The IsoGrid: Scalable Mesh Networking for a Better World

By Travis.Martin@isogrid.org

Version 0.235

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. To support isochronous streams across the entire network, the framerate of every link is a power of 2 frequency relative to TCG time, this opens the door to many new scenarios that require precise relative timekeeping. Micro-transfers of *Energy*, which are made 'by simple agreement' between each neighbor along a route, are used to cover 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. The hash-based locating mechanism gives a convenient solution to distributed data storage. 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. Today, 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 must decode the entire header of a packet and then do a lookup in 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 need 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 is 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-transfers of *Energy* 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, Internet Protocol (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 Overview

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
- Defines how the network routes *μPkts* across the IsoGrid
- Defines how nodes send *Energy* 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 *Energy* 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 arbitrary time period
- Forward Error Correction coding, safety, and multi-path segmentation
- Defines how nodes use the network transport to distribute routing information

Layer 5: Application Layer

- Globally-Scalable mesh mapping and routing using HashMatchLogMap (*HMLM*)
- Distributed content addressable storage (CAS)
- Distributed self-certified naming using *GetNodeInfoFromLocatorHash*
- 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. DNS BGP	CAS *GetNodeInfoFromLocatorHash* *HMLM*
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 much *Energy* to send. 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. IsoGrid byte order is Little-Endian, ref COHEN; also the LSB is numbered 0, so: Little-Endian follows logically and naturally.

5.3 *Energy*

To avoid a tragedy of the commons, the IsoGrid allows for exchanging *Energy* between neighbor nodes to cover the data sent or processed. This allows tiny transfers to be made among neighbors instead of having to have a centralized transfer 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 require *Energy* for use of those resources. Settlement SHOULD be by simple agreement (there is no required third party).
Nodes MAY require different amounts of *Energy* for different outbound links.

5.4 IsoGrid Secondary Limitations

No network is without limits. The design of a protocol standard necessitates making tradeoffs 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
- Highly mobile nodes that want to offer isochronous data transit services might have a harder time competing against stationary nodes (because route tracking across transient links might not be scalable)
- 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-word-rate *IsoStreams* 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) could take a (longer latency) three-way handshake (and an *Energy* amount) 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 IsoGrid 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 or Nexus	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 switches
Switch	A device with multiple links that can route data using the IsoStream Protocol. Generally acts as an automaton by direction from a nearby Nexus Node.
Word	The atomic unit of transmission across the network. 128 bits of data, plus 1 additional bit to declare the word valid or invalid.
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 switch on the other end of the link.
Output Slot	A slot on an output link. Has a matching input slot on the switch on the other end of the link.
Energy	Units of approximately 10^-15 watt-years. Used for fairness handling across the *IsoGrid*.
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 switches that comprise The IsoGrid
µPkt	Micro-packet, a small (8 word/128-octet) block of data that flows asynchronously from a source to a destination across the IsoGrid. Contains dynamic data portions that MUST 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 nicely

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 to define a word's validity. 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 switch to the neighbor switch 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 *Energy* transfer 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 tags are defined and advertised by each Nexus 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*
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.

8.1 Alternatives to Parity

It is acceptable to substitute equivalent implementations of the Parity bit at the link layer. Encoded (or one might say, compressed) in the parity bits is both the concept of validity in addition to whether the slot holding the word is *Allocated* (*IsoStream* vs. *μPkt*). However, in the parity implementation, the link layer MUST keep careful track of each slot and have good recovery code if it misses an *IsoStream* allocation, because the information can only be decompressed from the parity bit over multiple frames on a given slot.
Alternatively, a link layer could simplify by using two bits: One *Allocated* bit, and one *Erasure* bit. In this implementation, the word is allocated to an *IsoStream* (STRM_ODD) if the *Allocated* bit is set, or available for *μPkt* data (MSG_EVEN) if the *Allocated* bit is 0. Instead of using the parity of the word to define its validity, it uses an additional *Erasure* bit for this. The word is valid if the *Erasure* bit is 0, and the word is assumed to be lost or corrupted if this *Erasure* bit is set. In this way, the link layer doesn't need to have complex recovery code for when it misses an *InitIsoStream*.

