DEVELOPMENT OF SOCIAL AND EVENT MANAGEMENT DISTRIBUTED APPLICATION

(Розробка соціального розподіленого додатку керування подіями)

by Dmytro Kulomin

Presented in Partial Fulfilment of the Requirements for the Degree

Master of Software Engineering

American University Kyiv

2024

APPROVED BY:

Viktor Putrenko, Dr. Habil., Prof.

Table of contents

Table of contents **1**

Abstract **3**

Introduction **4**

Description of the industry 4

Relevance of the topic 5

Goal 6

Tasks 6

Structure of the work 7

Main part **8**

Use cases 8

Non-functional requirements 13

High-level architecture 14

Main implementational decisions 17

Load balancing and service discovery 18

Load balancing 18

Service discovery 20

Communication 22

Client-server communication 22

Server-server communication 23

Storage 28

Database types 29

DB solution selection 31

Conclusions **34**

Used sources **35**

Appendices **36**

Application screens 36

Abstract

Purpose of this work is a creation of the product helping people and business to find each other, plan and organise activities and events. During the work on the problem different use-case were analysed and matched against existing solutions on the market.

Work consists of observation of current state of industry, available products comparison, fit map creation and main use-cases identification.

The result of the work is presented by the engineering solution in the form of main architectural decisions, storage and communication approaches analysis and the proof of concept implementation.

Introduction

Description of the industry

Industry of the domain is represented by numerous products which usually solve a single problem:

- Board of events;
- Tickets selling and business promotion;
- Dating;
- Meeting new people;
- Business events.

One of the most significant and radically different players on the market are Google with its Calendar and Meta's Facebook Events. While both of them solve the problem of the scheduling they still cover only a part of possible features like: events scheduling and collisions handling or business promotion with the help of ads and a huge user base. Events binded to coordinates and subsequent opportunities related to integration with other businesses and people, their promotion based on user interests, activities or rating hasn't become a central point for none of them.

Among others, next companies who share the same niche can be highlighted:

- Eventbrite - for business events discovery;
- Meetup - for searching events nearby;
- Friender - for making friends.

Relevance of the topic

Modern world is very dynamic and open. Vast distances can be overcomed within hours. A lot of people travel, move to other cities because of opportunities or even decide to immigrate. Most people in new places face the same problems: lack of knowledge of a new place (entertainment, local spots, attractions), limited or no relationships, paused hobbies. Another category - active people who are open to the world, want to meet new friends, create connections or develop a community.

Several companies have created products which try to solve mentioned problems with different functional set and purposes. Usually they focus on a single area: business promotion, dating, friendship, hobbies etc.

It is worth mentioning that existing products neither perceive end users and businesses as equal participants, nor provide rich integration with service suppliers inside the product. For example, it's impossible to select the concert as an event, pay for a ticket, order a taxi and save all these parts as a unique, atomic chain of future activity and complete all these actions in a single application.

Next table demonstrates comparison of existing products from the perspective of different features.

Product	Event scheduling	Rich info about an event/place	Geospatial information	Public/Private events	Information about participants and host	External services integration	Event broadcasting	Communication among participants
Google calendar	yes	no	no	no	yes (weak)	no	no	no
Eventbrite	yes	no	yes (weak)	yes (public)	yes (host)	Yes (payments)	no	no
Friender	yes	no	yes (weak)	yes	yes	no	no	yes

foursquare		yes	yes	no	yes	no	no	yes (comments)

As can be seen there is no single app which allows user to use their location as a main source of events or spots depending on current needs (professional, interests, wishes) receiving information not only in a form of textual description, but also in graphical and video representation, with possibility to collaborate with other participants (similar to Uber for rides, Foursquare for places discovery or Booking for hotel booking). All mentioned products do not provide additional services (besides concert tickets buying in one of them). At the same time business is also limited in long-term communication with its clients, because of unidirectional way of communication (business proposes events, users join or ignore them).

Goal

Purpose of this work is a creation of the new product which integrates people, businesses and services they provide as a single network of first-class citizens with the help of geospatial information and locality. Of course the amount of tasks to be done is huge and cannot be accomplished is the scope of current work. Goal of the work in the first phase is narrowed down to: main use cases definition, architectural significant requirements definition, design of core parts of the system and proof of concept implementation.

Tasks

1. Functional requirements definition in a form of use cases;

2. Non-functional requirements and architecture significant requirements definition;
3. High-level architecture definition;
4. Main implementational decisions definition;
5. Communication patterns analysis and selecting best option from the perspective of architectural requirements;
6. Storages analysis and solution selection based on requirements and model structure;
7. Proof of concept implementation (critical services and part of main functional requirements).

Structure of the work

This document describes main functional and technical decisions in logical order. Work on the system starts from high level parts which are broken down into detailed analysis and description in subsequent paragraphs.

Main part

Use cases

This paragraph describes main use cases according to which core functionality of the product must be created.

Use case: Account creation

Actor: New user

Flow:

1. Use opens the app and sees login screen;
2. User presses button to create an account;
3. Registration screen proposes to specify email and password;
4. Application verifies that user is not already registered and password complies to minimum level of complexity (length, letters and digits, special symbols);
5. User presses on the Register button and is redirected to the Login screen.
6. Email with account creation confirmation link (with expiration of 2 hours) is sent to specified email.

Use case: Email confirmation

Actor: New user

1. User receives confirmation link in their mailbox;
2. User user follows the link;
3. If the link is not expired and a valid confirmation message is shown, otherwise - message describing a problem is shown.

Use case: Email confirmation

Actor: New user

1. User receives confirmation link in their mailbox;
2. User user follows the link;
3. If the link is not expired and a valid confirmation message is shown, otherwise - message describing a problem is shown.

Use case: Password reset

Actor: Registered user

1. On login screen user enters registration email and presses Password reset button;
2. Reset email is sent to user if account if valid;
3. User follows the link and enters new password;
4. New password is validated according to security rules which are equal to Registration case.
5. Password is updated.

Use case: Login to the application

Actor: New user

1. User enters email and password;
2. If credentials are correct main screen of the app is opened;
3. User is proposed to enter basic information: first name, last name, birthday, user’s country and city, interesting details;
4. Once information is stored, user can start to use the main features of the application.

Use case: Login to the application

Actor: User

1. User enters email and password;
2. If credentials are correct main screen of the app is opened;
3. Once information is stored, user can start to use the main features of the application.

Use case: Login to the application

Actor: User (event seeker)

1. User opens the app;
2. Main screen with list of events nearby (based on user geoposition) is presented to user;
3. Alternatively user can open the map and see public events in the form of balloons distributed across the map.

Use case: Event creation from Event creation screen

Actor: Event owner

1. User open the app;
2. User navigates to Event creation screen;
3. User enters Event name, from/to date and time, event description;
4. Use enters location of the event via selection a location on the map and then selects an address from automatically generated drop list;
5. User selects event type: public or private;
6. User submits an event with pressing on Create button;
7. Event is created and can be viewed in the My events screen.

Use case: Public event participation

Actor: Event participant

1. User selects available public event on the main screen or on the map;
2. User opens an event info;
3. User presses Participate button;
4. Information about event's participants is updated;
5. Event is added to the My events screen.

Use case: Private event participation

Actor: User

1. User finds private event on the main screen;
2. User opens event info and presses Participate button;
3. User sees "Participation request was sent" message;
4. Event owner receives "Participation request notification".

Use case: Private event participation accepting

Actor: Event owner

1. Owner opens Notifications screen;
2. Notification consists of message and 3 buttons (Accept, Reject, Participant info);
3. Owner accepts participance;
4. Information about event participants is updated;
5. Notification is sent to participant.

