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

Összefoglaló **6**

Abstract **7**

1 Bevezetés **8**

1.1 A dolgozat szerkezete 8

2 Háttérismeretek **10**

2.1 Biometrikus autentikáció 10

2.2 Folyamatos autentikáció 11

2.3 Unix-szerű rendszerek grafikus felhasználói felülete 12

2.4 Megjelenítő protokollok 14

2.4.1 X Window System 14

2.4.2 Wayland 15

2.4.3 Összehasonlítás 16

2.5 Rust 19

2.6 Erlang 20

2.7 React 20

3 Kapcsolódó munka **22**

3.1 Megjelenítő protokollok 22

3.1.1 Kutatás és prototípus 23

3.1.2 Konklúzió 24

3.2 C-ben implementált adatgyűjtő 26

3.3 Adatbázis 26

3.4 Webes felület tervezése 27

4 A megoldás felépítése **28**

4.1 Használati esetek 28

4.2 Komponensek kommunikációja 30

5 Linux kliens alkalmazás **31**

5.1 Követelmények 31

5.2 Az alkalmazás felépítése 31

5.3 Adatgyűjtő 32

5.3.1 Platform specifikus metaadatok 36

5.4 A felhasználó státusza ... 37
5.5 Konfigurációs lehetőségek ... 39
5.6 Telepítés ... 42
6 Az alkalmazás szerver ... 43
6.1 Követelmények ... 43
6.2 Az alkalmazás felépítése ... 44
6.3 Üzleti logikai réteg ... 45
6.3.1 Események feldolgozása ... 45
6.3.2 Profil építés és verifikáció ... 46
6.3.3 A felhasználó státusza ... 49
6.4 HTTP API ... 49
6.4.1 Adatgyűjtő ... 50
6.4.2 Státusz ... 50
6.4.3 Statisztikák ... 51
6.5 Adatbázis ... 53
6.5.1 Séma ... 53
6.5.2 Telepítés ... 57
6.5.3 Lekérdezések ... 57
6.6 Kiértékelő szerver ... 58
6.7 Konfigurációs lehetőségek ... 59
6.7.1 Alkalmazás szerver ... 60
6.7.2 Kiértékelő szerver ... 61
6.8 Telepítés ... 62
7 Webes vékonykliens ... 64
7.1 Követelmények ... 64
7.2 Az alkalmazás felépítése ... 64
7.3 Összesített statisztikák ... 65
7.4 Felhasználó-specifikus statisztikák ... 67
7.5 Konfigurációs lehetőségek ... 68
7.6 Telepítés ... 70
8 A megoldás tesztelése ... 71
8.1 Linux kliens alkalmazás ... 71
8.2 Alkalmazás szerver ... 73
8.3 Webes vékonykliens ... 77

9 Összefoglalás..... 78
9.1 További fejlesztési javaslatok..... 79
10 Irodalomjegyzék..... 80
11 Függelék..... 82
11.1 Esemény séma..... 82

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. 12. 04

.....

Sándor Dávid

Összefoglaló

A koronavírus világjárvány alatt sok cég részben vagy teljes egészében átállt az otthoni munkavégzésre. Az otthoni munkavégzés a tapasztalatok szerint több szektorban (főleg IT) sikeres volt, a munkavállalók produktívabbak voltak, ezért sok vállalat a szabályozások feloldása után sem állt vissza a hagyományos irodai munkavégzésre. Felmerül a kérdés, hogy hogyan tudják ezek a vállalatok biztosítani azokat a biztonsági protokollokat, amik védték őket amíg a dolgozók az irodában tartózkodtak. Egyre nagyobb az igény egy olyan autentikációs módszerre, amellyel az alkalmazottak, a munkavégzésük megszakítása nélkül, az egyedi viselkedési sémájuk alapján azonosíthatók. A diplomamunkám keretein belül a Cursor Insight által jelenleg is aktívan fejlesztett kurzormozgás alapú biometrikus folyamatos autentikációs módszert kiegészítő megoldást mutatok be. A dolgozatban ismertetek egy natív Linux operációs rendszeren futó adatgyűjtő és munkamenet zároló klienst, egy webes vékonykliens adminisztrációs felületet, illetve a klienseket kiszolgáló alkalmazás szervert. A diplomatervet az elkészült megoldás tesztelésének bemutatásával és további fejlesztési javaslatokkal zárom.

Abstract

During the COVID-19 pandemic, many companies switched to working from home, either partially or entirely. Experience has shown that working from home has been successful in many sectors (especially IT), with workers being more productive, and many companies have not reverted to traditional office working even after the lifting of the regulations. The question arises as to how these companies can ensure the security protocols that protected them while employees were in the office. There is an ever-growing demand for a continuous authentication method which can identify employees based on their unique behavioural patterns without interrupting their work. In this paper, I will present a solution for a cursor movement based biometric continuous authentication method that is currently under active development by Cursor Insight. In the thesis I will describe a data collection and session locking client running on native Linux, a web client administration interface, and an application server. The thesis concludes with a presentation of the testing of the final solution and suggestions for further development.

1 Bevezetés

Egy vállalat életében, ahol a munkavállalók számítógépes munkát végeznek, amely során hozzáférnek szenzitív adatokhoz megkerülhetetlen bizonyos biztonsági protokollok használata. A cél, hogy külső személy ne férjen hozzá a munkavállaló eszközén keresztül a céges rendszerekhez. Hagyományos irodai munkavégzés esetén ezt többek közt azzal tudják biztosítani, hogy a munkaeszközök fizikailag hozzáférhetetlenek egy külső támadó számára (például dolgozói mágneskártya), illetve a gépek egy belső hálózaton működnek szigorú tűzfalszabályok mellett. A koronavírus járvány során sok cég részben vagy teljes egészében átállt az otthoni munkavégzésre. Az otthoni munkavégzés a tapasztalatok szerint több szektorban (főleg IT) sikeres volt, a munkavállalók produktívabbak voltak, ezért sok vállalat a szabályozások feloldása után sem állt vissza a hagyományos irodai munkavégzésre.

Felmerül a kérdés, hogy ezek a vállalatok hogyan tudják biztosítani az eddig használt protokollok által nyújtott biztonságot, abban a környezetben, ahol a fizikai hozzáférhetetlenség már nem áll fent. A biometrikus folyamatos autentikáció többek közt erre a problémakörre nyújt megoldást.

A Cursor Insight, ahol a diplomamunka készítése során aktív munkaviszonnyal rendelkezem évek óta foglalkozik biometrikus felhasználó azonosítással. A diplomamunkámban egy folyamatos biometrikus autentikációs megoldást mutatok be, amely kurzormozgás alapján azonosítja a felhasználókat. Az általam készített megoldás tartalmaz egy natív Linuxon futó kurzormozgás adatgyűjtő kliens alkalmazást, egy alkalmazás szervert és egy webes vékonykliens adminisztrációs felületet. A Cursor Insight által fejlesztett kiértékelő algoritmus nem része a diplomamunkának, erre az általam készített megoldás külső szolgáltatásként tekint.

1.1 A dolgozat szerkezete

A diplomamunka a háttérismeretek bemutatásával kezdődik. Ebben a fejezetben olyan témakörök kerülnek bemutatásra, amelyek a diplomaterv megértését segítik. Bemutatásra kerülnek a releváns autentikációs módszerek, a Unix-szerű rendszerek grafikus felhasználói felületeinek felépítése, illetve áttekintjük a felhasznált technológiákat, keretrendszereket. A következő fejezetben a diplomatervhez kapcsolódó

munkámat mutatom be, prototípusokat ismeretetek és megindokolom a felmerült tervezői döntéseket. Ezt követően a megoldás architektúráját és a megoldást felépítő komponensek közti kommunikációt ismertetem. Az ezután következő három fejezet sorban a natív Linuxon futó adatgyűjtő kliens, az alkalmazás szerver és a webes vékonykliens dokumentációját tartalmazza. Az alkalmazások dokumentációja után az elkészült megoldás tesztelését mutatom be. A diplomatervet az elvégzett munka értékelésével és további fejlesztési javaslatokkal zárom.

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 autentikáció

A biometrikus autentikáció alatt a hitelesítési mechanizmusoknak egy olyan családját értjük, amelyek alapjául valamilyen biológiai megkülönböztető jel, vagy viselkedésbeli karakterisztika szolgál. Általánosságban a biometrikus autentikációs 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 valószínűségi alapon működnek. Egy jelszavas hitelesítés esetén a rendszer *mindenkit* beenged, aki a helyes jelszót adja meg, tehát tulajdonképpen bárki, aki megismeri a jelszót hozzáfér a rendszerhez. Ezzel szemben egy biometriában tárolt információ tipikusan nehezen hozzáférhető egy támadó számára. Amennyiben a támadó mégis hozzáfér ehhez az információhoz, az adatbevitel nehezen reprodukálható. Például egy ujjlenyomatot könnyű olvasni és tárolni, de ahhoz szakértelemre van szükség, hogy olyan formában elő tudják állítani, hogy azt egy ujjlenyomat leolvasónak beadható legyen. Ez egy lényegesen nehezebb feladat, mint egy szöveges jelszó beírása egy beviteli mezőbe.

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

- **Fiziológiai (statikus)**

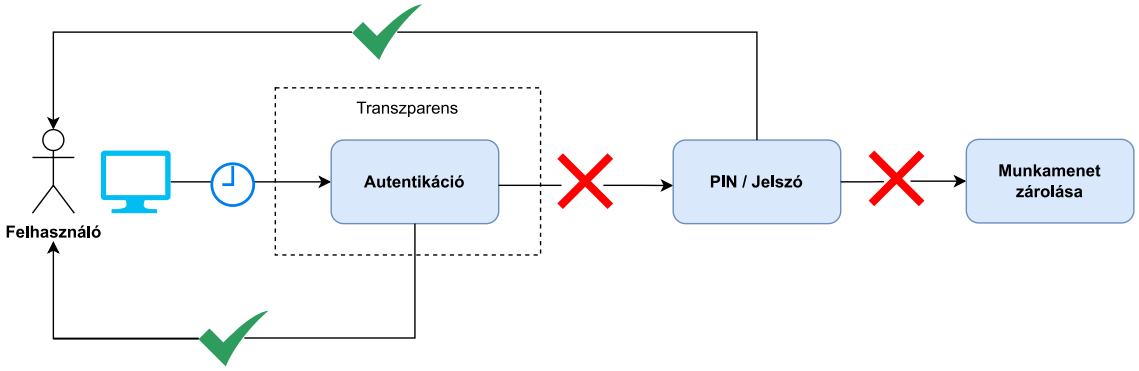
  A felhasználó fizikai karakterisztikáit használják, például ujjlenyomat, retina, arc formája, vénák szerkezete stb.

- **Viselkedés alapú (dinamikus)**

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

2.2 Folyamatos autentikáció

A folyamatos autentikáció egy olyan hitelesítési módszer, ahol a felhasználó személyazonosságát valós időben lehet megerősíteni. A klasszikus statikus autentikációs megoldásoktól (például egy jelszó megadása vagy egy második faktor megadása a bejelentkezésnél) abban tér el, hogy a felhasználót a teljes munkamenet során, folyamatosan ellenőrzi. A folyamatos autentikációs 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 autentikáció

A folyamatos autentikáció működéséről ad egy áttekintő képet az 1. ábra A felhasználó valamilyen munkafolyamatot végez (például banki tranzakció). Bizonyos időközönként a háttérben a felhasználó személye ellenőrzésre kerül, a folyamatos autentikáció 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 autentikációból. Amennyiben az ellenőrzés elutasítja a felhasználót, akkor javasolt újra megbizonyosodni a felhasználó kilétéről egy vagy több azonosítási faktor bekérésével (például PIN kód, vagy jelszó megadása). Ha így sikerült azonosítania magát a felhasználónak, akkor visszatérhet a munkamenethez, ha nem, akkor a munkamenet zárolásra kerül például a tranzakció megszakításával vagy a banki felületről való kijelentkeztetéssel.

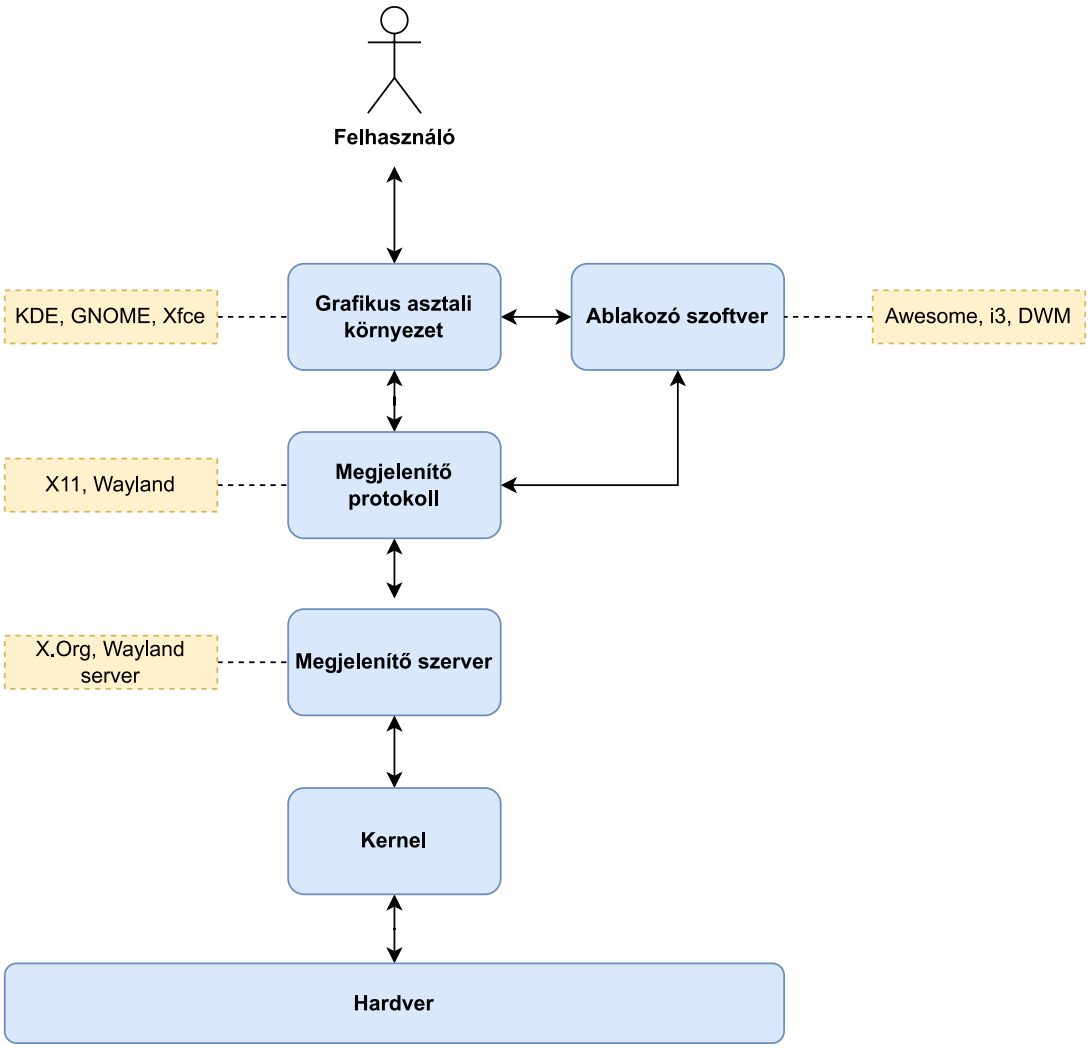
2.3 Unix-szerű rendszerek grafikus felhasználói felülete

A Unix-szerű operációs rendszerek esetén a grafikus felület architektúrája, amennyiben a rendszer egyáltalán rendelkezik GUI-val (Graphical User Interface) összetett. Legtöbbször a GUI több komponensből áll, a teljesség igénye nélkül:

- **Asztali környezet**, amely több kisebb komponenst foglal össze úgy, hogy egy közös grafikus felületet nyújtson a felhasználó számára.
- **Ablakozó szoftver**, ami az ablakok grafikus megjelenését és elrendezését vezérli.
- **Widget könyvtárak**, azaz olyan függvények összessége, amelyek létrehozzák a grafikus felhasználói felületet, amellyel a felhasználó interakcióba lép. Például GTK, QT, Electron stb.
- **Bemeneti / kimeneti eszközök**, például számítógépes egér, billentyűzet, monitor stb.
- **Megjelenítő szerver**, ami az alsóbb szintű (kernel közeli) funkciókat összefogja és interfészeket nyújt a felsőbb szintű komponensek számára.
- **Megjelenítő protokoll,** amin keresztül a megjelenítő szerver a klienseivel kommunikál.

