Nanomsg 代码分析 V0.1

MeteorKL 2016 年 06 月

https://github.com/MeteorKL/nanomsg-analyse

目录

nanomsg 是一个套接字库，提供了多种常见的通信协议，其目标是使网络层更快、更具扩展性、更容易使用。它是 zeromq 作者 Martin Sustrik 用 C 重写的一套具有可扩展协议的一套通信框架，nanomsg 的初衷是因为虽然 zeromq 已经足够快，但当面对一些变态的需求时（如同时处理>1 亿的请求时），由于对 zeromq 的内存存储优化不够，对系统负载会变的很大，这样对于强调性能和速度的 zeromq 来说，就显得捉襟见肘了。具体 nanomsg 与 zeromq 的不同与改进之处及为什么要用 C 重写在 nanomsg 官网和 Martin Sustrik 的博客有详细的描述。

1.用于新传输协议的 API——对于 ZeroMQ，人们经常抱怨的问题是它没有提供用于新传输协议的 API，这从根本上把用户限制在 TCP、PGM、IPC 和 ITC 上。而 nanomsg 提供了一个可插拔的接口，用于新的传输（如 WebSockets）和消息协议。

2.POSIX 兼容性——nanomsg 完全兼容 POSIX，而且 API 更简洁，兼容性更好。在 ZeroMQ 中，套接字用空指针表示，然后绑定到上下文；而在 nanomsg 中，只需要初始化一个新的套接字并使用它，一步即可完成。

3.线程安全——ZeroMQ 在架构上有一个根本性缺陷：其套接字不是线程安全的。在 ZeroMQ 中，每个对象都被隔离在自己的线程中，因此不需要信号量和互斥锁。并发是通过消息传递实现的。nanomsg 消除了对象与线程间的一对一关系，它不再依赖于消息传递，而是将交互建模为一组状态机。因此，nanomsg 套接字是线程安全的。

4.内存和 CPU 使用效率——ZeroMQ 使用一种很简单的 Trie 结构存储和匹配发布/订阅服务。当订阅数超过 10000 时，该结构很快就显现出不合理之处了。nanomsg 则使用一种称为“基数树（radix tree）”的结构来存储订阅，并提供了真正的零复制 API，允许内存从一台机器复制到另一台机器，而且完全不需要 CPU 的参与，这极大地提高了性能。

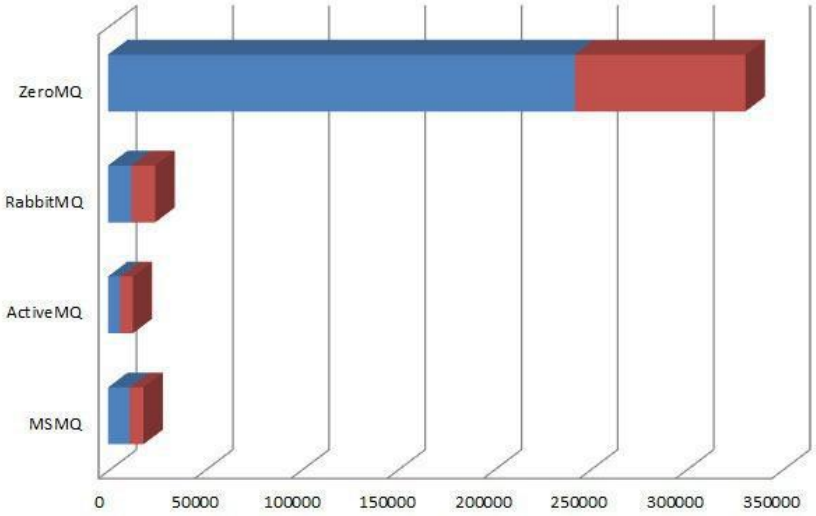
5.负载均衡算法——ZeroMQ 采用了轮转调度算法。虽然该算法可以平均分配工作，但也有其局限性。比如，有两个数据中心，一个在伦敦，一个在纽约。在理想情况下，一个位于伦敦数据中心的网站，其请求不应该路由到纽约。但在 ZeroMQ 的负载均衡算法里，这完全有可能。而 nanomsg 避免了这种情况的出现。

下面说一些我自己的体会。

nanomsg 用 c 实现，不依赖系统特性。需要先安装 cmake 编译工具，并源码下载下来，然后利用 cmake 根据 cmakelist 自动编译为当前操作系统可以加载的动态链接库，从而实现了支持多个操作系统的功能。

Nanomsg 支持创建新的通讯协议（例如 WebSockets, DCCP, SCTP）和新的信息发送模式。完全符合 POSIX 标准，提供的 API 十分简单，并且兼容各种操作系统。

关于性能比较，我并没有找到 nanomsg 的性能测评，但是找到了 zeromq 与其他软件的比较，下面是 mikehadlow 的博客中得到 MSMQ，ActiveMQ，RabbitMQ，ZeroMQ 这四款消息队列软件的性能比较，显然 ZeroMQ 和其它的软件不是一个级别。（性能测试代码在 GitHub 上）它的性能惊人的高，那么我也有理由相信 nanomsg 会是一款非常优秀的作品。（或许目前还在发展阶段，但是他未来一定会变得优秀）



	MSMQ	ActiveMQ	RabbitMQ	ZeroMQ
Send	11406	6452	12278	242659
Receive	7021	6453	12281	8[illegible]

1.2、信息发送模式

nanomsg 是一个实现了几种“可扩展协议”的高性能通信库。可扩展协议的任务是定义多个应用系统如何通信，从而组成一个大的分布式系统。

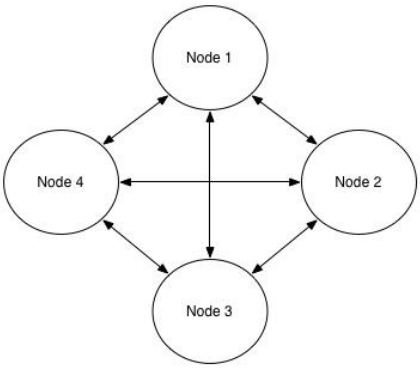
当前版本 nanomsg 支持以下信息发送模式：

- PAIR - simple one-to-one communication

  配对模式：简单的一对一的通信，支持两个节点的双向通讯。两个节点都可以发送和接收消息

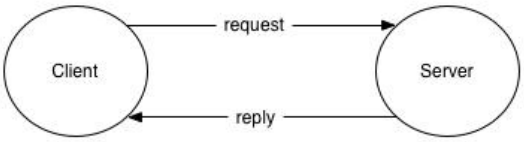
- BUS - simple many-to-many communication

