The IsoGrid: Scalable Mesh for a Better World

By Travis.Martin@isogrid.org

Version 0.215

1 Abstract

The TCP/IP Internet has:

- High and Unbounded Latency
- Wasteful, Underused Links
- Limited Node/Switch/Hop Counts (no IoT support)
- Low Redundancy
- A Tendency to Centralize Power
- Choke-point Surveillance and Censorship
- Disaster Vulnerabilities
- Tragedy of the Commons

What follows is a free and open proposal for a solution: A new globally-scalable network protocol with a mesh topology. Instead of being limited to traditional address-routed packets, the protocol uses source routing to set up bounded-latency isochronous streams avoiding the problem of congestive collapse. Once a stream is set up, the route is given a numeric name to support routing micro-packets (*μPkt*) both directions along the route. In order to support isochronous streams across the entire network, the framerate of every links is a power of 2 frequency relative to TCG time, this opens the door to many new scenarios that require precise relative timekeeping. Micro-payments in arbitrary settlement instruments, which are made 'by simple agreement' between each neighbor along a route, are used to pay for sending data across the network, avoiding the Tragedy of the Commons. Client endpoints are responsible for building up multi-path redundant link maps through the network, relying on the advertised 3D-Geohash locations of the nodes to track only a subset of nodes within a given area; providing scalability, redundancy, and wider distribution of power. Contrasted with TCP/IP, the new protocol stack's layering model provides additional options for streams, packets, safety, reliability, robustness, latency, and extensibility. Most importantly, the entire protocol was morally designed with its socioeconomic side-effects as a guide.

This document is an early draft. If you'd like to help improve it, check out the IsoGrid Forum.

PART I: Introduction

2 What's wrong with Internet Protocol (IP)?

The goal of IP was simple: Create an interop-protocol to connect the world's networks. In this regard, IP has seen fantastic success. However, it wasn't designed with socioeconomic goals in mind. In this day and age, when much of global commerce seems to rely on IP, it's tempting to think that "Anything can run on IP, why not run the IsoGrid on top of IP?" But this would be akin to asking "Why not build the grid of roads only on top of the hub-and-spoke railway network?" When you think about the goals of the IsoGrid, you'll see that this is a ridiculous proposal. It's reasonable to try to rely on the existing IP infrastructure in the early phases of building its successor: Like the railways connected the cities prior to interstate highway systems.

High and Unbounded Latency

IP switches have to decode the entire header of a packet and then lookup routing tables before the packet can be routed to the next switch. But it's worse than that, with IP, there are no guarantees that a given packet will be routed within a certain amount of time, or even serviced at all. If 10 packets arrive on 10 different links at the same time, and all 10 have the same destination link, then some of the packets needs to wait for their turn. If the switch in this case only has buffers for 8 packets, then the last 2 are simply dropped.
The IsoGrid must provide bounded latency, and must provide low latency even as the network scales up.

Wasteful Underused Links

The majority of the links that comprise the Internet run at less than 50% utilization. This is related to the same latency/buffering problem above: In order to have even reasonable latency and low levels of packet-loss, links must typically be at less than 50% utilization.
The IsoGrid must allow for 100% link utilization without any increase in latency on existing connections and without suffering congestive collapse.

Limited Node/Switch/Hop Counts

IP has limits on the number of switches, nodes, and hops that make it ill-suited to an "Internet-of-*Everything*".
The IsoGrid must have no limits on the number of participating nodes or switches, and routes must be able to have an arbitrarily large number of hops.

Low Redundancy

Typically, most consumers and businesses have only a single link to the IP Internet. This is often because there is only one high-speed provider at a given location. But also, Internet links are mostly paid for by link bandwidth, rather than the actual bandwidth used. It becomes cost-

prohibitive to pay for multiple under-utilized links. Finally, IP itself doesn't have good support for multi-path.
The IsoGrid must promote a mesh topology, where it actually makes sense to have more than just one link.

Centralization of Power, and thus Wealth

With the IP Internet, Economies-of-Scale make massive centralized services cheaper than distributed services (even if similarly massive). These centralized services seem to leave little room for a healthy middle class.
With the IsoGrid, distributed services should be cheaper to provide than centralized services. Distributed services can spread the benefits of a growing economy more widely.

Choke-point Surveillance and Censorship

Because the Internet and the services running on it are so centralized, powerful governmental systems have the clear capability to surveil and/or censor it.
The IsoGrid must scale up to have no Choke-points *or* Check-Points: You should be able talk with your neighbors without permission from a central authority.

Vulnerable to Disaster

Major damage to a critical building in most major cities is likely to bring down IP Internet service in the region for weeks or months. War, terrorism, earthquakes, or coronal mass ejections could all completely bring down the Internet, knocking out both local and long distance communications, hindering recovery efforts.
The IsoGrid must not rely on central hubs.

Tragedy of the Commons

The Internet is a Commons, where everyone is expected to behave themselves or face removal from the network by network admins. This is expensive to police. The biggest example of this how it costs practically zero for spammers to send comment and email spam.
The IsoGrid must not suffer from Tragedy of the Commons, it should rely on micro-payments in exchange for accepting requests.

3 Socioeconomic Effects of TCP/IP vs. IsoGrid

It should be immediately clear to the reader that communication protocols, like TCP/IP, have dramatically changed the world. What is less clear, however, is that the specifics of protocol design can have wide-ranging social and economic effects, some positive, some negative.

Rail Networks are to Road Networks, as the TCP/IP Internet is to the IsoGrid

3.1 Economies of Scale

Many networks have a design that reinforces economies of scale. One can see this clearly with the train railroad network: The bigger, more interconnected systems beat out the smaller, less interconnected systems. This also creates huge barriers for new entrants, making it practically impossible for them to compete against the established providers. We submit that economies of scale with the present Internet are creating fewer and fewer providers, and concentrating control in a few individuals in the same way that the rail system of the late 1800s did. This doesn't have to be the case though. The grid topology of our roadways doesn't seem to have these same effects. The barrier-to-entry for personal use of the roadways is much smaller compared to that of the railway providers. How small can we make the barriers to entry in the telecommunications market?

The Railroads lead to Railroad Tycoons
The Internet lead to Internet Tycoons
Where are the Roadway Tycoons?
In the same way, the IsoGrid is designed to be less effective than the Internet at centralizing wealth and power.

3.2 Isochronous Streams

The "Iso" in IsoGrid is short for Isochronous, meaning 'same time'. Isochronous means that bits are sent (and then arrive) at a specific well-defined frequency. A theoretical isochronous stream running at a frequency of 1 MHz would transmit one word of data every millionth of a second. Implementations of Isochronous protocols can operate with statically sized buffers, and bounded latency guarantees. VoIP, Internet video, and Internet radio are best sent over an Isochronous connection. Over 50% of peak Internet traffic is actually perfectly suited to Isochronous streams. The early POTS telephone network operated with an analog audio stream, and so early digital communications protocols that evolved from this network could be considered Isochronous. However, over time, the world has instead settled on a packetized asynchronous solution, which is now called the Internet.
Being based on packets, the Internet has terrible support for isochronous connections. This is why your YouTube movies always need to 'buffer' before playing: It's sending a few seconds (or more) of the video to the recipient before it starts playing so that it can compensate for the randomly-timed delivery of packets. If the net had support for true isochronous connections, then it would be possible to watch YouTube and Netflix videos without having to wait for the 'buffering' to complete.

3.3 Topology

At its core, IP allows a maximum of 255 hops for any packet. This inherently restricts the topology of the Internet: As it stands, it can never be a world-wide mesh. Instead, you end up with large hubs and choke-points.

So far, all attempts to create interop standards for nodes that can hop between networks have been unsuccessful. The IP addressing scheme also makes it very difficult to have a mesh: IP addresses are assigned hierarchically.
The name "Inter-Net" describes the problem directly: The Internet isn't a global network that just anyone can contribute or connect to; instead, the Internet is just a protocol for *inter*-connecting the world's centrally owned and operated *net*works.

4 IsoGrid Requirements

The following are the goals and requirements for a new network protocol with a mesh topology called the IsoGrid.

4.1 Socioeconomic Vision:

- Lower barriers to entry in markets for goods and services that rely on networks
- Empower individuals to improve their lives
- Increase individual freedom

4.2 Primary Tech Requirements

- Very-low maximum-latency bounds
  - No overcommit, no oversubscribe
  - Always QOS
- Efficiently scales to arbitrarily high bandwidth links
- Efficiently scales to arbitrarily high node/switch count
- Mesh Topology
  - Multi-path redundancy
  - Disaster Resistant
- Seamless connectivity for mobile and "Internet of Things" nodes
- Avoid global protocol mandates that limit economic freedom

4.3 Secondary Tech Requirements:

- Differential GPS everywhere, even indoors?
- Great multi-cast
  - Any switch is allowed to be a multi-caster
- Support for asymmetric links
- Enable high-quality crowd-sourced deep-space antenna arrays
- Enable efficient use of long-haul space-based wireless laser meshes
- IP Tunnels over The IsoGrid should be better than existing non-LAN networks with respect to:
  - Reliability

    - Latency
    - Cost
    - Speed
    - Security

4.4 Non-Goals:

- Does not need to be limited to wireless networks
- Does not need to be simple, or easy
- Does not need to fit into existing hardware
- Does not need to work easily with existing infrastructure
- Does not need to preserve or promote existing power structures
- Does not need to conform to any 'model' of networking

5 IsoGrid Design Proposal

This section outlines a proposed design for a new network protocol with a mesh topology called the IsoGrid.

5.1 Layers

Layer 0: Physical Layer

- Options include, but are not limited to:
    - Ethernet, ATM, USB, etc.
    - Tunnels through other networks (like the TCP/IP Internet, etc.)

Layer 1: Link Layer

- Defines how nodes directly communicate with each other across links
- Defines how a *μPkt* can be sent between two nodes
- Provides for mesh-wide frequency synchronization
- The IsoGrid does NOT mandate a globally-required protocol at this layer
- The IsoGrid does impose generic requirements at this layer

Layer 2: Network (*μPkt*) Layer

- Defines the extensibility model for *μPkt* types and *μRoute* types
- Defines how the network routes *μPkts* across the IsoGrid
- Defines how nodes send credits in exchange for forwarding *μPkts*

Layer 3: Transport (*IsoStream*) Layer

- Defines how the network uses source routing for Isochronous Streams (*IsoStream*)
- An *IsoStream* is switched at the word level, one word at a time
- Defines how nodes send credits in exchange for switching *IsoStreams*

Layer 4: Session (*EccFlow*) Layer

- Defines how remote nodes use *IsoStreams* to safely and reliably communicate with each other over an arbitrarily long time period
- Forward Error Correction coding, safety, and multi-path segmentation
- Defines how nodes use the network transport to distribute routing information

Higher Layers: All the normal protocols you would expect to run on networks

5.1.1 IsoGrid vs. TCP/IP Comparison

Layer	TCP/IP	IsoGrid
Application	SSH, FTP, HTTP, etc.	TODO, let's not get ahead of ourselves ☺
Session/ Transport	TCP, UDP, etc.	*EccFlow* *IsoStream*
Network	IP	*μPkt*
Link	Point-To-Point, Ethernet, subnet Broadcast, token ring, ATM, etc.	Point-To-Point only, Isochronous USB, ATM, Point-to-point Ethernet.
Physical	Any	Point-To-Point only: Communication mediums that have collisions aren't well suited to IsoGrid

5.2 How it works

The IsoGrid is a mesh network that supports routing of one-way isochronous streams (*IsoStreams*) and small packets (*μPkts)*. In order to provide isochronous streams, the IsoGrid runs with a synchronized frequency, very similar to the way an electrical grid runs on Utility Frequency (except much higher frequency). The source provides a series of route instructions to be used at each hop (Source Routing). Each switch along the way uses its route instruction to establish the *IsoStream* connection. The *μPkt* that starts a connection defines the length of the *IsoStream*, and how many credits to send per word. The IsoGrid network and transport layers are optimized to support the IsoGrid session layer (*EccFlow*) which fragments the data, applies a forward error correction code, and sends the data over many paths across the network.

5.3 Credits and Micro-Payments

In order to avoid a tragedy of the commons, the IsoGrid allows for exchanging credits between neighbor nodes so as to pay for the data sent or processed. This allows payments to be made among neighbors instead of having to have a centralized payment processor.

In the IsoGrid model, each node owns its outbound links (but not really its inbound links), its CPU hardware, its storage, etc. Each node SHOULD charge credits for use of those resources. Each pair of neighbor nodes SHOULD agree on the settlement instrument that will represent the value of a credit between themselves, and payment SHOULD be by simple agreement (there is no required third party).

Any settlement instrument is possible, but here are some examples:

- electricity
- bitcoins
- cash
- check
- ACH transfer
- propane
- gasoline
- gold
- water

Nodes MAY have different costs for different outbound links.

5.4 IsoGrid Secondary Limitations

No network is without limits. The design of a protocol standard necessitates making tradeoffs in order to meet the requirements. Here are some of the known limitations due to the design of the IsoGrid Protocol:

- Links that use a shared physical medium (ex: those with collisions) aren't well suited to be used by the network (too much latency).
    - The IsoGrid is best suited for running on top of physical layers that have exclusive access to the underlying communication medium. Like fiber, copper, and point-to-point wireless such as 60 GHz
- A single *IsoStream* can use no more than the fastest available slot along a route
    - However, an endpoint can create multiple connections such that nearly 100% utilization with *EccFlow* is possible
- Low-rate connections have Higher Latency
- Half-Duplex links not supported (except with unacceptably high latency)
- Streams through nodes that move will have non-trivial buffering/timing requirements
    - The faster it moves, the more demanding the requirements
- New Links (for example, link tunnels) can take a three-way handshake (and a credit cost) for remote nodes to be able to effectively use the new links
- TODO: Add more as new limitations are identified

PART II: IsoGrid Protocol Specification

What follows is a free and open specification for a new open network protocol with a mesh topology.

6 Licensing & Legal

This protocol specification (*The Protocol*) is freely available for all to use as is.

