A Look at Nanomsg and Scalability Protocols (Why ZeroMQ Shouldn’t Be Your First Choice)

Earlier this month, I explored ZeroMQ and how it proves to be a promising solution for building fast, high-throughput, and scalable distributed systems. Despite lending itself quite well to these types of problems, ZeroMQ is not without its flaws. Its creators have attempted to rectify many of these shortcomings through spiritual successors Crossroads I/O and nanomsg.

The now-defunct Crossroads I/O is a proper fork of ZeroMQ with the true intention being to build a viable commercial ecosystem around it. Nanomsg, however, is a *reimagining* of ZeroMQ—a complete rewrite in C[1]. It builds upon ZeroMQ’s rock-solid performance characteristics while providing several vital improvements, both internal and external. It also attempts to address many of the strange behaviors that ZeroMQ can often exhibit. Today, I’ll take a look at what differentiates nanomsg from its predecessor and implement a use case for it in the form of service discovery.

Nanomsg vs. ZeroMQ

A common gripe people have with ZeroMQ is that it doesn’t provide an API for new transport protocols, which essentially limits you to TCP, PGM, IPC, and ITC. Nanomsg addresses this problem by providing a pluggable interface for transports and messaging protocols. This means support for new transports (e.g. WebSockets) and new messaging patterns beyond the standard set of PUB/SUB, REQ/REP, etc.

Nanomsg is also fully POSIX-compliant, giving it a cleaner API and better compatibility. No longer are sockets represented as void pointers and tied to a context—simply initialize a new socket and begin using it in one step. With ZeroMQ, the context internally acts as a storage mechanism for global state and, to the user, as a pool of I/O threads. This concept has been completely removed from nanomsg.

In addition to POSIX compliance, nanomsg is hoping to be interoperable at the API and protocol levels, which would allow it to be a drop-in replacement for, or otherwise interoperate with, ZeroMQ and other libraries which implement ZMTP/1.0 and ZMTP/2.0. It has yet to reach full parity, however.

ZeroMQ has a fundamental flaw in its architecture. Its sockets are not thread-safe. In and of itself, this is not problematic and, in fact, is beneficial in some cases. By isolating each object in its own thread, the need for semaphores and mutexes is removed. Threads don't touch each other and, instead, concurrency is achieved with message passing. This pattern works well for objects managed by worker threads but breaks down when objects are managed in user threads. If the thread is executing another task, the object is blocked. Nanomsg does away with the one-to-one relationship between objects and threads. Rather than relying on message passing, interactions are modeled as sets of state machines. Consequently, nanomsg sockets are thread-safe.

Nanomsg has a number of other internal optimizations aimed at improving memory and CPU efficiency. ZeroMQ uses a simple trie structure to store and match PUB/SUB subscriptions, which performs nicely for sub-10,000 subscriptions but quickly becomes unreasonable for anything beyond that number. Nanomsg uses a space-optimized trie called a radix tree to store subscriptions. Unlike its predecessor, the library also offers a true zero-copy API which greatly improves performance by allowing memory to be copied from machine to machine while completely bypassing the CPU.

ZeroMQ implements load balancing using a round-robin algorithm. While it provides equal distribution of work, it has its limitations. Suppose you have two datacenters, one in New York and one in London, and each site hosts instances of "foo" services. Ideally, a request made for foo from New York shouldn't get routed to the London datacenter and vice versa. With ZeroMQ's round-robin balancing, this is entirely possible unfortunately. One of the new user-facing features that nanomsg offers is priority routing for outbound traffic. We avoid this latency problem by assigning priority one to foo services hosted in New York for applications also hosted there. Priority two is then assigned to foo services hosted in London, giving us a failover in the event that foos in New York are unavailable.

Additionally, nanomsg offers a command-line tool for interfacing with the system called nanocat. This tool lets you send and receive data via nanomsg sockets, which is useful for debugging and health checks.