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

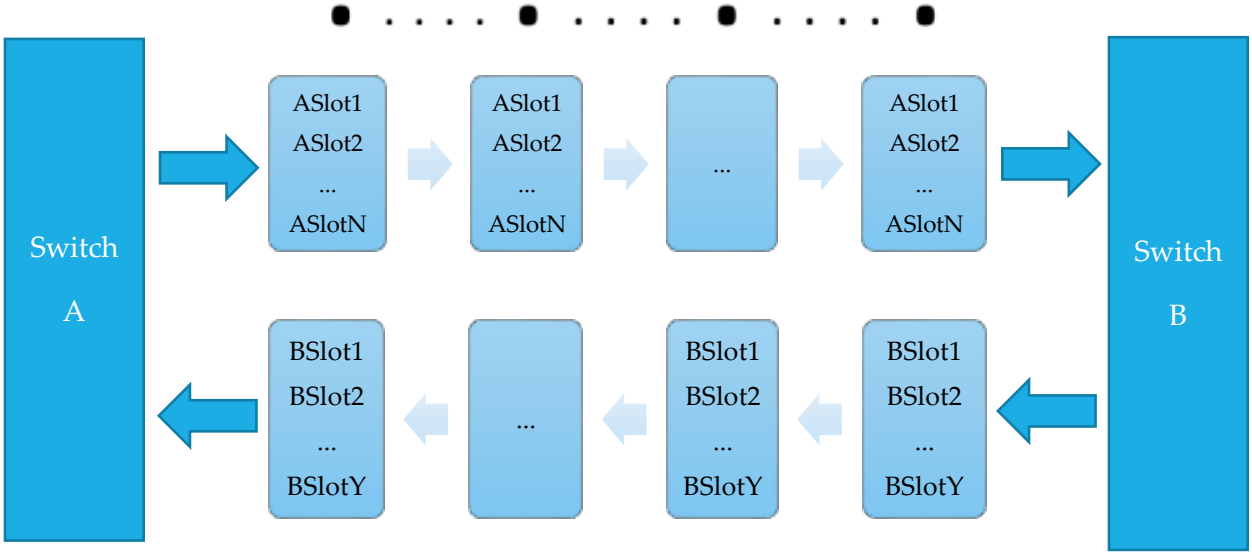


Figure 1. Link Layer frames exchanged between two neighbor switches

Figure 1 shows a generic full duplex link exchanging isochronous frames between two neighbor switches. 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 switch MUST be able to identify the slots of a valid frame that arrives on one of its links
- A switch SHOULD be able to identify the slots of a valid frame even immediately following missed or corrupt frames
- A switch 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 switch MUST advertise the best-case and worst-case word-corruption rates of its output links
- A switch MAY use error-correction codes to ensure the error rate meets the advertised value
- A frame MAY contain hashes to determine validity
- A switch 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 switch 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, *Energy*, 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 *LinkLayerNexusAdvertisement*
- 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 *Energy*
- Check *Energy* 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.
To provide isochronous streams, the IsoGrid runs with a synchronized frequency, similar to the way an electrical grid runs on Utility Frequency (except much higher frequency).

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

9.2.1 Stationary Switches

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

If a switch is stationary relative to some number of neighbor switches, the switch 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 switches MAY be arbitrarily long-lived.

To illustrate how this might be implemented within a switch, 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 switches 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 switches closer to TCG. The IsoGrid as a whole is reliant on the collective work of all switches 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 switch, 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 switch 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 switches 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 switches rely on their neighbors to collectively lock to the TCG frame rate, a switch 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 switch alone has the consequences of the bad clock.
There is no need for a Time Cop :-)

9.2.5 Mobile Switches

Switches 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, switches that move arbitrarily long distances, MUST have transitory links. These switches MAY pre-compute the buffer requirements for a transitory link. As the switch continues to move, it can only maintain the link for so long before the buffers are exhausted.
For a switch that is moving axially between two other switches, 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 switches that don't speak the same link-layer protocol.
All link-layer protocols MUST have some sort of *µPkt* enveloping mechanism that allows the switches 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 a *µPkt*.
All link-layer protocols MUST support a means to envelope the following member that exists within *InitIsoStream* and *InitIsoStreamByBreadcrumb* (see *IsoStream* layer):

Member	**Description**

IsoStreamWordRate	The word rate of the IsoStream expressed as a power of 2.

The *µPkt* Types that are supported by the switch MUST be declared in its *NexusAdvertisement*.

9.4 Initial Linkups

A mobile switch 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). Switches also have to handle connecting to the network for the first time. A newly arrived switch MAY make a request to establish a link with a nearby switch. Switches 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 switch could require the new switch to:

1. Perform a compute task, or proof of work
2. Perform a data relay task
3. Be in close physical proximity to the switch
4. Make lots of attempts, and the switch 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.

10 Network Layer (*µPkt*)

The network layer supports routing of extensible *µPkts*.

10.1 µPkt Types

- Extendable *µPkt* Types
- Able to be implemented in hardware
- Exactly 8 words (128 octets) long
- Even if only the sender understands an extended Type, the full *µPkt* can still make it to the destination
- Switches can treat a *µPkt* as a base type if they don't understand the extended version

µPkt Types follow a strict single-inheritance extension hierarchy. At the root of the hierarchy, *µPktRoot* primarily holds the 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* Types. If it gets to that state, perhaps a new root or a completely different network layer would be necessary.
This specification defines the routing mechanisms by branching a *µPkt* Type off *µPktRoot*: *µRouteByBreadcrumb*

All other *µPkt* types defined in this specification inherit (either directly or indirectly) from these Types.
It's possible to define additional roots or routing mechanisms, but to have it 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.
Since *µPkt* Types follow a strict single-inheritance hierarchy, if a derived type adds fields, it MUST only do so in bits not used by the fields of the base type. Fields are 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 require minimal CPU branching.

10.2 µPkt 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: 0x4BD293D2

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

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

Member	Size	Word#	Offset
Reserved for Link Layer	4 octets	0	0
FullType	4 octets	0	4
µPktEnergy	8 octets	0	8
TickAndPriority	1 octet	1	7

10.2.1.1 µPktEnergy

This is a 64 bit integer value that describes the amount of energy being transferred along with the *µPkt*.

To calculate the amount of *Energy* to provide for a simple *µPkt*, simply add up all the *Energy* requirements along the route.