A 2. ábra betekintést nyújt egy tipikus GUI felépítésébe. 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, widgetek, 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. ábra A GUI felépítése

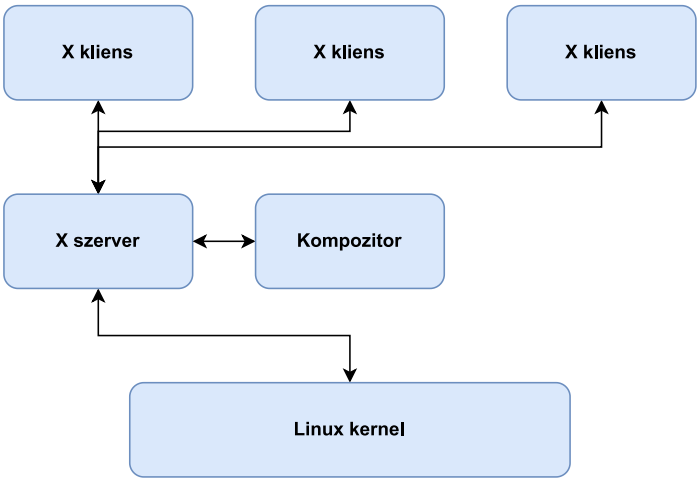
2.4 Megjelenítő protokollok

Linux rendszerek esetén többféle megjelenítő protokollal találkozhatunk és egy adott protokollhoz általában többféle implementáció is létezik. [1] Az idők során két megjelenítő protokoll terjedt el nagyobb körben, az X Window System, illetve a Wayland protocol. 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 diplomamunka 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, az X.Org Server. A protokollt megvalósító elterjedtebb C-ben implementált kliensoldali könyvtárak az Xlib és az XCB.

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



3. ábra Az X Window System architektúrája

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 rajzolni. Ez egyszerre több monitoron is megjelenhet (akár duplikálva, akár kiterjesztve).
- **display (kijelző)** – Képernyők gyűjteménye, amely gyakran több monitort foglal magába. A Linux-alapú rendszerek 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 létrehozását, 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.

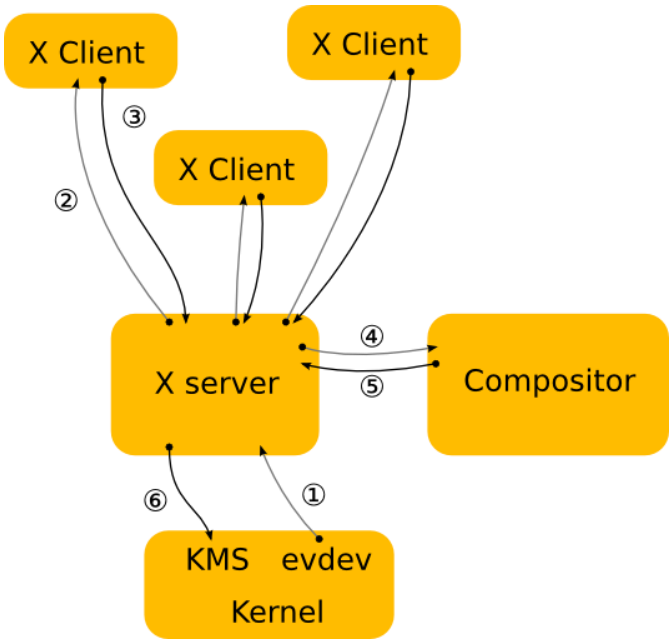
2.4.2 Wayland

A Wayland [2] egy ingyenes, nyílt forráskódú megjelenítő protokoll. A projektre „következő generációs” megjelenítő szerverként hivatkoznak és célja, hogy leváltsa az

X Window Systemet egy modernebb, egyszerűbb és biztonságosabb protokollra. A protokollt 2008-ban kezdték el fejleszteni (többen az X.Org szerver fejlesztő csapatából) és a mai napig aktívan dolgoznak rajta. Az X Window Systemhez 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 architekturális 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.

2.4.3 Összehasonlítás

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:

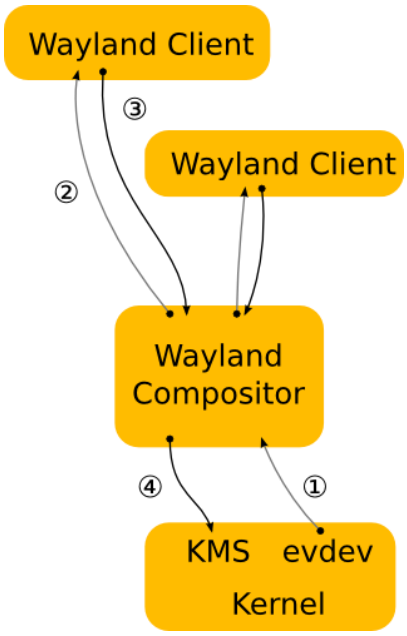


4. ábra Egy esemény feldolgozása X11-en

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 checkboxot. Az esemény feldolgozása után a kliens egy újrarajzolási kérést (request típusú üzenet) küld a szervernek.
4. Az X szerver megkapja az újrarajzolási kérést és egy illesztőprogramon keresztül szól a hardvernek, hogy az végezze el a rajzolást. Az X szerver továbbá kiszámítja az újrarajzolási kérés 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 a 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 megjelenítő 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 Waylenden

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 adott 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 architektúrában a 4-es és az 5-ös lépés.
3. Az X-nél látott folyamathoz hasonlóan a kliens megkapja az eseményt és feldolgozza azt. Egy újabb különbség a protokollok között, hogy a Wayland esetén a rajzolás (render) kliens oldalon történik, gyakorlatban a kliens a kompozitorral megosztott közös videómemória pufferbe renderel. Ennek az az előnye, hogy egy újra rajzolási kérésnek nem kell átmennie egy plusz komponensen (X szerver). 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 megjelenítő szerver helyére lép a kompozitor lényegesen csökkent a rendszer komplexitása, illetve a kommunikációs többlet.

2.5 Rust

A Rust [3] egy viszonylag új, általános célú programozási nyelv. Az első stabil verziója 2014-ben jelent meg. A nyelv megalkotásakor a hangsúly a teljesítményen, a konkurencia támogatásán és a biztonságon volt. Leggyakrabban rendszerszintű programozásra szokták használni, de magasabb szintű programok, alkalmazások implementálása során 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 fordított (compiled) nyelv, amelynek az egyik legfontosabb tulajdonsága a biztonságos memóriakezelés, *garbage collector* mechanizmus használata nélkül. A memóriabiztonság betartására, illetve a versenyhelyzetek elkerülésére a Rust a tulajdonossági mechanizmust (ownership) használja, ami leegyszerűsítve az objektumok élettartamának követését és a változók hatókörének (scope) ellenőrzését jelenti fordítási időben. Ennek eredménye, hogy futásidőben nincs vagy nagyon ritka a végzetes memóriahiba.

A diplomamunka implementálása során az adatgyűjtő és munkamenet zároló 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 könnyebb. Ezen kívül fontos volt még, hogy a nyelvhez elérhető X11 protokoll implementáció [5], amelyet a protokollt leíró XML leíróból generáltak, nem 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/OTP (Open Telecom Platform) – az Erlang és az Erlang/OTP kifejezést sokszor felcserélhető módon használják – az Erlang környezetből, számos „off-the-shelf" Erlang könyvtárból és tervezési mintából 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 konkurens 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. Mivel az Erlang programok nem dedikált hardveren futnak ezért a kemény valós idejű működéssel (hard real-time) szemben egyes kérelmek esetében előfordulhat, hogy azok lemaradnak a határidőkről.
- **Kód cserélése futásidőben** (Hot swapping): Az Erlang/OTP-ben található tervezési minták generikus módon támogatják egy futó folyamat kódjának frissítését anélkül, hogy a programot le kéne állítani.

2.7 React

A React [6] 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 Reactet gyakran használják valamilyen ráépülő keretrendszerrel (pl. Next.js, Gatsby, CRA). Többek közt alkalmas SPA (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 Reactet 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 Megjelenítő protokollok

A natív Linuxon 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-et 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 Systemet. 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.

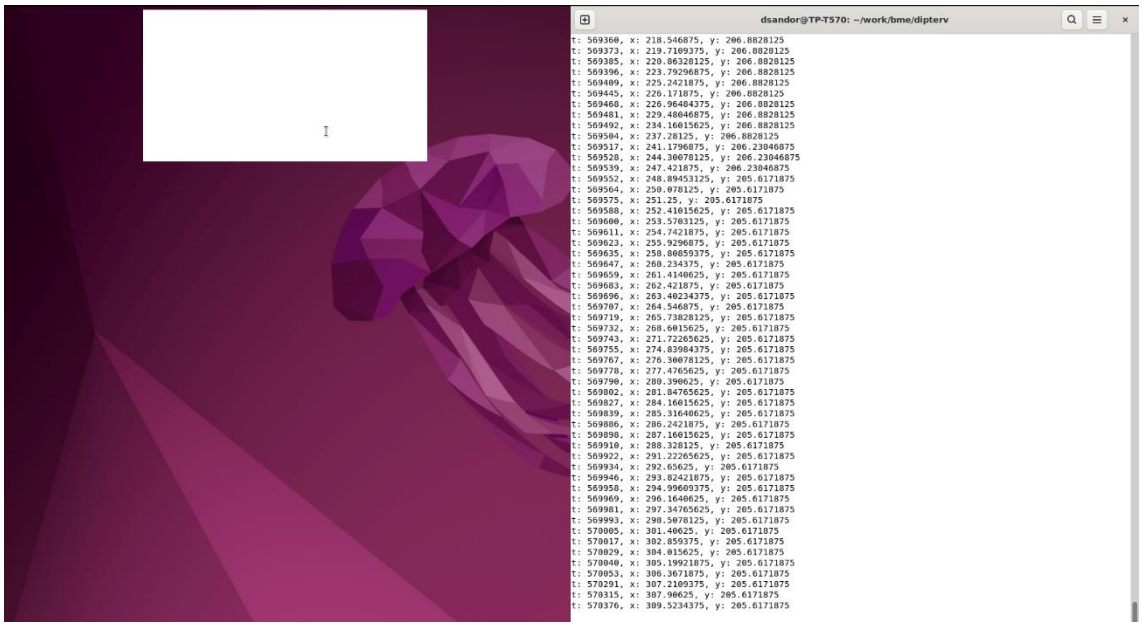
Több nagyobb felhasználóbázissal rendelkező Linux disztribúció (például: Debian, Ubuntu, Fedora) elkezdett átállni a Waylandre, mint alapértelmezett megjelenítő protokoll (legalábbis a GNOME asztali környezetet használó verziók). Mivel a Wayland protokoll modernebb és a jövőben minden bizonnyal át fogja 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 kezdtem 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 libwaylandhez 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 PyWaylandet, egy Python nyelvhez készült burkoló könyvtárat választottam.

A Pythonban 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. 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 kurzor mozgatási 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 Waylandel kapcsolatos kutatáshoz és elkezdtem mások által készített alkalmazások forráskódját tanulmányozni. Elsősorban olyan alkalmazásokra koncentráltam, amelyeknél nagy valószínűséggel felmerült az a probléma, amivel én is találkoztam. 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 követést valósít meg (xeyes). A kutatás eredményeként azt állapítottam meg, hogy általánosságban három különböző típusú működéssel lehet találkozni az említett szoftvereknél:

1. A vizsgált alkalmazások egy része egyáltalán nem támogatja a Wayland protokollt.
2. A vizsgált alkalmazások egy része kísérleti jelleggel, részlegesen támogatja a Wayland protokollt, azaz bizonyos funkciók nem elérhetőek a szoftverben Wayland alatt.
3. A vizsgált alkalmazások egy része valamilyen megszorítások mellett támogatja a Waylandet, például csak GNOME asztali környezeten működik.

Lényegében azt a problémát figyeltem meg, hogy az alap Wayland protokoll meglehetősen szűkre szabott funkcionalitású 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 Waylandet, mert a GNOME által használt Wayland kompozitor (Mutter) implementál bizonyos protokoll kiegészítéseket, amelyek ezt lehetővé teszik. Például a Mutter implementál egy protokoll kiegészítést (gnome-shell-touchpad-window-move), amely lehetővé teszi a Three Finger Window Move GNOME bővítmény számára, hogy gesztusvezérléssel lehessen az ablakokat pozícionálni. A helyzet viszont még ennél is bonyolultabb, ugyanis különböző Wayland kompozitor implementációk különböző protokoll kiegészítéseket definiálnak, amelyek egymással általában nem kompatibilisek. Tehát ahhoz, hogy egy olyan alkalmazást készítsek, ami minden Waylandet használó disztribúción működik vagy az alap protokollt szabad használnom, vagy implementálnom kellene a különböző asztali környezetekhez szükséges 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 Waylandre. Az itt olvasható okok miatt én a továbbiakban a kliens alkalmazás implementálása során visszatértem az X Window System protokollhoz.

3.2 C-ben implementált adatgyűjtő

A natív Linuxon futó adatgyűjtő kliens program tervezési fázisában rendelkezésemre állt egy korábbi, a Cursor Insight által készített implementáció, amely C-ben készült el. Ez a program az X Window System megjelenítő protokollhoz készült és az Xlib kliens oldali könyvtárt használja. A céges implementáció funkcionalitását tekintve az általam készített megoldás funkcióinak csak egy részhalmazával rendelkezik, például nem tud munkamenetet zárolni, az eseményeknek nincs kidolgozott sémája, nincs felhasználó kezelés stb. Mindezek mellett a megjelenítő protokoll használatára, a megjelenítő szerverrel való kommunikációra egy jó példát nyújt, amelyből az általam készített implementáció során is merítettem.