Use case: Notifications reading

Actor: User

1. User opens Notifications screen;

2. User sees their unread notifications (of different types: informational or requiring an action);
3. User performs action on notification;
4. Necessary operation is performed (mark as read, accept participant, reject participant);
5. Notification is removed.

Use case: Information about event reading
Actor: User

1. User opens map or main screen;
2. User opens an event;
3. User can see event description, current participants amount and address (if event if public) and event owner.

Use case: Information about event where user participates reading
Actor: Event owner

1. User opens My events screen;
2. User selects an event;
3. User can see information about an event, participants list (with basic information about each participant), event owner and location (location can be also shown on the map).

Use case: Event editing
Actor: Event owner

1. Owner opens My events screen;
2. Owner selects own event;
3. Owner opens an event and edit information inside;
4. Owner save changes;

5. If critical information was changed (event date and time) a notification is sent to participants.

Non-functional requirements

There is a probability that the product will be popular with continuous growth of active users creating and participating in events. It is a goal to build the product which will be used worldwide. In perspective, integrations with local businesses will be created. It is important to guarantee that all users are provided the same level of service (quality, speed, reliable connection) and language support. Client side part of the product has to make the most of features which user devices provide being fast and effective. Information which users share with the product must be stored in a safe and durable way.

Next items can be highlighted as non-functional requirements:

- Availability;
- Durability;
- Efficiency;
- Elasticity;
- Maintainability;
- Extensibility;
- Observability;
- Internationalisation and localisation.

The whole set of the functionality the product will consist of is unknown and can be changed/extended dynamically. Development of

the product should not be limited with initially selected technologies, but instead new technologies must be onboarded easily depending on business needs. One of the significant requirements is to make the product cost-effective which takes maximum from the available resources and release resources when they are not needed or request new ones in runtime via automatic scaling. Solution must not depend on the environment specifics where it currently runs (for example be cloud agnostic), but at the same time the usage of cloud managed services is preferable. Backend part in production has to require as little human attention as possible.

High-level architecture

Architectural style of the product is dictated by uncertainty of the product evolution and a desire of making it cost-effective and flexible for extension. Often, companies with limited financial capabilities for development of their product lean toward monolithic architectures because of instantaneous benefits this approach can offer: easier dependency management, deployment process, minimum amount of network IO operations and set of complexities related to it. At the same time the current state of industry is well developed and provides multiple ready to use and tested frameworks and protocols which reduce probability of simple glaring mistakes (service discovery solutions, optimal serialisation/deserialisation protocols, solutions for interprocess and distributed communication etc).

General architecture of the current product will be based on service oriented architecture with the shift to microservices approach. Such a decision is intentional because there are no strict requirements

related to the size of each service. Not all services will be “micro” and their size, structure and components decomposition are based on the technical requirements and possible benefits of the concrete solution.

Such hybrid approach provides next significant benefits:

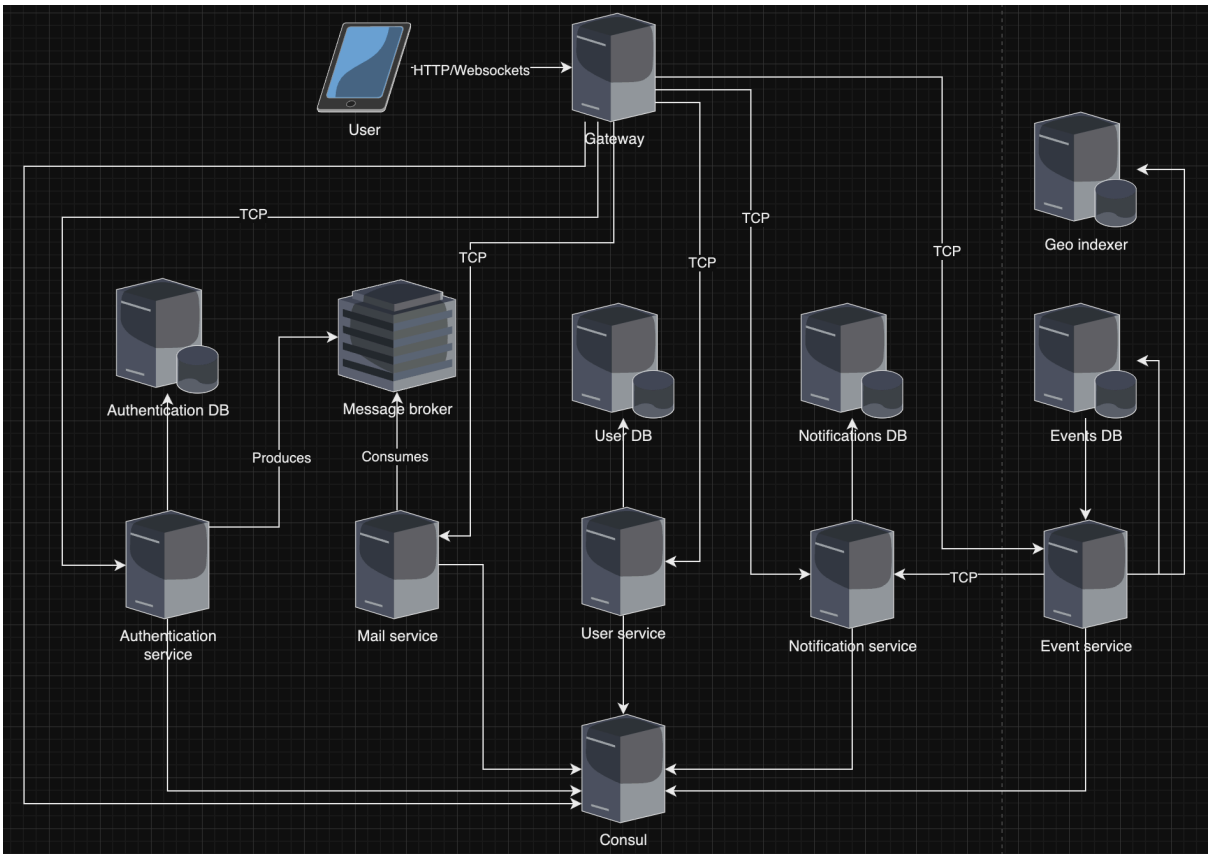
- Usage of specific technologies depending on specific needs;
- Independent development and deployment;
- Better utilisation of resources depending on service needs;
- Service’s scope, architecture and communication mechanism are selected independently from existing parts of the system and according to business needs and non-functional requirements.

Drawbacks of such an approach are common for all distributed solutions: more complex infrastructure, complexities related to communication via network (delays, failed requests, cascade failures etc), necessity of services discovery mechanism, optimal protocols etc.

Next diagram presents a high-level architectural view of the system. It consists of next components:

- User (mobile app) - native implementation of the client for Android and iOS platforms;
- Gateway - entry point to the system responsible for requests routing, caching, session management. Consists of dynamically loaded modules each of them presenting their own part of functionality. A module take not only a responsibility of requests routing, but also operate as an orchestrator among several services;
- Authentication service - responsible for authentication/authorisation details of user account;

- Authentication DB - stores authentication/authorisation related information (internal ID, password, email etc);
- Mail service - service for asynchronous communication with users (emails);
- Message broker - work as a durable transport for stateless services;
- User service - stores all user related information, different from authentication;
- User DB - dedicated storage for user related information;
- Notification service - asynchronous in-app communication channel;
- Notification DB - storage for notifications which must be delivered/still unseen;



- Event service - responsible for everything related to events management and delivery;

- Event DB - storage for events related information;
- Geo indexer - index storage of geospatial information for events fetching in defined radius according to location of a user.
- Consul - service discovery and configuration infrastructure.

