Many Cubed; Solving the *Many-to-Many-to-Many* Problem for Scalable Reliable Secure Distributed Internet Applications

Samuel M. Smith Ph.D.

sam@prosapien.com

Version 1.1

2017/01/05

Overview

This white paper provides an approach to the scalable design of secure distributed internet applications. The key insight is that, in order to reliably scale, secure distributed internet applications must solve the *many-to-many-to-many*, "*many cubed*"," problem in how the various components of the application share information. In this discussion the term Internet means systems primarily built on top of the Internet Protocol (IP) stack.

The *Many-to-Many-to-Many* Problem

The *many-to-many-to-many* or "*many cubed*" problem arises from several critical constraints relevant to building scalable reliable secure distributed internet applications.

Security

The first of these critical constraints is security. The conventional approach to internet security is often described as defense-in-depth where each nested layer forms a secure perimeter that restricts access to higher levels of trust. These perimeters typically include some form of host:port based firewall. The outermost layer faces the wild wild west of the open internet where every interaction may be hostile. The next layer may be in a DMZ and/or behind a firewall. The following layer might be a trusted service running on a private network. VPNs provide a way for an external remote host to tunnel through the outer layers to access a trusted space. What a perimeter model assumes is that network locality can be trusted. Internet hosts and services running in an inner layer are allowed to share information with each other because thay are in a trusted "safe" space.

This conventional approach fails for many reasons. Fundamentally it fails because it does not account for the fact that hostile users exist everywhere not just on the open internet. Any mechanism that allows for lower priviledged users to access a network layer provides an attack vector for that user to elevate their priviledges thereby gaining access to everything in that layer. Hence, network locality is not security. Moreover, perimeter security is static and does not respond to dynamically evolving exploits.

The solution is to use a diffuse trust perimeterless security approach. Some call this a trustless or zero-trust security model but that is a misnomer. There is still trust, it is just diffused in such a way that security is greatly enhanced. The assumptions and principles of diffuse trust perimeterless security are:

1) The network is always hostile both internally and externally, locality is not trustworthy.

2) By default, every network interaction or data flow must be authenticated and authorized using best practices cryptography.

3) By default, inter-host communication must be end-to-end encrypted and data must be stored encrypted using best practices cryptography.

4) Policies for authentication and authorization must be dynamically modified based on behavior (reputation).

5) Policies must be governed by distributed consensus.

Distributed consensus diffuses the trust for any policy decision to a group of hosts. In order to defeat the policy, an attacker must exploit some majority of the hosts. This makes exploits exponentially more difficult. Using end-to-end encryption and storage prevents exploits from anyone that merely has access to the network or the data storage device. By authenticating and authorizing every network interaction or data flow, security becomes granular. A successful exploit of one interaction does not bleed into any other. Compromising one data flow does not compromise any other. Escalation opportunities are minimized. Many security exploits are discovered through repeated probes and experiments to find bugs, buffer overflows, or weaknesses in network protocols or software implementations. Dynamic policy modification that uses AI to first profile and detect anamalous behavior and then restrict the authorization of that user prevents discovery. This adds time as a defence.

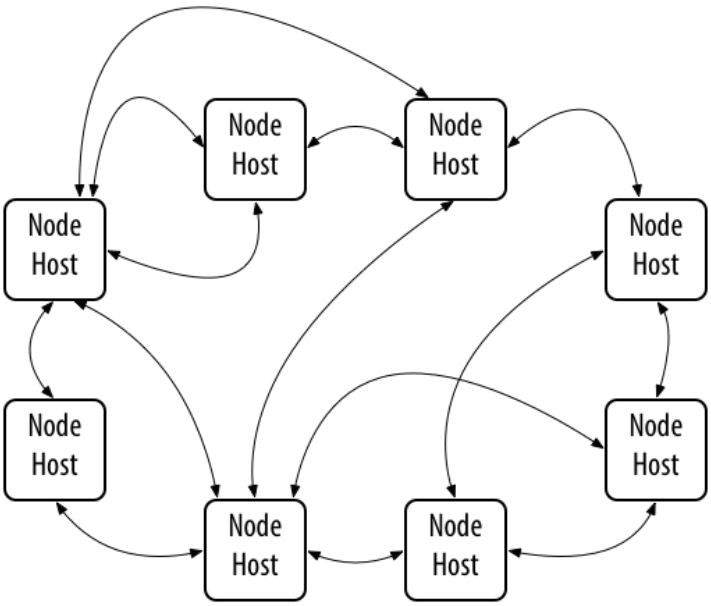
A diffuse trust perimeterless security architecture requires a many-to-many IP communications architecture for several reasons. Distributed consensus requires that the communications topology be many-to-many. A central server for pool communications is a single point of failure and a vector for exploit. Likewise, granular authentication and authorization policy requires a many-to-many communications topology to avoid single point failures and centralized loci of exploit. Indeed, for the same reasons any service interacting with a distributed consensus pool needs to be able to

access many pool members not just one. Transparent detection of anamolous behavior with dynamic changes to authorization also requires a many-to-many communications topology. This many-to-many is only the first two *manys* in the many-to-many-to-many. The last *many* will be discussed later.

