Machine Learning for Vehicular Wireless Network and Edge Computing - Vehicular Network

Department of Electrical & Computer Engineering

College of Design and Engineering

National University of Singapore

April 2022

Supervisor: Tham Chen Khong

Name: Liu Weihao

Student ID: A0232935A

Module: EE5003, MSc project

Contents

Abstract

Vehicular edge computing (VEC) is an effective method to increase the computing capability of vehicles. The BS (base station) is usually used to select a specific service vehicle to complete the heavy task generated from task vehicle. However, due to the high mobility of the vehicle, the offloading BS is always different, and have different policies. In this project, we developed a blockchain-enabled VEC framework to exchange vehicles' historical information. Besides, we use deep reinforcement learning (DRL)-based computation offloading scheme for the smart contract of blockchain, where task vehicles can offload part of computation-intensive tasks to neighboring vehicles. Considering the offloading program may cost too much computing resources of the BS, we use the DDQN algorithm to select the blockchain consensus node to minimize the communication delay. The experiments result shows that Blockchain-enabled VEC network is much efficiency.

Keywords: Vehicular edge computing (VEC), task offloading, Ray, Blockchain, Hyperledger

1. Project Overview

With the rapid development of autonomous driving and vehicular network, various of IoT applications are equipped with to improve the intelligence level of the vehicles. However, because of the limitations of the IoT equipment, a single vehicle may not complete the hard task in a limited time. VEC (vehicular edge computing) is a good choice to solve this problem.

As shown in Fig 1.1, we consider a simple VEC scenario where multiple BSs (base stations) are distributed alongside the roads, and each BS is equipped with an edge server that is responsible for computing resource allocation among vehicles. When a vehicle enters the communication range of a BS and is willing to participate in the VEC. When a vehicle cannot execute all its local tasks due to the limited onboard computing resources, it can offload it tasks to either BSs or some neighboring vehicles. In this project, we focus on V2V (vehicle to vehicle) computation offloading in VEC.

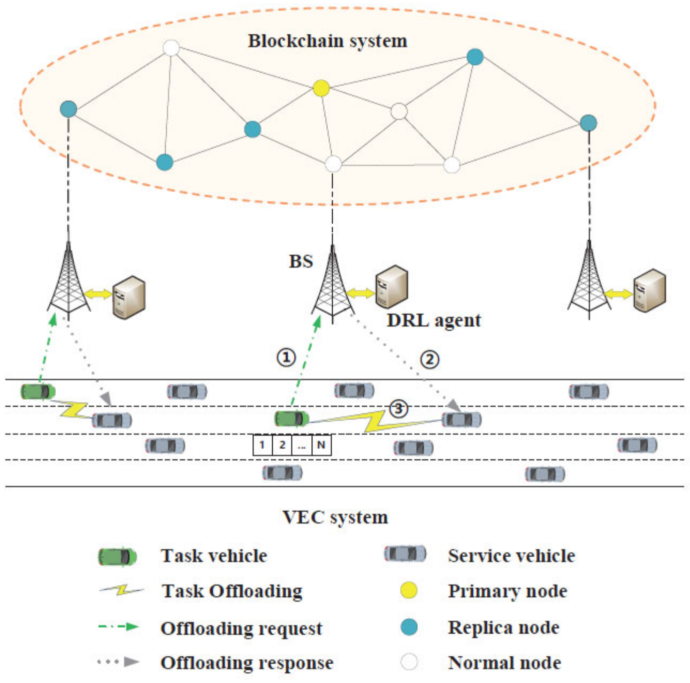


Fig. 1.1 The architecture of the blockchain-enabled VEC system

Since the position of BSs is stationery and BSs serve nearby vehicles all the time, BSs can be used to collect vehicular computing service requests, observe the states of the vehicles inside the communication range, and allocate vehicular computing tasks to vehicles with idle computing resources. If a vehicle needs to offload part of its computing tasks due to the limited onboard computing resources, it first sends a service request to the nearby BS, and we call this vehicle as the task vehicle, and call the vehicles in the communication range of the task vehicle as the service vehicles. Then, the BS performs the vehicular task allocation and selects some of the service vehicles to execute the computing tasks from the task vehicle. During the process of vehicular task allocation, the BS can collect massive experiences in terms of vehicular task offloading which can be used for training the DRL-based algorithm of vehicular task allocation.

1.1 Offloading Algorithm (SAC)

In the problem of vehicular task allocation, the action in terms of the selection of service vehicles and the determination of service price contains both discrete decision variables and continuous variable, we thus employ soft actor-critic (SAC) method to obtain the hybrid decision policy in vehicular task allocation. SAC is a reinforcement learning algorithm which is based on off-policy actor-critic model. To be specific, the actor network in SAC is used to generate action according to the observed system state and improve the policy, while the critic network is responsible for evaluating the policy provided by the actor network.

1.2 Blockchain Network

In the VEC system, a task vehicle with several offloading tasks transmits the offloading request to a nearby BS, after which the smart contract in the BS is triggered. The BS gathers information from the cars around the task vehicle and analyzes the dependability of the vehicles based on the transactions stored in the consortium blockchain before selecting a service vehicle for task offloading. After the V2V computation offloading is completed, the blockchain consensus generates and verifies a transaction that reflects the outcome of the computation offloading. The consensus nodes in each round of consensus are chosen from the VEC system's BSs and then conduct the PBFT-based consensus.

1.3 DRL-Based consensus node selection (DDQN)

After each offloading task is completed, the offloading information is recorded as a transaction in the blockchain system, which includes the IDs of the task vehicle and service vehicle, the task profile, the offloading delay, the execution result of the task, and the timestamp. In each round of blockchain consensus, newly generated transactions are verified by blockchain nodes and permanently stored in blocks, and the verified transactions can be used for evaluating the reliability of service vehicles and BSs. Considering that the accuracy of the estimated reliability depends on the timeliness of the transactions, the delay of transaction verification should be optimized in the blockchain consensus. We use DDQN to select the consensus node in our project

2. Individual Work

In this project I'm responsible for the following three part:

- Vehicle to vehicle communication and vehicle to base station communication.
- Tasks transmission and execution
- Blockchain network set up

2.1 Communication Between Vehicle and Base station

In this project, we use PC as BS, raspberry pi as vehicle. The connect to the same AP, in the same network. The equipment topology is shown as Fig 2.1

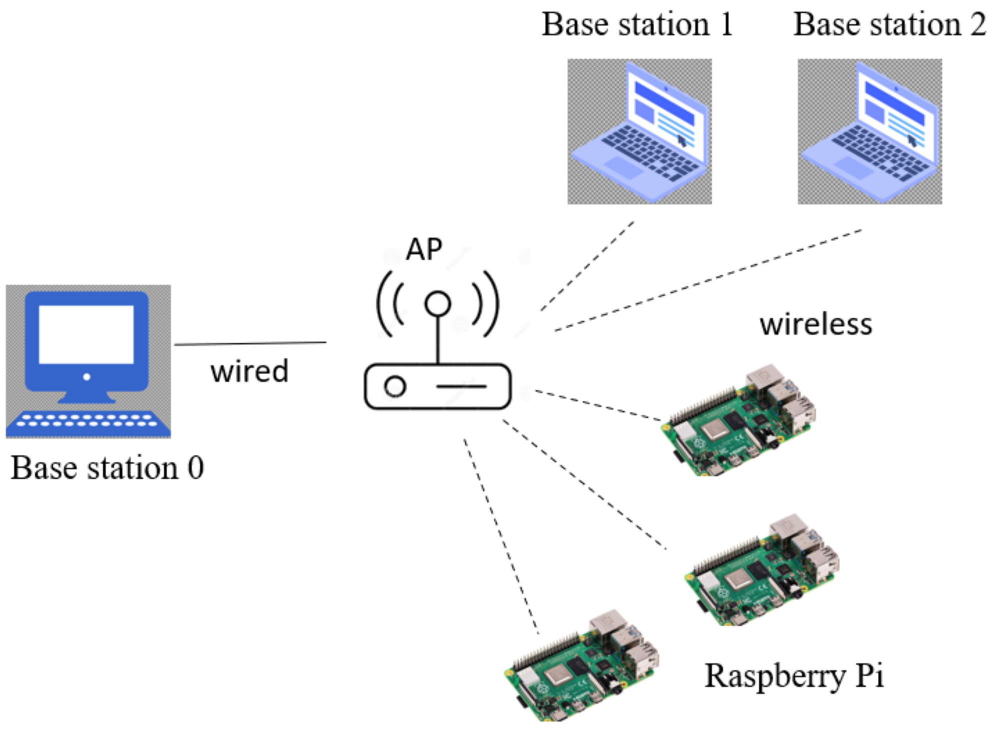


Fig. 2.1 Equipment topology

Whenever the task vehicle (V_t) needs to send a offloading request to the BS, or the BS allocate a specific service vehicle (V_s), they need to use some methods to transmit these commands and data. In the following, we use a UDP based application protocol to transmit these data.

2.1.1 Transport Protocol: UDP

We use Raspberry as our vehicles, and PC/laptop are our base stations. When a vehicle generates a heavy task which cannot be executed by itself, it will send request to our BS, this vehicle will be marked as "Task vehicle". We want to transmit the request packets as faster as possible to decrease the time delay. Besides, vehicles may don't know the BSs IP address, so they need to broadcast the message to the entire LAN.

Compare with TCP, there's no flow control, congestion control in UDP protocol, but its latency is very low, and it could broadcast the messages. So, UDP is very suitable in this scenario.

Table 2.1 Difference Between TCP and UDP

Feature	**TCP**	**UDP**
Connection status	Requires a stable connection to transmit data, the connection should be closed once transmission is complete	Connectionless protocol with no requirements for opening, maintaining, or terminating a connection
Speed	Slower	Faster
Latency	Higher	Lower
Broadcasting	Does not support	Support
Guaranteed delivery	Can guarantee delivery of data to the destination	Cannot guarantee delivery of data to the destination

Anyway, UDP cannot guarantee all the packets transmitted successfully, which may cause task offloading failed. So, we need to design our own application protocol to make sure that all the request messages are received by BS. In application layer, after a UDP server (BS) receives a UDP request, it will reply with a "ACK" packet indicating that the request has been received. The retransmission process is shown in Fig 2.2 If request/ACK packet lost, client will retransmit the request packet until it receives the ACK packet.

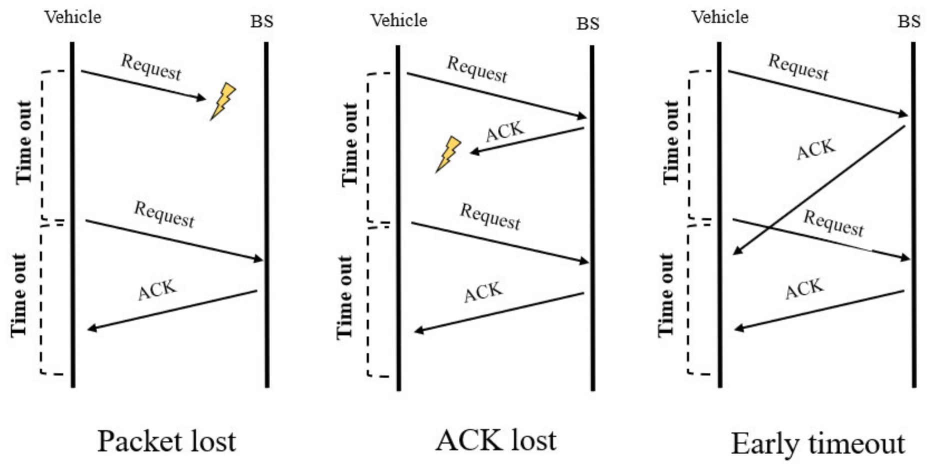


Fig. 2.2 Reasons for retransmission

2.1.2 Application Layer Protocol Design

There are 4 fundamental phases in the offloading process: 1) Task generate and request phase; 2) Service vehicle selection phase; 3) Task processing phase; 4) Task verification and rewards phase. In first phase, our Raspberry Pi generate some random tasks, and they will estimate the tasks' profile such as: task data size, required computation size, etc. And integrate these parameters in its UDP request packet and send to the BS.

In the vehicle selection phase, the BS put all the information received in first phase into its state space and apply some policy to select service vehicle. Then BS will reply to the task vehicle's request, send him a UDP packet and tell him which service vehicle is allocated to its task.

The task processing phase is our task offloading program which introduce in section 3. Once service vehicles complete the task, we come to phase 4, task verification and rewards phase. In this phase, task vehicles need to verify the completion of the task, and check whether it satisfy the time constrain. Then, task vehicles transmit this information to base station. According to the results, BS will give service vehicle penalty or reward. The entire process is shown bellow. (Fig 2.3)

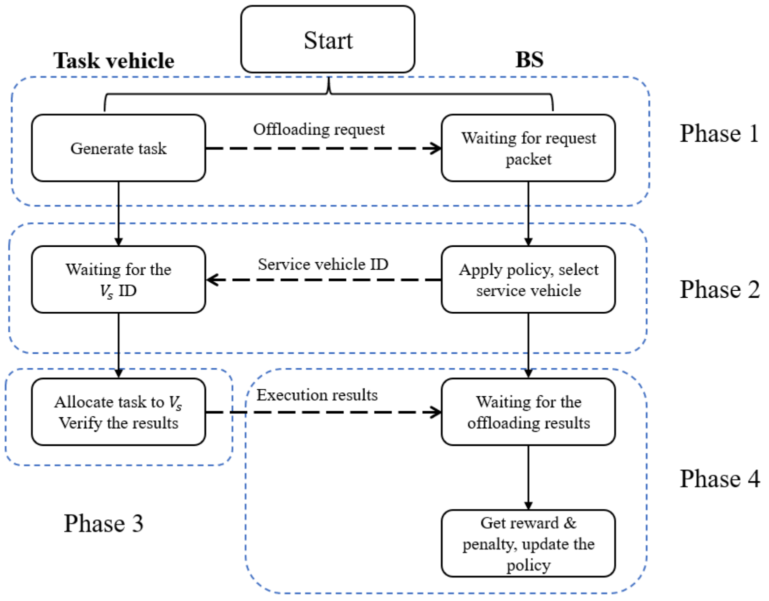

Start
Task vehicle
BS
Generate task
Offloading request
Waiting for request packet
Phase 1
Waiting for the V_s ID
Service vehicle ID
Apply policy, select service vehicle
Phase 2
Allocate task to V_s Verify the results
Execution results
Waiting for the offloading results
Phase 4
Phase 3
Get reward & penalty, update the policy

Fig. 2.3 Task offloading process

As it's shown in Fig 2.3, we have diverse types of UDP packets, so it is necessary to indicate the messages type in the packet. In the application layer, we split the packet into three parts: 1) Packet length; 2) Packet head; 3) Data set. Packet length determines the total length of this UDP packets. The packet structure is shown in Fig 2.4.

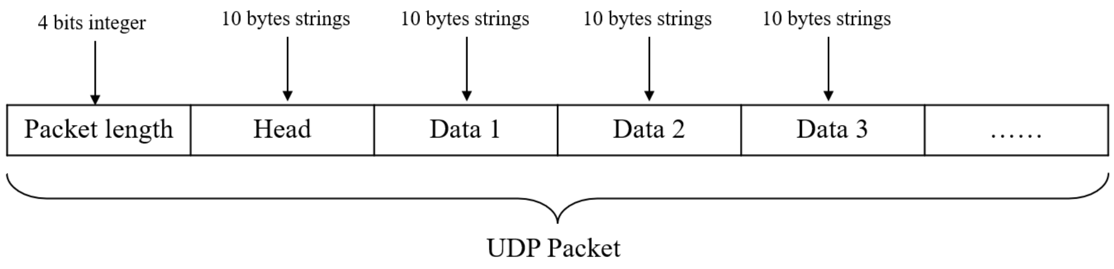


Fig. 2.4 Data Frame

In our project, there are five types of messages, which is shown in table 2.2. When the vehicle and BS need to transmit data, they can integrate all the data into one packet which can decrease the communication latency significantly.

Table 2.2 Application messages

Type	Length	Head	Data1	Data2	Data3	Data4	Data5
Offloading request	6 packets	request	Vehicle ID	Event ID	D_n	C_n	τ_n
Results verification	4 packets	complete	Vehicle ID	Time delay	Results	-	-
Update state	4 packets	update	Vehicle ID	F_s	γ_s	-	-
Select vehicle	3 packets	offloading	Vehicle ID	F_S	-	-	-
ACK	2 packets	ACK	Vehicle ID	-	-	-	-

Because UDP protocol is at Transport Layer, it can only transmit binary data. We need to convert our data into binary format. Alternatively, the first character of the format string can be used to indicate the byte order, size, and alignment of the packed data, according to the following table:

Table 2.3 Format strings

Character	Byte order	Size	Alignment
@	native	native	native
=	native	standard	none
<	little-endian	standard	none
>	big-endian	standard	none
!	network(=big-endian)	standard	none

Native byte order is big-endian or little-endian, depending on the host system, i.e., x86 are little-endian; ARM and Intel Itanium are bi-endian. Native size and alignment are determined using the C compiler's sizeof expression. This is always combined with native byte order.

