International Journal of Systems and Service-Oriented Engineering
Volume 11 • Issue 2 • July-December 2021

ServiceNet:
A Service Network for Peer to Peer

Ji Liu, University of Sydney, Australia
Shiping Chen, CSIRO Data61, Australia
https://orcid.org/0000-0002-4603-0024
Hang Zhao, University of Sydney, Australia
Jiyuan Yang, University of Sydney, Australia
Yu Shi, University of Sydney, Australia
Ruiqiang Li, University of Sydney, Australia
Dong Yuan, University of Sydney, Australia

ABSTRACT

Despite the enormous number of online docking services available, consumers sometimes struggle to discover the services they require from time to time. On the other hand, when finding matching or recommendation platforms from an academic or industry perspective, most of the related work they can find is centralized systems. Unfortunately, the centralized systems often have shortages, such as adv-driven, lack of trust, non-transparency, and unfairness. The authors propose a peer-to-peer (P2P) service network for service discovery and recommendation. ServiceNet is a blockchain-based service ecosystem that promises to provide an open, transparent, self-growing, and self-managing service environment. The article will provide the basic concept, the proto-architecture type's design, and the proto-initial type's implementation and performance assessment.

KEYWORDS

Blockchain, P2P, Service Discovery, Service Recommendation

1. INTRODUCTION

With the rapid development and evolution of social society and technology innovation, more detailed industry segments have been categorized, such as family wealth management, family health care, personal fitness instructors, etc. Besides, both the service demander and the provider tend to be personalized and individualized, instead of blindly pursuing the traditional unified form. Therefore, a flexible, diverse, and fair service docking platform that matches it is needed. This kind of docking platform is most suitable for service-related segments in current world. Furthermore, according to eMarketer Report 2019 for the global internet trading market (eMarketer, 2020), the order number of service-related segments will increase 20% each year, and the order number for 2019 is about 2

DOI: 10.4018/IJSSOE.20210701.oa1

billion total amount of those orders is around 200 billion dollars. There is a rigid demand for a flexible service docking platform with a huge market capacity and opportunity.

Traditional service docking platforms are mostly centralized internet platforms. The owners of these platforms are in charge of all communication and data of both the service requesters and the service providers, so it has asymmetric advantages and rights, which makes it challenging to guarantee: (1) fairness of docking; (2) protection of personal information; (3) and reasonable charges. This leads to mistrust among many parties and affects the efficiency of the entire ecosystem. Moreover, the personalized recommendation mechanism is not intelligent enough to service current personalized orders for both demanders and providers. Hence, a decentralized service docking platform with more intelligence recommendation mechanism features is urgent for service-related segments.

This paper proposes a peer-to-peer (P2P) service network for service discovery and service recommendation, called *ServiceNet*. ServiceNet is inspired by blockchain technology and has the characteristics of decentralization, security, justice, privacy protection, and uses peer-to-peer technology to construct a decentralized service docking network platform to ensure fair docking, privacy protection of data and entities. In addition, ServiceNet leverages some ordinary senses and existing service recommendation techniques to avoid unnecessary intermediate links and services for efficiency, saving bandwidth and minimizing interruption to service providers. Our goal is to create an open, fair, transparent, and intelligent P2P network that truly protects the individual's privacy, and provides a reliable, efficient, and win-win service docking ecosystem.

The rest of the paper is organized as follows. Section II describes our design principles and the architecture design based on the principles. Section III talks about the implementation of the design as a proof of concept (PoC) prototype. Section IV conducts some performance tests and analyses. We present and discuss some related work in Section V. And we conclude in Section VI.

2. SERVICENET ARCHITECTURE DESIGN

We envision a simple motivation example in the future to better present our basic idea and design principles for ServiceNet:

Alice and Bob use their spare time to provide proofreading services in Sydney and London, respectively. They register their services with ServiceNet as a service provider. Alex and Tom are both self-employed plumbers in Sydney and London, and registered their services with ServiceNet as a plumber service provider.
[Use Case 1] Merry is a PhD student in Sydney. She has completed her PhD thesis, and she would like to have her thesis checked before submitting it. She sends a thesis proofreading service request to ServiceNet. Since proofreading service can be conducted remotely, both Alice and Bob receive the request, and reply with their quotations. Since Bo's price is a little bit cheaper with a lot of favorable comments, Merry picks up Bob to do the proofreading for her thesis.
[Use Case 2] Chen's shower starts leaking. He sends the request to ServiceNet. Since the plumber needs to come to Chen's house to fix the shower, ServiceNet only forwards the request to Tom (and maybe a few other plumbers near Chen's home). Through a similar bidding process, Tom secures the deal.

2.1 Design Principles

Based on the above visionary example, we derive our design principles as follows:

- ServiceNet is a P2P network with no centralized servers and business entity, which control ServiceNet.
- It is (almost) intelligent, because of no HR and operational cost.

- It should be smart and fair enough to routine a service request to the trimmer and suitable service providers, who is likely the best to deliver the service.
- It is able to grow and manage by itself like nature and human society.

2.2 ServiceNet Architecture

To follow the above design principle, we design the high-level architecture of ServiceNeT as shown in Fig. 1.

As shown in Fig. 1, ServiceNeT consists of a number of nodes (or called peers), which can be deployed onto different continentals and are fully connected via the Internet. Each peer has similar local data and the same functionality, i.e., routine the service requests to the other nodes if required. Clients, either service providers or requesters, can send and receive service-related messages using laptops, iPads, and smartphones via any peer nodes. We provide a detailed design for the serviceNet node in the following subsection.