The Protocol, legally speaking, is completely Public Domain: CC0

However, just because you have a legal right to do something, does not mean you *should* do something. The right thing to do, morally speaking, is not codified in law.

The Protocol was released to further the following socioeconomic goals:

- Lower barriers to entry in markets for goods and services that rely on networks
- Empower individuals to improve their lives
- Increase individual freedom

In particular, I believe *The Protocol*, when widely implemented, will further the above goals.

If, in the 10 years following the release of this version of the specification, you want to implement a change to *The Protocol*: You MUST make a good faith effort to ensure that your changes to *The Protocol* do not undermine the above goals.
The simplest way to do this is to openly declare your intended changes at the IsoGrid Forum, and see if the community agrees.

Implementing a change to *The Protocol* that undermines the above goals MUST be considered a form of corruption; akin to taking more than your fair share from a commons. It MAY be legal, but you MUST expect negative social consequences if/when this comes to light.

I hereby release these moral conditions for all uses of this version that follow 10 years after this specification version is first released into the public domain.

7 Definitions

Term	Description
Node	A device that interacts with other devices on The IsoGrid using the IsoGrid Protocol Stack

Link	The protocol running on a physical wire or wireless line that connects two adjacent nodes
Switch	A node with multiple links that is able to route data using the IsoStream Protocol
Word	The atomic unit of transmission across the network. 128 bits of data, 1 bit of parity
Slot	A 1 word wide logical division of the link's bandwidth, delivered isochronously 1 word at a time. Each available slot on a link can be allocated to switch a single connection stream at any given moment.
Input Slot	A slot on an input link. Has a matching output slot on the node on the other end of the link.
Output Slot	A slot on an output link. Has a matching input slot on the node on the other end of the link.
IsoStream	A one way, End-to-end stream of words that flows isochronously from a source to a destination across a pre-defined subset of the nodes that comprise The IsoGrid
µPkt	Micro-packet, a small (maximum 256-octet) block of data that flows asynchronously from a source to a destination across the IsoGrid. May also have dynamic data portions that can be modified by the switches along the route.
Frame	A Link-Layer logical aggregation of individual isochronous slots to be sent together across a single link. Note: As a link layer construct, Frames are NOT to be thought of as network packets; they do NOT route across the IsoGrid network.
Advertise	To make something publicly known to any network participant that asks

8 Isochronous Word Format

The atomic unit of transmission across the IsoGrid network layer is called a *word*. A word is 128 bits of data, with an additional 1 bit for parity. The parity of a word is defined to be EVEN if all 129 bits of the word have an even number of 1 bits. The parity of a word is defined to be ODD if all 129 bits of the word have an odd number of 1 bits.

Valid words meant for *IsoStream* payload MUST have odd parity, and are called STRM_ODD words, or abbreviated as SOWORD.

Valid words meant for *μPkt* communication MUST have even parity, and are called MSG_EVEN words, or abbreviated as MEWORD.

The following table shows common words types and their expected parity

Type of word	Value	Parity	Description
NoData	Link Defined. Suggested: 0x0000	MSG_EVEN	This isn't necessary, but a Link Layer protocol MAY find it useful to have a designated word that indicates no data is available on a slot. Or as a delimiter between *μPkts*.
μPkt	Node Defined	MSG_EVEN	*μPkt* data sent from one node to the neighbor node using a Link Layer protocol. Not part of an *IsoStream*, but may mark the start of an *IsoStream*.
InitIsoStream μPkt	Semantics defined by IsoGrid Protocol Sent by Source Node	MSG_EVEN	These words initialize the payment and initial word count of an *IsoStream*. The required members and their semantics are specified in this document, and the values are produced by the source node.
IsoStreamRoute	Switch Defined Sent by Source Node	STRM_ODD	These words are defined and advertised by each switch to describe each step of a route that an *IsoStream* is going to take through the network.
IsoStreamHeader	Source Defined	STRM_ODD	These words are sent by the source as a series of headers to the payload of an *IsoStream*.
IsoStreamPayload	Source Defined	STRM_ODD	These words are sent by the source as the payload of an *IsoStream*
IsoStreamFooter	Switch Defined Sent by switch	STRM_ODD	These words are sent by each switch along the route as the final parts of the *IsoStream*. They MUST match the *IsoStreamRoute* words that were originally sent to this switch when the *IsoStream* was initialized.
Lost/Corrupted *IsoStream**	*LostWord* (0x0000)	MSG_EVEN	If an *IsoStreamRoute*, *IsoStreamHeader*, or *IsoStreamPayload* word is lost or

			corrupted along the route through the network, that word MUST be replaced with *LostWord* to signal this fact to the destination.

9 Link Layer Protocols

There are many possible ways to define a protocol for the Link Layer. The IsoGrid Protocol Stack does not mandate any specific protocol or implementation at the Link Layer. As such, it is NOT necessary that everyone in the world agree to any standard protocol(s). Deciding on a Link Layer protocol is entirely a local decision between two neighbor nodes.
That said, the IsoGrid Network Layer (*IsoStream*) does impose some non-trivial requirements on the Link Layer below it:

- The Link Layer MUST meet the Slot Isochronous Standard below
- The Link Layer MUST meet the Slot Frequency Standard below
- The Link Layer MUST meet the IsoGrid *μPkt* Standard below

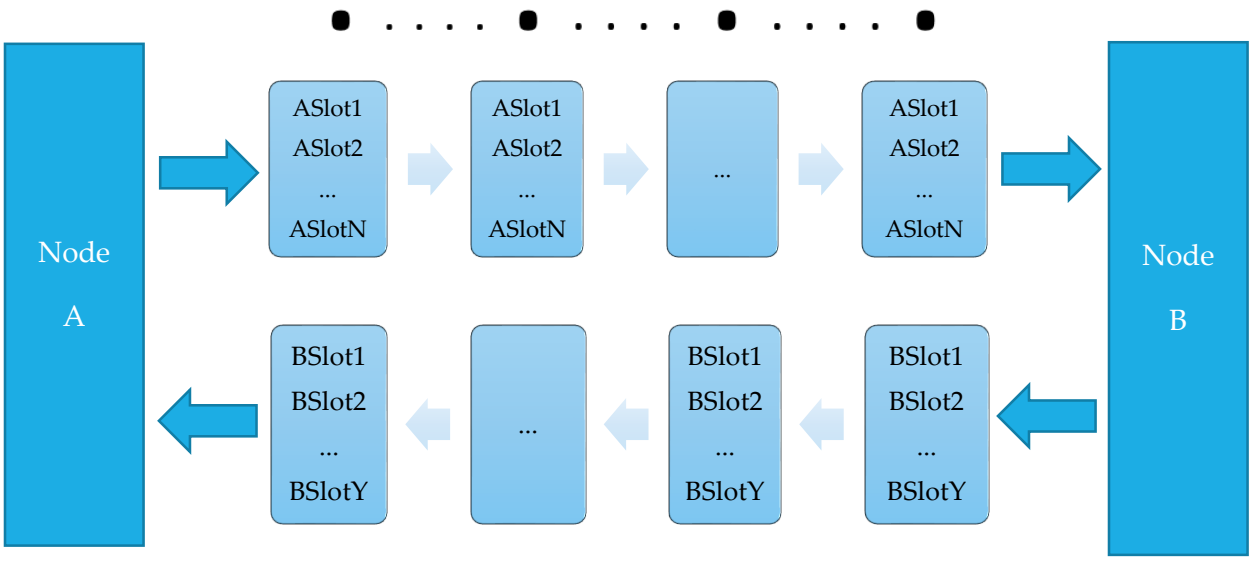


Figure 1. Link Layer frames exchanged between two neighbor nodes

Figure 1 shows a generic full duplex link exchanging isochronous frames between two neighbor nodes. Notice how the words on a slot arrive at well-defined periodic intervals and the input slots are unrelated to the output slots. Also, notice how the number of slots going one way does not have to match the number of slots going the other way (though often they will).

9.1 Slot Isochronous Standard

The IsoGrid Protocol Stack requires the existence of isochronous slots on all links across the IsoGrid. An isochronous slot is a 1 word long logical division of the link's bandwidth, delivered isochronously 1 word at a time.

This means that the words belonging to a slot MUST be sent across the link and arrive at well-defined 2^n word per second frequencies (where n MUST be a non-negative integer). A Link Layer protocol MUST define some sort of isochronous frame format logically divided up into slots:

- A frame MAY contain any number of words
- A frame MAY have all of the words broken into 2, 4, or 2^n slots such that each slot appears every other frame, or every 4th frame (and so on)
- Concurrently, A frame MAY have all of the words aggregated into a smaller number of slots, such that 2^n words arrive for a slot every frame.
- A node MUST be able to identify the slots of a valid frame that arrives on one of its links
- A node SHOULD be able to identify the slots of a valid frame even immediately following missed or corrupt frames
- A node MUST be able to assign an exact total ordering and count to every valid frame it receives
- Electronics MUST be fast and stable enough such that skipping a frame, or somehow seeing two when there was only one, MUST be extremely improbable
- A node MUST advertise the best-case and worst-case word-corruption rates of its output links
- A node MAY use error-correction codes to ensure the error rate meets the advertised value
- A frame MAY contain hashes to determine validity
- A node MUST be able to detect corruption of frames or words that occurs on its input links
    - Bit error(s) detected in a slot assigned to an *IsoStream* MUST be re-transmitted as *LostWord*.
    - If bit error(s) are detected in a slot being used for a *μPkt*, that *μPkt* MUST be dropped and ignored
    - A slot MUST have fewer than 1 undetected corruption of a word in every 1024^6 words
- Words from entirely failed links MUST be assumed to be *LostWord*.

Some example frame format protocols are:

- Statically sized frame:
    - Y ordered words, each their own slot
- Negotiation at initialization-time decides a static size of the frame, in words
- Negotiation at any time can dynamically change the size of the frame

The sending node SHOULD use slots not allocated to *IsoStreams* to send *μPkts* to its neighbor (multi-word *μPkts* MAY be sent in the same frame in multiple free slots). An *InitIsoStream μPkt* MUST be used to specify the size, payment, rate, slot, and next hop of a new *IsoStream*.
Some other possible *μPkt* uses include (but are not limited to):

- Probe for connection availability
- Request/confirm a link reservation
- Send *LinkLayerNodeAdvertisement*
- Synchronize clocks/frequencies (if used/needed)
- Switch *μPkts* for a different transport protocol
    - Computational *μPkts*
    - Physical location based routes?
- Update link state
    - Link drop announcements
- Transfer/exchange Credits
- Check credit statistics

μPkt data SHOULD be ignored/dropped/failed if uncorrectable errors are detected in a link layer frame.

The frequency of word arrivals for an isochronous slot MUST be a power of two words/second. For example, 2^13 = 8,192 words/second, or 2^14 = 16,384 words/second.

But at very high frequency, the precise definition of a second is relevant. The definition of a second on the IsoGrid is provided by the Link Slot Frequency Standard.

9.2 Link Slot Synchronized Frequency Standard

The IsoGrid system frequency standard is TCG: Geocentric Coordinate Time.
In order to provide isochronous streams, the IsoGrid runs with a synchronized frequency, very similar to the way an electrical grid runs on Utility Frequency (except much higher frequency).

Nodes MUST attempt to lock their link output frame rate using the TCG definition of a second. However, there are a number of ways that nodes MAY meet this requirement.

9.2.1 Stationary Nodes

If a node is stationary relative to its neighbors and has just a single link, the node MAY directly clock its output frame rate synchronized to its input frame rate. The link for these edge nodes MAY be arbitrarily long-lived.

If a node is stationary relative to some number of neighbor nodes, the node MAY tune an oscillator with respect to those input frame rates.

The tuned oscillator will be used to set the output frame rate. The neighbors that receive this as an input will use it to tune their own oscillator, which will be used to set the frame rate of the input links of its neighbors (including back to the first node). In this way, a stabilizing feedback loop will work to synchronize the entirety of The IsoGrid. The links for these stationary nodes MAY be arbitrarily long-lived.

To illustrate how this might be implemented within a node, here are a few examples:

- Combine all the input waveforms and use that combined waveform to tune the output frequency oscillator
- This waveform feedback could also be a digital process, where the 'fullness' of buffers determines the 'waveform' phase shift.
    - Buffers filling up: Advance the waveform
    - Buffers getting empty: Retard the waveform

This frequency synchronization strategy establishes a fundamental tradeoff between the following:

1. Higher link frame rate
2. Smaller buffers at each hop
3. Longer distance links
4. Higher tolerance for clock drift and clock skew

Longer links have more frames in transit. Higher frequency links also have more frames in transit. The more frames in transit, the more buffer is needed to accommodate clock instability versus ideal TCG.

Stationary nodes that have an amazingly stabile TCG input SHOULD bias their output frame rate to attempt to match the TCG frame rate. This is intended to pull its neighbor nodes closer to TCG. The IsoGrid as a whole is reliant on the collective work of all nodes with TCG inputs to ensure the entire network locks to the TCG frame rate over time. Note, this isn't likely to suffer from Tragedy of the Commons because there isn't any common economic reason for network participants to attempt to skew the network frame rate away from TCG.

9.2.2 Feedback-Mediated Link Frame Rate Synchronization

Given a Frame Rate, a Link Latency, and the measure of the clock's short-term stability, it's possible to specify the minimum buffer required to compensate for clock drift and skew.

Here is a basic mathematical expression that expresses the fundamental tradeoffs precisely.

Rate	Frame rate of the link, in frames / second

Distance	The round-trip link distance, in meters
c_{medium}	Speed of information travel in the transmission medium, in meters / second
Latency	The round-trip latency of the link := Distance/c_{medium}
ClockStability	Clock stability of the node, expressed as a fraction. For example: 1 part in a million --> 0.000001
MinBuffer	Minimum possible buffer required to accommodate clock drift and clock skew, in number of frames

*MinBuffer = Rate * Latency * ClockStability*

In practice, the required buffer will be larger, but it seems reasonable to expect that it's within two orders of magnitude. If a clock is stable to 1 part in 500,000 (ie. A simple quartz clock), then even if it takes 100x the *MinBuffer*, the underlying latency of the link is only increased by 0.02%.

