TNO report

Final report of the OnLive ClickNL Project: The social network of the real world

Date	31 Januari 2019
Author(s)	Wilco Wijbrandi, Eelco Cramer, Gjalt Loots, Robbert Schep (RSNMC)

Contents

“The complexity and urgency of the problems faced by us earth-bound humans are increasing much faster than our combined capabilities for understanding and coping with them. This is a very serious problem. Luckily there are strategic actions we can take, collectively.”
- Douglas Engelbart

Executive summary

It's just about 12 years ago that Apple and Google launched the smartphone into our world. They successfully designed a software system for combining the power of the internet with the power of the cell phone. Never in history a technology became so fast, so deeply rooted in society. Now we are paying (and playing) with our phones as if we never did it differently.

It's safe to say the smartphone changed our society in a lot of ways. In a lot of ways it made our lives more convenient and more entertaining (never a dull moment anymore!) It's also safe to say that the smartphone changed our social lives and has an influence on our mental and physical health. There are not many aspects in our lives that aren't touched by that one device. A device that we bring everywhere to be connected with everything, all the time.

As researchers, developers, engineers and designers, it's our job to be critical about the role of a device that has so much impact on different levels in our society. Mostly because we can not just blindly follow the trajectories of a few giant commercial tech corporations. Their motives are clear. The ideological goal of the tech giants probably are something in the line with 'having your digital life in your pocket' and 'always have access to all the information and functionality you need'. The actual goal, of course, was to make as much money as possible no matter what costs. There was a new market to conquer and of course, they foresaw that those who dominate this new market will be the leaders of the brand new information society.

The systems they came up with, IOS and Android, are basically the same systems in a different skin and some minor usability differences. Most importantly they both are based on the premises that users need to install applications made by third parties. These applications are mostly free and generate a lot of data about the one who is using them. Data which is sold to advertisers.

Since data harvesting and selling is the main business model, most applications are designed to be used as much as possible. Most software developers and user experience designers are constantly busy with one thing: to make users use their application as much as possible. They are busy capitalizing on the data from the users.

Vision

We believe there is a much higher potential to be capitalized on. We are not talking about commercial capitalization. We are not interested in making more money with the smartphone. We are talking about the capitalization of human intellect. We believe the smartphone can become a tool for intellectual progress for everyone on this planet. We believe the smartphone is an enabling and supportive technology. If used correctly, it can enable societies to make the transition they all need to make. It's important to understand that Onlive has one clear goal: to turn the smartphone into a tool for intellectual progress and problem-solving.

Mission

We want to build a platform that supports people in their daily lives when they need to interact with their varying surroundings. We want to optimize but minimize those interactions because we want to reduce redundant screen time and increase quality group collaboration processes.

We are not in the business of making money. Money is just a tool to get us there. We are in the business of contributing to the global issues that face us all. Our specific mission: to pivot the smartphone into a tool for intellectual progress so it can help communities all over the world to make better decisions and solutions. In today's world, one thing is clear: to leave a liveable planet to next generations, we need all the talent in the world working with relevant information and functionality on real solutions. Right now the smartphone is not enabling us to do that at all. It distracts us and makes us operate on a more individual level. Onlive wants to change that. This document is a blueprint for the foundation of new platform that can do exactly that.

In order to comprehend exactly what is proposed it’s advised to have an open and progressive mind that does not accept current reality and looks to alter it for the better of the planet and all nature that is living on it.

1 Introduction

Onlive is designed with the scientifically accepted idea in mind that humans are, by nature, social animals that always have relied on dynamic communities and groups for survival. In groups, humans are more effective, creative and better at survival[1]. With this evolutionary awareness in mind it's astounding to realize that currently almost all computer systems (and most certainly the most used ones) are designed to provide information and functionality to the *individual*.

Onlive is a project that does the opposite and appeals to the social, community driven species we are. It's a new information system that is designed to provide information and functionality to *groups of individuals* that need to work or act together in the physical world. This simple idea complies with the work of computer pioneer Douglas Engelbart (1925-2013). His research on Collective IQ and Dynamic Knowledge Repositories are used to explain the choices we made in designing the Onlive MVP.

"Whether developing a new product or service, researching a topic of interest, seeking a cure for cancer, or improving conditions in underserved communities, a group's Collective IQ is a key determinant of how effectively it will respond to the challenges presented. More specifically, regardless of the end goals, it comes down to how quickly and intelligently the group can identify needs and opportunities, develop and deploy solutions, and incorporate lessons learned, while continuously iterating and adapting to changing conditions until the goals are met" - (Douglas Engelbart Institute, about Collective IQ)[2].

The smartphone is one of those technologies where you clearly see the individualistic design. Everything about the smartphone is aimed at the individual. The main business model strategy is to retrieve as much as data possible from one person. It allows people to get information from and connect with people all over the world. Although this has many advantages, it also has a downside: it distracts people from their surroundings. Onlive is a concept that tries to change that: instead of connecting one person with the rest of the world, it connects groups of people with their local world, facilitating efficient interaction with things and people around them, instead of distracting them from exactly that.

[1] https://www.nature.com/articles/nature10601
[2] http://www.dougengelbart.org/content/view/172/130/

Onlive can be summarized in four core functionalities:

- Connect people with other people around them
- Connect people with companies/organisations around them
- Connect people with devices around them
- Facilitate interaction for verbal social interaction among groups of people

In the Onlive ClickNL project, research organization TNO collaborated with the RSNMC, Johan Cruijff Arena, FONK, Stampwallet, Groovidi, the Office Operators and Martin and Lewis to work out the Onlive concept and develop a proof-of-concept.

RSNMC organized workshops with all project partners to determine in what way Onlive could add value for their business. For all partners a design was made for their use case, which resulted in requirements for Onlive itself.

This document is the project report, in which the technical research and development of Onlive is described. This research and development was done in cooperation with all project partners, but was mainly executed by TNO and RSNMC. Also, Fair2Media was hired as a subcontractor, which assisted with working out the user experience and the user interface for the Onlive app.

2 Deliverables and research questions

This document is the main report of the Onlive project. However, there are additional results of the project made available.

- The Onlive Plugin *Software Development Kit* (SDK), consisting of:
    - Executable (App) for Android (in the form of an .apk file)
    - Developer documentation
    - Example Plugin
- The Onlive App
    - Executable (App) for Android (in the form of an .apk file)
    - Onlive App - Functionality and priority (document)
- The Onlive backend
    - Custom Onlive logic for integration with the XMPP server

- The Onlive registry, which hosts
  - Context group definitions
  - Plug-in executables (in the form of a .zip file)

In the project proposal 5 research questions (RQ) and 6 deliverables (D) were proposed. Since some of the resource questions and deliverables are addressed in one of the other project results. The following table indicates where research questions or deliverables are addressed.

Item	Where addressed
RQ1. How to enable the dynamic emergence of groups?	Section 5.3.2
RQ2. How to dynamically deploy plug-ins based on context information?	Section 5.2
RQ3. What kind of functionality should the platform offer plug-ins and how can functionality be extended in the future?	Section 5.3.3
RQ4. How can groups be created and data from plug-ins be exchanged without relying on an internet connection?	Section 5.4
RQ5. How can the Onlive backend be designed in a robust and scalable way?	Section 5.2
D1. Context engine description	Section 5.3.2
D2. Plugin run-time description	Onlive Plugin SDK documentation
D3. Feasibility report of direct or hybrid communication for Onlive	Section 5.4
D4. Platform architecture document	Chapter 5
D5. Onlive App	Onlive App
D6. Onilve development guide	Onlive Plugin SDK documentation

Many partners developed concepts or implementations for their integrations with Onlive. They are not covered by this report since they are the property of the respective project partners.

3 The Onlive concept

Onlive is a platform technology that allows end users to interact with people and things in their physical surroundings. From the point of view of the End user, Onlive is an App they install on their smartphone.

When people use the Onlive app, they automatically become part of Groups. When you are in a venue, the user can automatically become part of the group in that venue, together with staff and other guests. When the user is in a meeting room, it becomes automatically part of a group which consists of all other people in the room. A group can be small or big; in theory, there could also be a group for the entire city.

Onlive facilitates interaction among people and devices, but it does not *define* what this interaction should be; this is determined by Plugins. Plugins can be seen as mini-Apps that are available in a group and that are automatically downloaded on the smartphone when user becomes part of that group. Plugins can be developed by anyone. There could be generic plugins which facilitate common interactions, but there can also be branded plugins for specific locations or specific events. Plugins allow End users to interact with their surroundings (interaction with other people, with organisations or with devices).

“Interactions” is a relatively abstract term. An interaction could be sending someone a link to a website, it can be file, it can be contact information, it can be vote on a poll, it can be an answer to a quiz, but it can also be something much more dynamic, such as controlling a virtual character in a game. Each of these example interactions will require a different plugin. Sometimes people would like to save those interactions when they make use of a plugin, sometimes it makes no sense to save them. In order to provide a structure which allows people to save interactions for which that make sense, we introduced the concept of a *Card*. A card can be seen as a file on a computer. It can be stored and it can be transmitted. But a file on its own is usually not very useful. You need a program in order to open the file. In Onlive, this program is the Plugin.

In the following sections, the concepts Group, Plugin and Card are described in more detail.

3.1 Groups

When the user opens the Onlive App, a list of currently available groups is shown. Available groups are determined automatically; the user doesn’t have to find a group manually. There are two types of groups in Onlive, which have slightly different mechanics.