Main implementational decisions

Backend part of the system must be implemented with the help of industry standard technologies and frameworks. Next ones must be considered as primary ones:

- Java - for services implementation, big data, ML, distributed jobs;
- Go - for services implementation;
- Rust - for services implementation, specially when performance and is important;
- Industry standard frameworks like Spring, Guava etc must be used;
- AOT compilation can be used to decrease the size of Java deployments and startup.

Implementation of the backend must be cloud agnostic, so all interprocess communication and external accesses must be hidden behind abstractions. This approach provides flexibility in selection of concrete solutions distilling at the same time business logic from infrastructural details. Of course such goal is labour intensive and error prone when it comes to covering all possible cases, so the problem can be narrowed down with the help of next requirements:

- All interprocess communication must be asynchronous (nonblocking);
- Biggest part of the system is eventually consistent, which means strong consistency or linearisation are not required (such fact can simplify selection of storages or ways of error handling across distributed participants);
- It’s desired to solve tasks biassing to algorithm effectiveness, calculations and verified design decisions, rather to vendor specific features.

Load balancing and service discovery

Load balancing

Load balancing can be of two types: client-side load balancing and backend-side load balancing.

Client-side load balancing - is a form of load balancing where each client in the network knows about all its service providers and picks an instance to which request will be sent based on defined strategy (round robin, random, consistent hashing, by telemetry etc). Such approach often resides on L7 of the OSI model and is usually used in cases when a target of load balancing and its client are both part of the same system. Benefits of client-side balancing are: higher availability of the system (no single point of failure), higher throughput, simpler maintenance. Among drawbacks, the next ones can be highlighted: logic of traffic distribution is split among all clients, higher risk of security issues, more complex monitoring and administration.

Server-side load balancing - based on central component (group of components) proxying and distributing traffic among consumers. Such centralising is conceptual and can consist of a hierarchy of multiple balancers residing on different OSI levels which form massive and reliable solution (can be used in complex starting from edge point which balances egress traffic on OSI L3/L4 and then rising up to L7 to deliver requests to final consumers). Benefits: easier management and maintenance, higher security guarantees, works on all OSI levels, specialised hardware is available. Drawbacks: requires a dedicated environment (or is bought as a service), can be a single point of failure if set up without redundancy, requires additional efforts of management and maintenance.

DNS load balancing - this is a part of the idea of domain name resolution, where a name can be mapped to a set of IP addresses. DNS LB is often used as a part of server-side load balancing and one of its first steps. There is no big sense in DNS usage alone for load balancing, because of its caching nature and no possibility of actual balancing based on consumer’s health, presence load or throughput.

Taking into account that the product will be hosted on one of the main client providers, the problem of egress traffic load balancing is solved by the cloud provider. One of the future goals is utilisation of service mesh infrastructure. Taking into account pros and cons listed above and future possibility of switching to mesh, client-side load balancing looks preferable.

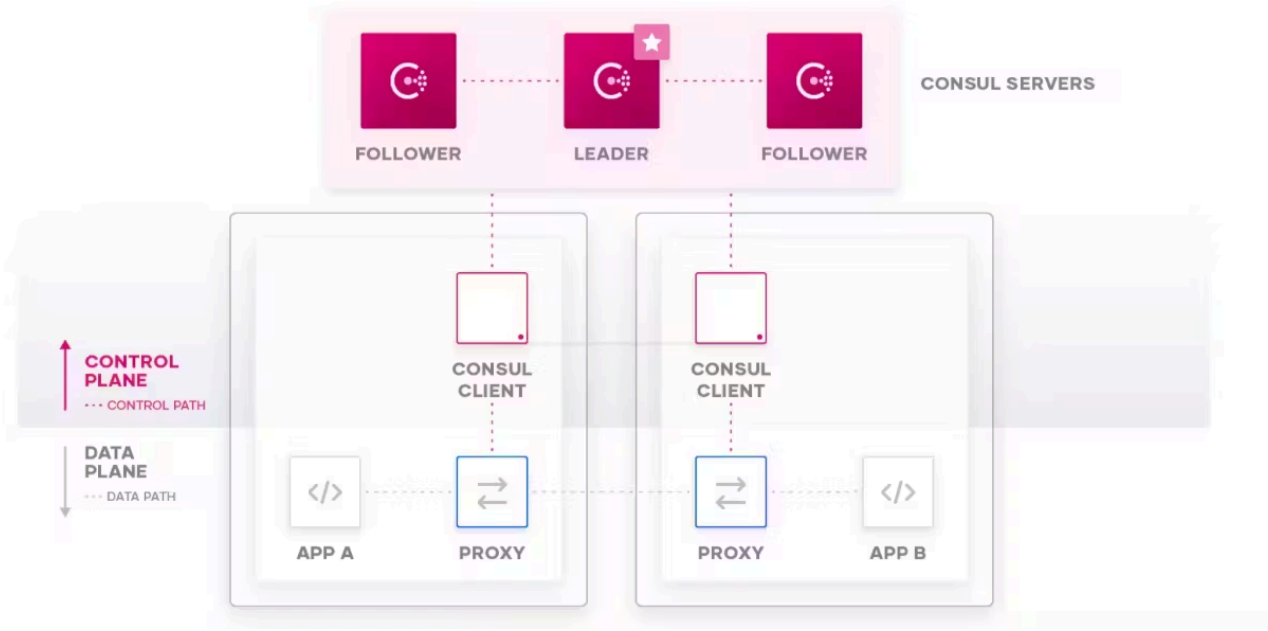
Service discovery

There are multiple ways to configure connection to remote services. Simplest one is the config file which provides an exhaustive list of connection information to all existing service providers. Such an approach has obvious limitations: such lists are static, duplicated in all consumers, so each time when a service instance is added or removed all copies of such configs must be updated manually.

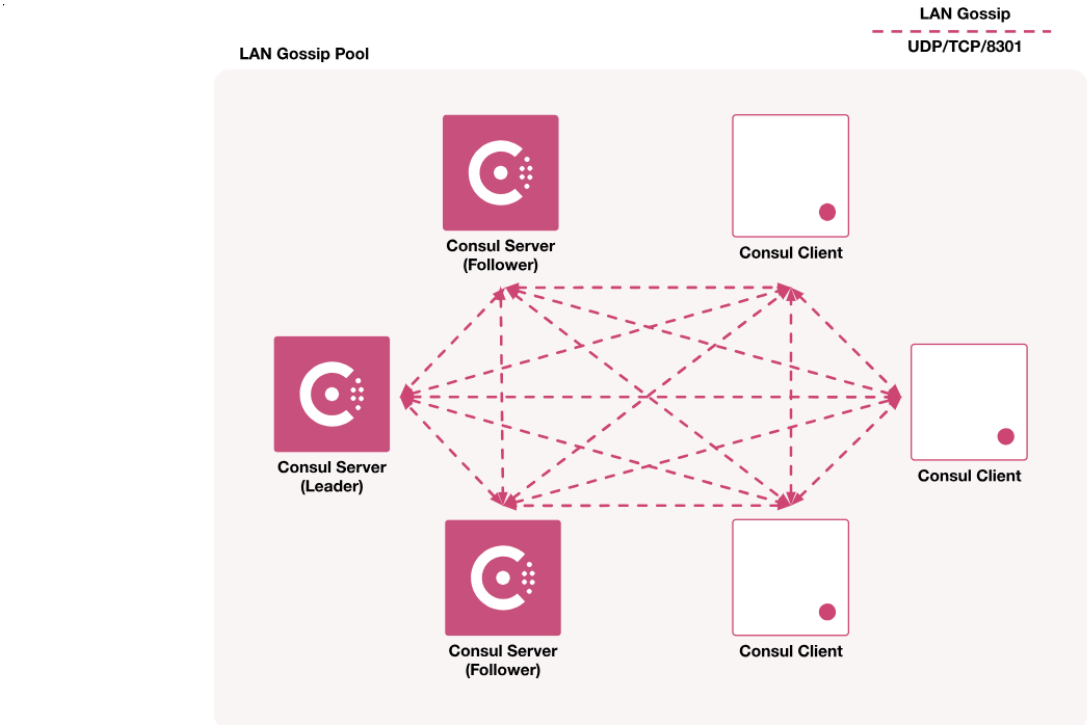
Standard way to solve the problem of dynamic nature of the environment and config updates is based on service registry mechanisms. Most known solutions in this area are: ZooKeeper, Eureka, Consul, Cloud Foundry and SaaS solutions provided by cloud providers.