9.2.3 Clock examples

A quartz clock, for example, typically has a short term stability of 1 part in half a million. This means that a link with one frame of buffer can have up to 500K frames in transit (round-trip) after which feedback-mediated link sync is impossible. For a 100km link, this leads to a maximum theoretical 50 mega-frame / second rate (with only 1 frame of buffer).
4 frames of buffer would allow 4 times the number of frames in-transit.
Here, a frame travelling round-trip across a single link is defined to be in-transit up until the frame is able to be used in the frequency feedback mechanism of its sender.
Clearly, quartz clocks alone aren't stable enough for use with extremely high-rate, long-distance connections, where link sync can be lost before the frequency feedback loop is able to correct the issue. A node MAY mitigate this issue by using larger buffers. Since the number of buffered frames at the end of a link would be quite small compared to the link itself, quartz clocks are likely to be good enough for most nodes for at least the next decade.

A GPSDO clock, on the other hand, typically has a short & long term stability of 1 part in 300,000,000,000. With this clock, a link with one frame of buffer can have up to 300G frames in transit (round-trip) before feedback-mediated link sync is impossible. For a 1000km link, this leads to a maximum theoretical 240 Tera-frames / second rate.

9.2.4 Dealing with bad clocks

Since most nodes rely on their neighbors to collectively lock to the TCG frame rate, a node with a misbehaving clock will have local impacts. It could potentially cause a group of neighbors to lose link sync frequently. Since this is a local issue, it can be dealt with at the local level by the

affected neighbors making the choice to stop using the bad clock as a clock synchronization source. That way, the bad clocked node alone has the consequences of the bad clock.
There is no need for a Time Cop :-)

9.2.5 Mobile Nodes

Nodes that move, but that stay 'near' a starting position, MAY compensate for the movement with buffers; either logical buffers, or physical buffers (in the form of a longer link distance). In so doing, they MAY provide arbitrarily long link connectivity.
However, nodes that move arbitrarily long distances, MUST have transitory links. These nodes MAY pre-compute the buffer requirements for a transitory link. As the node continues to move, it can only maintain the link for so long before the buffers are exhausted.
For a node that is moving axially between two other nodes, it MAY consider clocking the output frame rate based on the opposing input frame rate. Doing so could allow a smaller buffer.

9.3 IsoGrid µPkt Standard

The IsoGrid Protocol mandates the ability to transmit a *µPkt*. However, the specific link-layer protocol for sending these isn't globally specified. A bilingual node SHOULD translate between two nodes that don't speak the same link-layer protocol.
All link-layer protocols MUST have some sort of *µPkt* enveloping mechanism that allows the nodes to agree on where each *µPkt* begins and ends and to separate the *µPkt* data from the slots allocated to *IsoStreams*.
All link-layer protocol designs SHOULD consider how to version the *µPkt* enveloping mechanism.

All link-layer protocols MUST support a means to envelope the following *µPkt* members:

Member	**Description**
*µRoute*_Type	The routing protocol used by the *µPkt*.
*µRoute*_FullSize	Defines the size of the data portion of the *µRoute*
*µRoute*_Data	The full routing data for the *µPkt*
*µPkt*_Type	Defines the protocol used in the data portions the *µPkt*.
*µPkt*_FullSize	Defines the size of the data portion of the *µPkt*.
*µPkt*_Data	Holds the data fields of the *µPkt*.

All IsoGrid nodes MUST be able to support *µPkts* where *µRoute*_FullSize is up to (and including) 64 octets. All IsoGrid nodes MUST be able to support *µPkts* where *µPkt*_FullSize is up to (and including) 256 octets.

The *μPkt* Types that are supported by the node MUST be declared in its *LinkAdvertisement*. *μPkts* with an unrecognized *μPkt*_Type MUST be passed along the route unmodified.
The *μRoute*_Types that are supported by the node MUST be declared in its *LinkAdvertisement*. A node, knowing its neighbors well, SHOULD prefer to send *μPkts* to nodes that understand the FullType of the *μRoute*.

9.4 Initial Linkups

A mobile node might not have any active links if it arrives at a location without previously knowing it was heading there (and so it was unable to pre-provision a link). Nodes also have to handle connecting to the network for the first time. A newly arrived node MAY make a request to establish a link with a nearby node. Nodes MAY limit the number of these requests they accept per unit time to avoid abuse such as a denial of service attack. For example, a node could require the new node to:

1. Perform a compute task, or proof of work
2. Perform a data relay task
3. Be in close physical proximity to the node
4. Make lots of attempts, and the node only pays attention infrequently at random intervals
5. Wait a bit of time for any previous requests to complete (requests are simply throttled)
6. Etc.

9.5 Credits

Nodes SHOULD have their node-local credit units be scaled such that the cost of the least expensive link is as close to 1 as possible but not less than 1. If the cost (as expressed in node-local units) is less than 1, then the subtraction step becomes less precise (it MUST be rounded up to the nearest expressible unit).
Nodes MUST offer a rate at which it will exchange credits with each neighbor. Neighbor nodes MAY choose to have a maximum per-diem settlement, to provide a backstop against possible software bugs or security vulnerabilities in the system.

10 Network Layer (*μPkt*)

The network layer supports routing of extensible *μPkts* using extensible *μRoute* protocols.

10.1 μPkt and μRoute Types

- Extendable *μPkt* Types
- Separately extendable *μRoute* Types
- Able to be implemented in hardware
- Even if only the sender understands an extended Type, the full *μPkt* can still make it to the destination

- Switches are able to treat a *µPkt* or *µRoute* as a base type if they don't understand the extended version

µPkt and *µRoute* Types follow a strict single-inheritance extension hierarchy. The root base type of *µPkt* is called *µPktRoot*, and all *µPkt* Types defined in this specification derive at the root from it. The root base type of *µRoute* is called *µRouteRoot*, and all *µRoute* Types defined in this specification derive at the root from it.
It's possible to define additional roots, but in order to have a new root be useful, all switches along an IsoGrid route would have to support it: This would be a similar problem to the IPv4 to IPv6 transition.
µPktRoot and *µRouteRoot* each contain a 32 bit FullType field: This provides plenty of space for protocol diversity without having to have centralized assignment of precious numbers. It seems unlikely that switches will ever support more than a billion different *µPkt* or *µRoute* types. If it really gets to that state, perhaps a new root or a completely different network layer would be necessary.
Since *µPkt* and *µRoute* Types follow a strict single-inheritance hierarchy, if a derived type adds fields, it MUST only do so immediately following the fields of the base type. *F*ields are statically sized, so the whole Type is also statically sized. Extensibility design is in the class hierarchy, it isn't designed to exist within the fields. This design supports parallelizing and pipelining switch implementations: Once the Type is recognized, the subsequent actions of the switch regarding that *µPkt* SHOULD mostly be set in stone.

10.2 µPkt and µRoute Type Requirements

Each *µPkt* MUST be sent over the next link by applying the most-derived types supported by the next switch. If the next switch only supports a base type, then the *µPkt* MUST be sent to that switch formatted as that base type: However, the FullType field and the fields of the extended type MUST be passed along unmodified. If a switch receives a *µPkt* with a FullType it's able to support, it MUST do so, and MUST pass it on to the next switch with the most-derived Type that the next switch has announced support for.

10.2.1 *µPktRoot*

FullType: 0x 4B D2 93 D2

µPktRoot MAY be declared to be the root of any *µPkt*.

This has no inherited members, and just consists of the following:

Member	**Size**
FullType	8 octets
µPktPaymentCredits	8 octets

10.2.1.1 *μPktPaymentCredits*

This is a 64 bit floating point value that describes the amount of credits being transferred along with the *μPkt*.

0:3	(4 bit) Reserved, MUST be zero
4:55	(52 bit) a 53 bit fraction (with an assumed initial 1 'hidden bit')
56:63	(8 bit) exponent, biased by 0

To calculate the amount of credit to provide for a simple *μPkt*, and the route and prices are known, work backwards: Start with the amount the final node is requesting in node local units. Convert to the previous node's units, then add that link's cost (rounding up to the nearest expressible value), and repeat.

Requirements:

A. Arithmetic MUST be as per IEEE 754 standard *binary64*
B. The switch receiving this value MUST deterministically subtract the exact amount of credits that covers the node's advertised cost per word of handling the *μPkt*
C. If, after the subtraction step, the credits would be negative, the switch MUST drop the *μPkt*.
D. If, after the subtraction step, the value is greater than the advertised amount of credits the switch is willing to transfer per *μPkt*, the switch MUST fail the packet with *ExceededMaxCredit FailureCode* (or drop the *μPkt* if it doesn't inherit from *μPktWithReply)*.
E. Otherwise, the switch MUST scale the remaining credits appropriately at the advertised exchange rate, and MUST forward the resulting value in the *μPktPaymentCredits* field of the *μPkt* sent over the appropriate outgoing link

10.2.2 *μRouteRoot*

FullType: 0x 75 B3 6A A9

μRouteRoot MAY be declared to be the root of any *μRoute*.

This has no inherited members, and just consists of the following:

Member	Size
FullType	8 octets

10.3 Standard IsoGrid μRoute Type Definitions

The following *μRoute* Types MUST be supported by all IsoGrid switches. Support for *μRouteByIsoStreamHeaders* is also critically required, but this is described in the *IsoStream* layer.

10.3.1 *μRouteByCachedRoute*

Inherits: *μRouteRoot*

FullType: 0x 45 2E B7 CF

This MAY be used to simply have a *µPkt* follow a previously cached route in the original direction of the route. This *µRoute* Type isn't useful until a route has been cached using a different *µRoute* Type, like *µRouteByIsoStreamHeaders*.

In addition to its inherited members, it consists of the following:

Member	Size
RouteId	8 octets

10.3.2 *µRouteByCachedRouteReversed*

Inherits: *µRouteRoot*
FullType: 0x 41 2F 1A B2

This MAY be used to simply have a *µPkt* follow a previously cached route in the reversed direction of the original route. This *µRoute* Type isn't useful until a route has been cached using a different *µRoute* Type, like *µRouteByIsoStreamHeaders*.

In addition to its inherited members, it consists of the following:

Member	Size
RouteId	8 octets

10.4 Standard IsoGrid µPkt Type Definitions

The following *µPkt* Types MUST be supported by all IsoGrid switches. Support for *InitIsoStream* is also required, but this is described in the *IsoStream* layer.

10.4.1 *µPktWithHopCounter*

Inherits: *µPktWithHopCounter*
FullType: 0x 4F 8F 8B E2

µPktWithHopCounter MAY be used to simply count the number of hops along a route.

In addition to its inherited members, it consists of the following:

Member	Size
HopCounter	8 octets

10.4.1.1 ***HopCounter***

HopCounter is a 64 bit unsigned integer that counts each hop of the *µPkt*.

Requirements:

A. Each node MUST increment this integer by exactly 1 every hop.

B. The starting node MUST randomly choose a 64 bit number and clear the most significant bit to 0.

10.4.2 *µPktWithReply*

Inherits: *µPktWithHopCounter*
FullType: 0x 4D 7B 99 91
Success Reply: *µPkt_Success*
Failure Reply: *µPkt_Failure*

µPktWithReply MAY be used to send a request with a built-in reply and payment for the reply.

In addition to its inherited members, it consists of the following:

Member	**Size**
ReplyPaymentCredits	8 octets
µPktId	4 octets

TODO: Specify how to meet the requirement that a client that isn't fully aware of the cost of the route is able to calculate the round trip exchange rate of the PaymentCredits. For example, can the client use the initial *µPktPaymentCredits* & *ReplyPaymentCredits* along with the *ReplyPaymentCreditsAtSuccess* & *µPktPaymentCreditsAtSuccess* to do this? If not, an additional member that tracks the exchange rate at each hop will be required.

10.4.2.1 ***ReplyPaymentCredits***

ReplyPaymentCredits is a 64 bit floating point value that adds up the cost of a reply payment to send it back to the source node.

The value is formatted in the same way as *µPktPaymentCredits*.

Requirements:

A. The starting node MUST initialize this value with the cost that the node would charge to accept a simple 256 octet reply *µPkt*.
B. Each node MUST scale this value to node-local units by the exchange rate from the previous hop.
C. Each node MUST add to this value the cost that node would charge to switch a single 256 octet simple, data-only, reply *µPkt*.
D. Each node MUST round this value up to the nearest expressible value.
E. If *ReplyPaymentCredits* exceeds *µPktPaymentCredits*, then the node MUST fail the outbound *µPkt* and reply back with *µPkt_Failure* instead.

10.4.3 *GetRouteUtilizationFactor*

Inherits: *µPktWithReply*

FullType: 0x F6 3F F9 52
Success Reply: *GetRouteUtilizationFactor_Success*
Failure Reply: *GetRouteUtilizationFactor_Failure*

This MAY be used to determine how full a route is.

In addition to its inherited members, it consists of the following:

Member	**Size**
MostCongestionHopCount	8 octets
MostCongestionLevel	1 octet
LeastCongestionHopCount	8 octets
LeastCongestionLevel	1 octet

TODO: Specify member handling

10.5 Standard IsoGrid µPkt Success Reply Type Definitions

The following *µPkt* success Reply Types MUST be supported by all IsoGrid switches. Support for *InitIsoStream_Success* is also required, but this is described in the *IsoStream* layer.

10.5.1 *µPkt_Success*

Inherits: *µPktRoot*
FullType: 0x 47 9E 96 AA

This MUST be used as the success reply for a *µPkt* that has a FullType of *µPktWithReply*. This, or a type that inherits from this, MUST be used as the success reply for any *µPkt* that inherits from *µPktWithReply*.

In addition to its inherited members, it consists of the following:

Member	**Size**
µPktId	4 octets
HopCountAtSuccess	8 octets
ReplyPaymentCreditsAtSuccess	8 octets
µPktPaymentCreditsAtSuccess	8 octets

TODO: Specify member handling

10.5.2 *GetRouteUtilizationFactor_Success*

Inherits: *µPkt_Success*
FullType: 0x 94 2B 5C 20

This MUST be used as the success reply for a *µPkt* that has a FullType of *GetRouteUtilizationFactor*. This, or a type that inherits from this, MUST be used as the success reply for any *µPkt* that inherits from *GetRouteUtilizationFactor*.