Nearby group
The Nearby group is always available and consists of End users who are physically near to each other or in the same room. There is only one Nearby group. When there is a meeting going on in a meeting room, all users should be part of the same group, and no one outside of the meeting room can be part of the group. Onlive makes use of Google Nearby, a library that makes use of ultrasound signals in order to determine which phones are near to each other. Ultrasound cannot be heard by humans, but cannot travel through walls (unlike radio signals like Bluetooth and WIFI). This way, groups are formed in a way that is likely to make sense to users in the real world; people in the same room are in a group together. The downside of Google Nearby is that it requires an Internet connection to operate.
Users are always part of the nearby group. When no other Onlive users are nearby, the group will consist out of the end user him/herself.

Context group
In contrast to the Nearby group, there can be many Context groups. Context groups are defined for people who share the same context; this can for example be physical location (a certain venue) or an event (a location in combination with a certain period). Context groups are defined using Context rules, which are evaluated by the Context engine, which was originally developed in the Transient Apps project.

Context groups are defined by the Onlive backend. For example, a store might want to define a group for people who are physically present in the store. In contrast to the Nearby group, which is always available, Context groups need to be created manually by someone in order to become available to users.

The advantages of creating a Context groups over the Nearby group are:
- group size is not limited by ultrasound (so it’s usable in bigger areas)
- complete control over the available plugins (so clients can determine what there clients/end users can do when they are in their group

More information on how Context groups are defined using Context rules can be found in Appendix B.

3.2 Plugins

Plugins are the mini-Apps which allow people to interact with each other in a specific way in a group. For example, there could be a Plugin which allows you to send pictures to other people in the group. For each group, there is a list of which Plugins are available to the End users in that Group. For the Nearby group Onlive determines which Plugins are available, for Context groups this is defined by the group host (the one who created the group). .

Plugins are mini-Apps that are automatically downloaded installed whenever people become part of a Group, or are removed when they leave the Group. Plugins are executables. Users don’t have to worry about Plugins, and they don’t directly interact with them. Rather they interact through them.

3.3 Cards

A Card is how users experience a Plugin. A card represents a piece of data, which can be handled by a Plugin.

Whenever a user opens a Group which is available in the Onlive App, it enters the *Stage*. In the Stage, all the cards that are currently published in that Group are shown. A card could for example represent a business card. When the user clicks on the card, the card is *opened*. The Plugin is launched, displaying the information that was stored in the Card. How the information inside the card is represented, is determined by the Plugin. When a group allows multiple Plugins to be used, there are also multiple types of Card that can be present in the Stage. Each Plugin is associated with one type of Card.

Users can decide to save a card from the stage (if the card is a savable card). When they do that, the card is saved locally on smartphone. Saved cards can also be found and opened later under *saved cards*. It is also possible for users to publish saved cards to groups it is part of. So, for example, it is possible to save a card from the stage of Group A, and then later publish it in Group B, given than that the user is part of both groups and both groups allow the usage of the Plugin the card is associated with.

Users can also use a Plugin to create a new Card. When they decide to create a new Card of a certain type, the associated Plugin is launched and a wizard is shown to create the new card. How a new Card should be created is determined by the Plugin.

Privacy policies

Before diving deeper in the types of cards its important to shine some light on the very important privacy choices we made. For privacy related questions about how we deal with the users identity in a group and it’s interaction with cards and plugins, it’s important to understand we built Onlive to support people in their daily lives when they need to interact with their varying surroundings. We want to optimize but minimize those interactions because we want to reduce redundant screen time and increase quality eye talk. The goal is to cooperate and communicate with those in our immediate world, in such a way that we become better in fixing our world. If you want to build a digital system for the physical world you have to apply certain rules of the physical world when you design the system. One of the key rules in social group behaviour is that the more trust there is, the more effective we communicate.

"When trust is present, people step forward and do their best work, together, efficiently. They align around a common purpose, take risks, think out of the box, have each other's backs, and communicate openly and honestly. When trust is absent, people jockey for position, hoard information, play it safe, and talk about— rather than to—one another". - Dennis Reina, PhD Michelle Reina, PhD David Hudnut, MIA, Why Trust Is Critical to Team Success, 2017

Onlive has to become and stay a tool that people can trust like a strong spade that everyday helps people to work their ground. Something to rely on. Something you can just use and you don't have to worry about any scam or failure. To establish trust in groups we've mimicked our natural group conventions as close as possible when we made some key UX decisions. The most direct evidence of such a choice is that we follow the natural law of matter that dictates that you can't be invisible when you are participating in a group. Therefore with Onlive you can't be anonymous when you participate in a group.

This sensitive and complex matter seems to be an issue that can be handled in a later stage when people are actually using the platform but in present days it has never been more important that you have to be clear and transparent about your privacy policies from the beginning. Onlive wants to do things different than traditional social media and privacy is one of those areas where we can establish that difference.

To address this issue head on we introduce the Onlive passport. The premise of this idea is, again, mimicked from our physical world. When you travel to another country you bring your passport. When you want to use Onlive you have to have an Onlive passport.

The Onlive Passport is made when you register as a user for the first time. It's basically a quick registration process we are familiar with when we use other online platforms.

- Email (Mandatory and to be verified)
- Phone number (Mandatory and to be verified)
- Name (Mandatory and not verifiable)
- Username (Mandatory and not verifiable)
- Photo (Not mandatory and not verifiable

After this registration process Onlive will assign a unique number in the form of an UUID. This UUID is connected to your personal information and together they form your Onlive passport that allows you to enter and participate in groups.

Regarding privacy issues we have to address two specific situations:

1. What happens when you enter a group but don't interact with any of the available cards.
2. What happens when you enter a group and interact with one or more cards.

We separate these two situations because Onlive basically just facilitates the existence of groups. What happens in the group is mostly determined by the group host and the available plugins. In general you could say that Onlive only keeps track of the groups being formed and the users that ENTER the group but does not keep track of the INTERACTIONS that happen inside the group. Interactions with cards are being tracked by card owners. Interactions with plugins are being tracked by plugin owners.

1. *What happens when you enter a group but don't interact with any of the available cards.*

What happens when you enter a Nearby group is that:

A. your UUID will be used as a sign-in to the group and will display your Username and Photo in the attendees list for other people to see.
    i. notice that you your Username does not have to be your real name.
B. your UUID will be registered in the Onlive backend, connected to the time and place of the groups existence. The Onlive backend only saves the UUID for 24 hours and than deletes all registrations. The Onlive backend will ONLY save the UUID and not the personal information that belong to that UUID.
    i. This information will never be given to third parties unless that information can help verified government organisations solving criminal activities.

What happens when you enter a Context group depends on the mandatory privacy decisions the group host made when creating the group. We imagine that there will be two options group host can choose from.

A. Standard: the same privacy rules as for the nearby group
B. Custom: the sign-in process requires extra permissions of users depended on the Identity plugin the group host choices .

For example: a group host creates a group for his networking event. He wants his visitors to share more information about themself than just a username and a picture. In order to do that he has to install an Identity Plugin for his group that complies with his identity demands. In this case he chooses the business card plugin that demands from users to sign in with their business card. When the user enters the group that plugin will ask the user to publish his business card to the stage. If the user does not have a business card he can create one very quickly with the business card plugin.

This way we can foresee different kind of identity plugins that generate different kind of sign-in cards. This is a smart and controllable way of creating transparent situation regarding the privacy of users. They are either signing in with their Onive passport or are asked to hand in a different form of identification that is chosen by the groups host.

2. *What happens when you enter a group and interact with one or more cards.*

The general rule is that your Onlive passport will always be attached to your interactions with cards. It does depends if a card is marked by the owner as savable, collectable, reusable or re-publishable what exactly happens but in general this is what happens:

- If you create a new card, your Onlive passport is registered als ‘Owner’ of the card (there can only be one owner of a card)
- If you save a card of somebody else, your Onlive passport is registered as ‘collector’ of the card and the owner of card will be able to see who this collector is.
- If you publish a card from somebody else, your Onlive passport is registered as a ‘publisher’ of the card and the owner of the card will be able to see which collector has published his card in what group.
- If you reuse a card from somebody else to make your own card (with the same plugin) your Onlive passport is registered as ‘Owner’ of the card.

The idea is that a card always has a back side (you can flip a card on it’s back) and there you can see where the card originates from, how many times it has been collected, republished and of course, who is the owner.

3.3.1 Archetypes cards

In order to get a better understanding of what the functionality of a card can be, four archetypes of cards were defined. It are examples which demonstrate different forms of interactions possible with cards.

1. Identity Cards

Cards that contain a form of identity and can be ‘handed in’ when people enter a context group. Examples:

- ○ Business card
- ○ Facebook card
- ○ Instagram card

2. Document Cards

Cards that contain any form of document that can be viewable and/or editable. Examples:

- Word card
- Adobe card
- Powerpoint card

3. Questionnaire Cards

Cards that contain any form of questionnaire. These type of cards demand interaction from group members and generate live statistics based on those interactions, such as poll results. Examples:

- Poll Card
- Quiz Card

4. Presentation

Cards that contain any form of live stream. These type of cards will show what is on the screen of the card publisher. This is handy when you want to show your content plenary to the people around you. Examples:

- Powerpoint stream card
- Photo gallery stream card
- Video stream card

5. Objects

Cards that contain information about an object. These type of cards enable users to interact with an object that is in their surroundings. Examples:

- Light Card (to control the light in a room)
- Printer Card (to enable users to use a nearby printer)
- Screen Card (to enable users to use a screen nearby)

Some example functionalities that can happen within Onlive and the role of a plugin and cards.

Poll

A poll can be prepared by someone (with a plugin) and then published to a group. The publisher is the admin of the poll. He can see intermediate results and start and stop the poll. Other users can cast their vote and see the results of the poll.

- A poll card has a shared state when it is on the group stage
- There are different views for publisher and other users
- When a poll is taken from the stage the taker gets a card with the (intermediate) results.
- A new poll can be created from a saved poll (as a template).

