Nanomsg 代码分析 V0.1

MeteorKL 2016 年 06 月

<https://github.com/MeteorKL/nanomsg>

目录

# 一、nanomsg 简单介绍

## 1.1、概述

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](http://nanomsg.org/documentation-zeromq.html)和[Martin Sustrik](http://250bpm.com/)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。

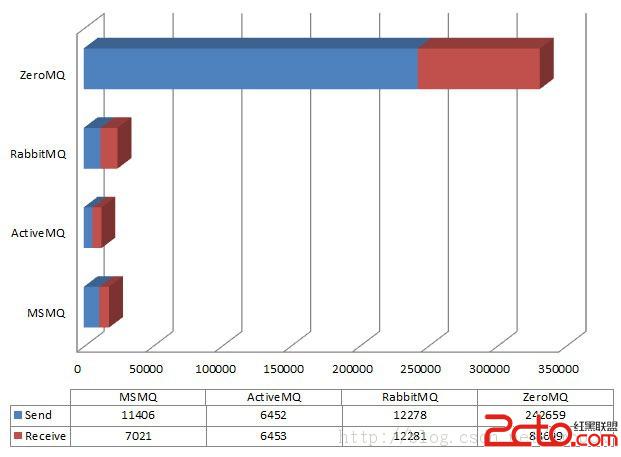
5.负载均衡算法——ZeroMQ采用了轮转调度算法。虽然该算法可以平均分配工作，但也有其局限性。比如，有两个数据中心，一个在伦敦，一个在纽约。在理想情况下，一个位于伦敦数据中心的网站，其请求不应该路由到纽约。但在ZeroMQ的负载均衡算法里，这完全有可能。而nanomsg避免了这种情况的出现。

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](http://mikehadlow.blogspot.com/2011/04/message-queue-shootout.html)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](https://github.com/mikehadlow/Mike.MQShootout)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）



## 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：

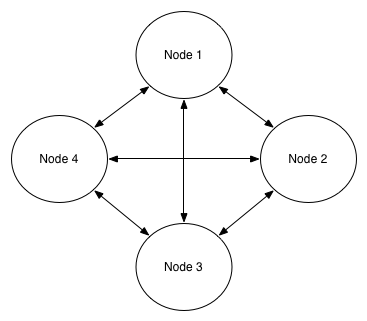
* PAIR - simple one-to-one communication

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



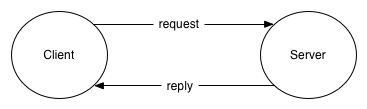
* BUS - simple many-to-many communication

总线模式：简单的多对多的通信



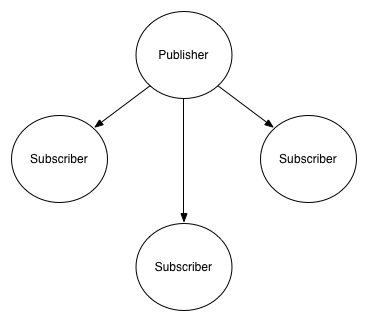
* Request/Reply - allows to build clusters of stateless services to process user requests

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



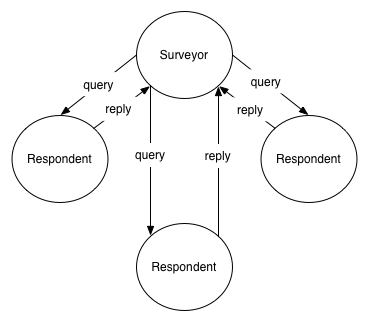
* PIPELINE - aggregates messages from multiple sources and load balances them among many destinations

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](https://www.zhihu.com/question/20215561)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：

