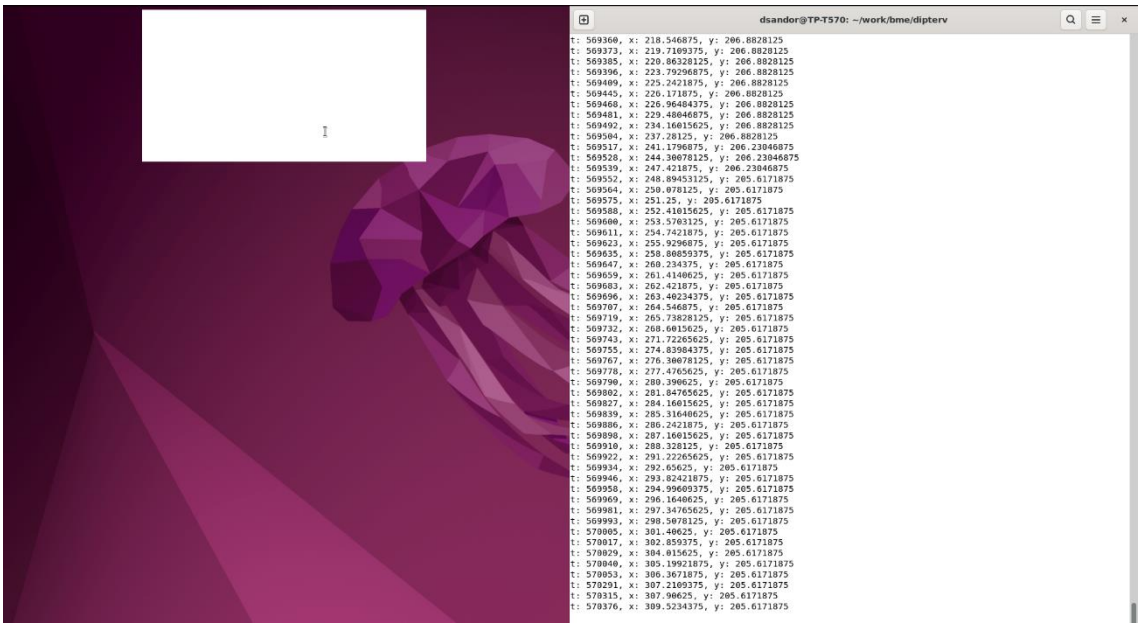
3.1.1 Kutatás és prototípus

Az adatgyűjtő alkalmazást tehát kezdetben egy Wayland kliens alkalmazás formájában próbáltam meg elkészíteni. Első lépésként egy implementációs nyelvet kellett választanom. Ahogy korábban már említettem a Wayland alapvetően egy protokoll, ami XML fájlok formájában van dokumentálva. A protokollhoz tartozik egy szerveroldali referencia implementáció (Weston), illetve egy kliens oldali könyvtár (libwayland), amely C-ben lett implementálva. Kliens oldalon több programozási nyelvhez is készült implementáció (amelyeket túlnyomó részt az XML fájlokból generáltak), illetve a libwayland-hez is elérhető több olyan nyelvi burkoló könyvtár, ami a C-ben implementált függvények hívását teszi lehetővé másik programozási nyelvekből. Mivel a kezdeti célom egy prototípus gyors implementációja volt, ezért a PyWayland-et, egy Python nyelvhez készült burkoló könyvtárat választottam.

A Python-ban implementált prototípus elkészítésével relatíve gyorsan falba ütköztem. Alapvetően azt a konklúziót szűrtem le a prototípus implementálása során, hogy a Wayland protokoll a jelenlegi formájában nem összeegyeztethető az általam elkészíteni kívánt adatgyűjtő alkalmazás követelményeivel. Mielőtt bővebben kifejteném, hogy miért jutottam erre a következtetésre röviden ismertetném a legfontosabb követelményeit az adatgyűjtő kliensnek. Ahogy azt korábban említettem az egyik fontos követelmény a felhasználók minél szélesebb körének támogatása. Ezen kívül a funkcionalitás szempontjából az alkalmazásnak képesnek kell lennie arra, hogy a háttérben, grafikus felhasználó felület nélkül fusson. Továbbá minden egér képességgel rendelkező bemeneti eszköz (egér, touchpad, trackpoint, stb.) által generált kurzor mozgás eseményt fel tudjon dolgozni.

A Wayland protokoll esetén az ablakok az X-hez hasonlóan hierarchikusan helyezkednek el, a tartalmazási fa gyökerében egy speciális ablak található, amit gyökér ablaknak (root window) neveznek. Az egyik központi probléma, amivel találkoztam az volt, hogy a Wayland esetében nincs lehetőség kliens oldalon a gyökér ablak eseményeire feliratkozni. Ennek alapvetően biztonságtechnikai okai vannak, a Wayland

kliensek egymástól izolált környezetben futnak és nem férhetnek hozzá a többi folyamat adataihoz. Egy olyan klienst el tudtam készíteni, ami létrehoz egy alkalmazás ablakot és amikor fókuszba kerül az ablak (a felhasználó az ablak területére mozgatja a kurzort) el kezdi gyűjteni az egér mozgás adatokat.



6. ábra A prototípus kliens működés közben.

Felmerült még ötletként egy teljesképernyős „overlay” alkalmazás ablak készítése, de ez több szempontból sem tűnt jó iránynak. Egyrészt a kurzor mozgás események elkapása csak akkor működik, amikor az ablak fókuszban van, így amikor a felhasználó egy másik ablakra váltana megállna az adatgyűjtés. Erre megoldás lehetne az explicit fókusz kérése (focus grab), de a Wayland esetében erre csak felugró ablak (popup) jellegű, rövid élettartamú ablakok esetében van lehetőség. Továbbá egy ilyen megoldás ellentmondana annak a követelménynek, hogy az alkalmazás a háttérben fusson, grafikus felhasználói felület nélkül.

3.1.2 Konklúzió

A sikertelen próbálkozás után visszatértem a Wayland-el kapcsolatos kutatáshoz és elkezdtem mások által készített alkalmazások forráskódját tanulmányozni. Elsősorban olyan alkalmazásokra koncentráltam, amelyeknél nagy valószínűséggel felmerült az a probléma, amibe én is belefutottam. Többnyire olyan szoftverek forráskódját néztem meg, amelyek képernyő megosztással, távoli asztal elérés (remote desktop) funkcióval vagy egér / billentyűzet emulálással foglalkoznak. Ezenkívül egy

GNOME asztali környezethez készült widgetet is tanulmányoztam, amely kurzor követéssel foglalkozik (xeyes). A kutatás sajnos kiábrándító eredményeket hozott. Általánosságban három különböző típusú működéssel találkoztam az említett szoftvereknél:

1. Az alkalmazás egyáltalán nem támogatja a Wayland protokollt.
2. Az alkalmazás kísérleti jelleggel, részlegesen támogatja a Wayland protokollt, azaz bizonyos funkciók nem elérhetőek a szoftverben Wayland alatt.
3. Az alkalmazás valamilyen megszorítások mellett támogatja a Wayland-et, például csak GNOME asztali környezeten működik.

Lényegében azt a problémát figyeltem meg, hogy az alap Wayland protokoll meglehetősen karcsú és nem biztosít interfészeket olyan feladatokhoz, mint a képernyőfelvétel vagy a bementi eszköz emuláció. Az ilyen esetekben általában a fejlesztők protokoll kiegészítéseket hoznak létre, amelyet az általuk használt kompozitor implementál és ezután már képes a megfelelő interfészeket nyújtani. Ezért áll fent az a helyzet is például, hogy a csak GNOME asztali környezetet támogató szolgáltatások képesek támogatni a Wayland-et, mert a GNOME által használt Wayland kompozitor implementáció (Mutter) implementál bizonyos protokoll kiegészítéseket, amelyek ezt lehetővé teszik. A helyzet viszont még ennél is bonyolultabb, ugyanis különböző Wayland kompozitor implementációk különböző protokollt kiegészítéseket definiálnak, amelyek egymással általában nem kompatibilisek. Tehát ahhoz, hogy egy olyan alkalmazást készítsek, ami minden Wayland-et használó disztribúción működik vagy az alap protokollt kellene csak használnom, vagy a különböző asztali környezetekhez implementálni az általuk nyújtott interfészeket.