Dim lights

A dim light card is a group card that allows users in the group to dim the light of a room. The card is not saveable nor can it be in the hand of a user. The card only exist in the context of a group.

- Interact with physical objects
- Can be made exclusively usable for specific group members

Business card

A business card is a card that contains static information of a user. When the card is published by the user he can add notes to the card. When a card is taken from the group stage the taker can also add notes to the card.

- Static information
- Allows for personal notes on a card
- When being published the plugin asks the user if the notes should be published as well (publishing involves the plugin).

Snake (game)
Snake is a real-time game where multiple snakes are on the screen and should avoid to collide with each other. When creating a new Snake card it can only be published to group and not saved locally, as there is no point as it is multi player game. When taking a card from the stage the saved card contains the current scores of the game.
The publisher can start a new game.

- Real time messages between card instances
- When not playing the game messages can be ignored

3.3.2 Card states

Although a Card is always opened by the same Plugin, it can have one of multiple states. Which states a card can have is determined by the Plugin.

For example, a Plugin which is used for casting polls, can have a state in which the designer can still change the option, it can have state in which other users can vote, and it can have a state in which the results of the poll can be shown.

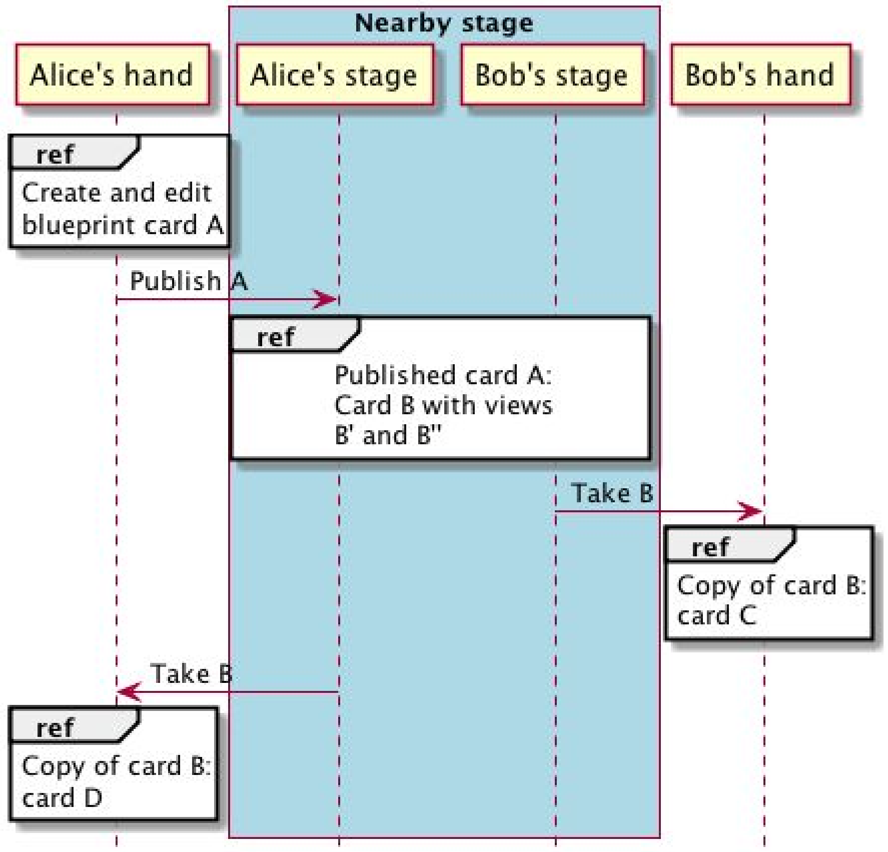
Cards can change state whenever they are published or saved. If and how this happens, is determined by the Plugin.

Technical aspects of card states are discussed in more detail in the SDK documentation.

3.3.3 Card distribution

When a Card is stored on the smartphone, the user can decide to publish the card to the stage of a group. And when the card is published in the stage of a group, a user can decide that the card can be saved on the smartphone. Whenever a card is saved or stored, the original card does not get removed, but rather a new card is created, either in the stage or locally on the smartphone. This new card can have a different state then the original card. If and how a card changes states whenever it is published or saved, is defined by the Plugin.

Consider a Plugin for casting polls in a group. This card is initially created locally, in which it has a state called ‘blueprint’. When the card is in this state, the user can use the card to edit the poll. It can, for example, change the question or modify the options for which can be voted. When the card is published to the stage of a group, the card changes to the state ‘live poll’. In this state, users can vote for one of the options. Although everyone is looking at the same instance of the card in the state, the card presents itself differently to the user that originally published the card to the stage. That user might see the current results of the poll, while other users only see the option to vote. When a user saves the card to the smartphone, the card changes to the state ‘poll results’. This purpose of this card is show the results of the poll. In this state it is no longer possible for users to vote.



Defining states and state transitions is a powerful concept for Plugin developers to define the behaviour of a card while it is being used by users. Technical aspects of publishing and saving cards is discussed in more detail in the SDK documentation.

3.3.4 Card lifecycle rules

Users should feel safe when publishing cards to the stage of a group. It should be transparent to users who is going to be able to see the cards they publish. Groups should only contain a manageable amount of cards.

In order to address these concerns, the right mechanics for when cards are visible to other users, and when not, needed to be defined. This was done in the form of rules. Since the

Context groups and the Nearby groups have different mechanics, the rules are also slightly different.

In a Nearby group a card belongs to the publisher, in a context group a card belongs to the group;

- A card is received only by the current members of the group. This means that not necessarily everyone in the group has the same cards on the stage;
- A card can always be removed from the stage by its publisher;
- Cards are automatically removed from the stage if a time limit is reached (e.g. 1 day)
- Nice to have: cards can be republished or made sticky by the publisher in order to make them available to new group members

Context groups

- When a user returns to a context group, cards that were received in a previous session are still available in his stage;
- When a user returns to a context group, cards that were removed in his absence will also be removed from his stage;

Nearby group

- When a publisher leaves the user's nearby group, that publisher's cards are removed from the user's stage;
- When a publisher returns to the user's nearby group, that publisher's cards are again available in the user's stage;
- When a publisher removes a card in his nearby group, that card will not be available anymore in the user's stage when that publisher returns to the user's nearby group.

3.4 Use cases in the app

For the user experience a comprehensive analysis was made of the user stories and use cases the app should provide. All use cases were prioritized using the MoSCoW principle (determining for every use case if this Must, Should, Could, or Won't be implemented). All uses cases with the Must priority have been implemented in the Proof-of-concept.

The analysis can be found in the 'Onlive App - Functionality and priority' document (which is partly in Dutch).

4 Onlive ecosystem

In order to get a better understanding of what the business side of the Onlive platform would look like, an analysis was done of the stakeholders and the business models.

4.1 Stakeholders

The following (types) of stakeholders were identified in the Onlive ecosystem.

1. **Onlive:** the owner of the Onlive platform

2. **End user:** the end user that uses the Onlive App on his or her smartphone

3. **Group owner:** the person or party that registered a Context Group in the Onlive backend. The Group owner determines which Plugins are available within the group.

4. **Plugin developer:** the person or company developing Onlive Plugins

5. **Plugin owner:** the person or party which owns a Plugin. Onlive will be the Plugin owner of some generic Plugins.

4.2 Business model

The business model of the Onlive platform is that of a two-sided platform: on one hand the platform becomes interesting for End users when there are a lot of groups and plugins available. On the other side, the platform is interesting for Group owners (to create groups and Plugins) when there are many End users on the platform.

This forms a challenge for bootstrapping the platform. In order to get End users on the platform there needs to be Group owners, and vice versa. The Nearby group was designed to address this problem. Since the Nearby group doesn’t require any Group owners or infrastructure, the Nearby group always adds value to End users. This way the platform can attract End users without Group owners present on the platform. So initially, Onlive can start off as an app that makes it easier to interact with people nearby, and when a critical mass has been established Group owners can be attracted. Another approach here is to start locally with Group owners. When Onlive is useful on a certain location or event, the user experience can be increased for a select group.

Three potential business models have been identified for Onlive that could also work together:

- Changing Group owners for creating groups: certain venues or events might want to provide a better customer experience using Onlive. They can develop for example custom Plugins which allows them to serve their customers better. In order to be able to do this, a Context group needs to be defined. Onlive could change companies to do this.
- Facilitating a marketplace between Plugin developers and Group owners: Certain Plugins will add value for a certain type of business. For example, meeting rooms in all kinds of companies can benefit from Plugins that help facilitate meetings. Not all meeting venues would be willing to develop their own Plugins. It would make sense to create a marketplace, where Plugin developers sell Plugins to Group owners that are interested in providing these plugins to their customers. Onlive could take a part of the revenue of the Plugin developers.