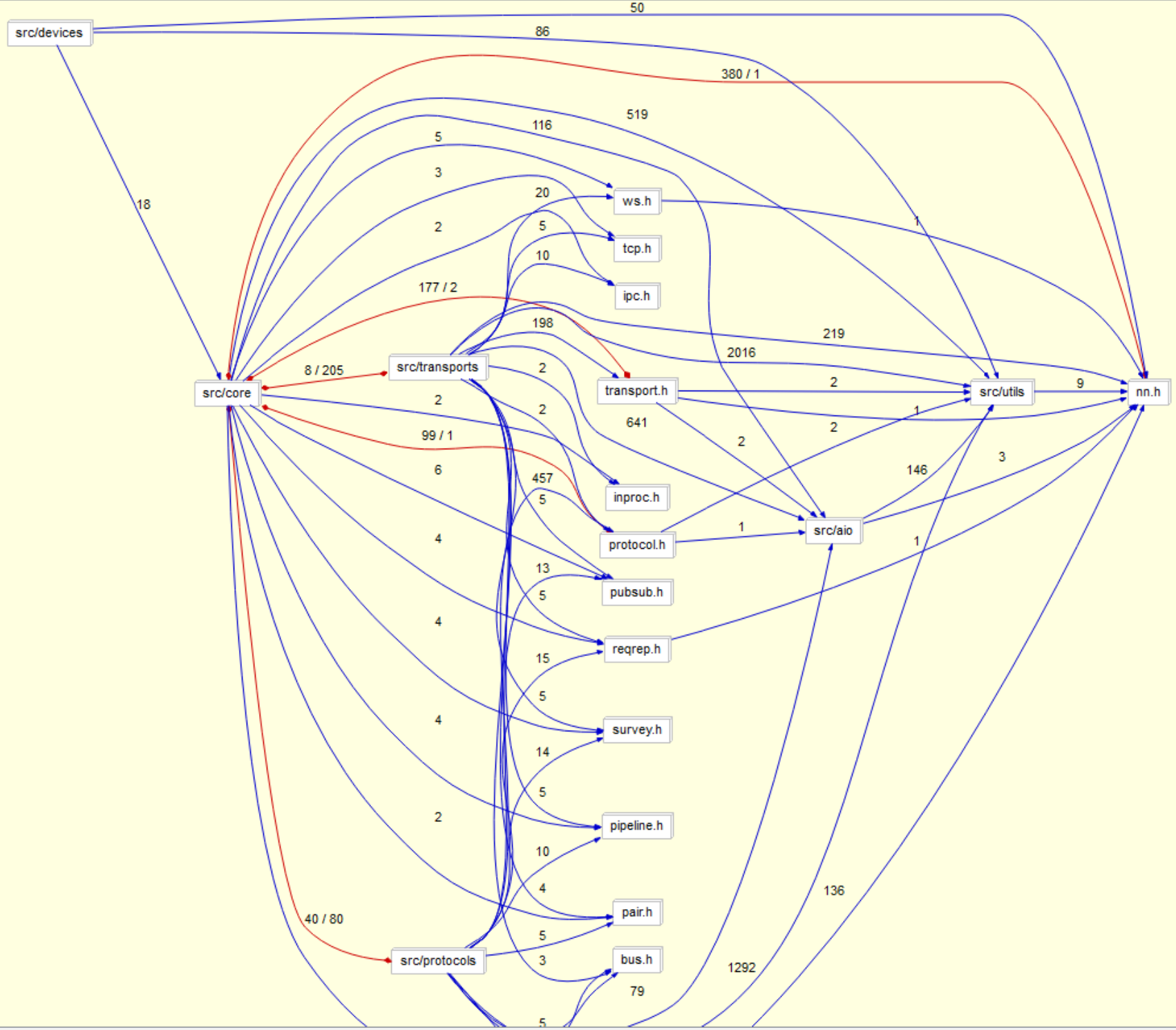
|  |
| --- |
| char buf[10];  int s = nn\_socket(AF\_SP, NN\_REP);  nn\_bind(s, "tcp://\*:5555");  nn\_recv(s, buf, 10, 0);  nn\_send(s, "World", 5, 0);  nn\_close(s); |