One of the most interesting features of Consul (besides registry itself, KV storage and ready to use UI) is a service discovery (together with health checks) based on Serf Gossip protocol. This protocol is based on the absence of a central point of failure but instead maintains a cluster membership list with the help of UDP broadcasts. It detects failed nodes, propagates configurations within seconds based on a gossip algorithm where all nodes participate in this process. Such an approach has several benefits: lack of central point of failure, ability to handle false positives in case of network partitioning, automatic members discovery, in terms of service mesh consul provides a functionality of control plane (which makes migration to mesh much easier).

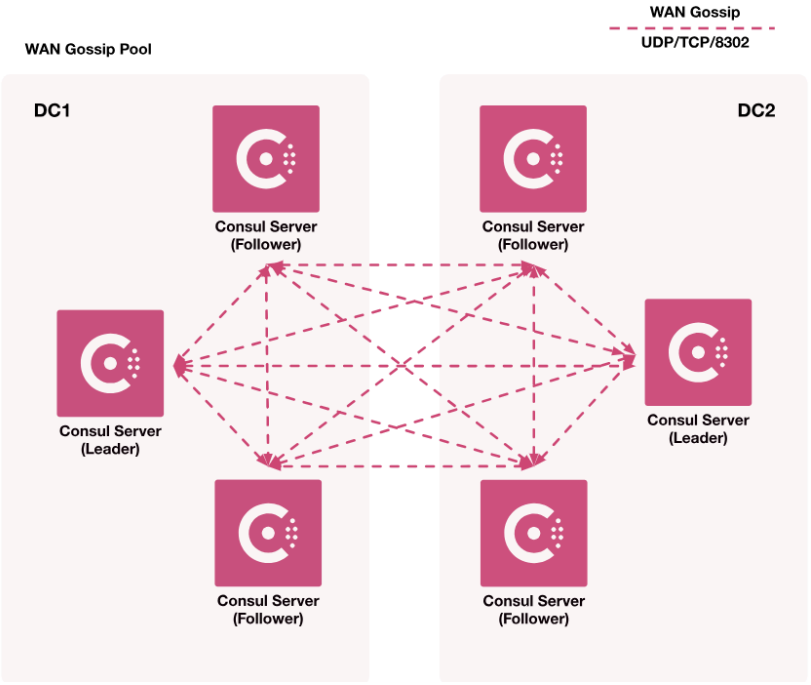
Next image demonstrates how can several participants organise cluster in a same datacenter with the help of Consul:



Nodes in a single LAN use gossip to propagate cluster information:



Consul can connect multiple datacenters to increase availability or event provide an ability of creating cross cloud providers deployments (gossip protocol is also used to exchange information):



Communication

Client-server communication

Access pattern of most of the operations the client will perform doesn't require persistent connection. Such a trait must be taken into account, because it significantly simplifies horizontal scaling and uniform resources utilisation (because of no connection pinning and additional logic related to connection context keeping). Industry standard for such scenarios is HTTP (HTTP/2). For future features which require persistent connection (in-app chats, video translations etc) other protocols can be used as additional: Websockets, WebRTC, QUIC, gRPC. This work is

related to only core functionality of the product and request-reply protocol (HTTP) covers current needs.

To achieve efficient usage of the network and protocol evolution without breaking existing native mobile clients a backward-forward compatibility must be guaranteed. For this purpose all request-replies must be based on Protobuf.

Server-server communication

Usually architectural style of the system affects communication mechanism: synchronous systems are based on synchronous protocols (for distributed system HTTP, synchronous RPC etc), while message-driven or event-driven utilise natively asynchronous channels (message brokers, streams). The goal of the current is to make parts of the product adjustable to specific needs instead of pinning the whole system to some pattern. At the same time flexibility and extensibility of communication infrastructure can greatly increase reusability, simplify and unify the way of how data is transferred from one endpoint to another being a building block.

As it was mentioned earlier, one of the architectural requirements defined for the solution is resources utilisation, throughput and cost efficiency. Solution of this task consists of several parts: transport protocol, serialisation mechanism, requests handling.

Network transport layer (for usual data transport) is represented by 2 main protocols: UDP and TCP. Main difference between these 2 ones is in guarantees of content delivery and order they provide. While UDP can guarantee only consistency of a single delivered (if it was delivered)

datagram with 64KB size, TCP provides reliable delivery of ordered segments stream. Such differences define the applicabilities of each protocol for data transferring. UDP is often used for voice or video transporting where loss of some part of information or delivering it in the wrong order is acceptable or in network games where commands do not exceed 64 KB. On the other side TCP is used when delivery guarantees are important and size of information passed over the network varies.

Next table summarises main differences between TCP and UDP:

Basis	**Transmission Control Protocol (TCP)**	**User Datagram Protocol (UDP)**
Type of Service	Connection-oriented protocol.	Datagram-oriented protocol.
Reliability	Reliable as it guarantees the delivery of data to the destination router.	The delivery of data to the destination cannot be guaranteed.
Error checking	TCP provides extensive error-checking mechanisms.	Basic error-checking mechanism using checksums.
Acknowledgment	An acknowledgment segment is present.	No acknowledgment segment.
Sequence	Packets arrive in order at the receiver.	There is no sequencing of data. If the order is required, it has to be managed by the application layer.
Speed	TCP is comparatively slower than UDP.	UDP is faster, simpler.

Retransmission	Retransmission of lost packets.	No retransmission of lost packets.
Weight	TCP is heavy-weight.	UDP is lightweight.
Handshaking Techniques	Uses handshakes such as SYN, ACK, SYN-ACK	Connectionless - no handshake
Broadcasting	Doesn't support.	Supports

As can be seen, TCP must be the primary transport protocol for communication on the backend side. It is worth mentioning that latency of request sending from one service to another can be decreased with the help of requests multiplexing, when connection is kept permanently, but not created on each request.

