Joni Savolainen

Reaktiivinen sovellusarkkitehtuuri kommunikointiin Bluetooth Mesh -verkon kanssa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

12.4.2023

Tiivistelmä

Tekijä:	Joni Savolainen
Otsikko:	Reaktiivinen sovellusarkkitehtuuri kommunikointiin Bluetooth Mesh -verkon kanssa
Sivumäärä:	44 sivua + 3 liitettä
Aika:	12.4.2023
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaajat:	Ohjelmistoinsinööri Eemeli Ristimella Lehtori Miikka Mäki-Uuro

---

Insinöörityössä suunniteltiin langattoman valonohjausjärjestelmän käyttöönottoon tarkoitetulle mobiilisovellukselle uusi arkkitehtuuri, jonka päätavoitteena on sovelluksen käytettävyyden parantaminen.

Nykyinen arkkitehtuuri kerää kaiken tarvitsemansa tiedon lyhyellä aikavälillä, mikä kuormittaa langattoman valonohjausjärjestelmän Bluetooth Mesh -verkkoa. Uusi arkkitehtuuri suunniteltiin käyttäen reaktiivisen ohjelmoinnin ja RxJS-kirjaston konsepteja. Ne mahdollistivat arkkitehtuurin, jossa tietoa kerätään passiivisesti yksittäisistä laitteista siten, että sovellus kysyy tarvitsemansa tiedot osissa. Tämä lähestymistapa osoittautui hyväksi sovelluksen käyttökokemuksen kannalta: odotusaika ennen laitteen parametrien muokkausta väheni alle kymmenesosaan.

Mobiilisovellus lähettää uudet parametrit Bluetooth-verkkoon joko yksittäisille laitteille tai kokonaisille laiteryhmille kerrallaan. Nykyinen arkkitehtuuri ei varmista uusien parametrien tallennusta kohdelaitteissa. Työssä toteutettiin parametrien lähetys, jossa mobiilisovellus varmistaa parametrien asetuksen kohdelaitteeseen ja lähettää ne uudelleen, mikäli parametrit eivät saavu kohteeseensa. Lähetystapa mahdollisti 200 uniikin parametrin lähetyksen yhteen kohdelaitteeseen minuutin aikana siten, että kohdelaite vastaanotti ne kaikki.

Uusi arkkitehtuuri ei sovi kommunikointiin laiteryhmien kanssa sellaisenaan. Työssä havaittiin, että tarvittaisiin joko arkkitehtuurin muuttamista joustavammaksi tai muutoksia valonohjausjärjestelmän Bluetooth-verkon protokollassa.

Avainsanat:	reaktiivinen ohjelmointi, mobiililaitteet, Bluetooth, verkko-ohjelmointi

Abstract

Author:	Joni Savolainen
Title:	Reactive software architecture for communication with a Bluetooth mesh network
Number of Pages:	44 pages + 3 appendices
Date:	12 April 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Systems
Professional Major:	Game applications
Supervisors:	Eemeli Ristimella, Software Engineer Miikka Mäki-Uuro, Senior Lecturer

---

This thesis aimed to design a new architecture for a mobile application used to commission a wireless lighting control system, with the main objective of improving the application's usability.

The current architecture collects all needed data in a short period of time, which puts a strain on the Bluetooth mesh network of the lighting control system. Reactive programming concepts and the RxJS library enabled an architecture where data is passively collected from individual devices, with the application interface components responsible for querying the data they need. This approach proved to be good for usability – the waiting time before editing device parameters was reduced to less than one tenth.

During commissioning of the lighting control system, the mobile app sends new parameters to the network. The current architecture does not ensure that parameters are stored on the target. In this thesis, a new form of parameter transmission was implemented. This method verified if parameters were saved on the target device and retried if not. Within a minute, 200 unique parameters were sent and received by the target device using this transmission method.

The new architecture is not suitable for communicating with groups of devices. It was concluded that this would require either a more flexible architecture or changes to the Bluetooth network protocol of the lighting control system.

Keywords: Reactive programming, mobile devices, Bluetooth, web programming

Sisällys

Lyhenteet

1 Johdanto 1

2 Reaktiivinen ohjelmointi ja RxJS 4

2.1 Tietovirrat 4
2.2 Operaattorit 7
2.2.1 Tap-operaattori 8
2.2.2 Luovat operaattorit 9
2.2.3 Muovaavat operaattorit 11
2.2.4 Rajaavat operaattorit 12
2.2.5 Yhdistävät operaattorit 15
2.3 Monilähetys 17
2.4 Virheenhallinta 20

3 Reaktiivinen arkkitehtuuri Angular-sovellukselle 21

3.1 Angular-komponentit 22
3.2 Reaktiiviset komponentit 24
3.3 Bluetooth-verkon liikenteen kuuntelu 25
3.4 Tietokannat 28
3.5 Riippuvuudet 32
3.6 Tiedon päivittäminen käyttöliittymässä 33
3.7 Lähistöllä tapahtuvan käyttöönoton kuuntelu 36

4 Lopputulos ja vertailu 37

5 Yhteenveto 43

Lähteet 44

Liitteet

Liite 1: Tietokannan tietueen kelpoisuuden määritys

Liite 2: Odotusajat ennen valotasojen ja ajoitusten muokkausta

Liite 3: Puhelinten vastaanottamat arvot

Lyhenteet

BLE: *Bluetooth Low Energy.* Langaton viestintäteknologia, joka on suunniteltu pieneen virrankulutukseen ja lyhyen kantaman tiedonsiirtoon.

CSS: *Cascading Style Sheets.* HTML-rakenteen esitystapaa kuvaava säännöstö.

HTML: *Hypertext Markup Language.* Merkintäkieli tiedon näyttämiseen verkkoselaimessa.

HTTP: *Hypertext Transfer Protocol.* Sovellusprotokolla tiedonsiirtoon internetissä.

RxJS: *Reactive Extensions for JavaScript.* JavaScript-kirjasto, joka sisältää työkalut asynkronisten tietovirtojen käsittelylle.

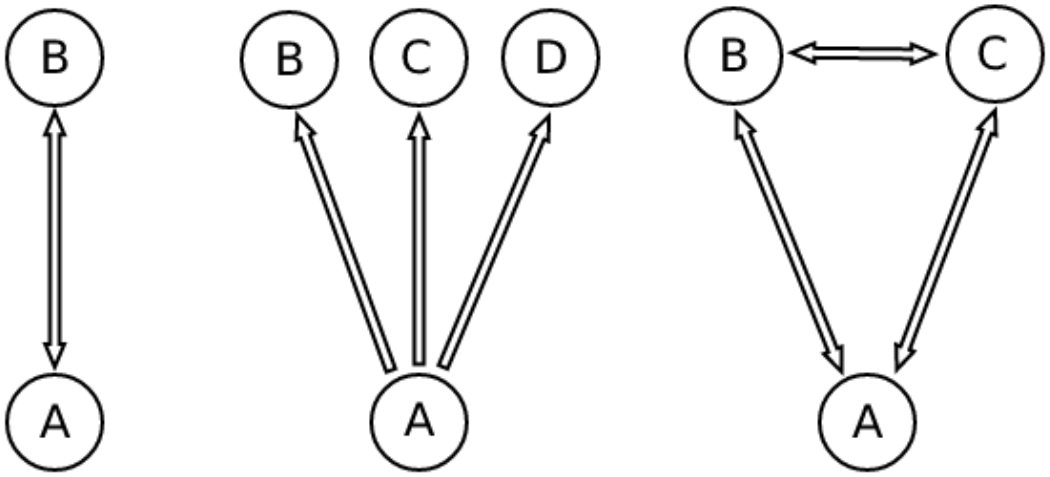
SQL: *Structured Query Language.* Kyselykieli operoimiseen relaatiotietokannan kanssa.

TTL: *Time-to-live.* Bluetooth Mesh -viestin arvo, joka määrittää, kuinka monta siirtymää viesti voi tehdä verkon laitteesta toiseen ennen sen hylkäystä.

1 Johdanto

Insinöörityön tarkoituksena on parantaa langattoman valonohjausjärjestelmän käyttöönottoon tarkoitetun mobiilisovelluksen käytettävyyttä luomalla sille uusi arkkitehtuuri. Työ tehdään suomalaiselle valaisinkomponentteihin ja valaistuksen ohjaukseen erikoistuneelle yritykselle Helvar Oy Ab:lle. ActiveAhead on yrityksen langaton valonohjausjärjestelmä, jossa valaisimiin asennetut ActiveAhead-yksiköt kommunikoivat keskenään Bluetooth Low Energy (BLE) -tekniikkaan perustuvan Bluetooth Mesh -protokollan mukaisesti (ActiveAhead – itseoppiva valaistuksenohjausratkaisu 2022: 6).

BLE on perinteisesti mahdollistanut kaksi topologiaa laitteiden muodostamille verkoille: yhdestä yhteen (engl. one-to-one, kuva 1), jossa kaksi laitetta ovat yhdistettyinä toisiinsa, sekä yhdestä moneen (engl. one-to-many, kuva 1), jossa yksi laite keskustelee monen muun laitteen kanssa (Bluetooth SIG 2021). Bluetooth Mesh mahdollistaa kolmannen verkkotopologian: monesta moneen (engl. many-to-many, kuva 1), jossa verkon viestit lähetetään kaikille radioetäisyydellä oleville vastaanottajille (Bluetooth SIG 2019).

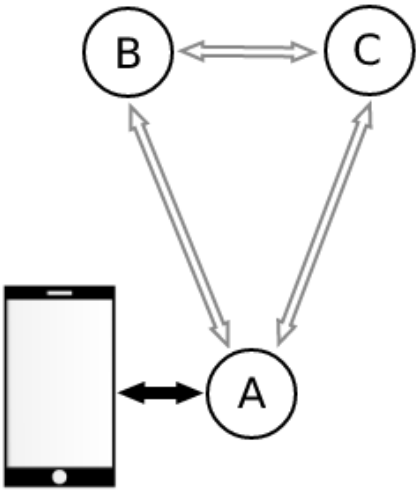


Kuva 1. BLE-verkkotopologiat: yhdestä yhteen (vasemmalla), yhdestä moneen (keskellä) ja Bluetooth Mesh -protokollan mahdollistama monesta moneen (oikealla).

Viestit liikkuvat Bluetooth Mesh -verkossa tulvaperiaatteella: kun viesti on lähetetty, sen vastaanottajat lähettävät viestin eteenpäin (Bluetooth SIG 2019). Viesteillä on numeerinen elinaika-arvo (engl. time-to-live, TTL). Mesh-verkon laitteet vähentävät saapuneen viestin TTL-arvoa yhdellä, ja sen laskiessa nollaan hylkäävät viestin kokonaan (Bluetooth SIG 2019). Täten yhden viestin elinikä on TTL-arvon mukainen ja määrää viestien hyppyjen (engl. hops) määrän verkon laitteesta toiseen. Bluetooth Mesh määrää välimuistin vastaanotetuille viesteille: jos viesti löytyy välimuistista, se hylätään välittömästi eikä sitä lähetetä verkkoon riippumatta viestin TTL-arvosta.

Bluetooth Mesh -protokollan viestintä sisältää valinnaisesti vahvistukset (engl. acknowledgement), joiden avulla viestin lähettäjä saa varmistuksen viestin perillemenosta (Bluetooth SIG 2019). ActiveAhead ei käytä vahvistuksia verkon toimintatehon parantamiseksi. Verkon tiedonsiirto on täten User Datagram Protocol (UDP) -protokollan kaltaista siten, ettei viestejä kuitata tai lähetetä uudelleen virhetapauksissa.

Valonohjausjärjestelmä otetaan käyttöön mobiilisovelluksen avulla, joka toimii käyttöönottoinsinöörien työvälineenä. Se kommunikoi ActiveAhead-laitteiden kanssa tekemällä yhdestä yhteen -yhteyden verkon laitteeseen. Välitinlaitteena toimiva ActiveAhead-yksikkö vie mobiilisovelluksen lähettämät komennot verkkoon ja antaa sovellukselle pääsyn verkon liikenteeseen (kuva 2). Se on tyypillinen tapa kommunikoida Bluetooth Mesh -verkon kanssa toisella päätelaitteella (Woolley 2022).



Kuva 2. Mobiilisovellus tekee yhdestä yhteen -yhteyden Bluetooth Mesh -verkon laitteeseen A. Laite välittää sovelluksen lähettämät viestit mesh-verkkoon, ja antaa sovellukselle pääsyn laitteen A saamiin viesteihin.

Mobiilisovelluksen arkkitehtuuri ei huomioi mesh-verkon topologiaa. Sovellus pyytää tietolähteiltään kaiken tarvittavan tiedon lyhyellä aikavälillä. Vaikeuksia luovat tilanteet, joissa mobiilisovellus pyytää tietoja monelta tietolähteeltä. Nämä tilanteet voivat luoda paljon liikennettä verkkoon muodostaen pullonkaulan, joka hidastaa verkon toimintaa (Aijaz ym. 2021).

Mobiilisovellus ei varmista lähetettyjen komentojen vastaanottoa. Mesh-verkon topologia mahdollistaa viestien menetyksen, varsinkin kun niitä ei kuitata. Se on todettu harvinaiseksi tavanomaisessa käyttöönotossa, mutta jos Bluetooth-verkko sijaitsee 2,4 GHz:n taajuusalueella, kommunikointi vaikeutuu ympäristöissä, joissa taajuuden kohinataso on suuri, kuten suurissa Bluetooth-asennuksissa (Rondón ym. 2020). Bluetooth-verkon laitteet voivat prosessoida vain rajoitetun määrän viestejä kerrallaan, mikä voi muodostaa niin sanotun sokean ajan (engl. blind time), jonka aikana laite ei reagoi uusiin viesteihin. Sen vaikutus kasvaa Bluetooth-verkossa viestejä välittävien laitteiden määrän lisääntyessä (Hernández-Solana ym. 2020).

Insinöörityössä suunniteltavan arkkitehtuurin tavoitteena on parantaa toiminnallisuutta. Tietojen pyyntö on jaettava osiin siten, että sovellus kysyy tarvitsemansa tiedon ainoastaan tarvittaessa. Lisäksi sovelluksen tulisi kysyä tietoja

uudestaan vastauksen uupuessa. Parametrien lähetykset tarvitsevat varmistusvaiheen, jossa sovellus varmentaa niiden siirtymisen kohteeseensa. Nämä kaksi parannusta tekisivät sovelluksesta nopeamman ja kohentaisivat sen käyttökokemusta.

2 Reaktiivinen ohjelmointi ja RxJS

Modernit verkkosovellukset ovat reaktiivisia: ne ovat viestipohjaisia ja reagoivat tapahtumiin niiden saapuessa järjestelmään. Tapahtumien prosessointi saattaa vuorostaan luoda uusia tapahtumia. Se tekee reaktiivisten järjestelmien suunnittelusta ja ylläpidosta vaikeaa. Koodin ohjauksen kulkua on vaikea ymmärtää, jos sen laukaisee epädeterministisesti saapuvat viestit. (Salvaneschi ym. 2015.)

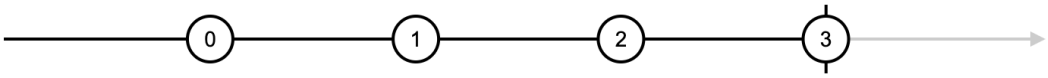
Reaktiivinen ohjelmointi on abstraktio perinteisten imperatiivisten järjestelmien päälle, ja sen tarkoitus on tukea reaktiivisten sovellusten kehitystä (Nurkiewicz & Christensen 2016). Se on ohjelmointia asynkronisilla tietovirroilla (Staltz 2018). Reaktiiviset verkkosovellukset koostuvat komponenteista, jotka reagoivat asynkronisesti ja riippuvuussuhteisesti tapahtumiin ja päivittävät sovelluksen esityksellisen osan.