Performance and Reliability

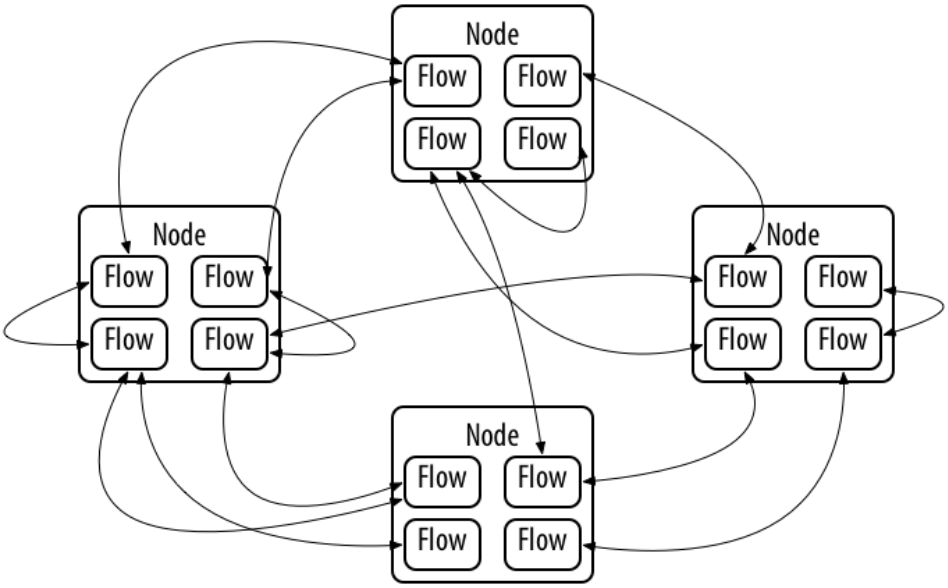
The second of these critical constraints is performance and reliability. Distributed internet applications suffer from performance constraints due to limited network bandwidth and latency. Applications with large numbers of users may exceed the computation and memory capacity of any given single computer. Consequently the application must be load balanced across multiple computers. Geographic distribution makes round trip latency a significant performance impediment. Hence the the load balancing must also be across multiple data centers and may even need to include edge computing devices.

Given that no single computing or network component is perfectly reliable, each component must be implemented with redundancy. Typically the redundancy is achieved using either primaries with hot failover secondaries or clusters of peers. The CAP (Consistency, Availability, Partition) theorem defines a trade-off between reliabilty and performance on a falable network. Clustered reliablity with M of N guarantees can achieve both load balancing, geographic distribution and robustness to failure. Clusters typically use a many-to-many communication topology. But more importantly for the distributed computing components to operate in a diffuse trust perimeterless security system, all the components operation must be governed by distributed consensus and have granular authentication and authorization which, as already discussed, require a many-to-many communication topology.



Component Flow Diversity

The last *many* of the *many-to-many-to-many* is the diversity in the type of components and how they communicate with each other, or in other words, diversity in the components and diversity in how these components share information. It is not enough to merely consider internet host:port addresses as the primary components of the distributed application. To do this would commit the network locality is security fallacy. It also imposes a rigid structure to the application that impedes redistribution. It induces premature or faulty optimization in the design.



The primary components of an distributed internet application can be called many things such as, concurrent data flows, or logically concurrent flows of execution, or network interaction flows or service flows. The term flow is used to express that the component is not just data or a resource but all the supporting functionality that is required to generate, deliver, maintain and store that data or resource over time and space. Each flow must be uniquely addressable. Each flow may have different requirements for delivery (sharing) of information in terms of scope, distribution, ownership, security (encryption, authentication, authorization, privacy), reliability, latency, and throughput. Best practices for end-to-end encryption cannot be bolted on to an existing higher level transport. Best practices require key management and exchange using asymmetric diffie-helman key exchange and muti-factor authentication. Encryption should use both long term and short term keys for future security. Messages should be authenticated and encrypted using sign after encrypt. Moreover, each flow may have different requirements for the generation, maintenance and storage of that information such as computation, memory, durability, and redundancy. The unique combinations of generation, delivery, maintenance, and storage requirements create a set of unique flows that constitute the last *many* in the many-to-many-to-many problem. Indeed, the internet protocol stack is usually augmented by an application protocol stack that sits on top of the IP stack to handle this final *many*.

Another way to describe a component flow is with the term "*network variable*", that is, a variable that has network scope, e.g. can be addressed by another component over the network. A network variable has a service delivery type that might be one of request/response, acknowledged/retry, repeated, or sequential. Other examples of component flows are pub/sub or other message queuing systems. URLs are an example of a scheme for merely addressing resources or variables with network scope.