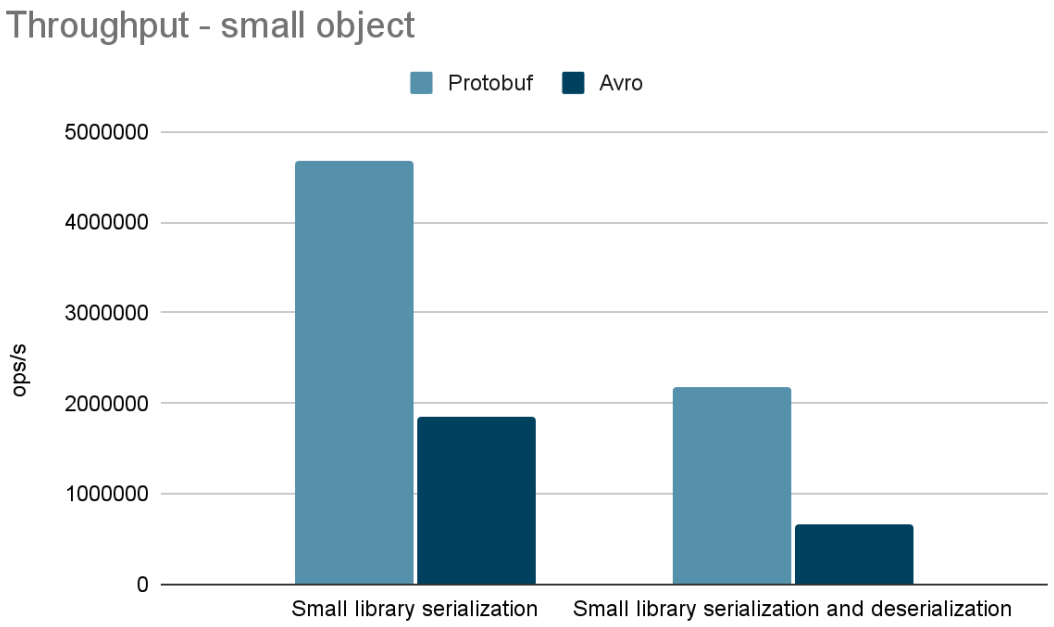
Payload serialisation is an important part which has a direct impact on serialisation/deserialisation time and amount of information transferred over network. Serialisation can be of different forms: plain text, structured text (XML, JSON), binary (Java serialisation), language and platform neutral binary protocols (Protobuf, Avro, Kryo), based on schema (Protobuf, Avro).

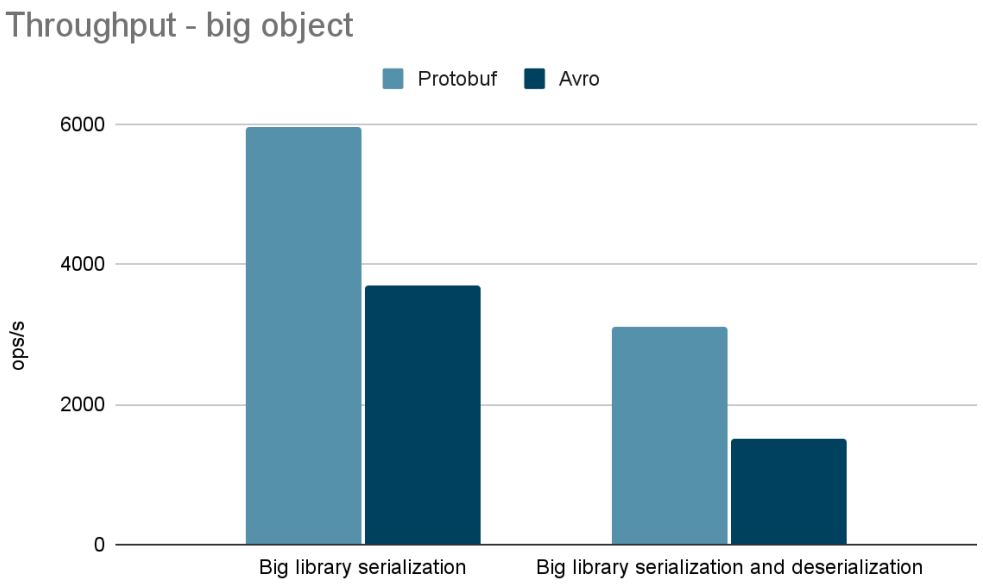
One of the acknowledged industry leaders is Protobuf. It has multiple important qualities:

- Code generation according to schema - supports portability (different platforms can communicate safely with the help of Protobuf);
- Null-safety;
- Backward and forward compatibility - allows save API evolution;

- Serialised data is much more compact compared to JSON or XML;
- Equal or faster serialisation/deserialisation compared to main competitors.

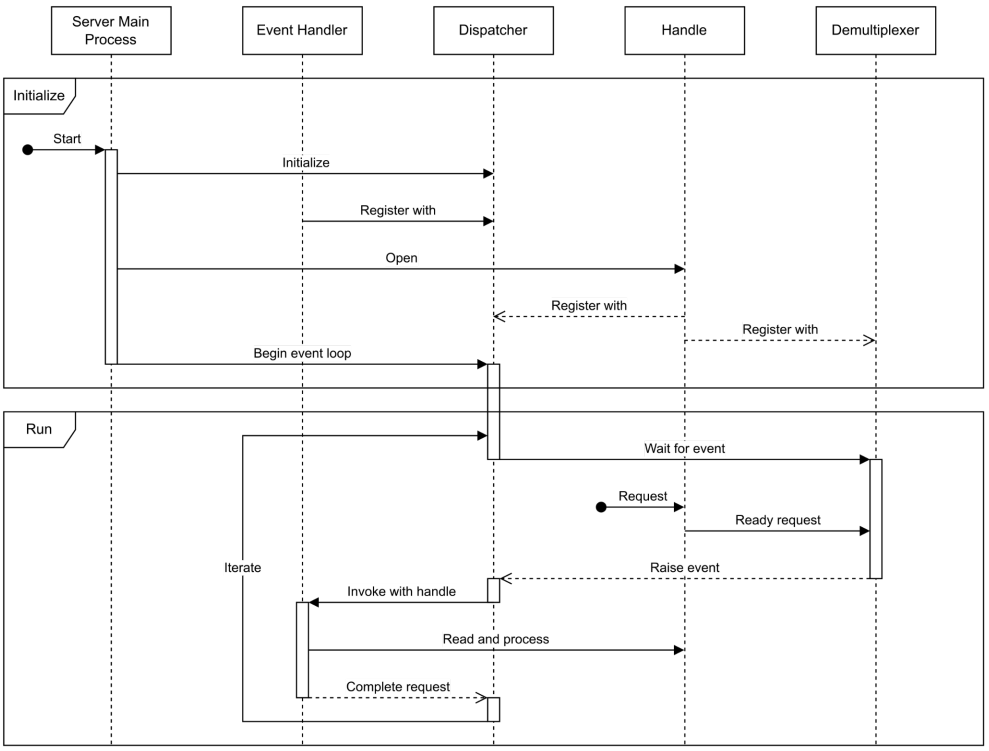
Next pictures demonstrate throughput Protobuf compared to another widely used serialisation mechanism:





There are 2 approaches to handle network calls: thread per request and event loop. Thread per request has one benefit - simplicity of implementation because of its imperative style. At the same time it is hard to achieve high utilisation of the environment and increase throughput of the system with this approach because of overheads related to resources allocated for multiple threads and context switches (unless cooperative multitasking like virtual threads is not being used). Event loop as an alternative, solves a problem of utilisation and throughput, performing the work with the help of a limited pool of threads (from one to the amount of available CPU cores).

Taking into account the asynchronous nature of network communication and OS support in a form of *poll* and *epoll* functions, event loop is a best choice for communication infrastructure in backends which can work under significant load and massive parallel requests. Next picture demonstrates idea of event loop on the example of Reactor pattern where network IO can be demultiplexed with the help of OS:



As a basis for building a communication layer it is preferable to use available, well tested and tuned frameworks. Fortunately there are multiple solid solutions implementing Reactor pattern, for example Netty.

Storage

One of architecture's significant requirements is making system cloud agnostic (with the help of creation DB details isolating layer of abstraction) and because of that this paragraph first of all analyses databases' types and selects applicable ones in the context of the product. A short list of databases currently available on the market which satisfy the needs of the product will be provided at the end.

