



Niko Ahonen, Niko Ala-aho, Perttu Harvala, Jami Hämäläinen, Riku Koski ja Sylvester Salo

Etäyhteyssovellus

Metropolia Ammattikorkeakoulu

Innovaatioprojektin dokumentaatio

22.5.2024

Sisällys

1	Johdanto	1
2	Teknologiat	1
2.1	WebRTC	1
2.2	React native	1
2.3	WebSocket	2
2.4	Docker	2
3	WebRTC	2
3.1	WebRTC:n käsitteet	2
3.2	WebRTC-yhteyden muodostaminen	3
3.3	Sovelluksen WebRTC-toteutus	4
3.3.1	connectionState-muuttuja	5
3.3.2	Esimerkki useConnection Hookin käytöstä	5
3.3.3	Esimerkki yhteyden muodostamisesta	7
4	Signalointipalvelin	8
4.1	Signalointipalvelimen implementaatio	9
4.1.1	yhdistävän laitteen kriteerit rekisteröitymistä varten	9
4.1.2	Pyynnöt palvelimelle:	9
5	Asennus	11
5.1	Kehitysympäristön asennus	11
5.2	APK:n luominen	12
5.3	Signalointipalvelimen asentaminen dockerin avulla:	12
6	Käyttöohjeet	13
6.1	Sovelluksen käyttäminen	13
6.2	Kielten lisääminen	16
6.3	Sovelluksen avaaminen toisen sovelluksen kautta	17

1 Johdanto

Tämä innovaatioprojektin raportti kuvaa tekemämme työn sisältöä ja sovelluksen toimintatapaa, selittää käyttämiämme teknologioita ja niiden valintaperusteita, avaa tekstissä käytettäviä lyhenteitä sekä ohjeistaa sovelluksen käyttämistä.

Tekemämme sovellus muodostaa helposti käytettävän etäyhteyden hoivakodissa hoitajien ja asukkaiden välille Android-laitteiden välityksellä. Käyttötarkoituksena on varmistaa asukkaiden hyvinvointi hoitajien toimesta sovelluksen avulla ja vähentää turhien huonekäyntien määrää, helpottaen hoitajien työtä sekä tuoden asukkaille enemmän rauhaa. Sovellus suunniteltiin ymmärrettävyyttä, toimivuutta ja helppokäyttöisyyttä etusijalla.

Raportin tavoitteena on saada lukijalle kattava ymmärrys projektin eri teknologioista sekä selkeyttää sovelluksen mahdollisesti monimutkaisia toimintatapoja tekstin, kaavioiden sekä kuvien avulla.

2 Teknologiat

2.1 WebRTC

Valitsimme videopuhelun toteuttamiseen WebRTC-rajapinnan, koska se on käytetyin avoimen lähdekoodin ratkaisu ja siihen löytyy kattava dokumentaatio.

[Linkki WebRTC-rajapinnan dokumentaatioon](#)

2.2 React native

Rakensimme sovelluksen React Nativella, koska WebRTC:n virallista android-kirjastoa ei enää ylläpidetä. React Nativen käyttö myös helpottaa mahdollisen iOS-version kehittämistä.

[Linkki React Native WebRTC -kirjastoon](#)

2.3 WebSocket

Asiakkaan pyynnöstä käytimme sovelluksen ja signaalointipalvelimen väliseen viestintään WebSocket-teknologiaa, joka mahdollistaa IBM cloud enginen tehokkaan käytön ja signaalointipalvelimen toimivuuden palomuurien alla.

2.4 Docker

Docker mahdollistaa helpon paketoinnin konttiin, jossa ohjelma voi pyöriä tehokkaasti ja helposti ilman ulkopuolista häirintää. Signaalointipalvelimen koko on 83mb, kun se on kontitettu ja se käyttää 0,03% Intelin i7-8700k prosessorin tehoista pyyntöjen tullessa. Projektia varten käytettiin Docker-konttitekniologiaa asiakkaan pyynnöstä. Signaalointipalvelimen täytyy pyöriä IBM Cloud Enginessä, joka käyttää Docker imageja pyöriäkseen.

Kun signaalointipalvelin asennetaan IBM code engineen, se ei käytä tyhjäkäynnillä lainkaan resursseja, vaan se käynnistyy, kun siihen yhdistetään.

[Dockerhub repository](#)

3 WebRTC

WebRTC (Web Real-Time Communication) on ohjelmointirajapinta, joka mahdollistaa video- ja ääniviestinnän peer-to-peer -yhteyden avulla.