  总线模式：简单的多对多的通信



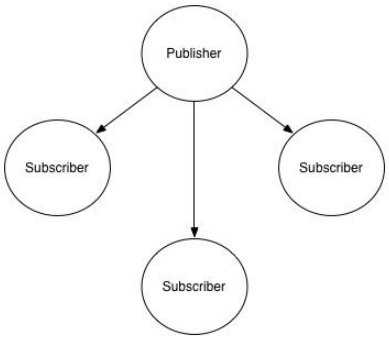
- Request/Reply - allows to build clusters of stateless services to process user requests

  请求/回复模式（应答模式）：为用户的请求创建无状态服务。客户端发送请求，服务器接收请求，做一些处理，并返回响应。支持组建大规模的集群服务来处理用户请求



- PUB/SUB - distributes messages to large sets of interested subscribers

发布/订阅模式：允许发布者发布多次信息到任意个订阅者，订阅者不可以反悔消息给发布者，但是可以选择他们想要接收消息



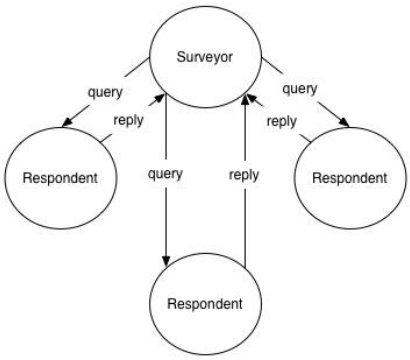
- PIPELINE - aggregates messages from multiple sources and load balances them among many destinations

  扇出模式：单向数据流实现负载均衡，producer 提交分发给多个 consumer 节点的任务

- SURVEY - allows to query state of multiple applications in a single go

  调查模式：允许在一个单一的请求里检查多个应用的状态。相当于是 REQREP 模式和 PUBSUB 模式的合并，从一个节点发布信息到多个节点，然后每个节点再返回信息。这个模式使你在一次查询下可以快速简单的得到多个系统的状态，

并且调查对象必须在指定的时间内做出反应。



1.3、网络通信协议

可扩展协议是在网络通信协议之上实现的，当前版本 nanomsg 支持一下网络协议：

- INPROC - transport within a process (between threads, modules etc.)
  进程间通信(Inter-Process Communication)
- IPC - transport between processes on a single machine
  进程间通信(Inter-Process Communication)
- TCP - network transport via TCP
  tcp 通信，传输控制协议(Transmission Control Protocol)
- WS - WebSocket protocol
  WebSocket protocol 是 HTML5 一种新的协议，它实现了浏览器与服务器全双工通信(full-duplex)，一开始的握手借助 HTTP 请求完成，具体可以看看知乎的解释

1.4、对外暴露的接口api

nanomsg 对外暴露的接口 api 定义在 nn.h 中：

```
NN_EXPORT int nn_socket (int domain, int protocol);
NN_EXPORT int nn_close (int s);
NN_EXPORT int nn_setsockopt (int s, int level, int option,
                             const void *optval, size_t optvallen);
NN_EXPORT int nn_getsockopt (int s, int level, int option,
                             void *optval, size_t *optvallen);
NN_EXPORT int nn_bind (int s, const char *addr);
NN_EXPORT int nn_connect (int s, const char *addr);
NN_EXPORT int nn_shutdown (int s, int how);
NN_EXPORT int nn_send (int s, const void *buf, size_t len, int flags);
NN_EXPORT int nn_recv (int s, void *buf, size_t len, int flags);
NN_EXPORT int nn_sendmsg (int s, const struct nn_msghdr *msghdr, int flags);
```

```
NN_EXPORT int nn_recvmsg (int s, struct nn_msghdr *msghdr, int flags);
NN_EXPORT int nn_device (int s1, int s2);
```

熟悉 socket 接口 api 的人应该对这些接口不陌生，一个简单的服务端应答程序大致是这样的：

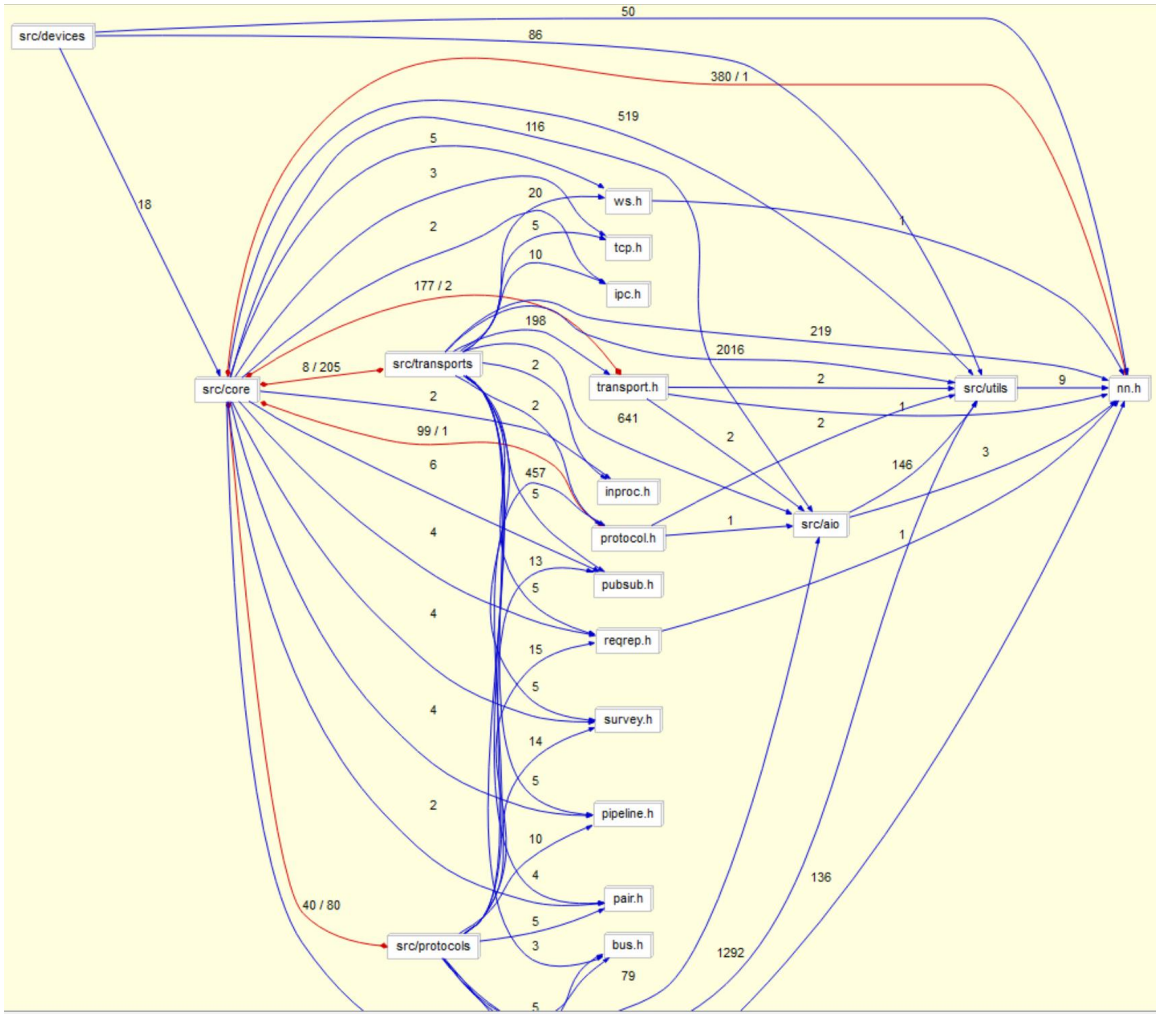
```
char buf[10];
int s = nn_socket(AF_SP, NN_REP);
nn_bind(s, "tcp://*:5555");
nn_recv(s, buf, 10, 0);
nn_send(s, "World", 5, 0);
nn_close(s);
```