Yksi suosituimmista kirjastoista reaktiivisen ohjelmoinnin toteuttamiseen JavaScript-ohjelmistoissa on Reactive Extensions for JavaScript (RxJS). Se on JavaScript-kirjasto, joka tarjoaa työkalut asynkronisten tietovirtojen hallintaan (Introduction) tarjoamalla SQL-kyselykielen kaltaisia operaatioita tietovirran tietueille. Sitä päätettiin käyttää tässä työssä, koska ActiveAhead-mobiilisovellus käytti kirjastoa jo aiemmin.

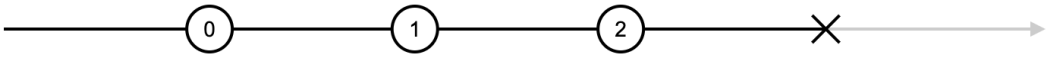
2.1 Tietovirrat

Tietovirrat ovat reaktiivisen ohjelmoinnin selkäranka. Ne koostuvat RxJS-kirjastossa kolmesta rakennusosasta: arvoviesteistä, virheviesteistä ja lopetusviesteistä. Tietovirrat sisältävät minkä tahansa määrän arvoviestejä, ja ne päättyvät joko lopetusviestiin tai virheviestiin (Observable).

Tietovirtoja kuvataan marble-diagrammien avulla, joissa arvoviestit ovat ympyröitä, lopetusviesti pystysuora viiva ja virheviesti rasti (Operators). Kuvat 3 ja 4 sisältävät marble-diagrammeja, jotka kuvaavat tietovirran kulkua.



Kuva 3. Marble-diagrammi tietovirrasta, joka lähettää neljä arvoa säännöllisin aikavälein ja päättyy heti neljännen arvon lähetyksen jälkeen.

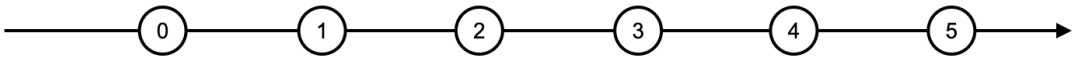


Kuva 4. Tietovirta, joka päättyy virheeseen kolmannen arvon jälkeen.

RxJS-kirjasto esittää tietovirrat Observable-luokan instansseina (esimerkkikoodi 1 ja kuva 5).

```
const numbers$ = interval(300);
```

Esimerkkikoodi 1. RxJS-kirjaston interval-funktio luo Observable-instanssin. Sen kuvaama tietovirta lähettää numeroita tietyin aikavälein, joka on esimerkissä 300 millisekuntia (API List). numbers$-muuttujan nimen dollarimerkki on yleinen käytäntö Observable-tyyppisten muuttujien nimeämiseksi (The RxJS Library 2022).

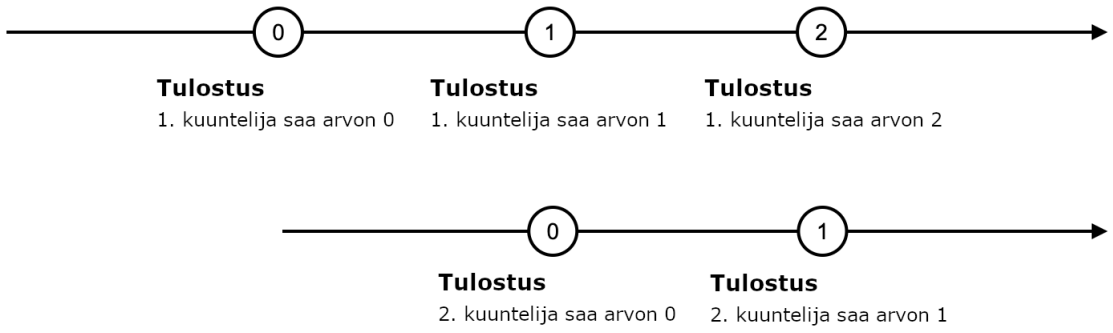


Kuva 5. Esimerkkikoodin 1 Observablen kuvaama tietovirta marble-diagrammina, joka kuvaa sen kulun ensimmäiset kaksi sekuntia.

Observable-instanssien suoritus ei tapahdu niiden luontihetkellä. Ne käynnistyvät kuuntelemalla niitä, ja jokainen kuuntelija saa oman kopionsa tietovirrasta (Observable). Esimerkkikoodi 2 ja kuva 6 kuvaavat tilannetta, jossa Observable-instanssilla on kaksi kuuntelijaa.

```
const numbers$ = interval(500);

numbers$.subscribe((number) => {
  console.log("1. kuuntelija saa arvon " + number);
});

setTimeout(() => {
  numbers$.subscribe((number) => {
    console.log("2. kuuntelija saa arvon " + number);
  });
}, 500);
```

Esimerkkikoodi 2. Observable-instanssin suorittamisen aloitus subscribe-funktiolla. Tietovirta saa toisen kuuntelijan 500 millisekunnin jälkeen. Subscribe-funktiokutsuille annetut takaisinkutsufunktiot tulostavat tietovirran numerot konsoliin.

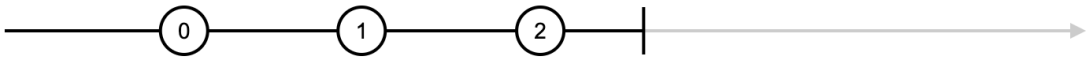


Kuva 6. Esimerkkikoodin 2 toiminnan kuvaus marble-diagrammeina.

Observablen subscribe-funktio palauttaa Subscription-olion, jonka unsubscribe-funktio lopettaa Observable-instanssin suorituksen ja päättää sen kuvaaman tietovirran (esimerkkikoodi 3, kuva 7).

```
const numbers$ = interval(500);

const subscription = numbers$.subscribe((number) => {
  console.log(number);
});

setTimeout(() => {
  subscription.unsubscribe();
}, 1800);
```

Esimerkkikoodi 3. Observable-instanssin suorittamisen lopettaminen unsubscribe-funktiolla (Observable).



Kuva 7. Esimerkkikoodin 3 luoman tietovirran kulku. Se päättyy ennen neljännen arvon lähetystä.

2.2 Operaattorit

RxJS-kirjaston tärkein ominaisuus on tietovirtojen muodostaminen operaattoreiden avulla. Ne ovat funktioita, jotka muovaavat, rajaavat ja yhdistävät tietovirtoja. (Operators)

```
numbers$.pipe(x(), y(), z());
```

Esimerkkikoodi 4. Syntaksi operaattoreiden käyttöönottoon pipe-instanssifunktiolla. x, y ja z ovat operaattoreita.

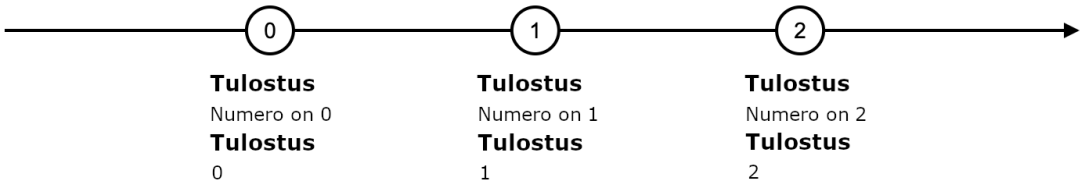
Operaattoreita käytetään Observablen pipe-instanssifunktiolla (esimerkkikoodi 4). Niitä on RxJS-kirjastossa yli 100 (API List). Seuraavissa alaluvuissa esitellään insinöörityössä käytettyjä operaattoreita ja osa muista yleisistä operaattoreista.

2.2.1 Tap-operaattori

Tap eroaa muista operaattoreista siten, ettei sillä ole konkreettista vaikutusta tietovirtaan (API List). Tap-operaattori mahdollistaa sivuvaikutusten luomisen tiedon kulkiessa tietovirrassa (esimerkkikoodi 5, kuva 8).

```
import { interval, tap } from "rxjs";

const numberArray = [];

const numbers$ = interval(500).pipe(
  tap((number) => {
    console.log("Numero on " + number));
    numberArray.push(number);
  })
);

numbers$.subscribe(console.log);
```

Esimerkkikoodi 5. Tap-operaattorin käyttö sivuvaikutuksen luomiseksi. Esimerkkikoodissa on kaksi sivuvaikutusta: ylimääräinen tulostus konsoliin ja tietovirran arvojen lisäys numberArray-taulukkoon. Seurauksena on kahden viestin tulostuminen konsoliin jokaista tietovirran arvoa kohti sekä numberArray-taulukon päivittyminen.

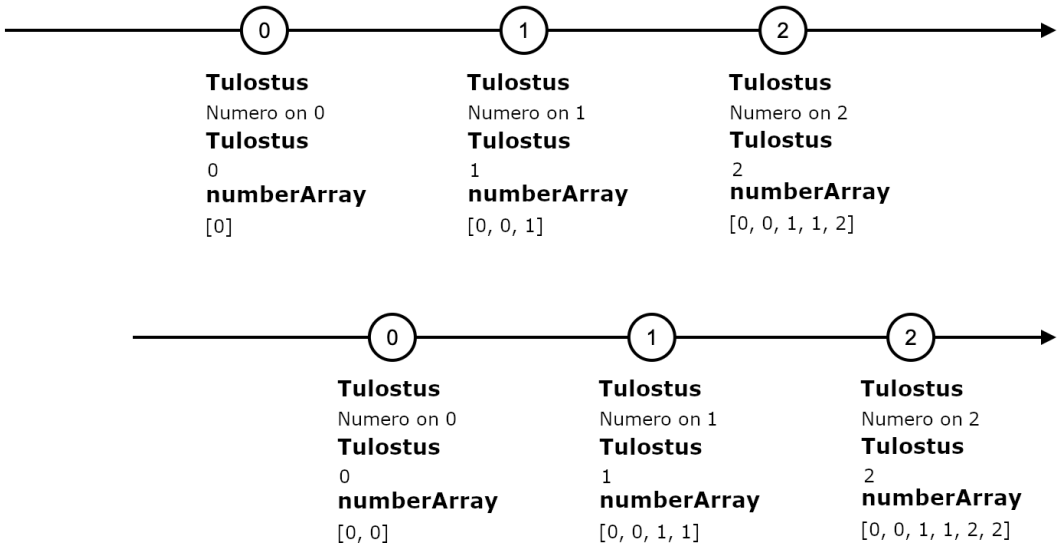


Kuva 8. Esimerkkikoodin 5 toiminnan kuvaus. Jokainen arvo aiheuttaa kaksi tulostusta konsoliin.

Tap-operaattorin sivuvaikutus suoritetaan jokaista kuuntelijaa kohti, sillä ne saavat oman kopionsa tietovirrasta. Esimerkkikoodissa 5 rekisteröidään toinen kuuntelija 250 millisekunnin jälkeen ensimmäisestä. Kuva 9 havainnollistaa numberArray-taulukon muutokset, kun kaksi samanaikaista tietovirtaa muokkaavat sitä.

```
numbers$.subscribe(console.log);

setTimeout(() => {
  numbers$.subscribe(console.log);
}, 250)
```

Esimerkkikoodi 6. Esimerkkikoodin 5 Observable-instanssin kuuntelu kahdella kuuntelijalla, joista toinen lisätään 250 millisekunnin jälkeen.



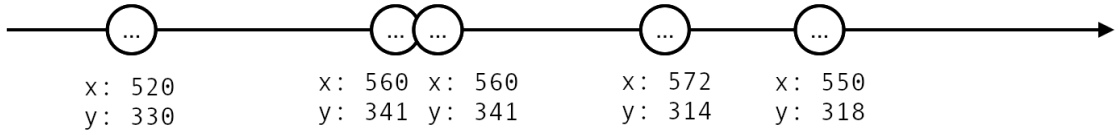
Kuva 9. Esimerkkikoodin 6 kuvaus. Huomattavaa on, kuinka kaksi samanaikaista Observablen suoritusta muokkaavat numberArray-taulukkoa.

2.2.2 Luovat operaattorit

Luovat operaattorit rakentavat Observable-instanssin ja eroavat muista operaattoreista siten, että niitä ei käytetä Observablen pipe-funktiolla. Aikaisemmista esimerkkikoodeista löytyvä interval on luova operaattori. FromEvent on verkko-ohjelmoinnissa käytetty operaattori (API List), joka luo tietovirran verkkosivun elementin tapahtumista (esimerkkikoodi 7).

```
const button = document.querySelector("button");

const buttonClick$ = fromEvent(button, "click");
```

Esimerkkikoodi 7. Observable-instanssin luonti painike-elementin napsahdustapahtumista fromEvent-operaattorilla.

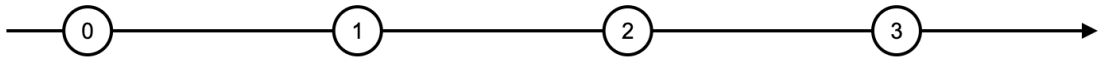


Kuva 10. Esimerkkikoodin 7 luoma tietovirta. Sen arvot ovat epädeterministisiä, sillä niiden määrä ja sisältämä tieto riippuvat käyttäjästä. Kuvan diagrammi kuvastaa yhtä mahdollista aikahaarukkaa, jossa käyttäjä napsauttaa painiketta viisi kertaa kolmen sekunnin aikana.

Timer on interval-funktion kaltainen operaattori, joka lähettää numeroita säännöllisin väliajoin (API List). Sen ero interval-funktioon verrattuna on ensimmäisen parametri, ajankohta, jolloin ensimmäinen numero lähetetään (esimerkkikoodi 8).

```
const numbers$ = timer(300, 1000);
```

Esimerkkikoodi 8. Tietovirran luonti timer-operaattorilla, joka lähettää arvon 0 kolmensadan millisekunnin päästä käynnistyshetkestä ja sen jälkeen kasvavia arvoja sekunnin välein.



Kuva 11. Esimerkkikoodin 8 tietovirran ensimmäiset neljä sekuntia.

Of-operaattori luo tietovirran, joka lähettää operaattorin parametriksi annetun arvon ja päättyy (API List). Esimerkkikoodi 9 ja kuva 12 kuvailevat of-operaattorin toimintaa.

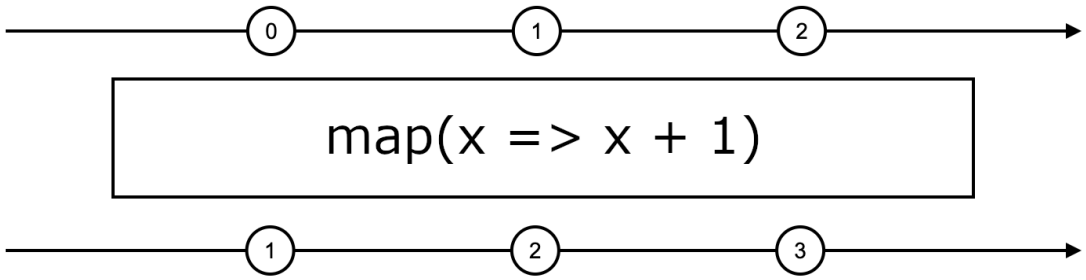
```
const A$ = of("A");
const B$ = of("B").pipe(delay(500));
```

Esimerkkikoodi 9. Tietovirtojen luominen of-operaattorilla. A-virta lähettää arvonsa heti käynnistyshetkellä ja päättyy. B-virta käyttää delay-operaattoria, joka siirtää arvon lähetyshetkeä 500 millisekuntia tulevaisuuteen.

Kuva 12. Esimerkkikoodin 9 luomat tietovirrat.