对应的客户端请求程序大致为：

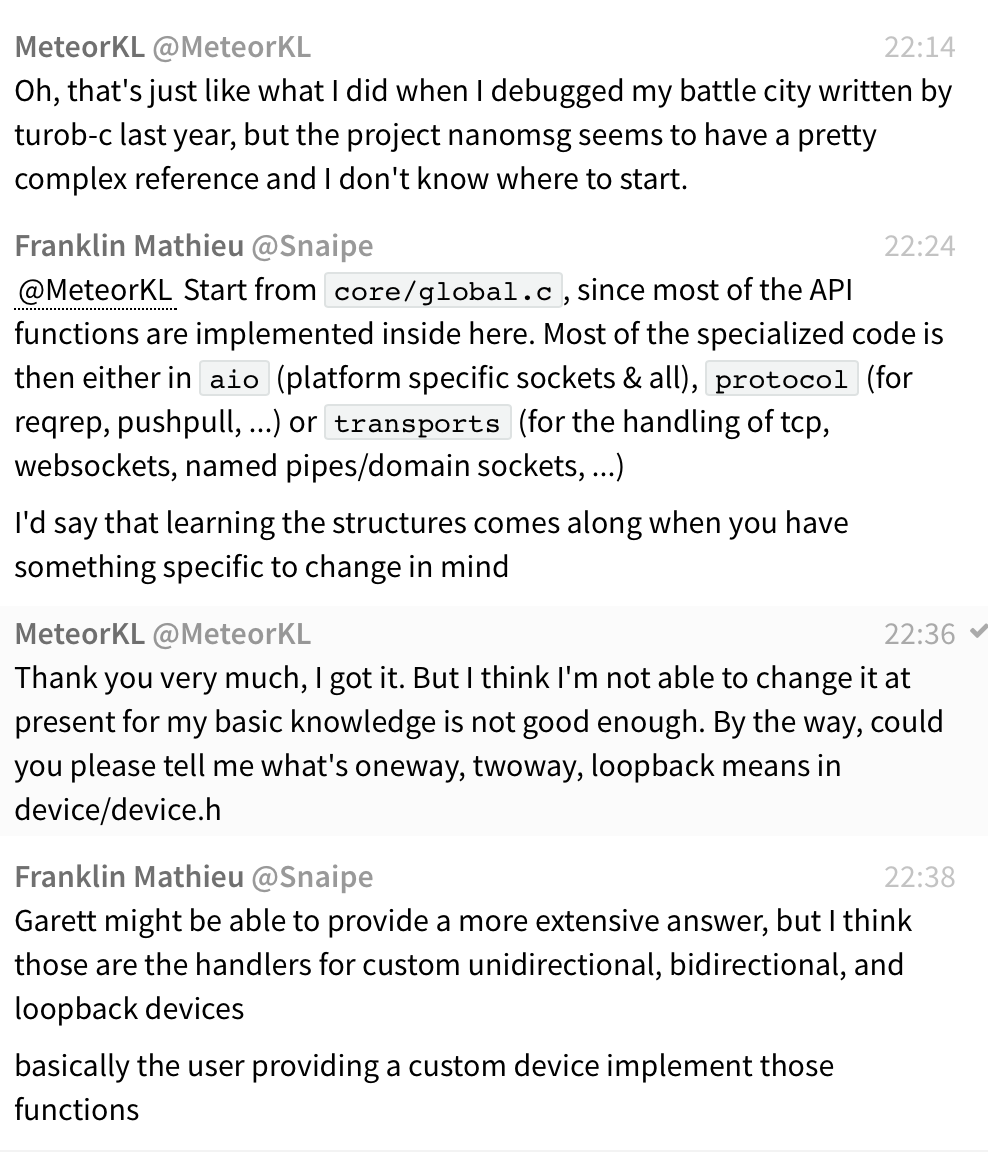
|  |
| --- |
| char buf[10];  int s = nn\_socket(AF\_SP, NN\_REQ);  nn\_connect(s, "tcp://localhost:5555");  nn\_send(s, "Hello", 5, 0);  nn\_recv(s, buf, 10, 0);  printf("Hello %sn", buf);  nn\_close(s); |