对应的客户端请求程序大致为：

```
char buf[10];
int s = nn_socket(AF_SP, NN_REQ);
nn_connect(s, "tcp://localhost:5555");
nn_send(s, "Hello", 5, 0);
nn_recv(s, buf, 10, 0);
printf("Hello %sn", buf);
nn_close(s);
```

二、nanomsg 模块分析

用 understand 软件分析文件夹引用情况，可以认为该系统主要由 transport，protocol，aio，core 这四个模块。



令人惊喜的是，我在官网上还发现了一个 [nanomsg 的 gitter chatroom](), 去上面问了几个问题，帮助挺大的。Franklin Mathieu 建议我从 core/global.c 开始阅读

MeteorKL @MeteorKL 22:14

Oh, that's just like what I did when I debugged my battle city written by turob-c last year, but the project nanomsg seems to have a pretty complex reference and I don't know where to start.

Franklin Mathieu @Snaipe 22:24

@MeteorKL Start from `core/global.c`, since most of the API functions are implemented inside here. Most of the specialized code is then either in `aio` (platform specific sockets & all), `protocol` (for reqrep, pushpull, ...) or `transports` (for the handling of tcp, websockets, named pipes/domain sockets, ...)

I'd say that learning the structures comes along when you have something specific to change in mind

MeteorKL @MeteorKL 22:36

Thank you very much, I got it. But I think I'm not able to change it at present for my basic knowledge is not good enough. By the way, could you please tell me what's oneway, twoway, loopback means in device/device.h

Franklin Mathieu @Snaipe 22:38

Garett might be able to provide a more extensive answer, but I think those are the handlers for custom unidirectional, bidirectional, and loopback devices

basically the user providing a custom device implement those functions

MeteorKL @MeteorKL Jun 14 22:58

maybe I got it, different device is used for deliver message in different conditions. for example, unidirectional device is used to send message from 1 to 2, and bidirectional device is used to both send and receive message for 1 and 2. Am I right?

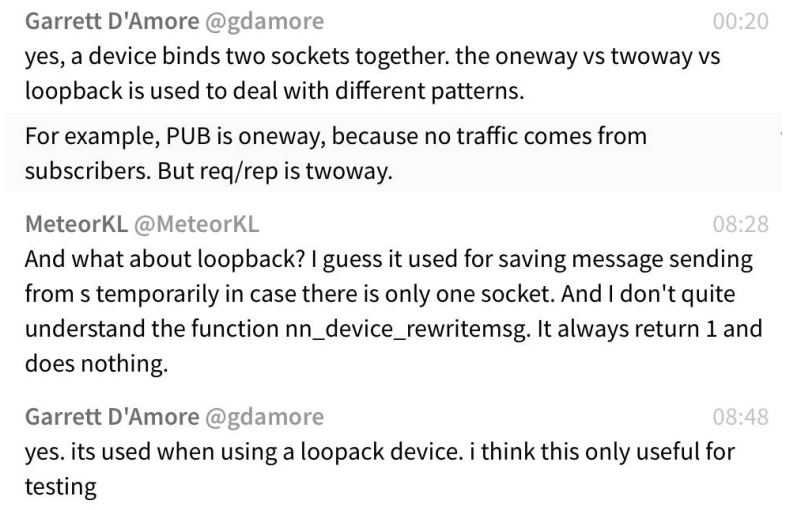
Franklin Mathieu @Snaipe Jun 14 22:59

Probably. Never worked with them directly, so others might have a better answer

Ioannis Charalampidis @wavesoft Jun 14 23:00

I am not sure but I think you can bind two sockets with a device

Ex a ta



2.1、接口

这个果真是用 c“重写”啊，到处都是面向对象的概念，我刚开始看代码的思路完全错了...很是尴尬。

总结了下该库有以下几个基类：

- nn_fsm 状态机基类
- nn_device 设备基类
- nn_sockbase 协议层基类
- nn_epbase 每个端点的基类
- nn_pipebase 管道基类
- nn_xreq

普通的类：

- nn_timerset 任务计时器的类
- nn_pipe 管道的类
- nn_socktype 协议类型的类
- nn_trie 高效单词查找树的类
- nn_transport 通信层的类
- nn_msgqueue 单向消息队列
- nn_chunkref 数据块引用的类