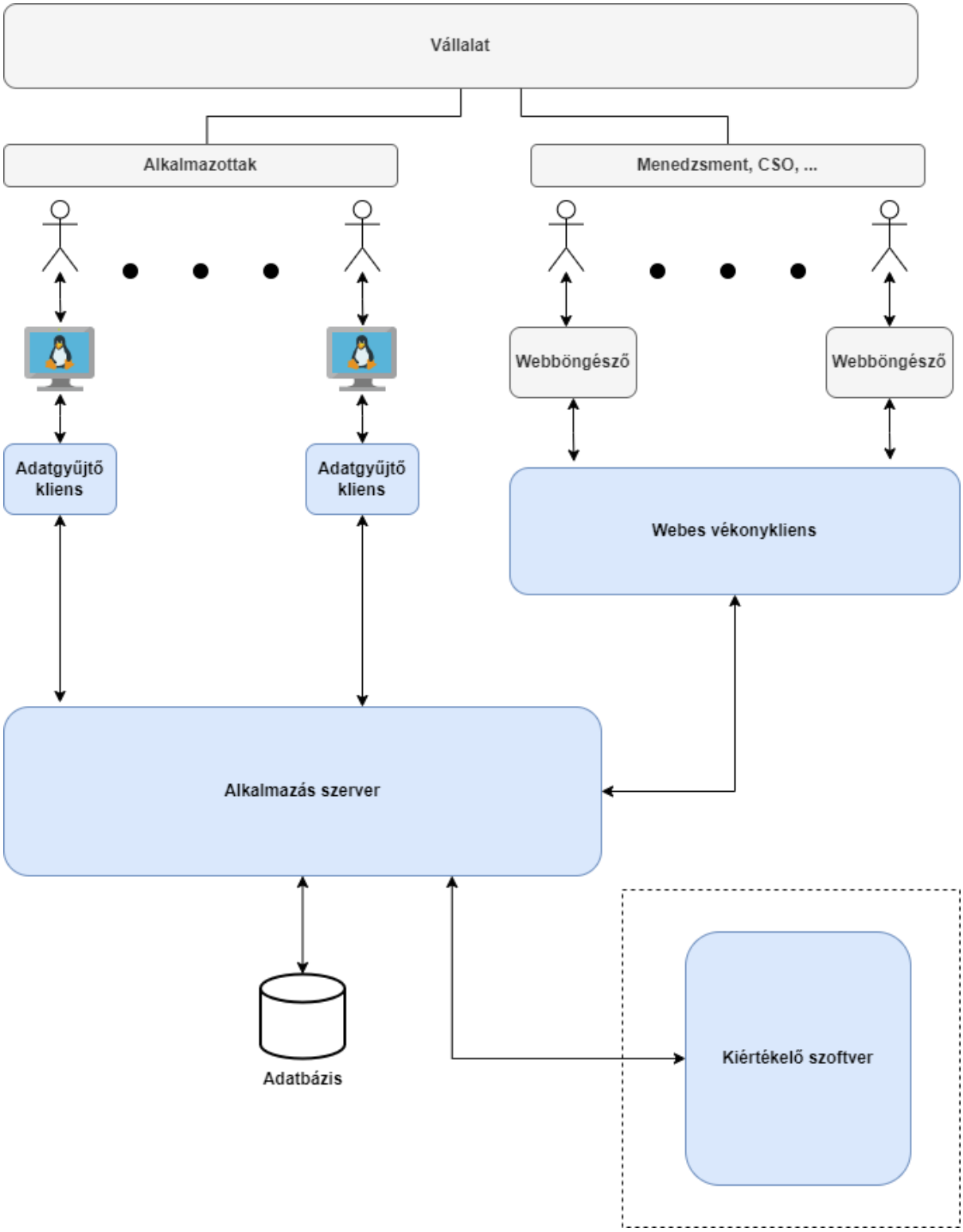
Az alap protokollon belül nincs lehetőség globális kurzor információ lekérdezésére, a különböző protokoll kiegészítések, amelyekkel ezt meg lehetne tenni pedig jelenleg nem állnak még rendelkezésre. Ezért arra jutottam, hogy a Wayland protokoll jelenlegi állapotában nem összeegyeztethető az adatgyűjtő kliens alkalmazás előzetes követelményeivel. A probléma különben nem egyedi, sok alkalmazás esetében megfigyelhető, hogy nehézkes az X-ről való átállás Wayland-re. Én a továbbiakban a kliens alkalmazás implementálása során visszatértem az X Window System protokollhoz.

4 A projekt felépítése

Ebben a fejezetben ismertetem a rendszer architektúráját, a rendszert felépítő alkalmazások kapcsolatait és határait. A fejezet célja egy absztrakt képet nyújtani a projekt felépítéséről, az egyes komponenseket feketedobozként kezelve. A fejezetnek nem célja az alkalmazások részletes belső működésének ismertetése, ezek a leírások a további fejezetekben találhatóak.

4.1 Architektúra

A *7. ábra* A projekt architektúrájaa projekt felépítését egy képzeletbeli vállalat példáján keresztül mutatja be. A vállalatnál dolgoznak alkalmazottak, illetve vannak más / magasabb, pozícióban dolgozó emberek, például menedzsment tagok vagy a biztonsági vezető (CSO, Chief Security Officer). Az alkalmazottak teljes köréből jelenleg csak az a részhalmaz a releváns, akik valamilyen Linux disztribúciót használnak munkavégzéshez. Az alkalmazottak számítógépére kerül telepítésre az **adatgyűjtő kliens** alkalmazás. Ez az alkalmazás a háttérben fut, az alkalmazottak számára transzparens módon (nincs semmilyen teendőjük vele, nem akadályozza őket a munkavégzésben). A vállalat alkalmazottjai a napi teendőik során kurzormozgás adatokat generálnak, amit a kliens alkalmazás továbbít az **alkalmazás szervernek**. A szerveren a beérkező adatok különféle ellenőrzéseken mennek keresztül, ezután a szerver egy **adatbázisba** menti a kurzormozgás adatokat. Miután egy alkalmazotthoz kellő mennyiségű mozgás adat gyűlt össze, a szerver egy hívást indít a **kiértékelő szoftver** felé, amely egy biometrikus profilt épít az adatokból. A kiértékelő szoftver nem része a diplomamunkának, ezt egy külső szolgáltatásként használja az alkalmazás. Amennyiben egy alkalmazott már rendelkezik biometrikus profillal, az újonnan generált kurzormozgás adatokat a kiértékelő szoftver verifikálja (azaz eldönti, hogy mennyire valószínű az, hogy az új mozgás adat az adott felhasználótól származik és nem valaki mástól). Az alkalmazás szerver a verifikáció eredményét továbbítja a kliens alkalmazásoknak, amelyek az adott konfigurációtól függően képesek a munkamenet felfüggesztésére (azaz a felhasználó kizárására), amennyiben a verifikáció eredménye nem megfelelő.