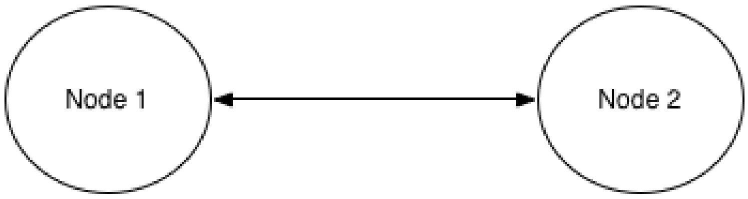
Scalability Protocols

Perhaps most interesting is nanomsg's philosophical departure from ZeroMQ. Instead of acting as a generic networking library, nanomsg intends to provide the "Lego bricks" for building scalable and performant distributed systems by implementing what it refers to as "scalability protocols." These scalability protocols are communication patterns which are an abstraction on top of the network stack's transport layer. The protocols are fully separated from each other such that each can embody a well-defined distributed algorithm. The intention, as stated by nanomsg's author Martin Sustrik, is to have the protocol specifications standardized through the IETF.

Nanomsg currently defines six different scalability protocols: PAIR, REQREP, PIPELINE, BUS, PUBSUB, and SURVEY.

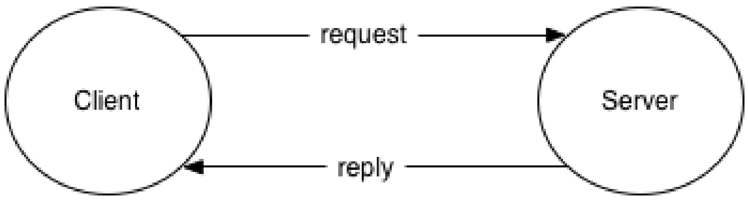
PAIR (Bidirectional Communication)

PAIR implements simple one-to-one, bidirectional communication between two endpoints. Two nodes can send messages back and forth to each other.



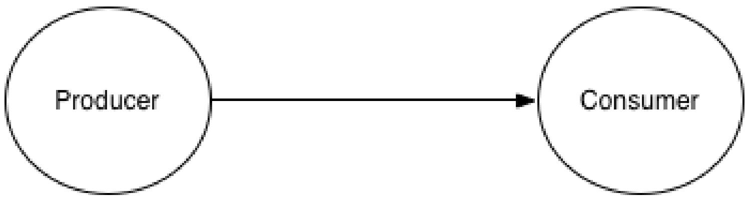
REQREP (Client Requests, Server Replies)

The REQREP protocol defines a pattern for building stateless services to process user requests. A client sends a request, the server receives the request, does some processing, and returns a response.



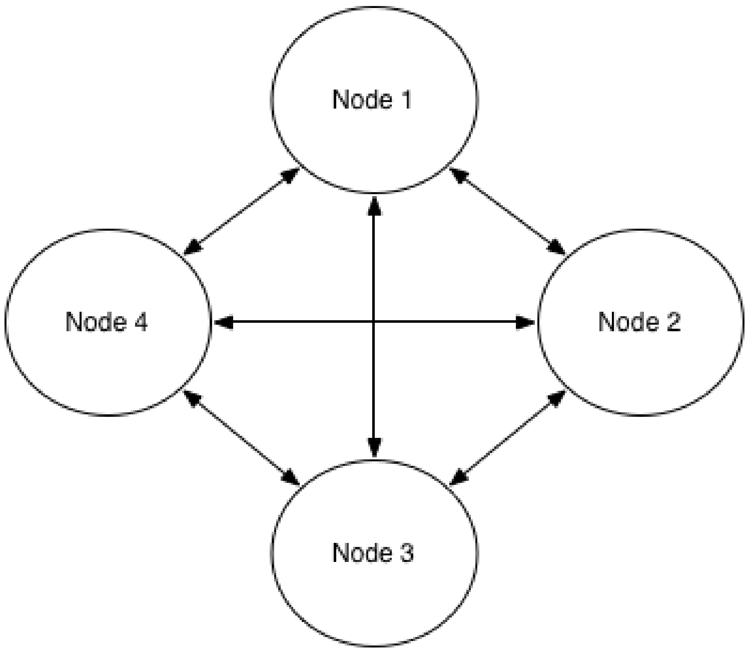
PIPELINE (One-Way Dataflow)

PIPELINE provides unidirectional dataflow which is useful for creating load-balanced processing pipelines. A producer node submits work that is distributed among consumer nodes.