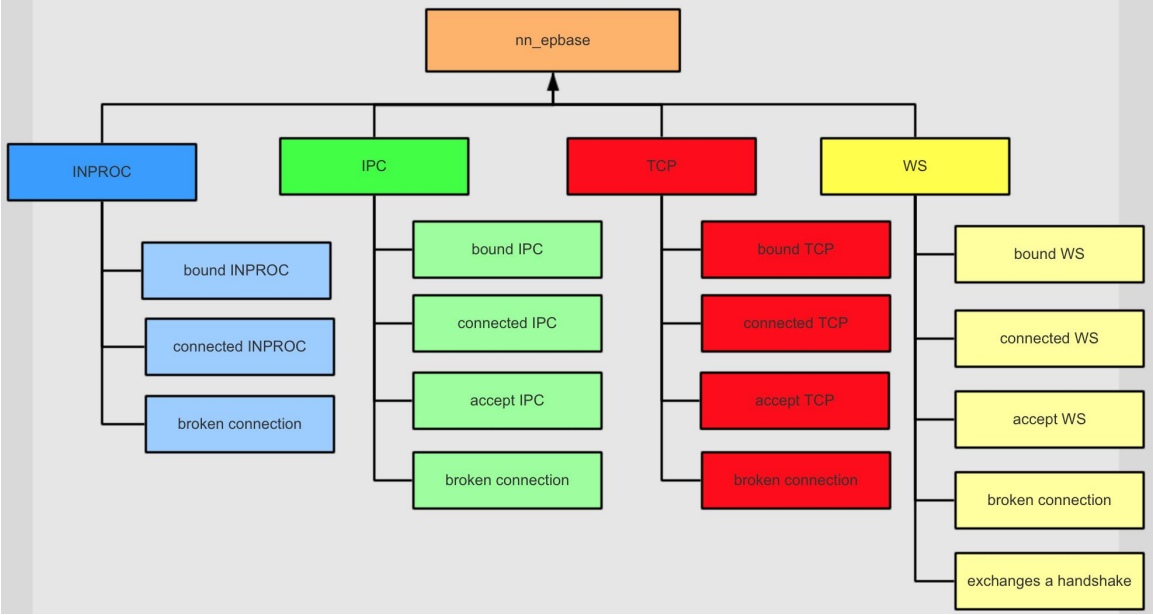
2.2、各大模块分析

2.2.1、utils

实用工具包，包含基本数据结构（list，queue，hash）互斥及原子操作（mutex，atomic）等

2.2.1、transports

通信层实现（包括 INPROC IPC TCP WS），完成 nn_epbase 接口中虚函数的实现



INPROC（进程内，线程间通信）传输方式的实现，由命名系统，队列类，随机生成 id，绑定，连接的支持

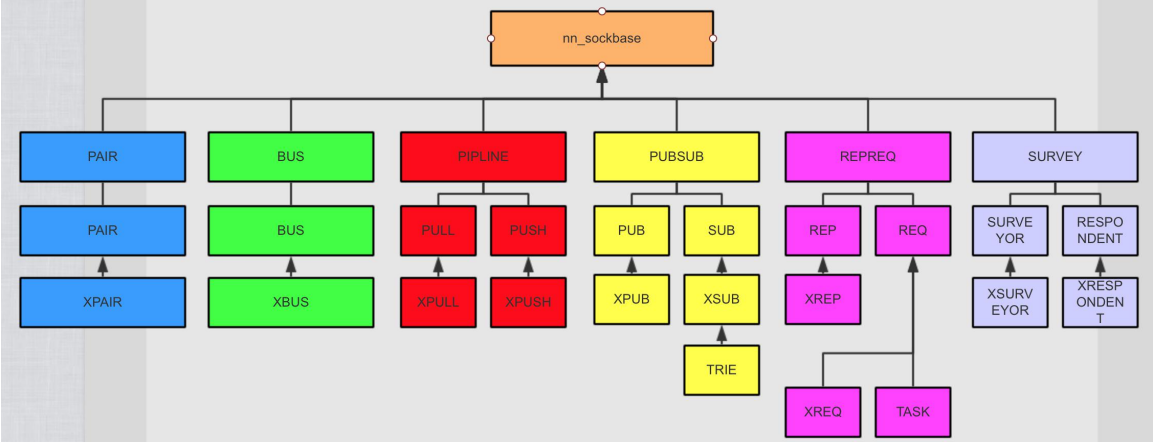
IPC（进程间通信）传输方式的实现，由 icp 的连接，绑定，接受，对方 icp 连接断开这四种情况的状态机支持

TCP 传输方式的实现，由 tcp 的连接，绑定，接受，对方 tcp 连接断开这四种情况的状态机支持

WS（web socket）传输方式的实现，由 ws 的连接，绑定，接受，对方 ws 连接断开，和客户端交换握手这五种情况的状态机支持

2.2.1、protocols

协议层实现（包括 PAIR BUS REQ/REP PUB/SUB PIPELINE SURVEY），完成 nn_sockbase 接口中虚函数的实现



当然这也是分层实现的，比如 bus 协议，先通过 busx 实现初始化，发送信息等函数功能，再右 bus 整合，添加检查函数，完成 nn_sockbase 接口中虚函数的实现

2.2.1、core

transport 和 protocol 等代码之间的“胶水”代码，负责各个“层” 之间的交流。

功能列举：

定义并实现了 nn_ep 结构函数以及系列函数(endpoint 指代一个连接的终端)，保存了一个终端的所有信息，实现了一个对终端初始化，析构，加入，删除等操作

通信层 nn_epbase 接口定义：定义了 nn_epbase 类，通过调用 nn_ep 系列函数实现，为其它函数调用提供接口

定义并实现了 nn_sock 结构函数以及系列函数，打包了协议层的相关代码

协议层 nn_sockbase 接口定义：定义了 nn_sockbase 系列函数，通过调用 nn_sock 系列函数实现，相当于是将 nn_sock 定义为了一个类给其他类调用