Requirements:

A. The switch receiving this value MUST deterministically subtract the exact amount of energy that covers the switch's advertised *Energy* of handling the *μPkt* at the given Tick and Priority
B. If, after the subtraction step, the energy would be negative, the switch MUST fail the *μPkt*.
C. Otherwise, the switch MUST forward the resulting value in the *μPktEnergy* field of the *μPkt* sent over the appropriate outgoing link

10.2.1.2 Tick

This field is the lower 2 bits of the TickAndPriority member. This field represents the current TCG time in 8 second Ticks (modulo 4 because there are only 2 bits). It is used to specify the approximate time when the *μPkt* was sent for purposes of deterministic energy transfer.

Requirements:

A. Route advertisements MUST include a validity period specified by start and end times.
B. A switch MUST interpret all values as either the current time or a time in the past (never a time in the future).
C. A switch MUST accept the three most recent Tick values {current Tick, previous Tick, and two Ticks ago} as valid.
D. A switch MUST fail all *μPkt* (with TICK_EXPIRED) that have a Tick value of three Ticks ago
    a. This gives a wide 8-16 second window to traverse the network before the *μPkt* would be failed

10.2.1.3 Priority

This field is 4 bits (bit 2 through bit 5) of the *TickAndPriority* member. This field represents the desired maximum priority level of the *μPkt* as decided by the source. A higher number value indicates a higher priority. 0 is the lowest priority level.

Requirements:

A. Route advertisements MUST include a priority value.
B. A switch MUST always have a valid advertisement for priority 0.
C. If a switch receives a *μPkt* with a priority value not advertised by the switch, it MUST accept it as if it were the closest advertised priority below the one sent, leaving the priority of the forwarded *μPkt* unchanged.
D. The *Energy* difference between each consecutive priority level SHOULD be approximately a factor of 2.
E. The switch MUST use the lowest available priority level at any given moment based on the utilization level of the switch
    a. This means that a *μPkt* with higher priority will automatically be given the lower priority energy consumption if congestion is low

10.3 Standard IsoGrid *µPkt* Type Definitions

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

10.3.1 *µRouteByBreadcrumb*

Inherits: *µPktRoot*
FullType: 0x452EB7CF

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

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

Member	Size	Word#	Offset
BreadcrumbTrailId	16 octets	3	0

The MSB of the most significant octet of the *BreadcrumbTrailId* defines the direction: If this bit is cleared, the direction is the normal (forward) direction. If this bit is set, the *BreadcrumbTrailId* refers to the reversed direction of the original route.

10.3.2 *µPktWithHopCounter*

Inherits: *µRouteByBreadcrumb*
FullType: 0x4F8F8BE2

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

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

Member	Size	Word#	Offset
HopCounter	4 octets	1	0

10.3.2.1 HopCounter

HopCounter is a 32 bit unsigned integer that counts each hop of the *µPkt*.

Requirements:

A. Each switch MUST increment this integer by exactly 1 every hop.
B. The originator MUST randomly choose a 32 bit unsigned number.

10.3.3 *µPktWithReply*

Inherits: *µPktWithHopCounter*
FullType: 0x4D7B9991
Success Reply: *µPkt_Success*

Failure Reply: *µPkt_Failure*

µPktWithReply MAY be used to send a request with a built-in reply and energy for the reply. Each switch that processes this MUST fail it with *OverCongestedReturn* if the input coming from the next hop has used more than 50% of the *µPkt* reservation. This incentivizes switches to keep the *µPkt* reservation appropriately sized to prevent dropped *µPkts* in practice.

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

Member	Size	Word#	Offset
HopCounter -> µPktId	4 octets	1	0
ReplyEnergy	8 octets	1	8

The *HopCounter* inherited member is aliased as *µPktId* for *µPktWithReply* and all the reply *µPkt* types. For the reply types, *HopCounter* MUST be decremented by each hop.
A client that isn't fully aware of the *Energy* requirements of the route is able to send a small amount of *Energy* and see how far it goes along the route. The destination MUST send back the excess *Energy* (except in the case of an *EccFlow* setup).

Requirements:

A. If the *µPktEnergy* exceeds the advertised maximum transfer limit (in either direction), the switch MUST fail the *µPkt* with the appropriate error code.

10.3.3.1 ReplyEnergy

ReplyEnergy is a 64 bit value that adds up the *Energy* needed to send a reply back to the originating source.

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

Requirements:

A. The originator MUST initialize this value with the amount of *Energy* that the originator would need to accept and decode a simple 8 word reply *µPkt*.
B. Each switch MUST add to this value the advertised amount of *Energy* that switch would need to switch a single 8 word simple, data-only, reply *µPkt*.
C. If *ReplyEnergy* exceeds *µPktEnergy*, then the switch MUST fail the outbound *µPkt* and reply with *µPkt_Failure* instead.

10.3.4 *GetRouteUtilizationFactor*

Inherits: *µPktWithReply*
FullType: 0xF63FF952
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	Word#	Offset
MostCongestionLevel	1 octet	3	4
LeastCongestionLevel	1 octet	3	5
MostCongestionHopCount	4 octets	4	0
LeastCongestionHopCount	4 octets	4	4

TODO: Specify member handling

10.4 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.