3.3 Adatbázis

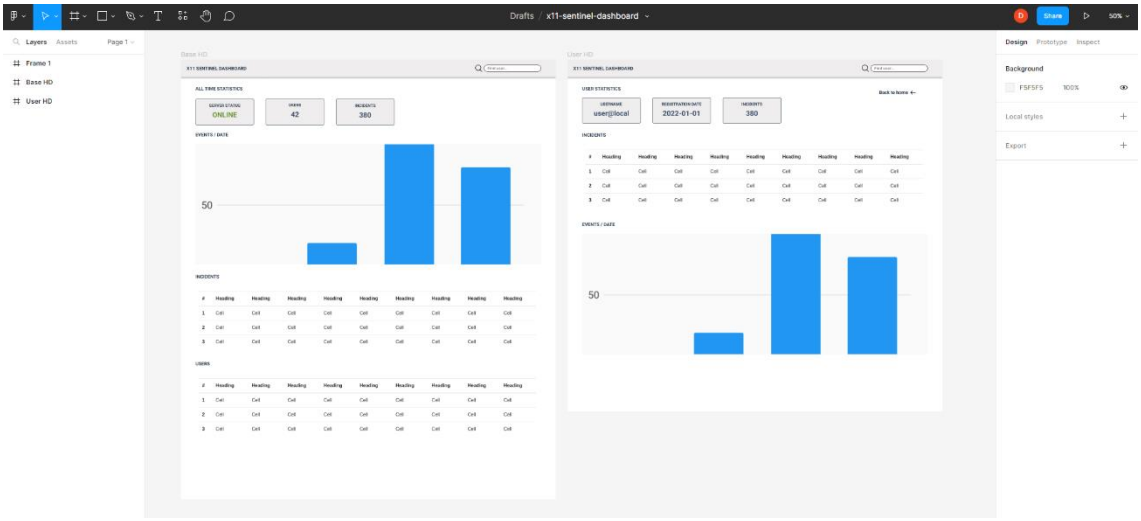
Az alkalmazás szerver tervezési folyamatának egy fontos része volt a perzisztenciával kapcsolatos kérdések eldöntése. Az első eldöntendő kérdés az adatbázis típusának kiválasztása volt. Felmerültek a relációs adatbázisokon kívül a NoSQL adatbázisok és az idősoros adatbázisok. A dimenziók, amelyek mentén a döntés megszületett a következők voltak:

- az adatmodell illeszkedése az egyes adatbázis típusokhoz,
- az adatbázisok teljesítménye és skálázhatósága,
- Erlang nyelvhez elérhető illesztőprogramok,
- személyes tapasztalat.

Ezek alapján végül a relációs adatbázisok mellett döntöttem, pontosabban a PostgreSQL mellett. 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, amit az általam használt lekérdezések egyszerűsége nem indokolt. Másrészt – ami szintén nyomós indok volt – a jelenleg elérhető Erlang nyelvhez készült ORM szoftverek nem bizonyultak kellőképpen naprakésznek és testreszabhatónak.

3.4 Webes felület tervezése

A webes vékonykliens tervezése során a felhasználói felület tartalmának és dizájn elemeinek megtervezésére a Figma [7] szoftvert használtam. A Figma egy kollaboratív webes alkalmazás a felülettervezéshez. A szoftver a felhasználói felület- és élménytervezésre összpontosít, különböző támogatásokat nyújtva a megtervezett felület implementációjához (például stílusok, vektorgrafikák exportálása). A szoftver segítségével különböző képernyőméretekre terveztem meg a webes vékonykliens alkalmazás felületét.



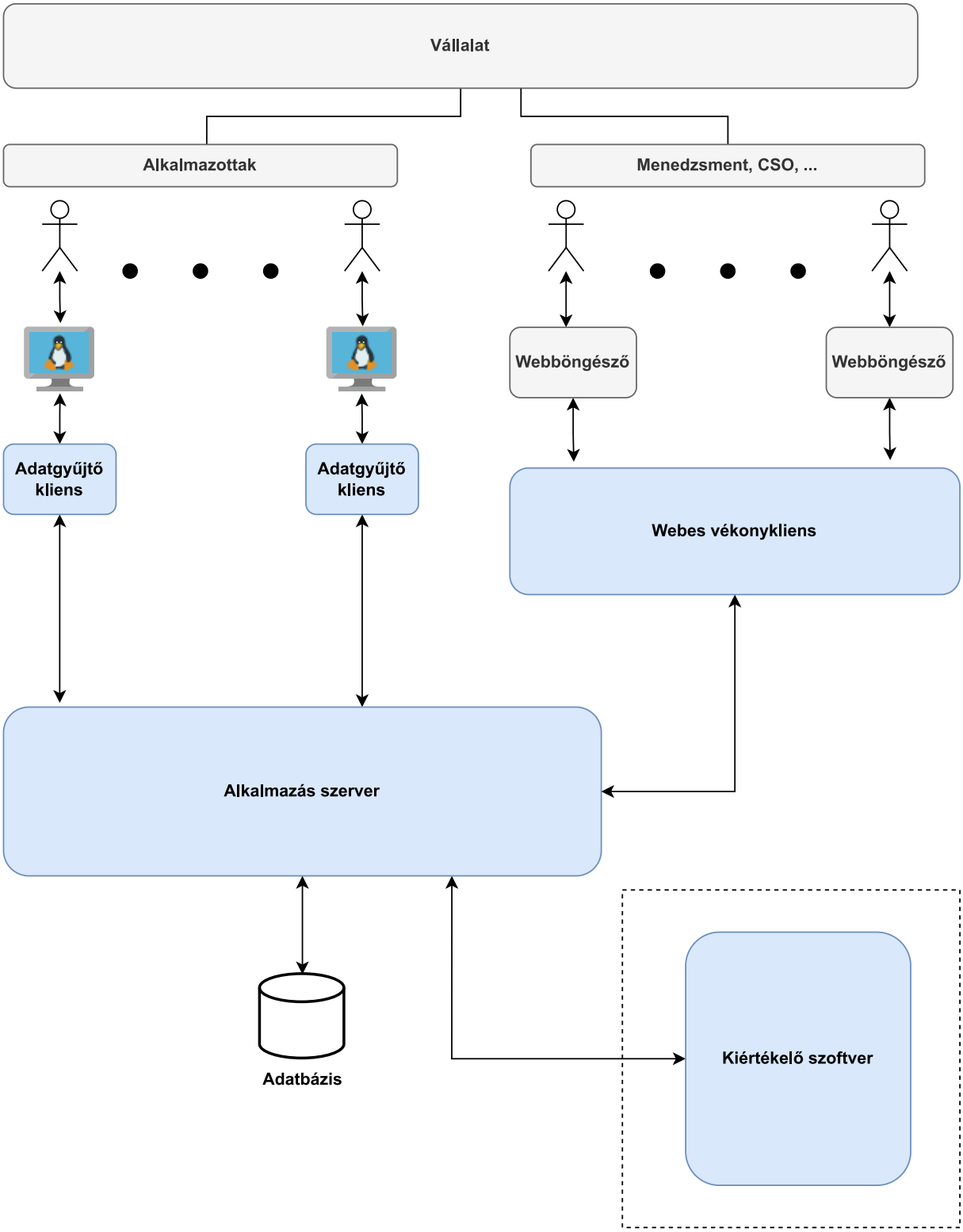
7. ábra A webes felület tervezése

4 A megoldás felépítése

Ebben a fejezetben ismertetem a megoldás (solution) architektúráját, a megoldást felépítő alkalmazások kapcsolatait és határait. A fejezet célja egy absztrakt képet nyújtani a megoldás 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 Használati esetek

A 8. ábra a megoldás felépítését egy képzeletbeli vállalat példáján keresztül mutatja be. A vállalatnál dolgoznak alkalmazottak, illetve vannak más pozícióban dolgozó emberek, például menedzsment tagok vagy a biztonsági vezető (CSO, Chief Security Officer). 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 alkalmazottai a napi teendőik során használják a számítógépes egeret, touchpadet stb., amellyel kurzoradatokat 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. Azokat a verifikációkat, amelyek egy előre megadott küszöbértéknél alacsonyabb pontot érnek el incidensnek nevezzük. Az alkalmazás szerver a verifikáció eredményét továbbítja a kliens alkalmazásnak, amely az adott konfigurációtól függően képes a munkamenet felfüggesztésére (azaz a felhasználó kizárására), amennyiben egy incidens történt.



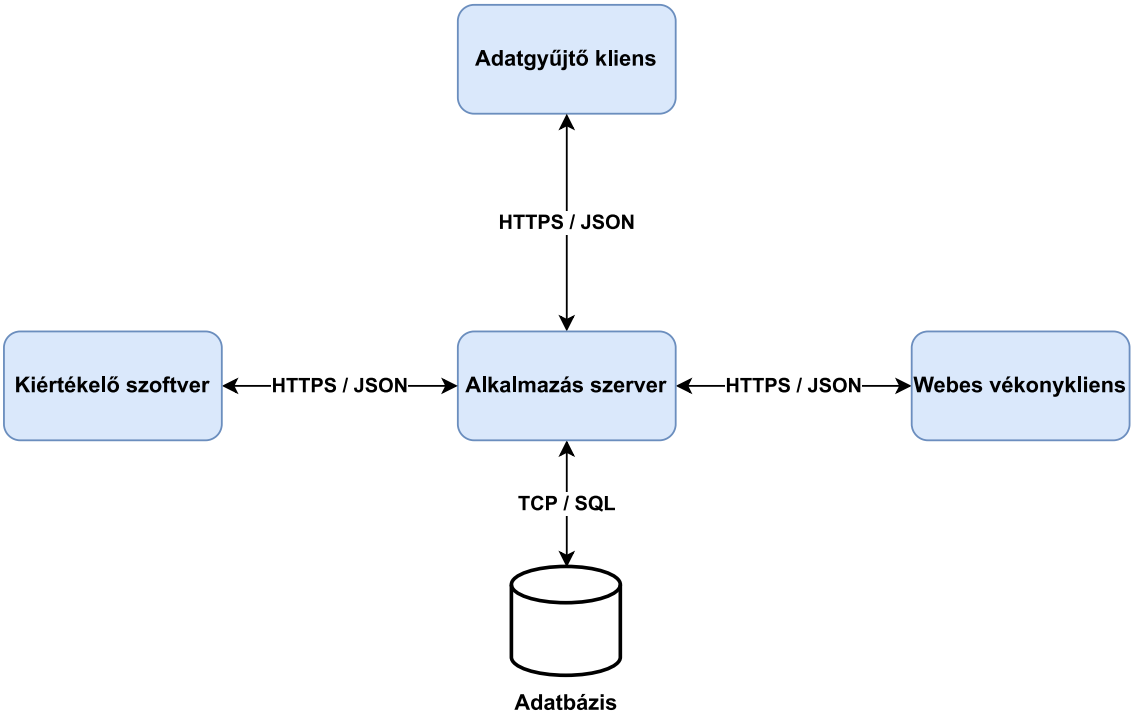
8. ábra A megoldás architektúrája

A vállalatnál dolgoznak egyéb pozícióban is 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 is.

4.2 Komponensek kommunikációja

A 9. ábra az alkalmazás komponensek egymással való kommunikációját szemlélteti. 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 formátumban utaznak. Az alkalmazás szerver által nyújtott API (Application Programming Interface) részletes dokumentációja a *6.* fejezetben olvasható. Az alkalmazás szerver az adatbázissal TCP protokollon keresztül kommunikál, az adatbázis lekérdezések SQL nyelven vannak megfogalmazva.



9. ábra Komponensek kommunikációja

5 Linux kliens alkalmazás

Ennek a fejezetnek a célja a natív Linux kliens kurzormozgás adatgyűjtő és munkamenet záró alkalmazás részletes dokumentációja és 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:

RC1 Az alkalmazás legyen képes GUI-val rendelkező natív Linux operációs rendszeren kurzormozgás adatot gyűjteni különböző beviteli eszközöktől.

RC2 Az alkalmazás el tudja küldeni a gyűjtött adatokat az alkalmazás szervernek.

RC3 Az alkalmazás le tudja kérdezni az aktuális felhasználó státuszát az alkalmazásszervertől.

RC4 Az alkalmazás meg tudja jeleníteni valamilyen formában a felhasználó státuszát.

RC5 Az alkalmazás rendelkezzen munkamenet záró funkcionalitással.

RC6 Az alkalmazás képes platform specifikus metaadatok gyűjtésére és azok elküldésére az alkalmazás szervernek.

RC7 Az alkalmazásnak felhasználói felület nélkül, a háttérben kell tudnia futni, a felhasználó munkamenetének megzavarása nélkül.

RC8 Az alkalmazásnak lehetőség szerint minél nagyobb felhasználói bázist kell tudnia támogatni.

RC9 Az alkalmazás könnyen konfigurálható legyen környezeti változók és / vagy parancssori argumentumok használatával.

5.2 Az alkalmazás felépítése

A *RC8*-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 Rustot választottam, ennek az indoklása a *2.5*-ös fejezetben olvasható.

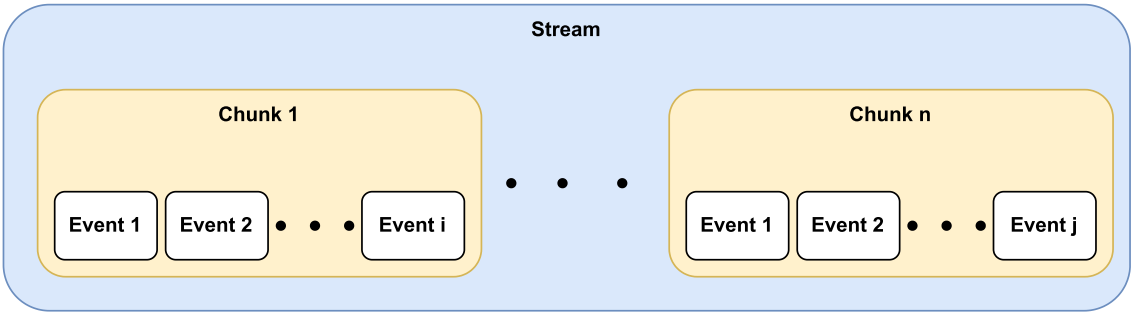
Ahhoz, hogy a felhasználó által generált kurzormozgás eseményeket a kliens program fogadni tudja szükséges a felhasználó számítógépén futó megjelenítő szerverrel történő kommunikáció. Ezt a program a megjelenítő protokollon keresztül tudja megtenni. A Rust nyelvhez elérhető több X11 implementáció is. 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 alkalmazás többszálú, a program indulásakor két szálat indít. Az alkalmazás főszála az *RC1*, *RC2*, *RC6* követelmények által leírt funkcionalitás valósítja meg, azaz a kurzormozgás adatok és platformspecifikus metaadatok gyűjtését és továbbítását az alkalmazás szerverre. A másik szál az *RC3*, *RC4*, *RC5* követelményeket elégíti ki, azaz lekérdezi és megjeleníti a felhasználó aktuális státuszát, illetve, ha szükséges, akkor a zárolja a munkamenetet. A kliens alkalmazás fordítási folyamatának eredménye egy futtatható állomány, amelyet egy parancssorból lehet elindítani leválasztott (detached) módban, ezzel kielégítve az *RC7*-es követelményt. Az alkalmazást környezeti változókkal és parancssori argumentumokkal egyaránt lehet konfigurálni (*RC9*).