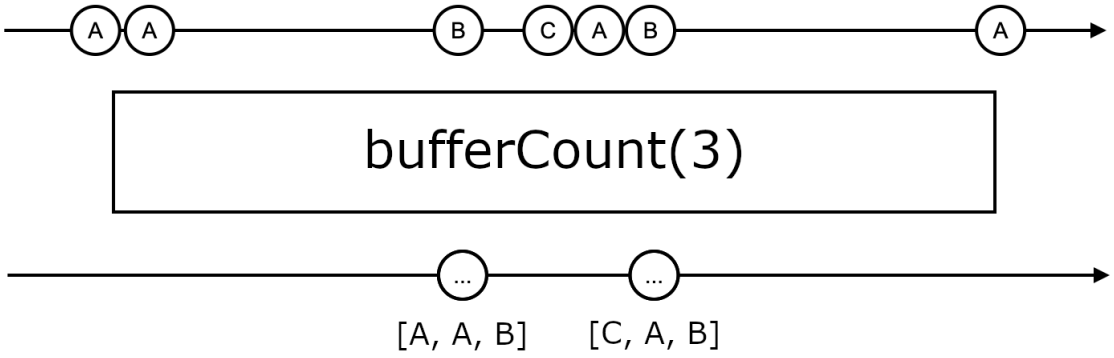
2.2.3 Muovaavat operaattorit

Muovaavat operaattorit muuttavat tietovirran arvoja. Niistä yleisin on map-operaattori, joka vaihtaa tietovirran arvon uuteen käyttäen takaisinkutsufunktiota (kuva 13).

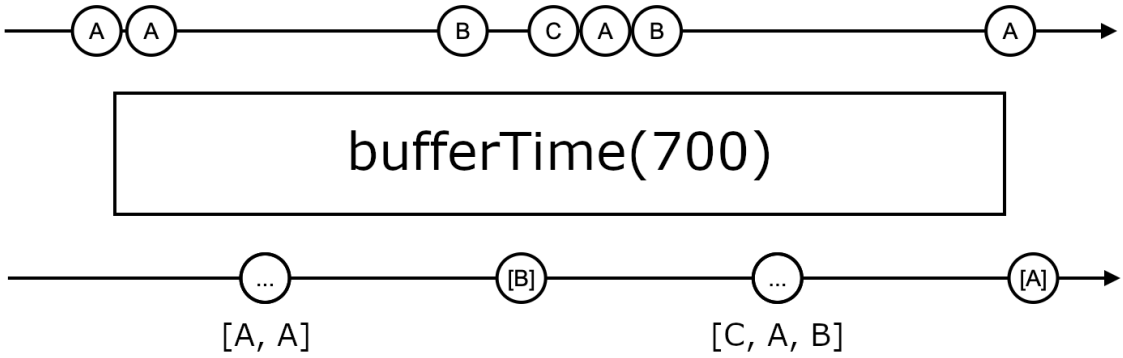


Kuva 13. Map-operaattori muuntaa tietovirran arvot. Kuvassa ylempi tietovirta kuvastaa tilannetta ennen operaattorin vaikutusta ja alempi sen jälkeen. Tämä diagrammirakenne on käytössä operaattorien vaikutusten visualisointiin RxJS-kirjaston omassa dokumentaatiossa. (API List.)

Puskurointiin tarkoitetut bufferTime ja bufferCount keräävät tietovirran arvoja ja lähettävät taulukon (kuvat 14 ja 15). Ne ovat hyödyllisiä esimerkiksi prosessoinnin jaksottamisessa tai arvojen liukuvan ikkunan luomisessa.



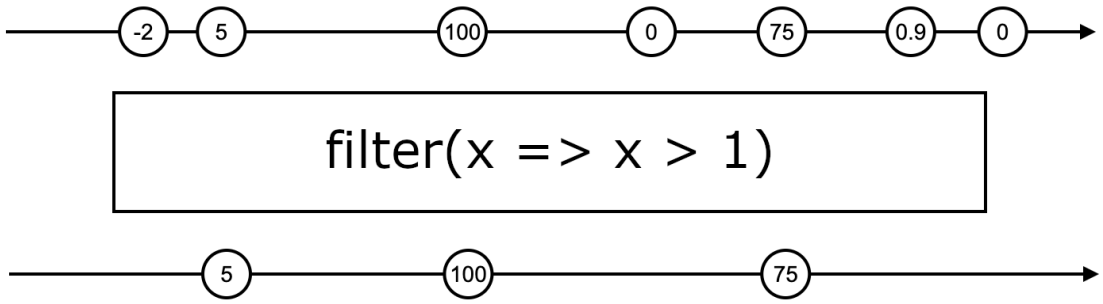
Kuva 14. BufferCount-operaattori kerää arvoja taulukkoon parametriksi annettuun lukumäärään saakka (API List).



Kuva 15. BufferTime-operaattori kerää arvoja taulukkoon parametriksi annetun millisekuntimäärän ajan (API List).

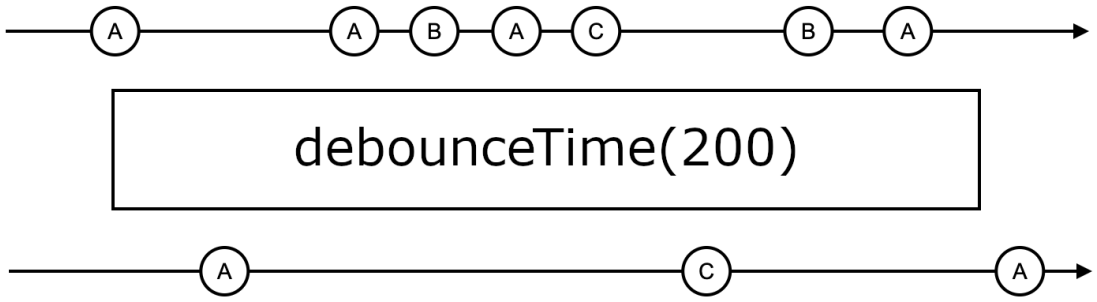
2.2.4 Rajaavat operaattorit

Rajaavat operaattorit poistavat arvoja tietovirrasta tai päättävät sen tietyin ehdoin. Filter-operaattori on yksinkertainen rajaava operaattori, joka karsii tietovirran arvot, jotka eivät täytä annetun takaisinkutsufunktion kriteeriä (kuva 16).



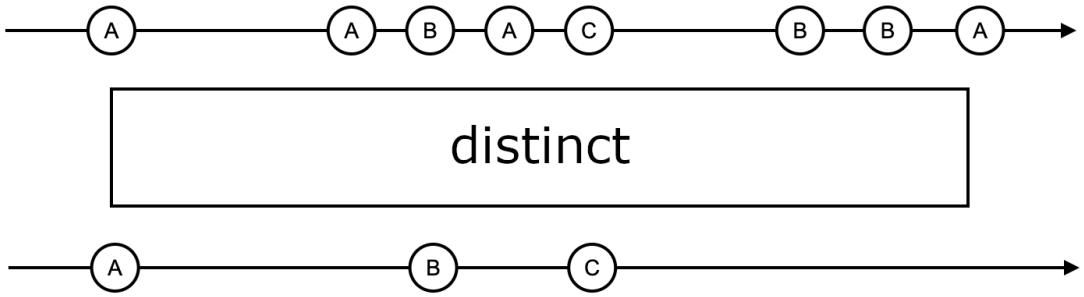
Kuva 16. Filter-operaattori suodattaa pois arvot, jotka ovat pienempiä kuin 1 (API List).

Debounce on prosessi, jossa lisätään viive syötteen havaitsemisen ja käsittelyn välille. Viiveen aikana muut syötteet jätetään huomiotta. Verkko-ohjelmoinnissa sitä käytetään yleensä käyttäjän syöttäessä tietoa näppäimistön kautta. Debounce-logiikka löytyy RxJS-kirjaston debounceTime-operaattorista (kuva 17).

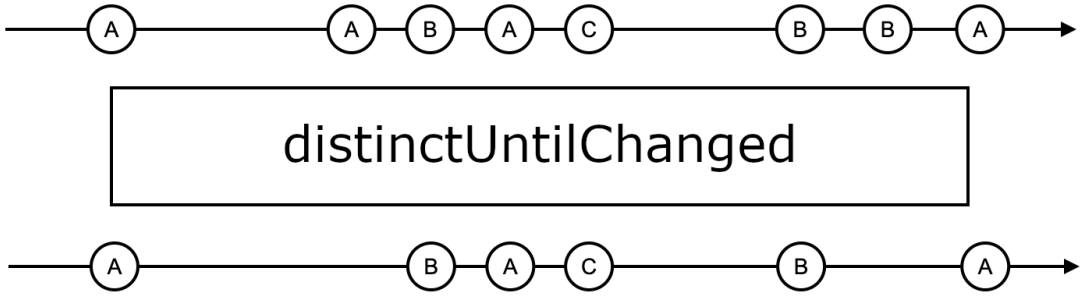


Kuva 17. DebounceTime-operaattori lisää parametriksi annetun viiveen millisekunteina ennen arvojen lähetystä (API List).

RxJS-kirjaston distinct- ja distinctUntilChanged-operaattorit suodattavat pois kaksoiskappaleet (kuvat 18 ja 19).

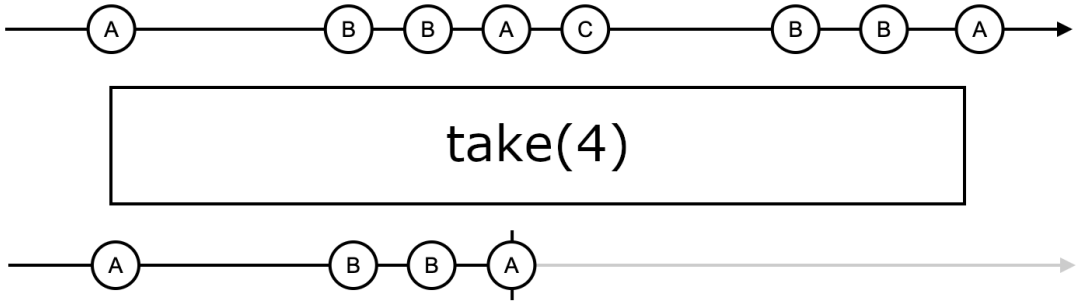


Kuva 18. Distinct-operaattori suodattaa pois tietovirran arvot, jotka vastaavat aikaisempia arvoja (API List).



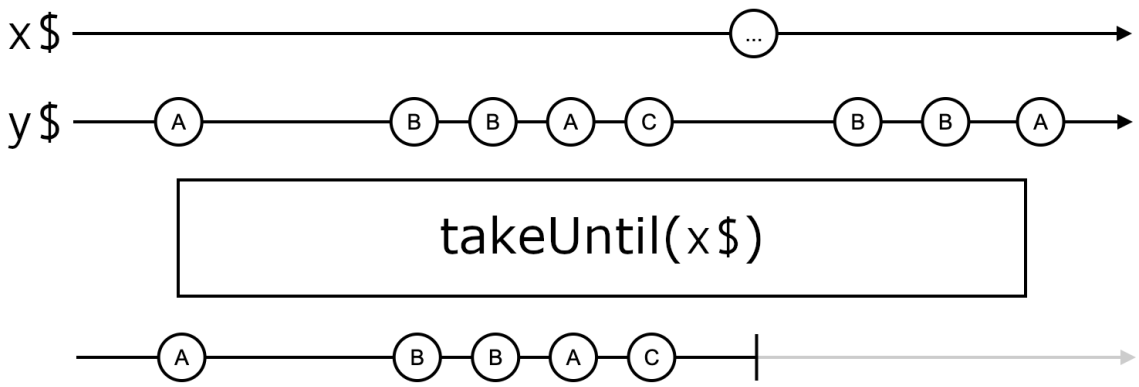
Kuva 19. DistinctUntilChanged-operaattori suodattaa pois ainoastaan tietovirran viimeisimmän arvon kaksoiskappaleet (API List).

Take-operaattori päättää tietovirran, kun se on lähettänyt operaattorin parametriksi annetun määrän arvoja (kuva 20).



Kuva 20. Take-operaattori päättää tietovirran neljännen arvon lähetyksen jälkeen (API List).

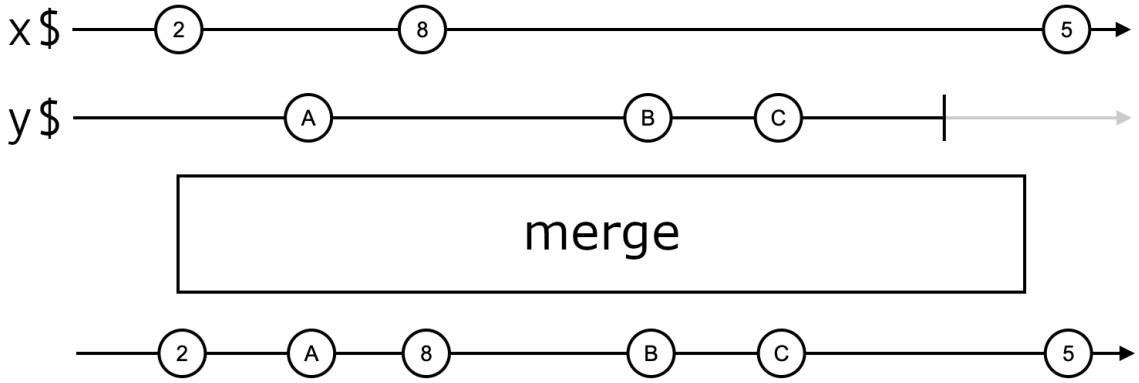
TakeUntil on versio take-operaattorista, joka ottaa parametrikseen toisen Observable-instanssin. Operaattori rekisteröityy sen kuuntelijaksi ja päättää tietoviran, kun toinen Observable lähettää arvon (kuva 21).



Kuva 21. TakeUntil-operaattori päättää y-virran x-virran lähettäessä arvon (API List).

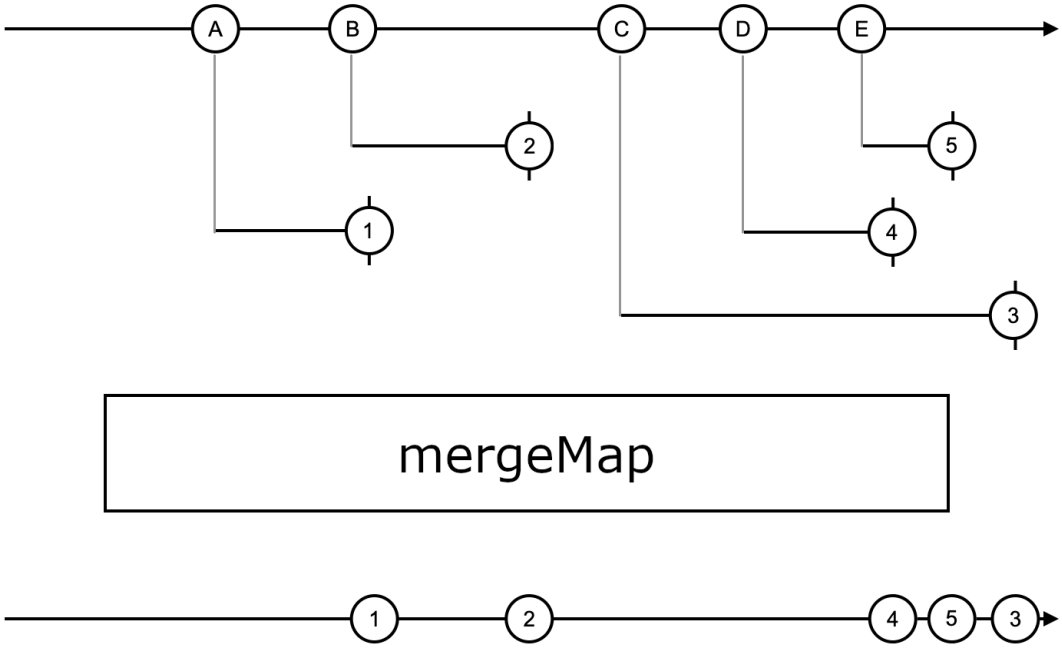
2.2.5 Yhdistävät operaattorit

Yhdistävien operaattorien perusajatuksena on ottaa kaksi tai useampia tietovirtoja ja yhdistää ne yhdeksi virraksi jonkin ehdon tai kriteerin perusteella. Merge-operaattori yhdistää kaksi tai useampia tietovirtoja yhdeksi (kuva 22). Sen luoma tietovirta päättyy kaikkien yksittäisten tietovirtojen päätyttyä.