10.4.1 *µPkt_Success*

Inherits: *µRouteByBreadcrumb*
FullType: 0x479E96AA

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	Word#	Offset
HopCountAtSuccess	4 octets	2	0
µPktId	4 octets	1	0
EnergyAtLastHop	8 octets	1	8

TODO: Specify member handling

10.4.2 *GetRouteUtilizationFactor_Success*

Inherits: *µPkt_Success*
FullType: 0x942B5C20

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	Word#	Offset
MostCongestionLevel	1 octet	3	12

LeastCongestionLevel	1 octet	3	13
MostCongestionHopCount	4 octets	4	0
LeastCongestionHopCount	4 octets	4	4

TODO: Specify member handling

10.5 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.

10.5.1 µPkt_Failure

Inherits: *µRouteByBreadcrumb*
FullType: 0xEA3835A4

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**	**Word#**	**Offset**
HopCountAtFailure	4 octets	2	0
µPktId	4 octets	1	0
FailureCode	1 octet	1	4
EnergyAtLastHop	8 octets	1	8
LocatorHashAtFailure	64 octets (4 words)	4	0

TODO: Specify member handling

10.5.1.1 FailureCode

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

Code	**Value**	**Description**
NoSuchRoute	1	The switch doesn't know how to route the *µPkt*
OverCongested	2	The next hop is too congested to send the *µPkt*
OverCongestedReturn	2	The return link from the next hop is too congested to expect a reply for the *µPkt* (so it's being failed early)
EnergyExhausted	3	*ReplyEnergy* exceeds *µPktEnergy*

ExceededMaxEnergy	4	*µPktEnergy* exceeded the maximum the switch is willing to transfer per *µPkt*.
InsufficientEnergy	5	The *µPktEnergy* were insufficient for the destination to perform the final handling of the *µPkt*.
IsoStream_BreadcrumbResourcesExhausted	32	Breadcrumb resource unavailable.
IsoStream_ExceededMaxWordRate	35	The *IsoStreamWordRate* exceeded the maximum that the switch supports.
IsoStream_BelowMinWordRate	36	The *IsoStreamWordRate* was below the minimum that the switch supports.
IsoStream_ExceededMaxEnergy	37	*IsoStreamEnergy* exceeded the maximum the switch is willing to transfer per word of *IsoStream*.

10.5.2 *GetRouteUtilizationFactor_Failure*

Inherits: *µPkt_Failure*
FullType: 0x4880A458

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**	**Word#**	**Offset**
MostCongestionLevel	1 octet	6	14
LeastCongestionLevel	1 octet	6	15
MostCongestionHopCount	4 octets	7	0
LeastCongestionHopCount	4 octets	7	4

Note that the above members are more important to have returned to the sender than the LocatorHashAtFailure member (which would ordinarily take these bit positions within the packet.

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. The *IsoStreamWordCount* is decremented by the amount of *IsoStreamRoute* header

words used up in the routing process. In this way, with an *IsoStream* with an initial 100 *IsoStreamWordCount*, all switches end the *IsoStream* at the same point in the stream.
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 InitIsoStreamByBreadcrumb

Inherits: *µPktWithReply*
FullType: 0x466A98C0
Success Reply: *InitIsoStream_Success*
Failure Reply: *InitIsoStream_Failure*

InitIsoStreamByBreadcrumb MAY be used to start an *IsoStream*. All IsoGrid switches MUST support the *InitIsoStreamByBreadcrumb µPkt*.

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

Member	Size	Word#	Offset
IsoStreamWordCount	2 octets	1	4

Requirements:

A. After the *InitIsoStream µPkt* is accepted, the switch MUST mark the targeted input slot as an *Active IsoStream*.
B. The switch MUST allocate the specified number of isochronous slots to support the specified *IsoStreamWordRate*.

11.1.1 *IsoStreamWordCount*

IsoStreamWordCount is a simple integer value describing the number of words to be sent on the *IsoStream*. The MSB is reserved and MUST be zero.

Requirements:

A. Each switch MUST remember this value as the remaining words for the *IsoStream* on this slot
B. Each switch MUST begin counting the words of the *IsoStream* with the final (8th) word of a valid *InitIsoStream µPkt*
C. After *IsoStreamWordCount* words the switch MUST look for an *IsoStreamContinuation* word in the slot.
D. If the *IsoStreamWordCount* is greater than the allowable maximum, as advertised by the switch, the switch MUST fail the *IsoStream* connection request with the appropriate error code.

11.1.2 *IsoStreamWordRate*

The word rate is defined via a link layer protocol (which is not required globally).

Requirements:

A. The rate MUST be a power of two, as in 2 ^ (*IsoStreamWordRate*) words per TCG second.
B. If the inbound word rate is greater than the advertised supported maximum word rate of the outbound link, the switch MUST fail the *InitIsoStream* with *ExceededMaxWordRate*.
C. If the inbound word rate is smaller than the advertised supported minimum word rate of the outbound link, the switch MUST fail the *InitIsoStream* with *BelowMinWordRate*.

11.1.3 IsoStreamContinuation

This word consists of the following:

Member	Size	Offset
IsoStreamContinue	1 octet	0
IsoStreamWordCountContinue	1 octet	1
TickAndPriorityContinue	1 octet	2
IsoStreamEnergyContinue	8 octets	8

The *IsoStreamContinue* field marks the intended use of this word. If the value is 0, the *IsoStream* in this slot ends with this word. If the value is '1', the *IsoStream* continues with the below requirements.