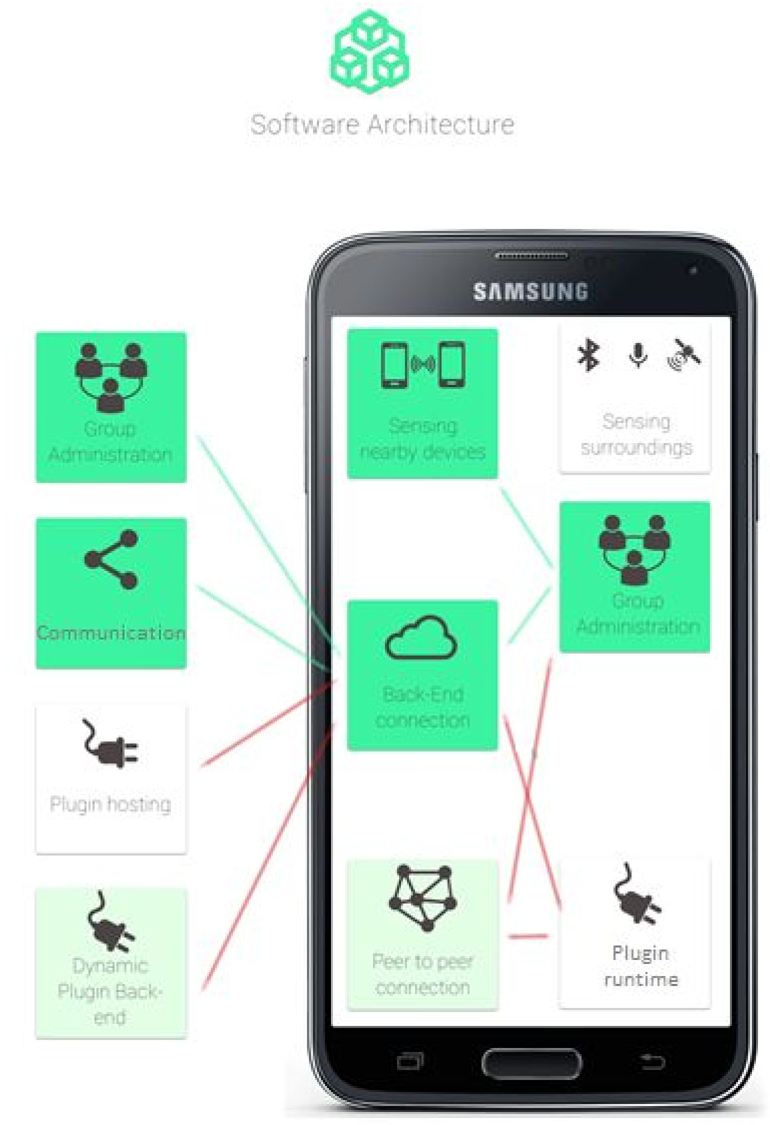
5 System architecture

This section describes the technical implementation of the proof of concept and the final version of the Onlive platform.

5.1 Overview

The system architecture consists out of two major parts: The backend, which are the centralized services which are reachable through the Internet, and the frontend, the application used by the End users.

Within the Onlive project, the frontend has been developed in the form of an Android app. Eventually, it would make sense to also developed it as an iOS app and possible in the form of a web application.



The backend consist of the following major parts:

- Group administration: The registry where all Context groups are define
- Communication: Component used for transferring information between frontends
- Plugin hosting: The registry where all plugins are stored. The frontend downloads these plugins automatically when needed.
- Plugin backends: Although not part of the platform, some Plugins might require their own backend for their own administration.

The frontend consists of the following major parts:

- Sensing nearby devices: Component necessary for determining which users should be in the Nearby group
- Sensing surroundings: Component necessary for determining in which Context groups the users should be (context engine)
- Group administration: Logic which determines in which groups the users is
- Back-end connection: Component for exchanging information with the backend
- Plugin runtime: Component executing the Plugins
- Peer-to-peer connections: Communication module for when no Internet connection is available (not implemented in the proof-of-concept)

5.2 Backend

The Onlive backend consists out of two technical components:

- Onlive communication (e.g. joining and leaving groups, retrieving and publishing card data, communication regarding cards)
- The context group definitions and Plugin hosting

Both parts have been implemented in the Onlive project. For both parts, research has been done on how this can be implemented in a scalable way.

5.2.1 Communication through XMPP

For the Onlive communication the decision was made to use of the XMPP protocol. Originally designed for chat application, XMPP has now evolved as a mature generic messaging system, which supports many use cases. The advantage of XMPP is that there are many server implementation available, both with commercial licenses as with and open source licenses. These existing server implementations differ in functionality, but usually scalability and security are already taken care of, making it an ideal starting point for the Onlive communication backend.

There is already a lot of functionality available in XMPP. For most of the functionality required for Onlive, a mapping could be made to existing XMPP functionality. This way development effort could be reduced. However, the functionality required for the administration of the Nearby group was not present in XMPP, since this concept is quite unique in its nature. In order to solve that, a custom XMPP plugin was developed. This component exclusively handles the administration of the Nearby groups, which means that the heavy lifting is done by the XMPP server. Although the plugin only runs on one server, it is unlikely that it needs to be scaled among multiple server. Although the implementation did not take it into consideration, it is possible to create a distributed version of this component, which makes it possible for the concept to scale to many users. The XMPP plugin was developed in the Go programming language.

A detailed description of the mapping between Onlive functionality and XMPP functionality can be found in Appendix A.

5.2.2 Context group definitions and plugin hosting

In the backend there is a registry, in which all Context groups are defined. For defining a Context group, the following information is required:

Field	**Type**	**Description**
Id	string	Unique identifier of the App Fragment in the java package naming convention
version	string (int . int)	Version of the Group (must be updated for every change in either of the three files)
name	string	Human-readable title of the Group
description	string	Human-readable description of the Group
predicate	object	Context-description as described in Appendix B
plugins	string array	Plugins that can be used within this group. Plugins are identified by the URL they can be obtained from.

In the current implementation, Context groups are defined in a text-file in JSON format. There is a server, implemented in Java, which makes the Context groups definitions available through a REST API. With this REST API, frontend software can synchronize the list of Context groups.

In order to allow the system to scale, a scheme was developed to create separate registries for different geological locations. This way, the frontend does not have to keep track of all Context group definitions in the world. Since this posed no issues for the proof-of-concept, this scheme was not implemented in the project.

An example of the group definition is shown below.

```
{
  "id": "nl.arenapoort",
  "version": "1.1",
  "name": "ArenAPoort",
    "description":  "ArenAPoort  is  een  uniek
gebied vol entertainment, winkels en horeca",
  "predicate": {
    "geographyDatas": [

{"latitude":52.316455,"longitude":4.941745,"alt
itude":0.0,"accuracy":1.0},

{"latitude":52.314277,"longitude":4.935179,"alt
itude":0.0,"accuracy":1.0},

{"latitude":52.310499,"longitude":4.93784,"alti
tude":0.0,"accuracy":1.0},

{"latitude":52.309659,"longitude":4.943676,"alt
itude":0.0,"accuracy":1.0},

{"latitude":52.310945,"longitude":4.947281,"alt
itude":0.0,"accuracy":1.0},

{"latitude":52.310814,"longitude":4.951358,"alt
itude":0.0,"accuracy":1.0},

{"latitude":52.313516,"longitude":4.960113,"alt
itude":0.0,"accuracy":1.0},

{"latitude":52.317451,"longitude":4.956293,"alt
itude":0.0,"accuracy":1.0},

{"latitude":52.315064,"longitude":4.947152,"alt
itude":0.0,"accuracy":1.0}
    ],
    "type":"geography.polygon"
  },
  "plugins": [
    "http://example.com/chat-v1.0.0.zip",
    "http://example.com/helloworld-1.0.0.zip"
  ]
}
```

For hosting the Plugins, a normal HTTP(S) server was used. There are many commercial and open source implementations available for HTTP(S) servers, and scaling them is a common practice.

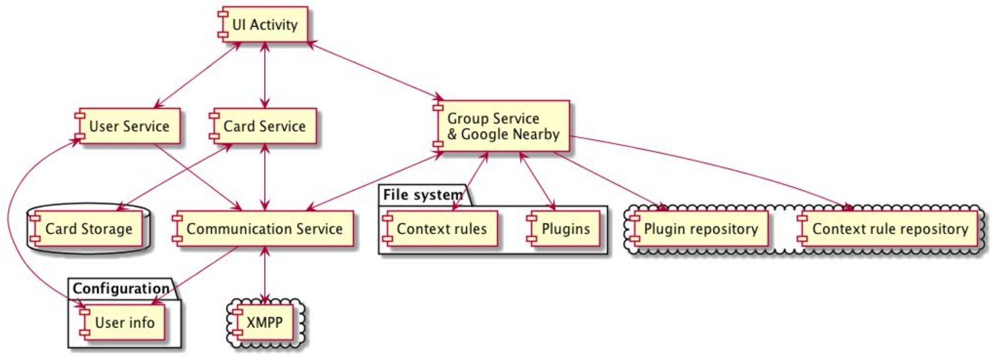
5.3 Frontend: The Onlive App

For this project, the frontend was implemented as an Android app.

5.3.1 Architecture

The Android app consist out of one Activity, which displays the user interface. The user interface consists out of several fragments, which can be switched dynamically. The architecture consist out of several background services, which provided services to each other and to the user interface.

The image below shows a high-level picture of the architecture. For the most important services, their responsibilities have been described.



Group Service
GroupService exposes functions towards UI Activty.

- listen for group availability
- listen for group membership changes
- get list of available groups
- get list of members for a group
- get list of plugins for a group

Other responsibilities:

- listen for other nearby users using Google Nearby and reporting it to Communication Service

Card Service
CardService exposes functions towards UI Activity.

- listen for card messages
- listen for new cards that were published
- publish a card
- publish a message to a card
- get cards for a group
- get all saved cards
- get messages for a card

User Service
UserService exposes functions towards UI Activity.

- retrieve information current user
- retrieve information off other users (async)
- update user details

Other responsibilities:

Create a new user first time app launches

Communication Service
CommunicationService exposes functions towards GroupService

- listen for group membership changes (new members joining or leaving a group)
- join a context group
- leave a context group
- publish a nearby observation

CommunicationService exposes functions towards CardService

- listen for published cards
- listen for card messages
- publish a card
- publish a card message

CommunicationService exposes functions towards UserService

- get user details (async)
- update user account (name, avatar)
- register a user

5.3.2 Context Engine
The Context engine is the software component tracking sensor data and determining if the user should be added to a Context group. It tracks Wi-Fi signals, GPS location, Bluetooth signals and NFC signals. It executes context rules, as described in Annex B, and evaluates them using fuzzy logic. When a context rule is evaluated above a fixed threshold, the user is added to the given context group.