3.1 WebRTC:n käsitteet

WebRTC-yhteyden muodostamiseen liittyvät seuraavat käsitteet:

- SDP (Session Description Protocol) on protokolla, jolla kuvataan puhelun alustusparametreja. Kuvaukseen kuuluu muun muassa puhe- lussa käytettävät mediat, kuten ääni ja video.

- ICE (Interactive Connectivity Establishment) on protokolla, jonka avulla WebRTC kiertää peer-to-peer -yhteyden muodostamiseen liittyviä esteitä, kuten palomuurit ja NAT (Network Address Translation).
- ICE-kandidaatti on osoite, jonka kautta laitteeseen voi muodostaa yhteyden.
- ICE gathering on prosessi, jonka aikana puhelun molemmat osapuolet selvittävät ICE-palvelinten avulla omat ICE-kandidaattinsa, ja lähettävät ne toiselle osapuolelle selvittääkseen tehokkaimman polun peer-to-peer -yhteyden muodostamiselle.
- ICE-palvelimet auttavat ICE-kandidaattien etsinnässä. Näitä palvelimia on kahdenlaisia: STUN ja TURN.
- STUN (Session Traversal Utilities for NAT) -palvelin kertoo laitteelle sen julkisen IP-osoitteen, vaikka se olisi palomuurin tai NAT:n takana. Jotkin reitittimet käyttävät "Symmetric NAT" -rajoitusta, joka estää yhteyksien muodostamisen uusista osoitteista. Jos laite on tällaisen reitittimen takana, STUN-palvelin ei riitä ja tarvitaan TURN-palvelin.
- TURN (Traversal Using Relays around NAT) kiertää "Symmetric NAT" -rajoituksen avaamalla yhteyden TURN-palvelimen kanssa ja välittämällä kaikki puhelunaikaiset paketit sen kautta vastaanottajalle. Tämä on hitaampaa, kuin suora peer-to-peer -yhteys, joten TURN-palvelinta käytetään vain, jos yhteyttä ei saada muodostettua STUN-palvelimen avulla.

3.2 WebRTC-yhteyden muodostaminen

WebRTC-yhteyttä muodostettaessa kaksi laitetta neuvottelevat keskenään signaalointipalvelimen välityksellä. Neuvottelun elinkaari on seuraavanlainen:

- Laite 1 lähettää SDP (Session Description Protocol) -tarjouksen signaalointipalvelimelle.
- Signaalointipalvelin välittää tarjouksen laitteelle 2.
- Laite 2 lähettää SDP-vastauksen signaalointipalvelimelle.
- Signaalointipalvelin välittää vastauksen laitteelle 1.
- ICE gathering -prosessi alkaa. Molemmat laitteet lähettävät ICE-kandidaatteja toisilleen signaalointipalvelimen välityksellä.
- Kun sopiva ICE-kandidaatti on löydetty, peer-to-peer -yhteys muodostetaan.

3.3 Sovelluksen WebRTC-toteutus

Sovelluksen WebRTC-toteutus koostuu kolmesta React hookista: `useMediaStream`, `useSignalingServer` ja `useConnection`.

`useMediaStream` luo [MediaStream](#)-olion ja lisää siihen laitteen kameran ja mikrofonin.

`useSignalingServer` luo WebSocket-yhteyden signaalintalvelimeen ja käsittelee palvelimelta saadut viestit.

`useConnection` toimii rajapintana muiden hookien ja käyttöliittymän välillä. Se myös luo [RTCPeerConnection](#)-olion, joka kuvastaa kahden laitteen välistä WebRTC-yhteyttä, ja asettaa siihen tarvittavat tapahtumakuuntelijat:

- `negotiationneeded`-tapahtuma laukaistaan, kun `MediaStream`-olio lisätään `RTCPeerConnection`-olioon. Yhteyden neuvottelu alkaa tästä.
- `icecandidate`-tapahtuma laukaistaan, kun uusi ICE-kandidaatti löydyt. Löydetty kandidaatti lähetetään heti signaalintalvelimen kautta vastaanottajalle.
- `connectionstatechange`-tapahtuma laukaistaan aina, kun `RTCPeerConnection`-olion `connectionState`-muuttujan tila vaihtuu. Puhelun päättymisen yhteydessä suoritettavat operaatiot tehdään, kun tämä tila on 'closed'.
- `iceconnectionstatechange`-tapahtuma laukaistaan aina, kun `RTCPeerConnection`-olion `iceConnectionState`-muuttujan tila vaihtuu. Puhelun alkamisen yhteydessä suoritettavat operaatiot tehdään, kun tämä tila on 'completed'.
- `track`-tapahtuma laukaistaan, kun yhteyteen lisätään uusi ääni- tai videoraita. Nämä raidat asetetaan `remoteMediaStream`-olioon.