11.1.3.1 IsoStreamWordCountContinue

The word count of the continued *IsoStream* encoded as an exponent.

0:7 (8 bit) base2 exponent, biased by 5

The final value is equal to: (1 << exponent) + 5

Requirements:

A. If the *IsoStreamWordCountContinue* value is greater than the allowable maximum, as advertised by the switch, the switch MUST set it to the allowable maximum and forward on the *Continuation*.
    a. Another *Continuation* on the *IsoStream* is not possible at this point, but the source should have enough time to receive the information that the failure occurred.

11.1.3.2 IsoStreamEnergyContinue

This is a 64 bit integer value that describes the amount of energy being transferred along with the *Continuation*. To calculate the amount of *Energy* to provide, simply add up all the *Energy* requirements along the route.

Requirements:

A. The switch receiving this value MUST deterministically subtract the exact amount of energy that covers the switch's advertised energy required to handle a word of *IsoStream* at the given Tick and Priority, multiplied by *IsoStreamWordCountContinue*
B. If, after the subtraction step, the energy would be negative, the switch MUST fail the *Continuation*.
C. Otherwise, the switch MUST forward the resulting value in the *IsoStreamEnergy* field of the *Continuation.*

11.1.4 Miscellaneous

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

11.2 InitIsoStream_Success

Inherits: *μPkt_Success*
FullType: 0x4AC2A8B1

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: 0x51AF6D77

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 InitIsoStream

Inherits: *InitIsoStreamByBreadcrumb*
FullType: 0x46FDBFD9
Success Reply: *InitIsoStream_Success*
Failure Reply: *InitIsoStream_Failure*

To declare a breadcrumb trail, the source MAY use the *InitIsoStream* Type (or a *μPkt* that inherits from it).

In this *µPkt* Type, the source provides a variable-length series of route instructions to be used at each hop (Source Routing).

All IsoGrid switches MUST support *InitIsoStream* (or a *µPkt* that inherits from it).

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

Member	**Size**	**Word#**	**Offset**
RouteTagShiftedBits	1 octet	1	6
CurrentRouteTags	16 octets	6	0
NextRouteTags	16 octets	7	0

The *BreadcrumbTrailId* is used in future *µRouteByBreadcrumb µPkts*.
The MSB of *BreadcrumbTrailId* MUST be 0.
The *RouteTagShiftedBits* defines the number of bits that are unavailable in the *CurrentRouteTag* field (because they were shifted out of the word). Initially, this value SHOULD be set to zero. The next hop's *IsoStreamRoute* tag MUST always start at bit offset 0 of word 6. A switch MUST NOT send a *RouteTagShiftedBits* greater than 127, and instead MUST wait for the next word of the IsoStream's slot to re-fill the completely empty *NextRouteTags* member.

Requirements:

A. All the requirements of *InitIsoStreamByBreadcrumb* also apply to *InitIsoStream*.
B. In the isochronous slot allocated by an *InitIsoStream µPkt*, the source MUST stream a series of *IsoStreamRoute* tags that define the route, one switch at a time.
    a. The link layer protocol defines which slot this is
C. The number of bits in each *IsoStreamRoute* tag consumed by each switch and their meaning MUST be defined and advertised by the switch.
D. If an *IsoStreamRoute* isn't recognized by a switch, the *InitIsoStream* MUST be failed.
E. 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 *BreadcrumbTrailId* is already associated with a different *IsoStreamRoute*, the *InitIsoStream* MUST be failed with *IsoStream_BreadcrumbCollision FailureCode*.
    c. If the breadcrumb cannot be allocated due to lack of resources, the *InitIsoStream* MUST be failed with *IsoStream_BreadcrumbResourcesExhausted FailureCode*.
    d. The switch MUST associate the tuple [*BreadcrumbTrailId*, InputLink] with the *IsoStreamRoute* at this hop
    e. The switch MUST ensure that the *BreadcrumbTrailId* sent to the next switch is unique for the OutputLink
    f. The switch MUST associate the tuple [forwarded *BreadcrumbTrailId*, OutputLink] with the *IsoStreamRoute* that heads back to the input link.
        i. This allows a *µPkt* to traverse the reverse path back to the original source.
    g. The breadcrumb MUST be usable for subsequent *µRouteByBreadcrumb* for at least 8 seconds after the InitIsoStream is recognized by the switch

h. All switches MUST shift-right out their decoded *IsoStreamRoute* tag and increment the *RouteTagShiftedBits* by the size of the decoded *IsoStreamRoute* tag
i. When the switch receives word #7 the frame counter for the incoming stream MUST be initialized to the *IsoStreamWordCount*.

F. *RouteTagShiftedBits* MUST be strictly less than 128
   a. If the value is incremented beyond 127, the switch MUST subtract 128 from *RouteTagShiftedBits* and place the forwarded *InitIsoStream μPkt* in the next frame so that the next switch receives only the next word of the list of *IsoStreamRoute* tags. This is referred to as *WordPopped* because a word is 'popped' off the stream.
   b. The frame counter of the outbound stream starts when the forwarded *InitIsoStream μPkt* is sent
   c. When an active stream was *WordPopped*, the switch MUST push one last new word on to the end of the outbound stream when the frame counter completes.
      i. This last word MUST contain random bits in the lower 32 bits. The rest of the bits MUST be zero.
      ii. One source of the randomness MAY be an XOR of all frame data up to this point.