One additional example, comes from automation systems, where the term endpoint is used to express what this paper is calling a component flow. An endpoint is a *source* or *sink* of information such as a sensor (source) or actuator (sink) or controller (both source and sink) with requirements for generation, delivery, maintenance, and storage imposed by the particular application. A single compute device may have one or many endpoints. Each endpoint is uniquely addressable by other endpoints in the network. Many real world automation systems have tens of thousands or even hundreds of thousands of endpoints at a single physical location. Thus combining multiple locations into one distributed application makes for a very large final *many* of the many-to-many-to-many.

Solving the Mapping Problem

Solving the many-to-many-to-many problem requires both finding a way to map the communications topology of many-to-many network hosts to the application protocol stack and also finding a way to map component data flow diversity to the application protocol stack. Solving the many-to-many-many or many cubed problem involves solving this mapping in a scalable, reliable, and secure way.

Scaling

Take for example a simple distributed application with N internet hosts. Each host runs on a computer with a NIC and a certain number of cores. The system must be reliably support the diffuse trust perimeterless security model. Given N internet hosts participating in a distributed consensus pool, then there are N^2 internet communication paths if each path is bidirectional and $2N^2$ if each path is unidirectional, but must also support bi-directional data flow. Suppose that D is the number of uniquely diverse types of component data flows per host (uniquely addressible with a unique combination of delivery, generation, maintenance, and storage type). The total number of data flow communications paths is therefore on the order of N^2D^2, where D may be an order of magnitude bigger than N. Take for example, a very small system where N = 10, and D = 10. This results in as many as 10^4 = 100,000 unique combinations of data flows/communication paths that must be supported by the application and as many as 10^2 = 100 that must be supported by any host.

One approach to this scaling problem is to make D close to one so there is only one type of data flow. This makes D^2 small. With D^2 small, the solution space is focused on optimizing network performance and does not need to be concerned with data flow performance. This will not work, in

general, for a perimeterless diffuse trust security model which requires more diversity in the data flows thereby making D much larger than one.

Another approach to this scaling problem is make N close to 1 so that N^2 is small. With N^2 small the solution space is focused on optimizing the compute/memory performance of each computer and does not need to be overly concerned with network performance. This also will not work in general for a perimeterless diffuse trust security because distributed consensus is needed to manage policy and larger N >> 1 is needed to provide enough security.

A third approach is to make one of the Ns a one, that is, use a many-to-one architecture. while this allows for bigger N and D, this will not work in general for a perimeterless security model because the one becomes a single point of failure/exploit.

This paper presents an approach that is designed to work for both large N^2 and large D^2.

Complexity

While many different approaches to solving the many cubed problem may be shown to work in theory, a critical practical constraint is the relative complexity of the approaches. Distributed internet applications are inherently complex. Those with large D are extremely complex. But not all complexity is created equal. In this discussion, *actual* complexity may be defined as the number of dependencies between elements in the system, whereas *apparent* complexity may be defined as the number of dependencies that developers must manage in order to install, configure, maintain, and enhance the system. The *apparent* to *actual* complexity ratio is a valuable design metric. A low ratio indicates good design.

Often in complex systems design the goal may be to reduce apparent complexity at the expense of performance in order that the system may be used and extended practically and reliably. Indeed the perceived risk developers face when attempting to add meaningful enhancements to a system could impose a performance barrier much harder to surpass than either network or computer resource constraints. Consequently, all else equal, low apparent complexity is an important design feature both aesthetically and practically.

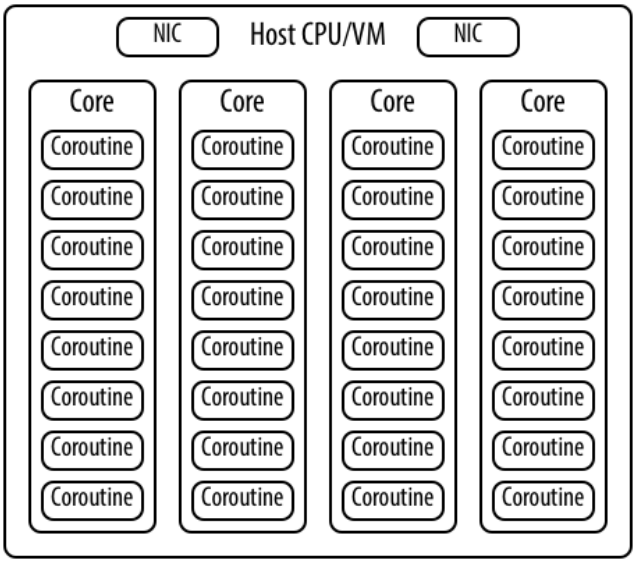
Computation

Distributed applications tend to have many concurrent logical flows. This arises in part from the need to communicate with other host and each communication path may require its own logical flow. Also distributed applications often need to manage events generated by users or other components of the application. Responding to these events typically requires concurrent logical flows. Additionally, keeping the application running continuously means additional logical flows to maintain the system. Finally, distributed applications often have time varying loads that require elastic scaling. Managing this elasticity requires additional concurrent logical flows.