Lisätietoa näistä näistä, sekä muista mahdollisista tapahtumakuuntelijoista, joita `RTCPeerConnection`-olioon voi lisätä, löytyy [React Native WebRTC:n GitHub-sivulta](#) ja [WebRTC:n dokumentaatiosta](#).

3.3.1 connectionState-muuttuja

Yhteyden tilaa seurataan connectionState-muuttujan avulla. Normaalissa toiminnassa tilat ovat seuraavat:

- 'setting up': Palvelinyhteyden muodostaminen on aloitettu.
- 'connected to server': Yhteys signaalintipalvelimeen on muodostettu.
- 'calling': SDP-tarjous on lähetetty/vastaanotettu, eli neuvottelu on alkanut.
- 'in call': Neuvottelu on valmis ja WebRTC-yhteys on muodostettu.
- 'closed': WebRTC-yhteys on suljettu.
- 'restarting': WebRTC-yhteys on suljettu ja yhteyden muodostaminen aloitetaan alusta. Tämä tila tapahtuu, kun sovelluksen rooli on huone, ja hoitaja katkaisee yhteyden.

Virhetilanteissa tilaksi asetetaan:

- 'server connection failed': Palvelinyhteyden muodostaminen epäonnistui.
- 'room already exists': Lisättävän huoneen tunnus on jo palvelimella.
- 'room not found': Huonetta, johon yhdistetään ei löydy palvelimelta.
- 'room in use': Toinen hoitaja on jo yhdistänyt huoneeseen.

3.3.2 Esimerkki useConnection Hookin käytöstä

Ensin käyttöliittymästä kutsutaan useConnection-hookin startConnection-funktiota.

```
function startConnection(roomCode_, serverAddress_) {  
  setConnectionState('setting up');  
  setPeerConnection(new RTCPeerConnection(peerConstraints));  
  setRoomCode(roomCode_);  
  setServerAddress(serverAddress_);  
}
```

Seuraavaksi useConnection-hookista kutsutaan useSignalingServer-hookin connectToServer-funktiota. Reactin state-muuttujien asettaminen tapahtuu

asynkronisesti ja useEffect-hookilla varmistetaan, että connectToServer-funktiota ei kutsuta ennen kuin tarvittavat statet on asetettu.

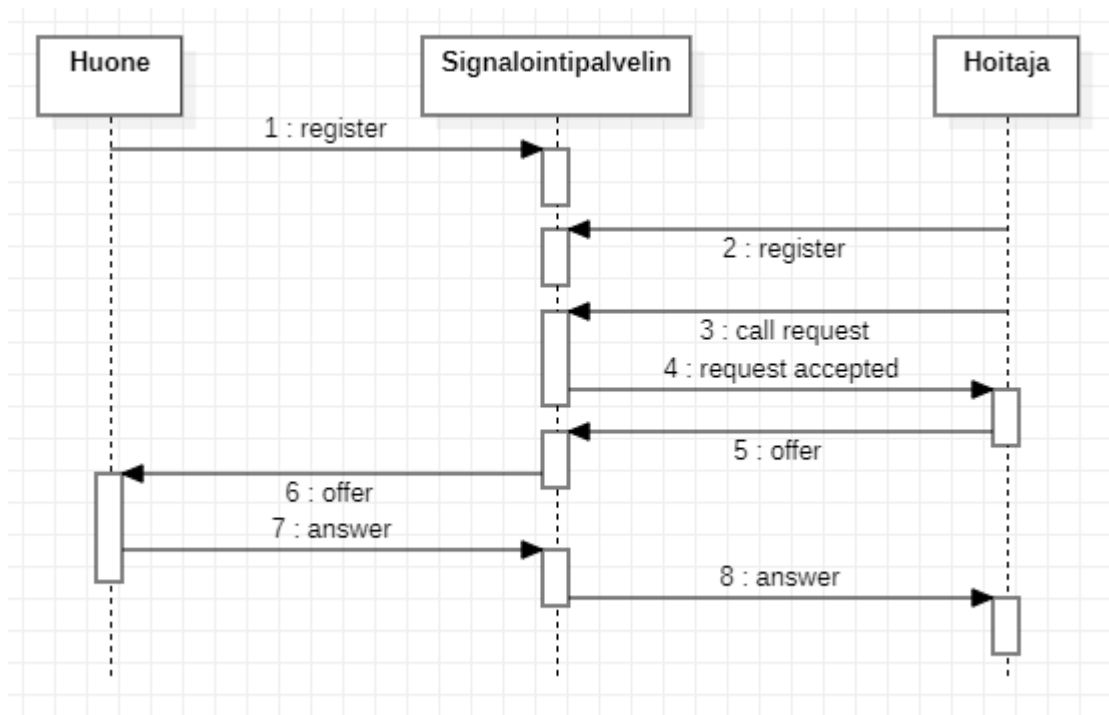
```
useEffect(() => {  
  if (connectionState !== 'setting up' || !peerConnection ||  
      !serverAddress || !roomCode) return;  
  connectToServer(serverAddress, peerConnection, roomCode);  
}, [connectionState, peerConnection, serverAddress, roomCode])
```