2.3 ServiceNet Peer Architecture Design

Each ServiceNet peer node contains at least three components to realize the system at our initial design: (a) peer management; (b) Pub/Sub messaging service; and (c) P2P connection service, i.e., the ICE server. While it is feasible to integrate these three components into a single server, we designed them separately for flexible configuration and deployment in the future. Fig. 2(a) shows the components of the ServiceNet Peer.

The ICE server is responsible for checking combinations of candidates and establishing a P2P connection between peers. On the other hand, the message transmitting server is a message broker in the Pub/Sub process. It should accept subscription requests from subscribers and route data

Figure 1. Overall architecture of ServiceNet

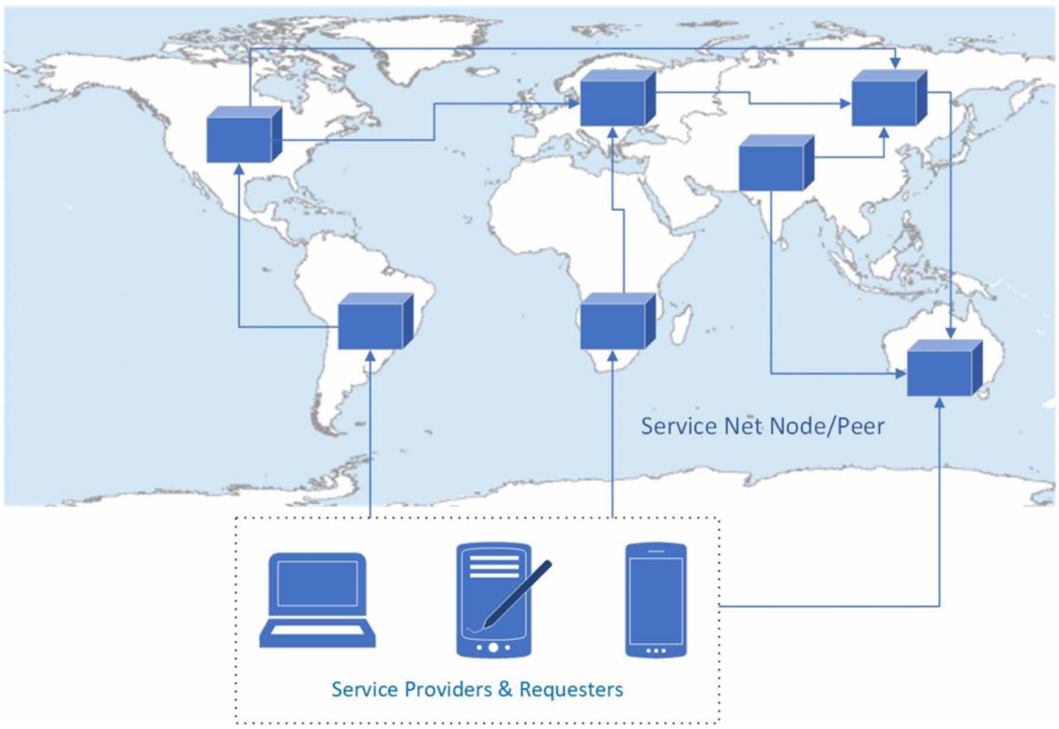
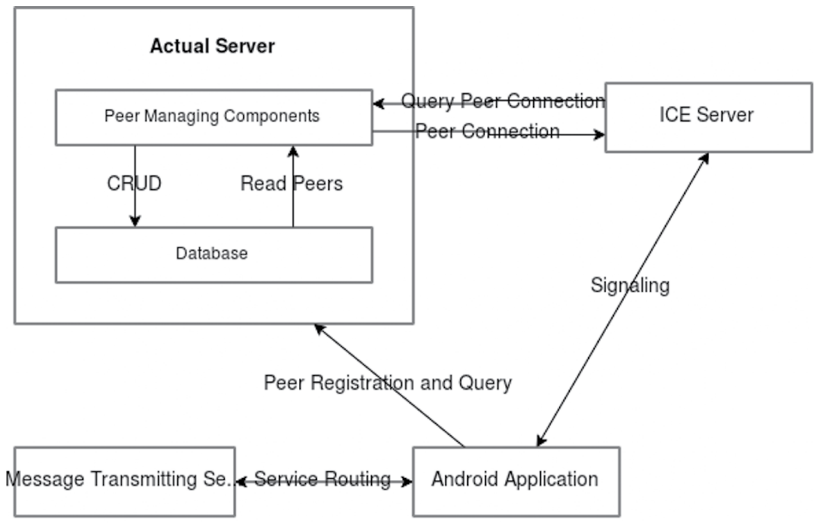
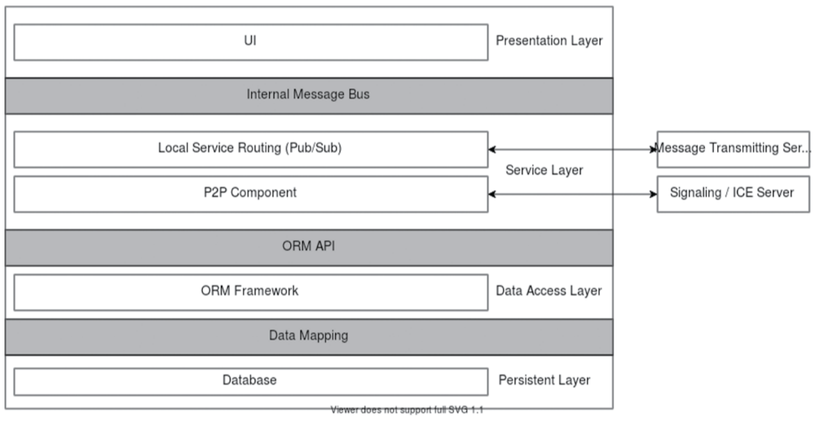


Figure 2. Architectures of ServiceNet Node and Client Model App



(a) ServiceNet Node



(b) Client Mobile App

according to the topic when the publishers publish messages. Since the ICE server and the message transmitting server can be complex to be designed and deployed from a fresh start, we should take advantage of matured technologies. Except for the three key attributes, the LTS (Long-Term Support) of technologies chosen should be considered.