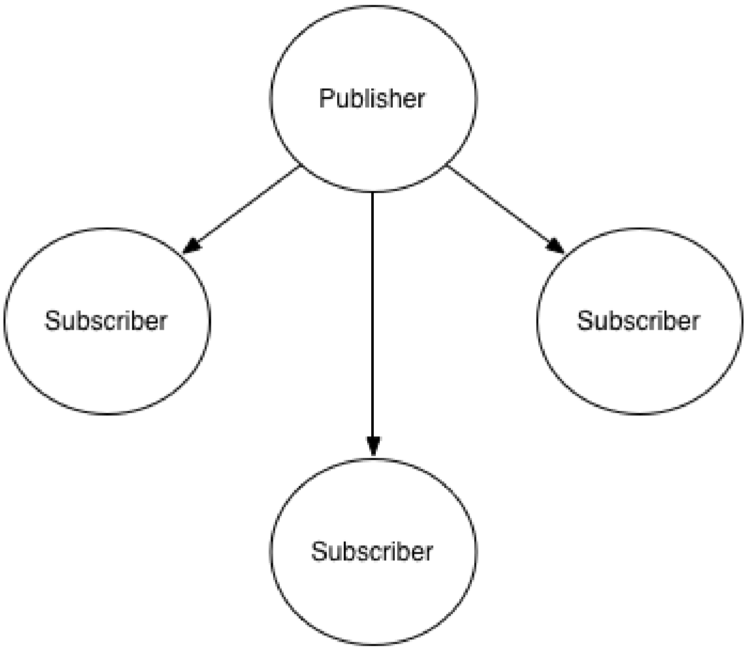
BUS (Many-to-Many Communication)

BUS allows messages sent from each peer to be delivered to every other peer in the group.



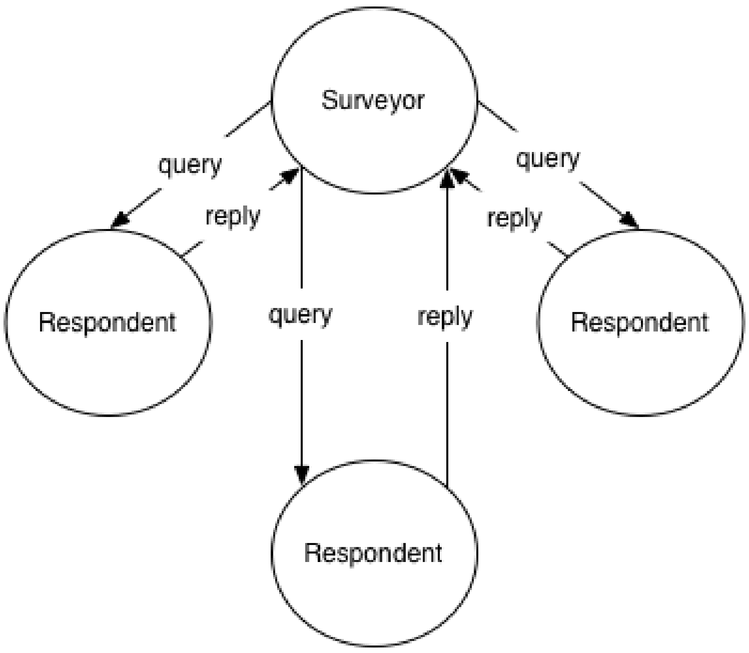
PUBSUB (Topic Broadcasting)

PUBSUB allows publishers to multicast messages to zero or more subscribers. Subscribers, which can connect to multiple publishers, can subscribe to specific topics, allowing them to receive only messages that are relevant to them.



SURVEY (Ask Group a Question)

The last scalability protocol, and the one in which I will further examine by implementing a use case with, is SURVEY. The SURVEY pattern is similar to PUBSUB in that a message from one node is broadcasted to the entire group, but where it differs is that each node in the group *responds* to the message. This opens up a wide variety of applications because it allows you to quickly and easily query the state of a large number of systems in one go. The survey respondents must respond within a time window configured by the surveyor.



Implementing Service Discovery

As I pointed out, the SURVEY protocol has a lot of interesting applications. For example:

- What data do you have for this record?
- What price will you offer for this item?
- Who can handle this request?