In our application protocol, the sender collects all the data should be transmitted at first, integrate them into the same data packet which is string format. Then, all the data will be encoded and sort as network byte streams. After that, sender could use UDP protocol to transfer the packet to the receiver. At the receiver side, our application will decode the byte data and split them up for different programs. This process is show in the following figure.

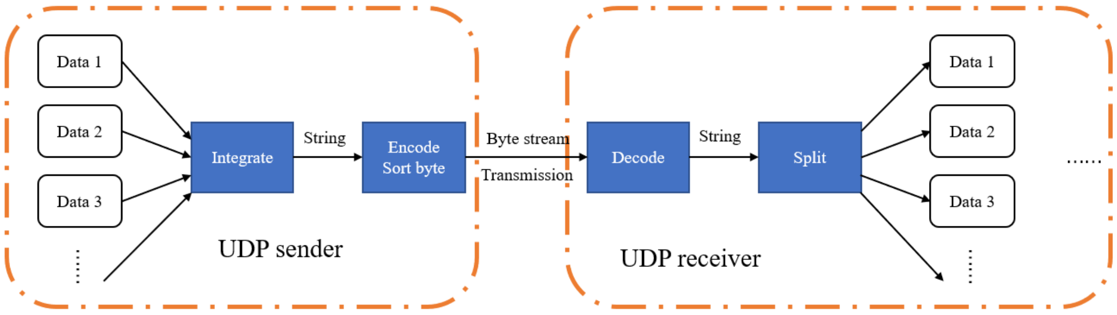


Fig. 2.5 Pack and transfer data

2.1.3 Handling of Abnormal Data

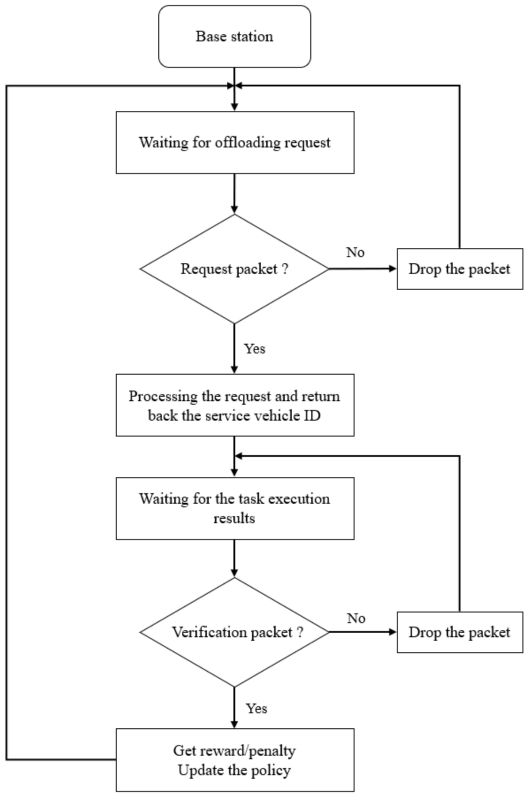

Base station
Waiting for offloading request
Request packet ?
No
Drop the packet
Yes
Processing the request and return back the service vehicle ID
Waiting for the task execution results
Verification packet ?
No
Drop the packet
Yes
Get reward/penalty Update the policy

Fig. 2.6 Base station handles abnormal packet

However, the base station needs some time to deal with the offloading request. But there are far more vehicles than base stations, the task request rate may exceed the base station processing rate. Besides, if a base station is waiting for the task execution results (shown in figure 2.3) but receive an offloading request, the base station may crash and get stuck in an infinite loop. So, we have to devise a propel flow to ensure that the base station can still process the request under high load. The flow chart is shown in Fig 2.6. When base station is waiting for a specific type of packet, it will drop all other packets until the correct packet is received.

Similar with base station, the vehicle also needs to deal with the error packet to avoid getting stuck in infinite loop. As it is shown in Fig 2.7, If the request packet transmits failed, the vehicle will get stuck in the waiting process, waiting for the nonexistent service vehicle ID. So, it needs to make sure the request packet is received properly via "ACK" packet. Otherwise, vehicle will sleep 1 seconds and retransmit the request packet.

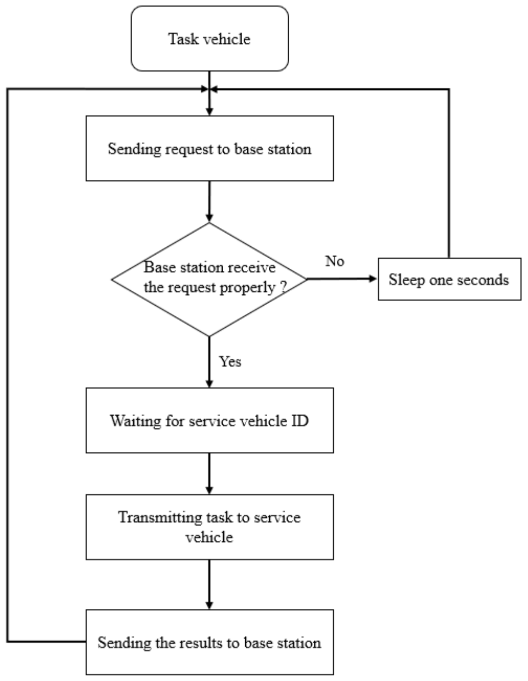
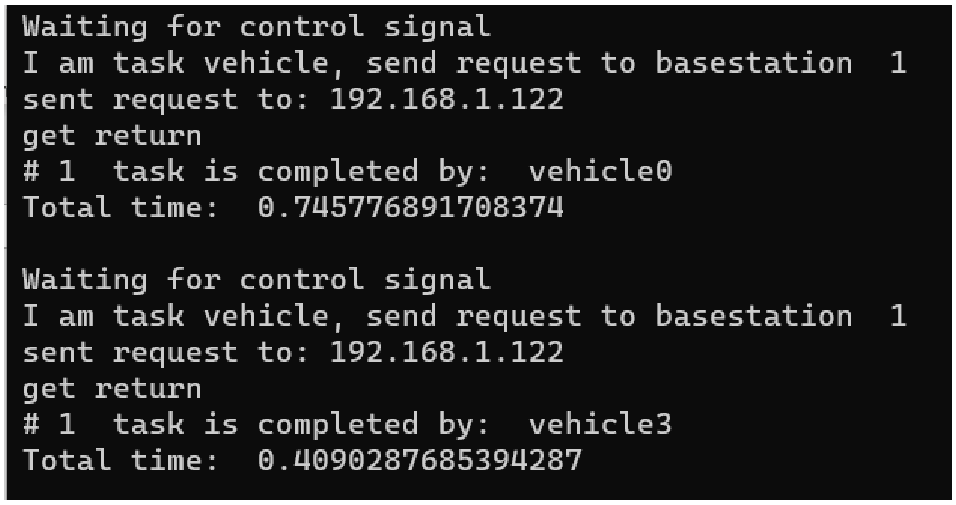

Task vehicle
Sending request to base station
Base station receive the request properly ?
No
Sleep one seconds
Yes
Waiting for service vehicle ID
Transmitting task to service vehicle
Sending the results to base station

Fig. 2.7 Vehicle handles abnormal packet

After we apply these two methods, our program never crashes again, and could run continuously for 2 days.

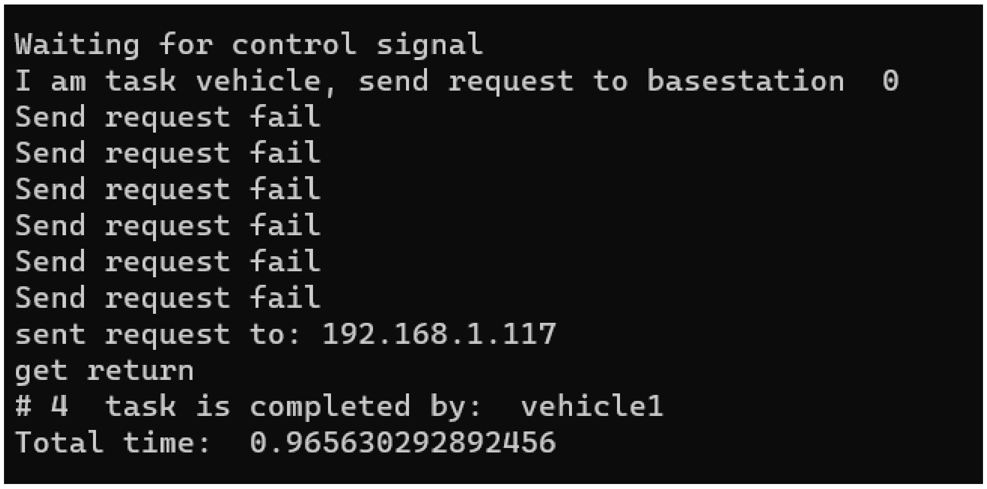
2.1.4 Results

The experiment results is shown as follows:



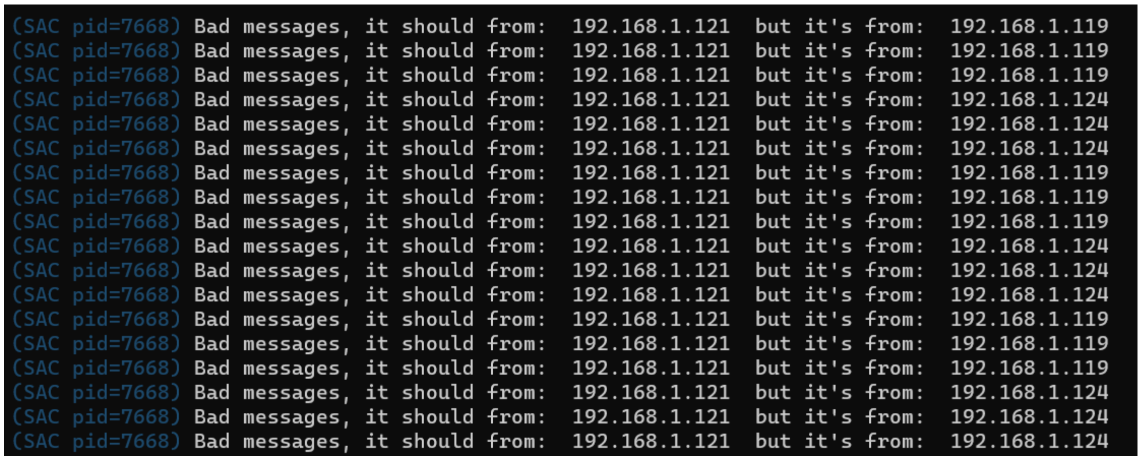
```
Waiting for control signal
I am task vehicle, send request to basestation  1
sent request to: 192.168.1.122
get return
# 1  task is completed by:  vehicle0
Total time:  0.745776891708374

Waiting for control signal
I am task vehicle, send request to basestation  1
sent request to: 192.168.1.122
get return
# 1  task is completed by:  vehicle3
Total time:  0.4090287685394287
```

Fig. 2.8 Transmit successful



```
Waiting for control signal
I am task vehicle, send request to basestation  0
Send request fail
Send request fail
Send request fail
Send request fail
Send request fail
Send request fail
sent request to: 192.168.1.117
get return
# 4  task is completed by:  vehicle1
Total time:  0.965630292892456
```

Fig. 2.9 Packet retransmit



```
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.119
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.119
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.119
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.124
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.124
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.124
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.119
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.119
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.119
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.124
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.124
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.124
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.119
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.119
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.119
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.124
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.124
(SAC pid=7668) Bad messages, it should from:  192.168.1.121  but it's from:  192.168.1.124
```

Fig. 2.10 Base station drop packet

2.2 Tasks Transmission and Execution

Task transmission is the most important part in the VEC, no matter which vehicle is selected as V_s (service vehicle), V_t (task vehicle) needs to transmit the task data and task program to the V_s. V_s will execute the task program and import the task data into the program. After that, it will return the results to our V_t. In this part, we use two methods:

- Transmit task, command via Ray
- Transmit task file and program using HTTP, and UDP to send command

After comparing these three methods, we finally us Ray to allocate task.

2.2.1 Task Allocation Based on Ray

2.2.1.1 Introduction of Ray

Ray is a general-purpose and universal distributed compute framework, we can flexibly run any compute-intensive Python workload — from distributed training or hyperparameter tuning to deep reinforcement learning and production model serving.

One of Ray's strengths is the ability to leverage multiple machines in the same program. When multiple machines form a Ray cluster, each one could call the computing resources of all machines. A Ray cluster consists of a head node and a set of worker nodes. The head node needs to be started first, and the worker nodes are given the address of the head node to form the cluster:

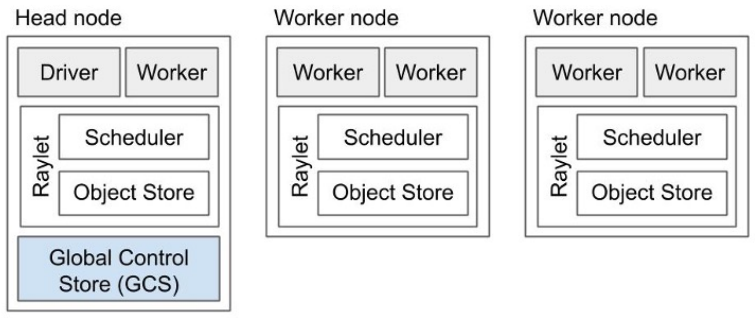


Fig. 2.11 General Ray cluster

We can use the Ray Cluster Launcher to provision machines and launch a multi-node Ray cluster. We can use the cluster launcher on remote server. In other word, we can also use the computing resources on other Ray cluster. The "job management" is the devised to handle the remote tasks.

Ray Job submission is a mechanism to submit locally developed and tested applications to a running remote Ray cluster. It simplifies the user experience of packaging, deploying, and manage their Ray application as a "job". Jobs can be submitted by a "job manager", like Airflow or Kubernetes Jobs. There are some basic concepts:

- **Package**: A collection of files and configurations that defines a Ray application, thus allowing it to be executed in a different environment remotely (ideally self-contained). Within the context of Job submission, the packaging part is handled by Runtime Environments, where we can dynamically configure your desired Ray cluster environment, actor, or task level runtime environment for the submitted Job.
- **Job**: A Ray application submitted to a Ray cluster for execution. Once a Job is submitted, it runs once on the cluster to completion or failure. Retries or different runs with different parameters should be handled by the submitter. Jobs are bound to the lifetime of a Ray cluster, so if the Ray cluster goes down, any running Jobs on that cluster will be terminated.
- **Job Manager**: An entity external to the Ray cluster that manages the lifecycle of a Job and potentially also Ray clusters, such as scheduling, killing, polling status, getting logs, and persisting inputs / outputs. Can be any existing framework with these abilities, such as Airflow

The following diagram shows the underlying structure and steps for each Job submission.

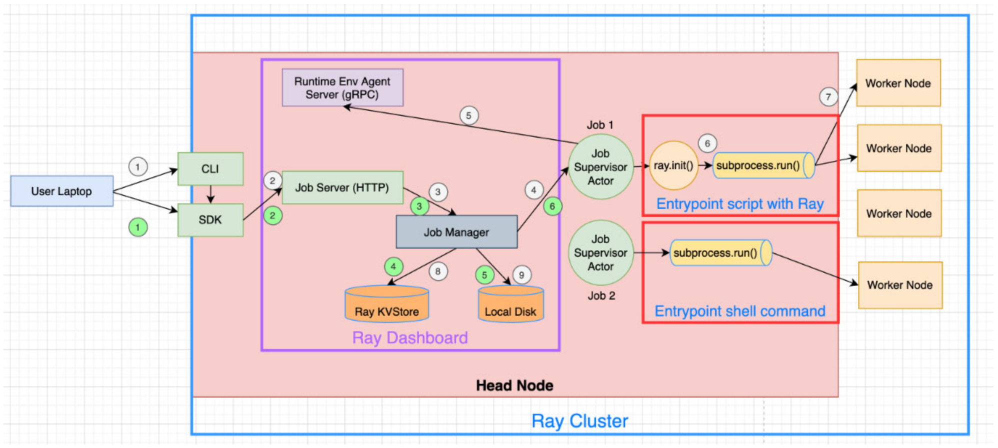


Fig. 2.12 Ray – Job submission architecture

When create a new job, and submit it to the remote Ray cluster, there are nine steps:

1) User submits new job via CLI/SDK entry, either with pre-assigned job id or empty
2) User all HTTP endpoint of Job Server that is running as a dashboard module
3) Job Server get or creates an instance of Job management and all into python API
4) Job management creates a Job supervisor Actor to manage lifecycle of the submitted job
5) Job Supervisor Actor initializes runtime environment passed in from user config for itself and the job
6) User's entry point command /script gets executed in subprocess.run() function
7) User's script gets scheduled and executed in current Ray cluster

8) New job's latest status will be written to current ray cluster's
9) New job's stdout & stderr log will be written to head node's local disk

Ray can also exchange data between the worker node via Ray Datasets. They are the standard way to load and exchange data in Ray libraries and applications. They provide basic distributed data transformations such as map, filter, and repartition, and are compatible with a variety of file formats, data sources, and distributed frameworks. A Dataset consists of a list of Ray object references to blocks. Each block holds a set of items in either an Arrow table or a Python list. When reading from a file-based data source, it creates several read tasks equal to the specified read parallelism (200 by default). One or more files will be assigned to each read task. Each read task reads its assigned files and produces one or more output blocks. In the common case, each read task produces a single output block. Read tasks may split the output into multiple blocks if the data exceeds the target max block size.