Kun palvelinyhteys on muodostettu, connectionState-muuttujan arvoksi asetetaan 'connected to server'. Tämän jälkeen kutsutaan setupPeerConnection-funktiota, joka lisää RTCPeerConnection-muuttujan tapahtumakuuntelijat.

```
useEffect(() => {  
  if (connectionState !== 'connected to server' ||  
      !localMediaStream) return;  
  setupPeerConnection();  
}, [connectionState, localMediaStream])
```

setupPeerConnection-funktion lopussa RTCPeerConnection-muuttujaan lisätään MediaStream-muuttuja. Tämä laukaisee 'negotiationneeded'-tapahtuman. Jos sovelluksen rooli on hoitaja, palvelimelle lähetetään 'call request'-pyyntö ja WebRTC-yhteyden muodostaminen alkaa.

3.3.3 Esimerkki yhteyden muodostamisesta



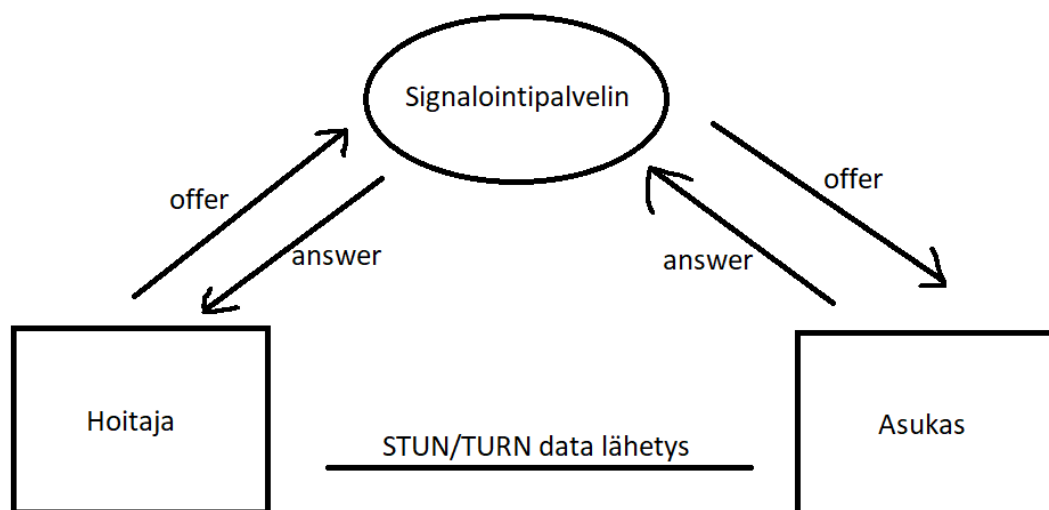
Kuva 1. Sekvenssikaavio WebSocket-viestien kulusta yhteyden muodostamisen aikana.

Yhteyden muodostamisen aikana signalointipalvelin ja kaksi sovellusta (huone ja hoitaja) viestivät seuraavasti:

1. Huoneesta lähetetään palvelimelle 'register'-pyyntö ja huoneen WebSocket-olio ja huonetunnus tallennetaan palvelimen room-listaan.
2. Hoitajalta lähetetään palvelimelle 'register'-pyyntö ja hoitajan WebSocket-olio ja huonetunnus tallennetaan palvelimen nurse-listaan.
3. Hoitaja lähettää palvelimelle 'call request' pyynnön.
4. Jos hoitajan huonetunnus löytyy room-listassa olevasta huoneesta, eikä toinen hoitaja ole jo yhdistänyt samaan huoneeseen, vastauksena lähetetään 'request accepted'. Muussa tapauksessa vastaus on 'request denied' ja yhteyden muodostaminen päättyy.

