Tian edge node

# Summary

Edge nodes are the devices that run the titan edge program.

As the edge device of the titan network, the edge node is responsible for the last mile of the titan network content distribution process, delivering content to users with the shortest network distance.

Edge nodes run on one of the area network instances of titan, which is composed of multiple titan area networks.

Edge nodes are generally home devices, so the titan network can have numerous edge nodes, and use these edge node storage and bandwidth resources to build a huge resource pool of the titan network.

Each edge node pulls various asset data to its own storage according to the instructions of titan scheduler, and sends these data to users in response to their requests.

The Edge node gets the corresponding revenue based on the storage and bandwidth resources it contributes.

# The sense of Edge node

For example, a user in Shenzhen, China needs to pull data from source located in the US, which needs to be transmitted from the US to China via submarine fiber optic cable, then through China's national backbone network before finally reaching the user's device. During peak periods, the submarine cable and national backbone are busy resulting in high packet loss, high latency, and poor user download experience.

The goal of Edge nodes is to make the download speed stable and improve user experience by saving data to local storage and users pulling data from close edge nodes, avoiding backbone network and submarine fiber optic cable.

Based on the user's IP address information, the Titan network estimates the user's city and assigns the user to an edge node in the same city/region. Therefore, users pull data from edge nodes, and the data only need to pass through metro network (metro IP network/metro Ethernet), which has large bandwidth and low latency, and the download speed of users is guaranteed.

Most of the ADSL in Chinese homes are FTTH, fiber to the home, and the home bandwidth is large, and the uplink bandwidth generally reaches 20Mbps~50Mbps, and this uplink bandwidth utilization is very low, because home users generally use the downlink bandwidth. Therefore, edge nodes run in the home network and are able to make full use of the idle uplink bandwidth.

During the low peak period, edge nodes use the idle downlink bandwidth to pull asset data, and during the peak period, they use the idle uplink bandwidth to distribute asset data, so that the data can circulate within the MAN as much as possible without the need of cross-domain backbone network and submarine fiber optic cable, and reduce the load of backbone network and submarine fiber optic cable.

# Resource requirements for Edge nodes

The main resources of edge nodes are storage and bandwidth, storage resources are used to store asset data and bandwidth resources are used to pull asset data to users.

We expect each edge node to have more than 2TB of physical storage space.

In terms of bandwidth, the downlink bandwidth of the edge node is mainly used to pull asset data from source to local storage, and the uplink bandwidth of the edge node is mainly used to send the asset data to users. The downlink bandwidth of the home broadband network is generally much larger than the uplink bandwidth, so the bandwidth of the edge nodes is mainly the uplink bandwidth, and we expect that the uplink bandwidth of each edge node can reach more than 20Mbps.

# Disadvantages of Edge nodes and countermeasures

ly devices that run in a home environment. A home environment means that there are factors such as unstable network, unstable power supply, etc. that can cause the device to go offline.

For example, home networks in China are typically ADSL, and the IP addresses obtained through dial-up are temporary and may change every day. Nowadays, due to the large increase of mobile devices leading to insufficient IP addresses, the IP address obtained by the home network is not even the public IP, but the ISP's intranet IP address, and the public network is accessed through the ISP's NAT device.

Therefore, the possibility of edge nodes going offline is high. We avoid users from being unable to download the asset data when a single node fails by replicating the asset data to multiple edge nodes.

Titan scheduler estimates the expected reliability of an edge node based on its online and offline status. the reliability of the asset data can be evaluated based on the reliability of all the edge nodes that have copies of the asset data. If the reliability of the asset data falls below a certain threshold, titan scheduler will select more edge nodes to hold the copy of the asset data.

# Edge node contributions and benefits

Edge nodes pull asset data from source to their own storage devices, and after that, when nearby users need the same data, they pull it directly from edge nodes, which reduces the bandwidth burden of source on the one hand, and reduces the time consuming for users to get data on the other hand, improving user experience.

Therefore, edge nodes contribute storage, and bandwidth to titan network. titan network rewards to edge nodes are also divided into two types: one is storage based rewards, the larger the storage contributed by edge nodes, the more rewards; the other is bandwidth based rewards, the larger the bandwidth contributed by edge nodes, the more rewards.

According to the development plan of titan network, the reward model is divided into two phases. The first stage corresponds to the early stage of titan network development. At this stage, there is not enough asset data in the titan network, so the storage of edge nodes is not highly utilized, and there will not be too many users pulling data from the edge nodes in the network, so the bandwidth utilization of edge nodes is not high either. At this point, the titan network leases the storage and bandwidth of the edge nodes, but it cannot fully utilize these resources yet, so a spot-checking process is needed to ensure the effectiveness of the storage and bandwidth of the edge nodes. As long as the resources of the edge nodes pass the spot-checking verification, the titan network will issue rewards for the edge nodes.

The second stage corresponds to the mature stage of titan network development, when there is more asset data in the network and more users pull data from the edge nodes of the network. edge node storage contribution is mainly based on the amount of real stored asset data, of course, the larger the physical storage, the more asset data can be stored, and the more the reward will be. The bandwidth contribution, on the other hand, is based on the amount of data pulled by users from the edge nodes, as well as the rate. If no users pull data from the edge nodes, then the bandwidth contribution of the nodes will be small.

The reward model will continue to be optimized during the development of the titan network in order to more reasonably reflect the contribution of the edge nodes.

# Edge nodes join the network

Edge nodes need to join the titan network in order for their storage and bandwidth resources to be utilized by the titan network. For example, after an edge node joins the network, the titan scheduler will select it as the storage for asset data and allow users to pull asset data from it.