5.3.3 Plugin runtime
The Plugin runtime is the component launching and executing the Plugins. This component was based on Apache Cordova, an open source project which was designed to create applications which can be packaged for many mobile platforms (e.g. Android, iOS or Windows Phone). It does this by packaging a web application as a native app. A key part of this project is that it can connect a JavaScript API (which is part of the web application) with native code. This is the main mechanism of the Plugin runtime.

In order for the Plugin to know what kind of information to display to the user, an API was developed. This API is for example used to tell the Plugin what data a card contains, what the state of the card is and how data is exchanged between different instances of Cards.

A detailed description of the Plugin API can be found in the Plugin SDK documentation.

5.4 Serverless communication

Although in the Netherlands a well-functioning wireless Internet connection is almost ubiquitous, this is not always true for large events or on other places of the world. The expectation is that when Onlive is able to function without an Internet connection, this might be a huge competitive advantage in those markets.

A technology that has the potential to make this happen is mesh-networking (see https://en.wikipedia.org/wiki/Mesh_networking). In a mesh network, nodes (such as a smartphone) connect to some nearby nodes using a close range networking technology such as Wi-Fi or Bluetooth. Two nodes that want to exchange data, A and B, might not be connected directly, but may be connected indirectly through another node C. In a mesh-network, a virtual, bigger network is created using the connections that every node has. Data sometimes has to make several “hops” to reach its destination.

Mesh networking has the potential to be used for the communication requirements for Onlive. Since Onlive focuses on the local environment, it makes sense to also distribute data locally, instead of always relying on a central server. For example, if a user wants to publish a card in the Nearby group, it could communicate the card data using a mesh network, without the requirement for an Internet connection. Even Plugins themselves have the potential to be transmitted using mesh networks. When a Plugin is not available on a smartphone, it can be downloaded from another smartphone in the network. The Onlive technical concept was designed in such a way all functionality must be available when only connected through a mesh network.

There are several libraries available already implementing mesh networks for popular mobile platforms. They usually operate with a combination of Bluetooth communication, BLE, Wi-Fi direct or Wi-Fi infrastructure (i.e. a Wi-Fi access point), sometimes using Internet as a back-up solution.

Research was done to all existing systems already available for providing mesh networking. Implementing a mesh networking solution was unfortunately not feasible during this project. The following tables display the available technologies as well as their most important properties. In this research also technologies for discovering nearby devices were included (relevant for determining members of the Nearby group).

	Bluetooth	BLE	Ultrasound	Wi-Fi direct
Serval Project	Yes	?	No	Yes
Underdark	Yes	?	No	Yes

Open Connectivity	?	No	?	?
WebRTC	No	No	No	No
MeshKit	?	?	No	?
p2pkit	?	Yes	No	?
Hype	Yes	Yes	No	Yes
Briar project	Yes	Yes	No	Yes
Google Nearby	Yes	Yes	Yes	No
Quiet	No	No	Yes	No
Rumble	Yes?	?	No	Yes?

	Wi-Fi infrastructure	**Interoperable**	**Needs Internet**	**Nearby discovery**
Serval Project	Yes	No	No	Yes?
Underdark	Yes	Yes (via infrastructure)	No	No
Open Connectivity	?	?	?	?
WebRTC	Yes	Yes	Yes	No
MeshKit	Yes	?	?	?
p2pkit	?	Yes	Yes	Yes
Hype	Yes	?	No	?
Briar project	Yes	No	No	?
Google Nearby	Yes	Yes	Yes	Yes
Quiet	No	Yes?	No	Yes
Rumble	Yes	No	No?	Yes

From this research was concluded that Serval project, Underdark and Briar have the most potential. In order to access the quality of these solutions, a proof of concept should be developed.

6 Conclusions and further work

During the project a lot of developments have taken place. First of all, the Onlive concept was refined. The main driver for this process was the workshops with the project partners, in which use cases were developed which could benefit their business. This way, the Onlive concept has been validated from the perspective of the future Group owners. From a technical point of view, it was demonstrated that the concept is technically feasible and scalable in the future. Research has been done on how the platform could operate without an (always) active Internet connection, giving it the potential as a successful platform for delivering digital services in places where there is no Internet connection. Also, the user interface has been tested on a select group of users using paper prototyping. A proof-of-concept was developed which demonstrates the possibilities of the Onlive platform, and ideas have been developed for business models for the platform.

The next step is to validate the platform in the market. Companies have to be willing to invest in the development of Plugins, and users must be willing to install and use the App. We believe that by creating a proof-of-concept, we have build to tools to take this next step. RSNMC is in the lead of creating real-world applications of Onlive together with the Onlive project partners. This way, we take a step towards a world where the smartphone allows us to communicate with the whole world, but mainly the world directly surrounding us.

7 Appendix A: Mapping of Onlive concepts to XMPP functionality

7.1 Interacting with the Onlive XMPP service

The Onlive XMPP service assists Onlive clients with group and card management.
For the Nearby group there are the following use cases:

- Posting the observation of another nearby user
- Getting nearby group change notifications
- Getting cards & messages of a nearby user

For context group the use cases are:

- Joining a context group
- Leaving a context group
- Getting current group memberships

And the following use cases are generic for both the nearby group and the context group:

- Registering with the Onlive service
- Listing other members of a group
- Publishing a card
- Removing a card
- Sending a message to a card

In the background the Onlive service will monitor the user state to check whether card event subscriptions should be cancelled and whether users should be removed from a group.
Where applicable messages will be designed to match existing or experimental XEPs, this minimizes changes to the code when these new XEPs become generally available.
The Onlive services uses PubSub nodes (XEP-0060) for its storage. PubSub nodes should be managed (created, removed). Some XMPP servers may support node auto creation. In this case they will advertise the feature http://jabber.org/protocol/pubsub#auto-create in the disco items.

7.2 Use cases for Nearby groups

7.2.1 Posting the observation of another nearby user

When the clients receives an event that another user is seen nearby it sends an observation message to the Onlive service

```
<message
    from="alice@example.com/stage"
    to="onlive.example.com"
    id="observ1">
  <observation xmlns="urn:onlive:1">
    <participant jid="bob@example.com"/>
  </observation>
</message>
```

The onlive service processes the message and MIGHT send notification messages to one or more users.

7.2.2 Getting nearby group change notifications

When the Onlive service decides that nearby groups did change it notifies the affected users
The decision to send a change notification can be based upon an observation message that was received before, or by detecting that an user went offline (presence unavailable).

```
<message
    from="nearby@onlive.example.com"
    to="alice@example.com/stage"
    id="notify1">
                                                         <items
node="urn:onlive:nodes:participants:1">
    <item jid="bob@example.com"/>
    <retract jid="carol@example.com"/>
  </items>
</message>
```

Cards that are on the stage from users that are retracted MUST be removed by the client from the stage of the Nearby group.

7.2.3 Getting cards of nearby user

This story can be used to get the cards of new nearby group members or to obtain a list of cards that are published by the user itself.

1. Query the service

```
<iq type="get"
    from="alice@example.com/stage"
    to="nearby@onlive.example.com"
    id="items1">
                                                        <pubsub
xmlns="http://jabber.org/protocol/pubsub">
    <items node="bob@example.com"/>
  </pubsub>
```

```
</iq>
```

2. The Onlive server responds by sending a list of published nearby cards from the queried user

The Onlive server MUST check if the users are connected in their nearby groups. The Onlive service retrieves the items from a PubSub node associated with the user. This step is omitted from this document for brevity.

```
<iq type="result"
    from="nearby@onlive.example.com"
    to="alice@example.com/stage"
    id="items1">
                                              <pubsub
xmlns="http://jabber.org/protocol/pubsub">
    <items node="bob@example.com">
                                                <item
id="bob@example.com#cards#123e4567-e89b-12d3-a4
56-426655440000">
        <plugin xmlns="onlive:plugin">
          <id>onlive:plugin:hello:1</id>
          <data>

qANQR1DBwU4DX7jmYZnncmUQB/9KuKBddzQH+tZ1ZywKK0y
HKnq57kWq+RFtQdCJ

WpdWpR0uQsuJe7+vh3NWn59/gTc5MDlX8dS9p0ovStmNcyL
hxVgmqS8ZKhsblVeu

IpQ0JgavABqibJolc3BKrVtVV1igKiX/N7Pi8RtY1K18toa
MDhdEfhBRzO/XB0+P

AQhYlRjNacGcslkhXqNjK5Va4tuOAPy2n1Q8UUrHbUd0g+x
J9Bm0G0LZXyvCWyKH

kuNEHFQiLuCY6Iv0myq6iX6tjuHehZlFSh80b5BVV9tNLwN
R5Eqz1klxMhoghJOA
          </data>

<published>2003-12-13T18:30:02Z</published>

<updated>2003-12-13T18:30:02Z</updated>
        </plugin>
      </item>
      <!-- more items if applicable -->
    </items>
  </pubsub>
</iq>
```

3. For each card the client MUST get the messages that belong to the card

```
<iq type="get"
    from="alice@example.com/stage"
    to="nearby@onlive.example.com"
    id="items2">
                                              <pubsub
xmlns="http://jabber.org/protocol/pubsub">
                                               <items
node="bob@example.com#cards#123e4567-e89b-12d3-
a456-426655440000"/>
  </pubsub>
</iq>
```