Requirements to a database solutions can be represented with next items:

- well tested and acknowledged solution;
- horizontal scaling support;
- automatic failover support;
- different modes depending on environment where it runs (for example, indexes optimised for SSD);
- extended data durability guarantees (replication, error correction etc);
- high throughput;
- eventual consistency (some features may require real consistency models, for example repeatable reads or serialisation);
- availability guarantees (for cloud managed DBs) - at least 99.99.

Database types

Relational databases - databases based on relational model. Systems used to maintain and operate relational data are called RDBMS (relational database management system). Most popular RDBMS apply ACID transaction and configurable transaction isolation level. RDBMS provides a strong consistency model, rich query language and data integrity. Most popular RDBMS systems are: Oracle DB, MySQL, PostgreSQL etc. Besides obvious benefits, common RDBMS has several drawbacks: it is hard to scale them horizontally, ACID guarantees and relational model support require additional resources compared to weaker models and negatively impacts a performance of a system in which they are not needed. Such databases are suitable in cases when consistency, data structuring, queries or transactions are required (financial, healthcare etc).

NoSQL databases - this is a category of databases which do not use relational model for data storage and processing. Such databases have weaker consistency guarantees in return higher throughput, lower latencies and native support of horizontal salability. NoSQL databases are represented with the next types: key-value, document, column-family, graph, time series and index databases.

Key-value databases - persistent or in-memory databases. Access to values is narrowed down to access by a key or a range of keys. Key-value databases are highly scalable and can handle high volumes of traffic, making them ideal for session management, caching or any functionality which doesn’t require queries. Examples: Memcached, Redis, DynamoDB.

Document databases - such databases can be seen as a subclass of key-value storage. Document databases store all information related to an object as a single entry and every stored object can be different from every other. Document-oriented systems rely on the internal structure of the document for extracting metadata for different optimisation and indexing particular fields. Examples: Couchbase, MongoDB, Firestore.

Column-family databases - in column family database each key is associated with multiple isolated groups of columns, each of them representing related collection of information. Strong side of this approach is that each group can be accessed independently (compared to a whole object in document databases) which greatly optimises read operations (from a perspective of network, memory and disk drive IO). Examples: BigTable, Cassandra.

Graph database - a key concept of such a database is the graph, (edges and relationships). The graph relates the data items in the store to a collection of nodes and edges, the edges representing the relationships between the nodes. Such databases help to model real life connections, for example network graph. Examples: Neo4j, Apollo.

Time series database - a time series database is a software system that is optimised for storing and serving time series through associated pairs of time and value. Examples: Prometheus, InfluxDB, TimeScaleDB.

Index databases - this type of database is similar to document DB with a prominent difference: each key is part of the associated

document's value. There are multiple types of indexes: reverse, geospatial etc. Examples: Elastic search, Solr.

DB solution selection

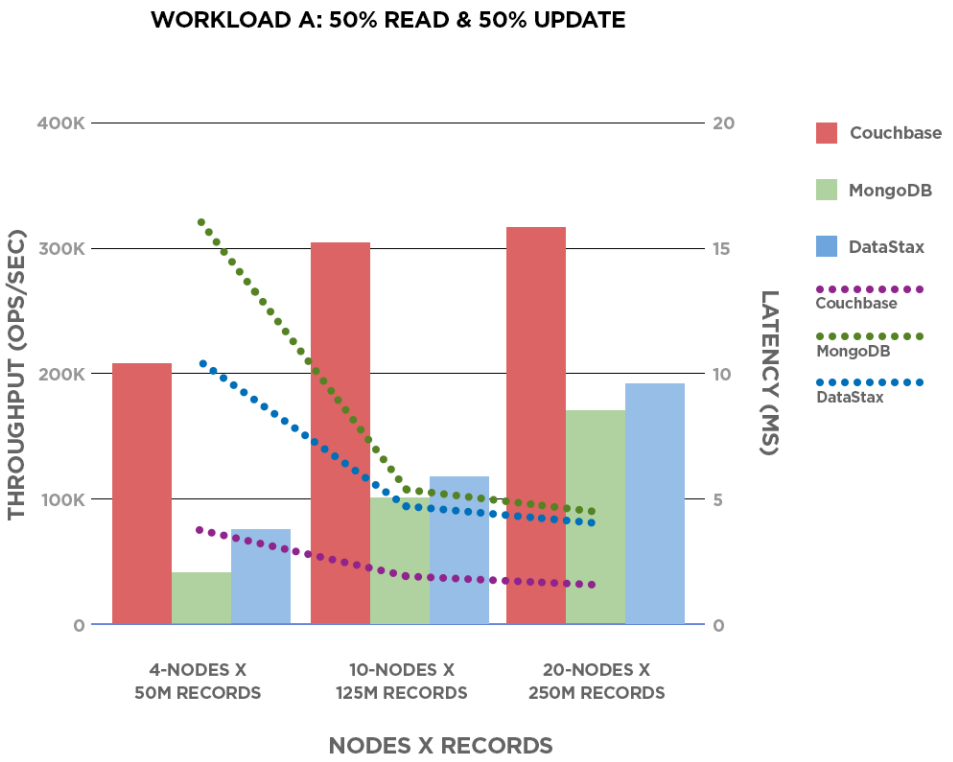
The system will consist of domains residing in separate databases:

- Authentication;
- Users;
- Notifications;
- Events;
- Geo events index.

Taking into account necessary system qualities of the product, DBs (and other parts) must be horizontally scalable. DB solutions must demonstrate high throughput, configurable replication factor and failover. At the same time basic functionality does not require complex joins or strong consistency models. RDBMS is not the best choice of storage for such types of values because its main features cannot be utilised to the fullest. Time services and graph databases are suitable for a specific range of problems, column family databases fit well when datasets are huge and values significant in size.

Document database type is the best choice for next features: authentication, users, notification and events. There are multiple solutions available on the market, both cloud managed. For example GCP Datastore can ramp up to support 750K of operations per second. Datastore supports multi-zone and multi-region configurations and 99.999 availability. Another solution with rich functionality, automatic failover, replication, query language (allows to select documents based

on some property) and management admin panel is Couchabe. There 2 options of Couchbase management: self-managed and cloud managed setups. Next picture demonstrates CB performance against other famous DBs:



Index of events related geo information differs from other product related information stored in databases. The only values that are being stored there are events' (at this moment) identifiers. Keys for such value are geo coordinates. Main modern databases have geospatial indexes in their inventories, for example: Firestore, Couchbase, Redis, Snowflake, Aurora etc. At the same time such databases are not full-fledged indexing engines. Mentioned databases solve the problem of the initial functional requirements, but are not the best choice for further development and extensions of functionality, where multilanguage

support, suggestions, error correction, synonyms usage and autocomplete are required. Such features can be applied in search events by information (name, description), participants or host which makes events finding process more effective improving user experience.

Main solutions in this domain are Elastic search and Solr. These databases are similar in possibilities they provide and compete with each other. Currently, each of them is suitable for the initial feature set of the product.

Conclusions

This work is dedicated to the creation of new product destined to wide audience purpose of which is helping fulfil natural human need in communication, collaboration, making network and learning new things.

During this work next tasks were accomplished:

- functional requirements gathering;
- system requirements formulation;
- high-level architecture defining and description;
- possible communication mechanisms evaluation and selection most suitable one at current moment;
- analysis of available DBMS solutions and selecting suitable database according to functional and nonfunctional requirements.