The actual server is responsible for peer management and WebRTC intermediary signaling support. In terms of peer management, it should be in charge of peer registration and keep an instance of peer connection. Though the management of peers requires help from persistent data saved in the database, the information stored should be minimized. Thus, the concern of information asymmetry can be mitigated. Besides, it should cooperate with the ICE server to finish the candidate's check between peers using the connections it keeps. The initial signaling phase can use the connection channel provided by the actual server to enable meta-data exchanging. The overall design of the application layers is following the MVP (Model-View-Presenter) pattern. Components are designed into modules to ensure high scalability and modularity, as shown in Fig. 2(b).

2.3.1 Design of Persistent Layer and Data Access Layer

Due to the decentralization nature of the system, the storage load is shifted from the server to peers. All peer are supposed to store their information and share with others when needed. Besides, synchronization is required for mutual data like chat messages between peers. In terms of the type of the database, we prefer the classical relational database to enable ORM (Object Relational Mapping). An ERD (Entity Relationship Diagram) in Fig. 3 is used to show the database design.

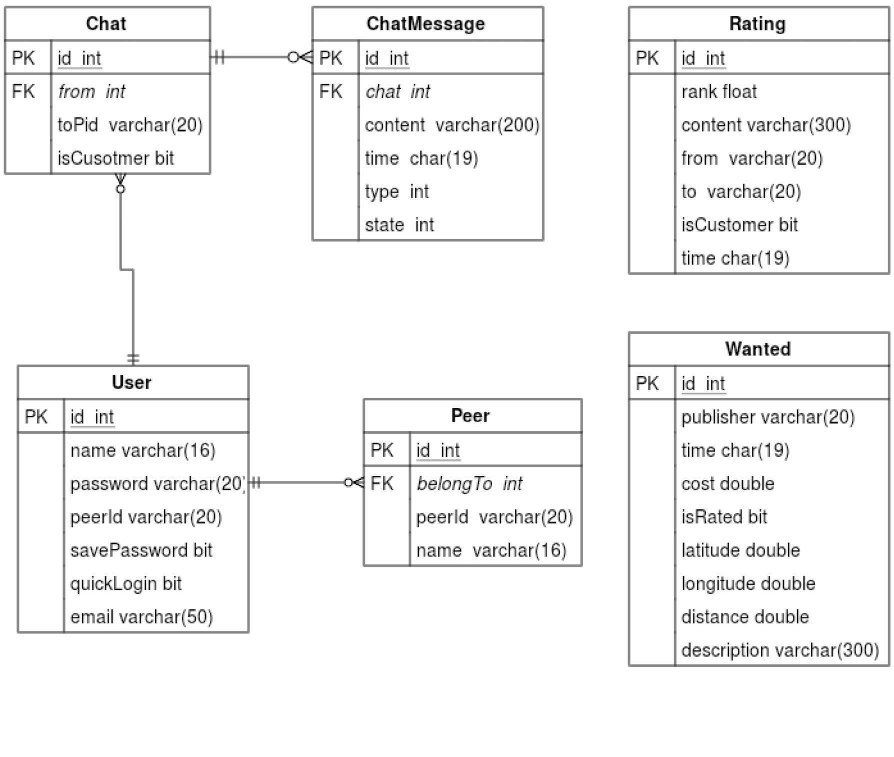
PeerId value is used for peer queries in the actual server. This means the querying using such value can be carried on without an actual peer record in the local database. Thus, peerId is not considered as a foreign key in the database. This leads to the disperse of the two tables, Rating and Wanted. However, a potential relationship does exist among the Peer, the Rating and the Wanted tables.

On the other hand, a data access layer is designed to enable access to the database using OO (Object-Oriented) languages. ORM framework is used to map the relational database into OO objects, thus avoiding raw SQL queries. Proper and well-supported ORM framework should be selected to realize the layer. The ORM should also expose pre-defined interfaces to regulate operations to the database.

2.3.2 Service Layer

The service layer is consisting of the P2P connection component and the service routing component. The P2P connection component is mainly responsible for maintaining a connection with the actual server and assisting with establishing a P2P connection with others. There is a coupling between the actual server and the ICE server due to the signaling service. Thus, the design of client-side also integrates the functions. Meanwhile, publishing and subscribing messages are handled by the local

Figure 3 Application Database ERD



service routing component. It is the endpoint in the service routing network. It should contain user-defined filters as a local message gateway. Additionally, the two components are supposed to work together to act as the presenter. A response to the user request will be created by querying the local database and exchanging data with the server. The pattern seems similar to the traditional centralized business system, but the difference is that data storage and business logic are injected into the peers. That is to say: the server only transmits messages, the peers handle the actual work.

2.3.3 Presentation Layer

The single UI component can be simple to design but complex to implement. Key attributes considered at this stage should be extensibility because we are not focusing on UI design at this stage. The UI should be modulized thus can be easily extended to increase user acceptability in the future. Embedded Ui framework could be considered to fulfill the requirement.