4. The Onlive service retrieves the messages and send them back to the clients

The Onlive service retrieves the items from a PubSub node associated with the user. This step is omitted from this document for brevity.

```
<iq type="result"
    from="nearby@onlive.example.com"
    to="alice@example.com/stage"
    id="items1">
                                              <pubsub
xmlns="http://jabber.org/protocol/pubsub">
                                               <items
node="bob@example.com#cards#123e4567-e89b-12d3-
a456-426655440000">
                                                <item
id="ae890ac52d0df67ed7cfdf51b644e901">
                   <event   xmlns="onlive:event"
jid="alice@example.com">
          <id>onlive:plugin:poll:1:vote</id>
          <data>
              { "pizza": false, "pancakes": true
}
          </data>

<published>2003-12-13T18:30:02Z</published>

<updated>2003-12-13T18:30:02Z</updated>
        </event>
      </item>
      <!-- more items if applicable -->
    </items>
  </pubsub>
</iq>
```

7.3 Use cases for context groups

7.3.1 Joining a context group

1. The client sends a join message to the Onlive service that it wants to join the wonderland channel.

```
<iq type="set"
    from="alice@example.com/stage"
    to="wonderland@onlive.example.com"
    id="E6E10350-76CF-40C6-B91B-1EA08C332FC7">
  <join xmlns="urn:onlive:1">
    <subscribe node="urn:onlive:nodes:cards"/>
                                         <subscribe
node="urn:onlive:nodes:participants"/>
  </join>
</iq>
```

2. The service responds when the join message is successful

```
<iq type="result"
    from="wonderland@onlive.example.com"
    to="alice@example.com/stage"
    id="E6E10350-76CF-40C6-B91B-1EA08C332FC7">
  <join xmlns="urn:onlive:1">
    <subscribe node="urn:onlive:nodes:cards"/>
                                         <subscribe
node="urn:onlive:nodes:participants"/>
  </join>
</iq>
```

When the join is not successful the server will send an error message.

3. The service now adds the user to the PubSub node that holds the active participants of the group.

```
<iq type="set"
    from="onlive.example.com"
    to="pubsub.example.com"
    id="publish1">
                                            <pubsub
xmlns="http://jabber.org/protocol/pubsub">
    <publish node="wonderland:participants">
      <item id="alice@example.com">
              <participant  xmlns="urn:onlive:1"
jid="alice@example.com/stage"/>
      </item>
    </publish>
  </pubsub>
</iq>
```

4. The PubSub service responds to the Onlive service

```
<iq type="result"
    from="pubsub.example.com"
    to="onlive.example.com"
    id="publish1">
                                            <pubsub
xmlns="http://jabber.org/protocol/pubsub">
```

```
    <publish node="wonderland:participants">
      <item id="alice@example.com"/>
    </publish>
  </pubsub>
</iq>
```

5. The PubSub service notifies other participants of the group that Alice just joined.

```
<message    from="wonderland@onlive.example.com"
to="bob@example.com/stage" id="foo">
                                            <event
xmlns="http://jabber.org/protocol/pubsub#event"
>
                                            <items
node="urn:onlive:nodes:participants">
      <item id="alice@example.com"/>
    </items>
  </event>
</message>
```

7.3.2 *Leaving a context group*

1. The client sends a leave message to the Onlive service that it wants to leave the wonderland group.

```
<iq type="set"
    from="alice@example.com/stage"
    to="wonderland@onlive.example.com"
    id="E6E10350-76CF-40C6-B91B-1EA08C332FC7">
  <leave xmlns="urn:onlive:1"/>
</iq>
```

2. The service responds when the leave message is successful

```
<iq type="result"
    from="wonderland@onlive.example.com"
    to="alice@example.com/stage"
    id="E6E10350-76CF-40C6-B91B-1EA08C332FC7">
  <leave xmlns="urn:xmpp:mix:1"/>
</iq>
```

3. The service now removes the user from the PubSub node that holds the active participants of the group.

```
<iq type="set"
    from="onlive.example.com"
    to="pubsub.example.com"
    id="retract1">
```

```
                                          <pubsub
xmlns="http://jabber.org/protocol/pubsub">
    <retract node="wonderland:participants">
      <item id="alice@example.com"/>
    </retract>
  </pubsub>
</iq>
```

4. The PubSub service responds to the onlive service

```
<iq type="result"
    from="pubsub.example.com"
    to="onlive.example.com"
    id="retract1"/>



The PubSub service notifies other participants
of the group that Alice just left.
<message    from="wonderland@onlive.example.com"
to="bob@example.com/stage" id="foo">
                                          <event
xmlns="http://jabber.org/protocol/pubsub#event"
>
                                          <items
node="urn:onlive:nodes:participants">
      <retract id="alice@example.com">
    </items>
  </event>
</message>
```

7.4 Generic use cases

7.4.1 Registering with the Onlive server

Before a client is going to interact with the Onlive server it should create a presence relationship with the Onlive service. This can be done by sending a presence message.

1. The client requests a presence relationship

```
<presence type="subscribe"
          from="alice@example.com"
          to = "onlive.example.com"/>
```

2. The service accepts the presence subscription

```
<presence type="subscribed"
          from="onlive.example.com"
          to="alice@example.com"/>
```

3. The service requests a presence subscription from the client

```
<presence type="subscribe"
          from="onlive.example.com"
          to="alice@example.com"/>
```

4. The client accepts the presence subscription

```
<presence type="subscribed"
          from="alice@example.com"
          to="onlive.example.com"
```

This use case can also be fulfilled by creating a global address list on the server that makes sure that all XMPP users have a subscription to the Onlive service. Some XMPP servers have such a feature.

7.4.2 Listing other members of a group

Clients can query what other participants are part of a group. The service should check if the client that issues the request is also part of the group and is allowed to get the list.

1. The clients sends the message to the group

```
<iq type="get"
    from="alice@example.com/stage"
    to="wonderland@onlive.example.com"
    id="items1">
                                                  <pubsub
xmlns="http://jabber.org/protocol/pubsub">
                                                   <items
node="urn:onlive:nodes:participants"/>
  </pubsub>
</iq>
```

2. In case of a context group the server retrieves the items from the PubSub node

The items can also be read from a local cache or the Nearby user graph. In that case the message below and the response in step 3 are not needed.

```
<iq type="get"
    from="onlive.example.com"
    to="pubsub.example.com"
    id="items2">
                                                  <pubsub
xmlns="http://jabber.org/protocol/pubsub">
    <items node="wonderland:participants"/>
```

```
  </pubsub>
</iq>
```

3. The PubSub service responds with the list

```
<iq type="result"
    from="pubsub.example.com"
    to="onlive.example.com"
    id="items2">
                                          <pubsub
xmlns="http://jabber.org/protocol/pubsub">
    <items node="wonderland:participants">
      <item id="bob@example.com">
              <participant  xmlns="urn:onlive:1"
jid="bob@example.com/stage"/>
      </item>
      <item id="carol@example.com">
              <participant  xmlns="urn:onlive:1"
jid="carol@example.com/stage"/>
      </item>
    </items>
  </pubsub>
</iq>
```

A node may have a large number of items associated with it, in which case it may be problematic to return all of the items in response to an items request. In this case, the service SHOULD return some of the items and note that the list of items has been truncated by including a Result Set Management (XEP-0059) notation. See also XEP-0060.

4. The Onlive service forwards the list to the client.

```
<iq type="result"
    from="wonderland@pubsub.example.com"
    to="alice@example.com/stage"
    id="items1">
                                          <pubsub
xmlns="http://jabber.org/protocol/pubsub">
                                           <items
node="urn:onlive:nodes:participants">
      <item id="bob@example.com"/>
      <item id="carol@example.com"/>
    </items>
  </pubsub>
</iq>
```

7.4.3 Publishing a card

There are some differences between publishing a card to the Nearby group or to a context group but most of the flow looks similar. The main difference is that the cards that are in a context group are not stored on the server. Cards that are

published to the Nearby group are stored on the server in a PubSub node that is affiliated to the user that published the card.
Where there are differences between the flows they are outlined below.

1. Alice publishes a card to the wonderland group

```
<iq type="set"
    from="alice@example.com/stage"
    to="wonderland@onlive.example.com"
    id="publish1">
                                          <pubsub
xmlns="http://jabber.org/protocol/pubsub">
    <publish node="urn:onlive:nodes:cards">
      <item>
        <plugin xmlns="urn:onlive:1">
          <id>urn:onlive:plugin:hello:1</id>
          <data>

qANQR1DBwU4DX7jmYZnncmUQB/9KuKBddzQH+tZ1ZywKK0y
HKnq57kWq+RFtQdCJ

WpdWpR0uQsuJe7+vh3NWn59/gTc5MDlX8dS9p0ovStmNcyL
hxVgmqS8ZKhsblVeu

IpQ0JgavABqibJolc3BKrVtVV1igKiX/N7Pi8RtY1K18toa
MDhdEfhBRzO/XB0+P

AQhYlRjNacGcslkhXqNjK5Va4tuOAPy2n1Q8UUrHbUd0g+x
J9Bm0G0LZXyvCWyKH

kuNEHFQiLuCY6Iv0myq6iX6tjuHehZlFSh80b5BVV9tNLwN
R5Eqz1klxMhoghJOA
          </data>

<published>2003-12-13T18:30:02Z</published>

<updated>2003-12-13T18:30:02Z</updated>
        </plugin>
      </item>
    </publish>
  </pubsub>
</iq>
```

If the client did id to the item element the service MUST ignore that ID and generate one according to the rules.