Kuva 22. Merge-operaattori luo uuden tietovirran, joka koostuu usean tietovirran arvoista (API List).

MergeMap on yleinen yhdistävä operaattori, joka muuntaa tietovirran arvon toisen tietovirran lähettämäksi arvoksi (kuva 23). Se on hyödyllistä tilanteissa, joissa on tehtävä lisätietopyyntö kunkin alkuperäisen virran arvon perusteella.



Kuva 23. Tietovirran arvojen luomat tietovirrat yhdistetään mergeMap-operaattorilla (API List).

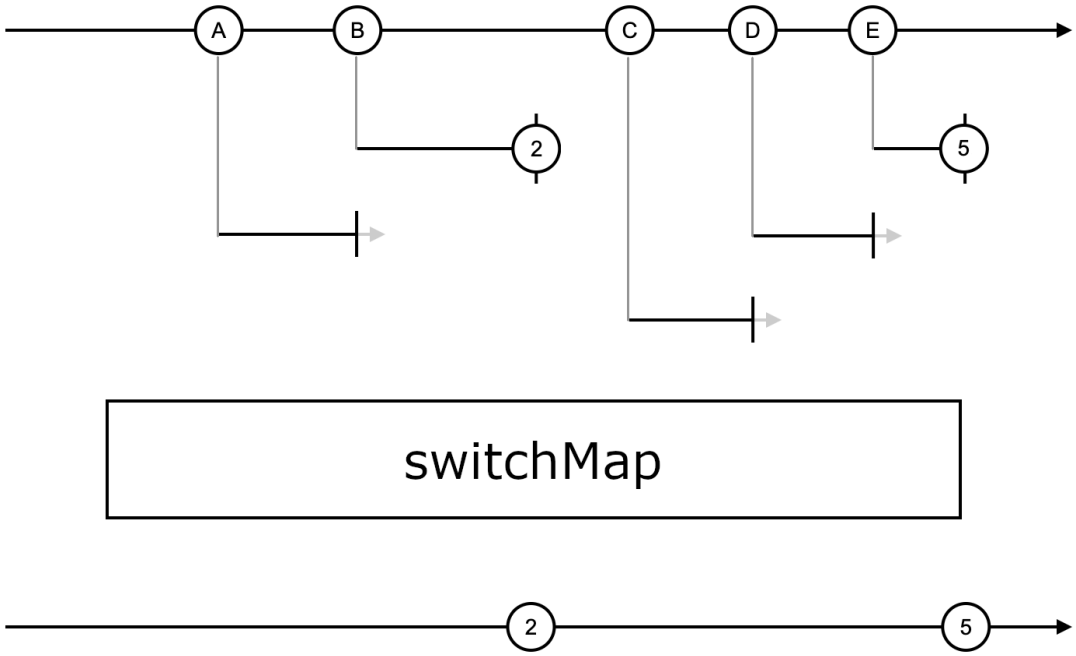
Esimerkkikoodi 10 toteuttaa hakukentän tuloksien virran, jossa käytetään mergeMap-operaattoria usean tietovirran yhdistämiseksi.

```
const searchField = document.getElementById("search");
const searchInput$ = fromEvent(searchField, "input");

const searchResults$ = searchInput$.pipe(
  debounceTime(50),
  map((event) => event.target.value),
  distinctUntilChanged(),
  mergeMap((text) => getHttp("/api/search?name=" + text));
);

searchResults$.subscribe((results) => {
  console.log("Tulokset:", results);
});
```

Esimerkkikoodi 10. Hakutuloksien virran toteutus. Koodissa kuunnellaan syöttökentän kirjoitustapahtumia ja lopulta ne muunnetaan hakutuloksiksi.

Esimerkkikoodin DebounceTime- ja distinctUntilChanged-operaattorit pitävät huolen siitä, ettei HTTP-pyyntöjä lähetetä jokaisen kirjoitustapahtuman jälkeen. getHttp-funktio ottaa parametrikseen osoitteen, johon HTTP-pyyntö lähetetään, ja palauttaa pyyntöä kuvaavan Observable-instanssin.

Esimerkin implementaatiossa on ongelma. Aikaisempia HTTP-pyyntöjä ei lopeteta käyttäjän kirjoittaessa. Ratkaisu on switchMap-operaattori, joka lopettaa aikaisemmat sisäiset tietovirtojen suoritukset (kuva 24).



Kuva 24. SwitchMap-operaattorin toiminta. B-arvon aloittama tietovirran suoritus lopettaa A-arvon tietovirran suorituksen. Myös C- ja D-arvojen tietovirrat päättyvät ennenaikaisesti (vrt. kuva 23). (API List.)

2.3 Monilähetys

Monilähetyksessä kuuntelijat vastaanottavat saman instanssin Observablen luomasta tietovirrasta. RxJS-kirjaston share-operaattori jakaa Observable-instanssin suorituksen kaikkien kuuntelijoiden kesken (API List). Share-operaattorilla

jaetun Observablen suoritus alkaa, kun ensimmäinen kuuntelija lisätään. Sen jälkeiset kuuntelijat eivät saa omaa kopiota tietovirrasta (esimerkkikoodi 11).

```
const numbers$ = interval(500).pipe(
  tap((number) => console.log("Lähetetään arvo " + number),
  share()
);

numbers$.subscribe((number) => {
  console.log("1. kuuntelija saa arvon " + number);
});

setTimeout(() => {
  numbers$.subscribe((number) => {
    console.log("2. kuuntelija saa arvon " + number);
  });
}, 500);
```

Esimerkkikoodi 11. Share-operaattorin käyttö monilähettävän Observablen luomiseksi.

Kuva 25. Esimerkkikoodin 11 toiminnan kuvaus. Tietovirran tap-operaattorin aiheuttama tulostus tapahtuu vain kerran, vaikka kuuntelijoita on kaksi (vrt. kuva 8).

Monilähetys on mahdollista myös RxJS:n Subject-luokalla (Subjects). Se on erityinen Observable-tyyppi, joka sallii arvojen työntämisen siihen sen lisäksi, että sitä voidaan kuunnella tavallisen Observable-tyypin lailla (esimerkkikoodi 12, kuva 26).

```
const numbers$ = new Subject<number>();

numbers$.next(5);

numbers$.subscribe((number) => {
  console.log("1. kuuntelija saa arvon " + number);
});

setTimeout(() => numbers$.next(10), 500);

setTimeout(() => numbers$.next(25), 1000);

setTimeout(() => {
  numbers$.subscribe((number) => {
    console.log("2. kuuntelija saa arvon " + number);
  });
}, 500);
```

Esimerkkikoodi 12. Subject-luokan käyttö. Sen next-funktio lähettää arvon kaikille Subject-olion senhetkisille kuuntelijoille.

Kuva 26. Esimerkkikoodin 12 toiminta. Subject-olio saa ensimmäisen kuuntelijansa arvon 5 lähetyksen jälkeen.

BehaviorSubject on Subjectin versio, joka alustetaan alkuarvolla (Subjects). Uudet kuuntelijat saavat välittömästi BehaviorSubjectin viimeisimmän arvon (esimerkkikoodi 13, kuva 27).

```
const numbers$ = new BehaviorSubject<number>(5);

setTimeout(() => {
  numbers$.subscribe((number) => {
    console.log("1. kuuntelija saa arvon " + number);
  });
}, 500);

setTimeout(() => numbers$.next(10), 1000);

setTimeout(() => {
  numbers$.subscribe((number) => {
    console.log("2. kuuntelija saa arvon " + number);
  });
}, 1500);
```

Esimerkkikoodi 13. BehaviorSubject-luokan käyttö. Luokan konstruktori ottaa parametrikseen alkuarvon. BehaviorSubjectin kuuntelijat vastaanottavat sen nykyisen arvon kuuntelun alussa.

Kuva 27. Esimerkkikoodin 13 toiminta. BehaviorSubject-olion kuuntelijat saavat kuuntelun alussa sen nykyisen arvon.

2.4 Virheenhallinta

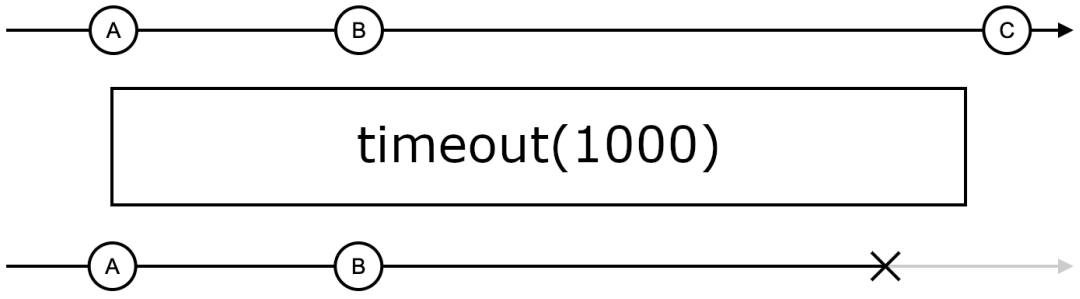
RxJS keskittää virheenhallinnan yhteen paikkaan välittämällä Observable-virtojen virheet kuuntelijoilleen subscribe-funktion kautta (esimerkkikoodi 14).

```
numbers$.subscribe({
  next: (number) => console.log("Arvo: " + number),
  error: (error) => console.error("Virhe:", error),
});
```

Esimerkkikoodi 14. Observablen subscribe-funktio ottaa parametrikseen vaihtoehtoisesti takaisinkutsufunktion sijasta JavaScript-olion, joka sisältää takaisinkutsufunktioita tietovirran eri viestityypeille. Esimerkin koodi lisää kuuntelijan arvo- ja virheviesteille.

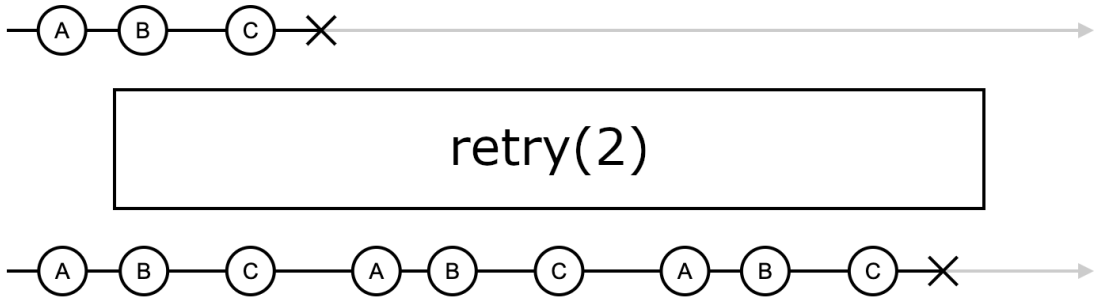
Aikakatkaisuja käytetään tilanteissa, joissa pyyntöihin odotetaan vastausta tietyssä ajassa. Jos vastaus ei saavu, ohjelma voi keskeyttää pyynnön ja ilmoittaa

käyttäjälle virheestä. RxJS-kirjaston timeout-operaattori päättää tietovirran virheeseen, jos se ei lähetä arvoa määräajassa (kuva 28).



Kuva 28. Timeout-operaattori päättää tietovirran virheeseen, jos se ei lähetä arvoa parametriksi annetussa määräajassa. Parametrin arvo on millisekunteina. (API List.)

Virhetilanteet voivat johtua verkkoyhteysongelmista tai muista ohimenevistä ongelmista, jotka voidaan ratkaista yrittämällä uudelleen. Retry-operaattori ottaa parametrikseen uudelleenyritysten lukumäärän ja välittää virheen eteenpäin vain, jos lukumäärä ylittyy (kuva 29).



Kuva 29. Retry-operaattori yrittää virhetilanteeseen johtanutta operaatiota uudelleen parametriksi annetun määrän verran (API List).

3 Reaktiivinen arkkitehtuuri Angular-sovellukselle

ActiveAhead-mobiilisovellus on Android- ja iOS -käyttöjärjestelmien sisäänrakennettua selainta käyttävä verkkosovellus, joka on kehitetty Googlen Angular-

ohjelmistokehyksellä. Siinä sovellukset koostuvat pienistä komponenteista, jotka on jaettu kolmeen osaan: rakenteeseen, tyylittelyyn ja ohjelmalogiikkaan (What is Angular? 2022). Insinöörityössä suunniteltiin arkkitehtuuri, jossa ActiveAhead-mobiilisovelluksen komponentit päivittyvät tiedon liikkuessa Bluetooth Mesh -verkossa.

3.1 Angular-komponentit

Angular-komponentit ovat JavaScript-luokkia, joihin on lisätty metatietoa TypeScriptin koristelijasyntaksin (engl. decorator) avulla (esimerkkikoodit 15 ja 16).

```
@Component({
  selector: "seconds",
  template: "<p>Seconds elapsed: {{ seconds }}</p>"
})
export class SecondsComponent implements OnInit {
  seconds = 0;

  ngOnInit(): void {
    setInterval(() => this.seconds++, 1000);
  }
}
```

Esimerkkikoodi 15. Yksinkertainen Angular-komponentti, joka laskee sekunteja.

```
<seconds></seconds>
```

Esimerkkikoodi 16. Esimerkkikoodin 8 komponentin käyttö HTML-tiedostossa.

Esimerkkikoodi 15 sisältää yksinkertaisen Angular-komponentin, joka laskee sekunteja. Sovelluskehyksen Component-koristelija sisältää komponentin asetukset. Selector-kenttä määrittää HTML-tunnisteen, jolla komponenttia käytetään muualla sovelluksessa (esimerkkikoodi 16).

Komponentin HTML-rakenne on määritelty template-kentässä. Esimerkin rakenteessa kahden aaltosulun sisällä oleva ”seconds” viittaa komponentin luokan seconds-instanssimuuttujaan. Angular sitoo instanssimuuttujan HTML-

rakenteeseen. Komponentti piirretään uudelleen arvon muuttuessa. (Introduction to components and templates 2022.)

Esimerkin ngOnInit-funktiota kutsutaan automaattisesti komponentin luontivaiheessa (Lifecycle hooks 2022). Siinä kutsuttu setInterval-funktio nostaa seconds-instanssimuuttujan arvoa yhdellä sekunnin välein. Esimerkin toteutus on puutteellinen, sillä seconds-muuttujan päivitystä ei lopeteta koskaan. Angularin ngOnDestroy-funktiota kutsutaan automaattisesti, kun komponentti poistuu sivulta (Lifecycle hooks 2022). Esimerkkikoodissa 17 komponentin toteutukseen on lisätty seconds-muuttujan päivityksen lopetus, jolloin komponentti lopettaa kaiken prosessointinsa tuhoamishetkellä.

```
export class SecondsComponent implements OnInit, OnDestroy {
  seconds = 0;

  private intervalId = -1;

  ngOnInit(): void {
    this.intervalId = setInterval(() => this.seconds++, 1000);
  }

  ngOnDestroy(): void {
    clearInterval(this.intervalId);
  }
}
```

Esimerkkikoodi 17. Seconds-muuttujan päivityksen lopetus ngOnDestroy-funktiossa, jota Angular kutsuu automaattisesti komponentin poistuessa sivulta.

Angular-sovelluskehys mahdollistaa riippuvuusinjektion komponenttiluokan konstruktorin kautta (esimerkkikoodi 18). Tällöin riippuvuusinjektiota tukevien luokkien instanssit tuodaan komponenttiin automaattisesti. (Dependency injection in Angular 2022.)

```
@Injectable()
export class ExampleService {
  doWork(): void { /* koodia */ }
}

@Component({
  selector: “dependency-test”,
  template: “<button (click)=”onClick()”>Button</button>”
})
export class DependencyTestComponent {
  constructor(private exampleService: ExampleService)

  onClick(): void {
    this.exampleService.doWork();
  }
}
```

Esimerkkikoodi 18. Riippuvuusinjektiota tukevan luokan luonti ja käyttö Angular-komponentissa.