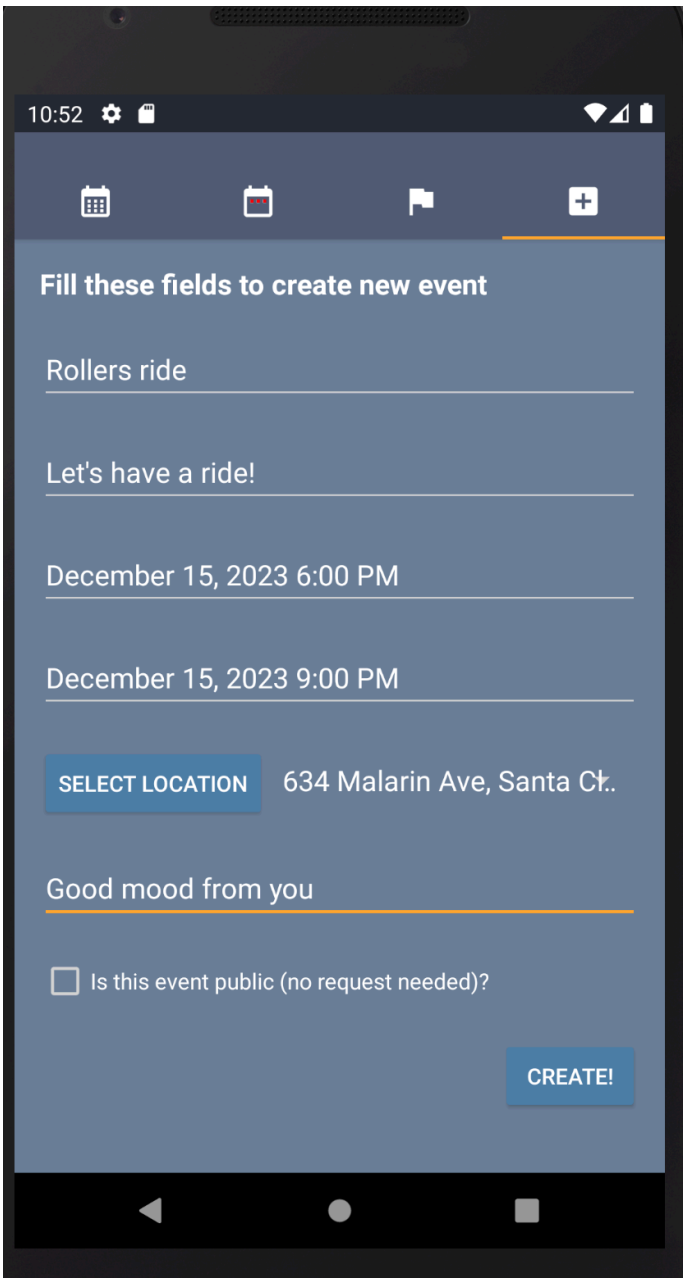
Proof of concept is created as the practical result of the work.