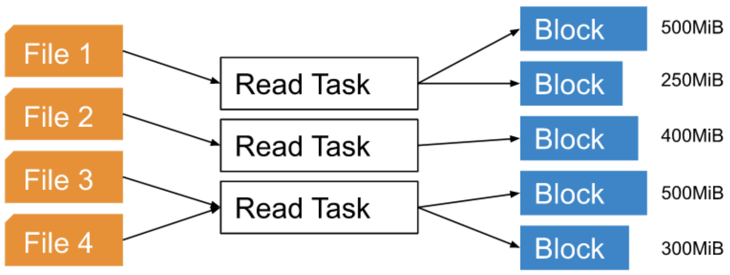


Fig. 2.13 Upload data to Ray Datasets

2.2.1.2 Task Transmission

One of the advantages of Ray cluster is that we don't need to store the IP address table manually, Ray could bind the worker node's IP address to any custom ID. What we only need to do is pre-assigning the ID to every vehicle when they join the ray cluster. In terms of task offloading, only V_s need to execute the task on the Ray cluster which uploaded by V_t. Unfortunately, Ray also has an allocation algorithm, all the idle nodes would execute the task together. But when V_t upload the task to the cluster, it could declare the necessary computing resources i.e., number of CPU/GPU or custom resources. So, when the vehicles join the Ray cluster, it will declare there is a unique computing resources with its own vehicle ID. Fig 2.15 shows that, there are four vehicles in this ray cluster, their unique resources name is vehicle0~vehicle3. When vehicle-x selected as V_s, task vehicle will upload a job and declare that this job needs one "vehicle-x" computing resources to execute.

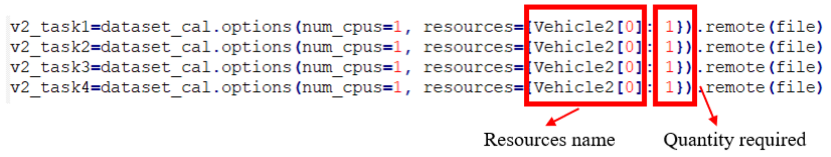
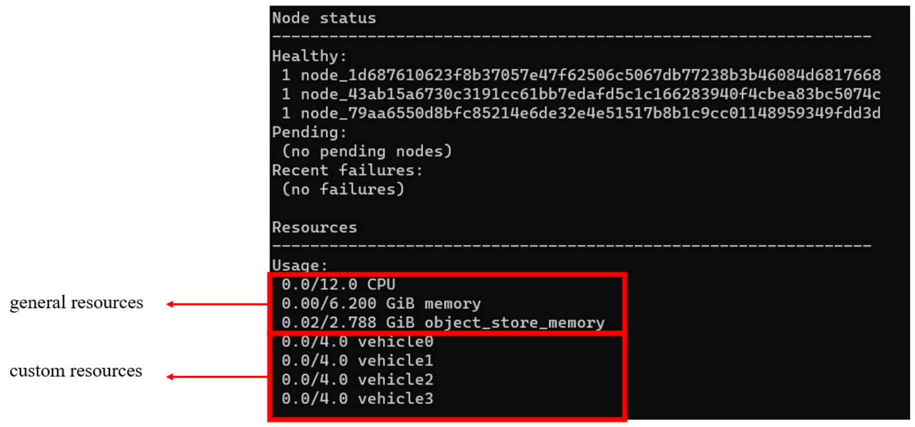

```
v2_task1=dataset_cal.options(num_cpus=1, resources={Vehicle2[0]: 1}).remote(file)
v2_task2=dataset_cal.options(num_cpus=1, resources={Vehicle2[0]: 1}).remote(file)
v2_task3=dataset_cal.options(num_cpus=1, resources={Vehicle2[0]: 1}).remote(file)
v2_task4=dataset_cal.options(num_cpus=1, resources={Vehicle2[0]: 1}).remote(file)
```

Fig. 2.14 Declare necessary computing resources



```
Node status
---------------------------------------------------------------
Healthy:
 1 node_1d687610623f8b37057e47f62506c5067db77238b3b46084d6817668
 1 node_43ab15a6730c3191cc61bb7edafd5c1c166283940f4cbea83bc5074c
 1 node_79aa6550d8bfc85214e6de32e4e51517b8b1c9cc01148959349fdd3d
Pending:
 (no pending nodes)
Recent failures:
 (no failures)

Resources
---------------------------------------------------------------
Usage:
 0.0/12.0 CPU
 0.00/6.200 GiB memory
 0.02/2.788 GiB object_store_memory
 0.0/4.0 vehicle0
 0.0/4.0 vehicle1
 0.0/4.0 vehicle2
 0.0/4.0 vehicle3
```

Fig. 2.15 Cluster computing resources

Because the task function is running on V_t, once the V_s complete the task, it will return the result to V_t directly.

2.2.2 Task Allocation Based on HTTP

The Hypertext Transfer Protocol (HTTP) is an application layer protocol in the Internet protocol suite model for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web, where hypertext documents include hyperlinks to other resources that the user can easily access.

As a request-response protocol, HTTP gives users a way to interact with web resources such as HTML files by transmitting hypertext messages between clients and servers. HTTP clients generally use Transmission Control Protocol (TCP) connections to communicate with servers.

HTTP utilizes specific request methods in order to perform various tasks. All HTTP servers use the GET and HEAD methods, but not all support the rest of these request methods:

- **GET:** Requests a specific resource in its entirety
- **HEAD**: Requests a specific resource without the body content
- **POST:** Adds content, messages, or data to a new page under an existing web resource
- **DELETE**: The DELETE method requests that the target resource delete its state
- **TRACE:** The TRACE method requests that the target resource transfer the received request in the response body.
- **OPTIONS**: Shows users which HTTP methods are available for a specific URL
- **CONNECT**: Converts the request connection to a transparent TCP/IP tunnel
- **PATCH**: The PATCH method requests that the target resource modify its state according to the partial update defined in the representation enclosed in the request.

According to the task offloading requirements, we need to use “GET” request to transmit task data and program, and use “POST” request to return the task results. However, HTTP based offloading program is separately running on task vehicles, service vehicles and base stations. Because all the tasks are temperate, V_s need to know the path of the file at first. HTTP is not suitable to send these commands and some basic information, we can use our UDP based application protocol which is mentioned is section 2.1 to send these commands and information. The entire process is shown in the Fig 2.16

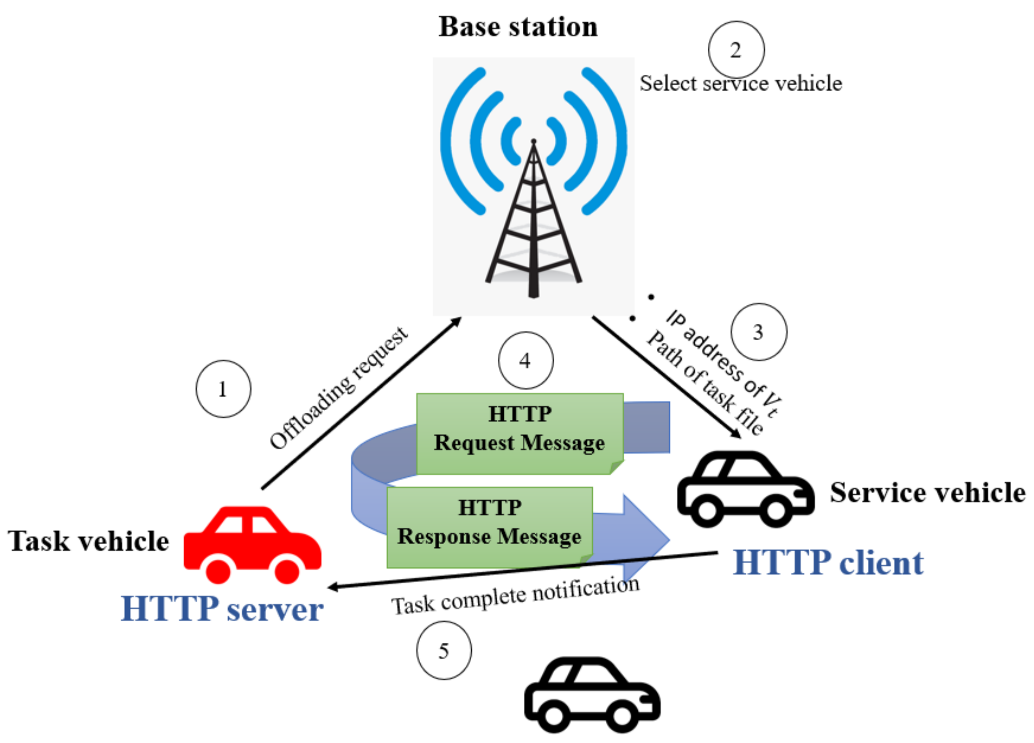


Fig. 2.16 HTTP based task offloading

Besides, HTTP is based on TCP, the vehicle ID is not enough to connect to the HTTP server. So, the base stations need to create an IP table. First, V_t generates a temporary task and send the offloading request and some necessary information to the base station (include the task file path and http server IP address). Once base station receives the request, it will apply its policy and select a suitable service vehicle. Then, it will integrate all this information and send the offloading command to the V_s. According to the offloading packet, V_s get the HTTP server's IP address and the task path of task file. It will send a "GET" request to the HTTP server to get the task file. After completing the task, it will send a "POST" request to the HTTP server and transmit the task results to the V_t. Because HTTP server and offloading program are separated, V_t don't know the task execution status only if V_s send additional notification.

2.2.4 Comparison of the two Methods

Because the vehicles on urban road are fast, they can only communicate for a short period of time. We have to split a heavy task into many tiny tasks. Each of them should be completed within a limited time (maximum 4 seconds). It's very important to decrease the communication delay so that we can remain enough time for the task execution. Besides, the complexity of the method is another important consideration. In terms of the communication delay, HTTP based method is much more efficiency. Because both Ray and HTTP use TCP to transmit the task data, their data transmission rate are similar. But except data transmit, Ray also applies some splitting, security algorithm which will reduce the communication efficiency.

On the other hand, Ray has integrated all the offloading function in its source code, we can just call these functions when we need to use them. It makes Ray based program much easier than HTTP based. Their comparison is shown in the following table:

Table 2.4 Comparison of Ray based method and HTTP based method

Aspects	Ray	HTTP+UDP
Transmission rate	Medium	Medium
Complexity	Simple	Complex
Communication latency	Medium	Low
Task execution time (data only)	-	0.31 seconds
Task execution time (data and program) **Execute once**	2.21 seconds	1.32 seconds
Task execution time (data and program) **Execute 50 times**	0.49 seconds	1.32 seconds

We use a 0.5 MB task file, 200 bytes task program to test the task execution time. The transmission rate is about 20 Mbps. "Data only" means, the task program pre-transmits to the V_s. It only needs to import the task data into the task program. "Data and program" mean V_s knows noting before receiving the offloading command, it needs to transmit task program and task file at the same time. Because our task program is a python program, V_s need some time to compile it into executable program. Compare the fourth and fifth row, the compile process cost about 1 seconds. Ray based method needs more time. But Ray cluster has a cache, it will store all the task's executable program which used before unless we delete it manually. So, after executing the same task 50 times, Ray methods can transmit the executable program directly which could greatly reduce the execution time. But HTTP method still need to compile the task program, which is shown in fifth and sixth row.

In terms of our task offloading requirements, Ray based method is much more suitable, so we decide to use this method in our project.

2.2.3 Task Execution

2.2.3.1 Computation Model

All the offloading tasks are independently in VEC. The profile of an offloading task can be expressed as $\{D_n, C_n, \tau_n\}$, where D_n is the task data size, C_n is the estimate computation size which represents the required CPU cycles for completing the task, τ_n is the maximum tolerance delay. Because our task result is just a number, we can ignore its transmission time. So, the task offloading delay consists of two parts: transmission and execution.

$$t_n = \frac{D_n}{r_{ts}} + \frac{C_n}{f_n}$$

Where r_{ts} is the V2V transmission rate, f_n is the allocated computing resources. Base station needs to use all these parameters to select the service vehicle. So, before scheduling the task to a specific service vehicle, V_t need to estimate the data size and required computing resources, send them to the base station. Besides, r_{ts} and f_n is also necessary. In the paper, the authors use the follow formula estimate the transmission rate:

$$r_{ts} = W_{ts} \log_2(1 + \frac{P_t d_{ts}^{-\alpha} h_{ts}^2}{\sigma_\omega^2 + I_{ts}})$$

Where W_{ts} is the allocated bandwidth of the V2V link between V_t and V_s, P_t is the transmission power of V_t, d_{ts} is the distance between V_t and V_s, α is a constant which represents the path loss factor, h_{ts} is the channel gain of the V2V link, σ_ω^2 represents the power spectrum density of additive white Gaussian noise, and I_{ts} denotes the interference introduced by other V2V transmissions.

In our project, we use SUMO & Omnet ++ simulate the vehicles' trace and calculate the transmission rate. Besides, there's only offloading program on our Raspberry, V_s CPU always free, so we use another program which running on base station will tell the vehicle its F_s and f_n. Our parameters table is shown as follows:

Table 2.5 Simulation parameters

Parameter	Value
$\boldsymbol{F_s}$	$[3, 7]$ GHz
$\boldsymbol{D_n}$	$[0.2, 4]$ Mbits
$\boldsymbol{C_n}$	$[0.2, 3.6] \times 10^9$ cycles
$\boldsymbol{\tau_n}$	$\{0.5, 1, 2, 4\}$ s

2.2.3.2 Task Generate

Table 2.5 shows the task file size is 0.2~4 Mbits. So, we create for .csv task file. Their size are 200 Kbits, 1 Mbits, 2 Mbits, 3 Mbits and 4Mbits. All this task file has the same format: two columns, the data in each column will increase from 1 to larger. When the task vehicle receives the command from SUMO program, it will select a task file from pool. We will apply different calculation types for these columns. When the calculation types are decided, every row will use it and sum up their results. Then, the task program will 2 random number, which determines the task type, cos, sin, log. They are used to select the calculation types for the two columns, this process can be expressed as:

$$result = \begin{cases} C\sum_{i=1}^{i=row}\cos(n_i) & ,rv=1 \\ C\sum_{i=1}^{i=row}\sin(n_i) & ,rv=2 \\ C\sum_{i=1}^{i=row}\log(n_i) & ,rv=3 \end{cases}$$

Where rv is a random number selected from 1,2,3, n_i is the numbers read from our task file. C is a constant.

2.2.3.3 Parameters Estimation

We assume all the Raspberry has same computing capacity, $F_s = 7\ GHz$. From 2.2.3.1 we know that the execution time is $t = \frac{C_n}{f_n}$, so we can test all the tasks required computing resources are given by: $C_n = 7\ GHz * t$. So, we test all the tasks generate in 2.2.3.2 and get all the C_n. We found that, the C_n only depends on the file size. From table 2.5 we can get that, $C_n = 0.2{\sim}3.6$ our execution time should in the range of: $[0.02s,\ 0.514s]$. Only execute our tasks once cannot satisfy the time constraints, so we create a loop in our programs to execute the tasks multiple times. Table 2.6 shows the execution time for different task in different iteration times. So, after we select the specific task and iteration times, we can get the required computing resources:

$$C_n = iter_b \times \frac{I_t}{t_e} \times 7GHz$$

where $iter_b$ is a random number selected in section 2.2.3.2, I_t and t_e is the iteration times and corresponding execution time from table 2.6 which decided by the file size. Anyway, this formula is based on the RPi's computing capability is a constant $7\ GHz$, but in our simulation, we want different vehicles has different computing capability. Besides, we also want the available computing resources could vary over time and control by the SUMO program. So, when the base station allocates the task to a specific vehicle, it will read the available computing resources f_n from SUMO program and transmit this data to V_s. Because the tasks' D_n and C_n has already selected before task request phase, these parameters are based on $F_s = 7\ GHz$. Because in our scenario, the RPi's computing capability is a constant, we need to multiple iteration times by another parameter.

$$t = \frac{C_n}{f_n} = \frac{C_n}{F_s}\frac{F_s}{f_n} = \frac{kC_n}{F_s} = \frac{k}{F_s} \times iter_b \times \frac{I_t}{t_e} \times F_s = k \times iter_b \times \frac{I_t}{t_e}$$

Where $k = \frac{F_s}{f_n}$. According to the formula, we just need to execute the original task $k \times iter_b$ times, the entire process could satisfy our requirements.