The conventional approach to supporting multiple concurrent logical flows is to use a multiple concurrent threads or processes. Typically in a multi-threaded/process approach, an external I/O operation blocks the thread of execution waiting for the I/O operation to complete. Originally, support for multi-processing was provided so that independent applications could share the same computer without having to cooperate. Consequently, the scheduling of threads and processes is handled by the operating system which can preempt a given thread or process. Each process/thread is given its own context of local memory and registers that is swapped in and out each time the process/thread is activated/deactivated by the scheduler. The primary difference between threads and processes is that multiple threads can use a shared memory, whereas each process has dedicated memory. The advantage of this approach is that independent logically concurrent operations can each have a dedicated thread of execution that follows the logical flow of a task.

When the goal is to implement multiple concurrent but cooperating logical flows, implementing them with a multi-process/threaded model can be problematic. Cooperation means sharing of information, but preemptive scheduling means that locks, semaphores and other techniques must be used to ensure atomicity of operations especially when using shared memory. Many-to-many communications between blocking processes/threads who share resources can result in difficult to debug dead-locks and spin-locks. Priority management of threads of execution can be complex and difficult to tune. The overhead of the context switching is significant. On systems with large numbers of threads/processes the context switching overhead can dominate CPU utilization. There is a fundamental mismatch between the design goals of independent multi-process systems and cooperative multiple logical flows of execution.

The essential enablement for distributed applications is support for cooperative multiple concurrent logical flows. Asynchronous coroutines with non-blocking I/O provide a well suited match. Coroutines provide the concurrent logical flows where logical context is not lost. Coroutines are scheduled with a cooperative scheduler within a single process so there is no heavy process or thread context switching overhead. Instead the overhead to switching coroutines is equivalent to the overhead of a subroutine call. Each cooperating coroutine yields back to the scheduler at appropriate places in the thread of execution. Atomicity is ensured. Non-blocking I/O means that no preemption is needed, instead a coroutine "blocks" waiting for I/O to complete by polling an I/O buffer and if empty yields to the scheduler. Coroutines are sometimes called micro-threads, weightless threads, fibers, or greenlets.



Cooperating coroutines provide the most performant use of processor resources and the least complex for the developer while still enabling concurrent logical flows. They do require that each coroutine yield appropriately so that it does not starve other coroutines. But this is much easier to design, debug, and tune than doing that same with preemptive multi-tasking prioritization, locks, semaphores, etc.

The ideal structure for optimal use of processor resources is to have one process per CPU core. This minimizes context switch overhead. Each process has a coroutine scheduler that can run thousands or tens of thousands of concurrent logical flows without incurring context switch overhead. This architecture frees the developer to design each concurrent logical flow with minimal dependencies. Each of the ND data flows per host can be directly and scalabley mapped to one or more coroutines

To fully use all the cores in a CPU, the coroutines in each process need a way to communicate with the other coroutines in the other processes. Unix domain sockets provide a convenient reliable performant mechanism for this.

Another well known approach to asynchronous cooperative multi-tasking is to use a call-back (promise, future) approach. In this approach the scheduler is more heavy weight in that it handles the non-blocking I/O for the concurrent logical flows. NodeJS, Tornado, and asyncio among others use this approach. The primary disadvanntage of this approach vis-a-vis coroutines is that the concurrent logical flows are split into code fragments at I/O boundaries. Any time a logic flow must wait for I/O to complete, control is returned to the scheduler but control is not resumed in the same code fragment. Each flow registers a call-back (or promise) to be executed by the scheduler when the I/O completes. A flow with multiple I/O steps will be split multiple times. This approach is less performant and much more complex to code and debug than the coroutine approach. The motivation for using a callback approach is that it can be implemented in any language. Whereas coroutines require support by the language compiler/interpreter. Recently, however, many major programming languages have added support for coroutines. The simplicity and performance advantages of coroutines over call-backs and multi-processing make them the best computing architecture for cooperative distributed applications.

Network

Components of a distributed internet application cooperate with each other primarily using asynchronous discrete messages. These messages support the logic/control/event/data flows for coordinated activity. Although there are many distributed applications that include streaming data (video, audio, etc), these streams are by products of the coordination between components. Consequently, the primary inter-host communications requirement for composing a distributed internet application is support for asynchronous many-to-many messages.

The internet protocol stack provides two primary protocols that could be used for inter-host communication in a distributed internet application, these are: TCP and UDP.