Esimerkkikoodi 18 määrittää ExampleService-luokan, jonka instanssi tuodaan automaattisesti DependencyTest-komponenttiin. Sen doWork-funktiota kutsutaan komponentin HTML-rakenteessa sijaitsevaa nappia painamalla.

3.2 Reaktiiviset komponentit

Luvun 3.1 Seconds-komponentin toteutus toimii, sillä Angular tekee säännöllisiä tarkistuksia komponenttien instanssimuuttujien muuttumiselle. Toinen Angularin tarjoama muutostentarkistusstrategia on OnPush, jossa jatkuva automaattinen muutostentarkistus on poistettu käytöstä. Tällöin sovelluskehittäjä on itse vastuussa tarkistuksien tapahtumisesta sovelluksen tilan muuttuessa.

OnPush-strategian käyttö on helppoa RxJS-kirjaston Observablen avulla. Idea on yksinkertainen: jos komponentti toimii Observable-instanssin kuuntelijana, sillä on mahdollisuus päivittää esityksellisen osansa vain silloin, kun Observablen tietovirta lähettää arvon. Tällöin komponentin tilan muuttuminen ei vaadi säännöllisiä tarkistuksia. (Chebbi 2022.)

Esimerkkikoodi 19 toteuttaa Seconds-komponentin OnPush-strategialla. Kokonaislukumuuttujan sijasta käytössä on RxJS-kirjaston timer-funktiolla luotu

tietovirta, joka lähettää arvon 0 heti kuuntelun alkaessa ja seuraavat luvut sekunnin välein.

```
@Component({
  selector: "seconds",
  template: "<p>Seconds elapsed: {{ seconds$ | async }}</p>",
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class SecondsComponent {
  readonly seconds$ = timer(0, 1000);
}
```

Esimerkkikoodi 19. Seconds-komponentin toteutus OnPush-muutostentarkistusstrategialla.

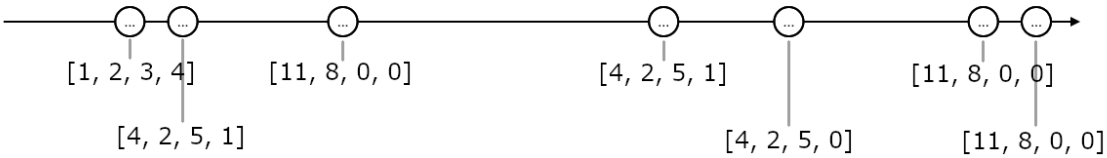
Esimerkkikoodin HTML-rakenteessa on uusi syntaksi tietovirran kuunteluun: "{{ seconds$ | async }}". Angularin async-syntaksi rekisteröi komponentin tietovirran kuuntelijaksi sen luontivaiheessa ja lopettaa kuuntelun komponentin poistuessa sivulta (Chebbi 2022). Täten tietovirran prosessointi päättyy komponentin poistuessa.

Esimerkkikoodin 19 komponenttia voidaan sanoa reaktiiviseksi. Sen esityksellinen osa, HTML-rakenne, päivittyy ainoastaan seconds$-tietovirran lähettäessä arvon. Huomattavaa on myös, että esimerkin komponentissa ei ole ngOnInit- tai ngOnDestroy-funktioita, mikä selkeyttää sen toteutusta.

Suurin osa ActiveAhead-mobiilisovelluksen komponenteista käyttää oletusarvoista muutostentarkistusstrategiaa. Insinöörityössä suunnitellussa arkkitehtuurissa esitettävä tieto on Observable-muodossa. Tällöin OnPush-strategia soveltuu hyvin käyttöön. Synergiaetuna sovelluksen suorituskyky paranee, kun säännöllinen muutostentarkastus on poistettu käytöstä (Chebbi 2022).

3.3 Bluetooth-verkon liikenteen kuuntelu

ActiveAhead-mobiilisovellus saa pääsyn ActiveAhead-verkkoon puhelimen käyttöjärjestelmän Bluetooth-toiminnallisuuden kautta. Jokainen verkon viesti on joukko tavuja eli kokonaislukuja välillä 0–255. Verkon liikenne on siis virta kokonaislukutaulukoita (kuva 30).



Kuva 30. ActiveAhead-verkon liikennettä kuvaava diagrammi. Diagrammin kokonaislukutaulukot eivät ole oikeita verkon viestejä.

Mesh-verkon kuuntelu vaatii yhdestä yhteen -yhteyden verkon laitteeseen. Yhdistetystä laitteesta on mahdollista muodostaa tietovirta käyttäen RxJS-kirjaston BehaviourSubject-ominaisuutta (esimerkkikoodi 20).

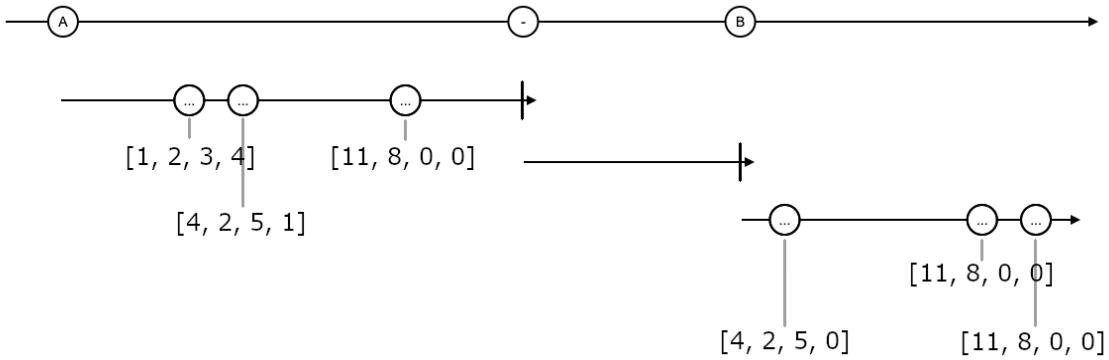
```
const connectedDevice$ =
  new BehaviourSubject<string | undefined>(undefined);

export function connect(id: string): void {
  connectedDevice$.next(id);
}

export function disconnect(): void {
  connectedDevice$.next(undefined);
}

export function getConnectedDevice(): Observable<string | undefined> {
  return connectedDevice$.asObservable();
}
```

Esimerkkikoodi 20. Pelkistetty rajapinta yhdistetyn laitteen noutamiseksi ja vaihtamiseksi. Esimerkin BehaviourSubject pitää sisällään merkkijonon eli laitteen yksilöllisen tunnisteen tai JavaScript-kielen undefined-arvon, joka edustaa määrittämätöntä arvoa. Esimerkissä undefined-arvo tarkoittaa tilannetta, jossa yhteyttä ei ole muodostettu.

Bluetooth-ominaisuuksien käyttöönotto verkkoteknologioihin pohjautuvassa mobiilisovelluksessa ei ole tämän raportin rajoissa. Bluetooth-verkon liikenteen kuuntelu tapahtuu esimerkkikoodissa 21 getMessages-funktion avulla, joka ottaa parametrikseen kuunneltavan laitteen yksilöllisen tunnisteen merkkijonona ja palauttaa verkon liikennettä kuvaavan Observablen. Esimerkin tietovirta saa kuuntelijan, joka lähettää verkon viestit prosessoitavaksi.

```
const messages$ = getConnectedDevice().pipe(
  switchMap((deviceId) => {
    if (deviceId === undefined) {
      return NEVER;
    }
    return getMessages(deviceId);
  }),
  map((bytes) => parseMessage(bytes)),
  share()
);

const meshSubscription = messages$.subscribe((message) => {
  processMessage(message);
});
```

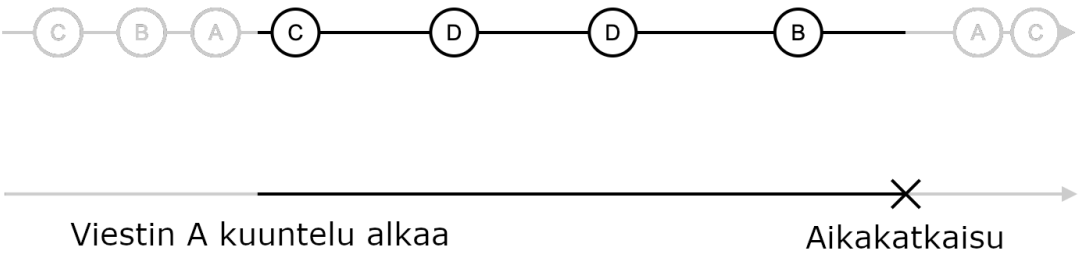
Esimerkkikoodi 21. Monilähettävä Observable, joka kuuntelee Bluetooth-verkon liikennettä yhdistetyn laitteen avulla. Esimerkissä oleva parseMessage-funktio muuntaa kokonaislukutaulukot JavaScript-olioiksi, ja processMessage-funktio lähettää ne käsiteltäväksi. NEVER on RxJS-kirjaston vakiomuuttuja. Se sisältää Observable-instanssin, joka ei lähetä arvoja (API List). Bluetooth-verkon liikenteen prosessointi lopetettaisiin kutsumalla meshSubscription-olion unsubscribe-funktiota.



Kuva 31. Esimerkkikoodin 21 tietovirran visualisointi.

Bluetooth-verkon tiedonkeruu on aktiivista nykyisessä arkkitehtuurissa. Mobiilisovellus kuulee kaikki välitinlaitteensa saamat viestit, mutta sivuttaa ne, mikäli viesteille ei ole tarvetta nykyisessä kontekstissa. Esimerkiksi sovelluksen saavuttama, yksittäisen laitteen parametrit sisältävä viesti ei vaikuta sovelluksen toimintaan millään tavalla, ellei sovellus ole sillä hetkellä tarkkailemassa liikennettä kyseistä viestiä varten. Kuvassa 32 on tilanne, jossa mobiilisovellus etsii viestiä A verkon liikenteestä kahden sekunnin ajan avatakseen käyttäjälle

uuden näkymän. Viesti saavuttaa mobiilisovelluksen ennen kuuntelua ja sen jälkeen. Koska viesti ei saavuta sovellusta rajatussa aikaikkunassa, tapahtuu aikakatkaisu, jolloin näkymän avaus epäonnistuu.

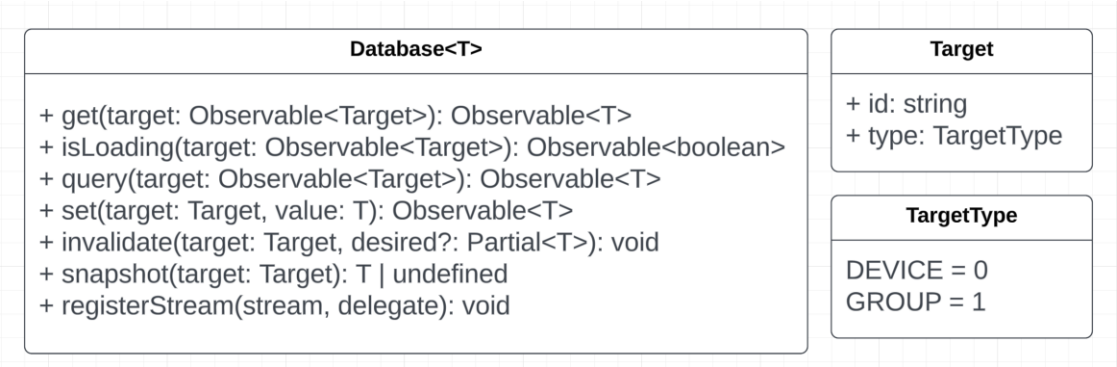


Kuva 32. Viestin A kuuntelu epäonnistuu, sillä se ei saavuta mobiilisovellusta halutussa aikaikkunassa.

Työssä toteutetussa arkkitehtuurissa verkkosovelluksen käyttöliittymän yksittäiset osat ovat vastuussa tarvitsemiensa tietojen kysymisestä. Lisäksi Bluetooth-verkon viestejä kuunnellaan jatkuvasti ja niiden sisältämä tieto päivittää sovelluksen tilan reaaliajassa. Täten tiedonkeruu on passiivista eikä mitään rajattua aikaikkunaa ole. Tarkoituksena on luoda käyttökokemus, jossa jokainen sovelluksen näkymä avautuu välittömästi ja näkymän yksittäiset komponentit päivittyvät sovelluksen kommunikoidessa Bluetooth-verkon kanssa.

3.4 Tietokannat

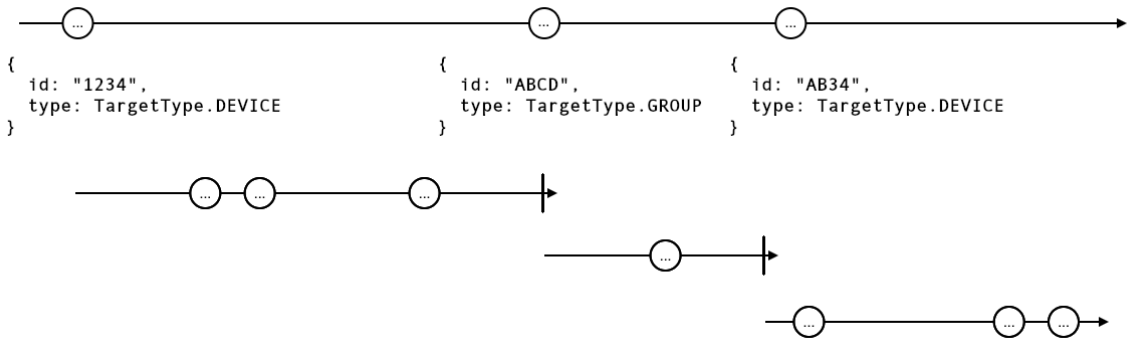
Suurin osa prosessoitavista verkon viesteistä sisältää yksittäisten laitteiden asetuksia. Työssä toteutettiin järjestelmä, jolla kaikki verkon viestien sisältämät asetukset päivitetään sovelluksen muistiin. Se toteutettiin tietokantaluokalla, joka tarjoaa rajapinnat tiedon hakemiseen ja kyselyyn sekä tietokannan rekisteröimiseen mesh-verkon tietovirtaan (kuva 33). Jokaisella parametrilla on oma tietokanta.



Kuva 33. Tietokantaluokan julkinen rajapinta.

Tietokantaluokan funktiot ottavat parametrikseen JavaScript-olion, joka kuvaa Bluetooth-verkon kohdetta. Target-olio sisältää merkkijonomuodossa olevan osoitteen sekä tiedon osoitteen tyypistä. Työssä keskityttiin kahden osoitetyypin implementoimiseen: yksittäisiin laitteisiin ja monien laitteiden muodostamiin ryhmiin. Tyyppi on määritelty oliossa kokonaislukuna: nolla tarkoittaa laitetta ja yksi ryhmää (kuva 33). ActiveAhead-mobiilisovelluksessa on mahdollista konfiguroida vain yhtä laitetta tai ryhmää samanaikaisesti. Täten nykyinen konfiguroitava kohde voidaan esittää tietovirtana.

Suurin osa tietokantaluokan funktioista ottaa parametrikseen virran Target-olioita. Tarkoituksena oli luoda skenaario, jossa tietokantaluokan rakentamat tietovirrat vaihtavat kuunneltavaa kohdetta uuden Target-olion saapuessa. Kuva 34 kuvastaa tätä toiminnallisuutta.



Kuva 34. Diagrammi kuvastaa tietokantaluokan funktioiden palauttamia tietovirtoja.

Luokan get-funktio palauttaa virran, jossa parametriksi annettu virta kohteita muunnetaan tietokannan nykyiseksi arvoksi. Jos kohteella ei ole arvoa, virta ei lähetä mitään, kunnes arvo ilmestyy tietokantaan. Get-funktion pääasiallinen käyttötarkoitus on nykyisten arvojen näyttäminen käyttöliittymässä käyttäen luvussa 3.2 mainittua Angularin async-syntaksia.