Table 2.6 Estimate task execution time

File size	Execution time (seconds)	Iteration times
200 Kbits	0.02	1
	0.1	15
	0.2	30
1 Mbits	0.04	1
	0.5	15
	1	30
2 Mbits	0.075	1
	0.62	10
	1.3	20
3 Mbits	0.1	1
	0.84	10
	1.4	15
4 Mbits	0.125	1
	1.1	10
	1.55	15

Another parameter is the maximum tolerance delay τ. We assume that the transmission rate is 2 Mbps, allocated computing resource is $5\ GHz$. C_n and D_n are get from task generate part. And we use these parameters to get the expectation offloading deal t, τ should be:

$$\tau = \begin{cases} 0.5, & t < 0.2 \\ 1, & 0.2 < t < 0.7 \\ 2, & 0..7 < t < 1.3 \\ 4, & \text{else} \end{cases}$$

After all these steps, the RPi will generate all the parameters which are base station needed and send them to BS via our UDP request.

2.3 Blockchain Based Database

In our scenario, due to the high dynamic vehicular environment, a task vehicle driving on the road may have different neighboring service vehicles in different time slots, and the task vehicle may not obtain much information from all of the neighboring vehicles in real time, which makes it difficult to evaluate the willingness of neighboring vehicles to contribute their computing resources. Blockchain is deemed as a promising solution, where the historical transactions of vehicular task offloading are stored and utilized to evaluate the reliability of vehicles in resource allocation.

2.3.1 Introduction of Blockchain

A blockchain is a distributed database that is shared among the nodes of a computer network. As a database, a blockchain stores information electronically in digital format. Blockchains are best known for their crucial role in cryptocurrency systems, such as Bitcoin, for maintaining a secure and decentralized record of transactions. The innovation with a blockchain is that it guarantees the fidelity and security of a record of data and generates trust without the need for a trusted third party.

One key difference between a typical database and a blockchain is how the data is structured. A blockchain collects information together in groups, known as blocks, which hold sets of information. Blocks have certain storage capacities and, when filled, are closed and linked to the previously filled block, forming a chain of data known as the blockchain. All new information that follows that freshly added block is compiled into a newly formed block that will then also be added to the chain once filled.

A database usually structures its data into tables, whereas a blockchain, like its name implies, structures its data into chunks (blocks) that are strung together. This data structure inherently makes an irreversible timeline of data when implemented in a decentralized nature. When a block is filled, it is set in stone and becomes a part of this timeline. Each block in the chain is given an exact time stamp when it is added to the chain. Here are some blockchains' features:

- Blockchain is a type of shared database that differs from a typical database in the way that it stores information; blockchains store data in blocks that are then linked together via cryptography.
- As new data comes in, it is entered into a fresh block. Once the block is filled with data, it is chained onto the previous block, which makes the data chained together in chronological order.
- Different types of information can be stored on a blockchain, but the most common use so far has been as a ledger for transactions.
- In Bitcoin's case, blockchain is used in a decentralized way so that no single person

or group has control—rather, all users collectively retain control.

- Decentralized blockchains are immutable, which means that the data entered is irreversible. For Bitcoin, this means that transactions are permanently recorded and viewable to anyone.

2.3.2 Introduction of Hyperledger Fabric

For the implementation, we select Hyperledger Fabric as our consortium blockchain. Hyperledger Fabric is an open-source enterprise-grade permissioned distributed ledger technology (DLT) platform, designed for use in enterprise contexts, which delivers some key differentiating capabilities over other popular distributed ledger or blockchain platforms. It allows components, such as consensus and membership services, to be plug-and-play. Its modular and versatile design satisfies a broad range of industry use cases.

Fabric has three level structures: 1) Entire Blockchain network; 2) Organization; 3) Peer nodes. As a consortium blockchain platform, Fabric network can be seen as a cluster of organizations. Each organization manages its own peer nodes, like a sub cluster. When a new organization want to join this exist network, it should submit a request to the network, and all of the organizations in this network can see the request. After all of this organizations approve the request, others can join this network. Peer node is the fundamental elements of Fabric. Each node belongs to an organization and managed by the organization. The structure is illustrated as figure 2.17.

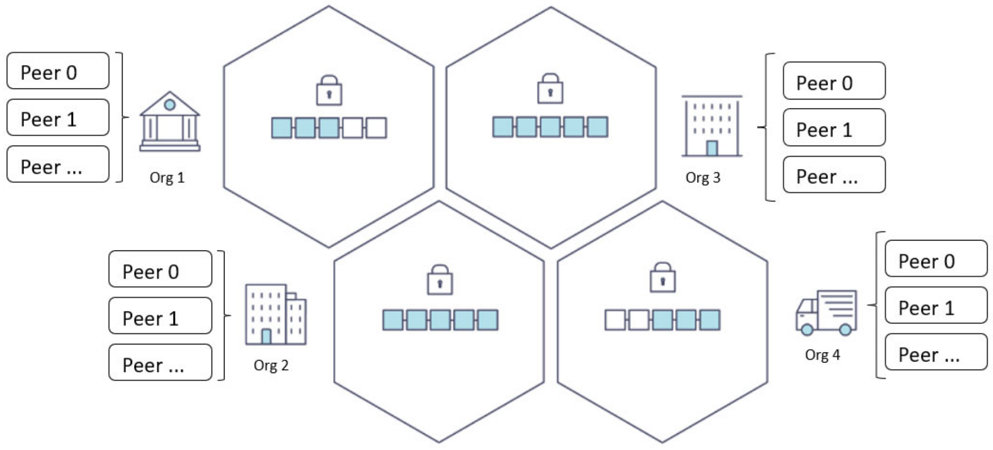


Fig. 2.17 Fabric Blockchain network structure

Here are some basic concepts in Fabric model:

- **Channel**: A channel is a key abstraction in Hyperledger Fabric. It forms a kind of subnet in the network isolating the state and smart contracts. All peers belonging to a channel have access to the same data and smart contracts. There is no access to it outside the channel.

- **Chaincode**: Chaincode execution is partitioned from transaction ordering, limiting the required levels of trust and verification across node types, and optimizing network scalability and performance.
- **Ledger Features:** The immutable, shared ledger encodes the entire transaction history for each channel and includes SQL-like query capability for efficient auditing and dispute resolution.
- **Security & Membership Services**: Permissioned membership provides a trusted blockchain network, where participants know that all transactions can be detected and traced by authorized regulators and auditors.
- **Consensus**: A unique approach to consensus enables the flexibility and scalability needed for the enterprise

2.3.2.1 The Network Topology

Participating organizations have their own roles in the network and manage different nodes. The diagram presents one of the simplest and probably the most common network topology, when the root organization is responsible for both the network membership and the consistency of the state.

There are basically three most important node types in the Hyperledger Fabric network. Certificate Authorities (CA) connect the whole network via certificate's chain of trust. Orderers maintain the consistency of the state and create blocks of transactions. Peers store the ledger, which consists of the transaction log (i.e., the blockchain) and the world state.

A channel is a key abstraction in Hyperledger Fabric. It forms a kind of subnet in the network isolating the state and smart contracts. All peers belonging to a channel have access to the same data and smart contracts. There is no access to it outside the channel.

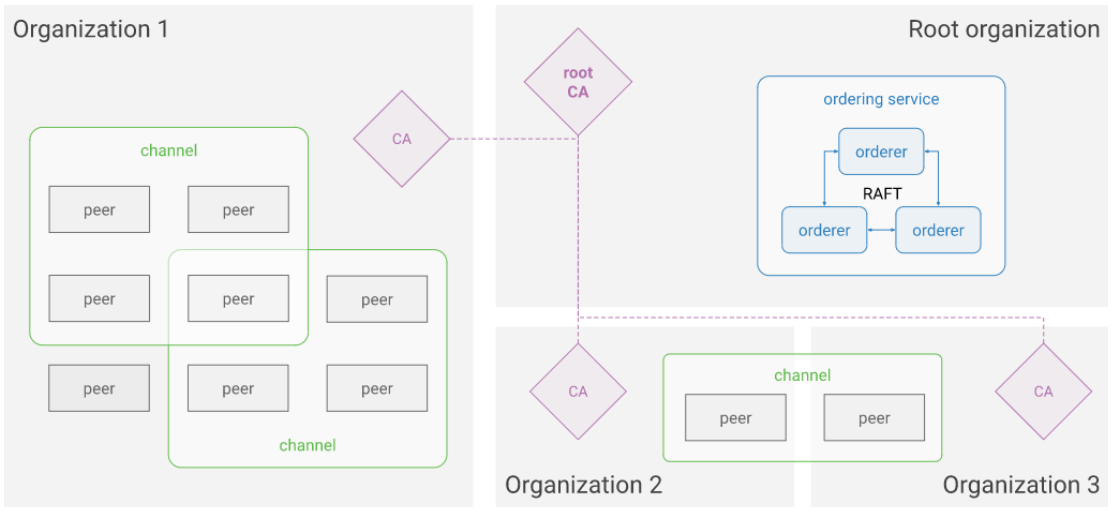


Fig. 2.18 Network topology

2.3.2.2 Blockchain Ledger

The ledger is the sequenced, tamper-resistant record of all state transitions in the fabric. State transitions are a result of chaincode invocations ('transactions') submitted by participating parties. Each transaction results in a set of asset key-value pairs that are committed to the ledger as creates, updates, or deletes. In Hyperledger Fabric, a ledger consists of two distinct, though related, parts – a world state and a blockchain. Each of these represents a set of facts about a set of business objects. Firstly, there's a world state – a database that holds current values of a set of ledger states. The world state makes it easy for a program to directly access the current value of a state rather than having to calculate it by traversing the entire transaction log. Ledger states are, by default, expressed as key-value pairs, and we'll see later how Hyperledger Fabric provides flexibility in this regard. The world state can change frequently, as states can be created, updated and deleted. The relationship between world state and block is shown as follows:

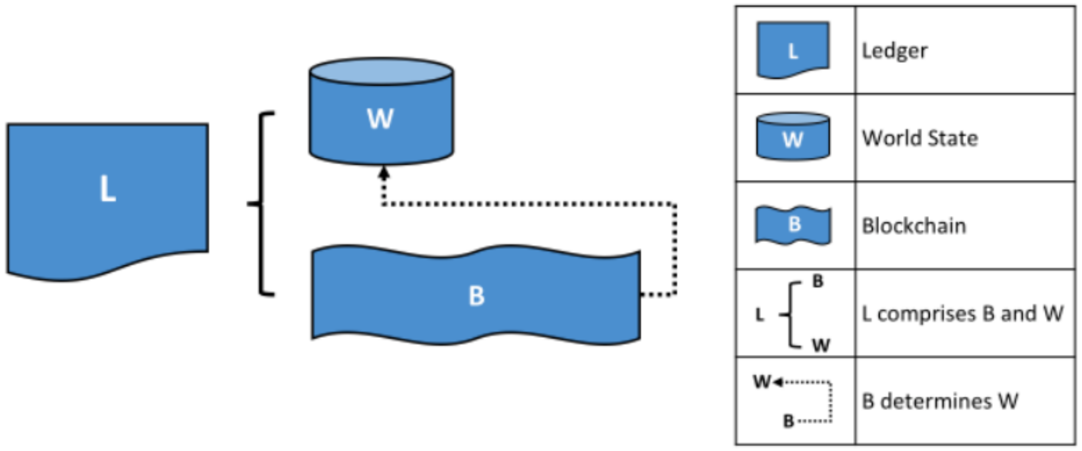


Fig. 2.19 Inside the Fabric ledger

Secondly, blockchain is a log of transactions grouped in blocks, ordered by ordering service. It is physically stored in the peer nodes and immutable. It cannot be changed. Each block contains:

- A header with the block number, a hash and the previous block hash. Thanks to the information from blocks' headers, the whole blockchain is interlinked and resistant to manipulations.
- Metadata with signatures of the block creator and some low-level data for ensuring the consistency of the state.
- A list of transactions with client application signatures, chaincode names, input parameters, reads and writes to the world state and lists of transaction execution outputs from all required organizations.

Block is the fundamental element of the ledger. Fig 2.20 illustrates the blocks' structure.

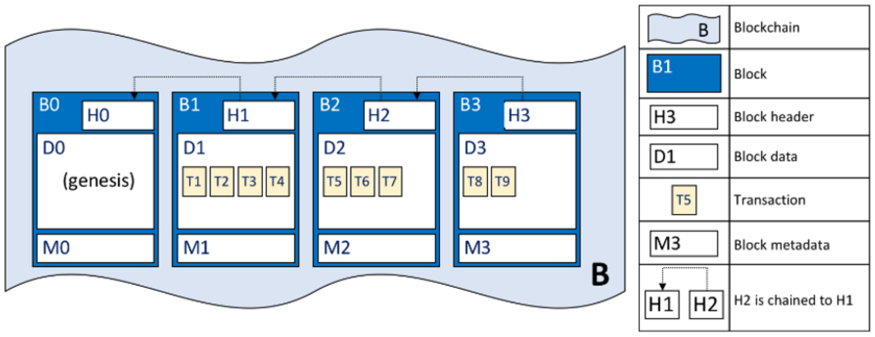


Fig. 2.20 The structure of block

Blockchain is a single source of truth for the ledger. The world state is derived from the blockchain. The world state is a key value store that contains data created by transactions from the blockchain. Unlike the blockchain, the world state is mutable. It represents the latest values for all keys updated in the transactions. Even if a smart contract "reads" and "saves" the world state data, during the smart contract execution, there is no actual change of the world state. A smart contract always has access to the world state from the previous block and the result of execution contains a read/write set, i.e., a list of keys read and a list of keys and values to be updated.

The world state might be stored in an embedded LevelDB database on the peer or in a separated CouchDB node dedicated to the peer. The main benefit of the latter one is the ability to execute complex queries against the world state

2.3.3 Implementation of our own blockchain

In the integration part, we only have two base stations. But in this part, we use four computers to set up our blockchain network, every computer is a node. As section 2.3.2 mentioned, we need at least one organization and one order node to form the network. Besides, we don't need authority manage part, so all the node belongs to the same organization. According to our requirement, we need to design four smart contracts (chaincode) to exchange data between different base station. Because different base station may have different system version, we decide to use the docker to set up the environment.

2.3.3.1 Network Set up

In our private blockchain network, we have two a order server: OrderOrg, and a common organization: Ray. There are three steps to set up the network.

Step one: Set up Certificate Authority (CA)

The first component that must be deployed in a Fabric network is a CA. This is because the certificates associated with a node (not just for the node itself but also the certificates identifying who can administer the node) must be created before the node itself can be deployed.

While it is not necessary to use the Fabric CA to create these certificates, the Fabric CA also creates MSP structures that are needed for components and organizations to be properly defined. If a user chooses to use a CA other than the Fabric CA, they will have to create the MSP folders themselves.

- One CA is used to generate the certificates of the admin of an organization, the MSP of that organization, and any nodes owned by that organization. This CA will also generate the certificates for any additional users. Because of its role in "enrolling" identities, this CA is sometimes called the "enrollment CA" or the "ecert CA".
- The other CA generates the certificates used to secure communications on Transport Layer Security (TLS). For this reason, this CA is often referred to as a "TLS CA". These TLS certificates are attached to actions as a way of preventing "man in the middle" attacks. Note that the TLS CA is only used for issuing certificates for nodes and can be shut down when that activity is completed. Users have the option to use one way (client only) TLS as well as two-way (server and client) TLS, with the latter also known as "mutual TLS". Because specifying that your network will be using TLS (which is recommended) should be decided before deploying the "enrollment" CA (the YAML file specifying the configuration of this CA has a field for enabling TLS), you should deploy your TLS CA first and use its root certificate when bootstrapping your enrollment CA. This TLS certificate will also be used by the fabric-ca client when connecting to the enrollment CA to enroll identities for users and nodes.