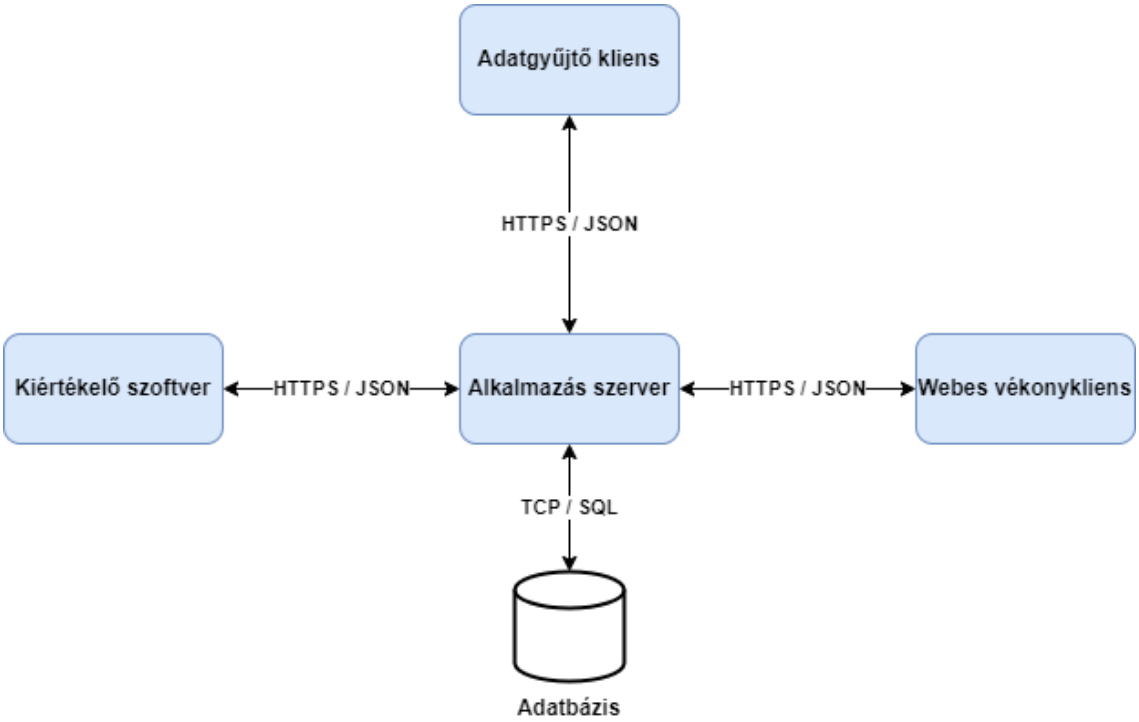


7. ábra A projekt architektúrája

A vállalatnál dolgoznak egyéb, nem alkalmazotti pozícióban dolgozó személyek, akik szeretnének egy képet kapni rendszer működéséről. Ezt a webes vékonykliens alkalmazáson keresztül tehetik meg, amelynek az eléréséhez elegendő egy webböngésző. Itt különféle statisztikákat tudnak elérni a rendszer aktuális állapotával kapcsolatban, mint például a felhasználók adatai, biztonsági incidensek vagy a gyűjtött adat mennyiségének időbeli eloszlása. A vékonykliens alkalmazáson lehetőség van a rendszerrel kapcsolatos statisztikák, illetve felhasználóspecifikus statisztikák böngészésére.

4.2 Komponensek kommunikációja

A *8. ábra* az alkalmazás komponensek egymással való kommunikációja látható. A kliens szoftverek, az adatbázis, illetve a kiértékelő szoftver egymással nem kommunikálnak, csak az alkalmazás szerverrel. A kommunikáció az alkalmazás komponensek között HTTP / HTTPS protokollon keresztül történik. Az egyes HTTP / HTTPS kérések során az adatok JSON fájlformátumban utaznak. Az alkalmazás szerver által nyújtott API (Application Programming Interface) részletes dokumentációja a *Az alkalmazás* szerver olvasható. Az alkalmazás szerver az adatbázissal TCP protokollon keresztül kommunikál, az adatbázis lekérdezések SQL nyelven vannak megfogalmazva.



8. ábra Komponensek kommunikációja

5 Linux kliens alkalmazás

Ennek a fejezetnek a célja a natív Linux kliens kurzormozgás adatgyűjtő és munkamenet záró alkalmazás részletes dokumentációja, a felmerülő tervezői döntések megindoklása.

5.1 Követelmények

Az alkalmazás tervezése során több előzetes követelmény merült fel, amelyek befolyásolták a tervezői döntéseket. Ezek a következők voltak:

R1 Az alkalmazás legyen képes GUI-val rendelkező natív Linux operációs rendszeren kurzormozgás adatot gyűjteni különböző beviteli eszközöktől.

R2 Az alkalmazás el tudja küldeni a gyűjtött adatokat az alkalmazás szervernek.

R3 Az alkalmazás le tudja kérdezni az aktuális felhasználó státuszát az alkalmazásszervertől.

R4 Az alkalmazás meg tudja jeleníteni valamilyen formában a felhasználó státuszát.

R5 Az alkalmazás rendelkezzen munkamenet záró funkcionalitással.

R6 Az alkalmazás képes platform specifikus metaadatok gyűjtésére és azok elküldésére az alkalmazás szervernek.

R7 Az alkalmazásnak felhasználói felület nélkül, a háttérben kell tudnia futni, a felhasználó munkamenetének megzavarása nélkül.

R8 Az alkalmazásnak lehetőség szerint minél nagyobb felhasználói bázist kell tudnia támogatni.

R9 Az alkalmazás könnyen konfigurálható legyen környezeti változók és / vagy parancssori argumentumok használatával.

5.2 Az alkalmazás tervezése

A *R8*-as követelmény értelmében, a *3.1*-es fejezetben kifejtettek szerint a kliens alkalmazás architektúrálisan a megjelenítő protokollok szintjén készült el, azon belül is az X Window System protokollt támogató rendszerekre. Implementációs nyelvként a Rust-ot választottam, ennek az indoklása a *2.5*-ös fejezetben olvasható. Ahhoz, hogy a

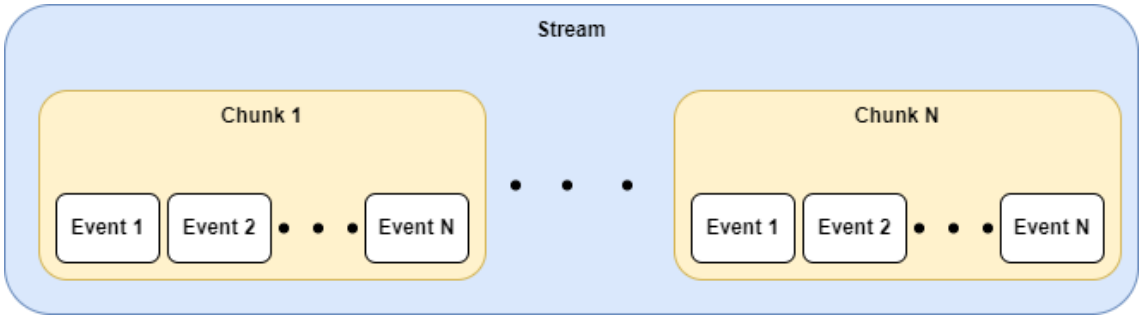
felhasználó által generált kurzormozgás eseményeket a kliens program el tudja kapni szükséges a felhasználó számítógépén futó megjelenítő szerverrel történő kommunikáció. Ezt a program a megjelenítő protokollon keresztül tudja megtenni. A Rust nyelvhez elérhető több X11 implementáció is. Én a **x11rb** nevű modult használtam, amelynek a kódját a protokoll XML leírása alapján generálták (azaz nem egy már meglévő, például C-ben írt implementációhoz ad burkoló függvényeket).

Az alkalmazást alapvetően többszálúra terveztem, a program indulásakor két szálat indít. Az egyik az *R1*, *R2*, *R6* követelmények által leírt funkcionalitás valósítja meg, azaz a kurzormozgás adatok és platformspecifikus metaadatok gyűjtését és továbbítását az alkalmazás szerverre. A másik szál az *R3*, *R4*, *R5* követelményeket elégíti ki, azaz a felhasználó státuszának lekérdezését és megjelenítését, illetve, ha szükséges, akkor a munkamenet zárolását. A kliens alkalmazás build-elési folyamatának eredménye egy futtatható állomány, amelyet egy Linux parancssorból el lehet indítani leválasztott (detached) módban, ezzel kielégítve az *R7*-es követelményt. Az alkalmazást környezeti változókkal és parancssori argumentumokkal egyaránt lehet konfigurálni (*R9*).

5.3 Adatgyűjtő

A kliens alkalmazás egyik alapvető funkcionális követelménye a felhasználó által kreált kurzormozgás adatok, illetve platform specifikus metaadatok gyűjtése és elküldése az alkalmazás szerverre. Az alkalmazásnak ezt a funkcionalitását egy különálló Rust modulban készítettem el.

Az adatgyűjtő egy eseményfolyam szerű viselkedést implementál (data streaming). A program az indulásakor generál egy egyedi azonosítót, amit az adatfolyam azonosítására használ. A küldött adat legkisebb atomi építőeleme az esemény (event). Ez lehet a felhasználó által kreált esemény (például az egér mozgatása, kattintás, görgetés, stb.) vagy más egyedi esemény (például platformspecifikus metaadatok változása). Mivel az adatgyűjtő az alkalmazás szerverrel HTTPS protokollon keresztül kommunikál, ezért nem lehet (és nem is lenne praktikus) tetszőleges mennyiségű eseményt egy üzenetben elküldeni. Ezért a kliens az eseményeket nagyobb csoportokban (chunk) küldi el a szervernek. Ezeknek az entitásoknak a kapcsolatáról nyújt egy áttekintő képet a *9. ábra*.