The process for an edge node to join the network is a two-step process. First, the edge node requests the titan locator server to get the address of the scheduler server for the edge node; then the edge node connects to the scheduler server and sends a login request signed with a private key.

When the Edge node asks the titan locator server, the latter can determine the titan scheduler to which the node belongs based on the ID of the edge node. When the Edge node asks the titan locator server, the latter can determine the titan scheduler to which the node belongs based on the ID of the edge node.

When the Edge node sends a request to the titan scheduler, the latter loads the information registered at the time of node registration from the database based on the node ID, and if the ID does not exist, the scheduler rejects the node login. Scheduler will use the public key provided by the node during registration to verify the login request.

## Node registration

The node manager first needs a titan account, which is a valid Filecoin wallet address that can be applied for through the titan website or through the lotus client.

After logging into the titan website with your account, you will enter the node registration page by selecting the node registration portal. This page mainly requires the user to make some choices, such as which area(city) the node plans to run in, the quota of resources the node plans to contribute, including storage quota and bandwidth quota, and so on. Then the registration page runs a local JS script to generate the node ID and key. Among them, the private key will let the user choose to save it, and the public key and node ID will be sent to the titan web server along with the user's selection information, which will be given to the titan scheduler, and the scheduler will complete the registration process.

Once the registration process is completed, the node will be bound to the account.

After that, the node manager needs to import the private key and node ID into the node's program, and the node program uses this information to log into the titan network.

## Node keepalive

Edge nodes need to periodically send live packets to the titan scheduler to keep the node online in the scheduler. If the edge node is not online, titan scheduler will not spot-check the node and assign users to download data from this node, so the edge node will not get the benefit.

# Resource spot check of Edge nodes

The resources of the titan network are composed of the resources of multiple edge nodes. The effectiveness of the resources of the edge nodes determines the effectiveness of the titan network. Therefore, the titan network periodically conducts random checks on the resources of edge nodes and links the results of the random checks to the revenue of the edge nodes. In this way, the edge nodes are encouraged to keep their resources stable and reliable, thus improving the reliability and availability of the titan network.

The random check is mainly to check the asset data storage of the edge nodes and to check the network bandwidth of the edge nodes. For example, when most of the titan network's bandwidth resources are used for user downloads, spot-checking can no longer check the bandwidth of the edge nodes, because it will take up the uplink bandwidth of the edge nodes and affect the user download experience.

The spot checks are planned and initiated by the scheduler and are performed several times a day on the edge nodes. First, scheduler will select some L1 nodes as verifiers, where L1 nodes are relative to edge nodes, which are called L2 nodes in the titan network. At the beginning of each round of spot-checking, the scheduler sends the information of the spot-checked edge nodes to some verifier node, and then the verifier node will spot-check the edge nodes.

The spot check process for edge nodes is as shown in the following figure:

At the beginning of each round of spot-checking, Titan scheduler obtains a distributed random number from the Filecoin main chain and combines it with the ID of each edge node to generate a random number of spot-checks for that node. this random number determines which asset of the edge node needs to be spot-checked and which block in the asset. Titan scheduler sends this Based on this information, the Edge node reads the block in the corresponding asset and sends it back to the verifier.

After receiving the block from the edge node, the verifier calculates the CID of the block, the network speed of the block sent by the edge node, and so on, and sends them to the titan scheduler as the result of the spot check.

After Titan scheduler receives the spot check results, it needs to compare the block CIDs with the block CIDs on the database, if not, it indicates that there is something wrong with the data of the edge node.

Titan scheduler packages the sampling results of all edge nodes in each round and generates a zero-knowledge proof, which is submitted to the FVM contract for backtracking.

# Edge Node Asset System

The core task of the edge node is to pull the asset data from the specified source to the storage system of the edge node, and then provide the download service to the user via HTTP server. Therefore, the management of the asset, including the data storage of the asset, the query of the asset object, and the consistency of the view of the asset data, is the core function of the edge node.

## Storage Management

Each asset requires corresponding storage for the data of the asset. edge nodes need storage for asset objects, but also for node management storage, data generated by edge nodes themselves, such as key/value database for recording CID of asset objects, node operation log, etc. Considering the failure rate of the storage devices, we suggest that these two types of storage belong to different physical storage devices. For example, the physical storage device that stores asset data, which is more frequently read and written, is therefore more prone to failures if the data used for node management is also located on top of it, resulting in the node not functioning properly.

The Edge node's Storage manager component is used to manage the node's storage. For example, the node manager can mount multiple storage paths, set the amount of storage available for each path, and set the purpose of the storage. The storage manager detects whether a storage path is online by regularly stating the storage path. When the storage path is offline, the asset data on it is not available, and the edge node will report the asset data failure to the titan scheduler, and the scheduler will avoid using the edge node with the failed asset data.

## Asset Object Management

The Assets manager component of the Edge node is responsible for managing all the asset objects on the node. For the asset objects that have completed data pull, Assets manager ensures that the asset data view of the edge node and the asset view of the scheduler are consistent by dividing these asset objects into different buckets and then periodically synchronizing them with the titan scheduler via bucket hash. The data view of the edge node and the asset view of the scheduler are consistent. In response to a request from the scheduler to create a new asset object, the Assets manager requests storage space for the new asset object through the Storage manager component, and is responsible for pulling the asset data from the source specified by the scheduler to the newly allocated storage, completing the asset request.

When the user downloads the asset data through the HTTP server of the edge node, the HTTP server needs to get the read interface of the asset data through the Assets manager. If the user only downloads a block of the asset data, the index data of the asset object is also needed, and the Assets manager is responsible for loading and managing the index, for example, to unload the index when it is no longer needed and to reduce memory usage.