Each IsoStream word is 128 bits (plus an additional parity bit). Many route hops don't need to use so many bits, so it's beneficial to be able to pack multiple smaller *IsoStreamRoute* tags into each word. When doing so, the next *IsoStreamRoute* tag begins at bit 0 of the *CurrentRouteTags* member.
When an *IsoStreamRoute* tag is recognized by a switch, it MUST little-shift out the bits used for that tag, leaving the next *IsoStreamRoute* tag starting at the LSB (bit 0) of *CurrentRouteTags*.

Let's say there's 1,000 octets of *IsoStreamRoute* (500 hops, 2 octets each hop)
Effect of 2 octet *RouteTags* 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*.

12 Session Layer: Error Correction Coded Flow

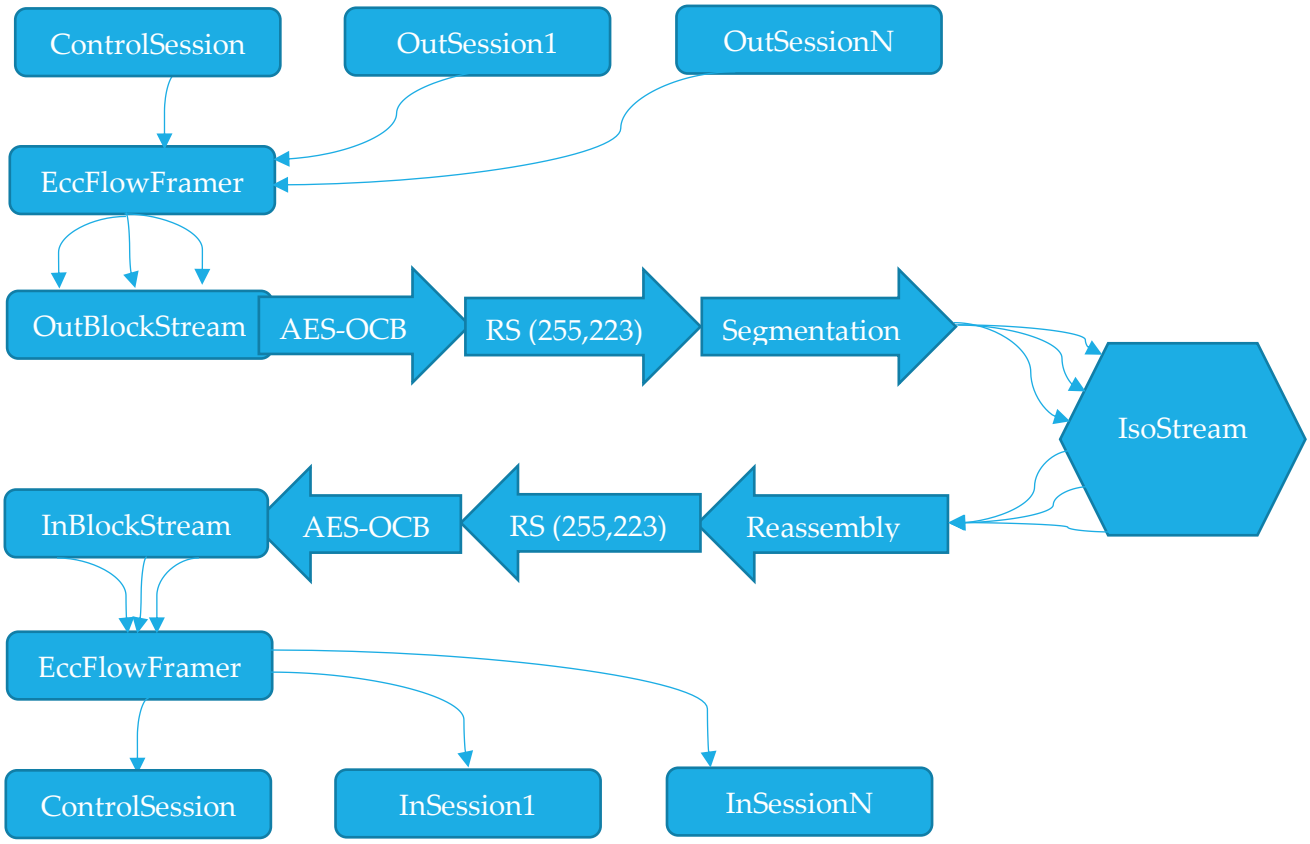
TODO: This section is still a work in progress and may contain raw notes!

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 switch. The client side is responsible for the energy 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-OCB
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-OCB 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 Properties of Sessions

12.2.1 Reliability

TCP's "Reliable"-ness is expensive and ultimately pointless:
It's expensive because retransmits at the transport layer require the network stack to make a copy of **all** data as it's being transmitted. It's ultimately pointless because on a low-latency route, TCP has a very fast timeout if packets start going unacknowledged: If the RTT of the connection is 5ms, then the connection will get closed after 315ms of unacknowledged packets. Because of this, TCP connections often get reset, and higher layer protocols are used to keep longer-term activities moving along. So rather than trying in vain to have the session protocol keep a connection active for a long time, just focus on making session failures quite rare. It makes no sense to force the network stack implementation to make copies of all data just to be able to support retries at the session layer when the session/transport can be made so resilient to failures.