9. ábra Az adatfolyam felépítése

A modul tartalmaz struktúrát, amelyben az adatgyűjtő állapota kerül eltárolásra, illetve egy eseményhurkot, amelyben a nyers események kerülnek feldolgozásra. A következő kód részletben a struktúra definíciója látható:

```
struct State {
    buffer: Vec<EventType>,
    buffer_size_limit: usize,
    api_key_name: String,
    api_key_value: String,
    submit_url: String,
    epoch: u64,
    session_id: String,
    stream_id: String,
    sequence_number: u64,
    user_id: String,
}
```

A struktúra egyes mezőinek jelentése a következő:

- **buffer:** Esemény puffer, ideiglenesen ebben az adatstruktúrában kerülnek eltárolásra a feldolgozott események, mielőtt a kliens elküldi az alkalmazás szervernek.
- **buffer_size_limit:** Felső korlát az esemény puffer méretére. Amennyibben a puffer mérete eléri ezt a korlátot a kliens elküldi az eseményeket.
- **api_key_name:** Egyedi API kulcs neve, amit a kliens hozzáfűz a HTTP kérések fejlécéhez. Ennek az a szerepe, hogy az alkalmazás szerveren azonosítsa a klienst.
- **api_key_value:** Egyedi API kulcs értéke.
- **submit_url:** Az alkalmazás szerveren található végpont elérhetősége, amelyre az adatokat küldeni kell.
- **epoch:** Unix időbélyeg, az adatgyűjtő kliens indulásának pillanatában kerül rögzítésre.

- **session_id:** A felhasználó munkamenetének azonosítója. Unix-szerű operációs rendszereken ez megfelel a *who* parancs kimenetének, ami tartalmazza a bejelentkezett felhasználó azonosítóját, illetve a munkamenet kezdetének idejét.
- **stream_id:** Az adatfolyam egyedi azonosítója. A kliens az indulásakor generálja.
- **sequence_number:** Szigorúan monoton növekvő egész szám, amellyel az adatcsomagok (chunk) vannak ellátva. Célja, hogy az adatfolyam reprodukálható legyen szerver oldalon.
- **user_id:** A felhasználó egyedi azonosítója.

A feldolgozott események tehát egy ideiglenes pufferben kerülnek eltárolásra. Amikor ez a puffer megtelik, vagy egy (konfigurálható) időtartamig nem érkezik új esemény a kliens elküldi a puffer tartalmát az alkalmazás szerverre, majd üríti a puffer tartalmát. A különböző eseményekre a kliens egy nem blokkoló eseményhurokban várakozik. Ennek az implementálására a Rust szálak közti kommunikációt támogató *mspc* (multi-producer, single-consumer) modulját használtam.

A felhasználó által kreált nyers X11 események egy külön szálon kerülnek feldolgozásra, majd a program továbbítja a feldolgozott eseményeket a fő eseményhuroknak. Ez a szál először kialakítja a kapcsolatot az megjelenítő szerverrel, majd kér egy referenciát a tartalmazási fa gyökerében található ablakra (root window). Ezután lekérdezi a megjelenítő szervertől azokat a beviteli eszközöket, amelyek rendelkeznek kurzor mozgatási képességgel (például USB egér, touchpad, érintőképernyő, stb). Ezt követően kliens a beviteli eszközök azonosítójával különböző esemény maszkokat regisztrál a *root window* elemre. Innentől kezdve, ha valamelyik beviteli eszköz kibocsájt egy regisztrált esemény típust, akkor azt a megjelenítő szerver a kliens program felé is hirdetni fogja. Ezután a szál egy hurokban nem blokkoló várakozásba kezd a nyers X eseményekre. Amennyiben érkezik egy új esemény, a program kinyeri belőle a releváns információkat (például koordináták, időbélyeg, stb) és a feldolgozott eseményt továbbítja a fő eseményhuroknak. A jelenleg támogatott X események listája a következő:

- **MOTION_EVENT_TYPE:** Kurzor mozgás esemény.
- **SCROLL_EVENT_TYPE:** Görgetés esemény.
- **TOUCH_BEGIN_EVENT_TYPE:** Érintés kezdete esemény.
- **TOUCH_UPDATE_EVENT_TYPE:** Érintés frissítése esemény.
- **TOUCH_END_EVENT_TYPE:** Érintés vége esemény.
- **BUTTON_PRESS_EVENT_TYPE:** Egérgomb lenyomása esemény.
- **BUTTON_RELEASE_EVENT_TYPE:** Egérgomb felengedése esemény.

A feldolgozott események ezután először az esemény pufferbe kerülnek, majd továbbításra az alkalmazás szerverre. Egy feldolgozott kurzor mozgás esemény a következőképpen néz ki:

```
MOTION_EVENT_TYPE: {
  types: [
    'type:type',
    't:timestamp:ms',
    'xIntegral:integer',
    'xFraction:integer',
    'yIntegral:integer',
    'yFraction:integer',
    'rootX:integer',
    'rootY:integer',
  ],
}
```

A kurzor mozgás esemény egyes mezőinek típusa és magyarázata:

- **type:** Az esemény típusa, kurzor mozgás esemény esetén az értéke 0.
- **timestamp:** Unix időbélyeg, a mértékegysége milliszekundum.
- **xIntegral:** A kurzor X axis menti elmozdulásának egész része.
- **xFraction:** A kurzor X axis menti elmozdulásának tört része.
- **yIntegral:** A kurzor Y axis menti elmozdulásának egész része.
- **yFraction:** A kurzor Y axis menti elmozdulásának tört része.
- **rootX:** A kurzor X koordinátája a gyökér ablakban.
- **rootY:** A kurzor Y koordinátája a gyökér ablakban.

A teljes esemény leíró séma megtalálható a függelékben, a *12.1*-es fejezetben.

5.3.1 Platform specifikus metaadatok

Az *R6*-os követelmény értelmében az alkalmazásnak különböző platform specifikus metaadatot kell tudnia gyűjteni és ezt elküldeni az alkalmazás szerverre. Ezt a viselkedést az adatgyűjtő részeként valósítottam meg egy külön modulban. Bevezettem egy új eseménytípust **METADATA_CHANGED_EVENT** néven, amely a metaadatok eseményfolyamba való injektálására használtam. A kliens egy külön szálon bizonyos (konfigurálható) időközönkét összegyűjt különböző adatokat a felhasználó által használt platformról és ezeket elküldi a fő eseményhuroknak. Ezáltal a metaadatok bekerülnek az ideiglenes esemény pufferbe, majd idővel a kliens elküldi őket az alkalmazás szerverre.

A kliens által összegyűjtött különböző adatokat struktúrába szerveztem, ugyanis a Rust *derive* mechanizmusán keresztül a struktúra egyszerűen szerializálható a struktúra JSON formátumba. A *serde* nevű külső modul segítségével pedig még a más nyelvekben nehézkes snake_case → camelCase konverzió is egy egyszerű annotációval megoldható. A metaadat struktúra definíció alább látható:

```
#[derive(Clone, Debug, Serialize)]
#[serde(rename_all = "camelCase")]
pub struct Metadata {
    user_name: String,
    host_id: String,
    monitor: Vec<MonitorMetadata>,
    input_device: String,
    os: os_info::Info,
}
```

Az egyes mezők jelentése:

- **user_name:** A jelenleg bejelentkezett felhasználó neve. Unix-szerű rendszereken ez a *USERNAME* környezeti változó értékében van tárolva, ezt használja a kliens program is.
- **host_id:** A felhasználó által használt számítógép egyedi azonosítója. A kliens program a */etc/machine-id* file tartalmát használja erre a célra.
- **monitor:** A jelenleg használt monitorok metaadatainak listája. A kliens program ezeket az adatokat a megjelenítő szervertől kérdezi le. A *MonitorMetadata* struktúrát a monitorok adatainak csoportosítására definiáltam. A következő értékek szerepelnek benne:

- **name:** A monitor azonosítója, egy nemnegatív egész szám.
- **primary:** *Bool* típusú, azt adja meg, hogy az adott monitor az elsődleges monitor-e.
- **x:** A monitor bal szélének X koordinátája abban a koordináta rendszerben, ahol a legbalrább található monitor bal széle az origó, egy egység pedig egy pixelnek felel meg.
- **y:** A monitor tetejének Y koordinátája abban a koordináta rendszerben, ahol a legfelső monitor teteje az origó, egy egység pedig egy pixelnek felel meg.
- **width:** A monitor szélessége pixelben.
- **height:** A monitor magassága pixelben.
- **width_in_millimeters:** A monitor szélessége milliméterben.
- **height_in_millimeters:** A monitor magassága milliméterben.
- **dpi:** A monitor DPI (Dots Per Inch) értéke.

- **input_device:** Azon beviteli eszközök listája, amelyek tudnak képesek kurzorként viselkedni (pointer devices). A kliens program ehhez az információhoz először kiolvassa a */proc/bus/input/devices* fájl tartalmát, majd szűri azt a releváns eszközökre.
- **os:** A felhasználó operációs rendszerével kapcsolatos metaadatok. Ennek a lekérdezésére egy külső modult, az *os_info*-t használtam. Ez az adat tartalmazza az operációs rendszer típusát, verzióját és *bitness* értékét (azaz, hogy 32 vagy 64 bites).