Step two: Create identities and MSPs

After we have created the CAs, we can use them to create the certificates for the identities and components related to the organization (which is represented by an MSP). For each organization, we need to, at a minimum:

- **Register and enroll an admin identity and create an MSP**: After the CA that will be associated with an organization has been created, it can be used to first register a user and then enroll an identity. This MSP will used by the organization admin.
- **Register and enroll node identities**: Just as an org admin identity is registered and enrolled, the identity of a node must be registered and enrolled with both an enrollment CA and a TLS CA (the latter generates certificates that are used to secure communications). Instead of admin, this MSP is generated for peer node.

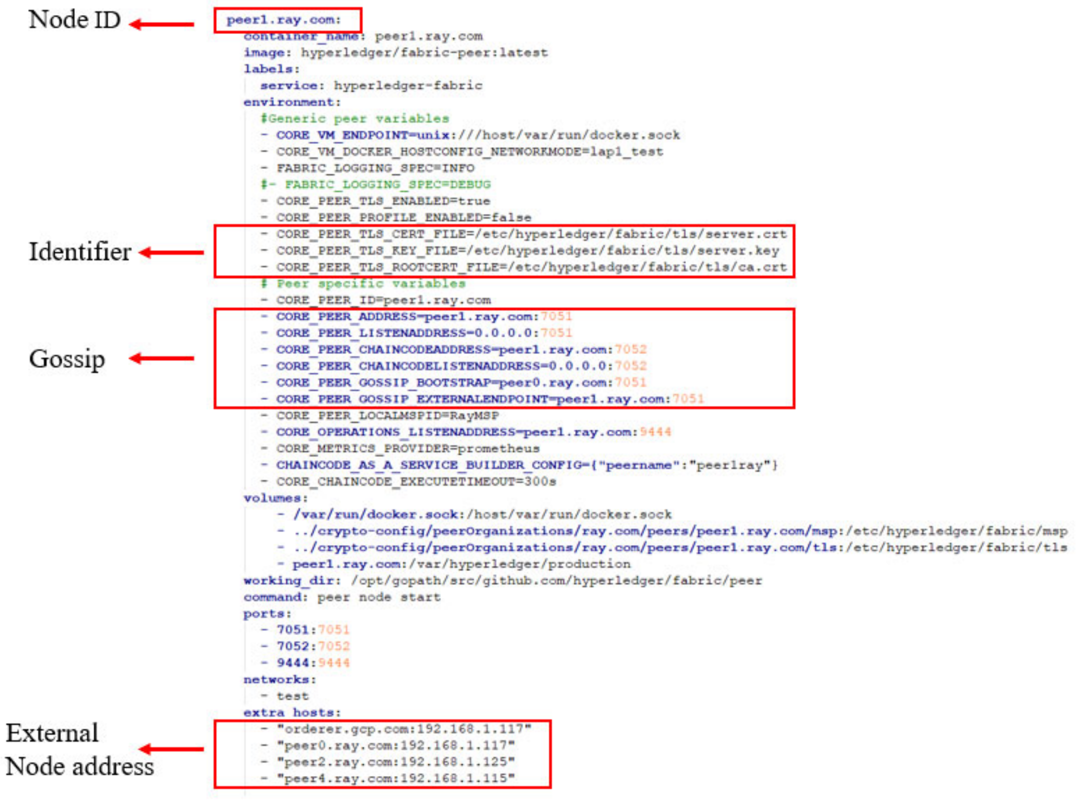
Step three: Deploy peers and ordering nodes

Once we complete the first two steps, we already have completed the preparation for running the blockchain network. The last step is the client configuration. Because we use docker as our environment, so all of the configurations should be stored in the docker-compose.yaml file:

1. **Identifiers**: these include not just the paths to the relevant local MSP and TLS certificates, but also the name of the peer and the MSP ID of the organization that owns the peer.
2. **Addresses and paths**: Because peers are not entities unto themselves but interact with other peers and components, we must specify a series of addresses in the configuration.
3. **Gossip**: components in Fabric networks communicate with each other using the "gossip" protocol. Through this protocol, they can be discovered by the discovery service and disseminate blocks and private data to each other.

2.3.3.2 Docker Environment Set up

Docker is an open platform for developing, shipping, and running applications. Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow us to run many containers simultaneously on a given host. Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host. Because Fabric is an ongoing project, it can only run on some specific system version. It's very suitable to use the docker container to set up our system environment. The docker configuration is shown as Fig 2.21



```
peer1.ray.com:
  container_name: peer1.ray.com
  image: hyperledger/fabric-peer:latest
  labels:
    service: hyperledger-fabric
  environment:
    #Generic peer variables
    - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
    - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=lap1_test
    - FABRIC_LOGGING_SPEC=INFO
    #- FABRIC_LOGGING_SPEC=DEBUG
    - CORE_PEER_TLS_ENABLED=true
    - CORE_PEER_PROFILE_ENABLED=false
    - CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.crt
    - CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.key
    - CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.crt
    # Peer specific variables
    - CORE_PEER_ID=peer1.ray.com
    - CORE_PEER_ADDRESS=peer1.ray.com:7051
    - CORE_PEER_LISTENADDRESS=0.0.0.0:7051
    - CORE_PEER_CHAINCODEADDRESS=peer1.ray.com:7052
    - CORE_PEER_CHAINCODELISTENADDRESS=0.0.0.0:7052
    - CORE_PEER_GOSSIP_BOOTSTRAP=peer0.ray.com:7051
    - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer1.ray.com:7051
    - CORE_PEER_LOCALMSPID=RayMSP
    - CORE_OPERATIONS_LISTENADDRESS=peer1.ray.com:9444
    - CORE_METRICS_PROVIDER=prometheus
    - CHAINCODE_AS_A_SERVICE_BUILDER_CONFIG={"peername":"peer1ray"}
    - CORE_CHAINCODE_EXECUTETIMEOUT=300s
  volumes:
      - /var/run/docker.sock:/host/var/run/docker.sock
      - ../crypto-config/peerOrganizations/ray.com/peers/peer1.ray.com/msp:/etc/hyperledger/fabric/msp
      - ../crypto-config/peerOrganizations/ray.com/peers/peer1.ray.com/tls:/etc/hyperledger/fabric/tls
      - peer1.ray.com:/var/hyperledger/production
  working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
  command: peer node start
  ports:
    - 7051:7051
    - 7052:7052
    - 9444:9444
  networks:
    - test
  extra_hosts:
    - "orderer.gcp.com:192.168.1.117"
    - "peer0.ray.com:192.168.1.117"
    - "peer2.ray.com:192.168.1.125"
    - "peer4.ray.com:192.168.1.115"
```

Fig. 2.21 Docker environment configuration

This node ID is peer1.ray.com. We need to use the Fabric image to ensure that all the environment satisfies the Fabric requirements. Besides we need to import the MSP and CA file to the environment, make sure this node can join the blockchain network. Unlike real computer, an instance in a docker container can not communicate with other computer directly, we need to declare other nodes' IP addresses in the "extra hosts" part. Besides we also need to bind the container's port to the port of our computer in the "ports" part.

2.3.3.3 Create a Channel

In order to create and transfer assets on a Hyperledger Fabric network, an organization needs to join a channel. Channels are a private layer of communication between specific organizations and are invisible to other members of the network. Each channel consists of a separate ledger that can only be read and written to by channel members, who are allowed to join their peers to the channel and receive new blocks of transactions from the ordering service. While the peers, ordering nodes, and Certificate Authorities form the physical infrastructure of the network, channels are the process by which organizations connect with each other and interact. The relationship between channel and node is shown as follows:

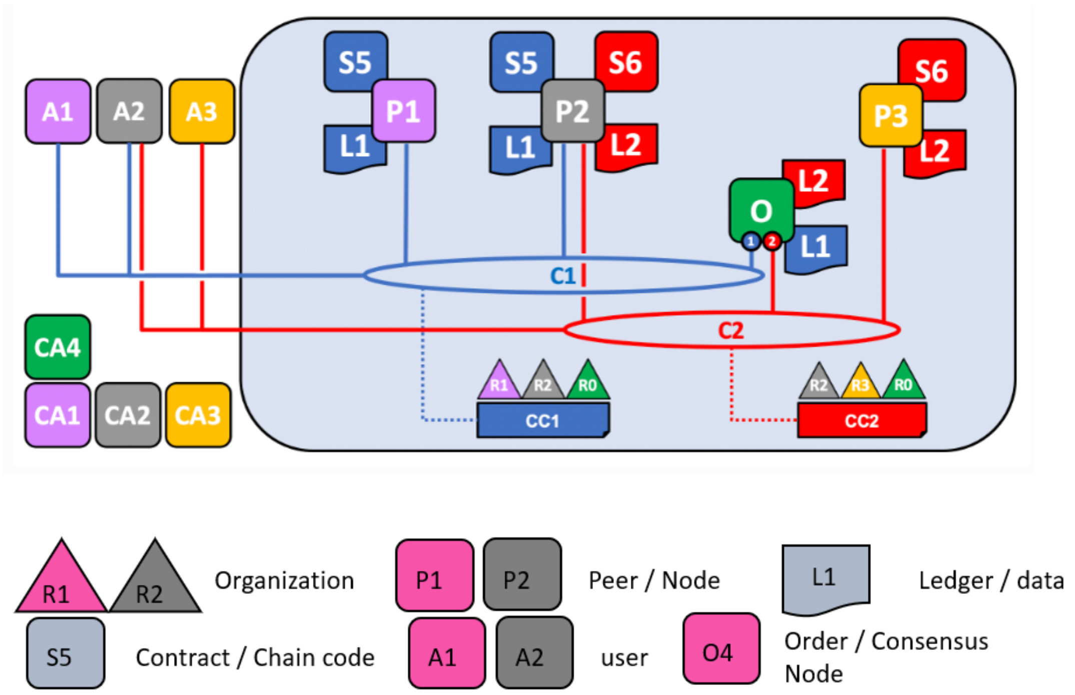


Fig. 2.22 Channel, organization, and ledger

There are three steps to create the channel

Step one: Generate the genesis block of the channel

The process of creating a new channel begins by generating the genesis block for the channel that we will later submit to your orderer in the channel join request. Only one member needs to create the genesis block and it can be shared out of band with the other members on the channel who can inspect it to ensure they agree to the channel configuration and then used by each orderer in the ordering service.

Step two: Add the first user to the channel

After we create the genesis block, all the channel configurations are completed. We need to submit this block to the network and active the channel. the first ordering node that receives the "*osnadmin channel join*" command effectively "activates" the channel, though the channel is not fully operational until a quorum of consenters is established. The following diagram summarize this process:

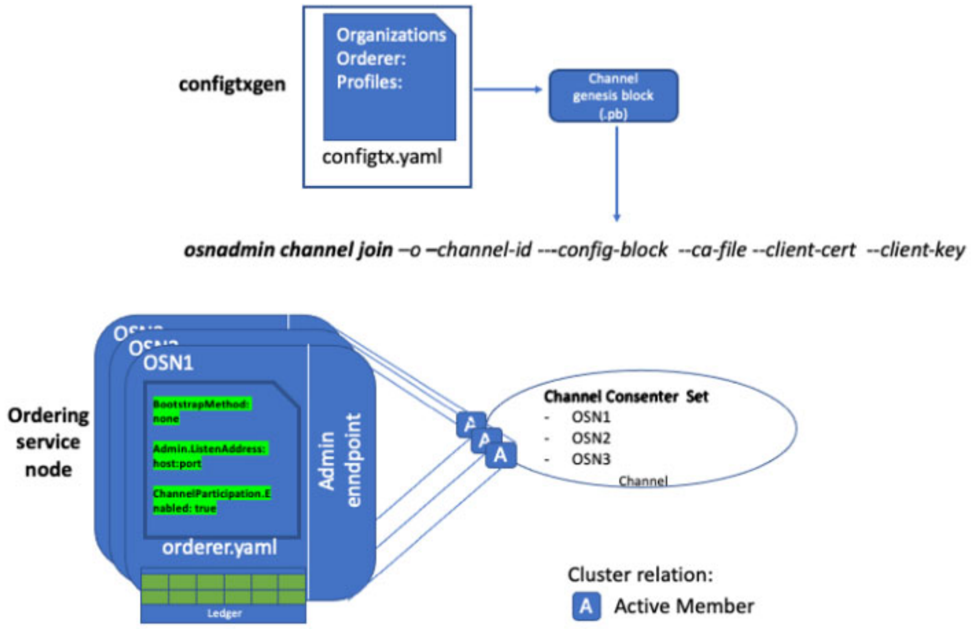


Fig. 2.23 Create and activate the channel

Step three: Join additional ordering nodes

Over time it may be necessary to add additional ordering nodes to the consenter set of a channel. Other organizations may want to contribute their own orderers to the cluster, or it might prove advantageous under production conditions to permit multiple orderers to go down for maintenance at the same time. This joining flow is same as step two.

2.3.3.4 Chaincode Configuration

End users interact with the blockchain ledger by invoking smart contracts. In Hyperledger Fabric, smart contracts are deployed in packages referred to as chaincode. Organizations that want to validate transactions or query the ledger need to install a chaincode on their peers. After a chaincode has been installed on the peers joined to a channel, channel members can deploy the chaincode to the channel and use the smart contracts in the chaincode to create or update assets on the channel ledger.

All of our chaincode are written in Golang language. After completing our chaincode, there are four steps to activate the chaincode in the channel.

Step one: Package the smart contract

Before install the chaincode, we need to packet our source code into installable packet. In terms of Golang language, we need to use a Go module to install the chaincode dependencies. The dependencies are listed in a *go.mod* file. The *go.mod* file imports the Fabric contract API into the smart contract package. And then, we use *go vendor* command to download all of this packet. After we download the dependency, we can packet the source code into a installable *tar.gz* file.

Step two: Install the chaincode package

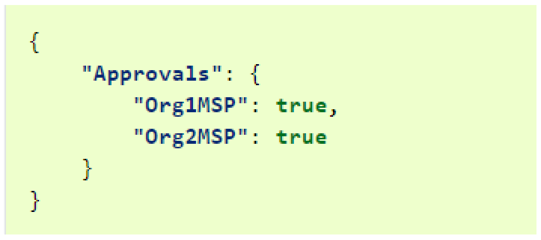
After we package the asset-transfer (basic) smart contract, we can install the chaincode on our peers. The chaincode needs to be installed on every peer that will endorse a transaction. Because we are going to set the endorsement policy to require endorsements from our organization, we need to install the chaincode on the peers operated by admi. Once the chaincode is installed successfully, we can get its packet ID and label.

Step three: Approve a chaincode definition

After we install the chaincode package, we need to approve a chaincode definition for the organization. The definition includes the important parameters of chaincode governance such as the name, version, and the chaincode endorsement policy. By default, this policy requires that a majority of channel members need to approve a chaincode before it can be used on a channel.

If an organization has installed the chaincode on their peer, they need to include the package ID in the chaincode definition approved by their organization. The package ID is used to associate the chaincode installed on a peer with an approved chaincode definition and allows an organization to use the chaincode to endorse transactions.

After all of the organization approve this chaincode, yhere will be an approved badge on the status, which is shown as follow figure:



```
{
    "Approvals": {
        "Org1MSP": true,
        "Org2MSP": true
    }
}
```

Fig. 2.24 The status of chaincode

Step four: Committing the chaincode definition to the channel

After a sufficient number of organizations have approved a chaincode definition, one organization can commit the chaincode definition to the channel. If a majority of channel members have approved the definition, the commit transaction will be successful, and the parameters agreed to in the chaincode definition will be implemented on the channel.

2.3.4 Develop the Chaincode (Smart Contract)

Smart Contracts are the programs in the fabric-network that contain the business logic. It describes the life cycle of an asset in the global economy. As a result, the business logic included within the smart contracts regulates all transactions.