Tietokannan tietueilla on kaksi tilaa: ne ovat joko kelvollisia tai kelvottomia. Tietue on kelvollinen, kun sille on vastaanotettu arvo. Luokan isLoading-funktio palauttaa totuusarvojen tietovirran. Totuusarvo on tosi, kun tietueen arvo on kelvoton. Funktiota käytetään käyttöliittymässä Angularin async-syntaksilla kelvottomien arvojen piilottamiseksi ja konfiguroinnin estämiseksi, kunnes sovellus vastaanottaa kelvolliset arvot laitteesta.

Query-funktio eroaa get-funktiosta siten, että sen luomassa virrassa on sivuvaikutus: mikäli kohteella ei ole kelvollista arvoa, sitä kysytään Bluetooth-verkosta. Kysely jatkuu, kunnes kohde saa kelvollisen arvon.

Set-funktio ottaa parametrikseen yksittäisen kohteen ja arvon, joka lähetetään kohteelle. Funktio palauttaa tietovirran, joka päättyy, kun kohde on vastaanottanut lähetetyn arvon. Funktionaalisuus on toteutettu muuttamalla tietue kelvottomaksi ja kuuntelemalla kohteen tietuetta arvon lähetyksen jälkeen. Arvo todetaan vastaanotetuksi ja kelvolliseksi, kun kohteen tietueen arvo vaihtuu lähetetyksi arvoksi. Set-funktio toimii query-funktion kanssa. Kun set muuttaa tietueen kelvottomaksi, query aloittaa tiedon kyselyn automaattisesti. Tietue muuttuu kelvolliseksi halutun vastauksen saavuttua mobiilisovellukseen.

Invalidate-funktio muuttaa kohteen tietueen kelvottomaksi. Se ottaa vaihtoehtoisena parametrina tietokannan arvon, joka vaaditaan tietueen muuttamiseksi kelvolliseksi. Set-funktio käyttää invalidate-funktiota. Se on käytössä paikoissa, joissa lähetetty komento vaikuttaa moniin tietueisiin. Esimerkkinä tästä on laitteen tehdasasetusten palautus. Invalidate-funktiota käytetään kaikkiin parametreihin, joihin se vaikuttaa, ja funktion parametriksi annetaan tehdasasetukset.

Tällöin käyttöliittymän käyttämä isLoading-funktio piilottaa arvot ja query-funktio lähettää kyselyjä, kunnes arvot vastaavat tehdasasetuksia.

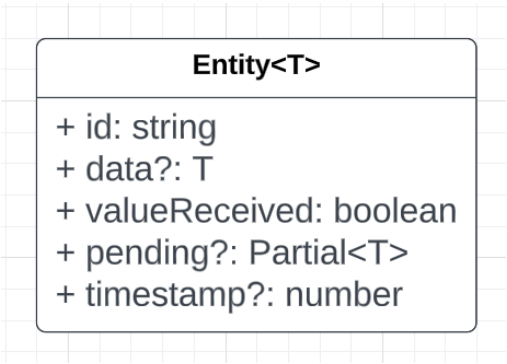
Tietokannan snapshot-funktio on imperatiivinen tapa saada kohteen arvot kutsuhetkellä. Funktio palauttaa JavaScript-ohjelmointikielen undefined-arvon, jos arvoa ei ole olemassa.

RegisterStream-funktio lisää tietokannan tietovirran kuuntelijaksi. Se saa parametrikseen kuunneltavan virran ja takaisinkutsufunktion, jota kutsutaan viestin saapuessa tietovirtaan. Takaisinkutsufunktio saa parametrikseen saapuneen viestin sekä apufunktioita tietokannan päivittämiseen (esimerkkikoodi 22).

```
LightLevelsDatabase.registerStream(meshStream, (message, { set }) => {
  if (isLightLevelsMessage(message)) {
    set(message.targetAddress, {
      maxLevel: message.maxLevel,
      transitionLevel: message.transitionLevel,
      minimumLevel: message.minimumLevel
    });
  }
}
```

Esimerkkikoodi 22. Valotasojen tietokanta rekisteröidään mesh-verkon liikenteen kuuntelijaksi. Jos saapunut viesti sisältää laitteen valotasot, ne lisätään tietokantaan. Näin tietokanta pysyy ajan tasalla.

Tietokanta sisältää joukon avain-arvopareja, joissa avaimet ovat yksittäisten laitteiden osoitteita ja arvot tietueita, jotka sisältävät Bluetooth-verkosta saadun arvon ja siihen liittyvää metatietoa (kuva 35).



Kuva 35. Tietokannan tietueen rajapinta.

Tietueen data-kenttä sisältää viimeisimmän Bluetooth-verkosta vastaanotetun arvon. ValueReceived-totuusarvokenttä on tosi, kun viesti on vastaanotettu Bluetooth-verkosta. Tietue on kelvoton, jos kentän arvo on epätosi. Invalidate-funktio vaihtaa valueReceived-totuusarvon epätodeksi koskematta data-kenttään. Pending-kenttä sisältää set-funktiolla lähetetyn arvon. Tietue on kelvoton, jos pending-kentän arvo ei ole sama kuin data-kentän arvo. Timestamp-kenttä sisältää viimeisimmän arvon saapumishetken Unix-aikaleimana eli määränä millisekunteja 1. tammikuusta 1970 alkaen. Tietue on kelvoton, jos viimeisin arvo on saapunut yli viisi minuuttia sitten. Tietueen kelpoisuuden kuvaava algoritmi on liitteessä 1.

Mobiilisovelluksen nykyisessä arkkitehtuurissa ei käytetä välimuistia. Yksittäisen laitteen konfigurointinäkymän avaus vaatii kaikkien parametrien kysymisen, mikä tehdään avaushetkellä. Lähestymistavan hyvä puoli on se, että näytetyt tiedot ovat aina ajantasaisia. Kun monta käyttöönottoinsinööriä työskentelee samassa tilassa, voi tulla tilanteita, joissa laitteen tiedot ovat muuttuneet toisen käyttöönottoinsinöörin toimesta, mutta välimuistia käyttävä mobiilisovellus ei tiedä siitä ja täten näyttää omassa välimuistissaan olevia vanhoja tietoja.

Työssä toteutettu välimuisti on voimassa viisi minuuttia. Kesto katsottiin riittävän pitkäksi nopeuttamaan sovelluksen toimintaa tilanteissa, joissa käyttäjä avaa laitteen useita kertoja lyhyessä ajassa. Kestoaika mahdollistaa vanhojen tietojen näyttämisen, mutta lyhyt kesto minimoi sen. Arkkitehtuurin tuotteistaminen vaatii tältä kannalta vielä lisää pohdintaa.

3.5 Riippuvuudet

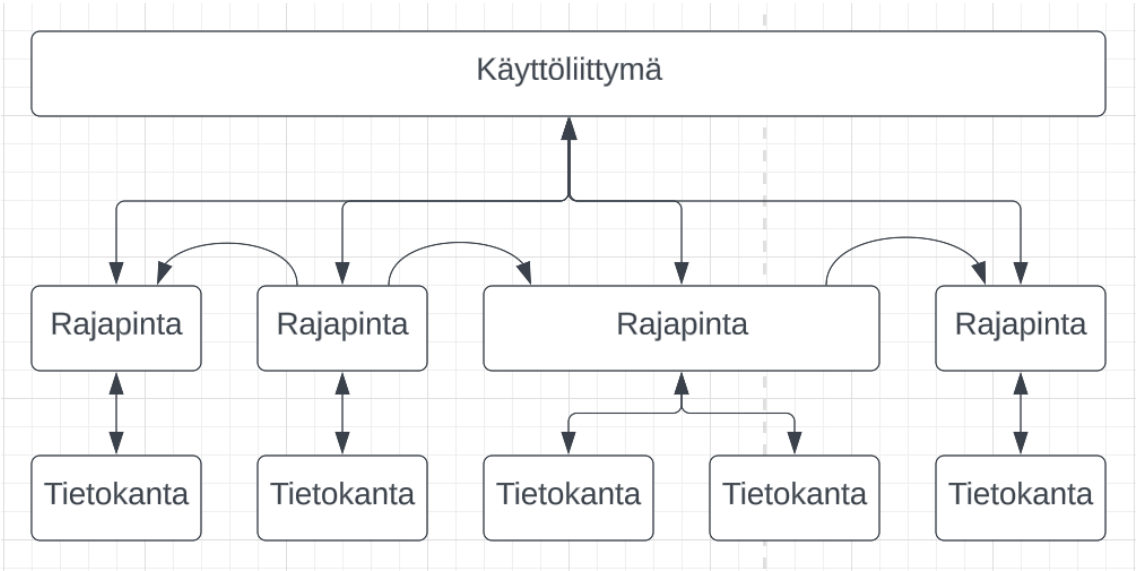
Joillain ActiveAhead-laitteiden parametreista on riippuvuuksia toisiin. Esimerkkinä tästä ovat yksittäisen laitteen valotasot. ActiveAhead-järjestelmä tukee monikanavaisia valaisimia, joilla on ylä- ja alavalo (kuva 36). Tässä tapauksessa valotasoja on kaksi ja niiden kysyminen ja asetus tehdään eri tavalla kuin muilla valaisimilla. (User Guide: Operation of Helvar ActiveAhead mobile application 2022.)

Levels	
Occupied	80%
Power save	20%
Minimum	0%

Levels	# 1	# 2
Occupied	100%	100%
Power save	50%	100%
Minimum	0%	0%

Kuva 36. Tavallisen yksikanavaisen valaisimen valotasot (vasemmalla) ja kaksikanavaisen valaisimen valotasot (oikealla) mobiilisovelluksessa.

Parametrien riippuvuudet ratkaistiin abstrahoimalla tietokannat rajapintaluokilla, jotka ovat vastuussa riippuvuuksien selvittämisestä. Mobiilisovelluksen käyttöliittymä kommunikoi tietokantojen sijasta rajapintaluokkien kanssa (kuva 37).



Kuva 37. Tietokantojen ja rajapintaluokkien hierarkia. Ainoastaan rajapintaluokat ovat suoraan yhteydessä tietokantoihin, ja ne ovat mahdollisesti myös yhteydessä toisiin rajapintaluokkiin.

3.6 Tiedon päivittäminen käyttöliittymässä

Työssä implementoitiin Angular-komponentti, joka näyttää latausanimaation tietokannan tietueen ollessa kelvoton. Se toteutettiin Angularin Input-ominaisuudella, joka mahdollistaa parametrien syöttämisen komponenteille. Esimerkkikoodi 23 sisältää komponentin toteutuksen.

```
@Component({
  selector: “loading-container”,
  template: `
  <loading-animation
    *ngIf=”loading$ | async; else content”
  ></loading-animation>
  <ng-template #content>
    <ng-content></ng-content>
  </ng-template>
`,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class LoadingContainerComponent {
  @Input()
  loading$: Observable<boolean> = of(true);
}
```

Esimerkkikoodi 23. Angular-komponentti, joka näyttää latausanimaation, kun komponentin parametriksi annettu totuusarvojen virta lähettää arvon true.

Esimerkkikoodin 23 LoadingContainer-komponentin HTML-rakenteessa on kaksi uutta Angular-konseptia. *ngIf sitoo elementtiin ehdon sen näyttämiseksi. Elementti lisätään verkkosivulle, jos ehto täyttyy. Muussa tapauksessa Angular näyttää ”content”-avainsanalla merkityn elementin. Sen sisältämä ng-content määrittää sijainnin, johon komponentille annettu HTML-rakenne piirtyy. Esimerkkikoodi 24 määrittää uuden komponentin, joka käyttää LoadingContainer-komponenttia.

```
@Component({
  selector: "example",
  template: `
  <loading-container [loading$]="dataLoading$">
    Data: {{ data$ | async }}
  </loading-container>
`,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ExampleComponent {
  public readonly dataLoading$: Observable<boolean>;
  public readonly data$: Observable<Data>;

  constructor(
    currentTargetService: CurrentTargetService,
    databaseService: DatabaseService
  ){
    const target$ = currentTargetService.get();
    this.dataLoading$ = databaseService.isLoading(target$);
    this.data$ = databaseService.get(target$);
  }
}
```

Esimerkkikoodi 24. LoadingContainer-komponenttia käyttävä Angular-komponentti.

Esimerkin komponentti saa nykyisten kohteiden eli Target-olioiden virran käyttöönsä CurrentTargetService-luokan kautta. DatabaseService on rajapintaluokka tietokannalle. Komponentti esittää nykyisen kohteen tietokannasta saadun arvon. Se rekisteröityy ensin dataLoading-virran kuuntelijaksi ja myöhemmin data$-virran kuuntelijaksi tietokannan arvon löytyessä ja muuttuessa kelvolliseksi.

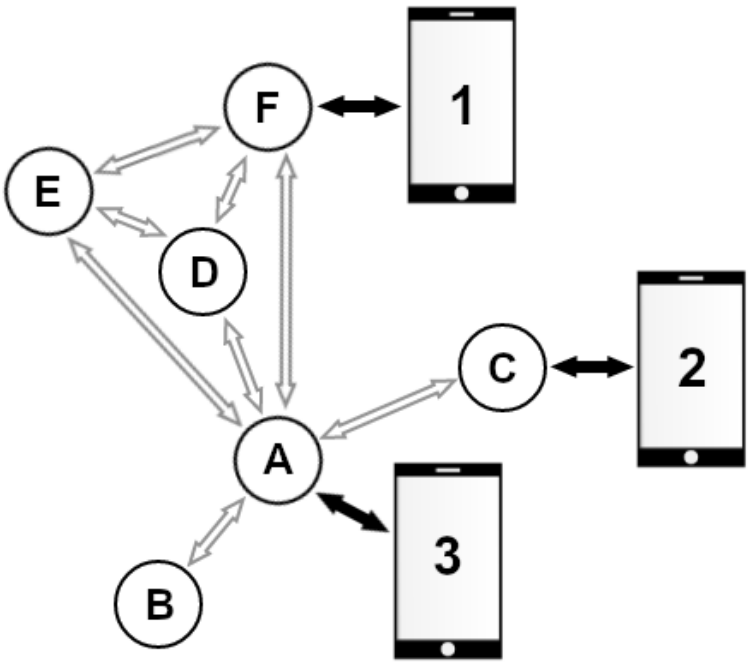
Esimerkistä puuttuu tiedon kysely. Data-muuttujan virran luominen tietokannan query-funktiolla ei ole tässä tapauksessa mahdollista, sillä komponentti rekisteröityy virran kuuntelijaksi ja käynnistää sen vasta kelvollisen arvon saapuessa. Komponentti ei siis koskaan kysyisi arvoa, jos sitä ei ole. Query-funktiota täytyy siis kutsua HTML-rakenteen ulkopuolella komponentin luokassa. Funktion palauttaman tietovirran kuuntelu lopetetaan RxJS-kirjaston Subject-oliolla, joka lähettää signaalin komponentin tuhoutuessa (esimerkkikoodi 25).

```
private readonly destroyed$ = new Subject<void>();

constructor(
  currentTargetService: CurrentTargetService,
  databaseService: DatabaseService
){
  // ...
  databaseService.query(target$).pipe(
    takeUntil(this.destroyed$)
  ).subscribe();
}

ngOnDestroy(): void {
  this.destroyed$.next();
}
```

Esimerkkikoodi 25. Tietokannan arvon kysely tietueen muuttuessa kelvottomaksi. Komponentissa luotu query-funktion virran prosessointi päättyy komponentin tuhoutumishetkellä.

3.7 Lähistöllä tapahtuvan käyttöönoton kuuntelu

Kuvassa 38 käyttöönottoinsinööri 1 on muodostanut yhteyden verkon laitteeseen F. Hän konfiguroi laitteen D, jolloin mobiilisovellus kirjoittaa viestejä verkkoon laitteen F kautta.



Kuva 38. Kolme käyttöönottoinsinööriä on työskentelemässä samalla työmaalla. Jokainen on muodostanut yhdestä yhteen -yhteyden verkon laitteeseen.