Our system design partially addresses the scalability issue from the following two angles: (a) Each node/peer will recommend the local service providers first according to the nature of the services (e.g. face-to-face services). As a result, the kind of requests will not be broadcasted further to the rest of the network and thus can improve the performance and scalability; (b) our system has inherent better performance and scalability by using asynchronous messaging for communications between peers (i.e., non-blocking sending and receiving by notification).

Another issue is unfair/malicious user behavior that a service provider itself or its relatives/ friends provide fake comments/high scores. This is a common issue for all service ecosystems, and some current work addresses the issue (Lau et al., 2012). Due to space, we are not going to address the issue in this paper.

3. IMPLEMENTATION OF SERVICENET PROTOTYPE

3.1 ServiceNode Implementation

ICE, NATS Messaging and the actual server can be standalone though we integrated the three servers together. Thus, if such integration becomes a bottleneck, they can be separated into clusters to disperse the load. The former two functional servers are deployed using existing server library NATS Server and Coturn, respectively. These libraries expose simple set-up configurations, thus can provide high portability and maintainability. Such matured and well-developed applications also provide high availability and reliability.

The actual server is deployed using Node.js. The service layer exposes an API for peer management and signaling assistance. The peer management function involves peer registry, storage, and etc., and is mainly handled in the SocketHandler.js. When a new peer is trying to sign up, the application will generate a UUID (Universally Unique IDentifier) and submit it together with the peer's email and name. The UUID is generated using complex data, including time and device identifiers etc., by Android provided API. This enables the binding of peer account and the device. The server will detect alternation of devices to protect the account. The server will then generate another two ids called PId (Permanent Id) and TId (Temporary Id) in the service layer. PIds are short, human-readable and unique ids that can identify a peer. They will be generated during peer registry only once. TIds are ids generated by Socket.io to distinguish clients in the session. They are generated each time when the peer comes online. Both ids will be returned to the client via socket. Registered peers can use email or PId to log in anywhere from now on. There is also a map containing peer PId and peer connection instances within the SocketHandler.js for peer management and querying. Peers exchange meta-data in the signaling process with these connection instances. The server can be considered as a yellow page system integrated with the signaling function. Peers can fetch connect information from it then establish a P2P connection with its assistance. In terms of storage, only basic information of peers is stored on the server. The information includes peer PId, nickname, email and UUID. The Constraints.js is created to ease the management of constants like the database connection configuration.

Since we identify availability and scalability as key attributes for the whole system, we postpone error handling to the client instead of in the server. If an error emerges during a client request, the error will be reported back to the client for further action. Scalability is achieved by separating logic. Key logic is being coded into separated.js files and functions are exposed via export keyword. Thus, modules can be changed and rewrite without interfering with encapsulation. On the other hand, though the server is written with Node.js, all functions can be easily rewritten to other languages as there is no complex business logic within it and substitution of key libraries can be found in other languages.

3.2 Mobile App Implementation

We implemented the client mobile App on Android. Fig. 4 below illustrates the implementation of the Android Mobile App.

Conforming to the design, the data access layer is designed to assist with accessing the persistent storage of the Android SQLite database by adopting ORM. It is composed of entity classes, greenDAO generated codes and model managers. Entities are POJOs (Plain Old Java Objects) or Java beans that only contain getters and setters. Each of the classes is mapped to a table in the relational database by greenDAO. Additionally, to be able to be transformed into byte streams, they implement the Parcelable interface. By overwriting the abstract methods, instances of the object can be transformed into Parcels. Parcels can be further processed into byte streams. Thus, the instances can be transported through the Internet or Android Bundles. On the other hand, to store unsupported database objects, converters are created. The converter is classes that override specific generic methods defined in the Property Converter interface, thus translating objects into database-supported types. For example, Date objects are transformed into strings to enable storage into the database. The StringDateTransformer sample code is shown below Fig. 5 (a)

The greenDAO module consists of a Database instance, DaoMaster, DaoSession, and entity DAOs (Data Access Objects) generated by the greenDAO library. It serves as the ORM API that maps the entities into tables. Connection to Android SQLite database is initiated in the database instance and maintained in the DaoMaster for future access. Transactions and object identity scope are managed in

Figure 4. Implementation of the Application Layers

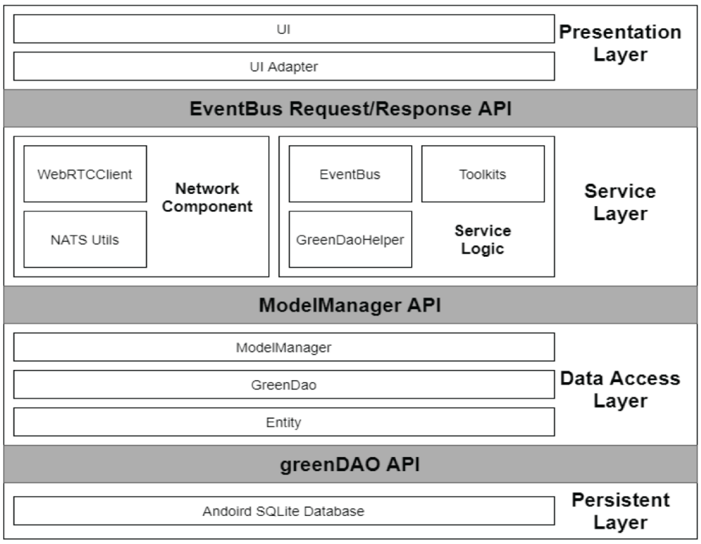
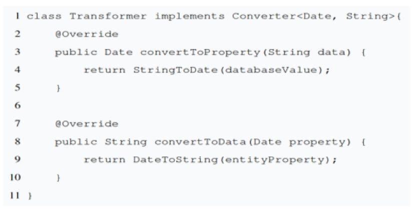