The most well known is TCP. The primary attraction of TCP is that it provides a level of reliability whereas UDP does not. The goal of TCP was to hide the packet switched nature of the internet by emulating a directly wired serial connection. In order to do this TCP uses handshakes to ensure sequential delivery of packets between two dedicated endpoints. The primary disadvantage of TCP is that this emulation of dedicated serial channel between two hosts is not a good match for the underlying packet switched nature of the Internet. This may result in latency, throughput, and memory problems at scale. It is noteworthy that a dedicated serial channel is also not a good match for the inherent asynchronous messaging requirement

Because each connection is one-to-one between two hosts with dedicated buffer and synchronous packet flow, TCP does not scale well when used for many-to-many asynchronous topologies. Revisiting the example above, where the inter-host communications paths are on the order of N^2. Then each of the N hosts must have at least N TCP connections with each of the other hosts. TCP uses ephemeral ports where only one side of the connection (the client) can initiate the creation of the connection with the other side (the server). This imposes a asymmetric complexity when configuring asynchronous many-to-many communications. TCP uses an asymmetric client/server model. But what is needed is a symmetric peer-to-peer model where each host can initiate communications with any other host asynchronously. One way to avoid this complexity is to emulate

peer-to-peer by having each host to be both a client and server with any other host. This means the number of TCP connections on each host is 2N.

Because it is sequential, the throughput of TCP is a function of the buffer sizes which determine the degree of pipelining via congestion windows. Each connection requires memory for its buffers and processor overhead to manage the handshakes.This means for large N the computer may become TCP bound. Furthermore, the connections, typically, are not equally loaded. But TCP does not allow for sharing of resources between connections. The load balancing and tuning of large numbers of TCP connections can be complex.

Moreover, TCP is not fully reliable, i.e. connections do fail. When a TCP connection fails and has to be re-established any data in the TCP buffers is lost catastrophically thereby violating the sequential, synchronous nature of the data flow. Consequently the application stack must always add another layer of reliability on top of TCP to ensure that the data was actually transmitted end-to-end. Also TCP is stream based so the application stack must also add packet/message framing on top so that the recipient can detect message boundaries.

UDP is packet based (datagram) with built in support for packet framing. UDP also provides the source and destination hosts and ports in its packet headers which are passed on to the application stack. What UDP does not provide is reliability. UDP makes no attempt to ensure a packet was delivered. This means that the application stack must add a transaction layer for ensuring reliable delivery. Remember that TCP does not provide end-to-end reliability either. In both cases the application stack has to add a transaction layer to ensure full end-to-end reliability. The tradeoff is a little more complexity in providing reliabiilty with UDP in return for scalability, performance, and overall simplicity. UDP is inherently peer-to-peer. Each UDP host is a server that can accept packets from any other host that are addressed to the UDP port the server is listening on. Sent packets can be associated with the same source port so the recipient nominally knows where it came from (obviously for secure communications individual packets need to be authenticated but this true for both TCP and UDP). Thus each host needs only one socket no matter how large N is. Only one set of memory buffers is needed no matter the size of N. Only one coroutine is needed to service that one socket. Hence a UDP socket is already load balanced and tuning is trivial compared to TCP.

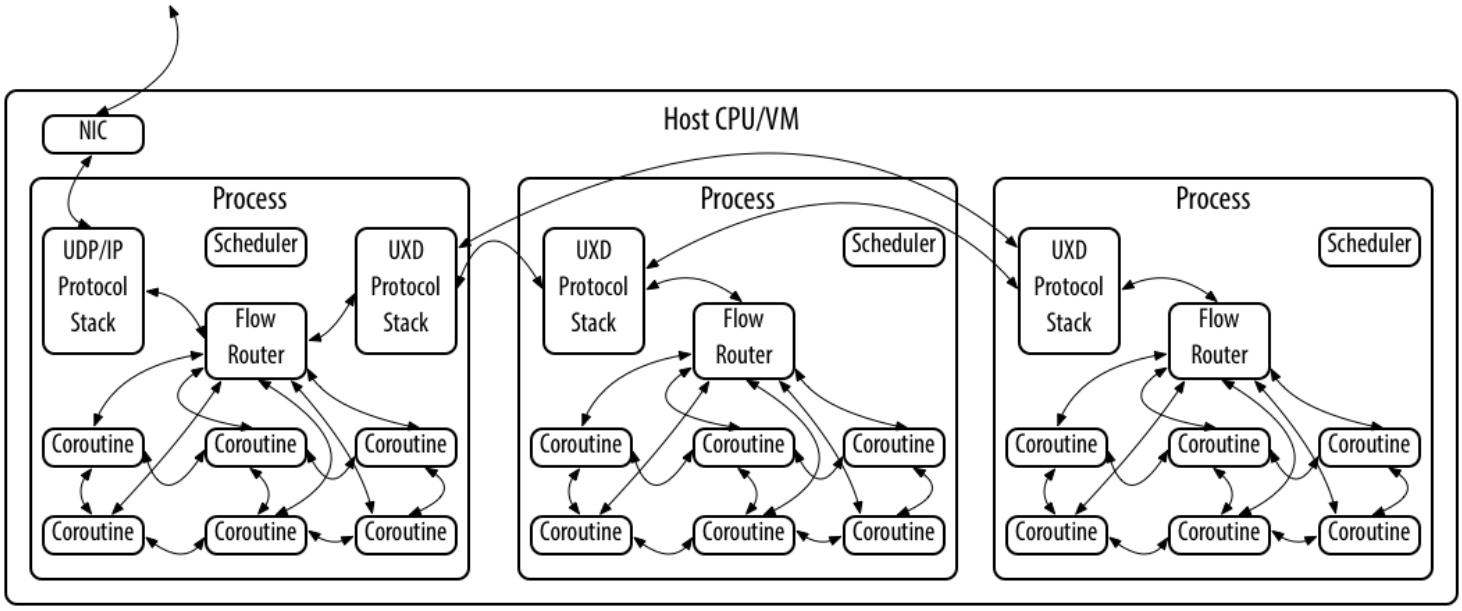
The primary limitation of UDP is reliability. Historically, one of the reasons that the IP stack did not provide a reliable datagram service is that there was no consensus on the type and degree of reliability that it should have and adding reliability to datagrams is simply a matter of adding a transaction layer so the each application stack could add the reliability it needed. Many protocols have been developed over the years to add reliable delivery to UDP.