5.4 A felhasználó státusza

Az *R3, R4, R5* követelmények értelmében az alkalmazásnak le kell tudnia kérdezni az alkalmazás szerverről a felhasználó aktuális státuszát, megjeleníteni azt, illetve, ha szükséges zárolnia kell a felhasználó munkamenetét. Ezeket a funkcionalitásokat az adatgyűjtőtől elválasztva egy különálló modulban (és szálon) implementáltam.

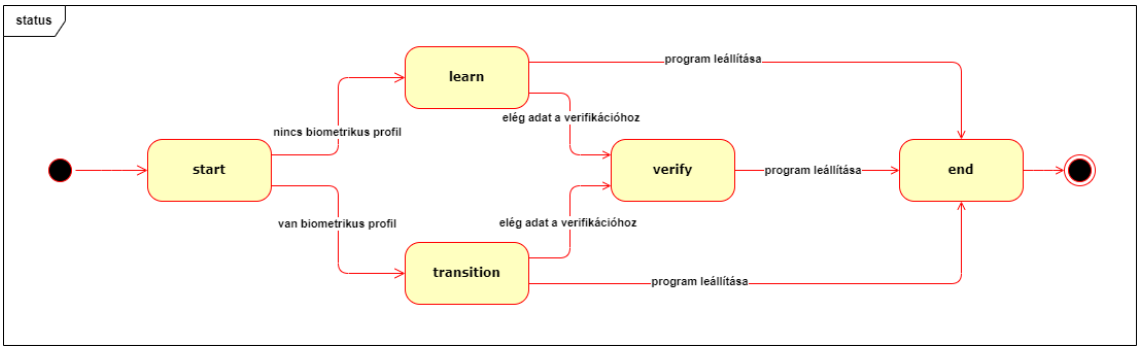
A státusz lekérdezése a kliens program indulása után periodikusan történik. A státusz lekérdezések közti idő intervallum konfigurálható. A felhasználó státusza a következőképpen épül fel:

```
struct Status {
    phase: String,
    description: String,
    value: f64,
}
```

Az egyes mezők jelentése és lehetséges értékei:

- **phase:** Azt adja meg, hogy a felhasználó milyen fázisban van jelenleg. A mező lehetséges értékei a következők:
    - **learn:** Tanuló fázis. A kliens program nem gyűjtött meg elegendő adattot ahhoz, hogy a kiértékelő szoftver a felhasználónak biometrikus profilt tudjon építeni.
    - **verify:** Verifikációs fázis. A felhasználó már rendelkezik biometrikus profillal, így lehetséges a felhasználó verifikációja. Az újonnan érkező mozgásadatot verifikálja a szerver.
    - **transition:** Átmeneti fázis. A felhasználó már rendelkezik biometrikus profillal, viszont az aktuális adatfolyamban nincs még elég adat a verifikációhoz.
- **description:** Opcionális kiegészítő leírás, amit a szerver küldhet a kliensnek.
- **value:** Egy 0.0 és 1.0 közötti lebegőpontos érték. Ez a mező a különböző fázisok esetén mást és mást jelent:
    - **learn:** A felhasználó által gyűjtött adat mennyiségének százalékos értéke, ahol a 0.0 érték a 0 események felel meg, az 1.0 érték pedig annak az adatmennyiségnek, amennyire a kiértékelő szoftvernek szüksége van arra, hogy biometrikus profilt építsen.
    - **verify:** A felhasználó legfrissebb verifikációjának eredménye. Ez az érték azt adja meg, hogy a kiértékelő szoftver mennyire tartja valószínűnek azt a beérkező mozgás adatok alapján, hogy a felhasználó valóban az, akinek mondja magát.

- **transition:** A felhasználó által gyűjtött adat mennyiségének százalékos értéke, ahol a 0.0 érték a 0 események felel meg, az 1.0 érték pedig annak az adatmennyiségnek, amennyire a kiértékelő szoftvernek szüksége van arra, hogy verifikáljon.



10. ábra A felhasználó státusza

A státusz megjelenítésére egy *notify-rust* nevezetű külső modult használtam. Ezt a könyvtárat Linux / BSD alapú asztali környezetekhez írták, amelyek követik az XDG specifikációt. Ezek közé tartozik többek közt a GNOME, a KDE és az Xfce. A modul segítségével DBUS alapú értesítéseket lehet küldeni az adott asztali környezet alapértelmezett értesítés kezelőjének, ami megjeleníti ezeket.

Amennyiben a felhasználó verifikációs fázisban van a státusz *value* mezője a verifikáció eredményét tartalmazza. Ehhez tartozik kliens oldalon egy küszöbérték, ami alatt a felhasználó verifikációja sikertelennek minősül. Amennyiben a kliens programban konfigurálva van a munkamenet zároló funkció, és a küszöbértéken aluli verifikációs értéket kap a szervertől a kliens meghív egy külső programot. Ez a külső program végzi el a munkamenet zárolását. Alapértelmezetten az *slock* [9] nevű program van beállítva erre a célra. Ez egy meglehetősen egyszerű és biztonságos képernyőzároló segédprogram X11-et használó operációs rendszerekhez. A konfigurálható külső program hívás azért is kedvező, mert előfordulhat, hogy bizonyos vállalatok a munkamenet zároló funkció helyett például egy csendes riasztás (silent alert) szerű viselkedést preferálnak.

5.5 Konfigurációs lehetőségek

Az *R9*-es követelmény értelmében a kliens alkalmazásnak könnyen konfigurálhatónak kell lennie. Az alkalmazás konfigurációjával kapcsolatos kódokat

kiszerveztem egy különálló modulba. Az alkalmazás három szinten konfigurálható, ahol a következő szinten definiált változó mindig felülírja az előzőt:

1. Forráskódban definiált („beégetett”) alapértelmezett értékek.
2. Futási időben definiált környezeti változók.
3. A program indításakor definiált parancssori argumentumok.

A környezeti változók feldolgozására a Rust beépített mechanizmusát használtam, a parancssori argumentumok kezelésére pedig a *clap* nevű külső modult. A modul segítségével a parancssori argumentumok egyszerű kezelése mellett lehetőség van igényes és esztétikus használati útmutató generálására is a kódban írt kommentekből, amit a *--help* parancssori argumentummal lehet megjeleníteni. A kliens alkalmazás a következő konfigurációs beállításokkal rendelkezik:

- **APP_API_KEY_NAME:** Egyedi API kulcs neve, amit a kliens hozzáfűz a HTTP kérések fejlécéhez. Ennek az a szerepe, hogy az alkalmazás szerveren azonosítsa a klienst.

  Alapértelmezett értéke: **„api-key”**.

- **APP_API_KEY_VALUE:** Egyedi API kulcs értéke, amit a kliens hozzáfűz a HTTP kérések fejlécéhez. Ennek az a szerepe, hogy az alkalmazás szerveren azonosítsa a klienst.

  Alapértelmezett értéke: **„x11-sentinel-client”**.

- **APP_BUFFER_SIZE_LIMIT:** Felső korlát az adatgyűjtő esemény pufferének méretére. Amennyibben a puffer mérete eléri ezt a korlátot a kliens elküldi az eseményeket az alkalmazás szerverre.

  Alapértelmezett értéke: **100**

- **APP_IDLE_TIMEOUT:** Amennyiben az adatgyűjtőhöz nem érkezik új esemény ennyi milliszekundumig, a kliens elküldi az esemény pufferben tárolt adatokat az alkalmazás szerverre.

  Alapértelmezett értéke: **10000**

- **APP_LOCK_ENABLED:** Ez a konfigurációs beállítás azt szabályozza, hogy a munkamenet zároló funkció be van-e kapcsolva.

Alapértelmezett értéke: **false**

- **APP_LOCK_THRESHOLD:** A munkamenet zároló funkció esetén használt küszöbérték.

  Alapértelmezett értéke: **0.5**

- **APP_LOCK_UTILITY:** A munkamenet zárolás esetén hívott külső segédprogram.

  Alapértelmezett értéke: **„slock”**

- **APP_METADATA_QUERY_INTERVAL:** A platform specifikus metaadatok lekérdezése periodikusan történik. Ez a konfigurációs paraméter azt adja meg, hogy két lekérdezés között mennyi idő teljen el milliszekundumban.

  Alapértelmezett értéke: **600000**

- **APP_STATUS_BASE_URL:** Az alkalmazás szerver státusz lekérdezési végpontjának URL-je.

  Alapértelmezett értéke: **„http://localhost:3000/status”**

- **APP_STATUS_INTERVAL:** A felhasználó státuszának lekérdezése periodikusan történik. Ez a konfigurációs paraméter azt adja meg, hogy két lekérdezés között mennyi idő teljen el másodpercben.