nn_pipebase 系列函数的实现，在对方突然断开连接的时候用到

实现异步 io，多路传输支持 poll

I/O 复用典型使用在下列网络应用场合：

当客户处理多个描述符（通常是交互式输入和网络套接字）时，必须使用 I/O 复用

一个客户同时处理多个套接字是可能的，不过比较少见。在 16.5 节结合一个 web 客户的上下文给出这种场合使用 select 的例子

如果一个 TCP 服务器既要处理监听套接字，又要处理已连接套接字，一般就要使用 I/O 复用

如果一个服务器既要处理 TCP，又要处理 UDP，一般就要使用 I/O 复用。8.15 节有这么一个例子

如果一个服务器要处理多个服务或者镀铬协议（在 13.5 节讲述的 inetd 守护进程），就要用 I/O 复用

全局符号表的定义，可以通过下标返回符号属性的值和名称
系统初始化和系统终止
开始运行：
全局环境创建的私有函数
（如果是 windows 系统）初始化 socket 库
初始化内存管理子系统
为假随机数生成器设定种子
分配 SP socket 的全局表
如果存在错误信息，就输出
分配未使用的文件描述符的栈空间
初始化传输方式和 socket 类型的全局状态
添加传输方式 inproc,ipc,tcp,ws
添加 socket 类型 pair,xpair,pub,sub,xpub,xsub,rep,req,xrep,xreq,
push,xpush,pull,xpull,respondent,surveyor,xrespondent,xsurveyor,bus,
xbus
开启工作线程
结束运行：
全局环境终止的私有函数
如果没有 sockets 剩余，解构全局环境
关闭工作线程
让所有的 transport 收回他们的全局资源
从列表中移除 socket 类型
终止 socktypes，transports 列表，释放 socks 的空间，并指向 null
关闭内存管理子系统
（如果是 windows 系统）解构 socket 库

2.2.1、aio

线程池模拟的异步操作，带状态机的事件驱动

功能列举：
AIO 子系统的环境搭建 nn_ctx 系列函数
有限状态机 fsm 的基类实现
使用数组，队列，链表三种方式实现异步 io（多路传输支持）
线程池的实现
任务计时器的实现
任务的时间链表，各种操作的实现
异步 IO 的基础套接字的实现
任务的各种操作实现

