

1. 节点通信和多节点通信的实现方法

1.1 两节点之间通信的实现

节点通信的基本实现步骤如下：

- i. 首先建立TCP连接；（在此之前要定义好数据帧格式，例如：| header | frame |，各种数据包的id.....）
- ii. 在发送正式数据之前先进行握手，握手阶段主要要认证对方节点是否合法，保证双方的处理协议对等。如需加密可进行协商密钥，如有协议或节点版本升级，可检查版本信息是否匹配；
- iii. 经过握手成功后，才可以发送正式的数据；

1.2 多节点通信的实现方法

方案一：源路由的方式

- 1. 通过gossip协议同步全网的连接拓扑，得到全网拓扑图
- 2. 根据拓扑图，利用Dijkstra最短路径算法计算最短路径
- 3. 沿最短路径发送信息

方案二：路由表的方式

当节点数量非常大的时候，可采用路由表的方式。路由表中存储了到目的节点的下一跳节点，只需查表转发到下一跳节点即可。

dst	next	distance
1	2	1
...	...	

实现逻辑如下：

- 0. 每个节点维护一个路由表，路由表中的每一项<dst, next, distance>表明了到目的节点dst的下一跳节点是next节点，距离为distance。
- 1. 每个节点都维护自己的连接节点列表，当节点列表变化时更新本地路由表（如本节点1，新连接了节点2,则更新本地路由表中dst项为<2, 2, 1>，如断开连接，则直接删除对应项）
- 2. 邻节点间定时同步路由表信息，采取push-pull方式，将对方节点的路由表与本节点路由表进行合并。
- 3. 路由表合并规则如下：
 - 遍历邻节点 r_id 路由表中的每一项 <r_dst, r_next, r_distance>，发现有新的 r_dst 则添加到本地路由表中，添加为 <dst=r_dst, next=r_id, distance=r_idstance+1>
 - 如果本节点的 dst 项存在，则比较距离，邻节点对应项 r_distance+1 的值小于本节点 <l_dst, l_next, l_distance> 中的 l_distance，则发现了更优的路径，更新本地路由表。该项更新为<l_dst, l_next=r_id, l_distance = r_distance+1>。

4. 如果收到发送到目的节点dst的数据包，而查找本地路由表后没有该目的节点项，则从已连接的节点中随机选择一个邻节点发送。（可根据需要设置最大跳数限制）

代码实现见附录。

2. 跨图消息传递如何保持一致性

解决办法：主要的想法是对每个消息分配一个唯一ID，利用重发机制保证发送到对方，接收方收到消息并处理后，返回Ack，由接收方保证消息幂等。

发送方：

0. 对每个消息生成一个消息ID，可采用snowflake算法生成一个64bit的
ID <0-时间戳(41bit)-主机id(10bit)-序列号(12bit)>
1. 收到上层的消息后，首先检查当前待确认消息数量是否超出了限制，如果超过，通知上层。
2. 发送该消息，同时针对该消息注册一个超时事件，将该消息存放在待确认消息列表中保存，如果超时时间内，收到对方回应的 Ack ，则注销该超时事件，从待确认消息列表中删除该消息。
3. 如果发生超时，则可按指数退避算法进行重发，超过重发限制后向上层报告错误，并删除该消息。

接收方：

0. 维护一个消息确认表 <msg-id, timestamp> ，定时清理。
1. 收到消息后，进入消息处理逻辑，先读取消息ID，查表，看有无该ID，如有，返回Ack，return；如果没有，处理该消息，处理完后，将该消息ID添加到消息确认表中。返回Ack。