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. Ezt a funkciót egy különálló Rust modulban valósítottam meg az alkalmazás implementálása során.

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 HTTP 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 10. ábra



10. ábra Az adatfolyam felépítése

A modul tartalmaz egy 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. Amennyiben 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. Az eseménysor időpecsétei ehhez az epochhoz képest értendők.

- **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 ha változó sorrendben érkeznek meg a chunkok a szerver oldalon, akkor is sorrendezhető maradjon az adatsor.
- **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 puffert. 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. A külön szál előnye, hogy nem blokkolja a bejövő események tárolását – várhatóan jó minőségű adatgyűjtést lehet így megvalósítani, az időkülönbségek eltérése várhatóan nem lesz nagy. 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 tengely menti elmozdulásának egész része.
- **xFraction:** A kurzor X tengely menti elmozdulásának tört része.
- **yIntegral:** A kurzor Y tengely menti elmozdulásának egész része.
- **yFraction:** A kurzor Y tengely 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 *11.1*-es fejezetben.

5.3.1 Platform specifikus metaadatok

Az *RC6*-os követelmény értelmében az alkalmazás különböző platform specifikus metaadatot gyűjt és ezt elküldi 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) időközönként 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 JSON formátumba. A *serde* 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 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. A metaadatok kitöltésére az *os_info* külső modult 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 *RC3, RC4, RC5* 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.

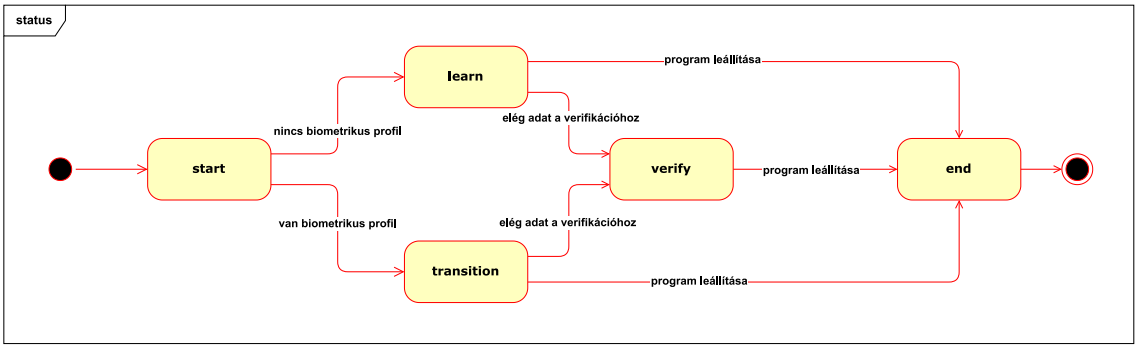
A státusz lekérdezése a kliens program indulása után periodikusan történik. A státusz lekérdezések közti idő intervallum konfigurálható. A felhasználó státusza a következőképpen épül fel:

```
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.



11. 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* [8] nevű program van beállítva erre a célra. Ez egy 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 a csendes riasztást (silent alert) preferálnak.

5.5 Konfigurációs lehetőségek

Az *RC9*-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:

- **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”**

- **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”**

- **BUFFER_SIZE_LIMIT**

  Felső korlát az adatgyűjtő esemény pufferének méretére. Amennyiben a puffer mérete eléri ezt a korlátot a kliens elküldi az eseményeket az alkalmazás szerverre.

  Alapértelmezett értéke: **100**

- **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 (10 másodperc)**

- **LOCK_ENABLED**

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

Alapértelmezett értéke: **false**

- **LOCK_THRESHOLD**

  A munkamenet zároló funkció esetén használt küszöbérték. Ez alatt a küszöbérték alatt a felhasználó verifikációja sikertelennek minősül.

  Alapértelmezett értéke: **0.5**

- **LOCK_UTILITY**

  A munkamenet zárolás esetén hívott külső segédprogram.

  Alapértelmezett értéke: **„slock”**

- **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**

- **STATUS_BASE_URL**

  Az alkalmazás szerver státusz lekérdezési végpontjának URL-je.

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

- **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**

- **SUBMIT_URL**

  Az alkalmazás szerver adatküldő végpontjának URL-je.

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

- **USER_ID**

A felhasználó azonosítója.

Alapértelmezett értéke: **„default_user”**

5.6 Telepítés

Az adatgyűjtő kliens alkalmazás telepítéséhez két lépésre van szükség. Először a forráskódból létre kell hozni a futtatható állományt (build), majd a futtatható állományt el kell indítani opcionálisan parancssori argumentumok vagy környezeti változók megadásával. Az alkalmazás fordítási folyamatát Dockerrel valósítottam meg. Az alkalmazást az operációs rendszeren natív módon kell futtatni, ugyanis egy Docker konténeren belülről biztonság technikai okok miatt nem lehet (vagy legalábbis meglehetősen nehéz) hozzáférni a *host* környezet megjelenítő szerveréhez. Ezért a Docker támogatás csak az alkalmazás *build* fázisra terjed ki. Ehhez a következő parancs kiadására van szükség:

```
$ docker build -o bin .
```

Ezzel létrejön a *bin* mappán belül a *x11-sentinel-client* nevű futtatható állomány, amelyet parancssorból, például a következő paranccsal lehet elindítani:

```
$ bin/x11-sentinel-client --user-id user@test.com --lock-enabled true
```

6 Az alkalmazás szerver

A következő fejezet célja a kliens programokat kiszolgáló alkalmazás szerver részletes dokumentációja. A leírásban először ismertetem a szerver programmal szemben felállított követelményeket, majd az alkalmazás felépítését. Ezután az üzleti logikai réteg dokumentációja következik. Ezt követően a kliens programok felé nyújtott HTTP API-t ismertetem. Ezután az adatbázis réteggel kapcsolatos tudnivalókról, majd a kiértékelő szoftverről lesz szó. A fejezetet végül a szerver konfigurációs lehetőségeinek bemutatásával zárom.

6.1 Követelmények

RS1 A szerver biztosítson HTTP REST API-t az adatgyűjtő kliens által küldött események fogadására és feldolgozására.

RS2 A szerver legyen képes adatbázisba menteni az adatgyűjtő kliens által küldött eseményeket.

RS3 A szerver tartsa karban adatbázisban a felhasználókat, a felhasználói munkameneteket és az adatfolyamokat az adatgyűjtő kliens által küldött információ alapján.

RS4 Amennyiben egy felhasználónak elegendő mozgásadata összegyűlt, a szerver hívjon meg egy külső kiértékelő szolgáltatást, amely a felhasználótól gyűjtött adatokból biometrikus profilt készít.

RS5 Amennyiben egy felhasználónak elegendő mozgásadata összegyűlt egy adatfolyamban és rendelkezik biometrikus profillal, a szerver hívjon meg egy külső kiértékelő szolgáltatást, amely a felhasználótól gyűjtött adatok alapján verifikálja a felhasználót.

RS6 A verifikációs kérés minden új adatcsomag (chunk) beérkezésekor fusson le.

RS7 A szerver tárolja adatbázisban a biometrikus profilokat és a verifikációkat.

RS8 A szerver biztosítson HTTP REST API-t az adatgyűjtő kliens által küldött státusz lekérdezések fogadására és feldolgozására.

RS9 A szerver legyen képes a felhasználó aktuális státuszának megállapítására a rendelkezésre álló információk alapján.

RS10 A szerver biztosítson HTTP REST API-t a webes vékonykliens által küldött felhasználói és egyéb statisztikákat érintő lekérdezések fogadására és feldolgozására.

6.2 Az alkalmazás 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 a követelmények által leírt különböző funkcionalitásokat 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 (*RS2, RS3, RS7*)**

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 tulajdonságaihoz, illetve típus definíciókat.

- **szerializációs / deszerializációs modulok (*RS2, RS3, RS7*)**

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 kontroller modulok (*RS1, RS8, RS10*)**

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 (*RS2, RS3, RS7*)**

Az adatbázis kapcsolat felépítését, a tranzakciók futtatását kezelik.

- **egyéb üzleti logikát megvalósító modulok (*RS1, RS4, RS5, RS6, RS9*)**

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.

6.3 Üzleti logikai réteg

Az üzleti logikai rétegben a követelmények által leírt használati eseteket (use case) kielégítő funkciók kerültek implementálásra. Az egyes használati esetek és ezeket implementáló funkciók magyarázata a következő fejezetekben olvasható.

6.3.1 Események feldolgozása

Az *RS1*-es követelmény értelmében az alkalmazás szervernek képesnek kell lennie fogadni és feldolgozni az adatgyűjtő kliens által küldött eseményeket. A kérést kezelő kontroller 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 (*RS3*).
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 6.3.2-es fejezetben vannak részletezve.
8. HTTP válasz összekészítése és elküldése a kliensnek.

6.3.2 Profil építés és verifikáció

Az *RS4, RS5, RS6* követelmények értelmében az alkalmazás szervernek képesnek kell lennie a felhasználó aktuális állapotától függően egy külső kiértékelő szoftveren keresztül biometrikus profil építésére, illetve verifikáció készítésére. Ehhez alapvetően egy *push* architektúrájú rendszert készítettem. Amikor az adatgyűjtő klienstől egy új adatcsomag (chunk) érkezik a szerverre, az adatok feldolgozása és mentése után (ld. *6.4.1*) a kontroller aszinkron módon meghív egy callback metódust. Ez a metódus elindít egy folyamatot, amely ellenőrzi az adatcsomagot küldő felhasználó állapotát és a rendelkezésre álló adatok alapján hoz egy döntést, hogy milyen akciót kell végrehajtani. A lehetséges akciók a következők:

- **build_profile**

  A felhasználónak biometrikus profilt kell építeni. A szerver ilyenkor elindítja a profil építési folyamatot. Az adatbázisba mentésre kerül egy üres profil. Ezután az adatbázisból lekérdezésre kerülnek a felhasználóhoz tartozó adatcsomagok, majd az alkalmazás szerver elküldi ezeket a kiértékelő szoftvernek. A kiértékelésnek három kimenetele lehet:

  1. A profil építés sikeres.
  2. A profil építés valamilyen hiba miatt sikertelen.
  3. A profil építést indító HTTP kérés kifut az időkorlátból (request timeout).

  Amennyiben a profil építés sikeres volt, a szerver frissíti az adatbázisban korábban létrehozott profilt a profil bináris adattartalmával és a sikeres kiértékelés időbélyegével. Amennyiben sikertelen volt a profil építés a szerver frissíti a korábban létrehozott profilt a sikertelen kiértékelés időbélyegével. Mindkét esetben naplózásra kerül a profilépítés eredménye, amennyiben sikertelen a kapcsolódó hibaüzenettel együtt.

- **verify**

A felhasználót verifikálni kell. A szerver elindítja a verifikációs folyamatot. A szerver lekérdezi az adatbázisból a felhasználó legfrissebb adatfolyamához tartozó adatcsomagok sorszámát (sequence number). Ezután elkezdi sorban felolvasni az adatcsomagokat a legújabbtól kezdve visszafelé haladva, amíg a felolvasott események száma meghaladja a verifikáció elvégzéséhez szükséges minimumot. Amennyiben nincs ennyi esemény az adatfolyamban, a folyamat megszakad és az eset naplózásra kerül. Amennyiben van kellő mennyiségű adat az adatfolyamban az adatbázisban mentésre kerül egy üres verifikáció. Ezután az alkalmazás szerver elküldi a kiértékelő szoftvernek a verifikáció alapját képező eseményeket és a felhasználó legfrissebb profilját. A kiértékelésnek három kimenetele lehet:

1 A verifikáció sikeres.

2 A verifikáció valamilyen hiba miatt sikertelen.

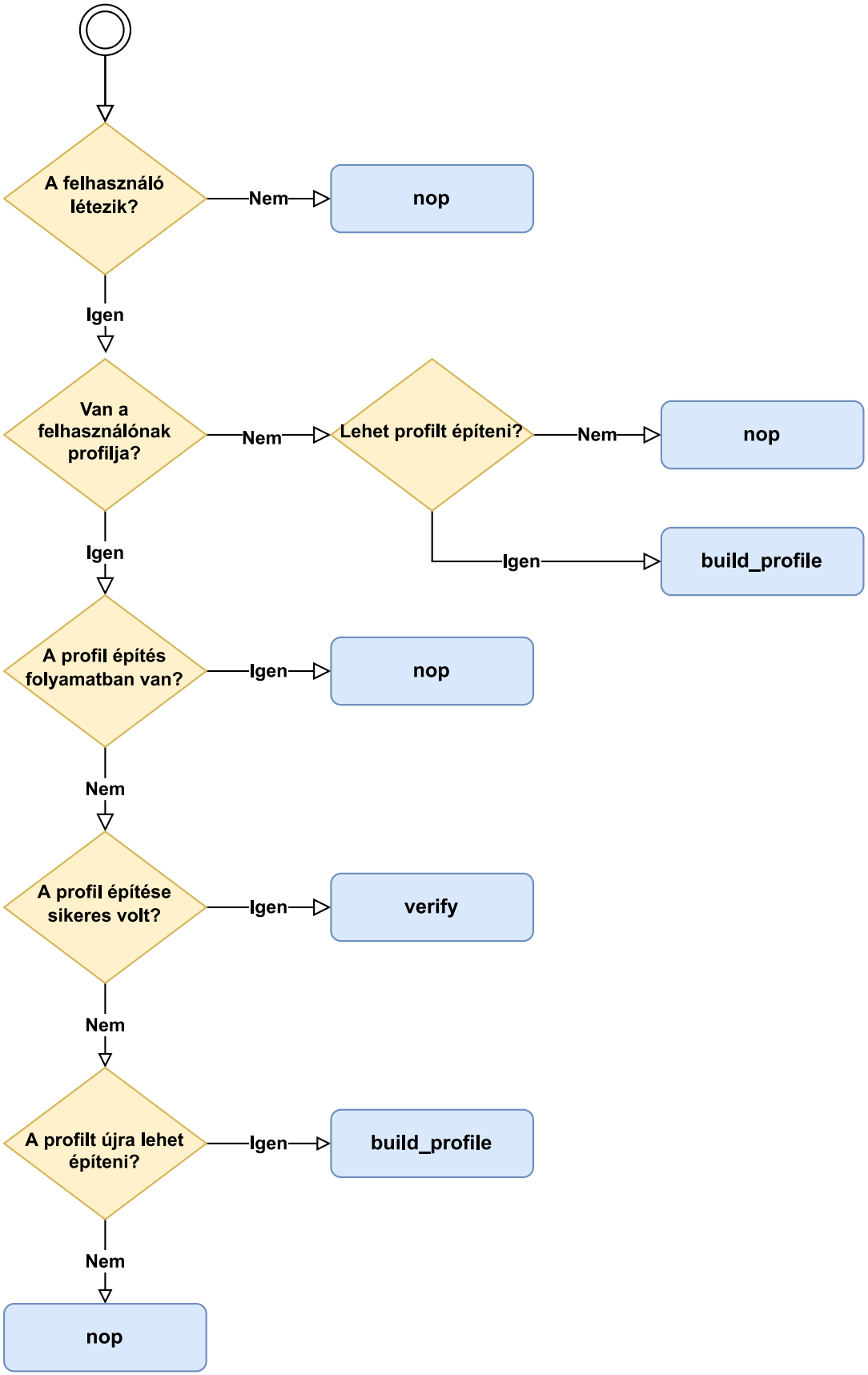
3 A verifikációt indító HTTP kérés kifut az időkorlátból (request timeout).

Amennyiben a verifikáció sikeres volt, a szerver frissíti az adatbázisban korábban létrehozott verifikációt a verifikáció eredményével és a sikeres kiértékelés időbélyegével. Amennyiben a verifikáció sikertelen volt, a szerver frissíti a korábban létrehozott verifikációt a sikertelen kiértékelés időbélyegével. Mindkét esetben naplózásra kerül a verifikációs hívás eredménye, amennyiben sikertelen, a kapcsolódó hibaüzenettel együtt.

- **nop**

  Nem szükséges további akció elvégzése. Az akció hiánya és az indok (amennyiben van) naplózásra kerül.

A végrehajtandó akció kiválasztásának folyamatát a 12. ábra mutatja. Először ellenőrzésre kerül, hogy a folyamatot kiváltó adatcsomaghoz tartozó felhasználó létezik-e. Ha nem, akkor a folyamat megszakad, nincs további teendő. Ha létezik a felhasználó, akkor a hozzá tartozó legfrissebb profil kerül ellenőrzésre. Ha nincs profilja a felhasználónak, de lehet építeni, akkor a **build_profile** akció kerül végrehajtásra. Előfordulhat, hogy bár nincs profilja a felhasználónak, de nem is lehet építeni neki, mert például még nem gyűlt össze elég adat hozzá (learn fázisban van).



12. ábra Az akció kiválasztása

Ha a felhasználóhoz tartozik már profil az adatbázisban, akkor ellenőrizni kell, hogy a profil építés befejeződött-e (vagy csak egy üres profil van bejegyezve). Ha még nem fejeződött be, akkor nincs további teendő. Ha már befejeződött a profil építése, akkor ellenőrizni kell, hogy a profil építése sikeres volt-e. Amennyiben igen, verifikálni kell a felhasználót, a végrehajtott akció a **verify.** Ha nem volt sikeres a profil építés, akkor meg kell nézni, hogy újra lehet-e kezdeni a profil építését. Ehhez tartozik a szerver oldalon egy konfigurálható minimális időkorlát, amelynek el kell telnie két profil építési próbálkozás között. Ha még nem telt el ez az idő, akkor nem lehet újraépíteni a profilt, tehát nincs további teendő. Ha már eltelt elegendő idő, akkor meg lehet próbálni újraépíteni a profilt, a végrehajtandó akció a **build_profile**.

6.3.3 A felhasználó státusza

Az *RS8, RS9* követelmények értelmében az alkalmazás szervernek fel kell tudnia dolgozni az adatgyűjtő kliensről érkező státusz lekérdezéseket. Amikor egy ilyen lekérdezés érkezik a szerverhez, a státusz lekérdezéseket kezelő kontroller modul meghatározza a felhasználó státuszát és válaszol a kliensnek. Az egyes státuszok leírása és magyarázata, illetve a státuszok közti tranzíció az adatgyűjtő kliens dokumentációjában, az 5.4-es fejezetben olvasható.

6.4 HTTP 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. 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.

6.4.1 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(),          // Időbélyeg mértékegysége
             "value": integer()         // Időbélyeg értéke
        },
        "sessionId": string(),          // Munkamenet azonosítója
        "streamId": string(),           // Adatfolyam azonosítója
        "sequenceNumber": integer(),    // Adatcsomag sorszáma
        "userId": string()              // Felhasználói azonosító
   },
   "chunk": array()                     // Események
}
```