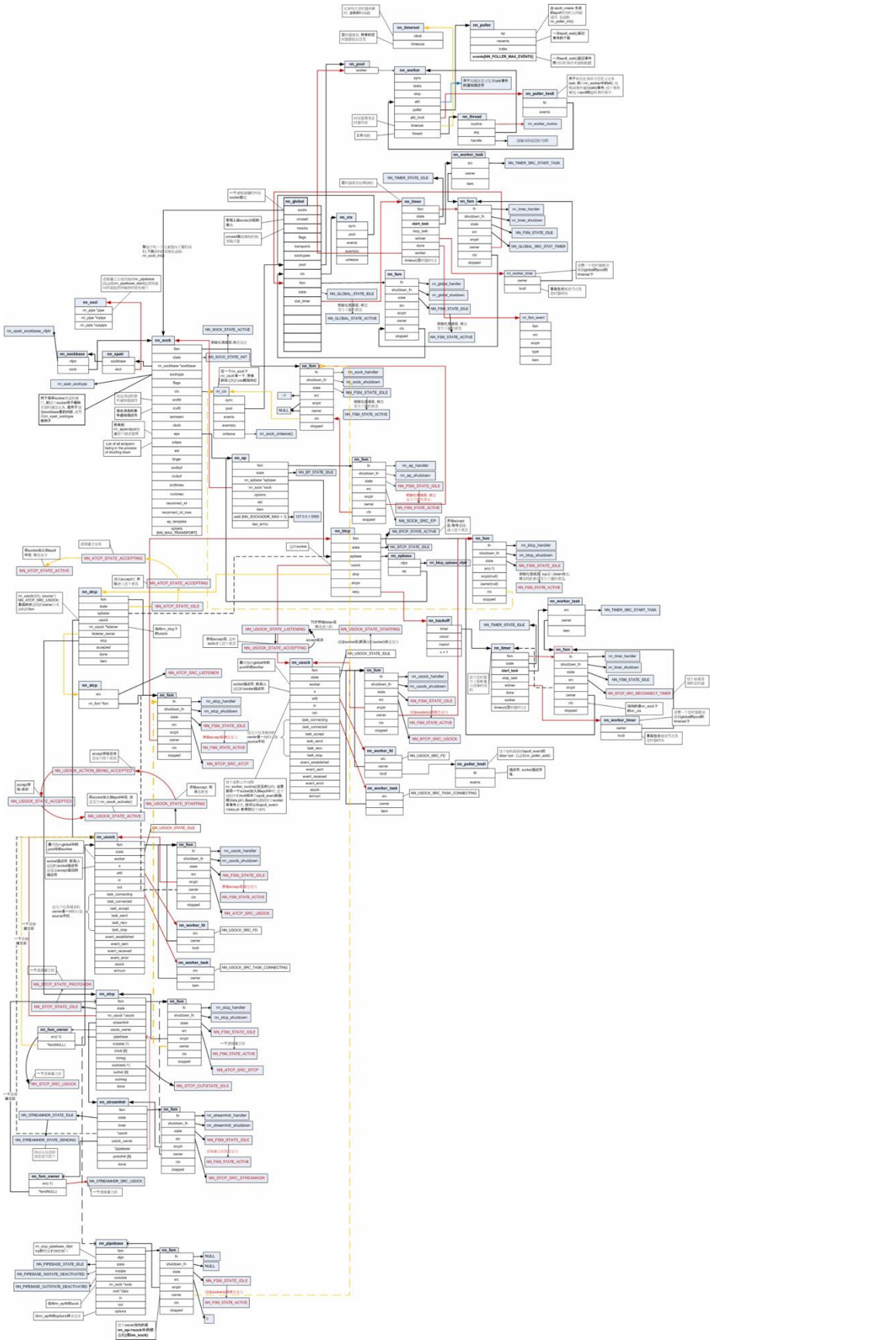
2.2.1、device

传递信息的设备

这个十分强大，可以通过这个装置绑定两个 socket 实现单方面共享和双方面共享信息，就像 qq 的关联账号一样啊哈

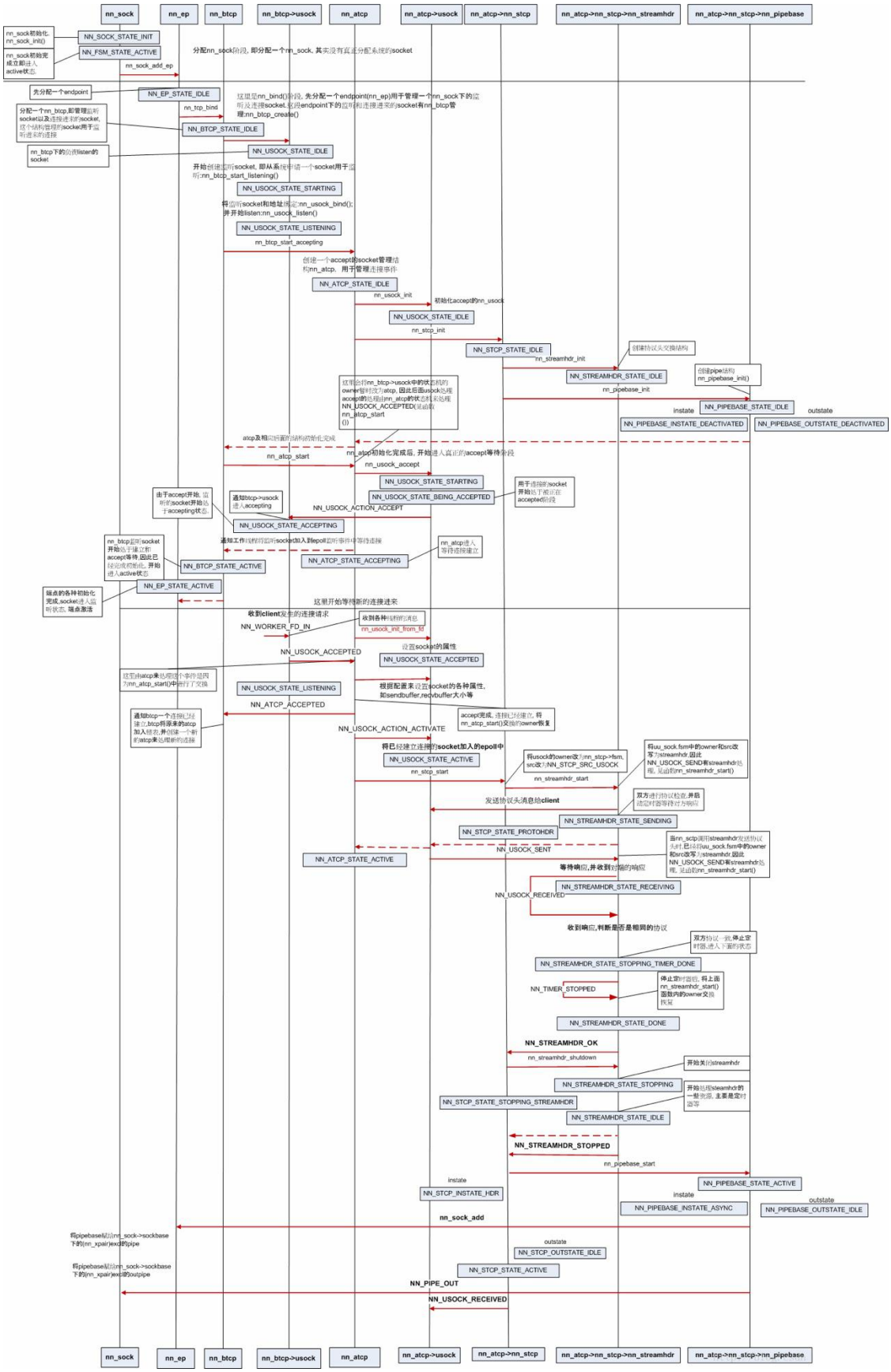
三、数据结构分析

以 PAIR, TCP 的 SERVER 端数据结构为例



四、模块状态分析

以 pair tcp SERVER 端各个模块的状态为例


nn_sock
nn_ep
nn_btcp
nn_btcp->usock
nn_atcp
nn_atcp->usock
nn_atcp->nn_stcp
nn_atcp->nn_stcp->nn_streamhdr
nn_atcp->nn_stcp->nn_pipebase
nn_sock初始化 nn_sock_init()
NN_SOCK_STATE_INIT
nn_sock初始完成立即进入active状态
NN_FSM_STATE_ACTIVE
分配nn_sock阶段, 即分配一个nn_sock, 其实没有真正分配系统的socket
nn_sock_add_ep
先分配一个endpoint
NN_EP_STATE_IDLE
这里是nn_bind()阶段, 先分配一个endpoint(nn_ep)用于管理一个nn_sock下的监听及连接socket, 这个endpoint下的监听和连接进来的socket有nn_btcp管理nn_btcp_create()
nn_tcp_bind
分配一个nn_btcp,即管理监听socket以及连接进来的socket, 这个结构管理的socket用于监听进来的连接
NN_BTCP_STATE_IDLE
nn_btcp下的负责listen的socket
NN_USOCK_STATE_IDLE
开始创建监听socket, 即从系统申请一个socket用于监听:nn_btcp_start_listening()
NN_USOCK_STATE_STARTING
将监听socket和地址绑定:nn_usock_bind();
并开始listen:nn_usock_listen()
NN_USOCK_STATE_LISTENING
nn_btcp_start_accepting
创建一个accept的socket管理结构nn_atcp, 用于管理连接事件
NN_ATCP_STATE_IDLE
nn_usock_init
初始化accept的nn_usock
NN_USOCK_STATE_IDLE
nn_stcp_init
NN_STCP_STATE_IDLE
nn_streamhdr_init
创建协议头交换结构
NN_STREAMHDR_STATE_IDLE
创建pipe结构nn_pipebase_init()
nn_pipebase_init
这里会将nn_btcp->usock中的状态机的owner暂时改为atcp, 因此后面usock处理accept的处理由nn_atcp的状态机来处理
NN_USOCK_ACCEPTED(回调函数nn_atcp_start())
instate
NN_PIPEBASE_STATE_IDLE
outstate
NN_PIPEBASE_INSTATE_DEACTIVATED
NN_PIPEBASE_OUTSTATE_DEACTIVATED
atcp及相应资源的结构初始化完成
nn_atcp初始化完成后, 开始进入真正的accept等待阶段
nn_atcp_start
nn_usock_accept
NN_USOCK_STATE_STARTING
NN_USOCK_STATE_BEING_ACCEPTED
用于连接的socket开始处于相应在accepted阶段
由于accept开始, 监听的socket开始处于accepting状态.
通知btcp->usock进入accepting
NN_USOCK_ACTION_ACCEPT
NN_USOCK_STATE_ACCEPTING
nn_btcp监听socket开始处于建立新accept等待,因此已经完成初始化, 开始进入active状态
通知工作线程将监听socket加入到epoll监听事件中等待连接
nn_atcp进入等待连接建立
NN_BTCP_STATE_ACTIVE
NN_ATCP_STATE_ACCEPTING
端点的各种初始化完成,socket进入监听状态, 端点激活
NN_EP_STATE_ACTIVE
这里开始等待新的连接进来
收到client发来的连接请求
NN_WORKER_FD_IN
收到各种线程的消息
nn_usock_init_from_fd
设置socket的属性
NN_USOCK_ACCEPTED
NN_USOCK_STATE_ACCEPTED
这里将atcp来处理这个事件是因为nn_atcp_start()中进行了交换
NN_USOCK_STATE_LISTENING
根据配置来设置socket的各种属性, 如sendbuffer,recvbuffer大小等
NN_ATCP_ACCEPTED
accept完成, 连接已经建立,将nn_atcp_start()交换的owner恢复
通知btcp一个连接已经建立,btcp将原来的atcp加入链表,并创建一个新的atcp来处理新的连接
NN_USOCK_ACTION_ACTIVATE
将已经建立连接的socket加入的epoll中
NN_USOCK_STATE_ACTIVE
将usock的owner改为nn_stcp->fsm, src改为NN_STCP_SRC_USOCK
将usock.fsm中的owner和src改写为streamhdr,因此NN_USOCK_SEND有streamhdr处理, 见函数nn_streamhdr_start()
nn_stcp_start
nn_streamhdr_start
发送协议头消息给client
双方进行协议检查,并启动定时器等待行为响应