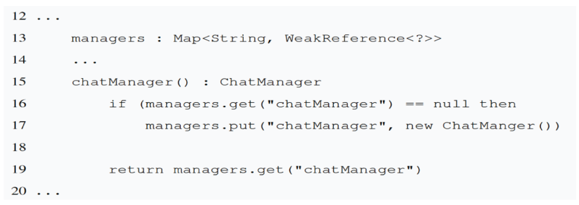
Figure 5. Snapshot of the mobile App implementation



```
class Transformer implements Converter<Date, String>{
    @Override
    public Date convertToProperty(String data) {
        return StringToDate(databaseValue);
    }

    @Override
    public String convertToData(Date property) {
        return DateToString(entityProperty);
    }
}
```

(a)



```
...
    managers : Map<String, WeakReference<?>>
    ...
    chatManager() : ChatManager
        if (managers.get("chatManager") == null then
            managers.put("chatManager", new ChatManger())

        return managers.get("chatManager")
...
```

(b)

DaoSession. In other words, DaoSession defines Daos and uses them to query the desired result from the database. The session can also define whether caches of results are used, that is to say, whether all queries return the reference to the same object. Model managers are encapsulations of entity DAO objects. They expose supported operations to the entities via interfaces. It is the API that the service layer uses to negotiate with the data access layer. All managers should implement a corresponding interface and are managed in the ModelManagerFactory. This fosters the modularity and scalability of the module. The factory is a singleton and keeps a map of managers' name and their instances. The instances are created in the form of weak references. This will save RAM and avoid duplicate creation of instances in a short period. The pseudo-code is shown in Fig. 5 (b), and the client mobile App UI is shown in Fig. 6(a).

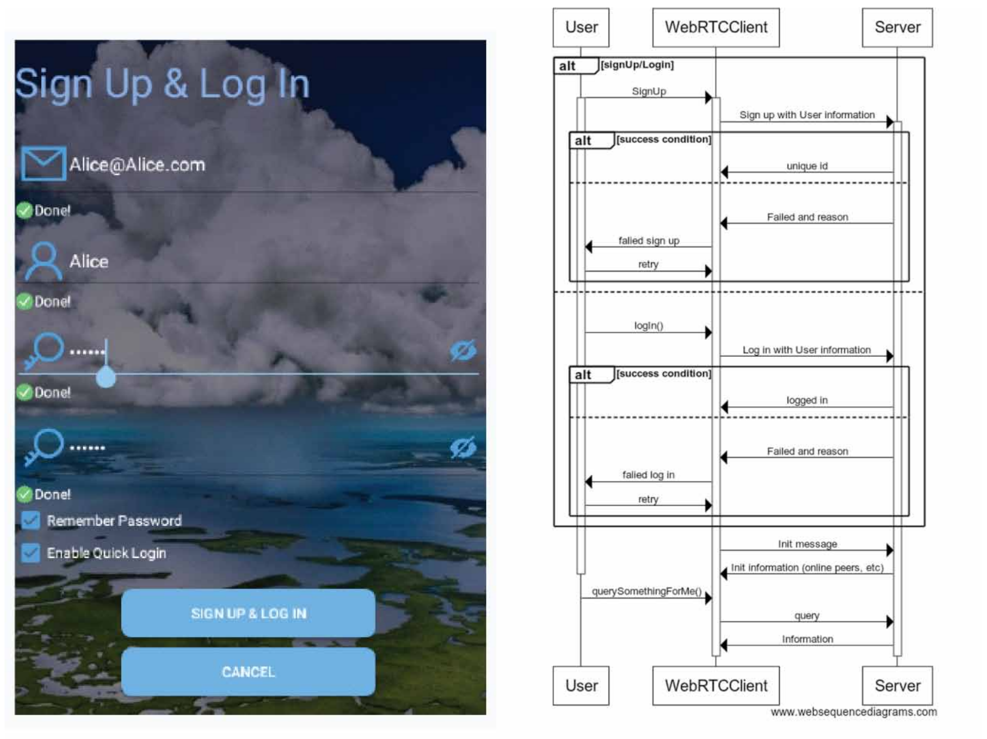
3.3 P2P Connection Implementation

The layer mainly consists of two network components and multiple internal service logic components. The WebRTCClient module is a daemon service running at the backend since the user logged in. It is used to establish and maintain the connection with the actual server and handle P2P connection-related operations. It uses EventBus message bus to communicate with other components. In detail, when the user is trying to sign up or log in, the service will be started and try to connect with the server. Then after several message exchanging, it will keep a stable connection with the server. It also supports several query functions which can fetch data from the server. EventBus events are exposed as interfaces for these functions. The sequence diagram in Fig. 6(b) below can show the processes.

Socket.io is used for establishing a connection with the server. We referenced git repository from Naoyuki Kanezawa for Java support (Darrachequesne, 2020). The application can offer high scalability and portability in terms of client-server connection. WebRTCClient is also responsible for establishing a P2P connection with peers. It will use WebRTC's RTCPeerConnection API for signaling and peer instance creation. A peer is abstracted as a Peer object containing the peer's identification, a peer connection instance, and a data channel instance. A map with a peer's permanent Id and a Peer instance is kept for querying.