In our project, we devised three smart contracts for the SAC and DDQN program. The data store in the contract can be fetched by all the base stations. The structure is shown as follows:

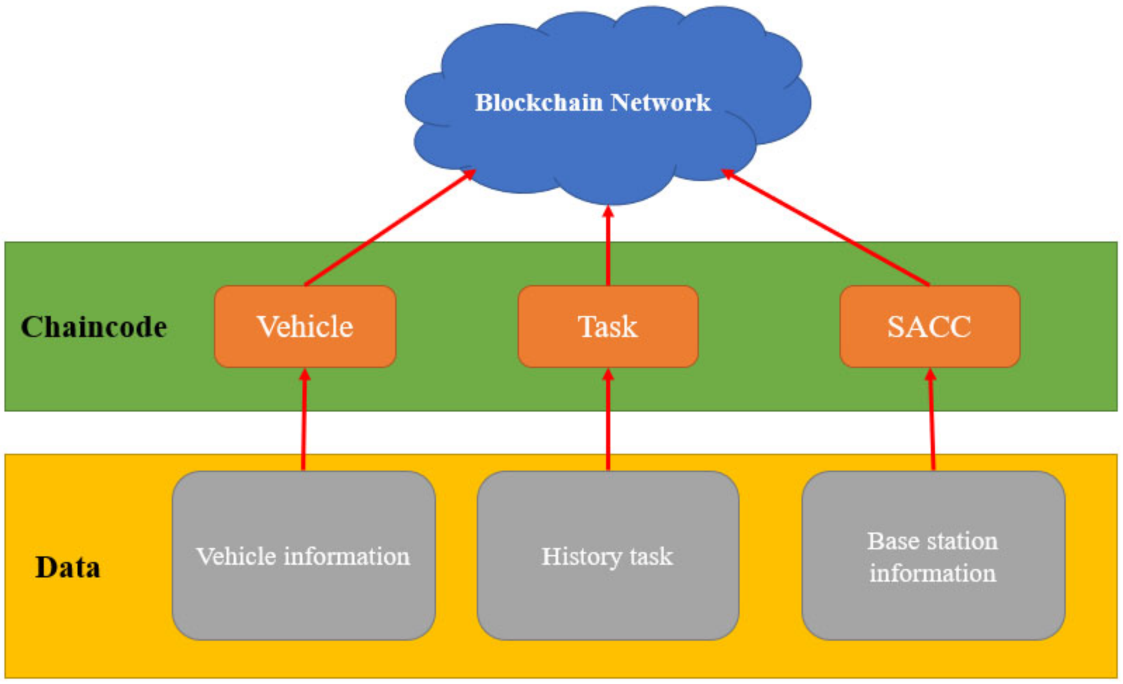


Fig. 2.25 Chaincode and data

2.3.4.1 Chaincode: Vehicle

This chaincode is stored the vehicle’s historical data, contain the task completion ratio and the reliability of each vehicle. After the base station allocates the service vehicle for the task vehicle to offload the task, the service vehicle needs to send the task related update to base station so that when the base station runs the SAC algorithm next time it must have the service vehicle information for example what percent of the task was previously completed by a particular service vehicle or how reliable the service vehicle is for a particular type of task. Hence, the service vehicle will update this information on the blockchain as soon as a task is completed through the vehicle chaincode. Whenever the base station trains the SAC algorithm with the previous vehicle information it will consider this past information for making decision. There are three columns in this chaincode:

```
ID                                                  string  `json:"id"`
RATIO                                               float64 `json:"completion_ratio"`
RELIABILITY                                         float64 `json:"reliability"`
```

Fig. 2.26 Format of chaincode “vehicle”

2.3.4.2 Chaincode: Task

This chaincode is designed for the DDQN program. When the task vehicle sends the offloading request to the base station, the base station will run the SAC algorithm as well as DDQN algorithm to select the service vehicle alongside the anchor node. The information after the computation from the service vehicle will then be sent to the base station. The DDQN will give some base station related information and submit the transaction via the task blockchain API. The data format is shown as below:

```
// BaseStation's information
type BaseStation struct {
    ID                                              string  `json:"id"`
    OFFLOAD_VEHICLE_ID                              float64 `json:"offload_vehicle_id"`
    SERVICE_VEHICLE_ID                              float64 `json:"service_vehicle_id"`
    ALLOCATION_BASESTATION_ID                       float64 `json:"allocation_basestation_id"`
    DELAY                                           float64 `json:"delay"`
    DONE_STATUS                                     float64 `json:"done_status"`
    VEHICLE_DENSITY                                 string  `json:"vehicle_density"`

}
```

Fig. 2.27 Format of chaincode “Task”

2.3.4.3 Chaincode: sacc

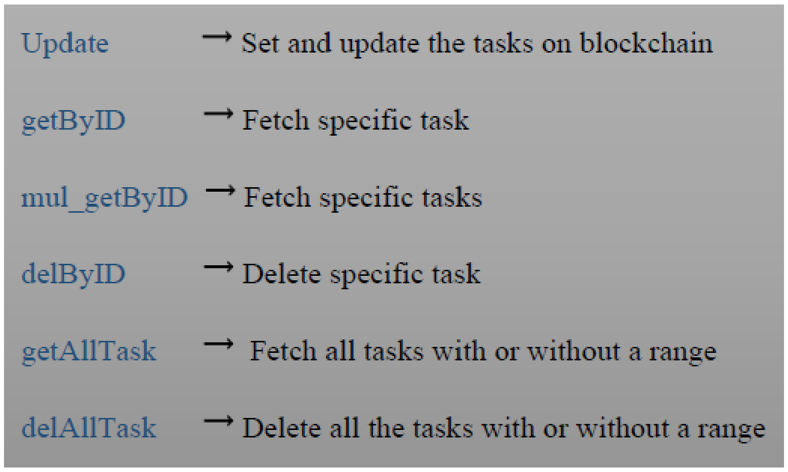
This chaincode is stored the basic information of base station. There are in total 4 base stations and similarly 4 nodes of the blockchain. Among which the DDQN has to choose which base station is the optimal node for the transaction submission. Hence, real time system computation resource has to be known to DDQN. For that reason, two of the base stations are continuously sending their CPU utilization power as well as the RAM information for every 5 seconds on the blockchain. During the DDQN training stage, the information of those two base stations will be fetched from blockchain continuously for real time selection. The data format of sacc is shown as follows:

```
type BaseStation struct {
    ID                                          string  `json:"id"`
    CPU_UTIL                                    float64 `json: "cpu_util"`
    TOTAL_MEMORY                                float64 `json:"total_memory"`
    FREE_MEMORY                                 float64 `json:"free_memory"`
}
```

Fig. 2.28 Format of chaincode “sacc”

2.3.5 Chaincode Execution

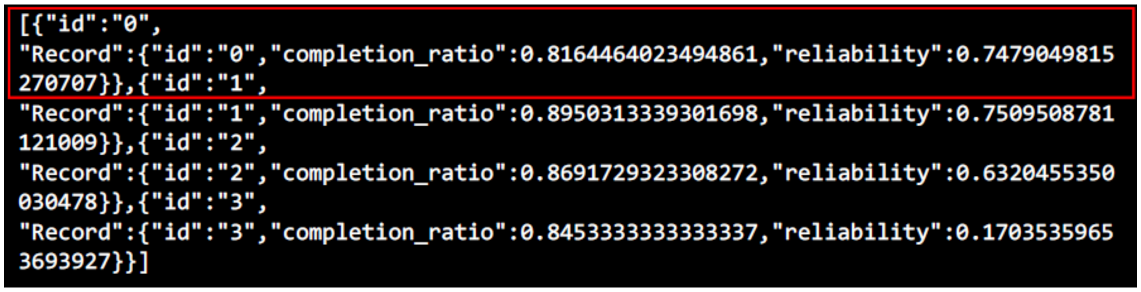
After the chaincode definition has been committed to a channel, the chaincode will start on the peers joined to the channel where the chaincode was installed. The asset-transfer (basic) chaincode is now ready to be invoked by client applications. In our custom chaincode, we devised four functions to update block:



Update	→ Set and update the tasks on blockchain
getByID	→ Fetch specific task
mul_getByID	→ Fetch specific tasks
delByID	→ Delete specific task
getAllTask	→ Fetch all tasks with or without a range
delAllTask	→ Delete all the tasks with or without a range

Fig. 2.29 The function of chain code

In the command line, we can use *Invoke* command to call these functions. Because all of the Fabric commands can only execute in command line, all of the data output to the command line in text format. In the python program, we have to split them and store in different variables. So, we decide to convert all the data into *json* format so that we can read them in other python program. Our output data is shown as below figure:



```
[{"id":"0",
"Record":{"id":"0","completion_ratio":0.8164464023494861,"reliability":0.7479049815
270707}},{"id":"1",
"Record":{"id":"1","completion_ratio":0.8950313339301698,"reliability":0.7509508781
121009}},{"id":"2",
"Record":{"id":"2","completion_ratio":0.8691729323308272,"reliability":0.6320455350
030478}},{"id":"3",
"Record":{"id":"3","completion_ratio":0.8453333333333337,"reliability":0.1703535965
3693927}}]
```

Fig. 2.30 Query data from Blockchain

2.3.6 Anchor Node Configuration

Anchor peers are used by gossip to make sure peers in different organizations know about each other. When a configuration block that contains an update to the anchor peers is committed, peers reach out to the anchor peers and learn from them about all of the peers known to the anchor peer(s). Once at least one peer from each organization has contacted an anchor peer, the anchor peer learns about every peer in the channel. Since gossip communication is constant, and because peers always ask to be told about the existence of any peer they don't know about, a common view of membership can be established for a channel.

3. The Integrated System

3.1 The Entire System

In our experiment, we use two computers as our base stations, three RaspberryPis as our vehicles. Besides, we use SUMO to simulate the vehicle trace and the urban road environment, obstruct. We also import SUMO map to the Omnet to simulate the V2V transmission rate. The vehicle selection algorithm and the anchor node selection algorithm are running on the base station. The entire process is shown as Fig 3.1

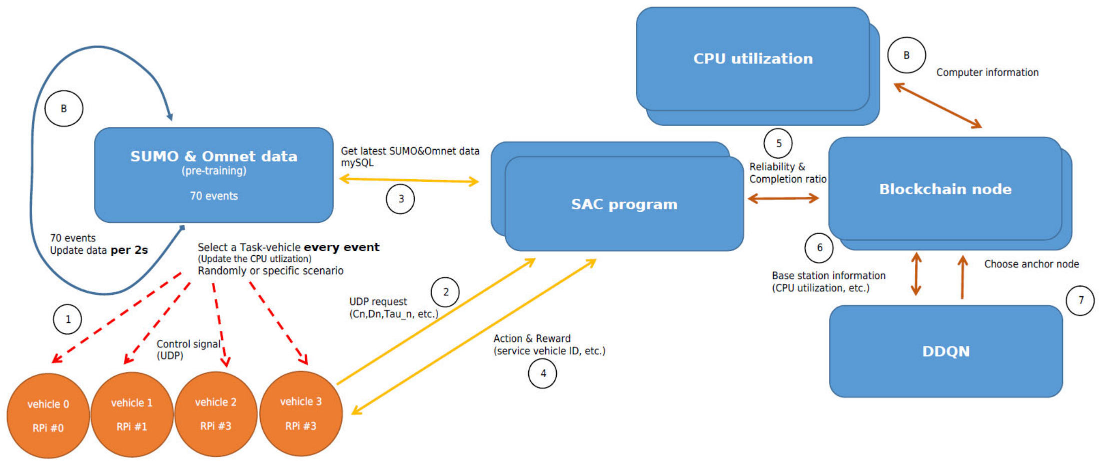


Fig. 3.1 VEC task offloading flow

Our SUMO and Omnet program are running background, they will update the simulation to the local SQL database. Other programs can also access this database. Fig 3.1 shows that there are 7 phases:

1) SUMO program update the data base per 2 seconds. After it update the data, it will select a RPi as our task vehicle based on the scenario designed before. Then, it will send the command packet to the RPi.

2) The task vehicle (RPi) will generate a random task, estimate the necessary parameters and send them to the SAC program.

3) Once the SAC program receiving the request packet, it will fetch the SUMO data from local SQL database. According to these parameters, SAC program could apply its policies to select the optimal service vehicle.

4) SAC program sends the V_S selection results to RPi via UDP protocol. Once the task vehicle gets the service vehicle ID, it could allocate the task to the V_S. Then, it will return the task results to the SAC program, so that SAC could get reward or penalty.

5) The SAC program will upload the reliability of the vehicle and the task completion ratio to Blockchain network so that other base station can fetch this information.
6) DDQN algorithm will monitor the base stations' information, if there are any tasks completed, DDQN will update it state space from the Blockchain and select a base station as the anchor node.
7) DDQN program will call the anchor node selection API to activate the new anchor node.

The system topology is shown as Fig. 3.2:

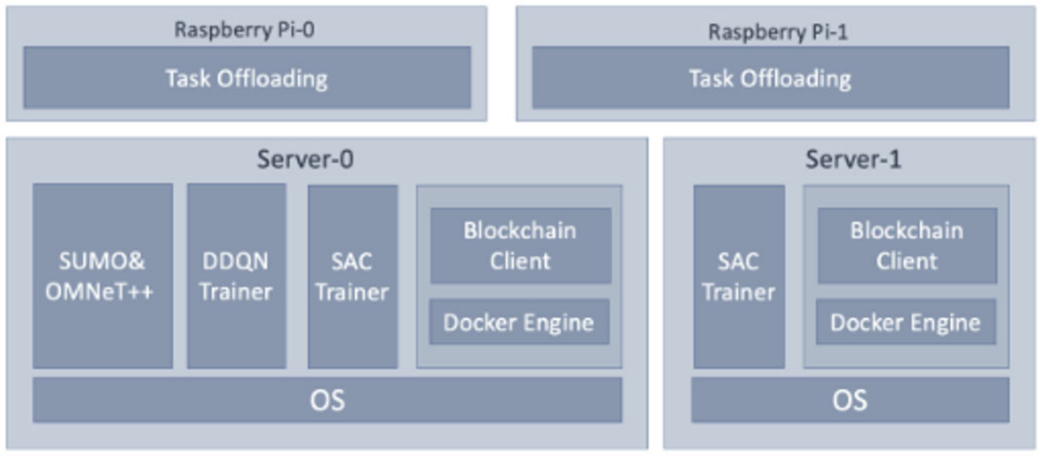


Fig. 3.2 Entire system topology

3.2 SAC Algorithm

The task offloading delay is defined in section 2.2.3.1. After defining the offloading delay, the reward function can be correlated and other parameters which required in this project can also be deduced. Before going into the expression of rewards, it is necessary to be introduced that how the utility of a computational task is calculated. The utility of a computational task is directly related to the offloading delay of that task. The lower the delay is, the higher the utility will be. Besides, if the offloading delay exceeds the maximum tolerance delay, which means that the task allocation is uncompleted, the utility should a minus value or a penalty. Therefore, the utility of a computation task can be deduced as a function of offloading delay, as shown below:

$$U_n = \begin{cases} \log(1 + T_n - t_n) & t_n \leq T_n \\ -\Lambda & t_n > T_n \end{cases}$$

where T_n is the maximum tolerance delay and Λ is the penalty of offloading failure. Besides, when a task vehicle offloads a task to a service vehicle, it needs to pay the service vehicle the service price, and it can be assumed that the service price of performing a computational task in a service vehicle is proportional to the computation size of the task. In the practical application scenario, one may wish to pay the service price as low as possible. Therefore, the utility for offloading a task should also be related to the service price. Consequently, the final utility becomes:

$$U_n^f = U_n - p_n C_n$$

where p_n denotes the unit price and $p_n C_n$ represents the service price. Then the reward can be deduced based on this final utility. To calculate the reward used to do training on the algorithm, authors of reference paper use the following formula:

$$R_t = \sum_{s=1}^{S} x_n^s U_n^f$$

where U_n^f is the utility of V_t for offloading a task, and $x_n^s = 1$ means that a service vehicle V_s is selected to execute a computational task. But in this project, there is only one service vehicle being selected every time, and the cumulative rewards will be considered by the algorithm. Therefore, the above reward formula can be simplified as following:

$$R_t = U_n^f$$

In addition to reward, completion ratio and reliability of a vehicle are two parameters that determines the probability of being selected as the service vehicle for offloading tasks. Besides, when this base station works with other parts of this project in an integrated way, the performance of a service vehicle needs to be accessible for other base stations so that they can choose service vehicles more efficiently. In this case, uploading the completion ratio and reliability of a service vehicle to database is a promising solution for the base station to share these two parameters.

To calculate completion ratio and reliability, a normalized utility of an offloading task can firstly be defined as follows:

$$\widetilde{U_n} = \frac{\log(1 + T_n - t_n)}{\log(1 + T_n)}$$

where $\log(1 + T_n)$ represents the maximum utility of task. After a service vehicle completes an offloading task, the computation efficiency of the service vehicle is updated by

$$\rho_s = (1 - \omega_1)\rho_s' + \omega_1 \widetilde{U_n} \qquad \omega_1 \in (0, 1)$$

where ρ_s' represents the previous computation efficiency of the service vehicle, which can be set as zero at the first step of training. Then the completion ratio of the service vehicle is updated by

$$\mu_s = \frac{N_s * \mu_s' + 1(t_n \leq T_n)}{N_s + 1}$$

where N_s represents the total number of received offloading tasks in the service vehicle and

μ_s' represents the previous completion ratio, which can also be set as zero at the first step of training like ρ_s'. Consequently, the reliability of the service vehicle can be defined as

$$\eta_s = \omega_2 \rho_s + (1 - \omega_2)\mu_s \qquad \omega_2 \in (0, 1)$$

It can be recognized that there are two weight factors in above formulas, ω_1 and ω_2. After testing, it can be found that 0.8 could be a reasonable value for both of factors. As a result, if a service vehicle cannot allocate appropriate computing resource for an offloading task according to the service price, the reliability of the service vehicle will be reduced.