In addition to its inherited members, it consists of the following:

Member	**Size**
MostCongestionHopCount	8 octets
MostCongestionLevel	1 octet
LeastCongestionHopCount	8 octets
LeastCongestionLevel	1 octet

TODO: Specify member handling

10.6 Standard IsoGrid µPkt Failure Reply Type Definitions

The following *µPkt* failure Reply Types MUST be supported by all IsoGrid switches. Support for *InitIsoStream_Failure* is also required, but this is described in the *IsoStream* layer.

10.6.1 *µPkt_Failure*

Inherits: *µPktRoot*
FullType: 0x EA 38 35 A4

This MUST be used as the failure reply for a *µPkt* that has a FullType of *µPktWithReply*. This, or a type that inherits from this, MUST be used as the failure reply for any *µPkt* that inherits from *µPktWithReply*.

In addition to its inherited members, it consists of the following:

Member	**Size**
µPktId	4 octets
HopCountAtFailure	8 octets
ReplyPaymentCreditsAtFailure	8 octets
µPktPaymentCreditsAtFailure	8 octets
GeoLocationAtFailure	16 octets
NodeIdAtFailure	16 octets
FailureCode	1 octet

TODO: Specify member handling

10.6.1.1 *FailureCode*

The *FailureCode* describes the reason for the failure. Possible reasons are:

Code	**Value**	**Description**
NoSuchRoute	1	The node doesn't know how to route the *µPkt*

OverCongested	2	The next hop is too congested to send the *µPkt*
CreditsExhausted	3	*ReplyPaymentCredits* exceeds *µPktPaymentCredits*
IsoStream_CacheCollision	4	The *RouteId* is already associated in the forward direction.
IsoStream_CacheCollisionReversed	5	The *RouteId* is already associated in the reversed direction.
IsoStream_ExceededMaxWordRate	6	The *IsoStreamWordRate* exceeded the maximum that the node supports.
IsoStream_BelowMinWordRate	7	The *IsoStreamWordRate* was below the minimum that the node supports.
IsoStream_ExceededMaxCredit	8	*IsoStreamPaymentCredits* exceeded the maximum the node is willing to transfer per word of *IsoStream*.
ExceededMaxCredit	9	*µPktPaymentCredits* exceeded the maximum the node is willing to transfer per *µPkt*.
InsufficientCredits	10	The *µPktPaymentCredits* were insufficient for the destination node to perform the final handling of the *µPkt*.

10.6.2 *GetRouteUtilizationFactor_Failure*

Inherits: *µPkt_Failure*
FullType: 0x 48 80 A4 58

This MUST be used as the failure reply for a *µPkt* that has a FullType of *GetRouteUtilizationFactor*. This, or a type that inherits from this, MUST be used as the failure reply for any *µPkt* that inherits from *GetRouteUtilizationFactor*.

In addition to its inherited members, it consists of the following:

Member	**Size**
MostCongestionHopCount	8 octets
MostCongestionLevel	1 octet
LeastCongestionHopCount	8 octets
LeastCongestionLevel	1 octet

TODO: Specify member handling

11 Transport Layer (*IsoStreams*)

An example *IsoStream* across 2 switches (3 links) might look like this:

Source Sends	Switch A Sends	Switch B Sends	Destination Processes as Payload
*	*	*	*
ME - *InitIsoStream µPkt* SO – IsoStreamRoute_A	*	*	*
SO – IsoStreamRoute_B	ME - *InitIsoStream µPkt* SO – IsoStreamRoute_B	*	*
SO – IsoStreamRoute_B	SO – IsoStreamRoute_B	*	*
SO – IsoStreamHeader0	SO – IsoStreamHeader0	ME - *InitIsoStream µPkt* SO – IsoStreamHeader0	SO – IsoStreamHeader0
SO – IsoStreamHeader1	SO – IsoStreamHeader1	SO – IsoStreamHeader1	SO – IsoStreamHeader1
…	…	…	…
SO – IsoStreamHeader9	SO – IsoStreamHeader9	SO – IsoStreamHeader9	SO – IsoStreamHeader9
SO – HeaderEnd	SO – HeaderEnd	SO – HeaderEnd	SO – HeaderEnd
SO – Payload0	SO – Payload0	SO – Payload0	SO – Payload0
SO – Payload1	SO – Payload1	SO – Payload1	SO – Payload1
SO – Payload2	SO – Payload2	SO – Payload2	SO – Payload2
SO – Payload3	SO – Payload3	SO – Payload3	SO – Payload3
SO – Payload4	SO – Payload4	SO – Payload4	SO – Payload4
SO – Payload5	SO – Payload5	SO – Payload5	SO – Payload5
SO – Payload6	SO – Payload6	SO – Payload6	SO – Payload6
SO – Payload7	SO – Payload7	SO – Payload7	SO – Payload7
SO – Payload8	SO – Payload8	SO – Payload8	SO – Payload8
SO – Payload9	SO – Payload9	ME – *LostWord*	No (*LostWord*)
SO – Payload10	SO – Payload10	SO – Payload10	SO – Payload10
SO – Payload11	SO – Payload11	SO – Payload11	SO – Payload11
SO – Payload12	SO – Payload12	SO – Payload12	SO – Payload12
…	…	…	…
SO – Payload79	SO – Payload79	SO – Payload79	SO – Payload79
SO – Payload80	ME – *LostWord*	ME – *LostWord*	No (*LostWord*)
SO – Payload81	SO – Payload81	SO – Payload81	SO – Payload81
SO – Payload82	SO – Payload82	SO – Payload82	SO – Payload82
ME – Filler (NoData)	ME – Filler (NoData)	ME – Filler (NoData)	No – Filler (NoData)
ME – Filler (NoData)	ME – Filler (NoData)	ME – Filler (NoData)	No – Filler (NoData)
ME – Filler (NoData)	ME – Filler (NoData)	ME – Filler (NoData)	No – Filler (NoData)
*	SO – IsoStreamRoute_A	SO – IsoStreamRoute_A	No (Footer)
*	*	SO – IsoStreamRoute_B	No (Footer)
*	*	SO – IsoStreamRoute_B	No (Footer)
*	*	*	*

'*' Indicates the data word isn't part of this specific *IsoStream* (it could be anything).
Read the rows as a type of timeline: A sending switch only has access to the previous and current rows.

In the above example, the payload being sent to the destination is exactly 83 words, with a 10 word header. The source sent an *InitIsoStream µPkt*, with the *IsoStreamWordCount* set to exactly 100 words. Switch 'A' handles a 1 word long *IsoStreamRoute* that points to Switch 'B'. Switch 'B' handles a 2 word long *IsoStreamRoute* that points to a link heading to the final destination endpoint. Switch 'A' inserts its 1 word footer (which matches the *IsoStreamRoute* that is stripped off the beginning) at the end of the 100 words it sends. Switch 'B' inserts its 2 word footer at the end of the 100 words it sends. In this way, with an *IsoStream* with 100 *IsoStreamWordCount*, all nodes process exactly 100 words.
Notice in this example, that the link between Switch 'A' and Switch 'B' lost or corrupted Payload9, and the destination receives *LostWord* instead of Payload9. Also, the link between the source and Switch 'A' lost or corrupted Payload80, and thus Switch 'A', Switch 'B', and the destination won't receive Payload80; receiving *LostWord* instead. This high rate of loss isn't expected, but shown merely as an example.

11.1 InitIsoStream

Inherits: *µPktWithReply*
FullType: 0x 46 6A 98 C0
Success Reply: *InitIsoStream_Success*
Failure Reply: *InitIsoStream_Failure*

InitIsoStream MAY be used as a means to start an *IsoStream*. All IsoGrid nodes MUST support the *InitIsoStream µPkt*.

In addition to its inherited members, it consists of the following:

Member	**Size**
IsoStreamWordCount	2 octets
IsoStreamPaymentCredits	8 octets
IsoStreamWordRate	1 octet

11.1.1 *IsoStreamWordCount*

IsoStreamWordCount is a floating point value describing the number of words to be sent on the *IsoStream*.

0:6 (7 bit) exponent, biased such that the minimum expressible value is 4
7:14 (9 bit) a 10 bit multiplier (with an assumed initial 1 'hidden bit')

Requirements:

A. Each node MUST remember this value for the *IsoStream* on this slot
B. Each node MUST begin counting the words of the *IsoStream* with the first word that follows a valid *InitIsoStream µPkt*
C. This value is used to know when the node should send the *IsoStreamFooter* words.
D. Minimum expressible value: 4 words
E. Maximum expressible value: approx. 2^75 words

F. If the *IsoStreamWordCount* is greater than the allowable maximum, as advertised by the switch, the switch MUST drop the *IsoStream* connection request.

11.1.2 *IsoStreamPaymentCredits*

IsoStreamPaymentCredits is a floating point value describing the number of credits per word to pay for the *IsoStream*.

The value is formatted in the same way as *μPktPaymentCredits*.

To calculate the amount of credit to provide for an *IsoStream*, and the route and prices are known, work backwards: Start with the amount the final node is requesting in node local units. Convert to the previous node's units, then add that link's cost (rounding up to the nearest expressible value), and repeat.

Requirements:

A. Arithmetic MUST be as per IEEE 754 standard *binary64*
B. The switch receiving this value MUST deterministically subtract the exact amount of credits that covers the node's advertised cost per word of handling the *IsoStream*
C. If, after the subtraction step, the credits would be negative, the switch MUST drop the *IsoStream* initialization request.
D. If, after the subtraction step, the value is greater than the advertised amount of credits the switch is willing to transfer per word, the switch MUST drop the *IsoStreamInit*.
E. In all other cases, the switch MUST scale the remaining credits appropriately at the advertised exchange rate, and MUST forward the resulting value in the *IsoStreamPaymentCredits* field of the next *InitIsoStream μPkt* over the appropriate outgoing link and targeting an available output slot.

11.1.3 *IsoStreamWordRate*

IsoStreamWordRate defines the rate at which words are sent on the *IsoStream*. The value is expressed as an 8 bit exponent of 2. This is a power of two, as in 2 ^ (*IsoStreamWordRate*) words per TCG second.

Requirements:

A. If the value is greater than the advertised supported maximum word rate of the outbound link, the switch MUST fail the *InitIsoStream* with *ExceededMaxWordRate*.
B. If the value is smaller than the advertised supported minimum word rate of the outbound link, the switch MUST fail the *InitIsoStream* with *BelowMinWordRate*.

11.1.4 Miscellaneous

Each node self-declares the number of words it MAY buffer (ideally 1 word) when transmitting the words across the required outgoing slot. The node MUST advertise this buffer size.

Reasoning for the existence of *InitIsoStream*: The credit value is a required part of every connection request and acts essentially as a credit-based hop-count limit.

11.2 InitIsoStream_Success

Inherits: *µPkt_Success*
FullType: 0x 4A C2 A8 B1

This MUST be used as the success reply for a *µPkt* that has a FullType of *InitIsoStream*. This, or a type that inherits from this, MUST be used as the success reply for any *µPkt* that inherits from *InitIsoStream*.
This *µPkt* Type has only the members it inherits.

11.3 InitIsoStream_Failure

Inherits: *µPkt_Failure*
FullType: 0x 51 AF 6D 77

This MUST be used as the failure reply for a *µPkt* that has a FullType of *InitIsoStream*. This, or a type that inherits from this, MUST be used as the failure reply for any *µPkt* that inherits from *InitIsoStream*.
This *µPkt* Type has only the members it inherits.

11.4 µRouteByIsoStreamHeaders

Inherits: *µRouteRoot*
FullType: 0x 46 FD BF D9

In order to declare an initial route, the source node MAY use the *µRouteByIsoStreamHeaders µRoute* Type in an *InitIsoStream* (or a *µPkt* that inherits from it).
In this *µRoute* Type, the source provides a variable-length series of route instructions to be used at each hop (Source Routing).

All IsoGrid nodes MUST support *µRouteByIsoStreamHeaders* for *InitIsoStream* (or a *µPkt* that inherits from it). *µRouteByIsoStreamHeaders* MUST NOT be used to route just a *µPkt* without an associated *IsoStream* (where the *µPkt* contains the listing of routes).

In addition to its inherited members, it consists of the following:

Member	**Size**
µPktId2	4 octets

The expression *µPktId* + (*µPktId2* << 32) forms the 8 octet *RouteId*. This *RouteId* is also used in *µRouteByCachedRoute* and *µRouteByCachedRouteReversed*.

Requirements:

A. In the isochronous slot(s) allocated by an *InitIsoStream µPkt*, the source node MUST stream a series of *IsoStreamRoute* words that define the route, one switch at a time.
B. The number of octets consumed by each switch and their meaning MUST be defined and advertised by the switch.
C. If an *IsoStreamRoute* isn't recognized by a switch, the *InitIsoStream* MUST be failed.
D. If an *IsoStreamRoute* is recognized by a switch:
    a. The switch MUST send an *InitIsoStream µPkt* that allocates output slot(s) on the required link which stream the remaining *IsoStream* words.
    b. If the InputLink+*RouteId* is already associated with a different *IsoStreamRoute*, the *InitIsoStream* MUST be failed with *CacheCollision FailureCode*.
    c. If the OutputLink+*RouteId* is already associated with a different InputLink, the *InitIsoStream* MUST be failed with *CacheCollisionReversed FailureCode*.
    d. The switch MUST associate the InputLink+*RouteId* with the *IsoStreamRoute* at this hop
        i. While valid, this MUST be usable for *µRouteByCachedRoute*.
    e. The switch MUST associate the OutputLink+*RouteId* with the InputLink at this hop
        i. While valid, this MUST be usable for *µRouteByCashedRouteReverse*
    f. These associations MUST be valid for a minimum of 1023 seconds following the last usage of this name.
    g. These associations MAY remain valid for up to 2049 seconds following the last usage of the name.
    h. These associations MUST NOT be recognized after 2049 seconds following the last usage of the name.
E. After the *InitIsoStream µPkt* is received, the switch MUST mark the targeted input slot as an *Active IsoStream*.