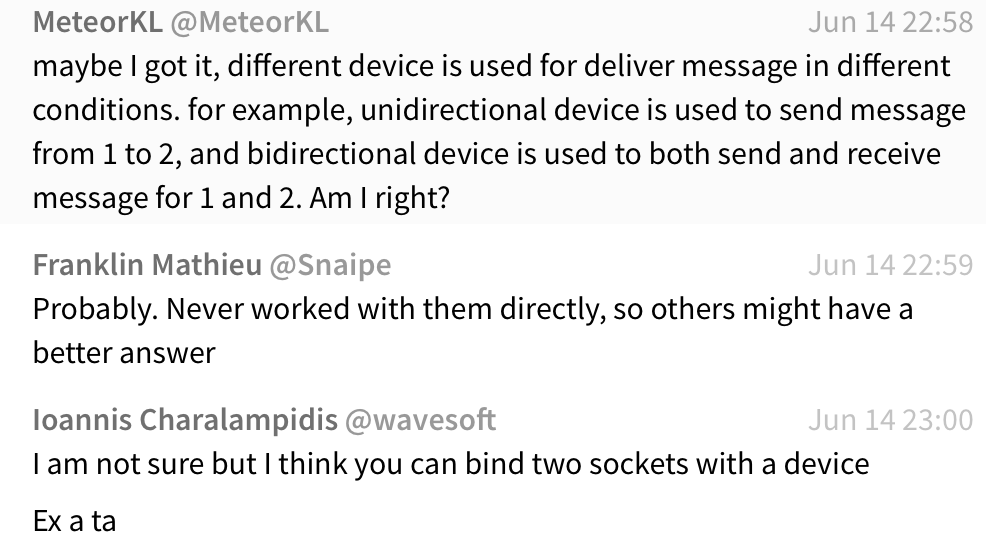
# 二、nanomsg 模块详细分析

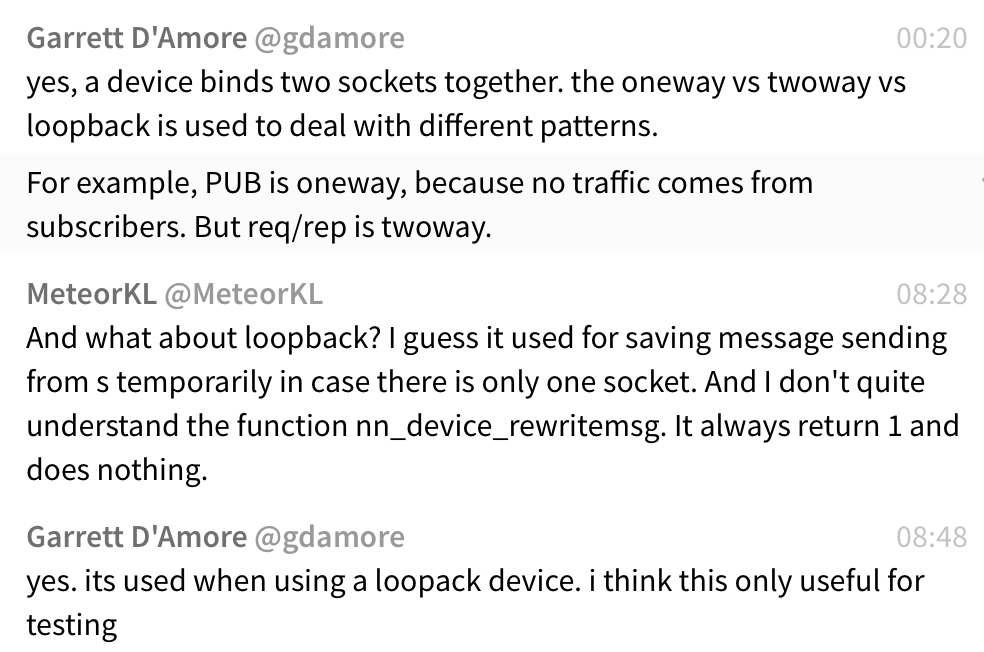
用understand软件分析文件夹引用情况，可以认为该系统主要由transport，protocol，aio，core这四个模块。



令人惊喜的是，我在官网上还发现了一个[nanomsg的gitter chatroom](https://gitter.im/nanomsg/nanomsg)，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