One of the reliability problems with UDP is the MTU (Maximum Transmission Unit) size. Historically this was 576 bytes but with IPV6 is now 1280. Any UDP packet frame greater than the MTU could be split during transmission between switches and routers. While the IP protocol makes some attempt to reassemble split frames, the error rate grows significantly with multiple hops. Consequently support for packet/messages greater than the MTU size is on the more useful additions a UDP based protocol can add. This is usually the most complex transaction because it requires re-ordering frames and re-requesting dropped frames. But once a UDP protocol provides support for reliable delivery of large messages it becomes comparable to TCP for reliability without the scalability problems. For messages smaller than the MTU, adding reliability to UDP merely requires an end to end acknowledged transaction. But remember this is also required for TCP. Finally the other type of reliability that may be needed is sequential order of messages (not the sequential order of frames of a single message). While TCP in general provides this as a by product of its sequential, synchronous nature, the fact that a TCP connection can fail and will lose any messages in its buffers means that the application stack must still add on support for end-to-end ordering of messages. Thus UDP is not at a disadvantage in this case either.

TCP's major advantage is that it is well know and many "application stacks" have been built on top of it, such as HTTP, that add the missing layers of reliability that TCP does not provide. But given its scalability and performance problems, TCP will always be problematic for large scale secure distributed internet applications that use diffuse trust perimeterless security. The additional effort to make UDP work is a one time short term acute resource problem, whereas managing scale with TCP is a long term chronic resource problem.

Data Flow Routing

The final key to a viable solution to the many-to-many-to-many problem is to use data flow routing to connect the data flows hosted by networked computers where each computer may have multiple cores with multiple processes and each process hosts a coroutine scheduler with multiple coroutines per process. Each process as a single shared memory. Interprocess communications uses datagram unix domain sockets that provide packet framing and are reliable by default. A routed data flow is uniquely addressable by any coroutine within the distributed application. Each routed data flow can have different reliability and service types. These are provided by the associated coroutines.



The natural way to address a dataflow is to map it to the underlying processing architecture. The two major use cases for storing a data flow are as:

- A best available value
- A sequence of values.

In either case a value could be a nested data structure such as a key:value (dictionary) or an array not just a single number or string of characters. A sequence of values can be provided by a queue or deque.

The first level of the address is an identifier for the particular data flow value or queue or values. Call this the data share identifier. The data share identifier could be a name spaced string such as "peter.piper.pickled".

The second level of the address is an identifier for the particular interprocess communications stack (unix domain socket) that runs in the process that provides the data flow memory.

The third level of the address is an identifier for the UDP socket on the computer or internet host.

The fourth level of the address is an identifier for the channel or group of internet hosts that are mutually authenticated for secure communication with each other over UDP sockets. Maintenance of a UDP channel requires that the credentials for each host on the channel be stored on each host. This requires a very small amount of memory so even very large N does not place a meaningful demand on resources.

The complete address is a tuple with the following elements:

```
(datashare name, UXD name, UDP name, channel name)
```

By using named identifiers instead of the actual underlying UXD and UDP addresses, the routed address can be mapped to virtual UXD and UDP stacks to enable transparent debugging, staging, and load balancing of the data flows and associated coroutines.

Each message between has a routing dictionary that includes both the source and destination addresses as tuples. Each UDP stack has a router that processes incoming and outgoing messages. Each UXD stack has a router that also process incoming and outgoing messages. If a member of an address tuple is empty, this implies that the route is local at that level. This allows relative addressing. This approach is also forward compatible should additional levels beyond four be added to the routing.

```
{
    "route":
    {
        "src": ["peter.piper.picked", "pepper", "peck", "garden"],
        "dst": ["suzy.sand.sell", "shell", "shore", "sea"]
    }
    "tag": "commerce.hot",
    "stamp": "20170125T13:45:62.340123UTC",
    "data":
    {
       "quantity": 5,
```

```
        "price": 1.50
    }
}
```

Routed data flows are inherently scalable. Total network resources scale linearly with respect to N and D not exponentially as is the case with pairwise connections. The scale limitation is not based on interconnection topology or connection type but merely the total network bandwidth available. Each router hands off the associated data to the addressed data flow location where other coroutines handle the data. Higher level reliability or service types are handled by this other coroutines. This provides a modular, extensible framework for building concurrent logic flows.