To continue exploring it, I will implement a basic service-discovery pattern. Service discovery is a pretty simple question that's well-suited for SURVEY: what services are out there? Our solution will work by periodically submitting the question. As services spin up, they will connect with our service discovery system so they can identify themselves. We can tweak parameters like how often we survey the group to ensure we have an accurate list of services and how long services have to respond.

This is great because 1) the discovery system doesn't need to be aware of what services there are—it just blindly submits the survey—and 2) when a service spins up, it will be discovered and if it dies, it will be "undiscovered."

Here is the ServiceDiscovery class:

```python
from collections import defaultdict
import random

from nanomsg import NanoMsgAPIError
from nanomsg import Socket
from nanomsg import SURVEYOR
from nanomsg import SURVEYOR_DEADLINE

class ServiceDiscovery(object):

    def __init__(self, port, deadline=5000):
        self.socket = Socket(SURVEYOR)
        self.port = port
        self.deadline = deadline
        self.services = defaultdict(set)

    def bind(self):
        self.socket.bind('tcp://*:%s' % self.port)
        self.socket.set_int_option(SURVEYOR, SURVEYOR_DEADLINE, self.deadline)

    def discover(self):
        if not self.socket.is_open():
            return self.services

        self.services = defaultdict(set)
        self.socket.send('service query')

        while True:
            try:
                response = self.socket.recv()
            except NanoMsgAPIError:
                break

            service, address = response.split('|')
            self.services[service].add(address)

        return self.services

    def resolve(self, service):
        providers = self.services[service]

        if not providers:
            return None

        return random.choice(tuple(providers))

    def close(self):
        self.socket.close()
```

nanomsg_service_discovery.py hosted with ❤ by **GitHub** **view raw**

The discover method submits the survey and then collects the responses. Notice we construct a SURVEYOR socket and set the SURVEYOR_DEADLINE option on it. This deadline is the number of milliseconds from when a survey is submitted to when a response must be received—adjust it accordingly based on your network topology. Once the survey deadline has been reached, a NanoMsgAPIError is raised and we break the loop. The resolve method will take the name of a service and randomly select an available provider from our discovered services.

We can then wrap ServiceDiscovery with a daemon that will periodically run discover.

```
import os
import time

```

```python
from service_discovery import ServiceDiscovery

DEFAULT_PORT = 5555
DEFAULT_DEADLINE = 5000
DEFAULT_INTERVAL = 2000

def start_discovery(port, deadline, interval):
    discovery = ServiceDiscovery(port, deadline=deadline)
    discovery.bind()

    print 'Starting service discovery [port: %s, deadline: %s, interval: %s]' \
        % (port, deadline, interval)

    while True:
        print discovery.discover()
        time.sleep(interval / 1000)

if __name__ == '__main__':
    port = int(os.environ.get('PORT', DEFAULT_PORT))
    deadline = int(os.environ.get('DEADLINE', DEFAULT_DEADLINE))
    interval = int(os.environ.get('INTERVAL', DEFAULT_INTERVAL))

    start_discovery(port, deadline, interval)
```

nanomsg_discovery_daemon.py hosted with ❤ by **GitHub** **view raw**

The discovery parameters are configured through environment variables which I inject into a Docker container.

Services must connect to the discovery system when they start up. When they receive a survey, they should respond by identifying what service they provide and where the service is located. One such service might look like the following:

```python
import os
from threading import Thread

from nanomsg import REP
from nanomsg import RESPONDENT
from nanomsg import Socket

DEFAULT_DISCOVERY_HOST = 'localhost'
DEFAULT_DISCOVERY_PORT = 5555
DEFAULT_SERVICE_NAME = 'foo'
DEFAULT_SERVICE_PROTOCOL = 'tcp'
DEFAULT_SERVICE_HOST = 'localhost'
```

```python
DEFAULT_SERVICE_PORT = 9000

def register_service(service_name, service_address, discovery_host,
                     discovery_port):
    socket = Socket(RESPONDENT)
    socket.connect('tcp://%s:%s' % (discovery_host, discovery_port))

    print 'Starting service registration [service: %s %s, discovery: %s:%s]' \
        % (service_name, service_address, discovery_host, discovery_port)

    while True:
        message = socket.recv()
        if message == 'service query':
            socket.send('%s|%s' % (service_name, service_address))

def start_service(service_name, service_protocol, service_port):
    socket = Socket(REP)
    socket.bind('%s://*:%s' % (service_protocol, service_port))

    print 'Starting service %s' % service_name

    while True:
        request = socket.recv()
        print 'Request: %s' % request
        socket.send('The answer is 42')

if __name__ == '__main__':
    discovery_host = os.environ.get('DISCOVERY_HOST', DEFAULT_DISCOVERY_HOST)
    discovery_port = os.environ.get('DISCOVERY_PORT', DEFAULT_DISCOVERY_PORT)
    service_name = os.environ.get('SERVICE_NAME', DEFAULT_SERVICE_NAME)
    service_host = os.environ.get('SERVICE_HOST', DEFAULT_SERVICE_HOST)
    service_port = os.environ.get('SERVICE_PORT', DEFAULT_SERVICE_PORT)
    service_protocol = os.environ.get('SERVICE_PROTOCOL',
                                      DEFAULT_SERVICE_PROTOCOL)

    service_address = '%s://%s:%s' % (service_protocol, service_host,
                                      service_port)

    Thread(target=register_service, args=(service_name, service_address,
                                          discovery_host,
                                          discovery_port)).start()

    start_service(service_name, service_protocol, service_port)
```