- válasz tartalma

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

6.4.2 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

- válasz tartalma

```
{
   "phase": string()              // Az aktuális fázis neve
   "description": string()        // A fázis opcionális leírása
   "value": float()               // A fázishoz tartozó érték
}
```

6.4.3 Statisztikák

A webes vékonykliens alkalmazás különböző statisztikákat jelenít meg a rendszerben található adatok alapján. Az alkalmazás szerver ehhez több különböző HTTP API-t biztosít. Az egyes HTTP lekérdezéseknél több helyen is megjelenik a *threshold* query string paraméter. Ennek a paraméternek megadásával az eredményben visszaadott verifikációkat lehet szűrni. Amennyiben a paraméter jelen van, a szerver csak azokat a verifikációkat adja vissza, amelyeknél a verifikáció eredménye kisebb, mint a megadott érték.

6.4.3.1 Szerver elérhetősége

- végpont

  GET /api/1/state

- válasz tartalma

```
{
   "result": "ok"
}
```

6.4.3.2 Felhasználók

- végpont

  GET /api/1/users?threshold=float()

- válasz tartalma

```
{
   "result": "ok",
   "users": [
     {
         "userId": string(),            // A felhasználó azonosítója
         "eventCount": integer(),       // Események száma
         "createdAt": date(),           // Regisztráció dátuma
         "verifications": integer(),    // Összes verifikáció
         "incidents": integer()         // Sikertelen verifikációk
     }
   ]
}
```

6.4.3.3 Egy specifikus felhasználó

- végpont

  GET /api/1/users/:user_id?threshold=float()

- válasz tartalma

```
{
   "result": "ok",
   "user": {
        "userId": string(),            // A felhasználó azonosítója
        "eventCount": integer(),       // Események száma
        "createdAt": date(),           // Regisztráció dátuma
        "verifications": integer(),    // Összes verifikáció
        "incidents": integer()         // Sikertelen verifikációk
   }
}
```

6.4.3.4 Verifikációk

- végpont

  GET /api/1/verifications?threshold=float()

- válasz tartalma

```
{
   "result": "ok",
   "verifications": [
     {
        "verificationId": string(),    // A verifikáció azonosítója
        "userId": string(),            // A felhasználó azonosítója
        "result": float(),             // A verifikáció eredménye
        "date": date()                 // A verifikáció dátuma
     }
   ]
}
```

6.4.3.5 Egy specifikus felhasználó verifikációi

- végpont

  GET /api/1/verifications/:user_id?threshold=float()

- válasz tartalma

```
{
   "result": "ok",
   "verifications": [
     {
        "verificationId": string(),    // A verifikáció azonosítója
        "userId": string(),            // A felhasználó azonosítója
        "result": float(),             // A verifikáció eredménye
        "date": date()                 // A verifikáció dátuma
     }
   ]
}
```

6.4.3.6 Események

- végpont

  GET /api/1/events

- válasz tartalma

```
{
   "result": "ok",
   "events": [
     {
        "date": date()                  // Dátum
        "eventCount": integer()         // Események száma
     }
   ]
}
```

6.4.3.7 Egy specifikus felhasználó eseményei

- végpont

  GET /api/1/events/:user_id

- válasz tartalma

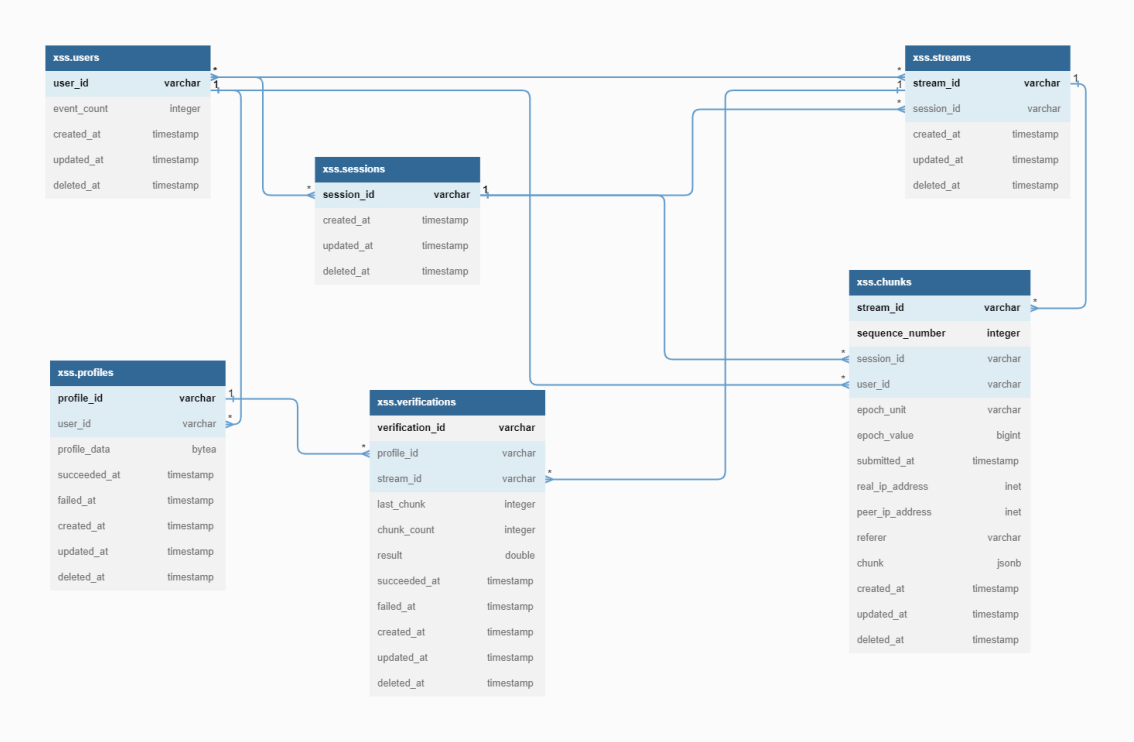
```
{
   "result": "ok",
   "events": [
     {
        "date": date()                  // Dátum
        "eventCount": integer()         // Események száma
     }
   ]
}
```

6.5 Adatbázis

Az alkalmazás szerver mellé egy relációs adatbázist, a PostgreSQL [9] relációs adatbázis-kezelő rendszert 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.

6.5.1 Séma

A 13. ábra az adatbázis sémáját mutatja. 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 diagramon.



13. ábra Az adatbázis sémája

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.

6.5.2 Telepítés

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 séma és a táblák létrehozására SQL nyelven írtam szkripteket, amelyeket közvetlenül az adatbázison egy kliens programmal (például psql [10]) lehet lefuttatni.

6.5.3 Lekérdezések

Az adatbázis lekérdezéseket szintén SQL nyelven írtam meg, viszont a feldolgozásukra használtam egy külső szoftvert (eql [11]), 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 lekérdezésre:

```
-- :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.

6.6 Kiértékelő szerver

Az *RS4*, *RS5* követelmények értelmében az alkalmazás szervernek a biometrikus profil építését és a verifikációs hívásokat egy külső kiértékelő szoftver segítségével kell végrehajtania. Ahogyan azt már korábban a 4. fejezetben írtam, a kiértékelő szerver nem képezi részét a diplomamunkának, egy külső szolgáltatásként kezeli az alkalmazás szerver. A megoldás implementálása során készítettem egy *mock* implementációt a kiértékelő szerverből. Ennek a célja, hogy a tényleges funkcionalitás (profil építés, verifikálás) megvalósítása nélkül imitálja ezeket a folyamatokat, ezzel biztosítva az alkalmazás szerver helyes működését.

Az általam készített implementáció a Node.js és Express.js technológiákat használja. A kiértékelő szerver a következő HTTP API-t nyújtja az alkalmazás szerver számára:

profil építés

- végpont

  POST /profile

- kérés tartalma

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

- válasz tartalma

```
{
   "status": "ok"
   "profileData": string()
}
```

verifikáció

- végpont

  /verify

- kérés tartalma

```
{
   "chunks": [
        {
             "metadata": {
                  "epoch": {
                       "unit": string(),
                       "value": integer(),
                  },
                  "sessionId": string(),
                  "streamId": string(),
                  "sequenceNumber": integer(),
                  "userId": string(),
             },
        "chunk": array(),
        }
   ],
   "profile": {
        "profileId": string(),
        "userId": string(),
        "profileData": string(),
   },
}
```

- válasz tartalma