Luvussa 3.4 mainittu tietokantaluokan query-funktio kirjoittaa verkkoon kysymyksiä parametrien tallennuksen varmistamiseksi. Laitteen D vastaukset saavuttavat laitteen F lisäksi laitteet A ja C, jotka ovat samalla työmaalla työskentelevien käyttöönottoinsinöörien välitinlaitteita. Tällöin kaikkien kolmen käyttöönottoinsinöörin sovelluksen tila päivittyy laitteen D uusilla parametreilla.

Samassa tilassa olevat sovellukset päivittyvät ainoastaan, jos ne ovat lähetetyn vastauksen kuuloetäisyydellä. Kaikkien rakennuksessa olevien sovellusten päivittyminen etäisyyksistä huolimatta vaatisi tiedonjakamisesta vastuussa olevan palvelimen.

Mobiilisovelluksen käyttöliittymän liukusäätimet (kuva 39) ja muut interaktiiviset osat eivät saisi vaihtaa arvoja kesken konfiguroinnin, vaikka kyseessä olisikin törmäystilanne, jossa kaksi käyttöönottoinsinööriä konfiguroi samaa laitetta. Työssä käyttöliittymän osille annettiin kopiot nykyisistä arvoista, jotka eivät päivity Bluetooth-verkon liikenteen kautta. Työssä toteutettu arkkitehtuuri mahdollistaisi varoituksen näyttämisen käyttäjälle mahdollisen törmäystilanteen tapahtuessa.

Kuva 39. Mobiilisovelluksen liukusäätimet valotasoille.

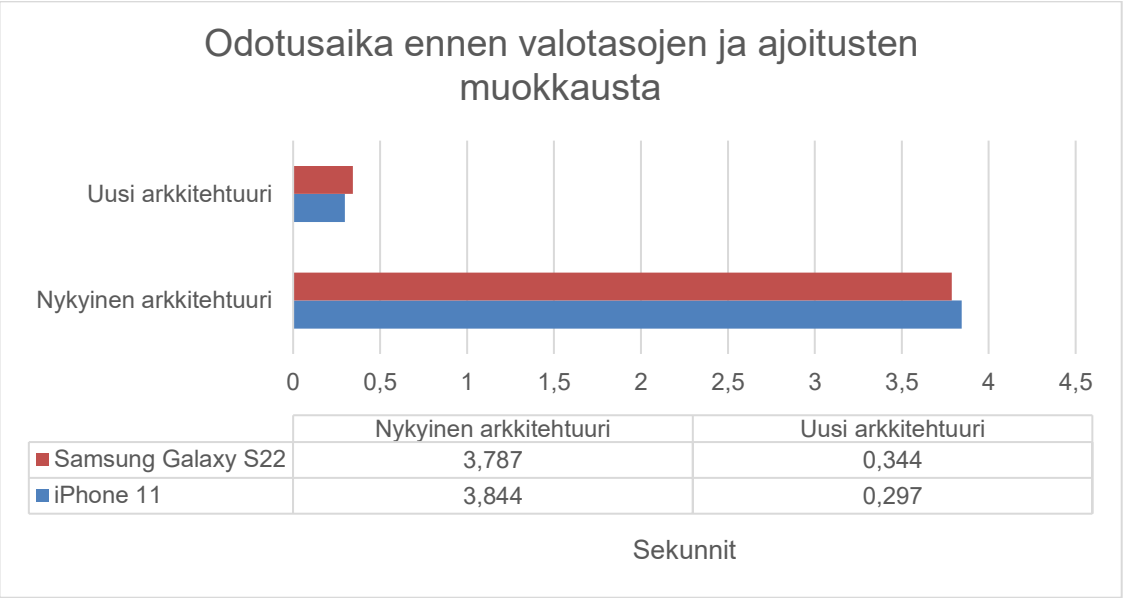
4 Lopputulos ja vertailu

Insinöörityössä implementoitiin yksittäisten laitteiden tiedon päivittäminen passiivisesti viestien liikkuessa Bluetooth-verkossa sekä käyttöliittymän päivittäminen reaaliajassa. Työssä tehdyt muutokset paransivat sovelluksen käytettävyyttä. Se todettiin mittaamalla odotusaika ennen yksittäisen laitteen

valotasojen ja ajoitusten muokkaamista. Parametrit valittiin, koska ne ovat järjestelmän keskeisiä ominaisuuksia.

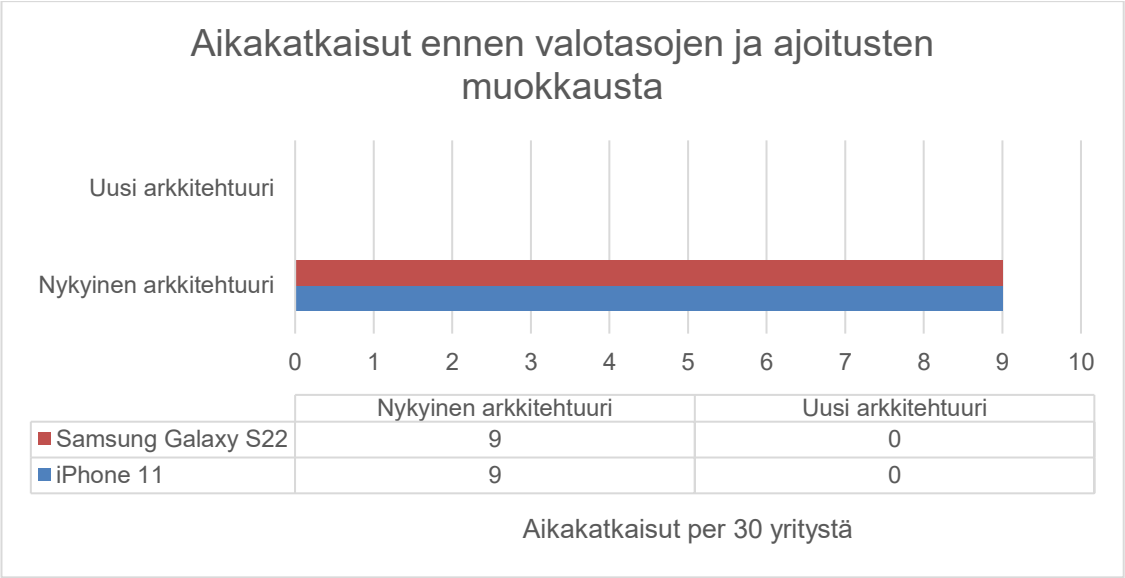
Mittaus tehtiin kehitysympäristössä, jossa ActiveAhead-verkon liikenne on tavanomaista vilkkaampaa. Siinä käytettiin Samsung Galaxy S22- ja iPhone 11 -puhelimia uudella ja nykyisellä arkkitehtuurilla. Kohdelaite avattiin 30 kertaa, ja keskiarvo laskettiin kaikista odotusajoista. Uuden arkkitehtuurin välimuisti poistettiin käytöstä mittauksen ajaksi, jotta laitteen arvot kysyttäisiin aina.

Nykyisessä arkkitehtuurissa oli keskimäärin 3,8 sekunnin odotusaika ennen valotasojen ja ajoitusten muokkausta, ja uudessa arkkitehtuurissa noin 0,3 sekuntia (kuva 40).



Kuva 40. Odotusaika sekunteina ennen laitteen parametrien muokkausta. Se on uudessa arkkitehtuurissa yli kymmenen kertaa lyhyempi.

Lisäksi nykyisessä arkkitehtuurissa tapahtui aikakatkaisu yhteensä kahdeksantoista kertaa kuudestakymmenestä (kuva 41). Yksittäiset mittaustulokset ovat liitteessä 2.



Kuva 41. Nykyisessä arkkitehtuurissa tapahtui yhteensä 18 aikakatkaisua mittausten aikana. Uudessa arkkitehtuurissa aikakatkaisuja ei ollut.

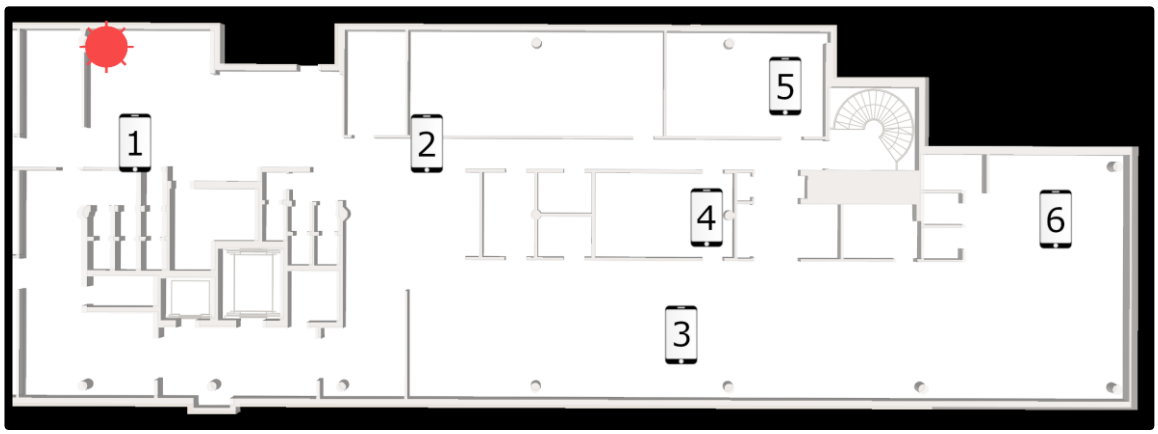
Nykyisessä arkkitehtuurissa tapahtuvat aikakatkaisut kertovat pullonkaulasta, joka aiheutuu monen viestin lähettämisestä samanaikaisesti Bluetooth-verkkoon (Aijaz ym. 2021). Työn arkkitehtuurin lyhyempi odotusaika ja nopea käyttöliittymä parantavat käyttökokemusta. Se sai positiivista palautetta Helvarin käyttöönottoinsinööreiltä.

Odotusaikoja olisi mahdollista lyhentää entisestään kysymällä käyttäjän läheisten laitteiden parametrit taustalla. Lisäksi mobiilisovellus voisi kysyä parametreja käyttäjän selatessa sovelluksen laiteluetteloa. Tällöin sillä olisi tärkeimmät tiedot, kuten valotasot ja ajoitukset, välimuistissa jo ennen laitteen avausta.

Teknisen työosuuden päätteeksi mitattiin myös luvussa 3.7 kuvailtua läheisten mobiililaitteiden synkronointia. Mittauksessa käytettiin seuraavia päätelaitteita:

1. Motorola One Hyper (Android 11)
2. iPhone 13 mini (iOS 16.3)
3. Samsung Galaxy S22 (Android 12)
4. iPhone SE (2020, iOS 14.8)
5. Samsung Galaxy A20 (Android 11)
6. iPhone 11 (iOS 16.2).

Motorola One Hyper lähetti kohdelaitteelle uudet valotasot ja ajoitukset sata kertaa käyttäen luvussa 3.4 mainittua tietokannan set-funktiota. Se oli tehnyt yhdestä yhteen -yhteyden toiseen verkon laitteeseen, joka oli noin viiden metrin päässä kohdelaitteesta. Muut viisi puhelinta sijoitettiin ympäri Helvarin toimistoa (kuva 42). Tutkimuksessa mitattiin, kuinka suuren osan lähetetyistä parametreista ne vastaanottavat.

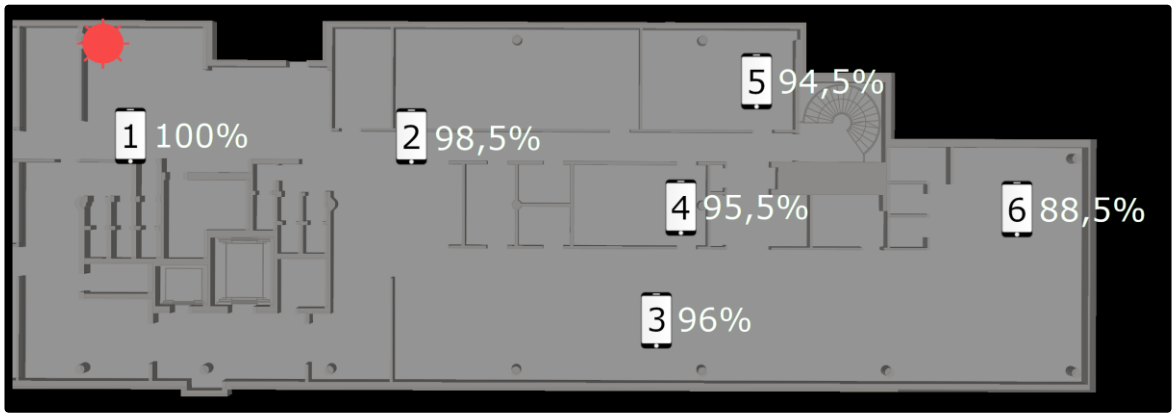


Kuva 42. Puhelinten käyttämien välitinlaitteiden sijainnit Helvarin toimistossa mittauksen aikana. Konfiguroitava kohdelaite on kartalla vasemmassa yläkulmassa.

Selvää oli, että laite 1, Motorola One Hyper, vastaanottaisi kaikki parametrit, sillä työssä toteutetun arkkitehtuurin parametrien lähetys sisältää niiden kysymisen uudestaan kohdelaitteelta. Oletuksena oli, että laitteet 2–5 saisivat keskimäärin 90 % lähetetyistä arvoista ja kaukaisin laite 80 %. Parametrien lähetys sisältää mahdollisesti arvojen kysymisen monta kertaa, mikäli parametrit eivät saavuttaneet kohdelaitetta tai lähettävä laite ei saa vastausta parametrien

kyselyyn. Se parantaa muiden laitteiden mahdollisuutta vastaanottaa uudet parametrit.

Parametrien lähetys kesti tasan minuutin, eli laite 1, Motorola One Hyper lähetti uudet parametrit kohdelaitteelle 0,6 sekunnin välein. Kohdelaitteesta kaukaisin puhelin, toimiston toisella puolella oleva iPhone 11 vastaanotti 89 % kaikista lähetetyistä parametreista. Muut laitteet vastaanottivat 90–100 % parametreista (kuva 43). Kaikki mittaustulokset ovat liitteessä 3.

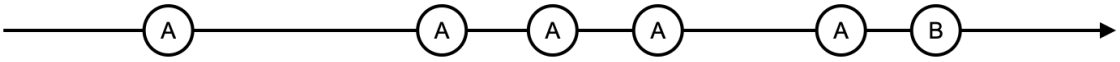


Kuva 43. Puhelinten vastaanottamien arvojen osuudet.

Parametrien lähetyksen jälkeen kaikki kuusi laitetta olivat vastaanottaneet viimeiseksi lähetetyt parametrit, ja ne olivat näkyvissä sovelluksen käyttöliittymässä. Käyttöönoton kuuntelun luotettavuus voisi mahdollistaa aggressiivisemman välimuistitallennuksen. Mobiilisovellus tietää karkean etäisyyden kuhunkin laitteeseen ja voisi tyhjentää yksittäisen laitteen välimuistin, jos etäisyys muuttuu huomattavasti.

Työssä toteutetussa arkkitehtuurissa mobiilisovellus kerää tietoja passiivisesti. Se osoittautui toimivaksi yksittäisten laitteiden kanssa, mutta ongelmia ilmeni, kun työskenneltiin laiteryhmien kanssa. Ryhmille suunnattuihin viesteihin saatiin vastauksia, joissa oli ryhmän osoite eikä ryhmille kuuluvien laitteiden omia osoitteita. ActiveAhead-järjestelmän ryhmiin liittyvät parametrit voidaan kysyä ja asettaa vain koko ryhmälle kerrallaan (User Guide: Operation of Helvar ActiveAhead mobile application 2022). Ryhmän jäsenten parametrit voivat

vaihdella, joten ryhmälle suunnattu kysely voi tuottaa kuvan 44 mukaiset vastaukset.