5. Hoitaja lähettää SDP-tarjouksen. Hoitajan ICE-gathering prosessi alkaa. Hoitaja lähettää kaikki löytämänsä ICE-kandidaatit palvelimen kautta huoneelle.
6. Huone vastaanottaa SDP-tarjouksen. Huoneen ICE-gathering prosessi alkaa.
7. Huone lähettää SDP-vastauksen.
8. Hoitaja vastaanottaa SDP-vastauksen. Peer-to-peer -yhteys muodostetaan, kun ICE-gathering prosessi on valmis.

4 Signaalointipalvelin



Kuva 2. Signaalointidatan tapahtumat.

Palvelin toimii osoitekirjana, josta molemmat osapuolet voivat löytää toisensa avoimesta ja paikallisesta verkosta. Syy tähän johtuu palomuuureista ja NAT-laitteista. Palvelin ei hoida audio- ja videoyhteyden dataa. Palvelin tarvitsee yhteyden muodostamista varten rekisteröitymisen laitteelta, koska laite on käytännössä näkymätön verkossa.

4.1 Signaalintipalvelimen implementaatio

Palvelin lähtee päälle portilla, joka on asetettu ympäristömuuttujiin nimellä SIGNAL_PORT.

```
let port = process.env.SIGNAL_PORT;
```

4.1.1 yhdistävän laitteen kriteerit rekisteröitymistä varten

Huoneen nimi ja/tai numero, isRoom totuusarvo Room=true Nurse=false ja automaattisesti tapahtuva websocketin tallentaminen. Kaikki tiedot tallennetaan key-value "karttoihin" ja erikseen paikalliseen muuttujiin tallennetaan roomNumber ja isRoom muuttujat, jotta yhteyden katketessa websocketin tiedot poistetaan palvelimen kartasta.

4.1.2 Pyynnöt palvelimelle:

Palvelin seuraa tapahtumaa message. Kun tapahtumaa kutsutaan, pyyntö jäsennetään aluksi json-muotoon ja tämän jälkeen pyynnön osat testataan if-lauseella ja käsitellään.

Kaikki paitsi register pyyntö käyttävät handelConn funktiota, joka hakee roomCode-muuttujalla kahdesta mahdollisesta kartasta vastaavan websocketin viestin lähettämistä varten.

```
function handelConn(json) {

    let response;
    let isRoom = json.isRoom && JSON.parse(json.isRoom);

    if(isRoom){
        if(room.get(json.roomCode) != undefined){
            response = room.get(json.roomCode);
            response = response.websocket;
        }
    }
    else if(!isRoom){
        if(nurse.get(json.roomCode) != undefined){
            response = nurse.get(json.roomCode);
            response = response.websocket;
        }
    }
    else {
        console.log("Socket not found!");
        response = null;
    }
    return response;
}
```

Esimerkki pyynnön tyypistä ja sen käsittelystä:

```
case "answer":

    console.log("Answering to " + json.roomCode + " call!");

    try {
        handelConn(json).send(JSON.stringify(json));
        console.log("Answer success!");
    }catch (e) {
        console.log("Answer failed! Socket could not be connected to!");
        ws.send(JSON.stringify({type: 'request denied',
                                reason: 'room not found'}));
    }
    break;
```

Kun lähetetään viesti eteenpäin etsitylle websocketille pitää Json-muoto vaihtaa string-muuttujaksi.