Unlike action space which simply contains actions, observation space is constructed by more detailed parameters about vehicle, environment, and task. According to the reference paper, the observation (state) space of the system at time *t* can be defined as

$$\boldsymbol{s_t} = [F_1(t), \dots, F_s(t), \dots, F_S(t); \gamma_1(t), \dots, \gamma_s(t), \dots, \gamma_S(t); l_1(t), \dots, l_s(t), \dots, l_S(t);$$
$$\eta_1(t), \dots, \eta_s(t), \dots, \eta_S(t); D(t), C(t), T(t)]$$

where $F_s(t)$ represents the available computing resource in V_s, $\gamma_s(t)$ is the signal-to-noise ratio (SNR) of the V2V link between V_t and V_s, $l_s(t)$ denotes the estimated V2V link duration between V_t and V_s, $\eta_s(t)$ is the reliability of V_s, $D(t)$, $C(t)$, and $T(t)$ are the data size, computation size, and deadline of the offloading task at time *t*, respectively. However, due the introduction of road traffic simulation, the second term of observation space, SNR, can be replaced by the simulated data transmission rate. Since these two parameters both describe the transmission performance which depends on the real environment. Besides, as what demonstrated in previous paragraph, the total number of vehicles is four. In this case, the expression of observation space can be simplified as

$$\boldsymbol{s_t} = [F_0(t), F_1(t), F_2(t), F_3(t); rt_0(t), rt_1(t), rt_2(t), rt_3(t); l_0(t), l_1(t), l_2(t), l_3(t);$$
$$\eta_0(t), \eta_1(t), \eta_2(t), \eta_3(t); D(t), C(t), T(t)]$$

where $rt(t)$ represents the transmission rate between two vehicles. Based on the demonstrated action space and observation space above, the diagram of SAC algorithm for reinforcement learning in service-vehicle selection can be created, and it is shown in Fig. 3.3.

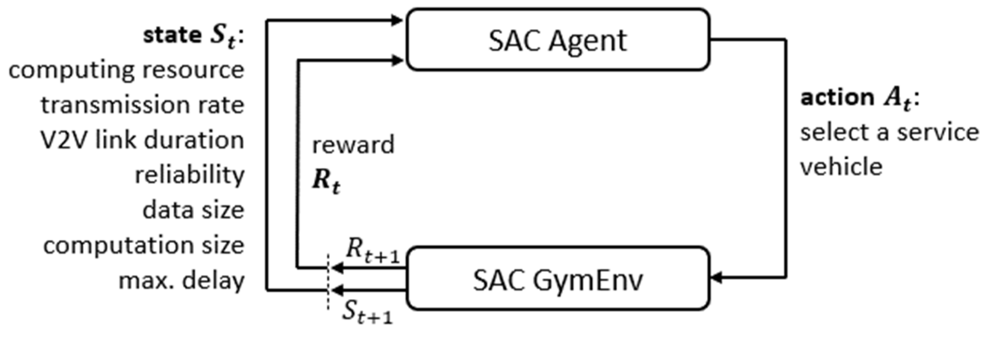


Fig. 3.3 The diagram of observation (state) space and action space

The SAC training results is shown as follows:

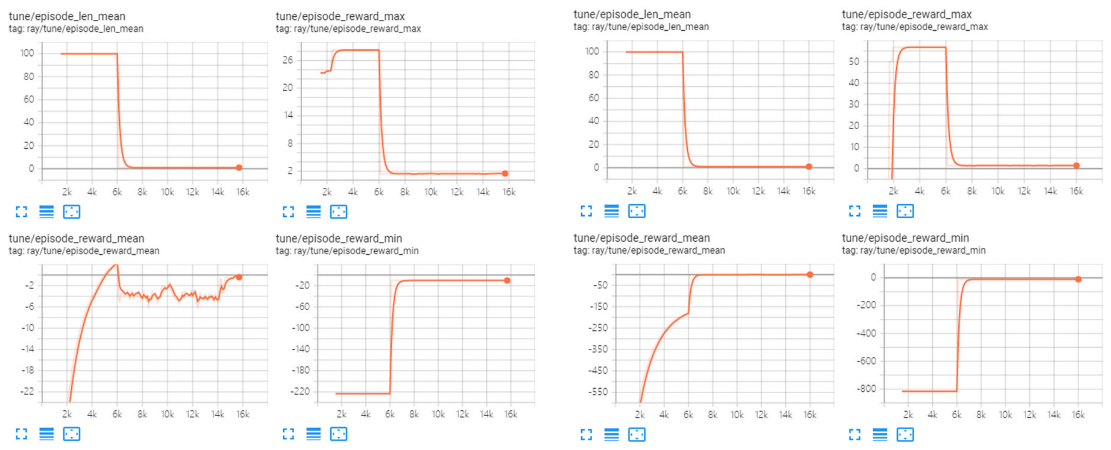


Fig. 3.4 (a)Base station 0 training results (b)Base station 1 training results

As shown in Fig. 3.4, the training results of the two base stations are illustrated, and the impact from different task allocation fault penalty are explored. The upper-left graphs in the four figures indicates that when the iteration number is smaller than 6000, a training episode contains 100 training steps, which means that the pre-training process is executed, and after 6000 iterations, there is only one step in one training episode. The reason is that in practical allocation part, the data size, computation size, transmission rate and computing resource is required to be updated every time the task vehicle sends request. As shown in the lower-left graphs of the above four figures, the episode_reward_mean graphs, it can be observed that the rewards earned by SAC agent all converge to about zero, which indicates the base stations can select the expected service vehicle after training can keep the rewards around the certain level. For the reason why the rewards are around zero stably, it is because the base station can get a reward of about 0.2 to 0.5 after every successful task allocation.

3.3 Co-simulation OMNeT++ with SUMO

The figure below shows importing SUMO maps in OMNeT++ and generating network scenarios. In order to better visualize the results of the algorithm, we design three specific scenarios to demonstrate the rationality of the SAC and DDQN algorithms.

3.3.1 Scenario I:

In this event, all vehicles have just been generated from the SUMO map. We select vehicle 0 as the task vehicle and other vehicles as service vehicles.

In order to demonstrate the task allocation method of SAC, we set the computing resource occupancy rate of vehicle 0, vehicle 2 and vehicle 3 to 90% at this time, and the computing resource occupancy rate of vehicle 1 is 3%.

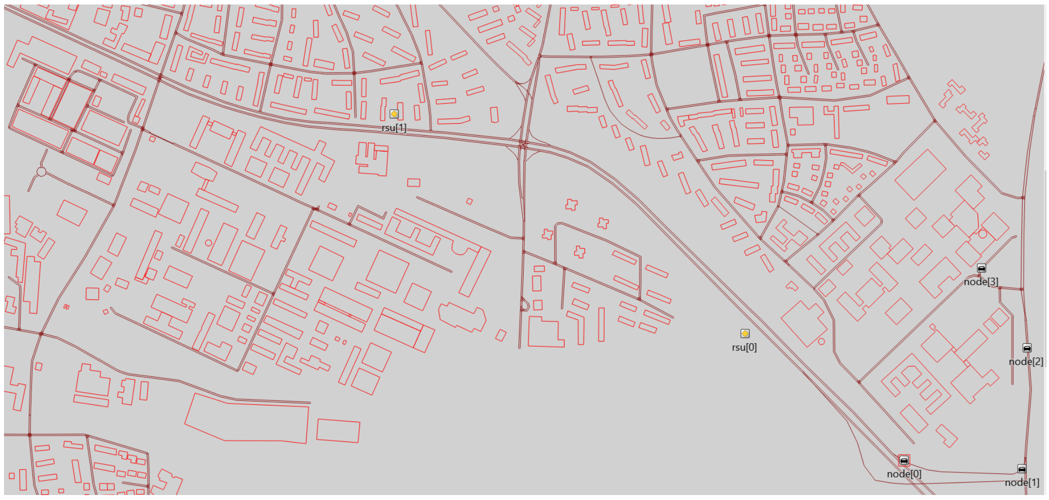


Fig. 3.5 Scenario 1: vehicles generate at beginning

Obviously, because the computing resource occupancy rate of vehicle 2 and vehicle 3 is high at this time, vehicle 1 is very low. Therefore, the result of the SAC algorithm should select vehicle 1 as the service vehicle. In addition, since the fleet is far from base station 0 and base station 1 at this time, but base station 0 is closer in comparison, the result of the DDQN algorithm should be more likely to select base station 0.

3.3.2 Scenario II:

In this event, all vehicles are concentrated near base station 0. We select vehicle 1 as the task vehicle and the other vehicles as service vehicles. We set the computing resource occupancy rate of vehicle 0, vehicle 1 and vehicle 3 to be high at this time, above 85%, and the computing resource occupancy rate of vehicle 2 is low.

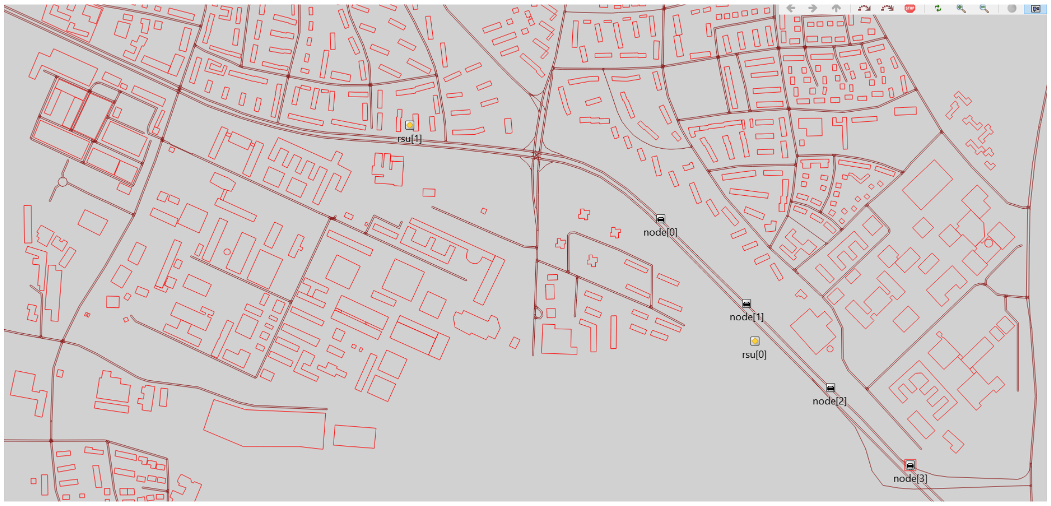


Fig. 3.6 Scenario II: vehicles near base station [0]

Since the computing resource occupancy rate of service vehicle 0 and vehicle 3 is high at this time, vehicle 2 is very low. Therefore, the result of the SAC algorithm should select vehicle

2 as the service vehicle. In addition, at this time, the fleet is concentrated near base station 0, that is, the traffic density is larger, so the result of the DDQN algorithm should be more likely to select base station 1 with a smaller vehicle density.

3.3.3 Scenario III:

In this event, all vehicles are concentrated near base station 1. We selected vehicle 3 as the mission vehicle and the other vehicles as service vehicles.

We set the computing resource occupancy rate of vehicle 0, vehicle 2 and vehicle 3 to be high at this time, above 84%, and the computing resource occupancy rate of vehicle 1 is low.

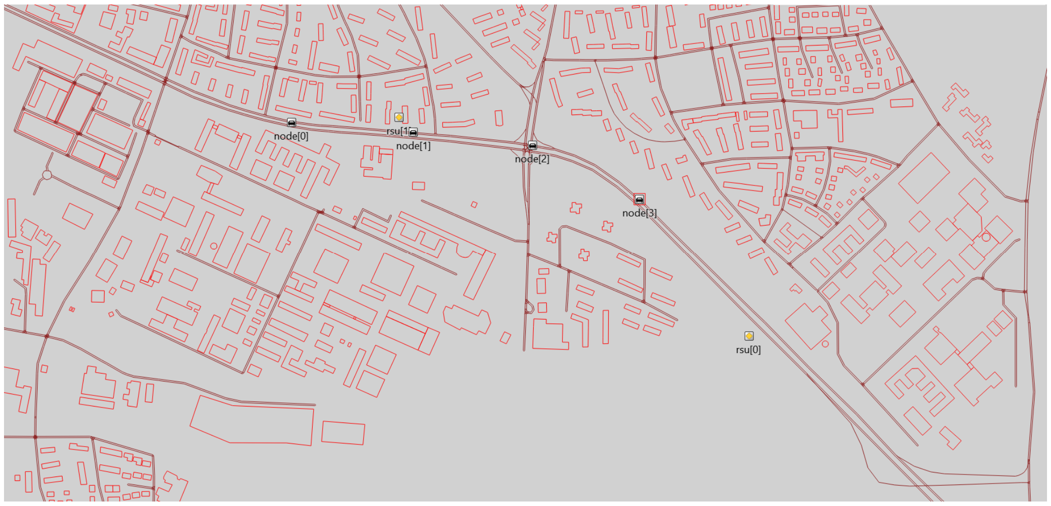


Fig. 3.7 Scenario III: vehicles near base station [1]

3.3.4 Other Scenario:

In addition to the above three typical scenarios, under other events, the computing resource utilization of the four vehicles is at a low level. In other words, when any vehicle issues a task offloading command, it is not constrained by computing resources, but pays more attention to the V2V communication transmission rate and vehicle density.

In conclusion, our purpose of setting three typical scenarios is to verify the rationality of the SAC algorithm and the DDQN algorithm by amplifying the advantages of the dominant service vehicle. The following table shows the computing resource occupancy rate of different vehicle in corresponding scenarios under different events.

3.4 V2V Transfer Communication Rate

In our project, we assume that all the vehicles use 802.11p to communicate. According to the channel model and 802.11p protocol we selected in the simulation, the theoretical maximum transmission speed is 6 Mbps, but it is difficult to achieve this transmission speed due to channel fading, signal interference and other reasons in the actual environment.

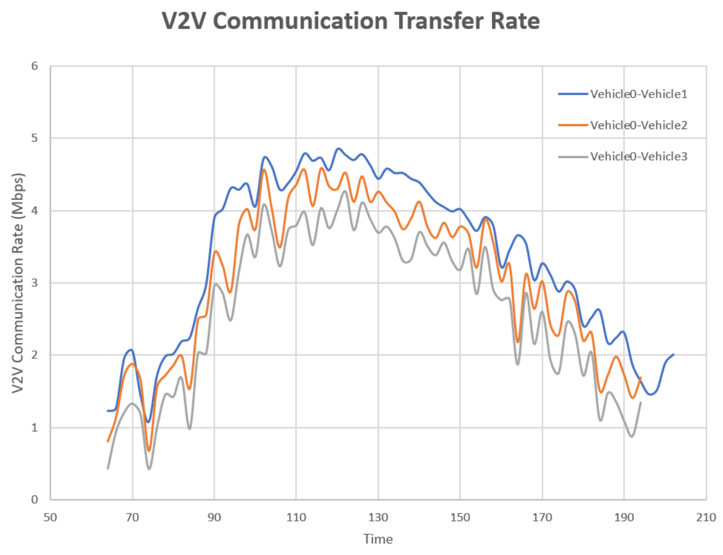


Fig. 3.8 V2V Communication transfer rate

It is not difficult to observe the above picture, in general, the farther the distance between the vehicles, the slower the transmission speed. When the transmission between vehicles is disturbed by obstacles such as buildings, even if the distance between the vehicles is very close, the transmission speed will become slower.

In our simulation process, the overall V2V communication transmission rate is a trend of rising first and then falling. The highest transmission rate is mostly around 120s to 130s. At this time, the vehicle is driving on a straight highway, and there is no building block between the vehicles. As the vehicle continues to travel along the road, the presence of buildings affects the transmission speed between vehicles, so that the transmission rate starts to slow down.

3.5 DDQN Algorithm

In this part we use DDQN algorithm to select the consequences node. Because Fabric use anchor node to broadcast the blocks, so we decided to use anchor node as our consensus nodes. The DDQN is based on gym environment, because the integrated system is a real-time simulation, our GymEnv must be linked to the upstream task database, and our DDQN trainer's activities must be linked to the downstream blockchain. As we discussed in the previous chapter, updating the BS status is mostly dependent on the task database. And in the integration system, we store all produced tasks on the blockchain.