  Alapértelmezett értéke: **100**

- **APP_SUBMIT_URL:** Az alkalmazás szerver adatküldő végpontjának URL-je.

  Alapértelmezett értéke: **„http://localhost:3000/chunk”**

- **APP_USER_ID:** A felhasználó azonosítója.

  Alapértelmezett értéke: **„default_user”**

6 Az alkalmazás szerver

A következő fejezetnek a célja a kliens programokat kiszolgáló alkalmazás szerver részletes dokumentációja. A leírásban először ismertetem a szerver program felépítését. Ezután az adatbázis réteggel kapcsolatos tudnivalókat, majd a kliens programok felé nyújtott API-kat. Ezt követően az üzleti logikai rétegről és a kiértékelő szoftverről lesz szó. A fejezetet végül a szerver konfigurációs lehetőségeinek bemutatásával zárom.

6.1 A szerver felépítése

Az alkalmazás szervert Erlang nyelven implementáltam. Ezt a döntést a platform *2.6*-os fejezetben bemutatott előnyei, illetve személyes tapasztalatom indokolják. A szerver forráskódja Erlang modulok összessége, amelyek különböző funkcionalitást valósítanak meg. Az egyes modulok jól csoportosíthatóak a betöltött funkciójuk szerint:

- **alkalmazással kapcsolatos modulok**

  Ezek a modulok az alkalmazás elindításával, telepítésével, üzemeltetésével és egyéb feladatokkal kapcsolatos kódokat tartalmaznak (például függőségek kezelése, felügyeleti fa, REST végpontok regisztrálása, stb.).

- **modellek**

  A modellek egy objektum orientált nyelv esetén leginkább egy osztálynak felelnek meg. Az alkalmazásban előforduló entitásokat írják le az Erlang nyelvben megtalálható struktúrákkal, adat típusokkal. Ezen kívül tartalmaznak *getter / setter* függvényeket a modell elemek egyes mezőihez illetve típus definíciókat.

- **szerializációs / deszerializációs modulok**

  Ezek a modulok az Erlang modellek és az adatbázisban tárolt elemek közti transzformációt végzik el. Az alkalmazás egyéb részei számára egy API-t nyújtanak, amelyen keresztül transzparens módon lehet elemeket az adatbázisból lekérdezni vagy módosítani.

- **REST API-t kezelő modulok**

A kliens programok felé nyújtanak REST végpontokat. A beérkező HTTP kéréseket kezelik, azokon különböző ellenőrzéseket futtatnak (például fejlécek, tartalom hosszának ellenőrzése), majd az üzleti logika szerint kiszolgálják az egyes kéréseket.

- **adatbázis működésével kapcsolatos modulok**

Az adatbázis kapcsolat felépítését, a tranzakciók futtatását kezelik.

- **egyéb üzleti logikát megvalósító modulok**

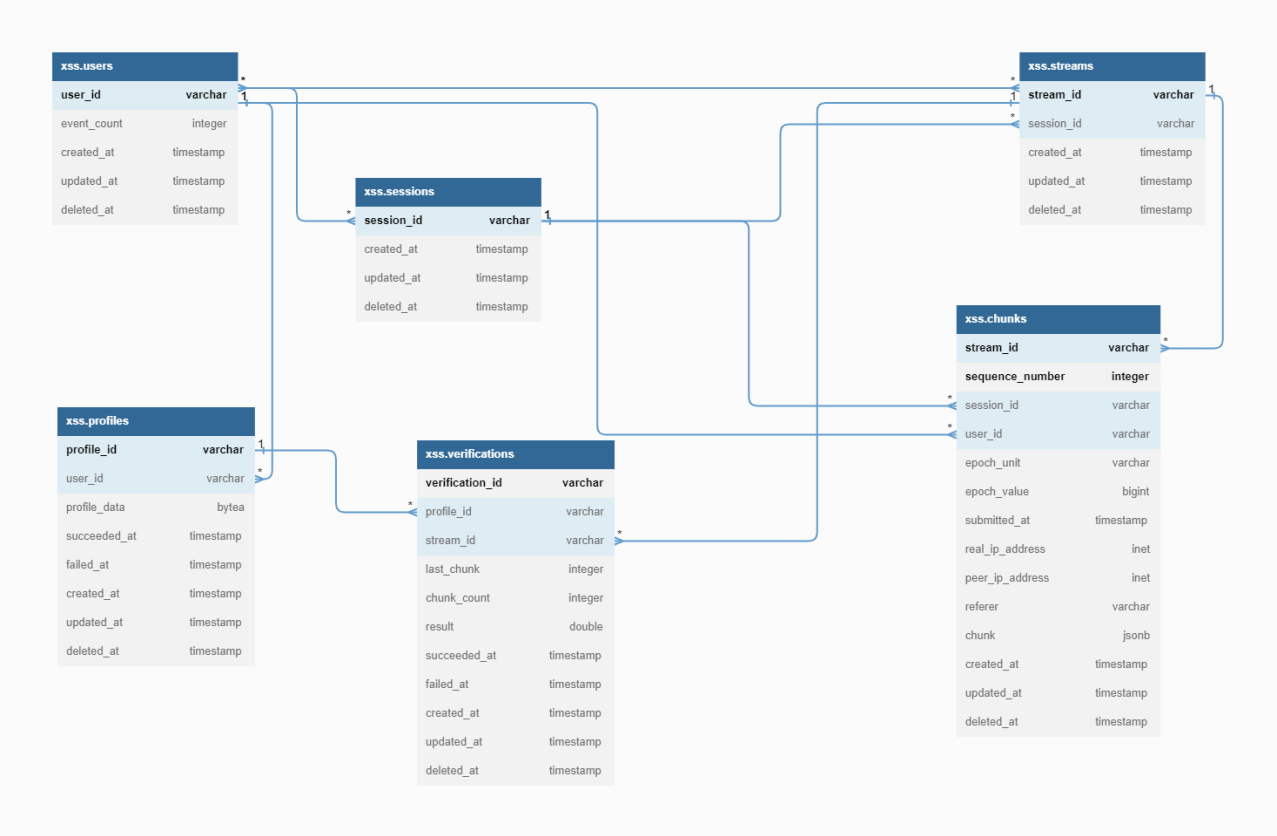
Ezek a modulok további, az előző csoportokba nem tartozó funkcionalitásokat valósítanak meg. Ide tartoznak a segédmodulok, a biometrikus profil építését és verifikációt vezérlő kódrészletek, illetve a kiértékelő szoftverrel való kommunikáció.

6.2 Adatbázis

Az alkalmazás szerver mellé egy relációs adatbázist, a PostgreSQL [10] nevű adatbázist választottam. Az adatbázist az alkalmazás szervertől elválasztva egy külön Docker konténerbe telepítettem, a szerverrel TCP protokollon keresztül kommunikál. Az Erlang kódból történő kommunikációt az *epgsql* nevű külső adatbázis kliens modul segítségével hajtottam végre. Az adatbázis réteg lazán van csatolva, az adatbázis cseréje viszonylag kevés munkával járna.

6.2.1 Séma

Az adatbázis sémája a *11. ábra* látható. Az ábrán az egy - sok kapcsolat az implementációban külső kulcs kényszert jelent, a sok - sok kapcsolat pedig kapcsolótáblát. Az áttekinthetőség kedvéért a kapcsolótáblákat nem ábrázoltam a diagrammon.



11. ábra Az adatbázis séma

Az egyes adatbázis táblák, a táblák oszlopai és azok magyarázata a következő:

- **users:** A felhasználókat tartalmazó tábla.
  - **user_id:** A felhasználó egyedi azonosítója (például e-mail cím).
  - **event_count:** A felhasználó eseményeinek a száma.
- **sessions:** A munkameneteket tartalmazó tábla.
  - **session_id:** A munkamenet egyedi azonosítója.
- **users_sessions:** Kapcsoló tábla a felhasználók és a munkamenetek egymáshoz rendelésére.
  - **user_id:** A felhasználó egyedi azonosítója.
  - **session_id:** A munkamenet egyedi azonosítója.
- **streams:** Az adatfolyamokat tartalmazó tábla.
  - **stream_id:** Az adatfolyam egyedi azonosítója, amelyet a kliens program generál az alkalmazás indulásakor.