12.2.2 Ordered

Since the underlying transport and session has no retransmission, the *EccFlowSession* provides ordered packets.

12.2.3 Misc

It's possible to layer TCP or UDP on top of a session to give backward compatibility with existing application software.
The *EccFlow* implementation MAY present the application layer with a choice to receive each inbound session as Raw, Dropping, or Reset. The outbound side has no options available, and as such these choices have no protocol impact.

Receive option	Description
Raw	All recognizable packets are delivered to the application layer, even if they contain bad data (and don't request resend)
Dropping	If enough *IsoStreams* lose data, drop the packet (and don't request resend) TCP or UDP MAY be layered on top of this option to provide backward compatibility with existing application logic.
Reset	If enough *IsoStreams* lose data and a packet is dropped, drop the entire session. This is expected to be the common case (at least for Async operation), as it has the simplest programming model.

Isochronous Sessions:

- This is an option on the sending side and has no protocol impact
- The sending stack MUST ensure that the *EccFlow* is sized appropriately to handle all the Isochronous sessions
- The sending stack MUST check if the system is capable of sending data at this rate

- 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:

- This is an option on the sending side and has no protocol impact
- EccFlow implementation MUST gauge the rate at which the destination can handle the data and the rate at which the source can send it
- Allocate (provide *Energy* 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. additional link redundancy
- Maintaining a mutually acceptable *Energy* balance between the two nodes
- Store the *EccFlow* context as long as the client has provided adequate *Energy* for the server to do so (the server MUST declare the *Energy* required for such storage)

The *EccFlow* between two nodes constitutes an arbitrarily long term bi-directional communication 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 provide *Energy* 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

12.4.1 *EccFlowSession* Packets

All data session packets MUST have the following structure:

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

The *PayloadSize* member is 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 structure of *Payload* is defined by the *SessionProtocolId* and associated with the *SessionId* when a new session is created.

12.4.2 *ControlSession* Packets

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* bits. *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 in a specific direction: Two sessions in opposite directions that share the same *SessionId* value are probably not related.

If the protocol for a *ControlSession* packet 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 *ControlSession* packet types (right now these are just notes):

- *EccFlowFrame* initialization parameters
- *StartSessionId*
    - 3 octet *AdditionalTypeSpace* (must be zero)
    - 8 octet *ProtocolId*
    - 2 octet *ServiceInstance*
    - 9 octet *SessionIdForRequest*
    - 9 octet *SessionIdForResponse*
    - Example *SessionProtocols*:
        - *GetLinkAdvertisement*(s)
        - *ProvideLinkAdvertisement*(s)
        - *GetBestRoutes*
- Alias (rename) a *SessionId*
- Query support for some App layer service (*ProtocolId*)
- Announce/confirm support for some App layer service (*ProtocolId*)
- 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 *Energy* balance
- Return Errors:
    - Unrecognized *SessionId* received
    - Unrecognized *ProtocolId*
    - No instance
    - 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
DelayAccept	1 octet
TotalStreams	1 octet
StreamId	8 octets
SplitEccFlowKey	32 octets
ReplyIsoStreamWordRate	2 octets
ReplyIsoStreamEnergy	8 octets

Client sends a set of *InitEccFlow* messages:

- The *InitEccFlow* MAY be sent within an *IsoStream* created via *InitIsoStream*

    - The client MUST terminate the route with the *InitEccFlow* Tag advertised by the destination
    - For each *InitEccFlow,* the client MUST provide enough *IsoStreamEnergy* to cover:
        - The total *Energy* to get the *InitEccFlow* stream delivered to the destination
        - The total *Energy* 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 *Energy*-effective relative to desired redundancy
- The client MUST set *ReplyIsoStreamWordRate* to the desired *IsoStreamWordRate* of the reply *AcceptEccFlow*
- The client MUST set *ReplyIsoStreamEnergy* to an amount that the client has calculated would be at least enough *IsoStreamEnergy* for the reply *AcceptEccFlow* to be sent via an *IsoStream* all the way back to the client on the reverse *BreadcrumbTrailId*
- The client MUST set *DelayAccept* to the amount of time the server should wait to receive additional streams after receiving the first stream
    - This value is encoded as a floating point value: The two LSB encode a multiplier with an assumed 'hidden' 3rd bit, and the 6 MSB encode a binary exponent
    - For this value, the unit of time is (1 TCG second)/(2^50)
- 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 *AuthTag* using AES-OCB 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 at staggered times, to avoid overwhelming the switches along the route.
    - The client SHOULD set *DelayAccept* to as small a value as possible that still allows all these streams to arrive at the server given expected network jitter

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 with matching *EccFlowId* + *DelayAccept* values (perhaps some are lost or corrupted along the way):