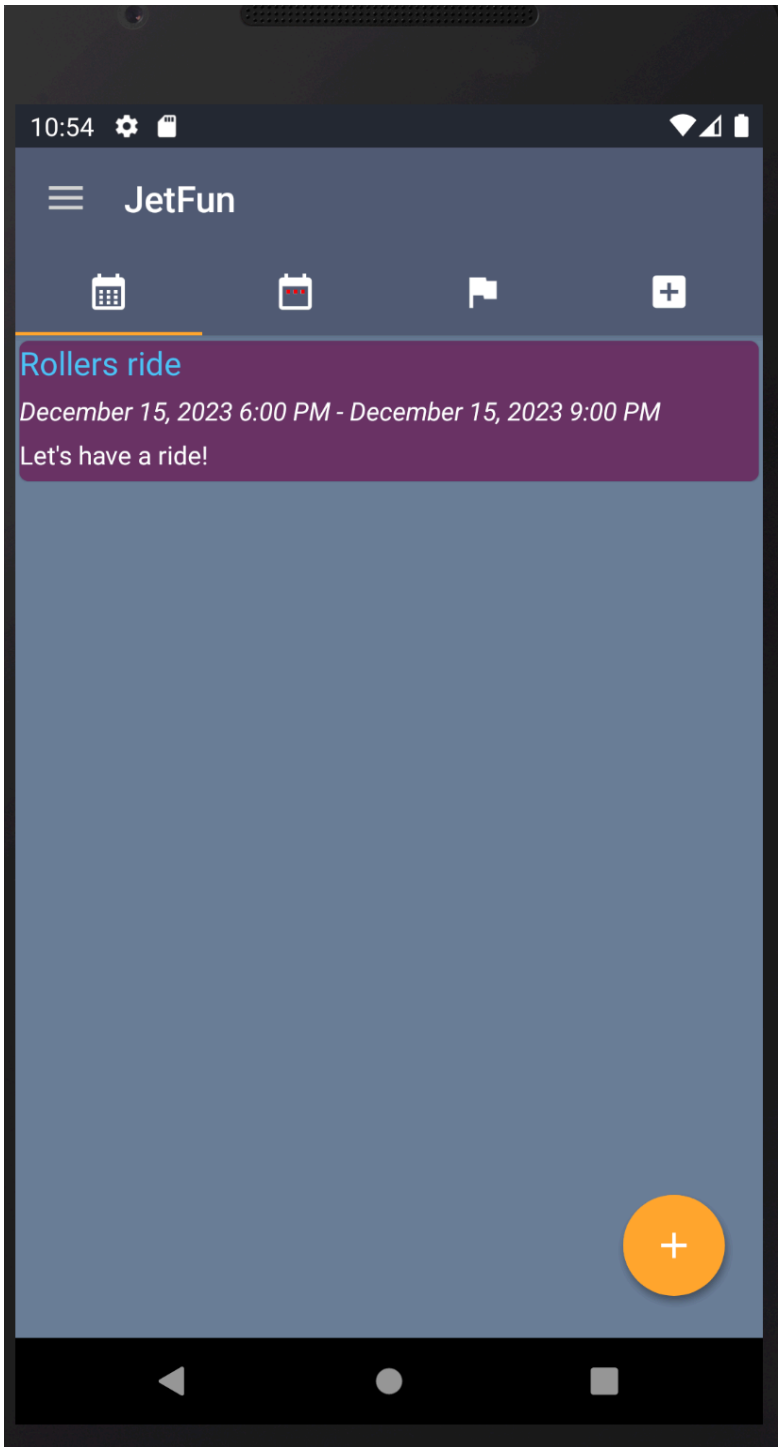
Used sources

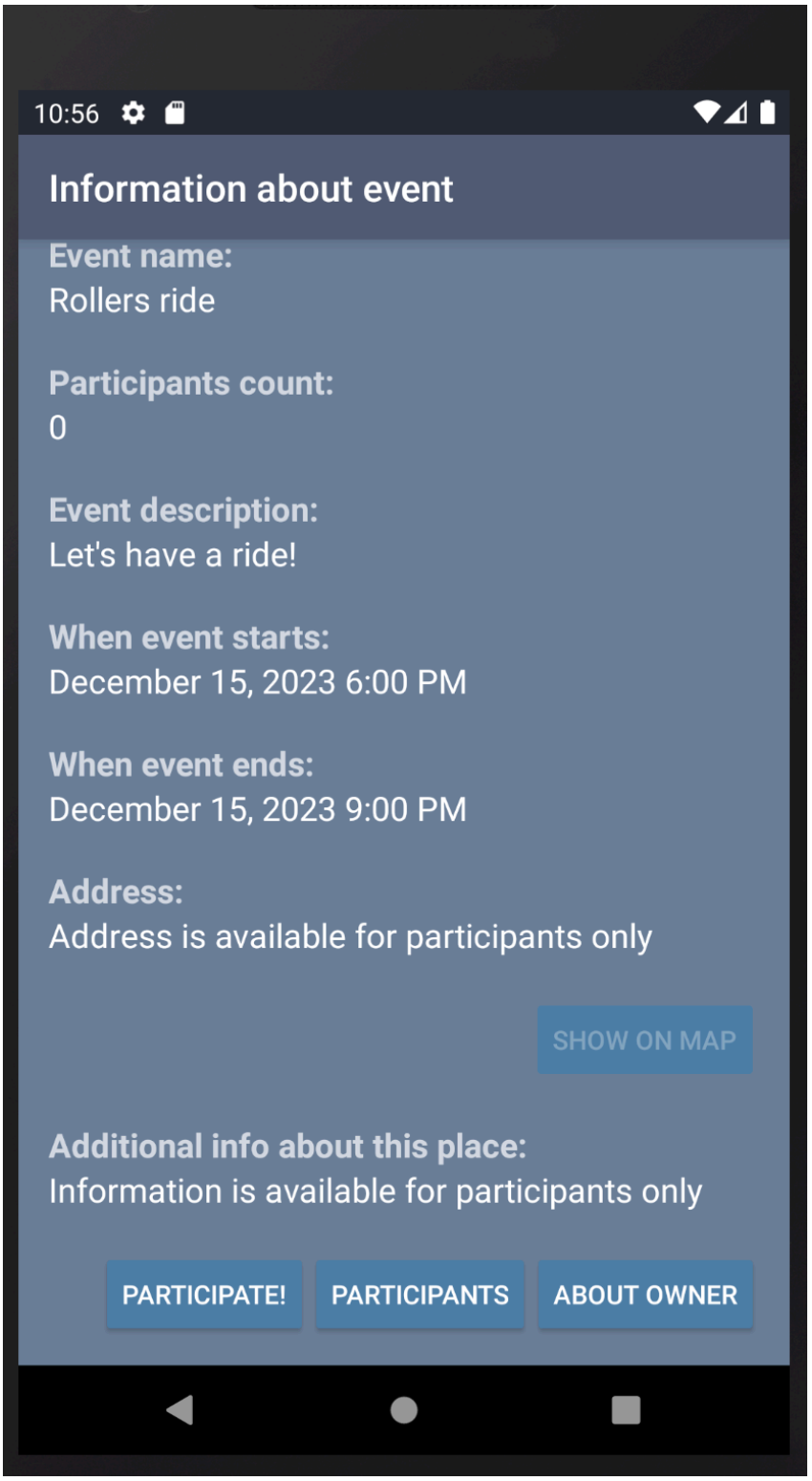
1. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems 1st Edition, Martin Kleppmann.
2. Pattern-Oriented Software Architecture, Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal.
3. TCP/IP Illustrated, W. Richard Stevens, Kevin R. Falls, Gary R. Wright
4. https://www.matillion.com/blog/the-types-of-databases-with-examples
5. https://developer.hashicorp.com/consul/docs

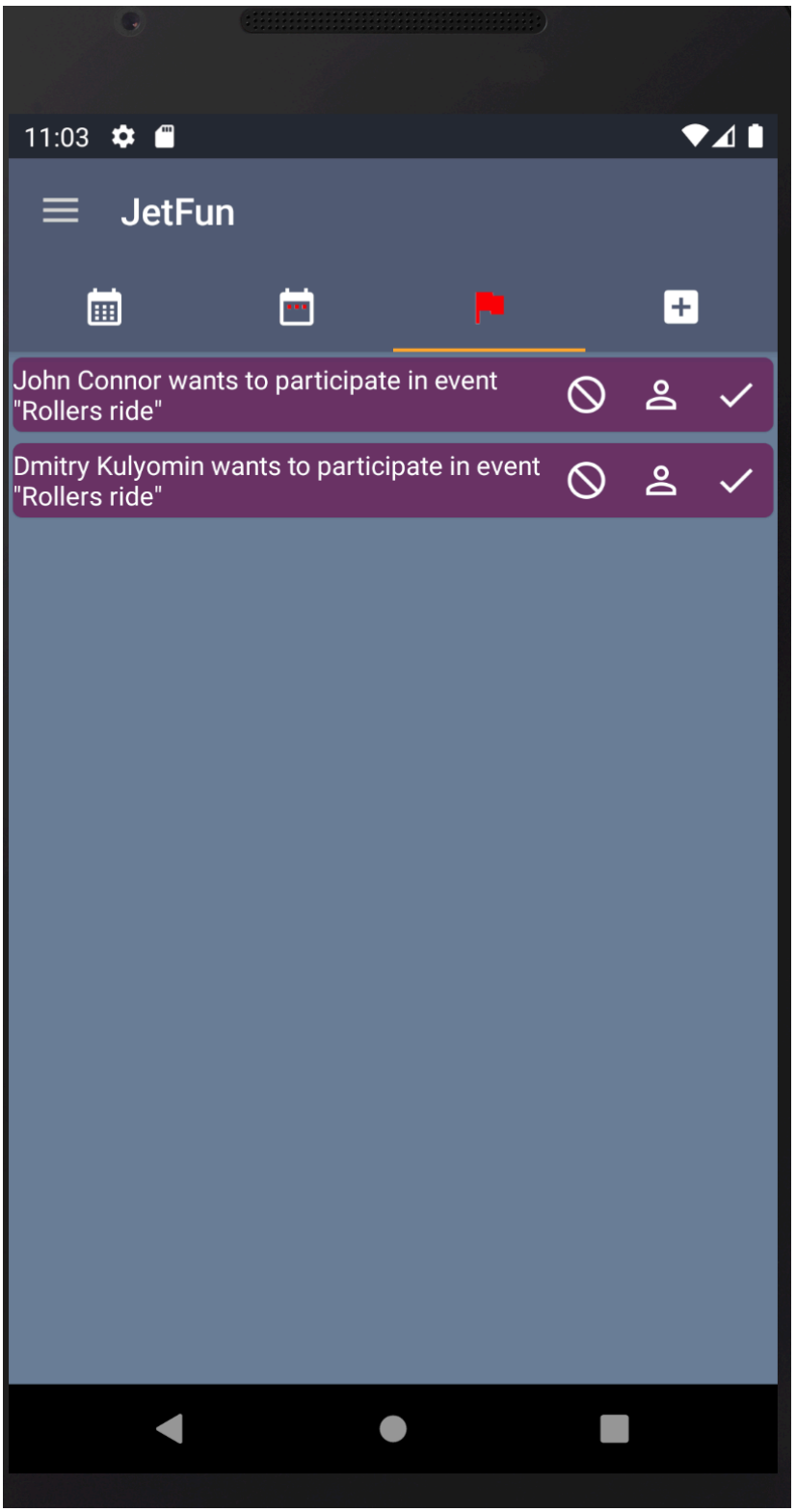
Appendices

Application screens




10:54
JetFun
Rollers ride
December 15, 2023 6:00 PM - December 15, 2023 9:00 PM
Let's have a ride!


10:56
Information about event
Event name:
Rollers ride
Participants count:
0
Event description:
Let's have a ride!
When event starts:
December 15, 2023 6:00 PM
When event ends:
December 15, 2023 9:00 PM
Address:
Address is available for participants only
SHOW ON MAP
Additional info about this place:
Information is available for participants only
PARTICIPATE!
PARTICIPANTS
ABOUT OWNER


11:03
JetFun
John Connor wants to participate in event "Rollers ride"
Dmitry Kulyomin wants to participate in event "Rollers ride"