TODO: Define how to pack multiple route hops into each word. Possible bit sizes of each hop: 8(-2), 16(-2), 32(-2), 64(-4), 128(-4), 256(-4), 512(-4).

Let's say there's 1,000 octets of *IsoStreamRoute* (500 hops, 2 octets each hop)
Effect of 2 octet headers on latency at 1MB/s when travelling a 1,000 hop route:
(2 B / 1,000,000 B/s) * 1,000 hops = 2 B / 1,000 B/s = 2ms

11.5 Active IsoStream

While an *IsoStream* is *Active*, a switch MUST copy each word on the input slot to the allocated output slot.
Temporary physical link interference MUST NOT deactivate an active *IsoStream*.
When the previously specified *IsoStreamWordCount* is reached, the switch MUST send the required *IsoStreamFooter* words, matching the exact number and value of *IsoStreamHeader* words that were consumed by this switch in the *IsoStreamRoute* sequence that activated this *IsoStream*.

12 Session Layer: Error Correction Coded Flow

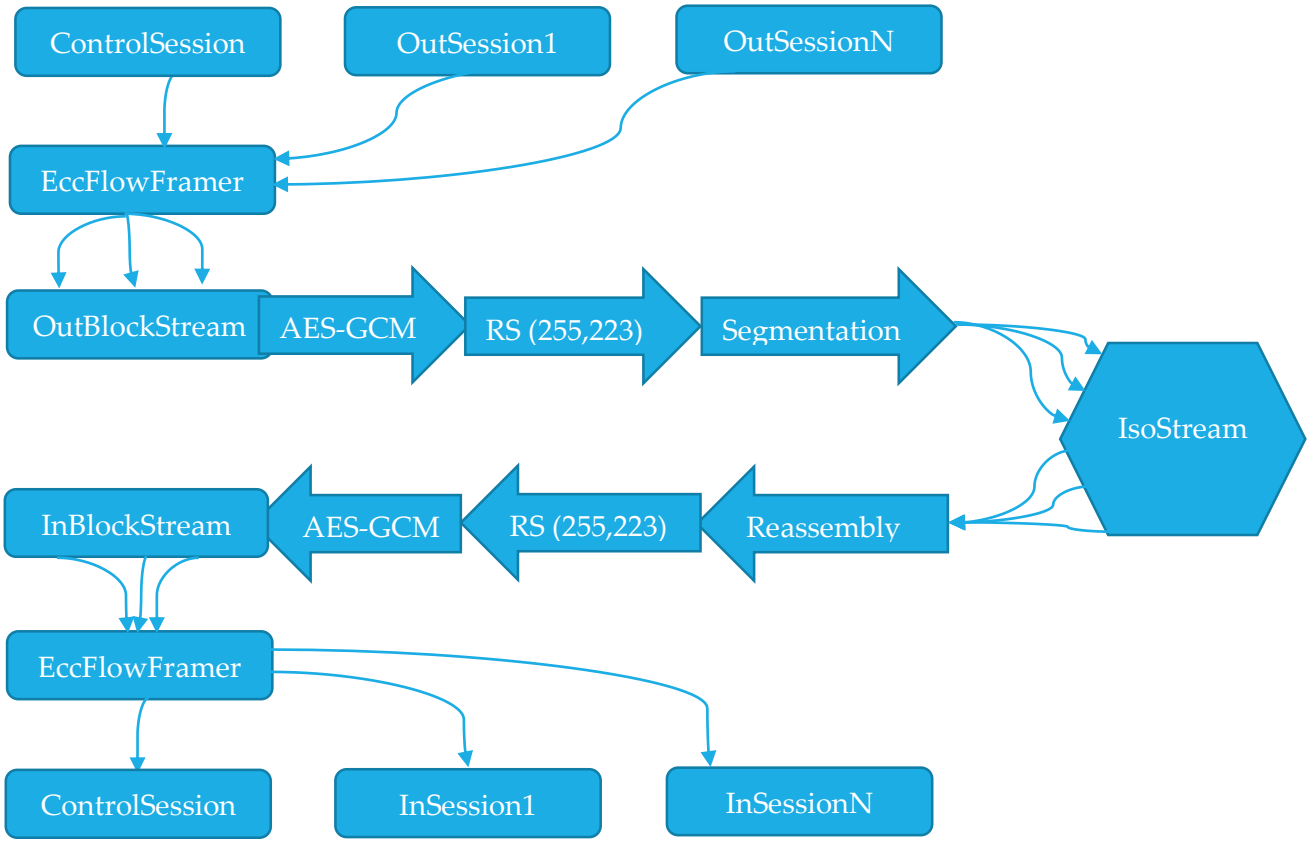
TODO: This section is still a work in progress and may contain raw notes!

The IsoGrid defines a single standard session layer protocol (*EccFlow*). Much like the TCP/IP protocol stack defines multiple standard Transport layer protocols (TCP, UDP, etc.), the *EccFlow* protocol has options and sub-options that cover the same use-cases (and more).

EccFlow provides the application layer with the capability to create and use one or more *InSessions* and *OutSessions*. Each *InSession* is presented to the application layer as a series of packets, and the application can send a series of packets down an *OutSession*.

EccFlow has a client side and a server side. The *EccFlow* client has an input *ControlSession* and an output *ControlSession* matched with the opposite on the server side of the *EccFlow*, through which top level protocol control messages can be sent to the other node. The client is responsible for the credit payments required to maintain the *EccFlow*.

12.1 EccFlow Overview



Once initialized, the *EccFlow* protocol involves 5 steps on the outbound and the 5 reverse steps on the input side.

1. The many sessions are formatted into a series of 32+1 octet *EccFlowFrames*
2. The frames are fed into one or more *OutBlockStreams*
3. The data is encrypted with AES-GCM
4. The data is encoded with Reed-Solomon (255, 223)
5. The data is segmented into multiple *IsoStreams* and sent over the IsoGrid to the destination

At the destination, the streams are reassembled into blocks to be fed into the Reed-Solomon decoder. Finally, the data is decrypted using AES-GCM and the result of the *InBlockStreams* are handed to the *EccFlowFramer* layer.

Each *EccFlow* is initialized by the client side with a 3-way handshake that performs a key exchange and tests the available routes.

12.2 Session Options

Each session MUST be set as either Isochronous or Asynchronous. The notion of "Reliable Isochronous" doesn't exist, because Isochronous is unreliable by definition. Also, since Isochronous has no retransmission, it's always ordered: So the *EccFlowIsochSession* option has no sub-options.
However, each Async session MUST be set as either reliable or unreliable. Each Reliable Async session MUST be set as either TCP or Unordered. Each Unreliable Async session MAY be presented to the application layer as Raw, Dropping, or Reset. The choice of Raw vs. Dropping vs. Reset has no protocol impact.

Async sub-option	**Description**
TCP	Uses actual TCP layered on top of an *EccFlow* session. Lost packets are resent, and later packets are held in a destination buffer until the lost packet arrives.
Reliable Unordered	Lost packets are resent, and delivered out of order
Unreliable Raw	All recognizable packets are delivered to the application layer, even if they contain bad data (and don't request resend)
Unreliable Dropping	If enough *IsoStreams* lose data, drop the packet (and don't request resend)
Unreliable Reset	If enough *IsoStreams* lose data and a packet is dropped, drop the entire session

Isochronous Sessions:

- 1 word to define rate: 7 bits for exponent, biased such that minimum expressible value is 128 octets/s. 9 bit multiplier (with an additional assumed initial 1 'hidden bit'). 0 means Asynchronous (Dynamic) Flow Control

Asynchronous Sessions Flow Control:

- Gauge the rate at which the destination can handle the data and the rate at which the source can send it
- Allocate (pay for) a ring buffer at the destination

Other EccFlow Responsibilities:

- Building up and tearing down the underlying *IsoStreams* at the transport layer to try to keep the buffer approximately half full
- Deciding the cost/benefit of additional ring buffer vs. paying for additional link redundancy
- Maintaining a mutually acceptable credit balance between the two nodes
- Store the *EccFlow* context as long as the client has provided adequate credits for the server to do so (the server MUST declare the cost of such storage)

Implicitly, all future reply messages go down the *EccFlow* in the reverse direction.
The *EccFlow* between two nodes constitutes an arbitrarily long term bi-directional session.

12.3 Congestion Control

TCP contains congestion avoidance and fairness algorithms. Instead of relying on endpoints implementing a 'fair' TCP (which breaks down with malicious nodes), IsoGrid requires endpoints to pay for their use of the *IsoStream* (network) layer. So, *EccFlow* doesn't have fairness algorithms.
IsoGrid doesn't exhibit the problem of congestive collapse, because active *IsoStreams* are able to use their slot even in the presence of 100% load, and new *InitIsoStream* requests MUST be dropped unless they have a higher priority. Higher priority *InitIsoStream* requests MUST replace existing low-priority *IsoStreams*. *EccFlow* implementations SHOULD consider dynamically dropping to lower word rates in the presence of congestion on a particular link (thereby giving preference to alternative routes with cheaper non-congested links). *EccFlow* implementations SHOULD consider doing this randomly, with a lower probability at medium-high congestion, and the highest probability for the highest congestion. This avoids the problem of nodes oscillating traffic above and below some predefined limit.

12.4 *EccFlowFramer*

The *EccFlowFramer* operates by creating two buckets of data, isochronous data and asynchronous data. The bandwidth of the *EccFlow* is dynamically sized with the following priorities:

1. Keep all isochronous data moving with minimal buffers
2. Increase the bandwidth to accommodate asynchronous data if there is enough available for a round-trip time's worth of additional data
3. Decrease the bandwidth ASAP if there is unused bandwidth

The *EccFlowFramer* turns the many sessions into a single stream of interleaved isoch & async packet data frames, called *EccFlowFrames*.

Each *EccFlowFrame* consists of a 1 octet preamble and a 32 octet payload fragment. Multiple consecutive *EccFlowFrames* can be strung together to form a larger payload. This style of framing is designed such that the data self-synchronizes to the start of packets even after one or more corrupted/lost frames. The isochronous nature of the *IsoStream* layer below means that it's not possible to skip a frame without noticing its absence.

Preamble octet bit assignments:

- 1 bit to indicate Change-Session (indicates the presence of *SessionId* fields)
- 1 bit to indicate Start-of-Packet
- 2 bits for number of additional *SessionId* octets 0,2,4,8,
- 4 bits for significant part of *SessionId*
- So, *SessionId* size is 0.5, 2.5, 4.5 or 8.5 octets

Optional Support:

- Support for larger fragment sizes
- Support for using the 6 least significant bits for:
    - Extra 0.75 octets of data per fragment (after the first fragment)
    - Splitting a fragment into 2 parts to fit multiple small packets

The *PayloadSize* fields of the below *SessionProtocols* are variable-length in order to handle a wide range of payloads with minimal overhead. The following table describes the sizing mechanism:

Value of 2 most significant bits	**Rest of *PayloadSize* field size in bits**	**Additional octets after the first *PayloadSize* field octet**
0	6	0
1	14	1
2	22	2
3	118	14

The packet structure is defined by the *SessionProtocol* and associated with the *SessionId* when a new session is created. The IsoGrid Protocol requires that all nodes support creating *EccFlow* sessions with the following specific *SessionProtocols*:

12.4.1 *EccFlowAsyncSession* Protocol

SessionProtocolId: 0x 973D AFC6 2F19 4497
Packets in this *SessionProtocol* MUST have the following structure:

Member	Size
PayloadSize	1-15 octets
Payload	*PayloadSize* octets

12.4.2 *EccFlowIsochSession* Protocol

SessionProtocolId: 0x B2BC BFAC 7C27 667C

Packets in this *SessionProtocol* MUST have the following structure:

Member	Size
PayloadSize	1-15 octets
Payload	*PayloadSize* octets

12.4.3 *EccFlowTcpSession* Protocol

SessionProtocolId: 0x 91E6 E8E0 CB45 1F72

This *SessionProtocol* has a normal TCP header and acts just like a TCP packet except it runs on top of an *EccFlow* session (as if it were a simple point-to-point link) rather than on top of IP.

12.4.4 *EccFlowReliableUnorderedSession*

TODO: Design an *EccFlowReliableUnorderedSession* packet structure and protocol.

12.4.5 *ControlSession* Protocol

The *ControlSession* MUST assume packets are formatted with the following statically-sized structure:

Member	Size
Type	1 octet
Payload	*31* octets

SessionId of 0 MUST be used to designate the *ControlSession* in both directions.

The most significant bit of *SessionId* defines the allocation ownership of the rest of the *SessionId*. *SessionIds* with the most significant bit cleared are reserved for allocation by the client. *SessionIds* with the most significant bit set are reserved for allocation by the server. A *SessionId* only uniquely identifies a session if the direction is already given: Two sessions in opposite directions that share the same *SessionId* value are not necessarily related.

If the protocol for an *EccFlowControlPacket* type requires reliable delivery, the protocol MUST provide for it. One way to do this could be to track a round trip time estimate and automatically resend the packet if the expected response isn't received in time.