2. The service creates a PubSub node where messages related to the card should be published to

The card id MUST be unique within the group. The node id is generated as follows:
For the Nearby group:

```
  <bare   JID   of   the   user   publishing   the
card>#cards#<id of the card>
```

For a context group:

```
  <bare   JID   of   the   user   publishing   the
card>#<jid of the group>#cards#<id of the card>
```

```
<iq type="set"
    from="onlive.example.com"
    to="pubsub.example.com"
    id="create1">
                                          <pubsub
xmlns="http://jabber.org/protocol/pubsub">
                                          <create
node="wonderland#cards#123e4567-e89b-12d3-a456-
426655440000"/>
    <configure>
      <x xmlns="jabber:x:data" type="submit">
        <field var="FORM_TYPE" type="hidden">

<value>http://jabber.org/protocol/pubsub#node_c
onfig</value>
        </field>
                                           <field
var="pubsub#deliver_notifications"><value>1</va
lue></field>
                                           <field
var="pubsub#deliver_payloads"><value>1</value><
/field>
                                           <field
var="pubsub#persist_items"><value>1</value></fi
eld>
                                           <field
var="pubsub#max_items"><value>1000</value></fie
ld>
                                           <field
var="pubsub#item_expire"><value>604800</value><
/field>
                                           <field
var="pubsub#access_model"><value>whitelist</val
ue></field>
                                           <field
var="pubsub#publish_model"><value>subscribers</
value></field>
                                           <field
var="pubsub#purge_offline"><value>0</value></fi
eld>
                                           <field
var="pubsub#presence_based_delivery"><value>1</
value></field>
                                           <field
var="pubsub#max_payload_size"><value>1028</valu
e></field>
      </x>
    </configure>
  </pubsub>
```

```
</iq>
```

Creating a pubsub node requires server support for create-nodes and the user requesting the node should be allowed to create pubsub nodes.

3. The PubSub server responds that the node was created.

```
<iq type="result"
    from="pubsub.example.com"
    to="onlive.example.com"
    id="create2">
                                            <pubsub
xmlns="http://jabber.org/protocol/pubsub">
                                            <create
node="wonderland#cards#123e4567-e89b-12d3-a456-
426655440000"/>
    </pubsub>
</iq>
```

4. In case of the Nearby group the service adds the card to the PubSub node associated with the user.

Note that context groups do not store cards on the server, hence this step is omitted for cards that are published to context groups

```
<iq type="set"
    from="onlive.example.com"
    to="pubsub.example.com"
    id="publish3">
                                            <pubsub
xmlns="http://jabber.org/protocol/pubsub">
    <publish node="alice@example.com">
                                              <item
id="alice@example.com#cards#123e4567-e89b-12d3-
a456-426655440000">
        <plugin xmlns="urn:onlive:1">
          <id>urn:onlive:plugin:hello:1</id>
          <data>

qANQR1DBwU4DX7jmYZnncmUQB/9KuKBddzQH+tZ1ZywKK0y
HKnq57kWq+RFtQdCJ

WpdWpR0uQsuJe7+vh3NWn59/gTc5MDlX8dS9p0ovStmNcyL
hxVgmqS8ZKhsblVeu

IpQ0JgavABqibJolc3BKrVtVV1igKiX/N7Pi8RtY1K18toa
MDhdEfhBRzO/XB0+P

AQhYlRjNacGcslkhXqNjK5Va4tuOAPy2n1Q8UUrHbUd0g+x
J9Bm0G0LZXyvCWyKH
```

```
kuNEHFQiLuCY6Iv0myq6iX6tjuHehZlFSh80b5BVV9tNLwN
R5Eqz1klxMhoghJOA
          </data>

<published>2003-12-13T18:30:02Z</published>

<updated>2003-12-13T18:30:02Z</updated>
        </plugin>
      </item>
    </publish>
  </pubsub>
</iq>
The PubSub server responds that the card was
added correctly:
<iq type="result"
    from="pubsub.example.com"
    to="onlive.example.com"
    id="publish3">
                                          <pubsub
xmlns="http://jabber.org/protocol/pubsub">
    <publish node="alice@example.com">
                                            <item
id="alice@example.com#cards#123e4567-e89b-12d3-
a456-426655440000"/>
    </publish>
  </pubsub>
</iq>
```

In case the node does not exsist the Onlive service MUST create the node before adding the card to the node.

5. The service forwards the card to other members of the group

The card will not be stored on the server but will only be available at the clients.

```
<message
    from="wonderland@onlive.example.com"
    to="example.com">
                                       <addresses
xmlns="http://jabber.org/protocol/address">
                        <address      type="bcc"
jid="bob@example.com/stage"/>
                        <address      type="bcc"
jid="carol@example.com/stage"/>
                   <address     type="replyroom"
jid="wonderland@onlive.example.com"/>
  </addresses>
                                         <publish
xmlns="http://jabber.org/protocol/pubsub"
node="urn:onlive:nodes:cards">
                                            <item
id="alice@example.com#cards#123e4567-e89b-12d3-
a456-426655440000">
      <plugin xmlns="urn:onlive:1">
```

```
        <id>urn:onlive:plugin:hello:1</id>
        <data>

qANQR1DBwU4DX7jmYZnncmUQB/9KuKBddzQH+tZ1ZywKK0y
HKnq57kWq+RFtQdCJ

WpdWpR0uQsuJe7+vh3NWn59/gTc5MDlX8dS9p0ovStmNcyL
hxVgmqS8ZKhsblVeu

IpQ0JgavABqibJolc3BKrVtVV1igKiX/N7Pi8RtY1K18toa
MDhdEfhBRzO/XB0+P

AQhYlRjNacGcslkhXqNjK5Va4tuOAPy2n1Q8UUrHbUd0g+x
J9Bm0G0LZXyvCWyKH

kuNEHFQiLuCY6Iv0myq6iX6tjuHehZlFSh80b5BVV9tNLwN
R5Eqz1klxMhoghJOA
        </data>

<published>2003-12-13T18:30:02Z</published>
        <updated>2003-12-13T18:30:02Z</updated>
      </plugin>
    </item>
  </publish>
</message>
```

6. The service confirms that the card was published correctly

```
<iq type="result"
    from="wonderland@onlive.example.com"
    to="alice@example.com/stage"
    id="publish1">
                                        <pubsub
xmlns="http://jabber.org/protocol/pubsub">
    <publish node="urn:onlive:nodes:cards">
                                          <item
id="alice@example.com#cards#123e4567-e89b-12d3-
a456-426655440000"/>
    </publish>
  </pubsub>
</iq>
```

7.4.4 *Removing a card*

When a user removes a card from the group it send a request to the onlive service. The service sends a card removal message to other members of the group and removes the message PubSub node of that card.

1. Alice request removal of her card

```
<iq type="set"
```

```
    from="alice@example.com/stage"
    to="wonderland@onlive.example.com"
    id="retract1">
                                          <pubsub
xmlns="http://jabber.org/protocol/pubsub">
         <retract  node="urn:onlive:nodes:cards"
notify="true">
                                            <item
id="wonderland@onlive.example.com#cards#123e456
7-e89b-12d3-a456-426655440000"/>
    </retract>
  </pubsub>
</iq>
```

2. The Onlive service removes the PubSub node that holds messages of the card

```
<iq type="set"
    from="onlive.example.com"
    to="pubsub.example.com"
    id="delete1">
                                          <pubsub
xmlns="http://jabber.org/protocol/pubsub#owner"
>
                                          <delete
node="wonderland@onlive.example.com#cards#123e4
567-e89b-12d3-a456-426655440000"/>
  </pubsub>
</iq>
```

3. The PubSub service responds to the removal request

```
<iq type="result"
    to="onlive.example.com"
    from="pubsub.example.com"
    id="delete1"/>
```

4. In case the card is published in the nearby group the card is removed from the node containing the user's nearby cards

```
<iq type="set"
    from="onlive.example.com"
    to="pubsub.example.com"
    id="retract2">
                                          <pubsub
xmlns="http://jabber.org/protocol/pubsub">
    <retract node="urn:onlive:nodes:cards">
                                            <item
id="wonderland#cards#123e4567-e89b-12d3-a456-42
6655440000"/>
    </retract>
  </pubsub>
</iq>
The PubSub server responds to the request:
```

```
<iq type="result"
    from="pubsub.example.com"
    to="onlive.example.com"
    id="retract2"/>
```

5. The Onlive service forwards the card retraction request to other members of the group

The group identifier of the group the card should be retracted from is identified by the localpart of the JID of the sender. In the case below it is wonderland.

```
<message
    from="wonderland@onlive.example.com"
    to="example.com">
                                         <addresses
xmlns="http://jabber.org/protocol/address">
                        <address      type="bcc"
jid="bob@example.com/stage"/>
                        <address      type="bcc"
jid="carol@example.com/stage"/>
                   <address     type="replyroom"
jid="wonderland@onlive.example.com"/>
  </addresses>
                                             <event
xmlns="http://jabber.org/protocol/pubsub#event"
>
    <items node="urn:onlive:nodes:cards">
                                           <retract
id="wonderland#cards#ae890ac52d0df67ed7cfdf51b6
44e901"/>
    </items>
  </event>
</message>
```

6. Finally the Onlive service confirms the removal to the requesting client

```
<iq type="result"
    from="onlive.example.com"
    to="alice@example.com/stage"
    id="retract1"/>
```

7.4.5 Sending a message to a card

Messages to cards in the Onlive group are send via the Onlive service. This service acts as a proxy between the PubSub node of the card and the clients that have access to the card. This proxy function should check whether clients are still in the group and should be able to post and receive messages on a card.