## 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

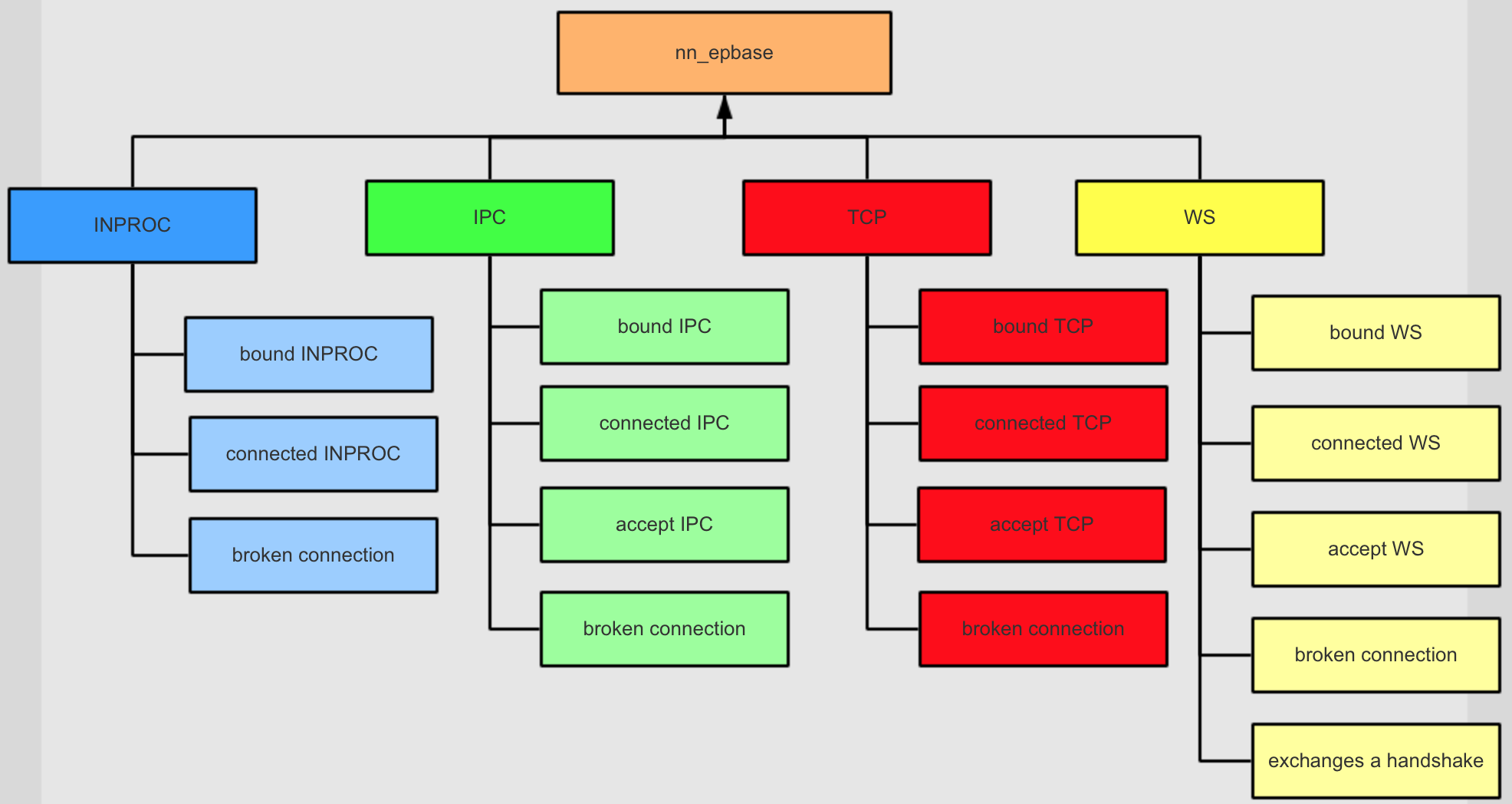
## 2.2、各大模块分析

### 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

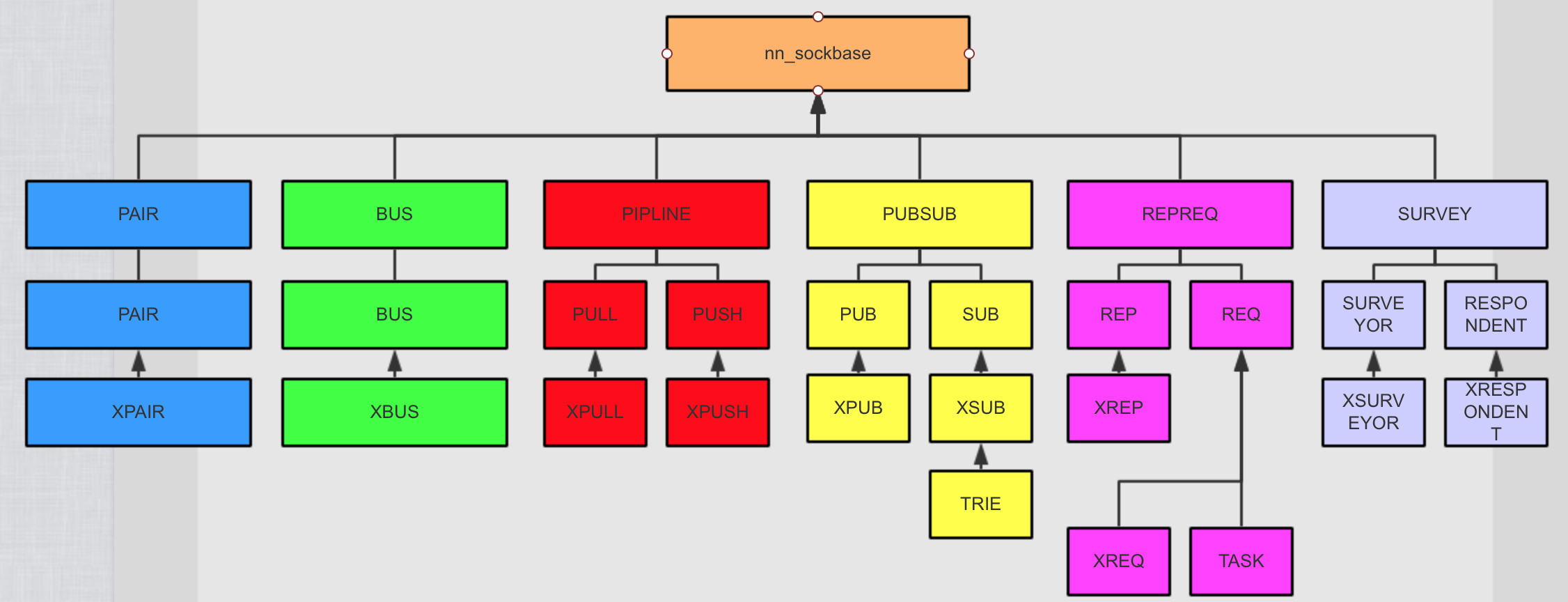
IPC（进程间通信）传输方式的实现，由icp的连接，绑定，接受，对方icp连接断开这四种情况的状态机支持