## Pull asset from source

Titan scheduler calls this interface and requests the edge node to pull an asset to local storage. If there are no errors, the edge node will pull all the blocks of the asset (carfile) from the source specified by the titan scheduler and write them to a carfile file in car v2 format using the BFS (depth-first traversal) method. The writing is done using the "go-car" golang library. go-car also builds the index of the carfile and writes it to the carfile.

The index of the carfile is needed because the download service of the edge node supports users to request individual block content by the ID of the block, so we need the index to find the offset of the block located in the carfile in order to read out the content of the block quickly.

After the Edge node receives the pull asset request, it first checks whether the asset has been pulled to local storage, and if so, it returns an error to inform the scheduler that the asset has been pulled to local.

The Edge node checks if there is currently an asset pull task in progress, and if so, it needs to let new asset pull requests queue, because the edge node allows only one ongoing asset pull task.

The Edge node makes a persistent record of both ongoing and queued asset pull requests, so even if the edge node program restarts, it can resume these assets pull requests from the persistent record.

The Edge node pulls the asset data from the source via the HTTP protocol.

For carfile, the edge node is requesting source block by block, which is mainly because when network error occurs, the lost data is a number of blocks, and the already downloaded blocks do not need to be re-downloaded. Later we may also support downloading carfile by range, after each range is downloaded, the data will be checked by the hash of the range to see if it is correct, and when the network error occurs, what we lose is a number of ranges, and the downloaded ranges do not need to be re-downloaded. The range approach is more granular than the block approach, reducing the number of HTTP requests, and the range approach can download not only carfiles but also other formats, such as raw video files.

After an edge node asset pull request, the scheduler will periodically check the progress of the asset pull request. If the edge node crashes, or if there is a network problem, the scheduler will consider the asset pull request to have failed after several failed queries, and the scheduler will reselect another edge node for the asset pull.

## Edge node release asset

Titan scheduler requests the edge node to release the assets that are no longer needed, and the edge node deletes the local carfile file and index file.

The edge node also needs to delete the asset records from the carfile manager and update the hash of the bucket to which the carfile belongs in order to quickly complete the state synchronization with the scheduler afterwards.

## Query the status of asset

Titan scheduler calls this API to get information about the asset. The edge node returns basic information about the asset such as file size, file creation date, etc. If the asset does not exist, an error message is returned.

## carfile integrity checking

Titan scheduler will call this API and ask the edge node to rescan the blocks contained in the carfile to check if the CID and block contents are the same.

This process requires scanning the entire carfile and can be time consuming. The edge node will control the number of concurrent calls to this interface and will return a "device busy" error if the number of concurrent calls exceeds the number.

## Synchronization with Scheduler

The Edge node's storage holds the asset data, and it needs to periodically align with the titan scheduler to ensure that the asset data is consistent. If the asset data is inconsistent, the edge node needs to retrieve the asset data from other nodes in the titan network that have that data.

Titan scheduler's database records a list of all the carfile hash lists saved by each node. These hashes are divided into multiple buckets to improve the processing speed of state synchronization.

The synchronization process is initiated by the titan scheduler. titan scheduler and edge nodes divide all carfiles into several buckets, and the synchronization is mainly done by comparing the bucket hash:

bucket\_hash = HASH (all carfile hash)

top\_hash = HASH (all bucket\_hash)

Titan scheduler first sends top\_hash to edge node. edge node also calculates local top\_hash by the same method, if both top\_hash are the same, then the data is consistent and state synchronization is complete.

If the top\_hash is different, it means there is inconsistent data. First, we need to find out the inconsistent bucket: titan scheduler sends all the bucket\_hash to the edge node, which compares them bucket by bucket to find out the inconsistent bucket.

After identifying the inconsistent buckets, the edge node requests the titan scheduler to send back the hash list of carfiles in the discrepant buckets, find the inconsistent carfiles, and finally fix the discrepant carfiles.

# Download service

Edge nodes provide download services to the public via HTTP server.

Considering that edge nodes usually run in a home environment, NAT penetration is the norm; NAT penetration requires collaboration between the server and the client to complete, so we also provide an SDK that integrates NAT penetration services, so users can pull resources from edge nodes by using our SDK, and the NAT penetration process will be described in detail in subsequent sections.

### Choice of download protocol

The Edge node HTTP server supports both TCP-based HTTP, and QUIC (UDP)-based HTTP.

The advantage of TCP-based HTTP is that it has wide client support, but the disadvantage is that NAT penetration is difficult.

The advantage of QUIC-based HTTP is that QUIC is based on UDP, which makes NAT penetration easy, and QUIC also improves bandwidth utilization in TCP congestion situations.

Edge nodes usually run in a home network environment, which means they are usually located behind NAT devices and therefore require NAT penetration, so QUIC-based HTTP download protocols are important.

## HTTP Server

HTTP server implementation for Edge nodes, referencing the definition of IPFS HTTP Gateway: <https://github.com/ipfs/specs/blob/main/http-gateways/PATH_GATEWAY.md>

### Trusted mode

The Edge node extracts the original file contained in the carfile and sends it back to the user. edge node assigns the most recently accessed (LRU) carfile original file in order to speed up the response, so that it does not need to repeat the extraction process.

Edge nodes ensure the correctness of the sent content, e.g., the CID computed from the carfile content or the block content must be the same as the carfile or block CID, which is already done by the edge node when its cache the content, so there is no need to repeat the process when the user requests it.

The user trusts the data sent back by the edge node.