3. 附录

多节点通信代码实现：

```

#[macro_use]
extern crate log;

mod host;
mod netproto;

use host::*;
use netproto::RouteMessage;
use std::collections::HashMap;
use std::env;
/*
1. 运行时输入 router nodeid, 例如节点id=1, 则输入router 1, 监听端口为30000+id值,
2. 忽略参数合法性检测, 相关错误处理被忽略.
3. 因没有连接失败错误处理, 需要延时进行连接任务
4. 简单测试, 输入route <id-value>, 目前id-value值为[0..5]
*/

fn main() {
    init_logger();
    let cmd: Vec<String> = env::args().collect();
    info!("cmd args: {:?}", cmd);
    let nodeid: u64 = cmd[1].parse().unwrap();
    info!("local nodeid: {}", nodeid);

    let mut runtime = tokio::runtime::Builder::new()
        .basic_scheduler()
        .enable_all()
        .build()
        .unwrap();
    runtime.block_on(run(nodeid));
}

async fn run(nodeid: u64) {
    let host = Host::new(nodeid);
    let connect_nodes = create_test_connect_table();
    let conn_host = host.clone();
    tokio::spawn(async move {
        tokio::time::delay_for(tokio::time::Duration::from_millis(80 * 1000)).await;
        conn_host.connect_nodes(connect_nodes).await;
    });

    let timer_host = host.clone();
    tokio::spawn(async move {
        tokio::time::delay_for(tokio::time::Duration::from_millis(100 * 1000)).await;
        timer_host.timer_task().await;
    });

    // 测试从节点4发送到节点5
    if nodeid == 4 {
        let route_host = host.clone();
        tokio::spawn(async move {
            tokio::time::delay_for(tokio::time::Duration::from_millis(150 * 1000)).await;
            let message = RouteMessage {
                dst: 5,
            }
        })
    }
}

```

```
        src: 4,
        payload: vec![1, 2, 3],
    };
    route_host.send_route_message(message).await;
});
}

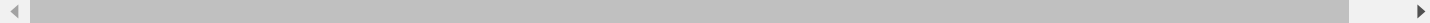
host.start().await;
}

fn init_logger() {
    simple_logger::init_with_level(log::Level::Info).unwrap();
}

// 测试用 图连接, 邻接列表
fn create_test_connect_table() -> HashMap<NodeId, Vec<NodeId>> {
    let mut table = HashMap::new();
    let n0 = vec![4, 1, 2];
    let n1 = vec![0, 3, 4];
    let n2 = vec![0, 3, 5];
    let n3 = vec![1, 2, 5];
    let n4 = vec![1, 0];
    let n5 = vec![2, 3];

    table.insert(0, n0);
    table.insert(1, n1);
    table.insert(2, n2);
    table.insert(3, n3);
    table.insert(4, n4);
    table.insert(5, n5);

    table
}
```



```

use super::netproto::network_client::NetworkClient;
use super::netproto::network_server::{Network, NetworkServer};
use super::netproto::{Hello, RouteAck, RouteItem, RouteMessage, RouteTable};
use async_trait::async_trait;
use parking_lot::RwLock;
use rand;
use std::collections::HashMap;
use std::net::{IpAddr, Ipv4Addr, SocketAddr};
use std::sync::Arc;
use tokio::time::Duration;
use tonic::transport::Channel;

type Session = NetworkClient<Channel>;
pub type NodeId = u64;

#[derive(Clone)]
pub struct Host {
    clients: Arc<RwLock<HashMap<NodeId, Session>>>,
    route_table: Arc<RwLock<HashMap<NodeId, RouteValue>>>,
    info: HostInfo,
}

impl Host {
    pub fn new(nodeid: u64) -> Self {
        Host {
            clients: Arc::new(RwLock::new(HashMap::new())),
            route_table: Arc::new(RwLock::new(HashMap::new())),
            info: HostInfo { nodeid },
        }
    }

    pub async fn start(&self) {
        let port = 30000 + self.info.nodeid();
        let addr = SocketAddr::new(IpAddr::V4(Ipv4Addr::new(0, 0, 0, 0)), port as u16);
        let network = Server::new(self.info.nodeid(), self.route_table.clone(), self.cl
        let networkservice = NetworkServer::new(network);
        tonic::transport::Server::builder()
            .add_service(networkservice)
            .serve(addr)
            .await
            .unwrap();
    }

    pub async fn timer_task(&self) {
        let interval = Duration::new(5, 0); // 1s timer
        loop {
            tokio::time::delay_for(interval).await;
            self.random_pull_route_table().await;
        }
    }
}

// 输入连接列表测试用
pub async fn connect_nodes(&self, table: HashMap<NodeId, Vec<NodeId>>) {
    if let Some(nodes) = table.get(self.info.nodeid_ref()) {