Pyyntö jossa pitää karttojen arvoa muuttaa (Esim offer pyyntö):

```
nurse.set(json.roomCode, {
    websocket : ws,
    inCall : false
});
```

websocket tapahtuman disconnect käsittely:

```

} else if(isRoom === false){
  nurse.delete(storageRoomNum);
  try {
    let tempRoom = room.get(storageRoomNum);
    tempRoom.websocket.send(JSON.stringify({type: "peer
      disconnected"})); // Send disconnect alert to remote peer
    //set inCall to false, so it could be called to!
    tempRoom = {
      websocket : tempRoom.websocket,
      inCall : false
    };
    room.set(storageRoomNum,tempRoom);
    console.log(storageRoomNum
      + " nurse disconnected successfully: ");
    console.log("Disconnect info sent to remote " + storageRoomNum
      + " room");
  }catch (e) {

    if("Cannot read properties of undefined (reading 'websocket')" ==
      e.message){
      console.log("Room " + storageRoomNum + " was not found! (inCall
        change to false not required)");
      console.log("disconnect info was not send to remote!");
    } else {
      console.log(e);
    }
  }
}

```

5 Asennus

5.1 Kehitysympäristön asennus

1. Asenna Node, JDK, Android Studio, React Native CLI ja luo Android Virtual Device seuraamalla [näitä](#) ohjeita.

2. Kloonaa repository:

```
git clone https://github.com/Arbit3r/Innovaatioprojekti.git
```

Jos näet 'Filename too long' -virheen, aja tämä komento:

```
git config --system core.longpaths true
```

3. Käynnistä Metro ajamalla tämä komento kansiossa ReactNativeApp:

```
npm start
```

4. Käynnistä sovellus ajamalla samassa kansiossa:

```
npm run android
```

5.2 APK:n luominen

Debug APK-tiedoston luominen tapahtuu seuraavasti:

1. Suorita tämä komento sovelluksen juurikansiossa:

```
react-native bundle --platform android --dev false --entry-file index.js --bundle-output android/app/src/main/assets/index.android.bundle --assets-dest android/app/src/main/res
```

2. Suorita tämä komento android-kansiossa:

```
./gradlew assembleDebug
```

3. APK löytyy polusta ReactNativeApp/android/app/build/outputs/apk/debug/app-debug.apk

Jos haluat julkaista sovelluksen Google Play -kaupassa, seuraa [näitä ohjeita](#).

5.3 Signaalintipalvelimen asentaminen dockerin avulla:

dockerhub: https://hub.docker.com/r/7riku/inno_projekti/tags

pull command: docker pull 7riku/inno_projekti:latest

docker run: docker container run -p {host portti}:{kontin portti} -e SIGNAL_PORT={kontin portti} 7riku/inno_projekti:latest

Esim.

```
docker container run -p 25565:5000 -e SIGNAL_PORT=5000 7riku/inno_projekti:latest
```

Nyt palvelin pyörii osoitteessa ws://localhost:25565 ja kontin portti on 5000.

HUOM! kontin portit tulee olla samat.

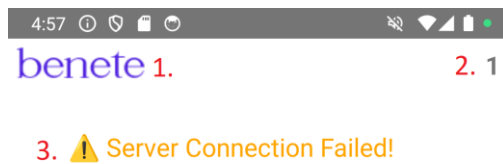
Portti on asetettu ympäristö muuttujaksi palvelimen koodissa, joten sen voi vaihtaa ajamisen alussa!

Signalointiserverin voi myös ajaa node serverinä asentamalla node moduulit ja ajamalla hakemistossa node signalingServer.js

Jos sen ajaa node serverinä, pitää ympäristömuuttuja olla asetettuna SIGNAL_PORT.

6 Käyttöohjeet

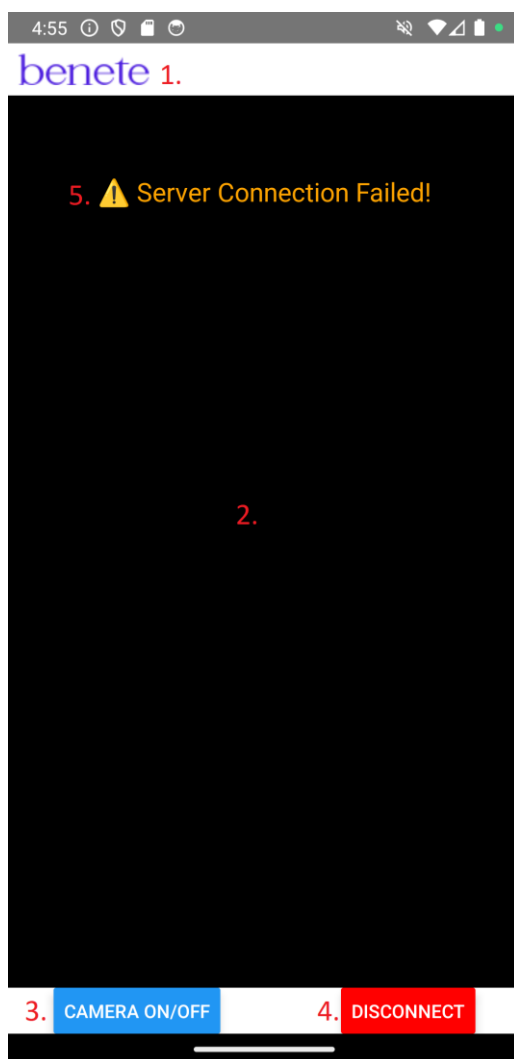
6.1 Sovelluksen käyttäminen



Kuva 3. Aukkaan näkymä.