### Trustless mode

The Edge node does not extract the original file contained in the carfile, but sends the carfile or block back to the user directly. After receiving the carfile or block, the user needs to verify the correctness of the received content himself, for example, the CID computed from the content needs to be consistent. The edge node does not perform the calculation.

On the other hand, since these contents are assets managed by the edge node, when pulling these contents, the edge node will calculate and verify the contents to make sure that the CID calculated by it is consistent.

Therefore, from the edge node's point of view, the difference between the trust and trustless modes is only whether the edge node extracts the original data within the carfile (decodes it) and sends it to the user, or sends the data of the carfile directly to the user.

### IPNS

We use the CID to represent the unique identifier of the block/carfile, which is based on a hash of the block/carfile content. The client cannot get the updated content based on a definite identity.

Therefore, we need an IPNS (InterPlanetary Name System), where each IPNS corresponds to a CID record that can be updated.

When the content is updated, we update the IPNS corresponding to the CID record at the same time, then this IPNS corresponds to the new CID record. This is similar to our DNS domain name system, where we have a domain name such as example.com, which corresponds to an IP address record that can be updated.

Subsequent versions of Edge nodes will need to support IPNS in order to be able to accommodate users using the same identifier to get the updated content.

### DNSLink

IPNS is based on a hash of the resource owner's public key, e.g. /ipns/hash{resource-owner-pub-key}, so readability is not good. Besides IPNS, you can also use DNSLink to achieve the effect of IPNS by using the TXT record of DNS domain name system, and the name of DNSLink is more readable, e.g. \_dnslink.docs.ipfs.tech, which can be resolved by DNS TXT record as

/ipfs/bafybeieenxnjdjm7vbr5zdwemaun4sw4iy7h4imlvvl433q6gzjg6awdpq

For DNSLink support, we plan to implement it to the Titan DNS service in a subsequent release, where the Titan DNS service will maintain TXT records and respond to DNS queries from users.

### Caching Control

Edge node's asset management is controlled by the titan network's scheduler server, e.g., only the scheduler can add or remove the edge node's asset content.

Edge nodes do not provide a cache control mechanism for users, so edge nodes ignore the cache-related settings of users' HTTP Headers.

### Subdomain Gateway

Nowadays, browsers have a cross-domain security mode. The introduction of the mode, the meaning of the mode.

We need to satisfy this mode thus we need the subdomain approach.

### HTTP APIs

#### GET

GET /ipfs/{cid}[/{path}][?{params}]

Parameter Description:

* cid: the CID of the specified resource, such as the CID of carfile or block
* path: optional resource path, used to specify the path of the UnixFS directory with CID as root
* params: optional query parameters, used to specify options when the server returns results

#### HEAD

HEAD /ipfs/{cid}[/{path}][?{params}]

Similar to the GET API, but the edge node does not send the content of the carfile/block and is used to support the user in querying information about the content corresponding to the CID.

For example, query the existence of the content corresponding to the CID, query the length of the content corresponding to the CID, query the type of content corresponding to the CID (images, etc.).

#### Query parameters

GET/HEAD request can combine with following query parameters:

1. filename

When the user's request specifies filename, for example filename=abc.jpg, the edge node sets the Content-Disposition parameter in the header of the HTTP Response and sets the Content-Type parameter based on the suffix of the file name.

1. download

When the user's request specifies download, e.g. download=true, the edge node sets Content-Disposition: attachment[;filename=...] in the header of the HTTP Response that tells the user side that instead of rendering the response, the Save as dialog should pop up and let the user save the file.

1. format

The user specifies the format of the content he expects via the format parameter. The effect is equivalent to setting the Accept parameter in the header of an HTTP Request.

The corresponding relationship is as follows:

|  |  |
| --- | --- |
| format=raw | Accept:application/vnd.ipld.raw |
| format=car | Accept:application/vnd.ipld.car |
| format=tar | Accept:application/x-tar |
| format=dag-json | Accept:application/vnd.ipld.dag-json |
| format=dag-cbor | Accept:application/vnd.ipld.dag-cbor |
| format=json | Accept:application/json |
| format=cbor | Accept:application/cbor |

### HTTP Request Header

#### Accept

Used to specify the content format that the user can receive.

* application/vnd.ipld.raw: the original block content, the edge node does not need to parse the content of the block, it can be sent directly to the user
* application/vnd.ipld.car: the original carfile content, the edge node does not need to parse the content of the carfile, it can be sent directly to the user
* application/x-tar: If the CID refers to a UnixFS folder, the folder will be packed into a tar file and sent back to the user. If the CID refers to a folder other than UnixFS, it returns 400, indicating a Bad Request.
* application/vnd.ipld.dag-json: The Edge node converts the contents of the CID into DAG-JSON format by IPLD Data Model, or sends it to the user without conversion if the contents themselves are in DAG-JSON format. If there is an error in the conversion process, 500 is returned to the user, indicating an internal error.
* application/vnd.ipld.dag-cbor: The Edge node converts the content referred to by the CID to DAG-CBOR format via the IPLD Data Model, or sends it to the user without conversion if the content itself is in DAG-CBOR format. If there is an error in the conversion process, 500 is returned to the user, indicating an internal error.
* application/json: similar to application/vnd.ipld.dag-json, unless the CID's codec already is json (0x0200), the edge node sends the original content (already in json format) directly back to the user.
* application/cbor: similar to application/vnd.ipld.dag-cbor, unless the CID's codec already is cbor (0x51), the edge node sends the original content (already in cbor format) directly back to the user.

#### Range

The range is used to specify a section of the original carfile or block when and only when the Accept parameter is application/vnd.ipld.raw or application/vnd.ipld.car. When the Accept parameter is not application/vnd.ipld.raw or application/vnd.ipld.car, the edge node returns a 400 error, indicating that such a request is not supported.