TODO: Flesh out the design for the following control packet types (right now these are just notes):

- *EccFlowFrame* initialization parameters
- *StartSessionId*

    - 8 octet *SessionProtocolId*
    - 8.5 octet *SessionIdForRequest*
    - 8.5 octet *SessionIdForResponse*
    - Example *SessionProtocols*:
        - *GetLinkAdvertisement*(s)
        - *ProvideLinkAdvertisement*(s)
        - *GetBestRoutes*
- Alias (rename) a *SessionId*
- Query support for some App layer service (*SessionProtocolId*)
- Announce/confirm support for some App layer service (*SessionProtocolId*)
- Packet that tells the receiver what the time counter was at the start of the packet. This MUST be used by the receiver to ensure that frames sent on two or more underlying *BlockStreams* are reliably ordered
    - Frames in each underlying *BlockStream* are both sent and received at very well-defined frame rate, so it should be relatively straightforward to increment a counter for each *BlockStream* every time a frame arrives based on the known framerate. This counter effectively becomes a simple clock that MAY be used to perform the required ordering.
    - The client MUST send this packet on each relevant *BlockStream* before attempting to send data for a single session across two or more *BlockStreams*. How long is it good for?
- Ratchet the key
- Announce credit balance
- Return Errors:
    - Unrecognized *SessionId* received
    - Unrecognized *SessionProtocolId*
    - No support for some App layer service
    - Data was lost for a specific *BlockStream*
- Retire an old *SessionId*
    - NOT NEEDED: Just re-use it when starting a new *SessionId*
- *ControlSession* sequence/ack numbers?
    - No, keep the *ControlSession* protocol simpler and lightweight in the common case. In the event of lost data on the *ControlSession*, the sender is expected to resend whatever is needed.

12.5 *EccFlow* Initialization

The below specifies the 3-way handshake required to initialize a new *EccFlow*.

12.5.1 *InitEccFlow*

The *InitEccFlow* message has the following statically-sized structure:

Member	Size

InitKey	32 octets
InitIV	12 octets
AuthTag	16 octets
EccFlowId	8 octets
StreamId	8 octets
SplitEccFlowKey	32 octets
TotalStreams	2 octets
ReplyIsoStreamWordRate	2 octets
ReplyIsoStreamPaymentCredits	2 octets

Client sends a set of *InitEccFlow* messages:

- The *InitEccFlow* MAY be sent within a new *IsoStream* with *μRouteByIsoStreamHeaders*
    - The *RouteId*
    - The client MUST terminate the route with the *InitEccFlow* Tag advertised by the destination
    - For each *InitEccFlow,* the client MUST provide enough *IsoStreamPaymentCredits* to cover:
        - The total cost to get the *InitEccFlow* stream delivered to the destination
        - The total cost to get the associated *AcceptEccFlow* stream delivered back to the client from the destination
- The client MUST create a reasonably large set of initial streams and find as many independent routes for them as is reasonably cost-effective relative to desired redundancy
- The client MUST set *ReplyIsoStreamWordRate* to the desired *IsoStreamWordRate* of the reply *AcceptEccFlow*
- The client MUST set *ReplyIsoStreamPaymentCredits* to an amount that the client has calculated would be at least enough *IsoStreamPaymentCredits* for the reply *AcceptEccFlow* to be sent via an *IsoStream* all the way back to the client on the reverse *RouteId*
- The client MUST create a cryptographically random *InitKey* for each stream and send it as the first part of the stream
- The client MUST choose a cryptographically random *InitIV* for each stream and send it as the first part of the stream
- The client MUST make a copy of *InitIV* and flip the MSB and call this *AcceptIV*
- The client MUST choose a cryptographically random 8 octet number, and use it as the *EccFlowId*.
- The client MUST choose a cryptographically random 8 octet number for each stream and use it as the *StreamId*.
- The client MUST create a cryptographically random 32 octet *SplitEccFlowKey* for each stream
    - The *SplitEccFlowKeys* MUST NOT be used for any encryption/authentication yet: These will be combined later to produce the *EccFlowKey*.
- The client MUST encrypt and provide the *AuthTag* of the rest of the message that follows the *InitKey* using AES-GCM with the *InitKey*.

    - The client MUST use *InitIV* as the IV
- The client SHOULD attempt to time sending the streams such that they arrive at the server as close in time as possible.

12.5.2 *AcceptEccFlow*

The *AcceptEccFlow* message has the following structure:

Member	Size
EccFlowId	8 octets
AcceptId	8 octets
StreamIdList	8 octets * *TotalStreams* (MAX 8 octets * 256)
AuthTagList	16 octets * *TotalStreams* (MAX 16 octets * 256)
AuthTag	16 octets

Server receives a subset of *InitEccFlow* messages (perhaps some are lost or corrupted along the way):

- If, after subtracting all the credits demanded by the server to process an *InitEccFlow* message, the number of credits would go negative, the server MUST fail the *InitIsoStream* message with *InsufficientCredits*
- Upon acceptance, the server MUST associate the provided *RouteId* with the *EccFlowId* using the same cache flushing rules as the switches along the route (specified in the *µRouteByIsoStreamHeaders* section)
- The server MUST decrypt and authenticate the rest of *InitEccFlow* with the provided *InitKey* and *InitIV*
- After waiting a number of seconds that will account for all possible network jitter, the server MUST assemble a reply *AcceptEccFlow* message
- The server MUST set the *EccFlowId* to match the one provided in the set of *InitEccFlow* messages received.
- The server MUST choose a cryptographically random 8 octet number, and use it as the *AcceptId*.
- The server MUST include within *StreamIdList* up to 256 *StreamIds* of valid *InitEccFlow* messages received with the same *EccFlowId*.
    - If there are more than 256 valid *InitEccFlow* messages received, the server MUST choose just 256 of them at random
- The server MUST take *InitIV*, flip the MSB, and call this *AcceptIV*
- The server MUST take *AcceptIV*, increment it by one, and call this *StreamIV*
- The server MUST include within *AuthTagList* a series of hash message authentication codes that each use the *InitKey* and *AcceptIV* corresponding to the *StreamIds* added to *StreamIdList*.
    - The ordered elements of *AuthTagList* MUST correspond to the ordered elements of *StreamIdList*

    - If the *StreamIdList* contains any fake random *StreamIds*, these MUST have a corresponding fake random *AuthTagList* entry
- The server MUST send this identical assembled *AcceptEccFlow* message within a *IsoStream* using *μRouteByCachedRouteReversed*
    - One *IsoStream* for each *RouteId* associated with the *EccFlowId* (to provide redundancy)
    - The *IsoStreamWordRate* MUST match the rate requested by the *StreamId*
    - The *IsoStreamPaymentCredits* MUST match the amount requested by the *StreamId*.
    - All the remaining credits from the *InitEccFlow* should be stored with the *EccFlow*, to be used as needed for future messages sent to the client
    - Each *AcceptEccFlow* MUST be sent without a header (the *AcceptEccFlow* type is assumed when it's the first *IsoStream* reply on the *RouteId*)
    - The server MUST encrypt the message with the *InitKey* corresponding to the *RouteId* using *StreamIV* and append the generated *AuthTag* to the end of the message

12.5.3 *ConfirmEccFlow*

The *ConfirmEccFlow* message has the following structure:

Member	**Size**
AcceptId	8 octets
BadStreamIdList	8 octets * *TotalStreams* (MAX 8 octets * 256)
KeyId	16 octets
AuthTag	16 octets

The client receives a subset of the *AcceptEccFlow* messages (perhaps some are lost on the way):

- The client SHOULD drop additional copies of redundant *AcceptEccFlow* messages.
- The client MUST authenticate the set of returned *StreamIds*
- If the *AcceptEccFlow* message contains **all** of the authentic *StreamIds*, this means **none** of the streams were compromised or **all** of the streams were compromised, either way the client SHOULD reply right away.
- If the *AcceptEccFlow* message contains **some** of the authentic *StreamIds*, the client MUST wait enough time to ensure that replies on the slowest route can arrive.
    - Once enough time has passed, the client MUST choose **only** the *AcceptEccFlow* message with the highest number of authentic *StreamIds*
- The client MUST separate the *StreamIds* into a list of authenticated *StreamIds* and a list of inauthentic *StreamIds*.
- For all the authenticated *StreamIds* the client MUST XOR together the associated *SplitEccFlowKeys* to create the *EccFlowKey* used for all further encryption and authentication tagging of the *EccFlow*.
- The client MUST now assemble a *ConfirmEccFlow* message

- The client MUST set this message's *AcceptId* to the *AcceptId* within the chosen *AcceptEccFlow* message
- The client MUST include within *BadStreamIdList* each of the inauthentic *StreamIds* within the chosen *AcceptEccFlow* message.
- The client MUST fill the rest of *BadStreamIdList* with randomly generated (inauthentic) *StreamIds*
- The client MUST create a randomly generated *KeyId* to be used to refer to the *EccFlowKey*
- The client MUST set *AuthTag* to a hash message authentication code (HMAC) using AES-GCM with *EccFlowKey*
    - The client MUST use *AcceptId* as the IV
- The client MUST send this identical assembled *ConfirmEccFlow* message within an *IsoStream* using *μRouteByCachedRoute*
    - One *IsoStream* for each *RouteId* that is to comprise a *EccFlowSegment* (selecting the desired/required amount of redundancy for the application)
- Following the *ConfirmEccFlow*, all future messages on the streams MUST be encrypted and HMAC tagged with AES-GCM using the *EccFlowKey*
- Both the client and the server MUST keep *EccFlowKey* a secret
    - TODO: Consider adding a double ratchet over time for better safety

12.6 Segmentation and Forward Error Correction Coding

The IsoGrid is designed to support sending portions of the data across the mesh over separate routes. With this in mind, it's good to have more connections to allow for redundancy. But if a node were to segment the data evenly, that just increases the likelihood that a link failure will cause data loss. Instead, a segmented *EccFlow* uses a Forward Error Correcting Code (FEC code, or just ECC), and with just a bit of overhead, the reassembled *EccFlow* can be tolerant of link failures. Desired redundancy can be targeted between 0-32 routes.
The fact that corrupted words are sent as *LostWord* allows them to be counted as Erasures for the FEC algorithm, which provides even more (efficient) redundancy.

The concept of a *BlockStream* is used to abstract the segmentation, ECC, flow rate, and safety aspects of an *EccFlow*. One or more *BlockStreams* can be created to transport the data within the *EccFlow*. Conceptually, an *OutputBlockStream* on the sending side is paired with an *InputBlockStream* on the receiving side.
The end-to-end *EccFlow* data packets at this layer look like this:

1. Data packet frames from the *EccFlowFramer* go in an *OutputBlockStream*
2. They are encrypted, auth-tagged, FEC-coded, and segmented
3. They are sent across the IsoGrid network via *IsoStreams*
4. The enter an *InputBlockStream*, where they are re-assembled, FEC-decoded, decrypted, and authenticated
5. Data packets leave the *InputBlockStream* and are handled by the *EccFlowFramer*

12.6.1 *InitEccFlowSegment*

This message type is implied when an *InitIsoStream* arrives using a *RouteId* that previously handled an *AcceptEccFlow* message or a *ConfirmEccFlow* message.
The client MAY include an implicit *InitEccFlowSegment* message immediately following a *ConfirmEccFlow* message within a single *IsoStream*. *InitEccFlowSegment* MUST only be sent within an *IsoStream* (not within a *μPkt*).

The *InitEccFlowSegment* message has the following statically-sized structure:

Member	**Size**
KeyId	8 octets (This field MUST be omitted if this message directly follows a *ConfirmEccFlow* message within the *IsoStream*)
BlockStreamId / HighIV	4 octets
LowIV	8 octets
SegmentationType	8 octets
SegmentationDetails	56 octets
AuthTag	16 octets

BlockStreamId of 0 is reserved (invalid) so that the associated IV can be reserved for use in the *ConfirmEccFlow*.
A server→client *BlockStream* is invalid if the following is true:

((*BlockStreamId* - (0xFFFFFFFF & *AcceptId*)) MOD 0xFFFFFFFF) <= 0x7FFFFFFF

A client→server *BlockStream* is invalid if the following is true:

((*BlockStreamId* - (0xFFFFFFFF & *AcceptId*)) MOD 0xFFFFFFFF) > 0x7FFFFFFF

When the sender needs to create an additional *BlockStream*:

- The sender MUST initially set *KeyId* equal to the *AcceptId* of the chosen *AcceptEccFlow* message
- The sender MUST choose a *SegmentationType* that is supported by the receiver
- The sender MUST set *SegmentationDetails* and *LowIV* as per the requirements of the *SegmentationType*
- The sender MUST use *HighIV* and *LowIV* together as a single 12-octet *SegmentIV*
- The sender MUST ensure that the *SegmentIV* and *EccFlowKey* referred to by *KeyId* forms a unique pair
- The sender MUST encrypt the *SegmentationType* and *SegmentationDetails* members and set the *AuthTag* with AES-GCM using *SegmentIV* and the *EccFlowKey* referred to by *KeyId*

Receiver Requirements:

- If the receiver does NOT have a *BlockStream* with the provided *BlockStreamId*, a new one MUST be created with this segment added
- If the receiver already has a *BlockStream* with the provided *BlockStreamId*, this segment MUST be added to it

- The receiver MUST interpret *HighIV* and *LowIV* together as a single 12-octet *SegmentIV*
- The receiver MUST decrypt and authenticate the *SegmentationType* and *SegmentationDetails* members with AES-GCM using *SegmentIV*, *AuthTag*, and the *EccFlowKey* referred to by *KeyId*
- The receiver MUST interpret *SegmentationDetails* as per the requirements of the *SegmentationType*

12.6.2 *ReedSolomon255p223*

Inherits: *None*
SegmentationType: 0x 4E 01 8E 57

All IsoGrid nodes MUST support this Reed-Solomon (255, 223) ECC segmentation algorithm for *InitEccFlowSegment*.