Kuva 44. Sarja ryhmälle suunnatun kyselyn vastauksia. Jokainen saapuu saman ryhmän jäseneltä.

Kaikki kuusi kuvan 44 vastausta saapuvat eri laitteista, mutta vastauksissa on sama ryhmäosoite ja niiden sisältämät arvot vaihtelevat. Työssä toteutettu passiivinen tiedonkeruu johtaisi sovelluksen käyttöliittymän päivittymiseen aluksi arvolla A ja sitten arvolla B. Jos ryhmän jäsenen arvot eroavat muista ja ne saapuvat viimeisenä, ne päivittyvät sovelluksen tietokantaan koko ryhmän arvona, mikä luo harhaanjohtavan kuvan ryhmän tilasta.

Työssä toteutetun tietokannan lähetysalgoritmi olettaa ryhmälle lähetettyjen parametrien tallentuneen, jos yksikin ryhmän jäsen vastaa parametrien kyselyyn lähetetyillä arvoilla. On mahdollista, että jokin ryhmän jäsen on siitä huolimatta jäänyt vastaanottamatta arvoja.

Ryhmien parametrien kuuntelu ja lähetys vaatisi tietokantaluokan pilkkomista pienempiin osiin siten, että yksittäisillä tietokannoilla olisi mahdollisuus valita oma strategiansa tiedon kuuntelulle ja lähetykselle. Ryhmien tiedon kuuntelu saattaisi olla mahdollista tehdä taustalla käyttämällä puskurointia; nykyisen strategian sijasta tietokantaan päivitettäisiin tietyllä aikavälillä kerätyt arvot listana. Tiedon lähetys tarvitsee tiedon ryhmän jäsenistä, jolloin ryhmälle lähetyt parametrit varmistettaisiin jokaiselta ryhmän jäseneltä yksitellen.

Ihanteellinen ratkaisu olisi mahdollisuus vastaanottaa yksittäisten laitteiden vastaukset ryhmäkohtaisiin parametreihin. Tällöin kaikki parametrit voitaisiin kerätä passiivisesti.

5 Yhteenveto

Opinnäytetyössä suunniteltiin ja implementoitiin Bluetooth-verkon kanssa kommunikointia varten arkkitehtuuri, jossa mobiilisovelluksen tila päivittyy passiivisesti viestien liikkuessa Bluetooth-verkossa. Arkkitehtuuri todettiin toimivaksi yksittäisten laitteiden kanssa: odotusaika ennen laitteen parametrien muokkaamista laski alle kymmenesosaan. Uuden arkkitehtuurin jatkuva mesh-verkon kuuntelu ja tilan päivitys mahdollistavat tiedonkeruun lähellä olevien käyttäjien tekemistä muutoksista.

Reaktiivinen ohjelmointi on ohjelmointia asynkronisilla tietovirroilla. RxJS-kirjasto mahdollistaa asynkronisten tietovirtojen helpon käsittelyn tarjoamalla laajan valikoiman operaattoreita, jotka muovaavat, rajaavat ja yhdistävät tietovirtoja. Bluetooth-verkon viestit voidaan esittää tietovirtana, jolloin RxJS-kirjaston operaattorit sopivat hyvin verkon käsittelyyn.

Työn mobiilisovelluksen käyttöliittymä on rakennettu Angular-sovelluskehyksellä. Sillä on tiivis integraatio RxJS-kirjaston kanssa, ja se tarjoaa syntaksin tietovirtojen kuunteluun käyttöliittymän komponenteissa. Työssä havaittiin, että uusi arkkitehtuuri mahdollistaa Angularin OnPush-muutostentarkistusstrategian käytön, joka nopeuttaa sovelluksen toimintaa.

Työssä toteutettu arkkitehtuuri aiheutti ongelmia laiteryhmien kanssa. Passiivinen ryhmäkohtainen tiedonkeruu johtaa harhaanjohtavaan käsitykseen ryhmän tilasta, sillä ryhmän jäsenten parametrit voivat vaihdella. Ratkaisuina ehdotettiin ryhmien parametrien kuuntelun puskurointia sekä parametrien lähetystä pienemmissä osissa.

Lähteet

ActiveAhead – itseoppiva valaistuksenohjausratkaisu. 2022. Verkkoaineisto. Helvar. <https://helvar.com/wp-content/uploads/2022/05/Helvar_ActiveAhead_5.3_May2022_FI.pdf>. Luettu 7.2.2023.

API List. Verkkoaineisto. RxJS. <https://rxjs.dev/api?type=function>. Luettu 8.2.2023.

Aijaz, Adnan; Stanoev, Aleksandar; London, Dominic & Marot, Victor. 2021. Demystifying the Performance of Bluetooth Mesh: Experimental Evaluation and Optimization. Wireless Days (WD), s. 1−6.

Bluetooth SIG. Core Specification 5.3. 2021. Verkkoaineisto. Bluetooth. <https://www.bluetooth.com/>. 13.7.2021. Luettu 7.2.2023.

Bluetooth SIG. Mesh Profile 1.0.1. 2019. Verkkoaineisto. Bluetooth. <https://www.bluetooth.com/> 21.1.2019. Luettu 7.2.2023.

Chebbi, Lamis. 2022. Reactive Patterns with RxJS for Angular. E-kirja. Packt Publishing.

Dependency injection in Angular. 2022. Verkkoaineisto. Angular. <https://angular.io/guide/dependency-injection-overview>. Päivitetty 2.8.2022. Luettu 27.2.2023.

Hernández-Solana, Àngela; Pérez-Díaz-De-Cerio, David; García-Lozano, Mario; Bardají, Antonio & Valenzuela, José-Luis. 2020. Bluetooth Mesh Analysis, Issues, and Challenges. IEEE Access, vol. 8, s. 53784−53800.

Introduction. Verkkoaineisto. RxJS. <https://rxjs.dev/guide/overview>. Luettu 8.2.2023.

Introduction to components and templates. 2022. Verkkoaineisto. Angular. <https://angular.io/guide/architecture-components>. Päivitetty 28.2.2022. Luettu 27.2.2023.

Lifecycle hooks. 2022. Verkkoaineisto. Angular. <https://angular.io/guide/lifecycle-hooks>. Päivitetty 28.2.2022. Luettu 27.2.2023.

Nurkiewicz, Tomasz & Christensen, Ben. 2016. Reactive Programming with RxJava. E-kirja. O’Reilly Media.

Observable. Verkkoaineisto. RxJS. <https://rxjs.dev/guide/observable>. Luettu 27.2.2023.

Operators. Verkkoaineisto. RxJS. <https://rxjs.dev/guide/operators>. Luettu 8.2.2023.

Rondón, Raúl; Mahmood, Aamir; Grimaldi, Simone & Gidlund, Mikael. 2020. Understanding the Performance of Bluetooth Mesh: Reliability, Delay, and Scalability Analysis. IEEE Internet of Things Journal, vol. 7, no. 3, s. 2089−2101.

Salvaneschi, Guido; Margara, Alessandro & Tamburrelli, Giordano. 2015. Reactive programming: a walkthrough. Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15). IEEE Press, s. 953–954.

Staltz, Andre. 2018. The introduction to reactive programming that you've been missing. Verkkoaineisto. GitHub Gist. <https://gist.github.com/staltz/868e7e9bc2a7b8c1f754>. Päivitetty 27.11.2018. Luettu 31.1.2023.

Subjects. Verkkoaineisto. RxJS. <https://rxjs.dev/guide/subject>. Luettu 27.2.2023.

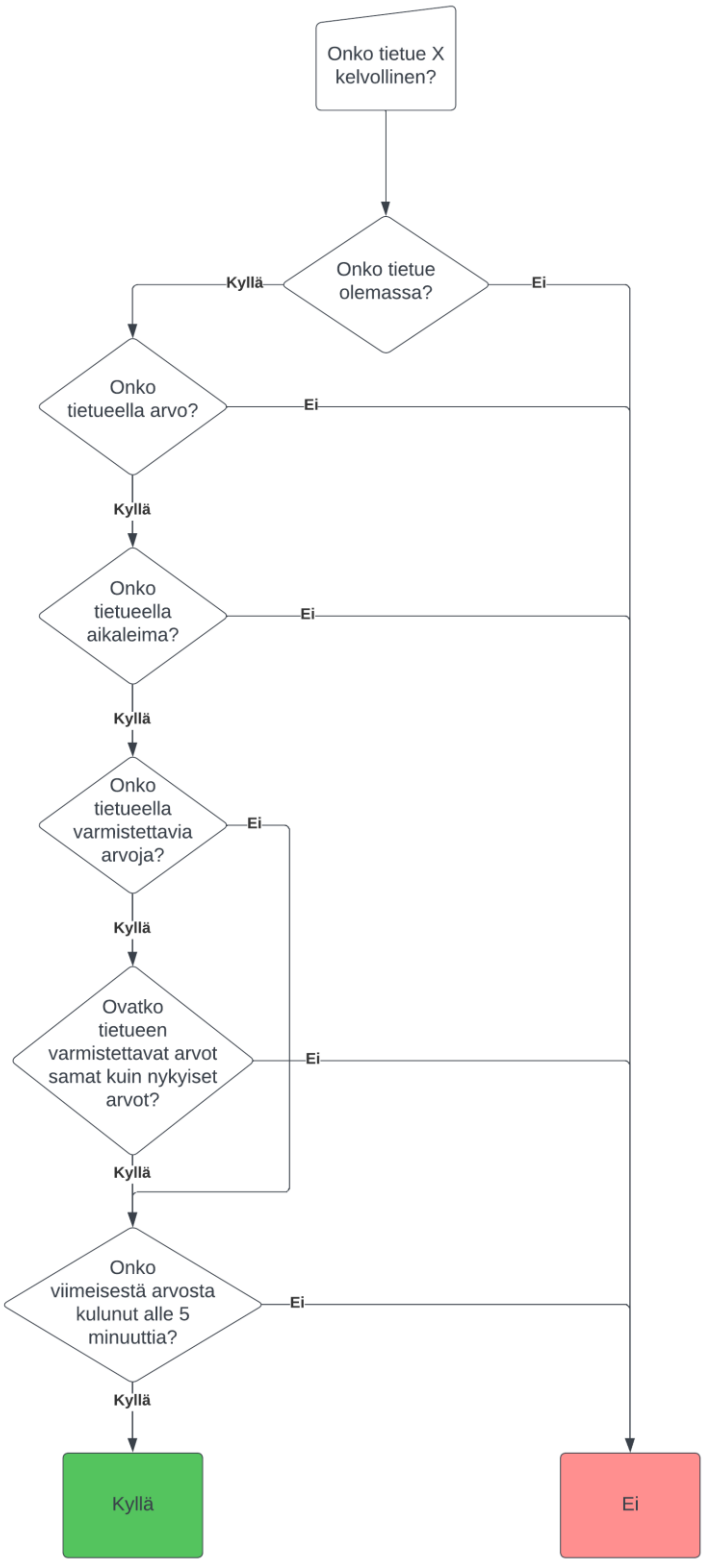
The RxJS Library. 2022. Verkkoaineisto. Angular. <https://angular.io/guide/rx-library>. Päivitetty 28.2.2022. Luettu 31.1.2023.

User Guide: Operation of Helvar ActiveAhead mobile application. 2022. Verkkoaineisto. Helvar. <https://helvar.com/wp-content/uploads/2020/09/app-activeahead_USERGUIDE_EN.pdf>. Luettu 13.3.2023.

What is Angular? 2022. Verkkoaineisto. Angular. <https://angular.io/guide/what-is-angular>. Päivitetty 28.2.2022. Luettu 9.2.2023.

Woolley, Martin. 2020. Bluetooth Mesh Networking – An Introduction for Developers. Verkkoaineisto. Bluetooth. <https://www.bluetooth.com/bluetooth-resources/bluetooth-mesh-networking-an-introduction-for-developers/> Luettu 27.2.2023.

Tietokannan tietueen kelpoisuuden määritys



Odotusajat ennen valotasojen ja ajoitusten muokkausta

	Vanha arkkitehtuuri		Uusi arkkitehtuuri	
	iPhone 11	**Galaxy S22**	**iPhone 11**	**Galaxy S22**
#1	2010ms	1984ms	391ms	369ms
#2	2139ms	1984ms	290ms	267ms
#3	2070ms	2042ms	270ms	216ms
#4	Aikakatkaisu (8000ms)	2081ms	343ms	527ms
#5	Aikakatkaisu (8000ms)	Aikakatkaisu (8000ms)	237ms	333ms
#6	Aikakatkaisu (8000ms)	1904ms	287ms	243ms
#7	2051ms	1891ms	204ms	242ms
#8	Aikakatkaisu (8000ms)	1978ms	274ms	270ms
#9	1973ms	Aikakatkaisu (8000ms)	194ms	195ms
#10	1975ms	1960ms	327ms	260ms
#11	Aikakatkaisu (8000ms)	1963ms	210ms	215ms
#12	1969ms	1962ms	910ms	729ms
#13	Aikakatkaisu (8000ms)	1986ms	190ms	245ms
#14	1987ms	Aikakatkaisu (8000ms)	97ms	242ms
#15	2032ms	1962ms	914ms	1136ms
#16	2131ms	1980ms	220ms	254ms
#17	2025ms	2049ms	568ms	416ms
#18	1997ms	1961ms	244ms	333ms
#19	Aikakatkaisu (8000ms)	Aikakatkaisu (8000ms)	207ms	186ms
#20	Aikakatkaisu (8000ms)	2017ms	168ms	394ms
#21	2101ms	Aikakatkaisu (8000ms)	133ms	185ms
#22	Aikakatkaisu (8000ms)	1969ms	183ms	159ms
#23	2074ms	1888ms	230ms	289ms
#24	2138ms	Aikakatkaisu (8000ms)	293ms	218ms
#25	2173ms	Aikakatkaisu (8000ms)	307ms	244ms
#26	2078ms	Aikakatkaisu (8000ms)	95ms	547ms
#27	2044ms	2110ms	237ms	271ms
#28	2173ms	2003ms	245ms	174ms
#29	1994ms	Aikakatkaisu (8000ms)	429ms	323ms
#30	2172ms	1925ms	202ms	832ms
Keskiarvo	**3844ms**	**3787ms**	**297ms**	**344ms**
Aikakatkaisut	**9**	**9**	**0**	**0**

Puhelinten vastaanottamat arvot

	1		2		3		4		5		6	
#1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#5	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#6	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#7	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#8	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#9	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#10	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#11	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#12	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#13	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#14	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓
#15	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#16	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#17	✓	✓	✓	✓		✓		✓		✓		✓
#18	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#19	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
#20	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#21	✓	✓	✓	✓		✓		✓		✓		✓
#22	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
#23	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#24	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#25	✓	✓	✓	✓	✓		✓		✓	✓	✓	
#26	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#27	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#28	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
#29	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#30	✓	✓	✓		✓		✓		✓		✓	
#31	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#32	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#33	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
#34	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓
#35	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#36	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#37	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#38	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#39	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#40	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#41	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#42	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#43	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
#44	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#45	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#46	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#47	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#48	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#49	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#50	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
#51	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	
#52	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
#53	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#54	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#55	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#56	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
#57	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓
#58	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#59	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

#60	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓
#61	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	
#62	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#63	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#64	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
#65	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#66	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#67	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#68	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#69	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#70	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
#71	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#72	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#73	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#74	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#75	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#76	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#77	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#78	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#79	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
#80	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#81	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#82	✓	✓	✓		✓		✓				✓	
#83	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#84	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#85	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#86	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#87	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#88	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#89	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#90	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	
#91	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#92	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#93	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#94	✓	✓	✓		✓		✓		✓		✓	
#95	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#96	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	
#97	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#98	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
#99	✓	✓	✓	✓		✓	✓	✓		✓		✓
#100	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Summa	**100**	**100**	**100**	**97**	**97**	**95**	**97**	**94**	**94**	**95**	**89**	**88**
Osuus	100,00 %		98,50 %		96,00 %		95,50 %		94,50 %		88,50 %	