nanomsg_service.py hosted with ❤ by **GitHub** **view raw**

Once again, we configure parameters through environment variables set on a container. Note that we connect to the discovery system with a RESPONDENT socket which then responds to service queries with the service name and address. The service itself uses a REP socket that simply responds to any requests with "The answer is 42," but it could take any number of forms such as HTTP, raw socket, etc.

The full code for this example, including Dockerfiles, can be found on GitHub.

Nanomsg or ZeroMQ?

Based on all the improvements that nanomsg makes on top of ZeroMQ, you might be wondering why you would use the latter at all. Nanomsg is still relatively young. Although it has numerous language bindings, it hasn't reached the maturity of ZeroMQ which has a thriving development community. ZeroMQ has extensive documentation and other resources to help developers make use of the library, while nanomsg has very little. Doing a quick Google search will give you an idea of the difference (about 500,000 results for ZeroMQ to nanomsg's 13,500).

That said, nanomsg's improvements and, in particular, its scalability protocols make it very appealing. A lot of the strange behaviors that ZeroMQ exposes have been resolved completely or at least mitigated. It's actively being developed and is quickly gaining more and more traction. Technically, nanomsg has been in beta since March, but it's starting to look production-ready if it's not there already.

1. The author explains why he should have originally written ZeroMQ in C instead of C++. [↩]

June 29, 2014 Tyler Treat Design Patterns, Distributed Systems, Messaging, Python, Software Architecture, Software Engineering crossroads i/o, distributed systems, message queues, messaging, nanomsg, python, scalability, scalability protocols, service discovery, zeromq

10 thoughts on "A Look at Nanomsg and Scalability Protocols (Why ZeroMQ Shouldn't Be Your First Choice)"

Pingback: Dissecting Message Queues | Brave New Geek

Garrett D'Amore

July 17, 2014 at 1:32 pm

Btw, there is a “pure Go” implementation of the Scalability Protocols that I wrote called mangos — https://bitbucket.org/gdamore/mangos — in my tests I found it offer performance competitive to nanomsg while offering a more idiomatic Go implementation and interface.

Victor

July 18, 2014 at 9:38 am

@Garrett do you consider your implementation to be production ready? are you using it in production yourself?

Garrett D'Amore

April 15, 2015 at 8:26 am

Btw, we are using mangos in production these days.

Thomasi

November 3, 2014 at 8:04 am

I really like nanomsg, but it is not as stable as zeromq yet. We have had huge problems with pubsub scalability that’s very hard to diagnose. So for now, I think the maturity of zeromq makes it a better choice than nanomsg, but this may change of course.

---

rich_d

December 3, 2014 at 11:26 am

@Thomasi were you able to fix the issues or get support? I am thinking about the possibility of using nano in a production setting. Is it still too beta for that in your view?

Mark

January 1, 2015 at 5:40 pm

Beautifully written post. Thank you.

Pingback: Fast, Scalable Networking in Go with Mangos | Brave New Geek

Pingback: 分散型メッセージングミドルウェアの詳細比較 | POSTD

Pingback: PHP ZeroMQ开发 | 勇气