NASTS utilities are responsible for connecting with the NATS server, subscribing to a specific channel, and receiving published messages from the channel. It serves as the backbone for the service routing process. Users are required to define filters when subscribing and publishing. The filters will block unqualified messages from users' awareness. Thus, all messages the user attempts to publish or possibly views will be first piped into a filter. This triggers intelligent service matching for individuals. The service logic module contains utility classes for the system. The Eventbus module is designed for application internal message exchange from Greenrobot. The coupling of components can be decreased because direct method calls are transformed to message exchange by EventBus. It adopts the pattern of Pub/Sub to realize internal message exchanging. An object can register and subscribe to a specific kind of event. When another registered object publishes an instance of the event, subscribers can get the instance in order of priority and react to it. There is a concept of a "sticky event" that can be received by subscribers even the event is sent in advance. The use of EventBus decreases the

Figure 6. The implementation of SericeNet



(a) User Mobile App UI (b) The interaction between the peers

employment of listeners and overall composition, thus increase modularity since all components can work separately with communication only via messages.

GreenDaoHelper helps with the initialization of GreenDAO functions. It redefines database update behaviors and provide support for writing and reading encrypted database. The toolkits include multiple utility classes, which are mostly static. It integrates commonly used functions like GPS, string transformer, UUID generator, etc.

4. PERFORMANCE EVALUATION

We have used scenario analysis, failure analysis, server testing and result analysis, load test, stress test, soak test, etc., to evaluate the performance of our system. Some of the significant analysis will introduce below. The performance of the server is tested in terms of response time and throughput. Response time is measured under a variety of circumstances. On the other hand, throughput is used to measure how many requests the server can handle in the given unit of time (normally second).

4.1 Test Setting

In terms of the test scenario, the server will be tested against the three operations identified above. In detail, the registration, login, and fetching peer list operations are being tested under different workloads. The workload refers to the client arrival rate in a unit time (second). For example, a test under workload 300 in 10 seconds will result in a total of 3000 (300*10) client's arrival. It should be clarified that there are overlaps between the operations. For instance, in the login operation, the virtual user will first sign up and then login. Such overlap will influence how we treat the statistics.

Then the response time and the throughput statistics will be collected and used to draw figures. The response time for login and fetch peers are the total time of accumulation to the previous operations, as explained above. That is to say, the response time of login operation is the sum of sign-up and the actual login operations. Tests were carried out five times under each load and the average result was calculated to gain better accuracy. Moreover, the tests were divided into load, stress and soak test types depending on the period and arrival rate. The throughput of the server is recorded during the soak test. The upper bound of server load is identified as 500, which is our ideal maximum client arrival rate in this early stage. The stress test will end at a rate of 1000 clients per second, which is enough to test the capability of the server. And the standard arrival rate is set to 300 clients per second.

On the other hand, we encountered issues when selecting test technologies. Though initially, we aimed to simulate the Android environment as real as possible, hardware capacity vastly limits the possibility. The cost of starting multiple Android emulators is extremely high thus is not considered. However, even simulating using Socket.io client with Java reveals infeasible after attempts. The high concurrent Socket.io clients will soon fully occupy the CPU. Thus, we decided to use Node.js based testing tool Artillery. It can simulate client operations by writing scripts and provides high performance even under high concurrency. Furthermore, for increasing the reality of the simulation, random realistic data is generated by faker.js. The p95 and p99 are selected as attributes in our system. Regarding the detailed test environment, it is listed together with the computer configuration in Table 1.

4.2 Baseline Load Test

The server was tested with 50, 100, 200 and 500 workloads for 10 seconds. Line charts are shown in Fig. 7 (1)-(3). Firstly, the figures are analyzed solely. We can see an obvious boost in response time for all three operations. This is caused by the limitation of the database. The maximum concurrent connection to the local MySQL database is limited to 500. Meanwhile, though there are only 200 clients sending requests each second, the server commonly cannot finish all the requests during the one-second interval. Thus the accumulation of requests results in reaching the bound of the database connection. Thus, some of the following requests are required to queue for database access. On the other hand, the increase of response time reaches its peak at 300-500 clients/s load. This is where the server is saturated. This means all incoming requests are immediately queued until another request is finished and a database connection is available.

Then the figures are compared with each other crosswise. There is a notable increase in response time between the registration and the other two operations. This is due to the increased query to the database. Because login needs to firstly sign up and then verify the login information, which is in total 2 queries, thus, can lead to longer response time. The double of queries also leads to a quick boost of the times, as shown in the figures. Another interesting point is the increase in response time between login and fetching peers' operations. It increases little when the arrival rate is low. However, when the rate is high, the distinction is visible. The storage pattern of peer information leads to the difference. Since the information of peers is attached to the connection instance then stored into a map that is in memory, the querying of the map can be finished in milliseconds when it is small. However, when more clients are online and the volume of the map keeps growing, the query is as

Table 1. Test Environment

Processor	Intel(R) Core (TM) i7-6700HQ @ 2.6GHz (8 CPUs)
RAM	16384MB
Database	MySQL Community Server 5.7.27
Client Emulator	Artillery 1.6.0
Data Generator	Faker.js 4.10.0

efficient. Hence, more key-value pairs can drop into the same bucket in the map, thus increase the time of finding the desired data.