Asukkaan näkymän toiminta:

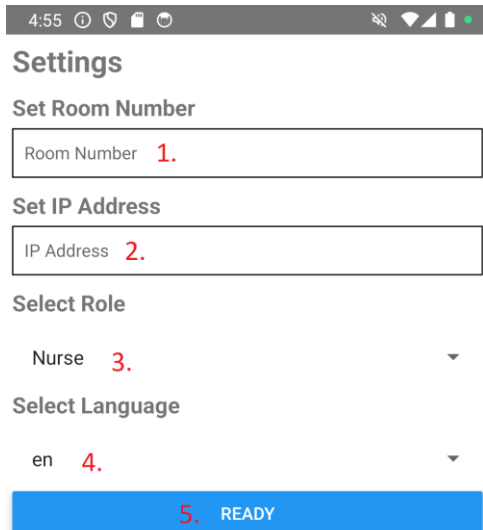
1. Painamalla Benete-logoa 3 sekuntia, avautuu valikko asetuksille.
2. Asukkaan huoneen tunnus (Muutettavissa asetuksissa).
3. Jos hoitajan sekä asukkaan välinen yhteys epäonnistuu, ilmestyy näytölle oranssilla tekstillä ilmoitus, joka kertoo yhteyden epäonnistumisen syyn.



Kuva 4. Hoitajan näkymä.

Hoitajan näkymän toiminta:

1. Painamalla Benete-logoa 3 sekuntia, avautuu valikko asetuksille.
2. Kuvayhteys asukkaaseen.
3. Nappi, joka avaa kuvayhteyden asukkaaseen.
4. Nappi, joka katkaisee yhteyden huoneeseen. Yhteys sulkeutuu myös, jos asetusvalikko avataan.
5. Jos hoitajan sekä asukkaan välinen yhteys epäonnistuu, ilmestyy näytölle oranssilla tekstillä ilmoitus, joka kertoo yhteyden epäonnistumisen syyn.



4:55

Settings

Set Room Number

Room Number 1.

Set IP Address

IP Address 2.

Select Role

Nurse 3.

Select Language

en 4.

5. READY



Kuva 5. Asetusnäköymä.

Asetusnäköymän toiminta:

1. Tekstikenttä, jolla määritetään asukkaan huoneen tunnus. Tekstikenttä hyväksyy minkä tahansa merkkijonon.
2. Tekstikenttä IP-osoitteelle. IP-osoite tulee kirjoittaa seuraavassa muodossa: ws://192.168.0.1:8080. Tässä esimerkissä 192.168.0.1 on signaalintipalvelimen IP-osoite ja 8080 on palvelimen asennuksen aikana asetettu portti.
3. Pudotusvalikko, josta valitaan rooli asukkaan tai hoitajan välillä.
4. Pudotusvalikko, josta valitaan sovelluksen kieli.
5. Nappi, joka tallentaa asetukset ja poistuu joko Hoitaja- tai Asukas-näkymään, riippuen mikä rooli valittiin. Jos rooli on hoitaja, puhelu Huonetunnus-tekstikentässä asetettuun huoneeseen alkaa automaattisesti.

6.2 Kielten lisääminen

Kielten lisääminen tapahtuu lisäämällä .json -päätteinen tiedosto locales-kansioon. Tiedoston voi tehdä millä tahansa tekstieditorilla. Tiedoston nimeksi tulee kielen nimi, joka halutaan tehdä. Tiedoston nimi voi olla esimerkiksi swedish.json tai se.json.

```
1  {
2    "everything_is_okay": "Kaikki kunnossa!",
3    "screen_title": "Asetukset",
4    "room_number_placeholder": "Huonetunnus",
5    "select_role_label": "Valitse rooli",
6    "resident": "Asukas",
7    "nurse": "Hoitaja",
8    "ready_button_title": "Valmis",
9    "select_language_label": "Valitse kieli",
10   "room_number_empty": "Huonetunnus ei voi olla tyhjä!",
11   "set_roomnumber_label": "Aseta huonetunnus",
12   "set_ipAdress_label": "Aseta IP-osoite",
13   "ipAddress_placeholder": "IP-osoite"
14 }
```

Kuva 6. fi.json -tiedosto.