NN_STREAMHDR_STATE_SENDING
NN_STCP_STATE_PROTOHDR
NN_USOCK_SENT
当nn_sctp调用streamhdr发送协议头时,已经将usock.fsm中的owner和src改写为streamhdr,因此NN_USOCK_SEND有streamhdr处理, 见函数nn_streamhdr_start()
NN_ATCP_STATE_ACTIVE
等待响应,并收到对端的响应
NN_STREAMHDR_STATE_RECEIVING
NN_USOCK_RECEIVED
收到响应,判断是否是相同的协议
双方协议一致,停止定时器,进入下面的状态
NN_STREAMHDR_STATE_STOPPING_TIMER_DONE
NN_TIMER_STOPPED
停止定时器后, 将上面nn_streamhdr_start()函数内的owner交换恢复
NN_STREAMHDR_STATE_DONE
NN_STREAMHDR_OK
nn_streamhdr_shutdown
开始关闭streamhdr
NN_STREAMHDR_STATE_STOPPING
开始处理steamhdr的一些资源, 主要是定时器等
NN_STCP_STATE_STOPPING_STREAMHDR
NN_STREAMHDR_STATE_IDLE
NN_STREAMHDR_STOPPED
nn_pipebase_start
instate
NN_PIPEBASE_STATE_ACTIVE
NN_STCP_INSTATE_HDR
instate
outstate
NN_PIPEBASE_INSTATE_ASYNC
NN_PIPEBASE_OUTSTATE_IDLE
nn_sock_add
将pipebase赋给nn_sock->sockbase下的(nn_xpair)exch的pipe
outstate
NN_STCP_OUTSTATE_IDLE
将pipebase赋给nn_sock->sockbase下的(nn_xpair)exch的outpipe
NN_STCP_STATE_ACTIVE
NN_PIPE_OUT
NN_USOCK_RECEIVED

五、测试用例

环境配置

下载[源码](https://github.com/nanomsg/nanomsg)
然后[build](http://nanomsg.org/development.html)
```
POSIX-compliant platforms

First, you have to have autotools installed.
Once that is done following sequence of steps will build the project:
    $ git clone git@github.com:nanomsg/nanomsg.git
    $ cd nanomsg
    $ ./autogen.sh
    $ ./configure
    $ make
    $ make check
    $ sudo make install
To build a version with documentation (man pages and HTML reference)
    included you will need asciidoc and xmlto tools installed.
To build it modify the ./configure step in following manner:
    $ ./configure --enable-doc
To build a version with debug info adjust the ./configure step as follows:
    $ ./configure --enable-debug
```
build 需要先安装 cmake
Mac OS 下
brew install cmake
src 里面是头文件 移动到 main.c 目录下并改名为 nanomsg

xcode
将 libnanomsg.1.0.0-rc2.dylib 添加到工程里面
点击工程->targets->Build Phases->Link Binary With Libraries
添加 libnanomsg.1.0.0-rc2.dylib

shell
将 libnanomsg.1.0.0-rc2.dylib 移到当前目录下
gcc -lnanomsg.1.0.0-rc2 -L. -o main main.c
./main

Ubuntu 下

apt-get install cmake
将 libnanomsg.so.5.0.0 移到/usr/local/lib 目录下
echo "/usr/local/lib" >> /etc/ld.so.conf
ldconfig
gcc main.c -o main -lpthread -lnanomsg
./main

Cent OS 服务器下
yum install cmake
git 失败
scp -r /Users/meteor/github/nanomsg-master root@115.159.36.21:/home
cp libnanomsg.so.5.0.0 /usr/local/lib
echo "/usr/local/lib" >> /etc/ld.so.conf
ldconfig
gcc main.c -o main -lpthread -lnanomsg

测试案例
本地 Mac OS 与 Ubuntu 虚拟机的测试
本地 ipc 通信，pair 模式，bus 模式
分别测试了 Mac OS 的 ip 地址和端口号 tcp://10.189.99.235:5555
Ubuntu 虚拟机的 ip 地址和端口号 tcp://192.168.250.135:5555