```

```

        for n in nodes {
            let rpc = self.create_new_client(n.clone()).await;
            let mut lock = self.clients.write();
            lock.insert(n.clone(), rpc);
        }
    }

pub async fn send_route_message(&self, message: RouteMessage) {
    let dst = message.dst.clone();
    // 如果dst是本节点, 返回
    if self.info.nodeid == dst {
        warn!("dst is local, return.");
        return;
    }

    let req = tonic::Request::new(message.clone());
    // 查找路由表, 如果dst在路由表中, 则发送给路由表中的next节点;
    let mut random_send_flag = false;
    let remote;
    {
        let lock_table = self.route_table.read();
        if let Some(r) = lock_table.get(&dst) {
            remote = r.clone();
        } else {
            // 如果dst不在路由表中, 则随机选择一个已连接的节点, 发送
            // self.random_send_route_message(message).await;
            random_send_flag = true; // fixme: 这块还没有写好, 先暂时忽略, 优先写正常情况
            return;
        }
    }

    if random_send_flag {
        self.random_send_route_message(message).await;
        return;
    }

    if let Some(c) = self.try_get_client(remote.next()) {
        trace!("remote {} is connected.", remote.next());
        let mut rpc = c;
        match rpc.route_message(req).await {
            Ok(_) => {
                info!("|----->route message to {}", remote.next());
            }
            Err(e) => {
                error!("route message to {} failure: {}", remote.next(), e);
                // todo: 从clients中删除对应项, 更新路由表
                return;
            }
        }
    }
}

// 定时执行拉取邻节点路由表
async fn random_pull_route_table(&self) {

```

```

// fixme:先根据自身的连接列表更新路由表，正常情况还应该时有新连接或者断开一个节点时更新路由表
let connect_list: Vec<NodeId> = self.clients.read().iter().map(|(k, _v)| k.clone())
let list_len = connect_list.len() as u64;
if list_len == 0 {
    return;
}

{
    for ref i in connect_list.clone() {
        let mut lock = self.route_table.write();
        lock.insert(i.clone(), RouteValue::new(i.clone(), 1));
    }
}

let mut items = Vec::new();
for (k, v) in self.route_table.read().iter() {
    let item = RouteItem {
        dst: k.clone(),
        next: v.next.clone(),
        distance: v.distance,
    };
    items.push(item);
}

let route = RouteTable {
    nodeid: self.info.nodeid(),
    item: items,
};

// 为便于测试，随机选取一个节点，实际可随机选取k个节点
let k = rand::random::<u64>() % list_len;
let random_node = connect_list[k as usize];

let mut rpc = self.clients.read().get(&random_node).unwrap().clone();
debug!("random pull node {} route table", random_node);
let r = rpc.pull_route_table(route).await;
//fixme: 这里返回的对方节点的路由表，应该进行一次合并，这里先不合并了
}

async fn random_send_route_message(&self, message: RouteMessage) {
    let nodes: Vec<NodeId> = self.clients.read().iter().map(|(k, _v)| k.clone()).collect();
    if nodes.len() > 0 {
        let k = rand::random::<u64>() % nodes.len() as u64;
        let random_next = nodes[k as usize];

        if let Some(c) = self.try_get_client(&random_next) {
            let mut rpc = c;
            let req = tonic::Request::new(message.clone());
            match rpc.route_message(req).await {
                Ok(_) => {
                    info!("|----->random route message to {}", random_next);
                    return;
                }
                Err(e) => {
                    error!("route message to {} failure: {}", random_next, e);
                }
            }
        }
    }
}

```

// todo: 从clients中删除对应项, 更新路由表

```

        return;
    }
}

error!("random select node failure.");
}

fn try_get_client(&self, remote: &NodeId) -> Option<Session> {
    if let Some(c) = self.clients.read().get(remote) {
        return Some(c.clone());
    }

    None
}

fn remove_peer(&self, id: &NodeId) {
    unimplemented!()
}

// todo: 忽略参数检查及错误处理
// 为了简化程序设计, 运行时输入 router nodeid, 例如节点id=1, 则输入router 1, 监听端口为30000
async fn create_new_client(&self, nodeid: u64) -> NetworkClient<Channel> {
    let port = 30000 + nodeid;
    let remote = SocketAddr::new(IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1)), port as u16);

    let mut addr = String::from("http://");
    addr.push_str(remote.to_string().as_str());
    trace!("prepare connecting to {}", addr);

    let endpoint = tonic::transport::Endpoint::new(addr).unwrap();
    let endpoint = endpoint.timeout(std::time::Duration::new(3, 0));

    let conn = endpoint.connect().await.unwrap();
    debug!("connected to {} success.", remote);
    NetworkClient::new(conn)
}

#[async_trait]
impl Handler for Host {
    async fn handle(&self, message: RouteMessage) {
        self.send_route_message(message).await;
    }
}

#[derive(Clone)]
struct HostInfo {
    pub nodeid: u64,
}

impl HostInfo {