Tiedosto sisältää rivejä, jotka jakautuvat kahteen osaan. Ensimmäinen osa määrittää placeholder-elementin sovelluksessa, jonka tilalle tulee teksti, joka asetetaan rivin toisessa osassa. Toiseen osaan kirjoitetaan käännös halutulla kielellä.

App.jsx tiedostossa oleva "LanguageProvider" huolehtii sovelluksen kielivalinnasta ja valinnan puuttuessa vaihtaa kielen automaattisesti englanniksi. Jos haluaa lisätä uusia näkymiä sovellukseen, niin lisäämällä ne App.jsx tiedoston Stack.Navigatorin väliin, joka on LanguageProviderin välissä, niin lisätyt näkymät toimivat sovelluksen näkymä- ja kielilogiikalla.

LanguageProvider-logiikan sisällä olevissa näkymissä voi käyttää itsetehtyjen kielitiedostojen sisältöjä. Käyttöä varten kutsutaan ensin `react-i18next:n` `useTranslation` ja tehdään sille muuttuja (`const translation = useTranslation();`). Kun `useTranslationia` käyttävä muuttuja on luotu, niin sitä käyttämällä voidaan käyttää kielitiedostojen tekstejä käyttämällä kielitiedostoissa asetettuja avainsanoja esimerkiksi näin: `translation("language_key")`. Kielen vaihtamista varten tehdään `useTranslation` muuttujan luonnissa toinen muuttuja esimerkiksi `"i18n"`. Kun tämä muuttuja on luotu, niin kieltä voidaan vaihtaa komennolla `i18n.changeLanguage("kieli")`. Kielen vaihdon yhteydessä kannattaa tallentaa valinta myös `AsyncStorageen`, jotta se tallentuu käyttäjän puhelimeen (`AsyncStorage.setItem('@language', lang);`).

6.3 Sovelluksen avaaminen toisen sovelluksen kautta

Sovelluksen avaaminen toisen sovelluksen (esim. Beneten olemassaoleva sovellus, johon hälytyksen tulevat) kautta toimii "DeepLinkin" kautta. DeepLinkillä voidaan siis avata muita sovelluksia jossakin sovelluksessa ja halutessaan viedä dataa sovelluksesta toiseen. Sovelluksessamme huonenumero, johon soitetaan tulee tällaisen linkin kautta. Sovelluksemme siis kuuntelee App.jsx-tiedostossa vastaanotettavia DeepLinkkejä ja ottaa vastaanotetun datan talteen.

Jotta DeepLinkit toimii, projektin AndroidManifest.xml-tiedostoon on lisätty seuraava koodi:

```
<intent-filter>
  <action android:name="android.intent.action.VIEW" />
  <category android:name="android.intent.category.DEFAULT" />
  <category android:name="android.intent.category.BROWSABLE" />
  <data
    android:scheme="sovelluksenNimi"
    android:host="data"
  />
</intent-filter>
```

Eli valitaan jokin nimi jonka DeepLinkit voivat tunnistaa tuohon `android:scheme="sovelluksenNimi"` ja siihen yhdistettävä metodin nimi, esimerkiksi tässä etäyhteyssovelluksessa käytetty `"data"` datan lähettämistä varten tuohon `android:host="data"`.

Linkkejä käytetään tuomalla React Nativen Linking-työkalu (`import { Linking } from "react-native";`) ja sitten voidaan avata sovelluksia URL osoitteiden kautta esimerkiksi näin:

```
const deepLink =
`sovelluksenNimi://data?textData=${encodeURIComponent(textData)}`;

Linking.openURL(deepLink);
```

Tässä esimerkissä luodaan URL-osoite etäyhteyssovelluksen avaamista varten ja siihen liitetään dataksi teksti `data`, joka tulee `textData`-muuttujasta (`textData` on testi sovelluksessa oleva testi String-muuttuja) ja sitten se avataan `Linking.openURL(url)` metodilla. Beneten sovellukseen tulisi siis lisätä tällä logiikalla toimiva DeepLinkin avaaminen niin, että linkin tekstidataksi tulee hälytyksen huonenumero.