两台 Cent OS 服务器和本地 Mac OS 的测试
tcp://115.159.36.21:5555
tcp://115.29.39.184:5555
pair 成功
bus 失败，需要至少 3 个外网 IP 我只有两个啊 orz

ws://115.159.36.21:5555 失败

Mac OS 与 Cent OS 服务器的 tcp 通信
tcp://115.159.36.21:5555
tcp://127.0.0.1:5555
tcp://localhost:5555

ws://115.159.36.21:5555
ws://127.0.0.1:5555
不成功

测试用例

PAIR - simple one-to-one communication

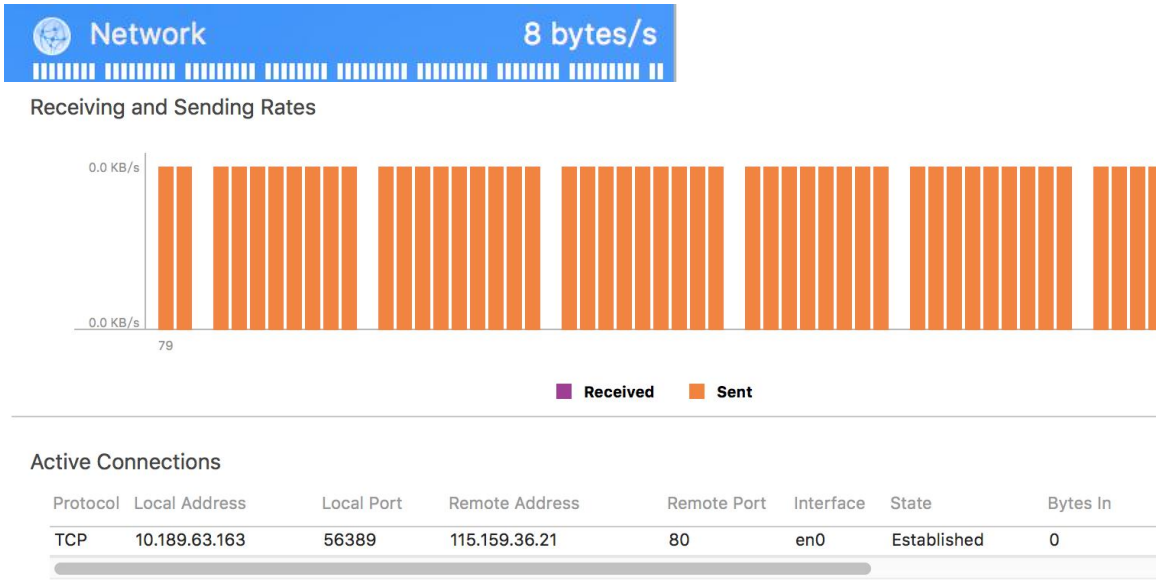
```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>
#include "nanomsg/nn.h"
#include "nanomsg/pair.h"
#include "nanomsg/bus.h"
#include "nanomsg/tcp.h"

void recv_msg(int sock)
{
    char *msg = NULL;
    printf("now you can receive messages...\n");
    while (1) {
        int result = nn_recv(sock, &msg, NN_MSG, 0);
        if (result > 0)
        {
            printf ("RECEIVED \"%s\"\n", msg);
            nn_freemsg (msg);
        }
    }
}

int main (const int argc, const char **argv)
{
    int sock;
    char transport[10];
    // choose transport : bus pair
    printf("please choose the transport...\n");
    while (1) {
        scanf("%s",transport);
```

```
    if (strcmp(transport, "pair")==0)
        sock = nn_socket (AF_SP, NN_PAIR);
    else if (strcmp(transport, "bus")==0)
        sock = nn_socket (AF_SP, NN_BUS);
    else {
        printf("no such transport\n");
        continue;
    }
    if(sock < 0) {
        printf("fail to create socket: %s\n", nn_strerror(errno));
        exit(errno);
    }
    break;
}

char bindOrConnect[10], url[100], next;
int flag;
// choose protocol
// ipc://tmp/pair.ipc
// tcp://115.29.39.184:5555
printf("bind/connect protocol://url\n");
while (1) {
    scanf("%s",bindOrConnect);
    scanf("%s",url);
    if (strcmp(bindOrConnect, "bind")==0)
        flag = nn_bind(sock, url);
    else if (strcmp(bindOrConnect, "connect")==0)
        flag = nn_connect(sock, url);
    else {
        printf("please select bind/connect\n");
        continue;
    }
    if ( flag >= 0 )
```

```
            printf("%s successful\n", bindOrConnect);
        else {
            printf("fail to %s to %s : %s\n", bindOrConnect, url,
nn_strerror(errno));
            continue;
        }
        printf("do you want to do next?(y/n)\n");
        scanf("%c", &next);
        if ( next=='y') {
            break;
        }
        else {
            printf("continue\n");
        }
    }

    int to = 100; // timeout
    if(nn_setsockopt (sock, NN_SOL_SOCKET, NN_RCVTIMEO, &to, sizeof (to))
< 0) {
        printf("fail to set sorket opts: %sn", nn_strerror(errno));
        exit(errno);
    }

    // sub thread: receive message
    pthread_t thread;
    pthread_create(&thread, NULL, (void *)(&recv_msg), (void *)sock);

    // main thread: send message
    char msg[1024];
    printf("now you can send messages...\n");
    while(1) {
        scanf("%s", msg);
        if (strcmp(msg, "q")==0)
```

```
            break;
        printf ("SENDING \"%s\"\n", msg);
        size_t sz_n = strlen (msg) + 1;
        nn_send(sock, msg, sz_n, 0);
    }
    printf("exit\n");
    nn_shutdown(sock, 0);
    return 0;
}
```

从下两个图可以看出它保持连接的方法是，以 8bytes/s 的速度发送 8～9 秒，然后暂停一次，然后本地的端口每次都在加 1。



可以看出该库对系统内存占用，网路宽带占用都非常少，是一个轻量级应用。

六、代码详细分析

见 readme.md 或者https://github.com/MeteorKL/nanomsg-analyse

七、简单的个人感想

我应该会在后期继续完善，这个报告的时间实在是太仓促了，而且自己的能力也不够，面向对象的思想在c里面实现真是令我猝不及防，这么灵活的使用c我还是第一次见到，不过不得不说这令我学到了很多。特别是和linux程序设计基础里面学到的socket编程形成了鲜明的对比，那时候写一个简单的聊天室无比的复杂，而现在只需要用一下bus模式就行了，不得不佩服这些造轮子的人。

八、参考

源码来自 github
部分分析参考 Tiger's Blog
简介参考 nanomsg.org
测试用例参考 Tim Dysinger's Blog