### HTTP Response Header

#### Status

* 200：Indicates that the request is successful
* 206：When the user presses RANGE to request content, 206 replies to indicate a successful request
* 400：Indicates an invalid request
* 429: indicates that the edge node is currently busy, try again later
* 500: indicates an internal error in the edge node

#### Response Headers

* Etag: Used to identify the received content on the user side, the value of the Etag must be inside double quotes. edge nodes use the CID to fill the Etag, e.g. "bafy...foo". If the user provides the filename parameter, the value of the Etag consists of the CID plus the filename suffix; if the user provides the format parameter, the value of the Etag consists of the CID plus the format, e.g., "bafy...foo.raw ", "bafy...foo.car", etc.
* Cache-Control: The edge node is filled with "public, max-age=29030400, immutable".
* Content-Type: the edge node is based on the format provided by the user at the time of request, if it is a carfile type, it must include the version number of the carfile, e.g., Content-Type: application/vnd.ipld.car; version=1; when the user does not specify the format, the edge node will fill the field with information such as the codec of the CID, or the metadata of the content.
* Content-Disposition: the edge node fills this field when the user sets download=true parameter or specifies format as carfile/raw. There are two types: Content-Disposition: inline, which means that the user side displays the content directly; Content-Disposition: attachment, which means that the user side pops up the "Save As" dialog and lets the user save the file.
* Content-Length: The edge node will fill this field to indicate the length of the content.
* Content-Range: The edge node populates this field when and only when the user provides the range parameter when requesting.
* Accept-Ranges: The edge node replies to the user HEAD request by filling this field if the content is available by range, informing the user that the content is available by range.
* X-Content-Type-Options: when the content format is carfile/raw, the edge node will fill this field with X-Content-Type-Options: nosniff to inform the client that it does not need to detect the content type based on the content.

### HTTP Response Payload

In general, the format of the payload sent by the edge node to the user is determined by the format specified by the user when requesting it.

As explained in the section Accept, for example, when the format specified by the user at the time of request is

application/vnd.ipld.raw: sends the raw carfile/block to the user.

application/vnd.ipld.car: sends the carfile to the user. Note that the order of the blocks in the carfile is arbitrary, such as depth-first or breadth-first.

When tar: sends the UnixFS folder to the user as a tar file.

When the user does not specify, the edge node tries to determine the format of the payload based on the codec of the CID.

## NAT punch

### NAT Type

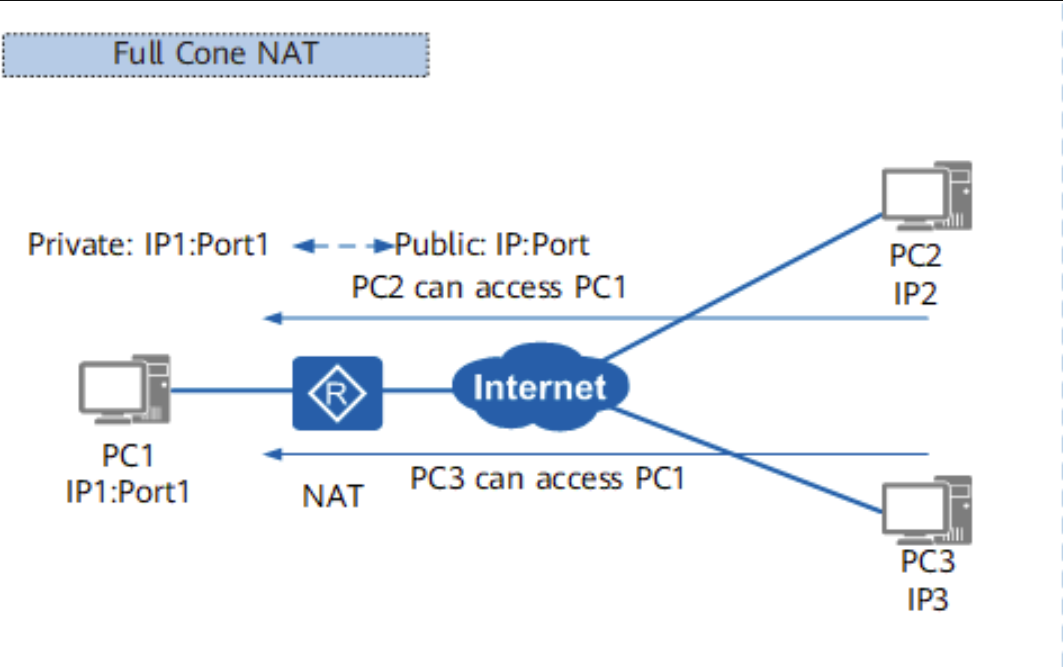
We identified the following four types of NAT devices based on the definition of STUN:

#### Full-cone NAT (full-cone NAT)

This type of NAT is the easiest to penetrate.

All packets sent outward from the same intranet IP and port will be mapped to the same outtranet IP and port, e.g., all intranet 192.168.1.100:3000 sent outward will be mapped to the public network 66.249.66.82:4000. and any outtranet IP and port that sends packets to the public network 66.249.66.82:4000 will be NAT device will be forwarded to the intranet 192.168.1.100:3000.