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

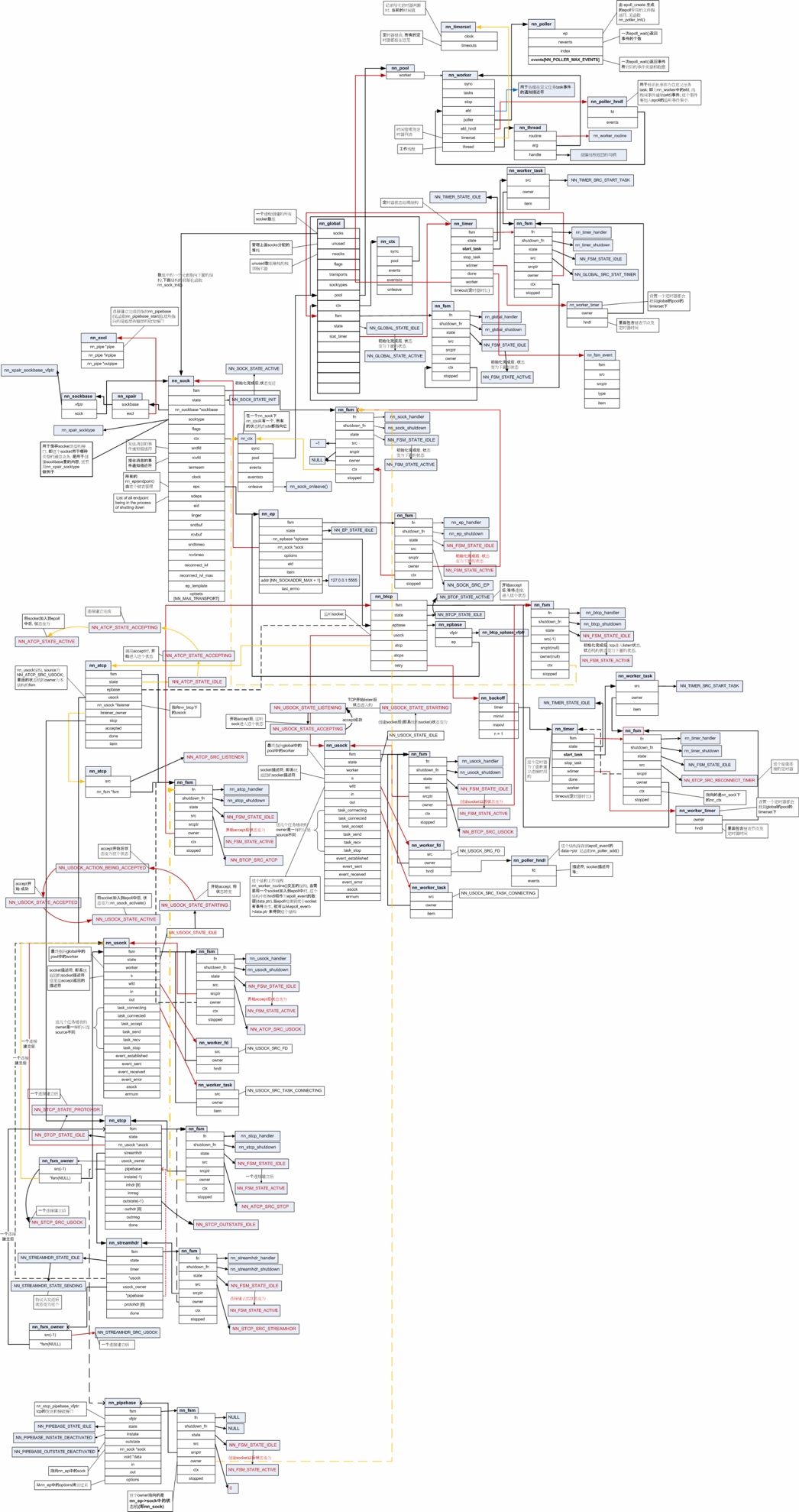
### 2.2.1、device

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

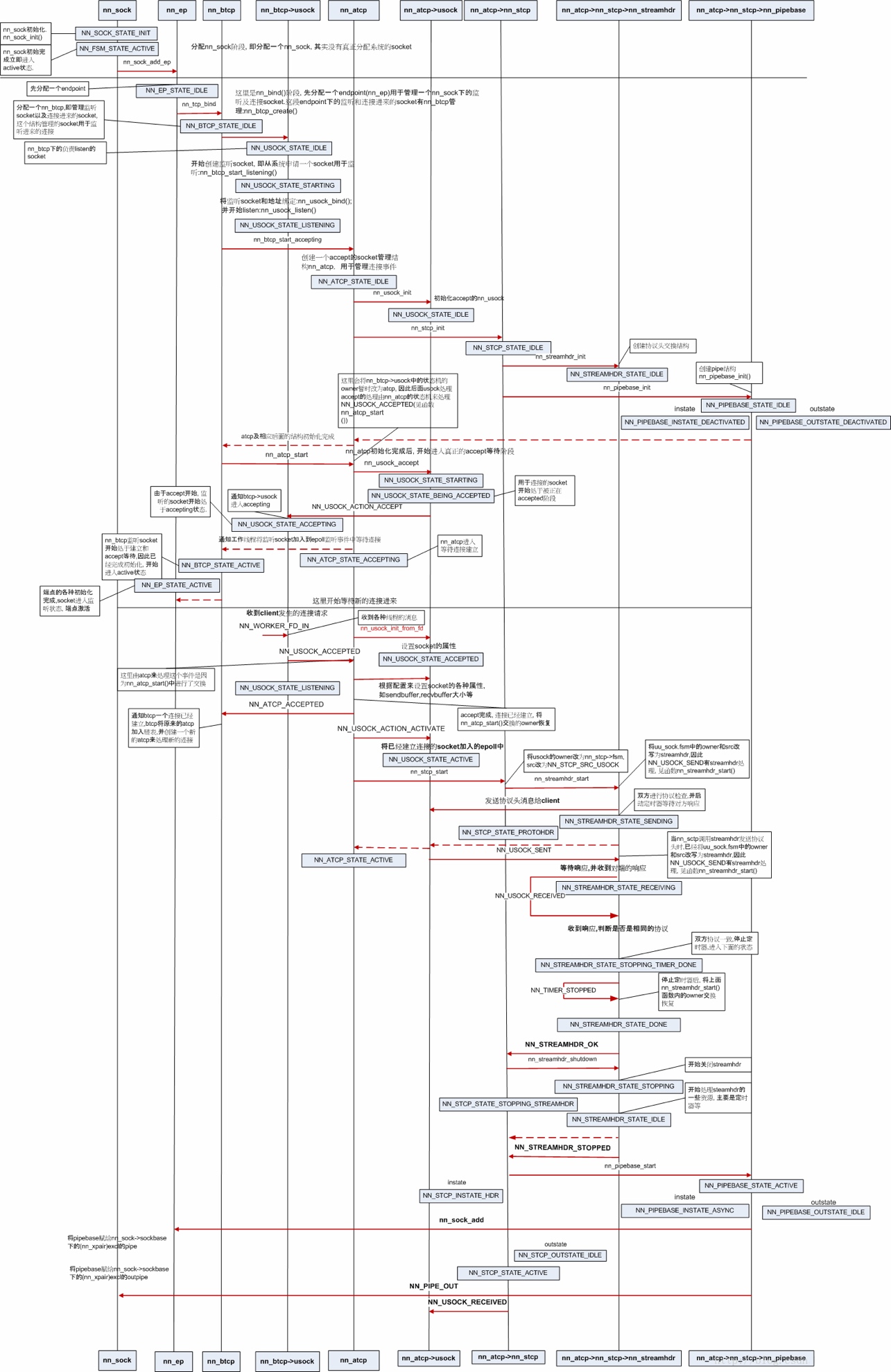
# 三、数据结构分析

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



# 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

#### 已测试案例

##### Mac OS与Ubuntu虚拟机的tcp通信pair模式

本地ipc通信，pair模式，bus模式

##### Mac OS与Ubuntu虚拟机的tcp通信pair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

##### 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"

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)

{

char url[100];

int sock = nn\_socket (AF\_SP, NN\_PAIR);

if(sock < 0) {

fprintf(stderr, "fail to create socket: %s\n", nn\_strerror(errno));

exit(errno);

}

// 本机测试，本地局域网测试

printf("please input the url...\n");

scanf("%s",url); // ipc:///tmp/pair.ipc 两个进程间的通信

if ( nn\_bind(sock, url) >= 0 )

printf("bind successful\n");

else {

fprintf(stderr, "fail to bind to %s : %s\n", url, nn\_strerror(errno));

if ( nn\_connect(sock, url) >= 0 )

printf("connect successful\n");

else {

fprintf(stderr, "fail to connect to %s : %s\n", url, nn\_strerror(errno));

exit(errno);

}

}

// 联网测试

// printf("please input the bind url...\n");

// scanf("%s",url); // ipc:///tmp/pair.ipc 两个进程间的通信

// if ( nn\_bind(sock, url) >= 0 )

// printf("bind successful\n");

// else {

// fprintf(stderr, "fail to bind to %s : %s\n", url, nn\_strerror(errno));

// exit(errno);

// }

//

// printf("please input the connect url...\n");