For diffuse trust perimeterless security, multiple channels may be required where each channel is authenticated at a different level of trust. For example the control plane channel that manages authorization of lower level channels, might need to be formed at a higher level of trust where the associated credentials use a higher degree of authentication. If M is the number of channels then the total network resources required will scale linearly with M.

Routed data flows allows support of any of the common use case connection patterns merely by adding logic to the associated coroutines. Routed data flows also enable using messages for error signaling throughout the infrastructure. This satisfies the *Reactive Manifesto* design principles.

Scaling Revisted

As described previously solving the *many-to-many-to-many* mapping problem is finding a way to map N^2D^2 data flow paths on the the host computers and applications stacks.

There are many IP based communication stack solutions we could analyze for the sake of comparison. An exhaustive list is beyond the scope of this paper.

DDS/RTPS

The DoD's Global Information Grid has a messaging layer that uses the DDS/RTPS Corba Standard. RTPS stands for Real Time Publish and Subscribe. Unlike other MQ (messaging queueing) stacks. RTPS supports peer-to-peer messaging between UDP hosts. RTPS also supports client/server broker based messaging as well as TCP. But the "real time" in RTPS is primarily acheived using peer-to-peer UDP. RTPS adds reliable delivery on top of UDP. The DDS (Distributed Data Service) is a binary protocol for the different data flows transported over RTPS. Because RTPS uses UDP it benefits from the scalability and performance of UDP. The limitation of RTPS is that it does not have built-in end-to-end encryption but must use bolt on security. Also DDS is a binary data protocol which is not flexible and makes it difficut to support arbitrary nested data structures. Although DDS/RTPS is an open standard the primary implementations are proprietary. These implementation are also thread/process not coroutine based. DDS/RTPS is also a very complex protocol which has hampered widespread open implementations.

ZeroMQ

ZeroMQ is based primarily on TCP but supports other more esoteric IP transports to support connection maintenance. It maps a dedicated tcp connection to a given flow type, i.e. ZeroMQ has multiple socket flavors. Thus ZeroMQ combines N and D. Consequently with ZeroMQ, more diversity in data flow types (D) between hosts increases the number of TCP connections (N) between those hosts and also consumes more IP buffer space and and expends more processing overhead. In this respect ZeroMQ places a relatively heavier demand on the IP stack in *many-to-many* topologies. For this reason ZeroMQ is best applicable to *one-to-many* or client/server topologies. Indeed, the primary use case examples for ZeroMQ are *one-to-many*. ZeroMQ does the mapping between IP sockets and zeromq connections in one or more background threads. This reduces context switching overhead vis-a-vis conventional one thread per tcp blocking I/O architectures. But logical concurrency still requires that the application use one thread per data flow. So large D is still a problem.

ZeroMQ provides message framing and various degrees of reliability with different connection service types. ZeroMQ also provides faux asynchronicity through its API by providing timeouts with queues when performing I/O access to each connection. These allow the developer some of the simplicity of coroutines but does not provide the context switching savings of true coroutines. Because the ZeroMQ threads are still preemptively scheduled by the OS, tuning of the queue sizes can be problematic under high load. The different inter-host connection types ZeroMQ provides include request-response, pub-sub, req-router, dealer-router, dealer-dealer, router-router, and push-pull. There is some complexity in design to get the correct combination of connection types tuned to an application. ZeroMQ can, however, make developing the communications component of conventional multi-threaded/process applications easier. The primary motivation for ZeroMQ was to replace centralized brokered AMQP systems such as RabbitMQ, ActiveMQ, etc. In that regard a well designed ZeroMQ system can provide better performance than a brokered system. This may be primarily a limitation of the centralized brokering approach. Finally security for ZeroMQ is bolt on not integrated. This alone makes ZeroMQ an inappropriate if not problematic choice for a diffuse trust perimeterless security model.

There is, however, a new version of ZMQ called CurveZMQ that uses CurveCP for end-to-end security, i.e. is not bolt on. Unfortunately, CurveZMQ imposes some topology limitations inherited from CurveCP. Quoting the CurveZMQ docs;

> “ ‘*CurveCP enforces a 1-to-many relationship from servers to clients*’.

This is an artifact of how CurveZMQ implements CurveCP. Notably CurveZMQ does not have a mechanism for the exchange of long term credentials, much less the peer-to-peer formation of channels. This must be bolted on.