As shown in the figure below:

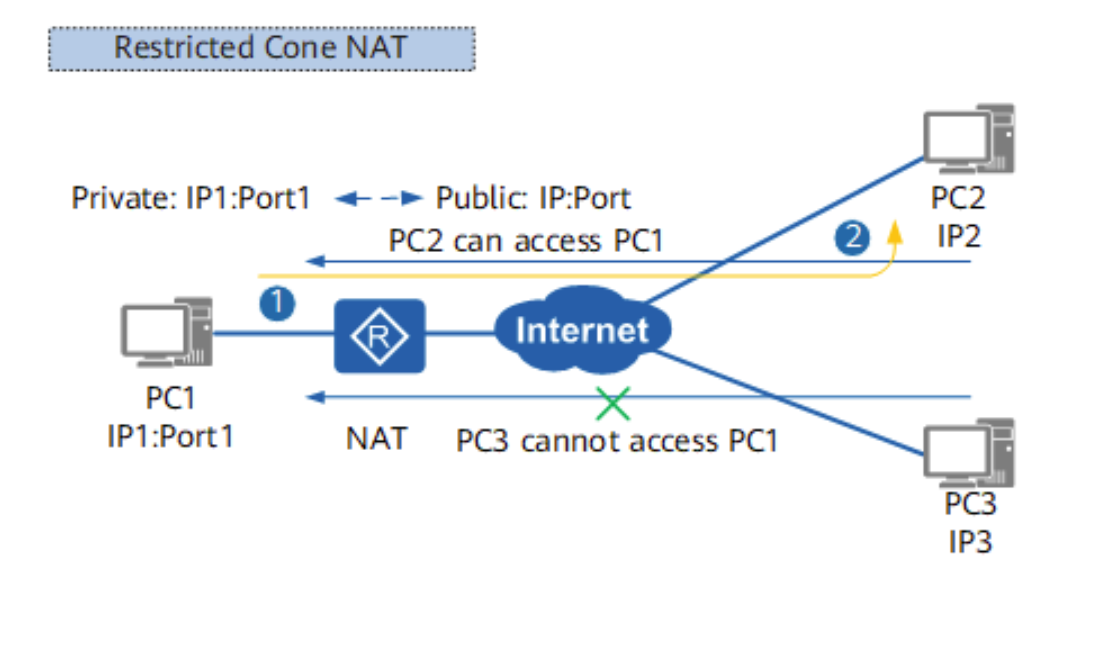


#### Restricted-cone NAT (IP-Restricted)

This type of NAT requires a punch action.

From the same intranet IP and port to send out will be mapped to the same outtranet IP and port, for example, all the intranet 192.168.1.100:3000 to send out will be mapped to the public network 66.249.66.82:4000, this is the same as Full-cone, but only when the recorded IP of the external network, to the public network 66.249. 66.82:4000 to send packets will only be NAT devices forwarded to the internal network 192.168.1.100:3000.

Briefly, this means that if and only if the intranet IP has sent a packet to an extranet IP, this extranet IP can send a packet to the intranet IP, as shown in the following figure, if IP3 wants to send a packet to IP1, then IP1 must first send a packet to IP3 in order for the NAT device to have a record of IP3, as shown in the figure below:

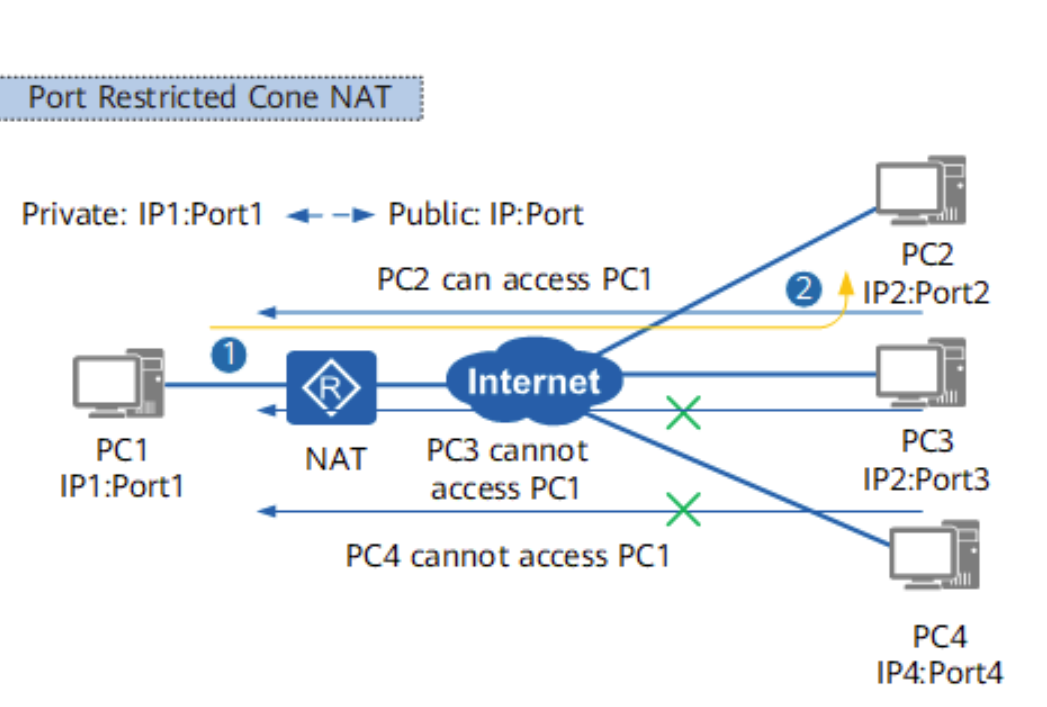


#### Port-restricted cone NAT

This type of NAT requires a punch action.

All packets sent outward from the same intranet IP and port will be mapped to the same outtranet IP and port, e.g., all packets sent outward from the intranet 192.168.1.100:3000 will be mapped to the public network 66.249.66.82:4000. but only if the logged outtranet IP and port, sending packets to the public network 66.249.66.82:4000 will be forwarded by the NAT device to the intranet 192.168.1.100:3000.