    - **session_id:** A munkamenet egyedi azonosítója, amihez az adatfolyam tartozik.
- **users_streams:** Kapcsoló tábla a felhasználók és az adatfolyamok egymáshoz rendelésére.
    - **user_id:** A felhasználó egyedi azonosítója.
    - **stream_id:** Az adatfolyam egyedi azonosítója.
- **chunks:** Az események egy nagyobb csoportját tartalmazó tábla.
    - **stream_id:** Az adatfolyam egyedi azonosítója, amelyhez az események tartoznak.
    - **sequence_number:** Szigorúan monoton növekvő egész szám, amellyel az adatcsomagok vannak ellátva.
    - **session_id:** A munkamenet egyedi azonosítója, amihez az események tartozik.
    - **user_id:** A felhasználó egyedi azonosítója, akihez az események tartoznak.
    - **epoch_unit:** A kliens program indulásakor rögzített Unix időbélyeg mértékegysége, a kliens program tölti ki.
    - **epoch_value:** A kliens program indulásakor rögzített Unix időbélyeg értéke, a kliens program tölti ki.
    - **submitted_at:** Az adatcsomag szerverre történő érkezésének időbélyege (Unix időbélyeg mikroszekundumban).
    - **real_ip_address:** Az adatcsomagot küldő IP címe (X-forwarded-for HTTP fejléc értéke).
    - **peer_ip_address:** Az adatcsomagot továbbító utolsó eszköz IP címe.
    - **referer:** Az adatcsomagot küldő Referer HTTP fejléc értéke.
    - **chunk:** Az adatcsomagban található események. Az adatbázisban bináris JSON (BSON) formátumban kerülnek eltárolásra.
- **profiles:** A biometrikus profilokat tartalmazó adatbázis tábla.
    - **profile_id:** A biometrikus profil egyedi azonosítója.

- **user_id:** A felhasználó egyedi azonosítója, akihez a profil tartozik.
- **profile_data:** A biometrikus profil adattartalma. Az adatbázisban bináris sztringként kerül eltárolásra.
- **succeeded_at:** A sikeres profil készítésének időpontja, amennyiben a profil építés sikeres volt.
- **failed_at:** A sikertelen profil készítésének időpontja, amennyiben a profil építés sikertelen volt.

- **verifications:** A felhasználói verifikációkat tartalmazó adatbázis tábla.
  - **verification_id:** A verifikáció egyedi azonosítója.
  - **profile_id:** A biometrikus profil egyedi azonosítója, amellyel a verifikáció történt.
  - **stream_id:** Az adatfolyam egyedi azonosítója, amelyből azok az események származnak, amelyek fel lettek használva a verifikációhoz.
  - **last_chunk:** A verifikációhoz tartozó adatfolyamban az utolsó olyan adatcsomag *sequence_number* értéke, amelynek az eseményei fel lettek használva a verifikáció során.
  - **chunk_count:** Az adatcsomagok mennyiségének száma, amelyek fel lettek használva a verifikáció során. Ebből az értékből, illetve a **last_chunk** és **stream_id** mezők értékéből pontosan visszaállíthatóak azok az események, amelyeket a verifikáció során használt a kiértékelő szoftver.
  - **result:** A verifikáció eredménye, egy lebegő pontos szám, amelynek az értéke 0.0 és 1.0 között van.
  - **succeeded_at:** A sikeres verifikáció készítésének időpontja, amennyiben a verifikáció futtatása sikeres volt.
  - **failed_at:** A sikertelen verifikáció készítésének időpontja, amennyiben a verifikáció futtatása sikertelen volt.

Minden adatbázis táblához tartozik még három oszlop, amelyek a következőek:

- **created_at:** Az adott rekord az adatbázisba történő beszúrásának ideje.

- **updated_at:** Az adott rekord az adatbázis történő utolsó frissítésének ideje.
- **deleted_at:** Az adott rekord adatbázisból való „törlésének” ideje. Rendes törlés helyett az alkalmazás gyenge törlést (soft delete) használ. Azaz, amikor egy rekordot törölni kell az adatbázisból, a szerver tényleges törlés helyett beállítja a **deleted_at** mező értékét és a többi lekérdezés az ilyen rekordokat figyelmen kívül hagyja.

6.2.2 Telepítés és lekérdezések

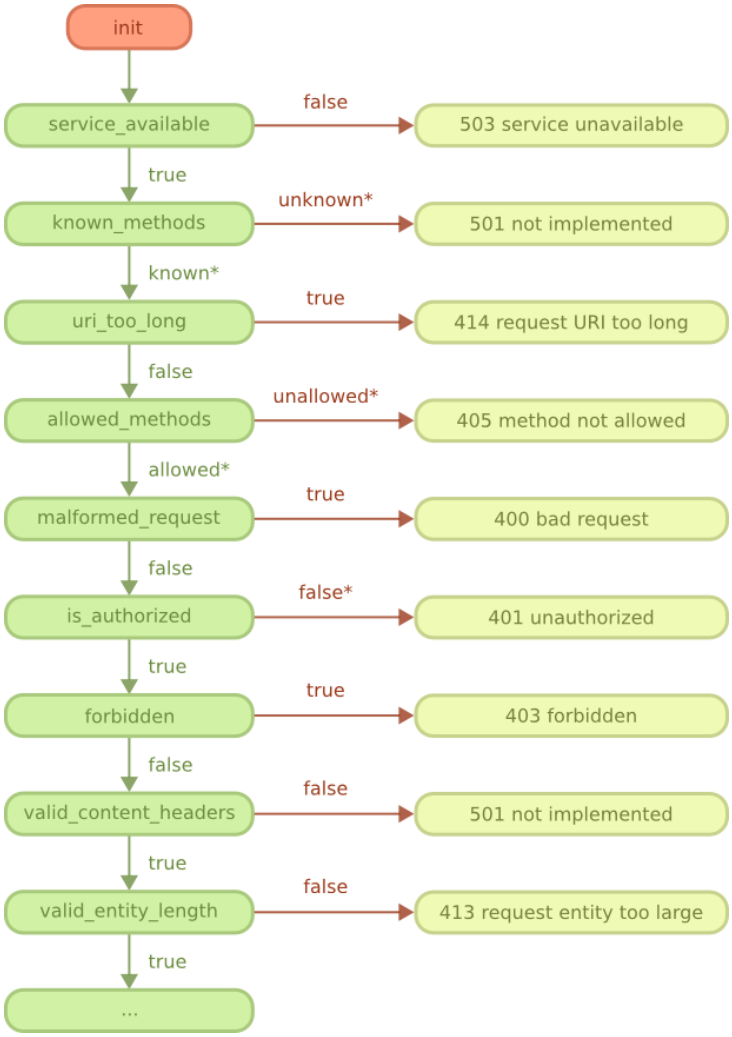
Az adatbázis kapcsolat kiépítését és a tranzakciók futtatását egy általános szerver (gen_server) modulban implementáltam. Ez a komponens egy interfészt nyújt az alkalmazás szerver egyéb részeinek, amelyen keresztül adatbázis lekérdezéseket lehet futtatni. A modul az alkalmazás szerver indulását követően a külső adatbázis kliensen (epgsql) keresztül létrehozza a kapcsolatot az adatbázissal. Az adatbázis kapcsolathoz tartozó referenciát a modul a belső állapotában tárolja. Az adatbázis táblák és a különböző lekérdezések létrehozásához kezdetben egy ORM (Object-Relational Mapping) szoftvert szerettem volna használni. Ezt végül elvetettem, ugyanis egyrészt az ORM szoftverek használata sokszor egy extra komplexitást vezet be a rendszerbe. Másrészt (ami nyomósabb indok volt) a jelenleg elérhető Erlang nyelvhez készült ORM szoftverek nem bizonyultak kellőképpen naprakésznek és testreszabhatónak. Ezután úgy döntöttem, hogy a két feladatot külön kezelem. Az adatbázis séma és a táblák létrehozására SQL nyelven írtam szkripteket, amelyeket közvetlenül az adatbázison egy kliens programmal (például psql [11]) lehet lefuttatni. A lekérdezéseket szintén SQL nyelven írtam meg, viszont a feldolgozásukra használtam egy külső szoftvert (eql [12]), amely az SQL fájlban található kommentek alapján leképezi a lekérdezéseket Erlang függvényekre. Így sokkal kényelmesebben lehet Erlang kódból adatbázis lekérdezéseket futtatni. Egy példa az ekképpen megírt lekérdezésekre:

```
-- :select_events_by_user_id_aggregated_by_day
SELECT
  date_trunc('day', submitted_at) "day",
  sum(jsonb_array_length(chunk))
FROM
  xss.chunks
WHERE (user_id = $1 AND
       deleted_at IS NULL)
GROUP BY "day"
ORDER BY "day" DESC
```

Ez a lekérdezés a *select_events_by_user_id_aggregated_by_day* nevű egy paraméterrel (*user_id*) rendelkező Erlang függvényre képződik le. Az üzleti logikát tartalmazó modulok ezeket a lekérdezéseket a szerializációs / deszerializációs modulokon keresztül hívják meg, amelyek elvégzik az Erlang modell ←→ adatbázis rekord átalakításokat, illetve hibakezelést is tartalmaznak.