```



```

    pub fn nodeid(&self) -> u64 {
        self.nodeid.clone()
    }

    pub fn nodeid_ref(&self) -> &u64 {
        &self.nodeid
    }
}

#[derive(Clone, Debug)]
struct RouteValue {
    pub next: u64,
    pub distance: u64,
}

impl RouteValue {
    pub fn new(next: u64, distance: u64) -> Self {
        RouteValue { next, distance }
    }

    pub fn next(&self) -> &u64 {
        &self.next
    }

    pub fn distance(&self) -> &u64 {
        &self.distance
    }
}

#[async_trait]
pub trait Handler: Send + Sync + Clone {
    async fn handle(&self, message: RouteMessage);
}

struct Server<H> {
    nodeid: NodeId,
    route_table: Arc<RwLock<HashMap<NodeId, RouteValue>>>,
    handler: H,
}

impl<H: Handler> Server<H> {
    pub fn new(
        nodeid: NodeId,
        route_table: Arc<RwLock<HashMap<NodeId, RouteValue>>>,
        handler: H,
    ) -> Self {
        Server {
            nodeid,
            route_table,
            handler,
        }
    }

    //合并路由表, 发现有新的id则添加, 如果本节点的某dst的距离比(邻节点的距离+1)大, 则更新本节点的该dst
    fn merge(&self, remote: &RouteTable) -> RouteTable {

```

```

let mut lock = self.route_table.write();
let remote_nodeid = remote.nodeid.clone();
let mut rev = Vec::new();

for r_item in remote.item.iter() {
    // 如果r_item的dst是本节点，则忽略合并
    if r_item.dst == self.nodeid {
        continue;
    }

    // 如果本地有对应dst项
    if let Some(l_item) = lock.get_mut(&r_item.dst) {
        let new_dis = r_item.distance + 1;
        // 如果比本地距离小，则更新
        if l_item.distance > new_dis {
            l_item.next = remote_nodeid;
            l_item.distance = new_dis;
        }
        let route_item = RouteItem {
            dst: r_item.dst.clone(),
            next: l_item.next.clone(),
            distance: l_item.distance.clone(),
        };
        rev.push(route_item);
    } else {
        // 如果本地没有对应dst项，在本地路由表中插入该dst项，next=remote_nodeid, distance
        let route_item = RouteItem {
            dst: r_item.dst.clone(),
            next: remote_nodeid.clone(),
            distance: r_item.distance.clone() + 1,
        };
        rev.push(route_item);
        lock.insert(
            r_item.dst.clone(),
            RouteValue::new(remote_nodeid, r_item.distance.clone() + 1),
        );
    }
}

RouteTable {
    nodeid: self.nodeid.clone(),
    item: rev,
}
}

#[async_trait]
impl<H: Handler + Send + 'static> Network for Server<H> {
    async fn hello(
        &self,
        request: tonic::Request<Hello>,
    ) -> Result<tonic::Response<Hello>, tonic::Status> {
        let nodeid = self.nodeid.clone();
        let hello = Hello { nodeid };
        Ok(tonic::Response::new(hello))
    }
}

```

```
}
```

```
/// 收到对方pull的请求，先与本节点路由表合并，再返回合并后的路由表
```

```
async fn pull_route_table(
    &self,
    request: tonic::Request<RouteTable>,
) -> Result<tonic::Response<RouteTable>, tonic::Status> {
    let remote_table = request.into_inner();
    let local_table = self.merge(&remote_table);
    debug!(
        "print route table after merge: {:?}",
        self.route_table.read()
    );
    Ok(tonic::Response::new(local_table))
}
```

```
/// 收到需要路由的消息， 如果dst是本节点，则路由终止，否则查找路由表，有对应dst项则转发到next节点
```

```
async fn route_message(
    &self,
    request: tonic::Request<RouteMessage>,
) -> Result<tonic::Response<RouteAck>, tonic::Status> {
    let message = request.get_ref();
    let mut rev = Ok(tonic::Response::new(RouteAck {
        src: message.src.clone(),
        dst: message.dst.clone(),
        result: 0,
    }));

    if message.dst == self.nodeid.clone() {
        info!(
            "|-----> received route message success from {}, payload: {:?}",
            message.src, message.payload
        );
        return rev;
    }

    self.handler.handle(message.clone()).await;

    // fixme: 临时先这么返回
    rev
}
```