The *ReedSolomon255p223 SegmentationType* has the following statically-sized structure:

Member	**Size**
FirstBlockCountOfSegment	8 octets
BlockStreamAuthBlockSize	3 bits
BlockStreamSegmentStreak	3 bits
Padding2A	2 bits (MUST be zero)
BlockStreamSegmentId	1 octet
BlockStreamDataSegmentCount	1 octet
Padding2B	5 octet (MUST be zero)
Padding3	8 octet (MUST be zero)
Padding4	8 octet (MUST be zero)
Padding5	8 octet (MUST be zero)
Padding6	8 octet (MUST be zero)
Padding7	8 octet (MUST be zero)

Sender Requirements:

- The sender MUST choose a random 8 octet number with the least significant 8 bits cleared and store it with the *BlockStream* as *FirstLowIV*
- The sender MUST interpret the *BlockStreamId*/*HighIV* and *FirstLowIV* together as *FirstSegmentIV*
    - The sender MAY make a copy of *FirstSegmentIV* and call it *CurrentSegmentIV*
    - The sender MUST make a copy of *FirstSegmentIV*, and call it *CurrentAuthBlockIV*
- The sender MUST set *BlockStreamId* to match the ID of the *BlockStream* to which this segment will contribute
- The sender MUST set *FirstBlockCountOfSegment* to the *BlockCount* at which this segment begins
    - The *BlockStream* MUST start with a *BlockCount* member equal to 0
- The sender MUST set *BlockStreamAuthBlockSize* to the *AuthBlockSize* of the *BlockStream*
    - Valid *AuthBlockSize* values are:

Value	Bytes per *AuthBlock*
0	64
1	512
2	4096
3	32768
4	262144
5	2097152
6	16777216
7	134217728

- The sender MUST set *BlockStreamSegmentId* uniquely for each segment of a given *BlockStream*
    - Data segments are numbered between 1 and *BlockStreamDataSegmentCount* (inclusive) and parity segments are numbered between 224 and 255 (inclusive).
- The sender MUST set *LowIV* to a previously unused value
    - The sender MAY add *CurrentSegmentIV* together with *BlockStreamSegmentId* to produce a previously unused value (decrementing *CurrentSegmentIV* by 256 after the set of *InitEccFlowSegment* messages have been sent)
- If some of the *IsoStreams* run at a higher rate than other *IsoStreams*, then the higher rate segments MUST have *BlockStreamSegmentStreak* set to a higher value.
    - A non-zero value means that the *IsoStream* backing the InitEccFlowSegment provides multiple consecutive octets per round of Reed-Solomon block
    - Valid *BlockStreamSegmentStreak* values are:

Value	Reed-Solomon segments per *IsoStream*
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

- The sender MUST set *BlockStreamDataSegmentCount* to the total number of data segments that the sender is providing for the *BlockStream*
    - This applies only to blocks numbered *FirstBlockCountOfSegment* or higher (until later overridden by an *InitEccFlowSegment* with a greater than or equal *FirstBlockCountOfSegment*)
    - *BlockStreamDataSegmentCount* MUST be a number between 1 and 223 (inclusive)
- The sender MUST create virtual (non-transmitted) segments for all the *BlockStreamSegmentIds* numbered both greater than *BlockStreamDataSegmentCount* and less than 224; the sender MUST assume these segments to be a string of valid zeros for the purposes of the Reed-Solomon encoder

The sender MAY want to increase the *DataSegmentCount* in order to increase the *BlockStream* data rate; in order to do this: The sender MAY send one or more new *InitEccFlowSegment*

streams with the new (higher) *DataSegmentCount*. The sender SHOULD consider sending more than one or two of these on multiple independent routes for redundancy.

TODO: Consider if there needs to be a mandatory in-band *EccFlow* packet that can decrease the *BlockStream* rate or if it's easier to just create a new *BlockStream* and simultaneously cease the old one.

Sender streaming requirements:

- The *BlockCount* of the *BlockStream* is incremented by one immediately after the last octet of an *AuthTag* is processed
- The sender MUST encrypt and *AuthTag* each *AuthBlock* of *OutputBlockStream* plaintext input stream with AES-GCM using *CurrentAuthBlockIV*, and the *EccFlowKey* referred to by *KeyId* in order to produce the ciphertext output stream
    - The sender MUST append the 12-octet *AuthTag* following each *AuthBlock* of ciphertext output stream
    - The sender MUST increment *CurrentAuthBlockIV* by one after each encrypted *AuthBlock*
- The sender MUST take *DataSegmentCount* octets of the ciphertext output stream and (223-*DataSegmentCount*) octets of zeros and feed them into a Reed-Solomon (255,223) encoder
    - Each round of the encoder outputs 255 octets which MUST be segmented by feeding one of these 255 octets into each segment *IsoStream* based on *BlockStreamSegmentId*

Receiver requirements when an *InitEccFlowSegment* is received:

- If the receiver does NOT have a *BlockStream* with the provided *BlockStreamId*, a new one MUST be created with this segment added
    - The created *BlockStream* MUST have a *BlockCount* member that starts at 0
    - The created *BlockStream* MUST have *AuthBlockSize* set to *BlockStreamAuthBlockSize*
    - The created *BlockStream* MUST have *DataSegmentCount* set to *BlockStreamDataSegmentCount*
    - The created *BlockStream* MUST have a 12 octet member named *FirstSegmentIV* set to the combined *BlockStreamId*/*HighIV* and *LowIV*
    - The created *BlockStream* MUST have a 12 octet member named *CurrentAuthBlockIV* initially set to *FirstSegmentIV*
- If the receiver already has a *BlockStream* with the provided *BlockStreamId*, this segment MUST be added to it
    - The receiver MUST validate that *BlockStreamAuthBlockSize* is equal to the *AuthBlockSize* of the *BlockStream*, and MUST drop the *InitEccFlowSegment* if it is invalid
    - The receiver MUST drop the *InitEccFlowSegment* if the *BlockStreamSegmentId* is both less than 224 **and** greater than *BlockStreamDataSegmentCount*

- The receiver MUST begin adding this segment to the *BlockStream* input based on its *BlockStreamSegmentId* after the *BlockCount* is equal to *FirstBlockCountOfSegment*
    - Data segments are numbered 1 to *BlockStreamDataSegmentCount* and parity segments are numbered 224 to 255.
- The receiver MUST set the *DataSegmentCount* member of the *BlockStream* to *BlockStreamDataSegmentCount* after the *BlockCount* is greater than or equal *FirstBlockCountOfSegment*
- The receiver MUST ignore segments with *BlockStreamSegmentIds* numbered both greater than *BlockStreamDataSegmentCount* and less than 224; the receiver MUST assume these segments to be a string of valid zeros (rather than erasures) for the purposes of the Reed-Solomon decoder
- The receiver MUST take *BlockStreamSegmentStreak* octet at a time from each *BlockCount*-aligned segment and feed them into a Reed-Solomon (255,223) decoder: The *DataSegmentCount* octets of decoded data MUST be appended to the ciphertext output stream
    - Any completely missing input segments MUST be treated as erasures
    - Any *LostWord* or otherwise corrupted input MUST be treated as erasures
- Following each *AuthBlock* of ciphertext output stream is a 12-octet *AuthTag*
    - The receiver MUST decrypt and authenticate each *AuthBlock* of ciphertext output with AES-GCM using *CurrentAuthBlockIV*, *AuthTag*, and the *EccFlowKey* referred to by *KeyId*
    - The receiver MUST increment *CurrentAuthBlockIV* by one after each decrypted *AuthBlock*

12.7 Mitigation for Denial of service attacks against EccFlow

In order to perform a 100% successful DoS, an attacker has to have control over all independent routes between the *EccFlow* endpoints. If an attacker only has control over most of the routes, then the attack is probabilistic, and costs significantly more than the client has to pay. This makes it economically infeasible to continue to deny service since the client can just continue to try again.

12.8 LinkTunnel Pattern

Tunneling an IsoGrid link across the IsoGrid between distant nodes can be efficient, with low overhead. This could make for quicker, easier, and more reliable long distance *IsoStreams*. This effectively reduces the hop-count on an *IsoStream* that traverses the tunnel.
It seems relatively straightforward to layer this on top of an *EccFlowIsochSession*.

12.9 AnchorForMobile Pattern

Another use for a link tunnel is to anchor a persistent link from a stationary node to a mobile node.

It doesn't seem possible to have scalable route determination with a mesh of switches that relies heavily on dynamic links. As such, long-distance data transport will mostly occur over switches with reasonably persistent links, or at least predictable links (like satellites). A cell tower or a Wi-Fi router are typical examples of switches with highly transient links, and as such, endpoints with highly transient links are generally just called mobile nodes. Fortunately, relying on highly transient links for only a few hops can be scalable. A simple way to achieve scalability is to allow for layers of indirection, where data flowing to and from a mobile node is routed through one or more persistent link tunnel(s) with stationary anchor nodes.
Each anchor node MAY:

1. Answer the question: "Where is the mobile endpoint?"
    a. Which IsoGrid nodes is the mobile endpoint near?
    b. What are the best routes to get to the mobile endpoint?
2. Provide credit payments as needed to sustain the mobile node's outbound streams
3. Provide a link tunnel to, from, or through the mobile node

In this *AnchorForMobile* pattern, the anchor node and the mobile node maintain an *EccFlow* between each other and use an *EccFlowIsochSession* as the link layer underlying an IsoGrid network link. This is only a suggested pattern, and the specific protocol does not need to be universally supported: The only requirement is that the mobile and anchor nodes MUST agree to use the same protocol.

12.9.1 Internet of Things Discussion

IoT devices:

- Are cheap
- Are relatively low bitrate
- Have limited power budget
- Might be mobile or stationary
- Aren't good isochronous switches, so will typically just be endpoints

Each IoT endpoint node will establish links with the X closest compatible IsoGrid node(s). Mobile IoT nodes should have anchor nodes setup if they want to accept incoming requests, just like regular mobile nodes. Stationary IoT nodes will setup persistent link(s) to the IsoGrid node(s) they are closest to.

13 Path Determination for Stationary Nodes

TODO: This section is a work in progress and may just contain raw notes!

This section covers the standardized *LinkAdvertisement* protocol that facilitates node and link advertisements for stationary switches and endpoints. Path determination using mobile switches on a mesh network might not be scalable, so it is left as an exercise to the reader ☺. Path determination for mobile endpoints is covered via the *AnchorForMobile* pattern, where the paths to a mobile endpoint go through one or more stationary anchor nodes.

When a new link comes online, each side MUST send *LinkLayerNodeAdvertisement*. Upon receipt of *LinkLayerNodeAdvertisement*, the node with the higher NodeID MUST initiate an *EccFlow* to the other node and use it as the transport for all the following packets. Then, the initiator side MUST send *LinkAdvertisement* containing each of its local links.

When a node sees a *LinkAdvertisement* to a remote node it doesn't already know about, and it knows how to reach it for a reasonable cost, it SHOULD initiate an *EccFlow* and then send *LinkAdvertisement* for each of its local links.

When a node decides to make a change to one or more links, the node SHOULD send the updated *LinkAdvertisement* to each remote node that it knows how to reach for a reasonable cost. This SHOULD be balanced with the expected frequency of changes: If the change frequency is expected to be high, then the *LinkAdvertisement* updates MAY be sent to fewer nodes.

When a node decides to change its *NodeAdvertisement*, it MUST send a new *LinkLayerNodeAdvertisement* across all its links. The node SHOULD continue to honor the previously advertised service declarations for 24 hours following the release of a new *LinkLayerNodeAdvertisement*.

NodeAdvertisement:

- NodeID (16 octets)
- 3D Cartesian Geohash (24 octets)
- Optional Text location (64 octets)
- µPkt Type Declaration Count (4 octets)
- µPkt Type Declarations (16 + 4 + 12 octets each [TypeID + Cost + TypeSpecificData])
- Route Type Declaration Count (4 octets)
- Route Type Declarations (16 + 4 + 12 octets each [TypeID + Cost + TypeSpecificData])
- Service Declaration Count (4 octets)
- Service Declarations: (16 + 16 + 4 + 12 octets each [ServiceID + Tag + Cost + ServiceSpecificData])
    - *LinkAdvertisement*
    - *InitEccFlow*
    - *InitEccFlowSegment*
- Name/Value Pair Count (4 octets)
- Name/Value Pairs (16 octets + 32 octets each)

LinkLayerNodeAdvertisement:
Same contents as *NodeAdvertisement*. Sent via Link Layer protocol agreement (0xCCCC tag in prototype). No credit cost because neighbors can be friendly (and avoid spamming each other).

LinkAdvertisement:
NodeID (16 octets)

LinkCount (8 octets)
LinkN:

- Tag (16 octets)
- Lowest Supported Rate (1 octet)
- Highest Supported Rate (1 octet)
- Review the Slot Isochronous Standard for other requirements
- Maximum *IsoStreamWordCount* (2 octets)
- Minimum *InBandSignal* period, expressed just like in *IsoStreamSetInBandSignal* (1 octet)
- Count of switching buffer size (in words), expressed as an exponent (1 octet)
- Minimum supported WordCount(*IsoStreamRoute*) - WordCount(*InitIsoStream*) (1 octet)
- Worst Case Latency (4 octets)
- Maximum credit transfer per word expressed as binary32 floating-point (4 octets)
- Link cost expressed as binary32 floating-point (4 octets)
- Exchange Rate expressed as binary32 floating-point (4 octets)
- Name/Value Pair Count (4 octets)
- Name/Value Pairs (16 octets + 32 octets each)

NodeAdvertisement of DestinationN (variable size)

Later Maybe add:
Beginning valid time (8 octets)
Expiration time (4 octets)

13.13D Cartesian Geohash

The *NodeAdvertisement* contains the location of the node, represented as a 3D Cartesian Geohash

Origin: Center of mass of the earth
Spin: Exactly equal to the spin of the earth.
1st bit: Z - Positive axis toward north pole (0 := southern hemisphere, 1 := northern hemisphere)
2nd bit: X - Positive axis towards equator at 0deg longitude
3rd bit: Y - Positive axis towards equator at 90deg longitude
Thereafter, every third bit refers to the relevant axis.
After the first digit, if the second axis digit is 0, the bounds of the space is from 0 to 2m^23=8,388,608m.
After the first digit, the number of repeated 1 digits (n) for a given axis defines the inner bounds as 2m^(n+21) and the outer bounds as 2m^(n+22).
Thereafter, the bounds are bisected like a normal Geohash.
75 bits gives 1m precision anywhere within a bounding box 2m^24 on a side centered on the earth. This covers the surface of the earth and most low earth orbital satellites. Another 30 bits (105) brings it to millimeter precision. Another 30 bits (135) brings it to micron precision. Another 30 bits (165) brings it to nanometer precision, which seems like enough. For locations at geostationary orbit, 2-4 additional bits are needed to expand the boundary of the space. 1-3

more bits would be needed to recover the same precision in that expanded coordinate boundary.