Finally, CurveZMQ as proposed by its creators is at best an interim solution to an eventual UDP based protocol. Again quoting the CurveZMQ docs;

> “ ‘The world wants a secure, reliable UDP protocol, but it will take years to see this implemented at the kernel level, where it belongs. ... As an interim solution, we have taken CurveCP's security handshake alone, and implemented that design on top of TCP as CurveZMQ.’

In contrast to the quote above, given the performance and scalability advantages of UDP and the ease with which reliability can be added, a reliable secure UDP based protocol does not have be implemented at the kernel level to be viable. A better interim solution is not to fix up TCP with security per CurveZMQ, but to fix up UDP.

RAET

This paper proposes the RAET (Reliable Asynchronous Event Transport) protocol as the prime candidate for the *eventual secure reliable UDP protocol wanted by the world*.

Notable Features

RAET supports all the design and architecture features outlined above for solving the *many-to-many-to-many* problem for a secure perimeterless security model. Without going into great detail about RAET, some notable features of RAET include;

- A peer-to-peer implementation of the CurveCP handshake that allows for either side of a pair of hosts to initiate the handshake.
- RAETs handshake resolves simultaneous CurveCP handshakes by both sides.
- RAET provides a mechanism for peer-to-peer exchange of long term keys with multiple levels of control over how the exchange occurs.
- The RAET CurveCP handshake, in addition to the signing and encryption keys, include name, role, and mode identifiers. Thus a completed handshake authenticates the name for data flow routing, the role for authorization, and the mode for how the node operates on a given channel. This bootstraps the major elements of a secure channel where each interaction is authenticated and authorized. Using routed data flows on top of the RAET channel credentials enables a data flow router to easily implement authorization policy. Authorization is per data flow address. Dynamic authorization policy can be easily implemented via a lookup table associated with each router.
- RAET is an extensible protocol that makes a trade-off on the side of simplicity and ease of debugging with a flexible header and a flexible data body. The header by default is ascii readable and achieves compactness using a default value policy. Each header field has a default value. Whenever a particular header uses the default for a given field then that field is not included. This makes the ascii header on average about the same size as a full binary header. The header can also be JSON or binary msgpack. Because the header is not fixed it is easily extensible and future proof. The RAET message body can be raw binary, or JSON, or msgpack. JSON and msgpack provide flexible serialization of nested key:value stores. Msgpack is not ASCII readable but is more compact.
- The standard RAET message transaction supports large message sizes (bigger than an MTU).
- Each individual message can be sent with different reliability parameters (timeout, retries, burst size).
- RAET also provides presence detection where each host can send periodic heartbeats to the other hosts on a channel.
- RAET also provides a simple interprocess communications protocol using Unix Domain Sockets (UXD) on unix and MailSlots on Windows.
- RAET is compatible with any of the asynchronous frameworks that run on Python including both coroutine and callback based.
- RAET is also designed to take advantage of the Ioflo, asynchronous flow based programming framework that naturally maps routed data flow messages to Ioflo data shares and hierarchically contextual coroutines.

Extending RAET

The main limitation of RAET is that it is currently only implemented in Python. RAET's interprocess communication protocol provides a simple and performant way of extending RAET support to other programming languages.

The proposed extension architecture is similar in some ways to Erlang. Erlang has an EPMD *Erlang port mapper daemon* (*router*) process that runs on each internet host. The port mapper is responsible for routing Erlang messages to Erlang schedulers. Erlang uses TCP. Typically Erlang fires up scheduler per core thus providing load balancing. Each scheduler runs in its own OS process. Each scheduler runs multiple Erlang 'processes'. An Erlang process is not an OS process but is analogous to a coroutine. Erlang *coroutine* schedulers are preemptive. Despite being preemptive, atomicity is ensured because the preemptions are soft, and only occur on reduction boundaries. In addition, Erlang shares nothing between its

processes. All interactions use queued messages.

Extending RAET to other languages is achieved by building the equivalent of an Erlang port mapper, called a RAET Router. A RAET Router includes a RAET UDP stack, a RAET UXD stack and a data flow router. Other languages need merely to implement a RAET UXD protocol stack to communicate with the RAET Router.

The RAET UXD protocol adds a header with the source and destination routing information as well as sectioning information for splitting large UXD packets into smaller frames. The header has a simple fixed ascii representation. Because UXD datagrams are framed, reliable, and sequentially ordered there is no need for retries or other complexities. The sectioning is merely to level the load and provide back pressure for large UXD packets. The body of a RAET UXD packet can be raw binary, or JSON, or msgpack just like the RAET UDP. This is a trivial protocol to implement.

Each language is then given an SDK that supports the RAET UXD socket API and a data flow router. The coroutines for that language run in a dedicated process. The RAET Python UDP Router runs in its own process. Installation consists of installing the Python RAET router and connecting the language specific UXD process to the RAET router's UXD interface. Multiple processes each supporting a different language SDK could share the same Python RAET router. Moreover, there are now good transpilers that can convert Python to Javascript and Go. Python can be run inside Ruby. These could be used to more quickly produce RAET SDKs in those languages.