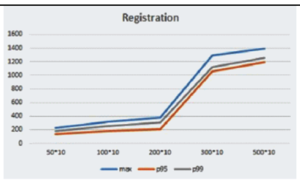
4.3 Stress Test

As 500 clients per second are identified as the ideal upper bound of arrival rate, 600, 700, 800, and 1000 clients per second exceeding the limit were tested. Thus, we can analyze the performance of the server under high stress. The results are shown in Fig. 7 (4)-(6). As shown in the figure, though the stress doubles to 1000clients/s, the response time does not very much. It stays at a similar level as the peek we identified previously. Thus, we believe that the server will not reveal further sudden response time spike under high stress in the current stage. Moreover, the server never fails or produces faults during the stress test such the availability and reliability can be confirmed. One thing worth noting is the gap between p95 and p99 in the fetching operation. This is caused by the map issue identified above. Some key-value pairs fall in the same array in the map, thus leads to a longer response time.

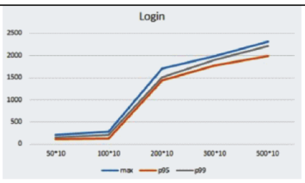
4.4 Soak Test

The result of the soak test is shown in Fig. 7 (7)-(9). This result does not expose much concern as no failure nor a sudden vary in response time occurred. There is a slight decrease in time when the time scale is enlarged, this might occur due to the repetition rate of data generated by faker. js. As time elapses, more comparable data can be generated, thus resulting in more rejection when

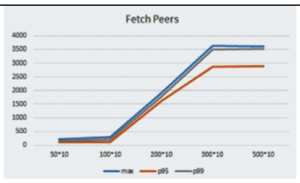
Figure 7. Put the testing results all together



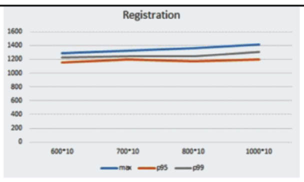
(1) Baseline testing



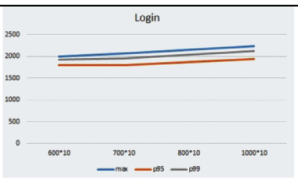
(2) Baseline testing



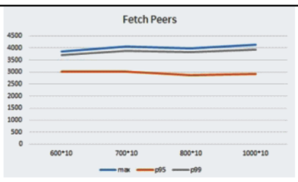
(3) Baseline testing



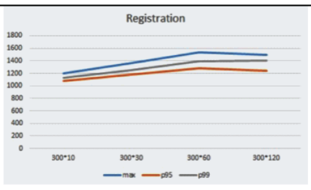
(4) Stress testing



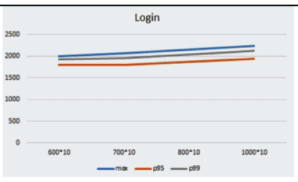
(5) Stress testing



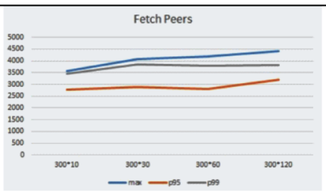
(6) Stress testing



(7) Soak testing



(8) Soak testing



(9) Soak testing

registration. This further leads to immediate failure when the virtual client tries to login or getting peers' information. On the other hand, because the three operations have over-laps, throughput is subtracted to gain a clearer image. For example, the final throughput of login is the product of login throughput subtracts registration throughput. It remains around 200-270 requests/s which is acceptable to handle the request at the current stage.

In summary, the server does offer high availability and reliability both in low and high workloads. It is acceptable that the server can respond in 4 seconds even under high stress for all operations for respond time. Moreover, throughput is enough to handle the workload we expected in such an early stage. The current bottleneck is in the database access.

5. RELATED WORK

Some researchers have contributed a lot to the field of services discovery. Kozlenkov et al. (2006) point that service discovery is regarded as one important aspect while developing service-centric systems. They propose a framework and build a prototype for helping architecture-driven service discovery. The design phase of development lifecycle can be reduced significantly by this framework, which can offer the system's functionalities and satisfy properties and constraints.

Hu, Guan, and Zhong (2006) introduced a decentralized service discovery algorithm, named DSDA, to make service discovery algorithm can be suitable for grid environment. Val, Vasirani, Rebollo, and Fernández (2012) believe that the efficient of service discovery is depending on agents' collaboration and the structure among the parties in a distributed system. They proposed a self-organization mechanism, which can improve the efficient. Yang, Wu, and Chen (2011) illustrate that web service is a new generation of web-based applications. Meanwhile, they also point that with the increasing of quality and quantity of services, it is an urgent question to provide appropriate services for personalized demand. For solving the above question, they introduce an ontology-based approach of service recommendation with dynamic programming theory, which is tested to be relatively accurate. Li, Wang, Sun, and Zhou (2017) also believe that it is very important to recommend personalized web services to users. They think that the temporal influence is an important key factor of Quality of Service. However, the existed papers all ignored it. Thus, they try to add the temporal influences as a factor to predict Quality of Service value, and the result shows that their method outperforms other existed methods.

Several scholars and authors have formerly presented reviews on several use cases of adopting P2P platforms in practice. Kellerer et al. (2007) built a P2P service platform for mobiles for new ways of service provisioning. The system they built has a relative lower cost and higher data privacy. In the field of social web services, Pantazoglou and Tsalgatidou (2008) built a P2P based platform for publication and discovery in order to solve the problem they found. The platform combines the features of decentralization of P2P and maintenance of Web service descriptions, introduces social networking idea of interactions between service demanders and service providers via collective intelligence emerging. Graffi et al. (2011) introduce a life social network platform based on P2P. Liu et al. (2015) propose an improvement of JXTA-Ovarlay with the idea of P2P. Kim and Chung (2018) adopt a hybrid P2P network in health field, and they use it to mine health-risk factors with PHR similarity. There are many papers about adopting P2P platforms in practice, some have been proved to be successful, and only limited applications of applying P2P platform for service-related segments.