6.3 API

Az adatgyűjtő kliens és a webes vékonykliens az alkalmazás szerverrel HTTP protokollon keresztül kommunikál. A szerver különböző JSON API-t nyújt a kliensek számára az egyes funkcionalitásokhoz kötődően.



12. ábra Cowboy callback metódusok **[13]**

A HTTP kérések kezelésére a Cowboy nevű szoftvert használtam. A Cowboy egy Erlangban nyelven készült letisztult egyszerű webszerver implementáció. Útválasztási (routing) funkcióval is rendelkezik, az egyes HTTP végpontokra érkező

kéréseket különböző Erlang moduloknak továbbítja, amelyek a *cowboy_rest* viselkedést implementálják. Ezek a modulok különböző callback metódusokat implementálnak, amelyek a HTTP kérések ellenőrzését (például a kérés mérete, a tartalom típusa, megengedett metódusok, stb.) és a válasz összeállítását végzik.

6.3.1 Adatgyűjtő

Az adatgyűjtő kliens egyik alapvető funkcionalitása az összegyűjtött események továbbítása az alkalmazás szerverre. A szerverre ehhez a következő HTTP API-t biztosítja:

- végpont

  POST /api/1/s

- kérés tartalma

```
{
   "metadata": {
        "epoch": { "unit": string(), "value": integer() },
        "sessionId": string(),
        "streamId": string(),
        "sequenceNumber": integer(),
        "userId": string()
   },
   "chunk": array()
}
```

- válasz tartalma

```
{
   "response": "ok"
}
```

A kérést kezelő modul, amennyiben a kérés megfelel az előzőleges ellenőrzéseknek elkezdi feldolgozni a kérés tartalmát. A szerver a következő lépéseket hajtja végre egy új beérkező adatcsomag esetén:

1. Az adatcsomagot küldő IP címe és a HTTP Referer mező kinyerése.
2. A kérés tartalmának (body) kiolvasása és Erlang objektummá alakítása.
3. Új chunk modell objektum készítése a kérés tartalma és egyéb metaadatok alapján (IP címek, feldolgozás kezdetének időbélyege, stb.).
4. Új felhasználó, munkamenet és adatfolyam mentése, ha még nincsenek jelen az adatbázisban.

5. Az adatcsomagot küldő felhasználó eseményszámának (event_count) frissítése az adatbázisban az új adatcsomag eseményeinek hozzáadásával.
6. Az adatcsomag elmentése az adatbázisba.
7. Az adatcsomag mentése utáni aszinkron metódusok futtatása. A pontos funkciók a *6.4* vannak részletezve.
8. HTTP válasz összekészítése és elküldése a kliensnek.

6.3.2 Státusz

Az adatgyűjtő kliens alkalmazás másik fontos funkcionalitása a felhasználó státuszának lekérdezése az alkalmazás szerverről és a státusz megjelenítése. Az alkalmazás szerver ehhez a következő HTTP API-t biztosítja:

- végpont

  GET /api/1/status/:user_id/:stream_id

- kérés tartalma

  ```
  {}
  ```

- válasz tartalma

  ```
  {
     "phase": string()
     "description": string()
     "value": float()
  }
  ```

6.3.3 Statisztikák

6.4 Üzleti logikai réteg

6.5 Kiértékelő szoftver

6.6 Konfigurációs lehetőségek

7 Webes vékonykliens

8 Elvégzett munka értékelése

Az elvégzett munka értékelése.

8.1 Tesztelés

Az alkalmazás tesztelésének bemutatása.

9 Összefoglalás

A diplomaterv összefoglalása.

9.1 Konklúzió

A konklúzió ismertetése.

10 Köszönetnyilvánítás

11 Irodalomjegyzék

[1] „Windowing System,” 13 03 2022. [Online]. Available: https://en.wikipedia.org/wiki/Windowing_system. [Hozzáférés dátuma: 03 05 2022].

[2] „Wayland Architecture,” wayland, [Online]. Available: https://wayland.freedesktop.org/architecture.html. [Hozzáférés dátuma: 22 10 2022].

[3] „Rust Programming Language,” [Online]. Available: https://en.wikipedia.org/wiki/Rust_(programming_language). [Hozzáférés dátuma: 22 10 2022].

[4] „Rust Benchmarks,” [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-gpp.html. [Hozzáférés dátuma: 25 10 2022].

[5] „X11 Rust Bindings,” [Online]. Available: https://github.com/psychon/x11rb. [Hozzáférés dátuma: 25 10 2022].

[6] „Erlang System Architecture Introduction,” Ericsson, [Online]. Available: https://www.erlang.org/doc/system_architecture_intro/sys_arch_intro.html. [Hozzáférés dátuma: 22 10 2022].

[7] „Erlang Behaviours,” [Online]. Available: https://www.erlang.org/doc/design_principles/des_princ.html#behaviours. [Hozzáférés dátuma: 22 11 2022].

[8] „React Tutorial,” [Online]. Available: https://www.simplilearn.com/tutorials/reactjs-tutorial/what-is-reactjs. [Hozzáférés dátuma: 25 10 2022].

[9] suckless.org, „slock,” [Online]. Available: https://tools.suckless.org/slock/. [Hozzáférés dátuma: 01 11 202].

12 Függelék

12.1 Esemény séma

```
# Schema

## Event types

```
MOTION_EVENT_TYPE = 0;
SCROLL_EVENT_TYPE = 1;
TOUCH_BEGIN_EVENT_TYPE = 2;
TOUCH_UPDATE_EVENT_TYPE = 3;
TOUCH_END_EVENT_TYPE = 4;
BUTTON_PRESS_EVENT_TYPE = 5;
BUTTON_RELEASE_EVENT_TYPE = 6;
METADATA_CHANGED_EVENT_TYPE = 7;
```

Version `20220519T201520Z`

```
{
  MOTION_EVENT_TYPE: {
    types: [
      'type:type',
      't:timestamp:ms',
      'xIntegral:integer',
      'xFraction:integer',
      'yIntegral:integer',
      'yFraction:integer',
      'rootX:integer',
      'rootY:integer',
    ],
    name: 'XinputRawMotion',
    description: 'Raw motion event',
  },

 SCROLL_EVENT_TYPE: {
    types: [
      'type:type',
      't:timestamp:ms',
      'valueIntegral:integer',
      'valueFraction:integer',
      'rootX:integer',
      'rootY:integer',
    ],
    name: 'XinputRawMotion',
    description: 'Scroll event',
  },

  TOUCH_BEGIN_EVENT_TYPE: {
    types: [
      'type:type',
      't:timestamp:ms',
```
```

```
    'xIntegral:integer',
    'xFraction:integer',
    'yIntegral:integer',
    'yFraction:integer',
    'rootX:integer',
    'rootY:integer',
  ],
  name: 'XinputRawTouchBegin',
  description: 'Raw touch begin event',
},

TOUCH_UPDATE_EVENT_TYPE: {
  types: [
    'type:type',
    't:timestamp:ms',
    'xIntegral:integer',
    'xFraction:integer',
    'yIntegral:integer',
    'yFraction:integer',
    'rootX:integer',
    'rootY:integer',
  ],
  name: 'XinputRawTouchUpdate',
  description: 'Raw touch update event',
},

TOUCH_END_EVENT_TYPE: {
  types: [
    'type:type',
    't:timestamp:ms',
    'xIntegral:integer',
    'xFraction:integer',
    'yIntegral:integer',
    'yFraction:integer',
    'rootX:integer',
    'rootY:integer',
  ],
  name: 'XinputRawTouchEnd',
  description: 'Raw touch end event',
},

BUTTON_PRESS_EVENT_TYPE: {
  types: [
    'type:type',
    't:timestamp:ms',
    'rootX:integer',
    'rootY:integer',
    'detail:integer',
  ],
  name: 'XinputRawButtonPress',
  description: 'Raw button press event',
},

BUTTON_RELEASE_EVENT_TYPE: {
  types: [
    'type:type',
    't:timestamp:ms',
    'rootX:integer',
    'rootY:integer',
```

```
      'detail:integer',
    ],
    name: 'XinputRawButtonRelease',
    description: 'Raw button release event',
  },

  METADATA_CHANGED_EVENT_TYPE: {
    types: [
      'type:type',
      't:timestamp:ms',
      'metadata:object',
    ],
    name: 'MetadataChangedEvent',
    description: 'Metadata changed event',
  },
}
```