```
{
   "status": "ok",
   "result": float()
}
```

A kiértékelő szoftver által visszaadott értékeket (például verifikáció eredménye) konfigurálható, futás közben cserélhető módon implementáltam. Továbbá az egyes hívások kezelése tartalmaznak beépített késleltetést, ezzel imitálva a tényleges profil építést és verifikációt (ezek ugyanis időigényes folyamatok).

6.7 Konfigurációs lehetőségek

Ennek a fejezetnek a célja az alkalmazás szerver és a *mock* kiértékelő szerver konfigurációs lehetőségeinek dokumentációja.

6.7.1 Alkalmazás szerver

- **NAME**

  Az alkalmazás neve.

  Alapértelmezett értéke: „**x11_sentinel_server”**

- **PORT**

  A port, amelyen keresztül az alkalmazás elérhető.

  Alapértelmezett értéke: **8081**

- **DB_HOST**

  Az adatbázis *host* neve.

  Alapértelmezett értéke: **„db”**

- **DB_USERNAME**

  Az alkalmazás szerver által használt adatbázis felhasználó neve.

  Alapértelmezett értéke: **„xss”**

- **DB_PASSWORD**

  Az alkalmazás szerver által használt adatbázis felhasználó jelszava.

  Alapértelmezett értéke: **„secret”**

- **DB_PORT**

  A port, amelyen keresztül az adatbázis elérhető.

  Alapértelmezett értéke: **5432**

- **DB_NAME**

  Az alkalmazás szerver által használt adatbázis neve.

  Alapértelmezett értéke: **„xss”**

- **EVALUATION_SERVICE_HOST**

  A kiértékelő szerver *host* neve.

  Alapértelmezett értéke: **„evaluation”**

- **EVALUATION_SERVICE_PORT**

A port, amelyen keresztül a kiértékelő szerver elérhető.

Alapértelmezett értéke: **4000**

- **MINIMUM_ELAPSED_TIME_FOR_FAILED_PROFILE_REBUILD**

  Az a minimális idő mennyiség milliszekundumban megadva, amennyi idő után egy sikertelen profil újraépíthető.

  Alapértelmezett értéke: **3600000** (1 óra)

- **MINIMUM_EVENT_COUNT_FOR_VERIFICATION**

  Az a minimális eseményszám, amennyi szükséges egy verifikáció futtatásához.

  Alapértelmezett értéke: **7200**

- **MINIMUM_EVENT_COUNT_FOR_PROFILE**

  Az a minimális eseményszám, amennyi szükséges egy profil építés futtatásához.

  Alapértelmezett értéke: **1000000**

6.7.2 Kiértékelő szerver

- **EVALUATION_PORT**

  A port, amelyen keresztül a kiértékelő szerver HTTP kéréseket fogad.

  Alapértelmezett értéke: **4000**

- **EVALUATION_PROFILE_BUILD_WAIT_TIME**

  Az a késleltetés milliszekundumban megadva, amennyi időt a kiértékelő szerver vár egy profil építési kérésre való válaszadás előtt.

  Alapértelmezett értéke: **60000 (60 másodperc)**

- **EVALUATION_RESPONSE_DIR**

  Annak a mappának az útvonala, ahonnan a kiértékelő szerver felolvassa profil építési és verifikációs kérésekre adott HTTP válaszok tartalmát (HTTP response body).

  Alapértelmezett értéke: **/srv/data**

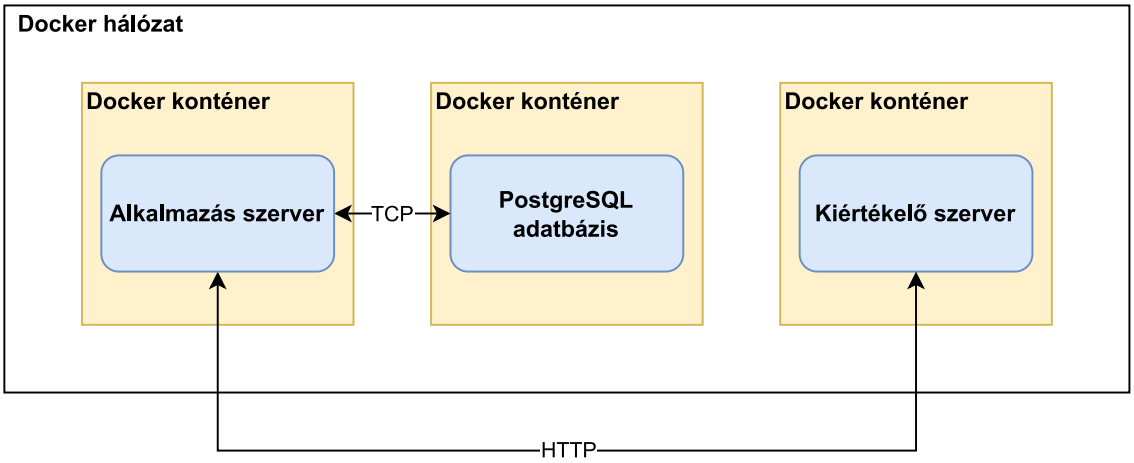
- **EVALUATION_VERIFY_WAIT_TIME**

  Az a késleltetés milliszekundumban megadva, amennyi időt a kiértékelő szerver vár egy verifikációs kérésre való válaszadás előtt.

  Alapértelmezett értéke: **5000 (5 másodperc)**

6.8 Telepítés

Az alkalmazás szerver, a PostgreSQL adatbázis és a *mock* kiértékelő szerver telepítéséhez Docker, illetve docker-compose támogatást implementáltam. Ahogyan azt a 14. ábra is mutatja, az egyes szolgáltatások saját Docker konténerben kerülnek telepítésre, egymástól elkülönítve, de közös Docker hálózaton futnak. Az alkalmazás szerver az adatbázissal TCP protokollon keresztül kommunikál. Mivel ez a kommunikáció a Docker hálózaton belül megy végbe, ezért az adatbázisnak nem szükséges a hálózaton kívülről érkező kérések fogadására nyitott portot biztosítania. Ennek elsősorban biztonságtechnikai jelentősége van. Az alkalmazás szerver bár a kiértékelő szerverrel egy Docker hálózatban kerül telepítésre, a két szolgáltatás egymással a hálózaton kívül, HTTP protokollon keresztül kommunikál. Ezt azért így valósítottam meg, hogy az általam készített alkalmazás szerver könnyen hozzáilleszthető legyen a valós funkciókkal rendelkező kiértékelő szerverhez.



14. ábra A szolgáltatások telepítése

A telepítés során Docker volume-okat is használnak a szolgáltatások. A konténerek között a volume-ok nem kerülnek megosztásra, szeparáltan működnek. Két volume jön létre a szolgáltatások docker-compose-on keresztül történő elindításakor. Az egyiket az adatbázis használja az adatbázis fájlok tárolására. A másikat a kiértékelő

szerver használja, ide kerülnek azok a JSON fájlok, amelyeket az egyes kérésekre küldött válasz törzsét alkotják.

A szolgáltatások telepítéséhez a következő lépésekre van szükség:

1. Docker image-ek készítése.
2. Docker hálózat létrehozása.
3. Docker konténerek elindítása.
4. Az adatbázis migrációs szkriptek futtatása, például:

```
$ docker exec -it x11-sentinel-server_db_1 \
    psql -h localhost -p 5432 -U postgres \
      -f migration-scripts/01_initialize_db_UP.sql
```

5. JSON fájlok másolása a kiértékelő szerver által használt Docker volume-ra.

Az egyes lépések részletes dokumentációja a futtatandó kódrészletekkel a projekt README.md fájljában kerültek dokumentálásra.

7 Webes vékonykliens

A következő fejezetnek a célja a webes vékonykliens program dokumentációja. A leírásban először ismertetem a kliens programmal szemben felállított követelményeket, majd az alkalmazás felépítését. Ezt követően a különböző nézetekről és az azokon megjelenített statisztikákról lesz szó. A fejezetet a konfigurációs lehetőségek bemutatásával és a telepítéssel kapcsolatos tudnivalókkal zárom.

7.1 Követelmények

RW1 A webes vékonykliens alkalmazás legyen elérhető egy tetszőleges modern böngészőn keresztül.

RW2 Az alkalmazás felülete legyen reszponzív, támogasson különböző képernyőméreteket. A legkisebb támogatott felbontás a HD (1280x720 pixel).

RW3 Az alkalmazás jelenítse meg az alkalmazás szerver állapotát (elérhető-e).

RW4 Az alkalmazás jelenítsen meg összesített statisztikákat a rendszerben található adatok alapján.

RW5 A felületen legyen lehetőség felhasználók keresésére.

RW6 Az alkalmazás egy felhasználó kiválasztása után jelenítsen meg az adott felhasználóhoz tartozó specifikus statisztikákat.

RW7 Az egyes nézetek esetén a statisztikák bizonyos időközönként automatikusan kerüljenek frissítésre.

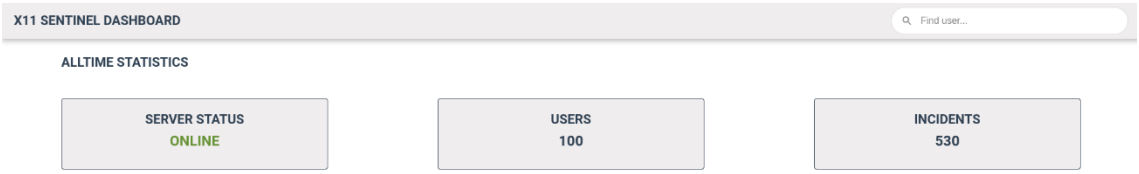
7.2 Az alkalmazás felépítése

A webes vékonykliens alapja egy React alkalmazás, amely az üzleti logikát és a különböző komponensek felépítését (HTML törzsét) nyújtja Az alkalmazásnak két nézete van. Az egyiken a szerver állapotával (RW3) kapcsolatos információt és összesített statisztikákat (RW4) lehet megtekinteni. Az oldal fejlécében található egy kereső, amellyel felhasználókra lehet szűrni (RW5). A felhasználó azonosítójára kattintva az oldal egy új nézetre navigál, amelyen az adott felhasználóhoz tartozó specifikus tartalmat lehet megtekinteni (RW6). Az egyes felületek dizájn elemeinek

implementálásához és a tartalom elrendezéséhez a *Sass* [12] és a *Bootstrap* [13] technológiákat használtam (RW2). A statisztikák lekérdezésére az alkalmazás szerver egy HTTP JSON API-t nyújt (ld. *6.4.3*). A React alkalmazás a nézethez szükséges adatokat konfigurálható időközönként periodikusan lekérdezi (RW7) és újra rendereli a nézethez tartozó komponenseket.

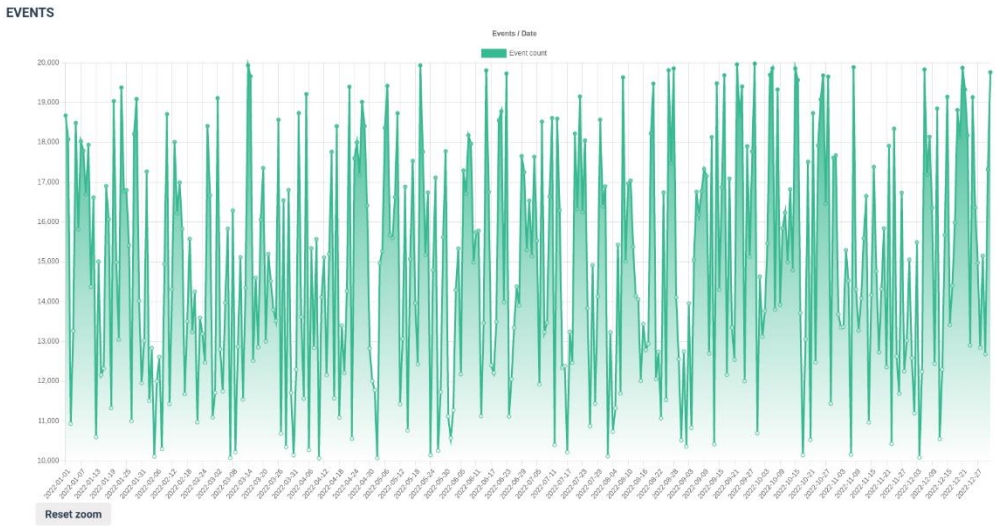
7.3 Összesített statisztikák

A weboldal betöltésekor az első nézet, ami a felhasználót fogadja az összesített statisztikákat mutatja. Az oldal tetején három panel található. Ezek a szerver elérhetőségét (online / offline), a regisztrált felhasználók számát, illetve az incidensek számát mutatják.



15. ábra Összesített statisztikák

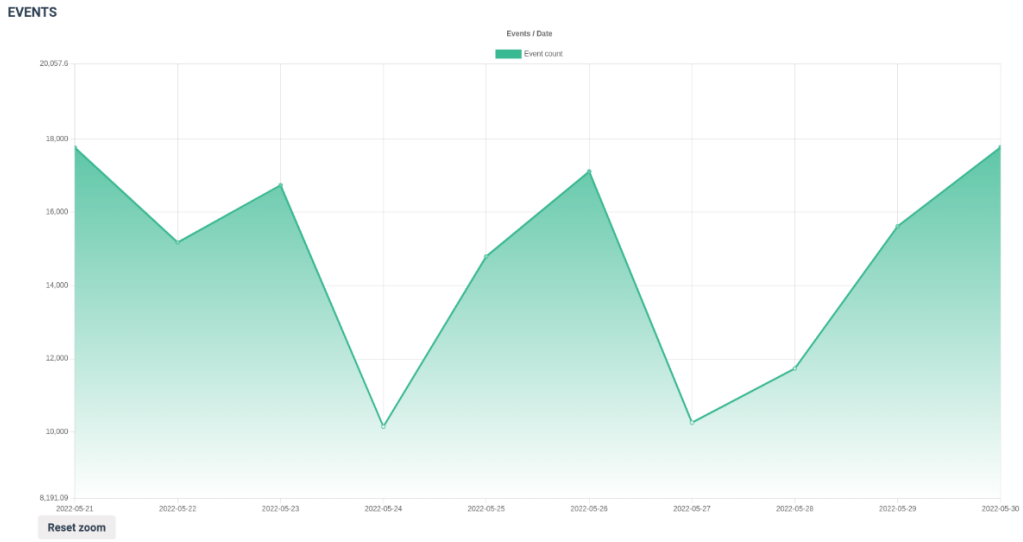
Az oldalon lejjebb görgetve az eseményekkel kapcsolatos grafikont tekinthet meg a felhasználó.



16. ábra Események a dátumok függvényében

A grafikon X tengelyén dátumok találhatóak, az Y tengelyen pedig az adott dátumon gyűjtött összes felhasználói esemény száma. A grafikont a *chart.js* [14] JavaScript függvénykönyvtárral felhasználásával implementáltam. A grafikon

interaktív, a bal egérgombot lenyomva ki lehet jelölni egy területet, amelyet a program kinagyít. A 17. ábra egy 10 napos intervallumra való közelítést mutat. Miután a felhasználó ráközelít a grafikon egy részére, a *Control* billentyű nyomva tartása mellett az egérrel léptetni lehet a grafikont az X tengely mentén.



17. ábra Esemény grafikon ráközelített módban

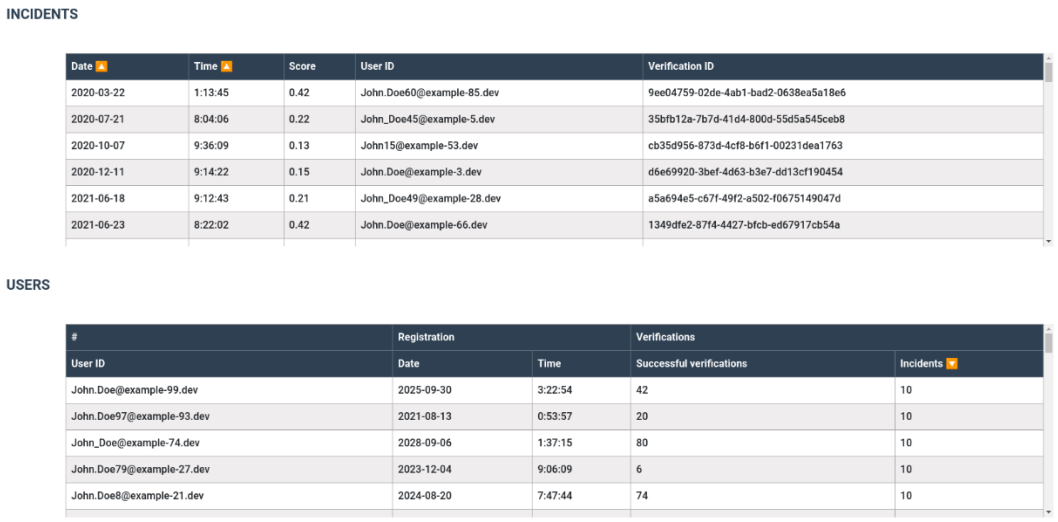
Az eseményekkel kapcsolatos információk alatt két táblázat helyezkedik el. Az első táblázatban az incidensekkel kapcsolatos adatok találhatóak:

- **Date:** Az incidens dátuma.
- **Time:** Az incidens ideje.
- **Score:** Az incidens verifikációs pontja.
- **User ID:** Az incidenshez tartozó felhasználó azonosítója.
- **Verification ID:** Az incidens azonosítója.

A második táblázatban a felhasználókkal kapcsolatos adatok vannak:

- **User ID:** A felhasználó azonosítója.
- **Date:** A felhasználó regisztrációjának dátuma.
- **Time:** A felhasználó regisztrációjának ideje.
- **Succesful verifications:** A felhasználó sikeres verifikációinak száma.
- **Incidents:** A felhasználó incidenseinek száma.

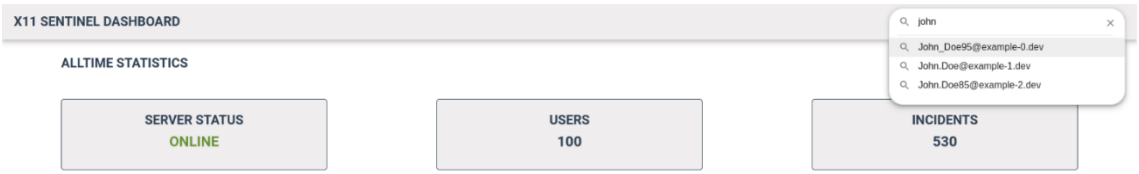
A táblázatokat a *react-table* [15] függvénykönyvtár használatával implementáltam. Mindkét táblázat sorai sorba rendezhetőek az egyes oszlopok értékei szerint, az adott oszlop nevére kattintva a bal egérgombbal. A táblázat sorai a *Shift* billentyűt nyomva tartva több oszlop szerint is rendezhetőek.



18. ábra Összesített statisztikákat tartalmazó táblázatok

7.4 Felhasználó-specifikus statisztikák

A weboldal fejlécében található egy keresőmező, amely segítségével felhasználókra lehet keresni az azonosítójuk alapján. A keresőmező automatikus kiegészítéssel van ellátva, azaz ahogy a felhasználó elkezdi begépelni a keresett kifejezést, a keresőmező felajánlja a legjobban illeszkedő találatokat. A keresőmezőben egy felhasználó azonosítójára kattintva az oldalt nézetet vált és megjeleníti a felhasználó-specifikus statisztikákat.

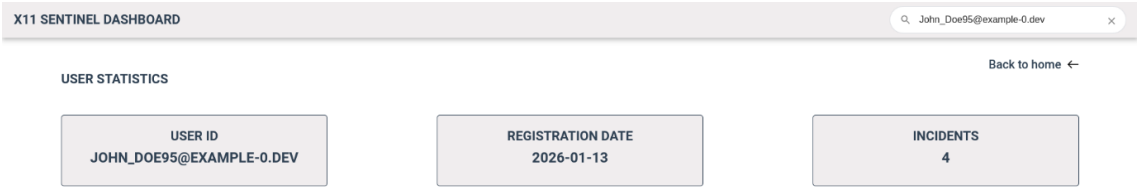


19. ábra A keresőmező működés közben

A felhasználó-specifikus nézet tetején három panel található. Ezek a következő értékeket mutatják:

- A felhasználó azonosítója.
- A felhasználó regisztrációjának dátuma.

- A felhasználó incidenseinek száma.

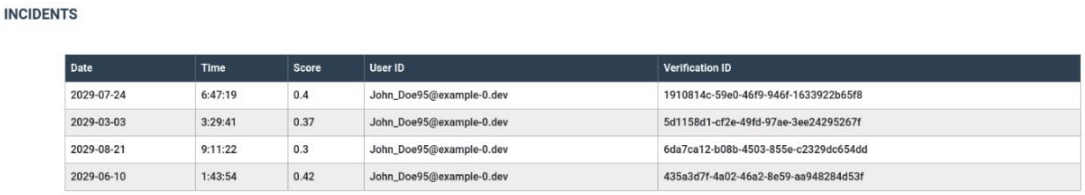


20. ábra Felhasználó-specifikus statisztikák

Az oldal jobb felső sarkában található egy *Back to home* feliratú gomb, amelyre kattintva a felhasználó visszajut az összesített statisztikákat mutató nézetre. Az oldalon lejjebb görgetve egy hasonló grafikon található, mint az összesített statisztikákat mutató nézeten. A grafikon X tengelyén dátumok találhatóak, az Y tengelyen pedig a felhasználó által gyűjtött eseményszám. Ez a grafikon is interaktív, nagyítható és az X tengely mentén léptethető. Az oldal alján egy táblázat található, amelyben a felhasználó incidensei vannak listázva. A táblázat a következő oszlopokkal rendelkezik:

- **Date:** Az incidens dátuma.
- **Time:** Az incidens ideje.
- **Score:** Az incidens verifikációs pontja.
- **User ID:** A felhasználó azonosítója.
- **Verification ID:** Az incidens azonosítója.

Az összesített statisztikákat mutató nézeten található táblázatokhoz hasonlóan ez a táblázat is rendezhető az egyes oszlopokban található értékek szerint.



21. ábra A felhasználó incidensei

7.5 Konfigurációs lehetőségek

Mivel a webes vékonykliens alkalmazás a felhasználó böngészőjében fut, ezért az alkalmazás szerverrel és az adatgyűjtő klienssel ellentétben a konfigurációs paraméterek nem futási időben kerülnek kiolvasásra, hanem már a webes frontend által

kiszolgált állomány készítésekor (bundling) rendelkezésre kell állniuk (build time variables). A konfigurációs paraméterek továbbra is megadhatóak környezeti változókként, ezeket a *react-scripts* függőség a *bundling* folyamat részeként feldolgozza és behelyettesíti azokra a helyekre, ahol a forráskód környezeti változókra hivatkozik. A webes vékonykliens a következő konfigurációs paraméterekkel rendelkezik:

- **APP_SERVER_PORT**

  A port, amelyen keresztül a kliens alkalmazás elérhető.

  Alapértelmezett értéke: **3000**

- **REACT_APP_TEST**

  Ha a paraméter értéke "**true**", akkor az alkalmazás *bundling* folyamata teszt profillal kerül elvégzésre. A webes vékonykliens teszt profiljáról bővebben a 8.3-as fejezetben lesz szó.

  Alapértelmezett értéke: **false**

- **REACT_APP_SENTINEL_SERVER_URL**

  Az az URL, amin keresztül az alkalmazás szerver által nyújtott HTTP API elérhető.

  Alapértelmezett értéke: **http://localhost:8084/api/1**

- **REACT_APP_VERIFICATION_THRESHOLD**

  Egy 0.0 és 1.0 közti lebegőpontos szám. Ha egy verifikáció pontja ez alatt a küszöbérték alatt van, akkor incidensnek minősül.

  Alapértelmezett értéke: **0.5**

- **REACT_APP_QUERY_INTERVAL**

  A statisztikák lekérdezése periodikusan történik. Ez a konfigurációs paraméter azt adja meg, hogy két lekérdezés között mennyi idő teljen el milliszekundumban.

  Alapértelmezett értéke: **5000 (5 másodperc)**

- **REACT_APP_TABLE_QUERY_INTERVAL**

Az oldalon található táblázatok adatainak lekérdezése periodikusan történik. Ez a konfigurációs paraméter azt adja meg, hogy a két lekérdezés között mennyi idő teljen el milliszekundumban. Ez az érték tipikusan nagyobb, mint a többi statisztika lekérdezési periódusa. Ezt az magyarázza, hogy míg egy grafikonnál indokolt a gyakori frissítés, egy táblázat esetén ez zavaró lehet (például a táblázat sorainak rendezése a táblázat frissítésével visszaáll az alapértelmezett helyzetbe).

Alapértelmezett értéke: **60000 (1 perc)**

7.6 Telepítés

A webes vékonykliens alkalmazás telepítéséhez Docker, illetve docker-compose támogatást implementáltam. Az alkalmazás *bundling* folyamata a *react-scripts* függőség segítségével kerül elvégzésre. Az összekészített csomag egy Docker konténerben kerül telepítésre, a fájlokat egy *nginx* [16] frontend szolgálja ki (RW1). A Docker image létrehozásához (lokális környezetet feltételező konfigurációs értékekkel) a következő parancs kiadása szükséges:

```
$ docker-compose --env-file env.local -f docker-compose.yml build
```

A Docker image alapján készült konténert a következő paranccsal lehet elindítani:

```
$ docker-compose --env-file env.local -f docker-compose.yml up -d
```

8 A megoldás tesztelése

A következő fejezetnek a célja az elkészült megoldást felépítő alkalmazások tesztelésének dokumentációja. Az egyes alkalmazások a betöltött funkciójukból adódóan meglehetősen különböznek egymástól, így nem meglepő módon a tesztelés módszertana is alkalmazásonként különböző. A fejezetben bemutatom az egyes alkalmazásokon elvégzett teszteket, dokumentálom a tesztek eredményét és megindokolom a felmerülő tervezői döntéseket.

8.1 Linux kliens alkalmazás

A Linux kliens alkalmazás tesztelése során, az adatgyűjtő funkció helyes működésének validációjára fókuszáltam. Két szempontot tartottam kiemelkedően fontosnak, az egyik a különböző beviteli eszközök által generált események helyes feldolgozása, a másik a több monitoros környezetek esetén gyűjtött adat validációja volt.

A beviteli eszközök vizsgálata során a következő paraméterekkel rendelkező tesztkörnyezetet használtam:

- modell: Lenovo ThinkPad T570
- operációs rendszer: Ubuntu 22.04.1 LTS (Jammy Jellyfish)
- asztali környezet: GNOME on XOrg
- beviteli eszközök:
    - standard USB 3.0 vezetékes egér
    - standard USB 3.0 vezeték nélküli egér
    - beépített touchpad
    - beépített TrackPoint
    - beépített érintőképernyő

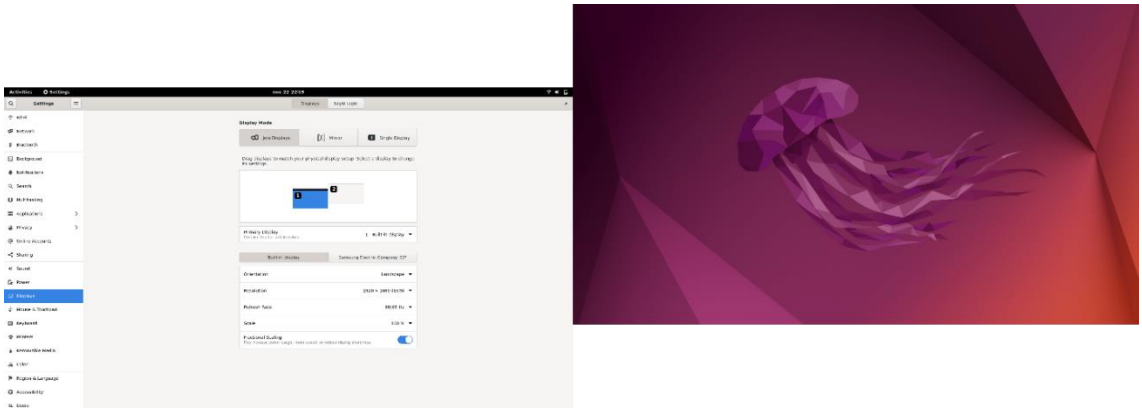
A vizsgált beviteli eszközök mellett az adatgyűjtő helyesen működött, a generált események (kurzormozgás, görgetés, gombnyomás stb.) helyesen kerültek rögzítésre. Az eszközök szimultán használata sem okozott gondot, illetve a „plug-n-play” használat

is támogatott. Azaz egy új eszköz csatlakoztatásakor nem kell az adatgyűjtő szoftvert újraindítani.

Bár a kiértékelő szoftver alapvetően a nyers egérmozgás eseményekből (elmozdulás vektorokból) dolgozik, diagnosztikai céllal a kurzor helyzete (X és Y koordinátája) is elküldésre kerül. A kliens tesztelése során szerettem volna meggyőződni arról, hogy a kurzor helyzetének meghatározása több monitoros környezetben, speciális monitor elrendezések mellett is helyesen működik. Ezt két módszerrel vizsgáltam, először a következő tesztkörnyezetben:

- modell: Lenovo ThinkPad T570
- operációs rendszer: Ubuntu 22.04.1 LTS (Jammy Jellyfish)
- asztali környezet: GNOME on XOrg
- külső monitor: Samsung 22” LCD monitor (S22F350, HDMI)

Amit vizsgáltam a monitorok sorrendje, a külső monitor Y irányú pozitív és negatív eltolása, illetve a projekciós beállítások (kivetítés, tükrözés, csak beépített monitor, csak külső monitor) voltak. A teszteket ezen halmazok Descartes szorzatán végeztem, azaz összesen 16 különböző kombinációt próbáltam ki. Az adatgyűjtő által küldött X és Y koordináták (illetve monitor metaadatok) minden esetben helyesnek bizonyultak.



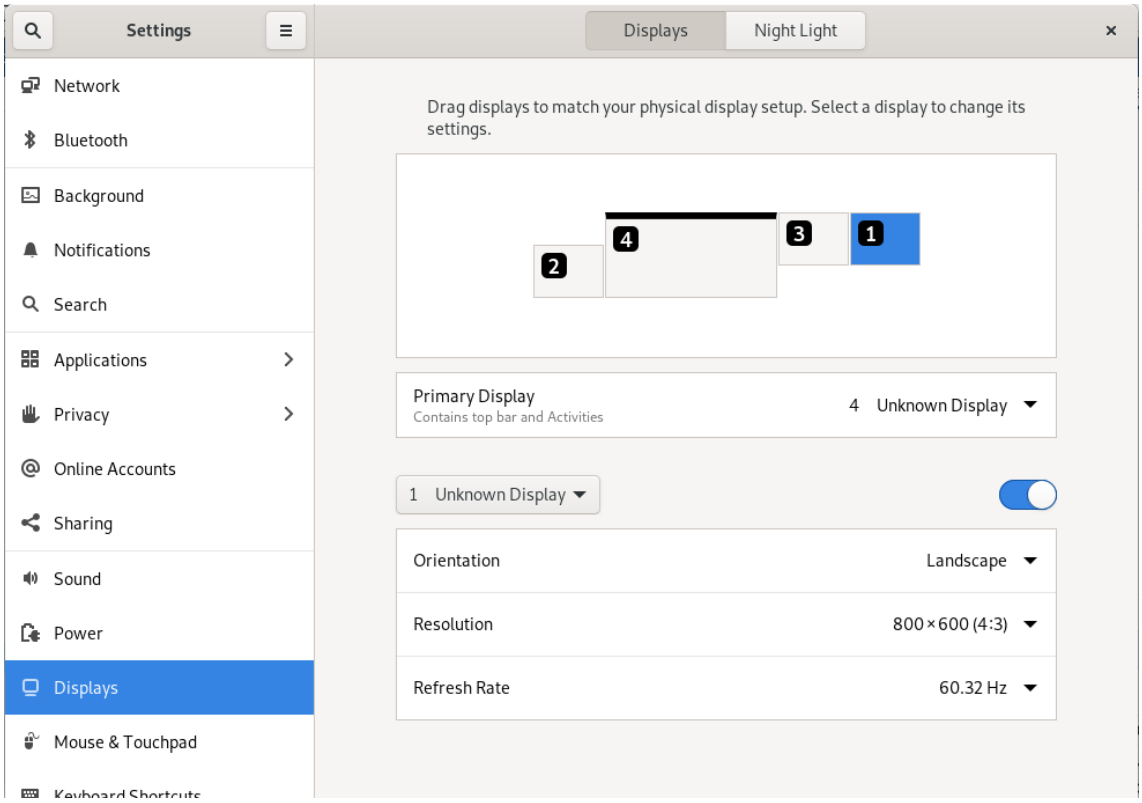
22. ábra Egy lehetséges monitor elrendezés

A másik eset, amelyre kíváncsi voltam, az olyan elrendezések, amelyek több mint egy külső monitort használnak. Mivel ezt fizikailag nem tudtam kivitelezni, ezért virtuális környezetben teszteltem. A virtuális tesztkörnyezet paraméterei:

- virtualizációs szoftver: Oracle VM VirtualBox
- operációs rendszer: Debian 11.5 (Bullseye)

- virtuális monitorok száma: 4
- projekciós beállítás: kivetítés

A monitorokat egy tetszőleges sorrendben elrendezve vizsgáltam az adatgyűjtő által regisztrált kurzor koordinátákat. A gyűjtött események az előzetes várakozásaimnak megfeleltek, nem tapasztaltam váratlan viselkedést a tesztek során.



23. ábra Virtuális monitorok

8.2 Alkalmazás szerver

Az alkalmazás szerver tesztelése során az volt a célom, hogy megbizonyosodjak az Erlang alkalmazás helyes működéséről. Azaz a hangsúly az üzleti logika validációján volt. Az alkalmazás szerver teszt módban történő telepítéséhez készítettem egy docker-compose konfigurációt felülíró teszt fájlt (*docker-compose.override.test.yml*), illetve a projekt könyvtárban megtalálható a tesztelés során használandó konfigurációs értékeket tartalmazó fájl is (*env.test).* A teszt profillal telepített alkalmazásban két lényeges eltérés van (a konfigurációs paraméterek értékén kívül). Az egyik, hogy az adatbázis konténer portja nyitott (*exposed),* így képes a Docker hálózaton kívüli kapcsolatok fogadására is. Erre azért van szükség, hogy a teszteket az alkalmazás szerver telepítése nélkül is lehessen futtatni. A másik különbség, hogy az alkalmazás szerver, az adatbázis és a

kiértékelő szerver mellett telepítésre kerül egy felhasználó felülettel rendelkező adatbázis adminisztrációs szoftver (pgAdmin [17]) is. Ez arra használtam, hogy az adatbázis belső állapotát kényelmesen, egy webes felületről tudjam vizsgálni.

A tesztek futtatásához Makefile támogatást készítettem, az összes tesztet a következő paranccsal lehet futtatni:

```
$ make test
```

Ennek a parancsnak a hatására a következő dolgok történnek sorban:

1. Dokumentáció generálása a forráskódban található kommentek alapján.
2. Futtatható állomány készítése (build).
3. Xref statikus analízis eszköz futtatása.
4. Dialyzer statikus analízis eszköz futtatása.
5. Unit tesztek futtatása.
6. Integrációs tesztek futtatása
7. Tesztlefedettség riport generálása.

Az Xref egy statikus analízis eszköz, ami kereszthivatkozásokat vizsgál. Ezt úgy teszi, hogy függőségeket keres funkciók, modulok és alkalmazások között, a definiált függvények és a függvényhívások elemzésével. A használatával például nem használt függvényeket lehet felderíteni. A Dialyzer szintén egy statikus analízis eszköz Erlang kód vizsgálatára. Ennek az eszköznek a segítségével típus hibákat, elérhetetlen kód részleteket vagy akár felesleges teszteseteket lehet felderíteni.

A Unit tesztek implementálására az Erlang környezet által biztosított EUnit könyvtárt használtam. Ezek a tesztek a modulokon belül kerülnek definiálásra és egy-egy funkcionalitást önmagukban vizsgálnak. Unit teszteket az *xss_utils* modul tesztelésére készítettem, amely egy olyan segédmodul, ami a többi modul számára nyújt különböző funkcionalitásokat. Ilyen funkciók például a Unix időbélyeg és az adatbázis által használt dátumformátum közti átalakítás, vagy az Erlang objektumok kulcsainak snake_case és camelCase közti transzformációja.

Az integrációs teszteket a forráskódtól elválasztva egy külön modulban implementáltam. Ehhez a Common Test keretrendszert használtam. Az integrációs tesztek futtatásakor a következő lépések zajlanak le:

1. Tesztcsomag (test suite) előtti inicializáló lépések futtatása.

   Itt kerül beállításra a tesztek során alkalmazott naplózási mechanizmus, a környezeti változók beállítás és az Erlang alkalmazások elindítása.

2. Teszteset (test case) előtti inicializáló lépések futtatása.

   Az egyes tesztesetek előtti specifikus beállítások és feladatok elvégzése. Például, ha egy teszteset előtt mock objektum létrehozása szükséges, akkor az ebben a lépésben kerül inicializálásra.

3. Teszteset futtatása.

   A tesztesetben implementált kód futtatása, ellenőrzések kiértékelése.

4. Teszteset utáni takarító lépések futtatása.

   Például az inicializáló lépésben létrehozott mock objektum törlése, a teszteset során létrehozott adatbázis bejegyzések eltávolítása stb.

5. Ha van még teszteset a tesztcsomagban, akkor folytatás a 2-es lépéstől.

6. Tesztcsomag utáni takarító lépések futtatása.

7. Ha van még tesztcsomag, akkor, akkor folytatás az 1-es lépéstől.

Az integrációs teszteket egy tesztcsomagban készítettem el, amely 2 egyszerűbb és 8 komplex tesztesetet tartalmaz. Az integrációs tesztek és a Unit tesztek lefedettsége modulokra bontva az alábbi táblázatban látható:

module	coverage
x11_sentinel_server_app	100%
x11_sentinel_server_sup	100%
xss_api_server	45%
xss_chunk	58%
xss_chunk_store	73%
xss_dashboard_api	75%
xss_database_server	80%
xss_events_rest_handler	93%
xss_profile	75%
xss_profile_store	100%
xss_session	25%
xss_session_store	100%
xss_state_rest_handler	88%
xss_status_rest_handler	92%
xss_stream	40%
xss_stream_store	90%
xss_submission_rest_handler	90%
xss_user	80%

```
|                  xss_user_store  |       92%  |
|          xss_users_rest_handler  |       96%  |
|                      xss_utils  |       91%  |
|                xss_verification  |       36%  |
|          xss_verification_store  |       94%  |
|  xss_verifications_rest_handler  |       94%  |
|----------------------------------|------------|
|                            total  |       78%  |
|----------------------------------|------------|
```

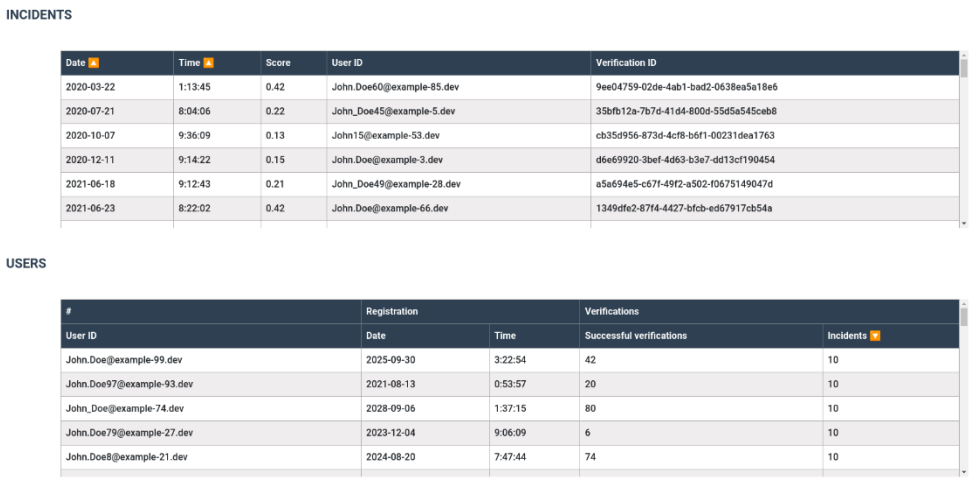
Amennyiben a lefedettség számításakor nem vesszük figyelembe azokat a modulokat, amelyek nem tartalmaznak üzleti logikát (például modellek), akkor az összesített lefedettség **80%**-ra emelkedik:

```
|----------------------------------|------------|
|                           module  |  coverage  |
|----------------------------------|------------|
|         x11_sentinel_server_app  |      100%  |
|         x11_sentinel_server_sup  |      100%  |
|                  xss_api_server  |       45%  |
|                 xss_chunk_store  |       73%  |
|              xss_dashboard_api  |       75%  |
|             xss_database_server  |       80%  |
|         xss_events_rest_handler  |       93%  |
|               xss_profile_store  |      100%  |
|               xss_session_store  |      100%  |
|          xss_state_rest_handler  |       88%  |
|         xss_status_rest_handler  |       92%  |
|                xss_stream_store  |       90%  |
|     xss_submission_rest_handler  |       90%  |
|                  xss_user_store  |       92%  |
|          xss_users_rest_handler  |       96%  |
|                      xss_utils  |       91%  |
|          xss_verification_store  |       94%  |
|  xss_verifications_rest_handler  |       94%  |
|----------------------------------|------------|
|                            total  |       80%  |
|----------------------------------|------------|
```

Az egyetlen kiugróan alacsony lefedettség (45%) az *xss_api_server* modulnál látható. Ez egy általános szervermodul (gen_server behaviour), amely interfészt biztosít a többi modul számára olyan feladatokhoz, amelyeket aszinkron módon kell elvégezni. Az alacsony tesztlefedettséget az magyarázza, hogy ez a modul tartalmazza a külső kiértékelő szerver felé indított profil építési és verifikációs hívásokat. Ezek a hívások pedig mockolva vannak az integrációs tesztek során, többek közt azért, hogy a teszteket egy külső szolgáltatás telepítése nélkül lehessen futtatni.

8.3 Webes vékonykliens

A webes vékonykliens tesztelése során a grafikonok, táblázatok, illetve további statisztikákat megjelenítő komponensek kinézetére és a tartalom rendezésére koncentráltam. A projekt teszt módban történő telepítéséhez szükséges konfigurációs értékeket a projekt könyvtárban található *env.test* fájl tartalmazza. Amennyiben a *REACT_APP_TEST* változó definiálva van („true” értékkel), a kliens alkalmazás az alkalmazás szervertől lekérdezett adatok helyett a tesztelés céljára készített teszt adathalmazt használja.



24. ábra Tesztadatokat tartalmazó táblázatok

A tesztadatok generálása során törekedtem arra, hogy az eredmény minél inkább hasonlítson a valós adatokra. A tesztadat készítésére a faker.js [18] külső JavaScript könyvtárat használtam. A függvénykönyvtár segítségével egyszerűen lehet nagy mennyiségű realisztikus teszt adatot generálni. Alább látható egy tesztelésre szánt felhasználó adatait előállító kód részlet:

```
function makeUser(userId) {
  return {
    userId,
    eventCount: faker.datatype.number({ max: 1000000 }),
    createdAt: faker.date.between(
                '2020-01-01T00:00:00.000Z',
                '2030-01-01T00:00:00.000Z',
              ).toISOString(),
    verifications: faker.datatype.number({ min: 0, max: 100 }),
    incidents: faker.datatype.number({ min:0, max: 10 }),
  }
}
```

9 Összefoglalás

- A diplomamunka keretein belül megismerkedtem a Unix-szerű rendszerek grafikus felhasználói felületének felépítésével, az elterjedt megjelenítő protokollokkal és szerverekkel.
- Elkészítettem az adatgyűjtő alkalmazás prototípusát a Wayland protokollhoz Python nyelven, amely során értékes tapasztalatokat szereztem.
- Elsajátítottam a Rust programozási nyelv alapjait, amelynek használatával implementáltam egy natív Linuxon futó adatgyűjtő és munkamenet zároló kliens alkalmazást.
- Az adatgyűjtő kliens által küldött információ fogadására, feldolgozására, illetve további üzleti logikai folyamatok elvégzésére készítettem egy alkalmazás szervert Erlang nyelven.
- Az alkalmazás szerver mellé az adatok perzisztálása céljából egy relációs adatbázist telepítettem és használtam. Kialakítottam az adatbázis sémát, migrációs szkripteket és lekérdezéseket írtam.
- A rendszerben lévő adatok különböző nézeteinek esztétikus megjelenítésére egy webes vékonykliens alkalmazást készítettem a React keretrendszert felhasználva.
- Az alkalmazásokat konfigurációs támogatással láttam el, ahol célszerű ott többszintes konfigurálási lehetőséggel, akár az alkalmazás futása közben is. A modern konténerizációs technológiák (Docker, docker-compose) használatának köszönhetően a megoldás telepítése egyszerű, nem feltételez nyelv – vagy programozásikörnyezet – specifikus ismeretet.
- Az egyes alkalmazásokat az általam fontosnak tarott funkciók mentén teszteltem, a tesztek eredményét dokumentáltam.

A megítélésem szerint az elkészült megoldás megfelel az előzetes specifikációnak, az egyes alkalmazásokkal szemben támasztott követelményeket az implementáció kielégíti.

9.1 További fejlesztési javaslatok

Alapvetően az elkészült megoldást funkcionalitás szempontjából teljes értékűnek tekintem, mégis (mint általában minden szoftver esetén) felmerülnek további fejlesztési lehetőségek, amelyekkel a jövőben érdemes lehet kibővíteni az implementációt. A Linuxos adatgyűjtő kliens esetén érdemes lenne a HTTP protokoll helyett kipróbálni egy olyan protokollt, ami jobban illeszkedik a küldött adat természetéhez (eseménysor). A WebSocket protokoll például egy jó iránynak tűnik, a használatával csökkenteni lehetne a késleltetést, illetve a HTTP fejlécek által okozott adatforgalom növekedést. Továbbá a nyitott csatornás adatküldés kiváltaná a *chunk* adatmodellt, így egy jól megválasztott adattárolási módszerrel a szerveroldali implementáció is egyszerűsödne.

Egy másik fejlesztési lehetőség a webes vékonyklienst érinti. Egy webes adminisztrációs felület (dashboard) esetén természetesen sokféle funkciót, nézetet el lehet képzelni. Szerintem egy hasznos kiegészítés lenne, ha a felületen a felhasználók platform specifikus metaadatai is megjelennének, illetve ezek mentén lehetne a felhasználókat szűrni. Ezzel hasznos kimutatásokat lehetne készíteni, például, melyik a leggyakrabban használt operációs rendszer, vagy a felhasználók között milyen arányban fordul elő, hogy valaki több mint egy külső monitort használ.

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

[7] „Figma,” [Online]. Available: https://www.figma.com/. [Hozzáférés dátuma: 18 11 2022].

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

[9] „postgresql,” [Online]. Available: https://www.postgresql.org/. [Hozzáférés dátuma: 02 11 2022].

[10] „psql,” [Online]. Available: https://www.postgresql.org/docs/current/app-psql.html. [Hozzáférés dátuma: 08 11 2022].

[11] „eql,” [Online]. Available: https://github.com/artemeff/eql. [Hozzáférés dátuma: 08 11 2022].

[12] „Sass,” [Online]. Available: https://sass-lang.com/. [Hozzáférés dátuma: 18 11 2022].

[13] „Bootstrap,” [Online]. Available: https://getbootstrap.com/. [Hozzáférés dátuma: 18 11 2022].

[14] „chart.js,” [Online]. Available: https://www.chartjs.org/. [Hozzáférés dátuma: 18 11 2022].

[15] „React table,” [Online]. Available: https://www.npmjs.com/package/react-table. [Hozzáférés dátuma: 18 11 2022].

[16] „nginx,” [Online]. Available: https://www.nginx.com/. [Hozzáférés dátuma: 19 11 2022].

[17] „pgAdmin,” [Online]. Available: https://www.pgadmin.org/. [Hozzáférés dátuma: 20 11 2022].

[18] „faker.js,” [Online]. Available: https://fakerjs.dev/. [Hozzáférés dátuma: 24 11 2022].

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

11 Függelék

11.1 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',
  },
}
```