13.2 Methodically Verified Routes

Routing information that has been built up hop by hop via *EccFlow* to each node is called a verified route. It's called 'verified' because *EccFlow* uses multiple unique *IsoStream* routes and secret splitting to be reasonably certain that the node and link information it receives is authentic. The only uncertainty would be the possibility of widespread collusion among neighboring network participants.

13.3 Just-In-Time Routes

Building up verified routes to a far-away node could be a time-consuming process. One option to avoid this is to use a Just-In-Time route determination algorithm as described in this section.

When a node purchases from any remote node a description of a route between two remote nodes, that route is not verified because the level of collusion required in order to successfully perform a man-in-the-middle attack is likely smaller. Additionally, colluding nodes could agree to send routes through higher-cost (more profitable) conspirators.
This is called a "Just-In-Time" route, because it is possible to build up routes to far away nodes with a relatively few round trips just prior to initiating communication.

A partial strategy to mitigate the increased risks of using Just-In-Time routes might be to:

1. Use many of them with an *EccFlow* layered on top
2. Prioritize finding unique routes (instead of inexpensiveness or low latency) for the *InitEccFlow* handshake.
3. Switch to inexpensive routes after the *EccFlow* is established
4. Purchase many routes from multiple nodes at random

13.3.1 Session Protocol: GetBestRoutes

Nodes MAY use the *GetBestRoutes EccFlowSessionProtocol* to ask a remote node for the best route(s) between a source and a destination node.
GetBestRoutes is an *EccFlowSessionProtocol* that MAY be initiated via a *StartSession* packet on the *ControlSession*. If the recipient understands this *SessionProtocol* it MUST acknowledge the message with a *MultiRouteContainer*. If the recipient does not find any routes, it MUST respond with an empty *MultiRouteContainer*. If the recipient does not understand the request, it MUST respond with *UnsupportedSessionProtocol*.
The recipient SHOULD attempt to find the best routes between the provided source and destination based on the criteria in the *GetBestRoutes* message.
The following packet data MUST be sent on the session specified by the *SessionIdForRequest* in the associated *StartSession* packet.

Field Name	Size	Description

RequestedRouteCount	1 octet	The number of routes to attempt to find
SourceNodeID	16 octets	The NodeID of the start of the route
DestinationNodeID	16 octets	The NodeID of the end of the route
DestinationGeohash	24 octets	The 3D Cartesian Geohash of the end of the route
PrioritizeUniqueRoutes	4bit	Maximize the uniqueness of the routes after the first one
PrioritizePrice	4bit	Minimize Price
PrioritizeExchangeRate	4bit	Maximize ExchangeRate
PrioritizeLatency	4bit	Minimize Latency
PrioritizeCreditTransfer	4bit	Maximize Credit Transfer
PrioritizeIsoStreamHeader	4bit	Minimize IsoStreamHeader length
PrioritizePrecision	4bit	Maximize supported IsoConnInitX
PrioritizeRateRange	4bit	Maximize Rate Range
PrioritizeWordCount	4bit	Maximize Word Count
PrioritizeInBandSignalPeriod	4bit	Minimize *InBandSignal* period
PrioritizeHopCount	4bit	Minimize hop count
MaxPriceLimit	4 octets	
MaxExchangeRateLimit	4 octets	
MaxLatencyLimit	4 octets	
MinCreditTransferLimit	4 octets	
MaxIsoStreamHeaderLimit	4 octets	
MinPrecisionLimit	1 octet	
RangeRateLimit	2 octets	
MinWordCountLimit	2 octets	
MaxInBandSignalPeriodLimit	2 octets	
MaxHopCountLimit	2 octets	

This message has four cost components: A base price, a price per returned complete route, a price per partial route, and a multiplier that equals (1 + the highest PrioritizeX value). No credits are exchanged if the recipient responds with *UnsupportedSessionProtocol*. The multiplier only applies to the number of returned routes.
If the recipient has a verified route to the DestinationNodeID, it MUST set IsSingleDestination := 1.
If the recipient doesn't have a verified route that goes all the way to the DestinationNodeID, then the recipient MUST set IsSingleDestination := 0, and provide its verified routes to the unique nodes closest to the destination node.

The response *MultiRouteContainer* packet structure MUST be sent on the session specified by the *SessionIdForResponse* in the associated *StartSession* packet.

13.3.2 MultiRouteContainer

MultiRouteContainer is a packet structure used to transmit a list of routes.

Field Name	Size	Description
SourceNodeID	16 octets	
IsSingleDestination	1bit	1: Only a single destination node is included 0: Each route leads to a different destination
DestinationNodeID	16 octets	(Included only if IsSingleDestination == 1)
DestinationGeoHash	24 octets	(Included only if IsSingleDestination == 1)
RouteCount	2 octets	The number of routes in the container
Routes	Variable	The list of routes:
DestinationNodeID	16 octets	(Included only if IsSingleDestination == 0)
DestinationGeoHash	24 octets	(Included only if IsSingleDestination == 0)
RouteCost	4 octets	The total cost of the route
RouteExchangeRate	4 octets	The exchange rate of the route
RouteLatency	4 octets	The total latency of the route
RouteCreditTransferLimit	4 octets	The most credits per word that will be transferred by the route (in destination credits)
RouteRateRange	2 octets	The range of rates available in the route
RouteWordCountLimit	2 octets	The maximum word count limit of the route
RouteInBandSignalPeriodLimit	2 octets	The minimum InBandSignal period of the route
RouteHops	4 octets	The length of the route in hops
RouteIsoStreamHeaderLength	8 octets	The length of the *IsoStreamHeader* needed to establish an *IsoStream*
IsoStreamHeader	Variable	The IsoStreamHeader used to establish a stream between the source node and the destination node.

13.4 Discussion on Scalable Verified Route Determination

The trivial plan of methodically sending link advertisements to all nodes on the IsoGrid is likely to work for some time (while IsoGrid node counts are low). Each node is likely to be able to keep up with 100,000 or more nodes with relatively persistent links. If the number of nodes on the IsoGrid rises high, or the cost of sending advertisements rises too high, or the frequency with which links are changed is too high, nodes will have to be smarter about where they spend their resources distributing and keeping up with verified routing data. One way nodes could be smarter about verified routing data is described here.

It's important that this solution avoids any negative socio-economic effects.

The 'home' node SHOULD use the following algorithm to create a spider-web like grid of verified routes through the rest of the IsoGrid:

- Start with a small distance limit from the 'home' node and proactively distribute its own link updates to all nodes that are within that limit and increase that limit until:
    - The maximum number of nodes is reached

    - The maximum budget is reached
- Use a 'trailblazing' algorithm like:
    - Define a budget for subscribing to link updates, once the budget is exceeded, stop 'trailblazing'
    - Start with the 64 closest nodes from the 'home' node that haven't had their links verified
    - Each of these 64 'trailblazers' proceed down hops somewhat round-robin style (slow down the 'trailblazers' that have X more hops than the others)
    - Follow the outbound links at each hop by preferring links using the following priority rules:
        1. Assume that a node that announces more than 512 links is lying or defective
        2. Prefer links that don't lead away by more than half the current distance from the starting node
        3. Prefer links that lead to unmapped nodes
        4. Prefer links that leads the furthest distance away from the starting node
        5. If there's only one non-loopback link left, take it
    - If all links lead to mapped nodes start over with the next unmapped node closest to the home node.
    - If a path is a dead-end, back up and choose a different outbound link
        - If backing up takes you back to one of the initial nodes, start a new trail by choosing the next closest node to the 'home' node that hasn't had its links mapped
- Use a 'trailconnector' algorithm like:
    - For every 'trailblazer', there is one 'trailconnector' that attempts to build up an independently verified mapped trail from the source 'connected' node on the associated 'trailblazer' to a separate trail.
    - The 'connector' destination target is the nearest trail node that is Y meters further from the 'home' node than the source 'connected' node.
    - The first 'connected' source node is the first node of each trail
    - Once connected to the nearest trail, the destination node becomes the next 'connected' source node for the 'connector' and Y is doubled
- Let subscriptions to link updates from dead-end nodes (those that aren't part of a trail) expire
- Whenever establishing an *EccFlow* to a node takes more than one hop of *GetBestRoutes* requests to reach the destination node, the 'home' node MAY consider using one or more 'trailblazers' to setup independently verified trails
- When something at the application layer actively uses an *EccFlow* to a node, the trail(s) used are marked with an updated 'last used' timestamp.
- Subscriptions to link updates will not be renewed for nodes along a trail that isn't used for some time.

14 Bootstrapping Discussion

In the early stages of implementing the IsoGrid, there won't be any widely-deployed native-IsoGrid services. The only practical use of the IsoGrid during the early stages would be as a gateway to the existing Internet: The IsoGrid Protocol Stack will need to act as a reasonably inexpensive alternative to traditional Internet Service Providers. Two of the biggest costs of traditional Internet Service Providers are: 1) infrastructure for connectivity over the last-mile to customers, and 2) customer acquisition costs. With a local IsoGrid, last-mile infrastructure is provided by the customer. Customer acquisition costs might be significantly reduced because the IsoGrid is likely to spread from neighbor to neighbor by word-of-mouth, precisely for the purpose of getting cheaper internet access. The early adopters are likely to be willing to start the IsoGrid before financial benefits are clear, due to being dissatisfied with their existing ISP options (or lack thereof). Once a small IsoGrid is started, the connected participants have a strong financial incentive to connect up more of their neighbors.
Once a significant portion of the population of developed countries start using the IsoGrid for Internet access, the hardware and software costs will get much lower. Having a mesh topology allows for implementations with very simple initial setup and maintenance. When these begin to appear, it is our belief that the IsoGrid will spread to developing countries, providing cheap, scalable, and dependable connectivity to the world.

14.1 IpVpn

TODO: Specify this in more detail.

CreateIpVpn is a service that is run on top of an *EccFlow*. The client is a node on the IsoGrid, the server node is dual-homed with both IsoGrid and a connection to an IP network (even behind a NAT). When executed, an *EccFlowAsyncSession* is created that will be used by the client to send packets via the server node directly to the target IP network (which may be the IP Internet). A second *EccFlowAsyncSession* is also created that is used by the server to route incoming packets back to the client node. The *EccFlow* layer is responsible for maintaining sufficient credits to be able to pay for the data heading toward the client. *IpVpn* uses a simple Point to Point Protocol (PPP). Both the service and client nodes are expected to layer a TCP/IP stack on top of the PPP link.

14.2 Network Management

Initially, the early adopters of the IsoGrid will have to manage their own networking equipment and handle credit settlement between neighbors. However, over time, we might expect to see the emergence of companies offering "Network Management" services. These netMan services are likely to attempt various business models:

- Consumer leases equipment, netMan service provides flat-rate internet service
- Consumer owns equipment, netMan service handles software management and takes a cut on credit exchanges between neighbors
- Open source

- Etc.

Many of these models might rely on consumer brand loyalty: If a brand of netMan becomes known for good service for the value, it's likely to gain customers from competitors. A brand with security holes is likely to suffer. Flat rate might die out as policing a commons can get expensive.

15 Undefined Higher-Level Services and Protocols

There are a number of services and protocols defined at a higher level that readers might find interesting. These services and protocols are intentionally left out of the global IsoGrid Protocol specification in order to allow the overall system more flexibility over time.

Distributed Naming

The problem of finding services by name exists in the IsoGrid just like the IP Internet. The problem appears to be substantially identical in both network designs. The only difference is that the IsoGrid can require micro-payments to handle name lookups: Solving one of the big Denial of Service attack vectors on the IP Internet today, and perhaps making it possible to design distributed protocols for name lookup that don't rely on blockchain technology. Not that blockchain technology is bad, just making the point that the designers of IsoGrid aren't doing all the work of designing the IsoGrid merely to sell you on a technology that relies exclusively on a blockchain.

Low Latency Game Streaming

With low latency access to a distributed network, it's possible to implement game streaming services; where folks share (or rent) game console access from your neighbors. Generalizing, this may evolve into distributed compute.

Distributed Storage

People will eventually be able to store data in distributed storage networks, with connectivity provided by The IsoGrid. Data should be both more secure (encrypted at the source) and reliable (distributed very widely, with excellent erasure FEC codes).

Alarm Systems

The ability of neighbors (and perhaps even neighborhoods) to link up their alarm systems can reduce the cost and/or increase the effectiveness of the systems. Additionally, it isn't necessary that the system be centralized, and smaller systems are less of a target for hackers.

IsoGrid Internet Service Providers

The success or failure of IsoGrid basically hinges on whether it's cheaper or less of a hassle to have an IsoGrid-based ISP instead of a Traditional ISP, like Comcast or Verizon.

IsoGrid-based ISPs are referred to as Minimal ISPs (or minISPs) and are very similar to Traditional ISPs except that they don't run links all the way to the customer, instead they rely on a local *IsoGrid* to provide the last-mile connectivity to and from their customers. Only customers that have at least one or (hopefully) more connections to an *IsoGrid* can use a minISP. A minISP must follow all the rules and regulations that apply to ISPs.

EccFlow to Data-Center based personal VPN

A client that has created many *IpVpn* sessions to various dual-homed server nodes could link up with a remote node on the IP Internet and then layer another *EccFlow* on top of all these links. Another *IpVpn* service can then be layered on top of that, allowing the client node to access the IP Internet via the remote VPN node. This acts like a multi-path redundant VPN (as long as the remote VPN itself has good uptime. This may not be necessary if instead the client can just hop from one *IpVpn* to another without affecting applications layered on top.

IP Transit

Tunneling other network protocols on top of *EccFlow* should be efficient, with low overhead.

Large Scale, High-Precision Timing

There are a number of large scale projects that aren't practical with the IP Internet, but could be undertaken if the IsoGrid were massively successful. One property of a full-scale IsoGrid is a very precise synchronized time source. With excellent, synchronized clocks it's potentially possible to build:

1. Cheap differential GPS, everywhere
2. Good-signal, Terrestrial GPS in urban areas
3. Indoor GPS
4. Distributed deep-space antenna arrays