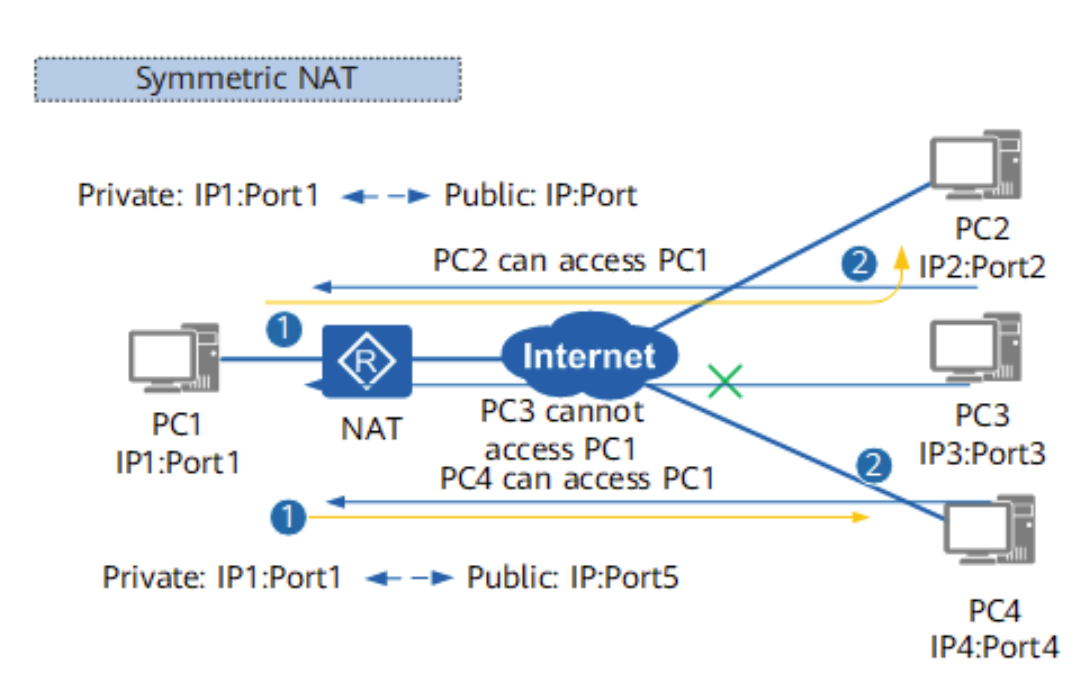
In other words: compared with Restricted-cone NAT, there is an additional port constraint, as shown in the figure below, packets sent from IP2:Port3 to IP:Port will be discarded by the NAT device because there is only IP2:Port2 record in the NAT device record, and there is no record of Port3, as shown in the figure below:



#### Symmetric NAT

This type of NAT is difficult to penetrate.

Because sending from the same intranet IP and port to the same outbound target IP and port is mapped to the same outbound IP and port, that is, this mapping is based on <intranet IP port; target outbound IP and port>'s and different target outbound IP and port, which belong to different mappings. As shown in the figure below, PC1's IP1:Port1 sends packets to IP2:Port2 and to IP4:Port4, and the NAT device establishes two different mappings to PortPort and PortPort5, which means that PC2 machine sees PC1's mapped IP:Port, while PC4 sees PC1's mapped IP:Port5. PC3 then cannot communicate with PC1 because PC1 does not actively send packets to PC3, as shown in the figure below:



### Our NAT Type Identification

Based on the UDP protocol, the edge nodes send UDP packets to our two servers separately, and the two servers get the public IP and port of the edge nodes and compare them. If they see the same public IP and port of the edge nodes, then the NAT device of the edge nodes is tapered; if they see different public IP and port of the edge nodes, then the NAT devices are symmetric.

If the edge node's NAT device is tapered, we continue with the following process:

We use the third group of servers to make a TCP connection to the TCP port of the edge node, and if the connection is successfully established, it indicates that the edge node belongs to the public IP.

We use the third group of servers to send UDP packets to the public IP and port of the edge node, and if the packet is successfully sent, the NAT device of the edge node is of Full-cone type.

We let the first group of servers send UDP packets to the public IP and port of the edge node on a different port, and if it succeeds, it indicates that the NAT device of the edge node is of Restricted-cone type.

When both tests fail as above, we assume that the NAT device of the edge node is of Port-restricted cone type.

As shown in the figure below:



### Our NAT punch implementation

Edge nodes and users generally run in a home network environment, and they generally need to go through two layers of NAT devices, as shown in the following:

(Edge node) ...... (edge node's home NAT device) ...... (edge node's carrier NAT device) ...... (public network) ...... (user's carrier NAT device) ...... (user's home NAT device) ...... (user)

To improve the NAT penetration success rate, we recommend fixed mapping in (edge node's home NAT device), including SNAT (source NAT) and DNAT (destination NAT) setting mapping to eliminate NAT in (edge node's home NAT device), so that the edge node only needs to traverse one layer of NAT devices:

(Edge node) ...... (edge node's carrier NAT device) ...... (public network) ...... (user's carrier NAT device) ...... (user's home NAT device) ...... (user)

The user side uses our SDK to confirm his NAT type. When the user requests the link information of the edge node where the resource is located, the scheduler will provide the NAT type of the edge node.