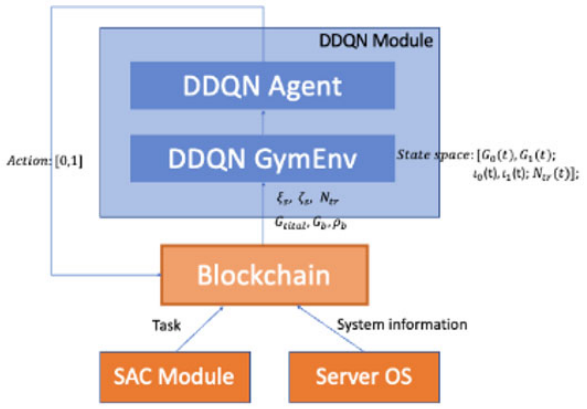


Fig. 3.9 DDQN module

The Figure 3.8 shows the flowchart of the DDQN Module. First of all, Since SAC Module will allocate tasks to the vehicles and store the task record to the base station as long as one task is succeeded or fail, DDQN Module will access to the blockchain and retrieve the new inserted data from it. And the new inserted tasks include information including id, offload vehicle id, service vehicle id, allocation BS id, done status, vehicle density delay.

The DDQN will then update the completion ratio and computation efficiency of the connected BS in line with the tasks given, so improving the BS's reliability. The formulas

$$\widetilde{U_n} = \frac{log(1 + \tau_n - t_n)}{log(1 + \tau)}$$

$$\xi_s = (1 - \omega_1)\xi' + \omega_1 U_n$$

$$\zeta_s = \frac{N_s \zeta_s' + 1(t_n \leq \ \tau_n)}{N_s + 1}$$

$$\iota_s = \omega_2 \xi_s + (1 - \omega_2)\zeta_s \,, \omega \in (0,1)$$

have been given to compute these states. At the same time, the servers will update the real-time computing resource information to the blockchain. The available computational resources can be calculated. The state space of this model is comprised of the information provided above, as well as the number of transactions generated within a certain time period, and can be described as:

$$\mathrm{S_t = [G_0(t), G_1(t); \iota_0(t), \iota_1(t); N_{tr}(t)]}$$

The above process constitutes the DDQN environment of integration system. The only object that the environment should depend on is the blockchain, since all significant information is stored and updated there and the DDQN Module only need to interact with the blockchain via docker client.

Besides the DDQN environment, another part of DDQN Module is the global agent to apply

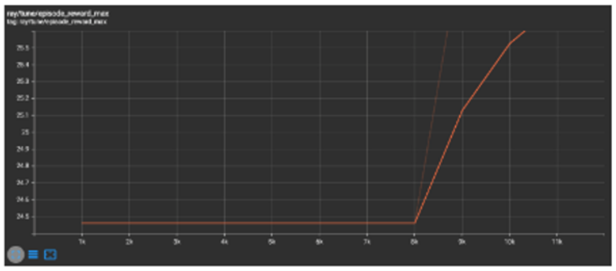
DDQN trainer. The agent play the role of training the model and output the action to the environment. In the integration system, our action is represented as:

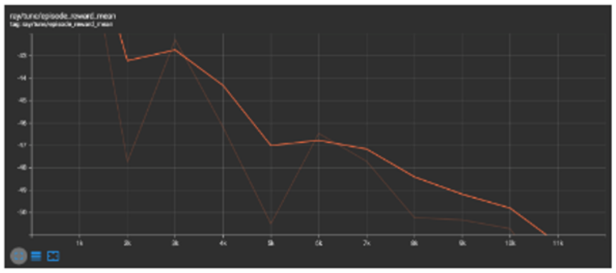
$$a_t = [0,1]$$

Since we have two blockchain nodes, and each node is deployed on a server, the action space represents the identity of the blockchain nodes.

Every iteration, the DDQN agent will make a decision according to the states of the DDQN environment. And when a complete epoch with 100 iteration finishes, the agent will reset the environment. However, considering the continuity of training, we do not reset all the state to the original state when a new epoch starts.

The DDQN agent's goal is to maximize the reward of an action to the environment. To make it, the agent will keep some data that has been influenced by previous incentives and will use this data to construct better rules in the future.


(a) max reward


(b) mean reward

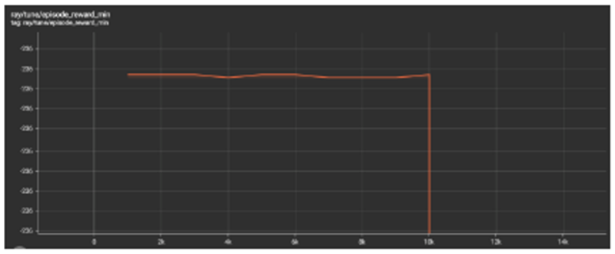

(c) min reward

Fig. 3.10 Results of DDQN

Figure 3.10 shows the result of DDQN training of the integration system, the total training epoch is 100. But from the perspective of reward, the final result did not reach convergence.

4. Exploration

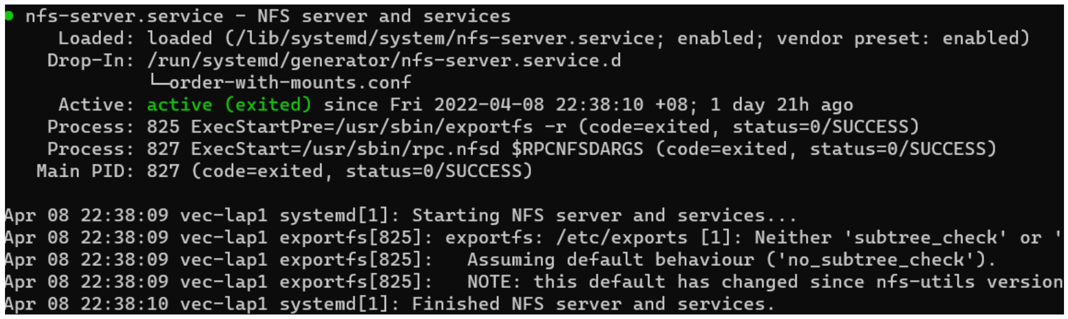
In this part, I will discuss some platform, tools, and software I explored in this project. But they are not used in our integrated part.

Because blockchain is different to use, at first, I try to use some other platform or tools to create an exchange space so that every base station could upload or fetch data together. I have tried three different ways to achieve the goals.

4.1 NFS Server / Client

Network File System (NFS) is a distributed file system. This protocol allows a user on a client computer to access files over a network in the same way they would access a local storage file. Because it is an open standard, anyone can implement the protocol. NFS started in-system as an experiment, but the second version was publicly released after the initial success.

To access data stored on another machine (i.e., a server) the server would implement NFS daemon processes to make data available to clients. The server administrator determines what to make available and ensures it can recognize validated clients. Fig 4.1 shows the configuration of the NFS server.



```
● nfs-server.service - NFS server and services
     Loaded: loaded (/lib/systemd/system/nfs-server.service; enabled; vendor preset: enabled)
    Drop-In: /run/systemd/generator/nfs-server.service.d
             └─order-with-mounts.conf
     Active: active (exited) since Fri 2022-04-08 22:38:10 +08; 1 day 21h ago
    Process: 825 ExecStartPre=/usr/sbin/exportfs -r (code=exited, status=0/SUCCESS)
    Process: 827 ExecStart=/usr/sbin/rpc.nfsd $RPCNFSDARGS (code=exited, status=0/SUCCESS)
   Main PID: 827 (code=exited, status=0/SUCCESS)

Apr 08 22:38:09 vec-lap1 systemd[1]: Starting NFS server and services...
Apr 08 22:38:09 vec-lap1 exportfs[825]: exportfs: /etc/exports [1]: Neither 'subtree_check' or '
Apr 08 22:38:09 vec-lap1 exportfs[825]:   Assuming default behaviour ('no_subtree_check').
Apr 08 22:38:09 vec-lap1 exportfs[825]:   NOTE: this default has changed since nfs-utils version
Apr 08 22:38:10 vec-lap1 systemd[1]: Finished NFS server and services.
```

Fig. 4.1 NFS server configuration

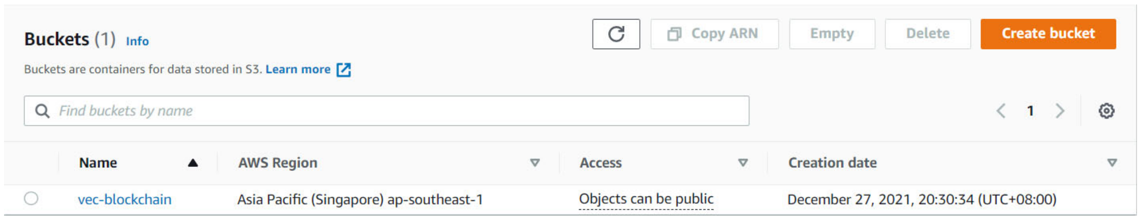
From the client's side, the machine requests access to exported data, typically by issuing a mount command. If successful, the client machine can then view and interact with the file systems within the decided parameters.

Once the client mounts the NFS folder, all the program on this machine can access the files in the folder as if they are local file. And because every one access the same file, all the user could get the latest data.

But NFS is only a network file system, it did not consider there are multiple user access and edit the same file. In our project, if there are many programs edit same file at same time, it may lead to conflict.

4.2 Google / Amazon Cloud Storage

The cloud storage is another choice to exchange data. This platform is similar to NFS server. Compare with NFS, we don't to install or configure our own server, Google and Amazon nearly complete everything. We only need to create a new instance and manage the access authority. Fig 4.2 is the Amazon bucket.



Buckets (1) Info

Buckets are containers for data stored in S3. Learn more

Name	AWS Region	Access	Creation date
vec-blockchain	Asia Pacific (Singapore) ap-southeast-1	Objects can be public	December 27, 2021, 20:30:34 (UTC+08:00)

Fig. 4.2 Amazon S3 bucket

Besides, both *numpy* and *pandas* are supporting these platforms, we can access the cloud storage in python code via its public address.

The disadvantage is same as NFS, when multiple users are editing the same file, the conflict may occur.

4.3 MySQL Database Server

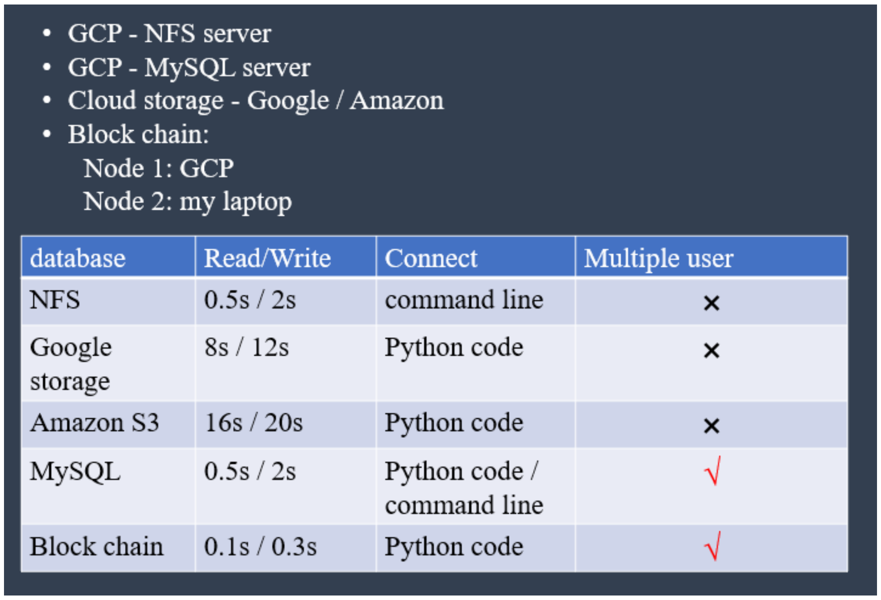
Unlike NFS and cloud storage, MySQL is designed for multiple users. Nowadays, we use relational database management systems (RDBMS) to store and manage huge volume of data. This is called relational database because all the data is stored into different tables and relations are established using primary keys or other keys known as Foreign Keys.

MySQL has stand-alone clients that allow users to interact directly with a MySQL database using SQL, but more often, MySQL is used with other programs to implement applications that need relational database capability. We first use Google cloud server as our MySQL server, and our own computer could access this database via the username and password.

But Google cloud server is far from campus, the network latency is very high. The client has to cost long time to communicate with the server.

4.4 Comparison

Compare all of these methods, only MySQL could satisfy our requirements. But if the server far from the client, we have to cost too much time on communication. Their differences and some features is shown as follows:



- GCP - NFS server
- GCP - MySQL server
- Cloud storage - Google / Amazon
- Block chain:
  Node 1: GCP
  Node 2: my laptop

database	Read/Write	Connect	Multiple user
NFS	0.5s / 2s	command line	×
Google storage	8s / 12s	Python code	×
Amazon S3	16s / 20s	Python code	×
MySQL	0.5s / 2s	Python code / command line	√
Block chain	0.1s / 0.3s	Python code	√

Fig. 4.3 Comparison of different database

Fig 4.3 shows that, the blockchain is much quicker than other methods. It only needs to spend 0.3s to upload the data to the database. Besides, it also supports multiple users edit the same data at the same time because of its consequences algorithm. So we decided to use the block chain to exchange data in the end.

5. Conclusion

In this project, we implemented a DRL based task offloading algorithm with the blockchain enabled VEC network. We use SUMO to generate the urban road map and vehicle traces. There are four cars running one the road, and two base stations on the roadside. Each vehicle corresponding to a raspberry pi. Then we import the map and traces into OMNeT++ to calculate the V2V transmission rate based on 802.11p protocol.

When our raspberry pi become the task vehicle, it will generate some random tasks and send an offloading request, based on UDP protocol, to the base station, SAC program. The SAC program will import the vehicles' information and SUMO & OMNeT++ data as its state space. Then, it applies its policies to select the optimal service vehicle. SAC program will return the ID of service vehicle to raspberry so that the task vehicle can allocate this task to the specific service vehicle. After the task is completed, V_t return the results to the SAC program. And SAC will get rewards or penalty according to the offloading result.

Besides, different base stations need to share the vehicle historical data, such as vehicle reliability, task completion ration, etc. We use blockchain based database (Hyperledger-Fabric) to exchange that information. Because the SAC program needs many computing resources, it may decrease speed of the blockchain consensus algorithm. So, we use DDQN program to select the consensus node. The information which DDQN needed is also shared in blockchain network. My contribution to this project comprises the following:

- Developed the "real" task offloading program based on Ray. The task vehicle could allocate the task to a specific service vehicle and get the results easily.
- Devised a UDP based application protocol, each part in our integrated part need to use this protocol to communicate. For example, SUMO send command packet to task vehicle, task vehicle sends offloading request to SAC program, etc.
- Set up the Blockchain network using the Hyperledger Fabric. I generate the TLS, MSP file for the network, and edit the docker containers' configurations file so that every base station could join the network and use it to exchange data.
- Designed some smart contracts for SAC algorithm and developed a python API so that we can use blockchain database in our python program.

In the end, we integrated all of our work and programs. The entire system is running well and archive our initial goals.

However, there are still many problems in our integrated system.

- When the base station is waiting for a task result, it cannot deal with other offloading requests. Although we didn't consider this problem in our project, but it may lead to the task offloading failure.

- We didn't consider the vehicles' energy consumption. It's also an important aspect of VEC. Because we use DC adapter give the raspberry pi power supply rather than battery, the vehicles are willing to contribute all their computing resources.
- We only have four vehicles and two base stations. Our SUMO scenario is too simple, it's hard to judge the effect of our algorithm.

Anyway, we introduce the blockchain based database into our project, the performance of the algorithm much better than those use other databases. And because we use the vehicle historical information in our algorithm, the task completion ration is much higher than other methods, which come to 95%

Through this project, I acquired a lot of knowledges that I have never exposed before like Linux system, Ray, blockchain, etc. I believe this knowledge could help me solve many similar problems in future.

Reference

1. J. Shi, J. Du, Y. Shen, J. Wang, J. Yuan and Z. Han, "DRL-Based V2V Computation Offloading for Blockchain-Enabled Vehicular Networks," in IEEE Transactions on Mobile Computing, doi: 10.1109/TMC.2022.3153346.

2. Z. Gao, M. Liwang, S. Hosseinalipour, H. Dai and X. Wang, "A Truthful Auction for Graph Job Allocation in Vehicular Cloud-assisted Networks," in IEEE Transactions on Mobile Computing, doi: 10.1109/TMC.2021.3059803.

3. J. Du, C. Jiang, H. Zhang, Y. Ren and M. Guizani, "Auction Design and Analysis for SDN-Based Traffic Offloading in Hybrid Satellite-Terrestrial Networks," in IEEE Journal on Selected Areas in Communications, vol. 36, no. 10, pp. 2202-2217, Oct. 2018, doi: 10.1109/JSAC.2018.2869717.

4. Haarnoja, Tuomas, Aurick Zhou, Pieter Abbeel, and Sergey Levine. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor." In International conference on machine learning, pp. 1861-1870. PMLR, 2018.

5. Androulaki, Elli, et al. "Hyperledger fabric: a distributed operating system for permissioned blockchains." Proceedings of the thirteenth EuroSys conference. 2018.

6. Moritz, Philipp, et al. "Ray: A distributed framework for emerging {AI} applications." 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 2018.