// scanf("%s",url); // ipc:///tmp/pair.ipc 两个进程间的通信

// if ( nn\_connect(sock, url) >= 0 )

// printf("connect successful\n");

// else {

// fprintf(stderr, "fail to connect to %s : %s\n", url, nn\_strerror(errno));

// exit(errno);

// }

int to = 100;

if(nn\_setsockopt (sock, NN\_SOL\_SOCKET, NN\_RCVTIMEO, &to, sizeof (to)) < 0) {

fprintf(stderr, "fail to set sorket opts: %sn", nn\_strerror(errno));

exit(errno);

}

pthread\_t thread;

// 创建一个线程接收信息

pthread\_create(&thread, NULL, (void \*)(&recv\_msg), (void \*)sock);

// 主线程发送信息

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; // '\0' too

nn\_send(sock, msg, sz\_n, 0);

}

printf("exit\n");

nn\_shutdown(sock, 0);

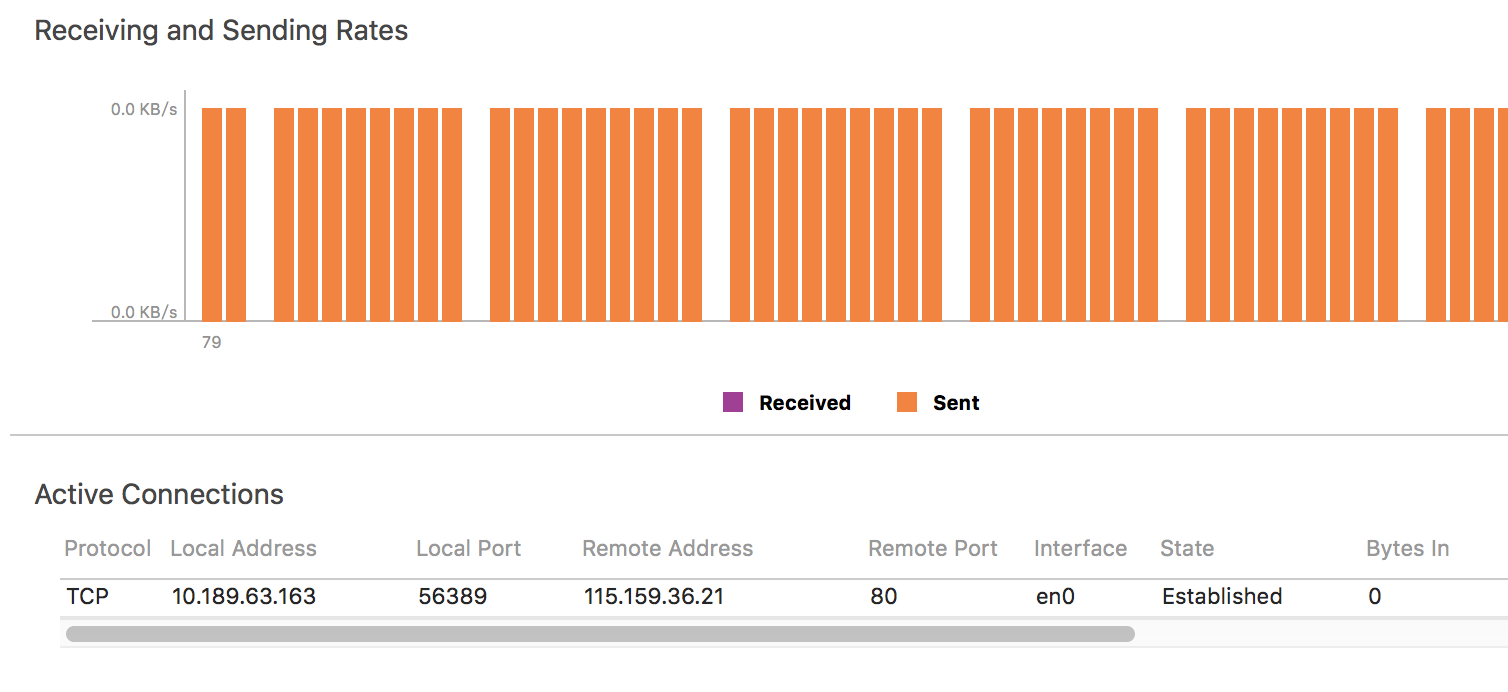
return 0;

}

```

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、参考

源码来自[github](https://github.com/nanomsg/nanomsg)

部分分析参考[Tiger's Blog](http://absolutetiger.com/?p=225)

简介参考[nanomsg.org](http://nanomsg.org/index.html)

测试用例参考[Tim Dysinger's Blog](http://tim.dysinger.net/posts/2013-09-16-getting-started-with-nanomsg.html)