Then the user side uses our SDK to establish a link with the edge node using QUIC packets, with the following flow:

1. The NAT type of the edge node is public IP, and the user side connects directly to the edge node
2. the NAT type of the edge node is Full cone NAT, the user side connects to the edge node directly
3. the user is a public IP, the user requests the scheduler to inform the edge node, the edge node links the user
4. the user is Full cone NAT, the user requests the scheduler to inform the edge node, and the edge node links to the user
5. either party is Restricted cone NAT, the user requests the scheduler to inform the edge node to send packets to each other, successfully penetrate and establish the link
6. both parties are Port restricted cone NAT, the user requests the scheduler to inform the edge node to send packets to each other, and successfully penetrate and establish the link
7. one side is Symmetric NAT, the other side is Symmetric NAT or Port restricted cone NAT, the user requests the scheduler to tell the edge node to send packets to each other and guess the possible ports of the other side, try several times, guess correctly to establish the link; otherwise, give up establishing the link

## Download flow

Our SDK gets the carfile process in the browser environment as follows:

1. The user uses our SDK to pull the carfile
2. SDK initializes with scheduler to determine its NAT type
3. SDK requests available edge nodes for the carfile from the locator
4. locator selects several schedulers based on the user's IP and asks them for the edge node of the carfile
5. the scheduler with the carfile returns a list of available edge nodes (and credentials) to the client
6. the SDK decides whether NAT penetration is required to establish a connection based on its own NAT type and the NAT type of the edge node
7. SDK's connection builder uses the list of edge nodes to build connections
8. SDK selects the connections and pulls the carfile

To pull content from an edge node, the user needs to submit service credentials. Edge nodes verify the validity of the credentials before providing the download service.

The service credentials are generated by a scheduler, each of which has a separate key pair and therefore generates different credentials.

The user uses the service credentials to obtain authorization to download, and the edge node uses the credentials to obtain proof of workload.

## Download proof

### Asset access credentials

Asset access credentials have two roles: first, they are the authorization for downloading resources, which cannot be downloaded from the edge node without the credentials; second, the edge node needs to generate workload certificates based on the credentials, indicating that the edge node provides download services for the resources specified by the credentials.

Asset access credentials are data blocks encrypted by the titan scheduler server using the edge node's public key and signed by the scheduler's private key. Since only the corresponding edge node can decrypt it, this credential is only valid for a specific edge node. scheduler signatures ensure that the credential is issued by the scheduler.

The data block of the voucher includes the following:

* Unique ID of the credential
* Carfile of the credential
* Download speed limit of the credential
* Client ID, the unique ID of the user side
* Credential expiration date, etc.

### Asset access credentials generation

The user initiates a resource request and the locator forwards the request to a titan scheduler based on the proximity principle.

Titan scheduler checks whether the user meets the access limit of the resource, and sends an error message to the user if it is not met.

Titan scheduler looks for the edge node where the resource is located, which usually has multiple copies stored in multiple edge nodes.

Titan scheduler selects several edge nodes and generates asset access credentials for each edge node.

### Work proof

The proof of workload submitted on the user side, encrypted using the public key of the scheduler, contains the following:

* Unique ID of the credential
* Edge node ID
* Download speed
* Amount of data downloaded
* Download start time
* Download end time, etc.

The proof of workload submitted by the Edge node, encrypted with the public key of the scheduler and signed with its own private key, includes the following:

* Unique ID of the access credentials
* User ID
* Download speed
* Amount of data downloaded
* Download start time
* Download end time, etc.

A final valid proof of workload is recognized if and only if both the edge node and the user submit identical and valid proofs of workload, i.e., only the edge node, or the user, submits a valid proof of workload, which is not counted as the contribution of the edge node.

### Considerations on cheating

#### Storage cheating

The Titan network randomly checks the blocks of the edge nodes every day, and each random check selects a number of blocks. The scheduler confirms whether the data of these blocks are correct. This mechanism requires the edge node to store all the blocks assigned to it correctly. in other words, it is difficult for the edge node to store cheats.

#### Bandwidth cheating

As mentioned earlier, the titan network is used to calculate the bandwidth contribution of the edge node through bandwidth spot-checking, which is done on the basis of storage spot-checking, that is, after the edge node finishes storage spot-checking, the L1 node can calculate the bandwidth of the edge node to send blocks. Unless the edge node can control the L1 node that spot-checks him to make this L1 node give a false bandwidth, but the L1 node is multiple and the L1 node that spot-checks the edge node is randomly assigned, so it is difficult for the edge node to cheat the bandwidth.

But after the bandwidth of edge nodes is increasingly used to respond to users' resource download requests, we need to calculate the bandwidth contribution of edge nodes by users' download reports. If a user side, which is controlled by a malicious edge node, then this user side, can report a false bandwidth, which enables the malicious edge node to bandwidth cheat.

The approach we currently use is to suppress cheating by making it more difficult to cheat. Since the resource is located in multiple edge nodes, the user end controlled by the edge node, each time he requests the resource, the list of edge nodes assigned to him by the scheduler is random and does not necessarily contain the malicious edge node. Therefore the malicious edge node, needs multiple attempts to get a chance to cheat. Later we will combine more methods, such as using TEE (Trusted Execution Environment), to further increase the difficulty of bandwidth cheating.

# Edge node updater

Updater is a stand-alone daemon program that is used to update the edge node program.

Some of our edge nodes are box devices, so it is not convenient to update the program manually, so an updater is needed to complete the update of the node program.

The Edge node manager logs into the titan website and clicks the "Update" button on his node management page. The titan scheduler sends a request to the updater program on the node, which downloads the new program to the standby zone and then modifies the Linux soft-link to make the Then, by modifying the Linux soft connection, the node program symbol will point to the new node program in the standby area and restart the node program to complete the update.