1. Events on the cards are send via the Onlive service

```
<iq type="set"
    from="alice@example.com/stage"
    to="nearby@onlive.example.com"
    id="publish1">
                                         <pubsub
xmlns="http://jabber.org/protocol/pubsub">
                                        <publish
node="bob@example.com#cards#123e4567-e89b-12d3-
a456-426655440000">
      <item>
                   <event  xmlns="onlive:event"
jid="alice@example.com">
          <id>onlive:plugin:poll:1:vote</id>
          <data>
              { "pizza": false, "pancakes": true
}
          </data>

<published>2003-12-13T18:30:02Z</published>

<updated>2003-12-13T18:30:02Z</updated>
        </event>
      </item>
    </publish>
  </pubsub>
</iq>
```

2. The server adds the message to the PubSub node of the messages

The server MUST check if Alice is still in the same Nearby group as Bob before adding the message.

```
<iq type="set"
    from="onlive.example.com"
    to="pubsub.example.com"
    id="publish2">
                                         <pubsub
xmlns="http://jabber.org/protocol/pubsub">
                                        <publish
node="bob@example.com#cards#123e4567-e89b-12d3-
a456-426655440000">
      <item>
                   <event  xmlns="onlive:event"
jid="alice@example.com">
          <id>onlive:plugin:poll:1:vote</id>
          <data>
              { "pizza": false, "pancakes": true
}
          </data>

<published>2003-12-13T18:30:02Z</published>

<updated>2003-12-13T18:30:02Z</updated>
        </event>
      </item>
    </publish>
  </pubsub>
</iq>
```

3. The PubSub service confirms that the message was added

```
<iq type="result"
    from="pubsub.example.com"
    to="onlive.example.com"
    id="publish2">
                                               <pubsub
xmlns="http://jabber.org/protocol/pubsub">
                                              <publish
node="bob@example.com#cards#123e4567-e89b-12d3-
a456-426655440000">
                                                 <item
id="ae890ac52d0df67ed7cfdf51b644e901"/>
    </publish>
  </pubsub>
</iq>
```

4. The service confirms that the message was send

```
<iq type="result"
    from="nearby@onlive.example.com"
    to="alice@example.com/stage"
    id="publish1">
                                               <pubsub
xmlns="http://jabber.org/protocol/pubsub">
                                              <publish
node="bob@example.com#cards#123e4567-e89b-12d3-
a456-426655440000">
                                                 <item
id="ae890ac52d0df67ed7cfdf51b644e901"/>
    </publish>
  </pubsub>
</iq>
```

5. The server sends the notification to members of Bob"s nearby group

```
<message
    from="nearby@onlive.example.com"
    to="example.com">
                                            <addresses
xmlns="http://jabber.org/protocol/address">
                        <address      type="bcc"
jid="bob@example.com/stage"/>
                        <address      type="bcc"
jid="carol@example.com/stage"/>
                   <address     type="replyroom"
jid="nearby@onlive.example.com"/>
  </addresses>
```

```
                                              <event
xmlns="http://jabber.org/protocol/pubsub#event"
>
                                              <items
node="bob@example.com#cards#123e4567-e89b-12d3-
a456-426655440000">
                                               <item
id="ae890ac52d0df67ed7cfdf51b644e901">
                    <event   xmlns="onlive:event"
jid="alice@example.com">
          <id>onlive:plugin:poll:1:vote</id>
          <data>
              { "pizza": false, "pancakes": true
}
          </data>

<published>2003-12-13T18:30:02Z</published>

<updated>2003-12-13T18:30:02Z</updated>
        </event>
      </item>
    </items>
  </event>
</message>
```

8 Appendix B: Context description language

Context description

The group selection process is all about the context of the smartphone. Smartphones have a lot of sensors to gather data from their surroundings, such as a GPS sensor, which can be used to determine the geographical location, a Bluetooth adapter, which can be used to detect Bluetooth beacons and a Wi-Fi adapter, which can be used to detect access points.
The process of selecting relevant groups works as follows: the owner of the group describes in which context the group is relevant. We call this the context description. The Onlive platform regularly checks if the current context information matches the description. If there is a match, the group is presented to the user.

8.1 Context Description Language

In order to describe a certain context, we developed a context description language. With this language a developer can describe a context. For all the context sources, one or more predicates are available. For example, if a group is relevant at a certain geographical location, the GeographyCirclePredicate can be used to indicate that a group is relevant at a certain location (described as latitude and longitude coordinates) with a radius (described in meters). When the smartphone is inside this circle, the group will be presented.
It is possible to make logical combinations between predicates. For example, a group for visitors of a restaurant might be relevant at a certain location AND when the restaurant is open. Predicates can be combined using logical operators.
Since context information is never perfect, we have to deal with uncertainty. For example, most of the time there will be a rough estimate on where the smartphone is, and not a precise location. In order to deal with uncertainty, fuzzy logic is used to evaluate the expression. This means that an expression does not evaluate to be true or false, but that the truth of the expression is indicated as a number between 0 and 1. A value of 0 would mean absolutely false and value of 1 means absolutely true. When the truthness reaches a certain threshold, the group is considered likely to be relevant and is presented to the user.
In the Context sources section you can see some examples of a context description for a certain context source. In the Operations section you can see how you can combine context descriptions in order to make logical combinations.

8.2 Context sources

There are currently several context sources implemented and available to group owners. In future versions of the Onlive Platform more context sources might be added.

8.2.1 Bluetooth

Name: BluetoothMacPredicate
Arguments: Mac-address (String)
Relevant: When device with given mac address is detected
Example:

```
{
  "type": "bluetooth.mac",
  "mac": "06-00-00-00-00-00"
}
```

Name: BluetoothAdvertisementPredicate
Arguments: UUID (String), major (Integer), minor (Integer)
Relevant: When given bluetooth beacon is detected
Example:

```
{
  "type": "bluetooth.advertisement",
                                         "uuid":
"123e4567-e89b-12d3-a456-426655440000",
  "major": 0,
  "minor": 1
}
```

8.2.2 Clock

Name: ClockDateRangePredicate
Arguments: start date (String), end date (String)
Relevant: When between given dates
Example:

```
{
  "type": "clock.daterange",
  "dateStart": "Apr 21, 2016 1:13:07 PM",
  "dateStop": "Apr 21, 2016 1:13:07 PM"
}
```

Name: ClockDayOfWeekPredicate
Arguments: day of week in capitals (String)
Relevant: When today is given weekday
Example:

```
{
  "type": "clock.dayofweek",
  "weekday": "MONDAY"
}
```

Name: ClockTimeOfDayPredicate
Arguments: minutes after midnight start (Integer), minutes after midnight end (Integer)
Relevant: When time is between interval
Example:

```
{
  "type": "clock.timeofday",
  "minutesStart": 540,
  "minutesStop": 1020
}
```

8.2.3 *Geography*

Name: GeographyPolygonPredicate
Arguments: List of points forming a polygon
Relevant: When inside given polygon
Example:

```
{
  "type": "geography.polygon",
  "geographyDatas": [
    {
      "latitude": 53.1996,
      "longitude": 6.522923,
      "altitude": 0.0,
      "accuracy": 1.0
    },
    {
      "latitude": 53.199728,
      "longitude": 6.523921,
      "altitude": 0.0,
      "accuracy": 1.0
    },
    {
      "latitude": 53.199053,
      "longitude": 6.524189,
      "altitude": 0.0,
      "accuracy": 1.0
    },
    {
      "latitude": 53.198967,
      "longitude": 6.523148,
      "altitude": 0.0,
      "accuracy": 1.0
    }
  ]
}
```

Name: GeographyCirclePredicate
Arguments: Center (latitude longitude) and radius in meters (Double)
Relevant: When inside given circle
Example:

```
{
  "type": "geography.circle",
  "center": {
    "latitude": 53.1996,
    "longitude": 6.522923,
    "altitude": 0.0,
    "accuracy": 1.0
  },
  "radius": 100.0
}
```

8.2.4 NFC

Name: NfcTextPredicate
Arguments: contexts of NFC tag (String)
Relevant: 30 seconds after a NFC tag with the given String is data is scanned
Example:

```
{
  "type": "nfc.text",
  "text": "contents"
}
```

8.2.5 Wi-Fi

Name: MacPredicate
Arguments: MAC address of access point (String)
Relevant: When access point with given MAC address is in range
Example:

```
{
  "type": ".mac",
  "mac": "06-00-00-00-00-00"
}
```

Name: WifiWi-FiSsidPredicate
Arguments: SSID of access point (String)
Relevant: When access point with given SSID is in range
Example:

```
{
  "type": ".ssid",
  "ssid": "SSID"
}
```

8.2.6 Constants

There are also three constants. They can be useful for testing or for some logical expressions.
True, will make the group always appear. Corresponds to the fuzzy-logical value of 1.

```
{
  "type": "true"
}
```

False, will make the group never appear. Corresponds to the fuzzy-logical value of 0.

```
{
  "type": "false"
}
```

Unknown, useful for testing. Corresponds to the fuzzy-logical value of 0.5.

```
{
  "type": "unknown"
}
```

8.2.7 Operators

There are three operators in order to make logical combinations between predicates: The And operator, the Or operator and the Not operator.
Example of the AND predicate
Evaluation: and(p1, p2) = p1 * p2

```
{
  "type": "and",
  "predicate1": {
    "type": "true"
  },
  "predicate2": {
    "type": "false"
  }
}
```

Example of the OR predicate
Evaluation: or(p1, p2) = 1 - ((1 - p1) * (1 - p2))

```
{
  "type": "or",
  "predicate1": {
    "type": "true"
  },
  "predicate2": {
    "type": "false"
  }
}
```

Example of the NOT predicate
Evaluation: not(p) = 1 - p

```
{
  "type": "not",
  "predicate": {
    "type": "false"
  }
}
```