Our platform absorbs the advantages of past scholars and removes their disadvantages in p2p respect. At the same time, the characteristics of automatic service discovery are added, which improves performance the docking platform a lot.

6. CONCLUSION

This paper explored the idea of building a P2P decentralized service docking network. The decentralized service docking platform is designed and implemented by integrating several techniques, including publish/subscribe system (NATS), peer-to-peer communication (WebRTC), to support users with flexibility, diversity, and equivalent fairness. This application is implemented on the Android mobile platform utilizing Android Studio for code development. Initial performance tests are also conducted, and the testing results are provided and discussed.

Although there is an initial proof of concept of ServiceNet idea, we foresee this is an interesting topic for research community and industry to further explore its research issues and potential applications along this direction. For example, two immediate future work for us are: (a) to provide interlining service recommendation/routing capability using machine learning technology; (b) to build on-line service booking and payment components with security in mind.

REFERENCES

Darrachequesne. (2020). *EVENT_CONNECT_TIMEOUT in the migration guide*. Retrieved March 2020, from https://github.com/socketio/socket.io-client-java

eMarketer. (2020). *eMarketer Report 2019 for global internet trading market*. Retrieved from http://www.eMarketer.com

Graffi, K., Gross, C., Stingl, D., Hartung, D., Kovacevic, A., & Steinmetz, R. (2011, January). LifeSocial. KOM: A secure and P2P-based solution for online social networks. In *2011 IEEE Consumer Communications and Networking Conference (CCNC)* (pp. 554-558). IEEE.

Hu, J., Guan, H., & Zhong, H. (2006, November). A decentralized service discovery algorithm for grid environment. In *Proceedings of the 4th international workshop on Middleware for grid computing* (p. 19). Academic Press.

Kellerer, W., Despotovic, Z., Michel, M., Hofstatter, Q., & Zols, S. (2007, January). Towards a mobile peer-to-peer service platform. In *2007 International Symposium on Applications and the Internet Workshops* (pp. 2-2). IEEE.

Kim, J. C., & Chung, K. (2018). Mining health-risk factors using PHR similarity in a hybrid P2P network. *Peer-to-Peer Networking and Applications*, *11*(6), 1278–1287.

Kozlenkov, A., Fasoulas, V., Sanchez, F., Spanoudakis, G., & Zisman, A. (2006, May). A framework for architecture-driven service discovery. In *Proceedings of the 2006 international workshop on Service-oriented software engineering* (pp. 67-73). Academic Press.

Lau, R. Y., Liao, S. Y., Kwok, R. C. W., Xu, K., Xia, Y., & Li, Y. (2012). Text mining and probabilistic language modeling for online review spam detection. *ACM Transactions on Management Information Systems*, *2*(4), 1–30.

Li, J., Wang, J., Sun, Q., & Zhou, A. (2017, June). Temporal influences-aware collaborative filtering for QoS-based service recommendation. In *2017 IEEE International Conference on Services Computing (SCC)* (pp. 471-474). IEEE.

Liu, Y., Sakamoto, S., Matsuo, K., Ikeda, M., Barolli, L., & Xhafa, F. (2015). Improvement of JXTA-overlay P2P platform: Evaluation for medical application and reliability. *International Journal of Distributed Systems and Technologies*, *6*(2), 45–62.

Pantazoglou, M., & Tsalgatidou, A. (2008, July). A P2P platform for socially intelligent web service publication and discovery. In *2008 The Third International Multi-Conference on Computing in the Global Information Technology (iccgi 2008)* (pp. 271-276). IEEE.

Val, del E., Vasirani, M., Rebollo, M., & Fernández, A. (2012, June). Enhancing decentralized service discovery through structural self-organization. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 3* (pp. 1429-1430). Academic Press.

Yang, Z., Wu, B., & Chen, J. (2011, July). A Measure standard for ontology-based service recommendation. In *2011 IEEE World Congress on Services* (pp. 137-144). IEEE.

Ji Liu obtained his master's degree from the University of Sydney in 2015. He is taking his doctor degree in School of Electrical & Information Engineering, University of Sydney currently. His research interests include blockchain, tokenization, fintech, data science management and analysis, and some combinations of finance and IT technologies.

Shiping Chen is an IT professional with over 20 years research experiences and combined R&D skills. From 1990 to 1999, he worked on real-time control, parallel computing and CORBA-based Internet gaming systems in research institutes and the IT industry. Since joining in CSIRO in 1999, he has worked on a number of middleware-related research and consultant projects, including software architecture, software testing, software performance modelling and trust computing. He has published extensively in these areas, ranging from academic research papers to in-depth industry reports. In the past several years, he has been working closely with the University of Sydney by teaching and co-supervising PhD/Master students. He is actively involved in research community services on web and service computing areas including WWW, ICSOC, ICWS, SCC. His current research interests include service computing (esp. Web Service Management Systems) and cloud computing (esp. Trusted Cloud Storage and Cloud-based Secure Collaboration Systems).

Dong Yuan was born in Jinan, China. He received the B. Eng. degree and M. Eng. degree from Shandong University, Jinan, China, in 2005 and 2008, the PhD degree from Swinburne University of Technology, Melbourne, Australia, in 2012, all in computer science. He is currently a Senior Lecturer in the School of Electrical and Information Engineering at the University of Sydney, Australia. His research interests include cloud and edge computing, data management in parallel and distributed systems, scheduling and resource management, business process management and workflow systems.