- If, after subtracting all the *Energy* required by the server to process an *InitEccFlow* message, the number of would go negative, the server MUST fail the *InitIsoStream* message with *InsufficientEnergy*
- Upon acceptance, the server MUST associate the provided *BreadcrumbTrailId* with the *EccFlowId* + *DelayAccept* using the same validity period rules as the switches along the route (specified in the *InitIsoStream* section)
- The server MUST decrypt and authenticate the rest of *InitEccFlow* with the provided *InitKey* and *InitIV*
- After waiting the amount of time represented by *DelayAccept* after the first *InitEccFlow* message of the set, the server MUST assemble and send 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* and *DelayAccept*.
    - 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 *InitIsoStreamByBreadcrumb*
    - The MSB of the *BreadcrumbTrailId* MUST be set to 1
    - One *IsoStream* for each *BreadcrumbTrailId* associated with the *EccFlowId* (to provide redundancy)
    - The *IsoStreamWordRate* MUST match the rate requested by the *StreamId*

- The *IsoStreamEnergy* MUST match the amount requested by the *StreamId*.
- All the remaining *Energy* 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 *BreadcrumbTrailId*)
- The server MUST encrypt the message with the *InitKey* corresponding to the *BreadcrumbTrailId* 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-OCB with *EccFlowKey*
    - The client MUST use *AcceptId* as the IV
- The client MUST send this identical assembled *ConfirmEccFlow* message within an *InitIsoStreamByBreadcrumb*
    - One *IsoStream* for each *BreadcrumbTrailId* 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-OCB 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

12.5.4 Algorithm Evolution

Over time, the chosen algorithms above may prove to have some deficiencies relative to the socioeconomic goals of this specification. If this occurs, a new handshake will have to be defined, say InitEccFlow2, and advertised as a service by nodes that support it. The designers of any replacement handshake algorithms MUST ensure that it promotes the socioeconomic goals of this specification.

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 *BreadcrumbTrailId* 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-OCB 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-OCB 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	Octets per *AuthBlock*	Octets of *AuthTag*
0	64	8
1	512	16
2	4096	16
3	32768	16
4	262144	16
5	2097152	16
6	16777216	16
7	134217728	16

- 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-OCB using *CurrentAuthBlockIV*, and the *EccFlowKey* referred to by *KeyId* in order to produce the ciphertext output stream
    - The sender MUST append the 8 or 16 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 an 8 or 16 octet *AuthTag*
    - The receiver MUST decrypt and authenticate each *AuthBlock* of ciphertext output with AES-OCB 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 each attack requires significantly more *Energy* than the client has to provide. 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. LinkTunnels are also especially useful for creating fan-out links near the self node that can target high-overhead redundancy over short distances. This is useful because we wouldn't want to have this level of high-overhead redundancy for EccFlows that go long distances; rather, they should use low-overhead redundancy that spreads the error correction codes across more links.
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 *Energy* 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 *NexusAdvertisement* protocol that facilitates 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.

A *Nexus* is a full-featured SoC, with all the SoC aspects like general compute, large storage, and arbitrary services. The *switch* is just a HW switch automaton. A *Nexus* can control many switches.
LocatorHash is for the full *Nexus*. All switches know how to route 'upstream' to the *Nexus*. The *Nexus* knows how to route to (and between) each of the switches it controls.
Advertisements are delivered at the granularity of a *Nexus*.

When a new direct link comes online, each side MUST send *LinkLayerNexusAdvertisement* with the new link included. Upon receipt of *LinkLayerNexusAdvertisement*, the *Nexus* with the higher *LocatorHash* value MUST initiate an *EccFlow* to the other *Nexus* node and use it as the transport for all the following packets. These links MUST be marked as direct (rather than *EccFlowLinkTunnel*).

When a *Nexus* sees a *Link* to a remote *Nexus* it doesn't already know about, and it knows how to reach the source *Nexus* for a reasonable *Energy*, it SHOULD use the *EccFlow* it has to the source *Nexus* and request the *NexusAdvertisement* for the unknown *Nexus*. Each *Nexus* MUST announce no more than 1024 links. Nodes SHOULD assume that other nodes that announce more than 1024 links is lying or defective.

A *Nexus* MUST never make conflicting *NexusAdvertisements*; which is defined as two different *NexusAdvertisements* where the [TickBeginValue, TickEndValue] ranges overlap, unless the difference is purely additive. A *Nexus* SHOULD make a series of *NexusAdvertisements* where the TickBeginValue of each successive *NexusAdvertisement* is equal to the TickEndValue of the previous *NexusAdvertisement* plus 1 Tick. A *Nexus* SHOULD specify a [TickBeginValue, TickEndValue] range of greater than 1 week. A Nexus MUST deliver its updated *NexusAdvertisements* to all its direct neighbors via *LinkLayerNexusAdvertisement*.

Advertisements are specified as Type-Length-Value binary-encoded hierarchical data. Since the data is hierarchical, it's often easiest to display it in human-readable form as XML

The first two (MSB) bits of Type defines the size of Type: 14-bit, 30-bit, 62-bit, or 126-bit
The Length Field is half the size of the Type field rounded up to the nearest byte.
14-bit Types are valid when contained in any size Type field
30-bit Types are valid when contained in a 30-bit, 62-bit, or 126-bit Type field
62-bit Types are valid when contained in an 62-bit or 126-bit Type field
126-bit Types are obviously only valid within a 126-bit Type field

The 4 classes of possible [Type][Length][Value] ordering and total lengths:
[2 byte][1 byte][0 to 255 bytes]

[4 byte][2 byte][0 to 65535 bytes]
[8 byte][4 byte][0 to (2^32)-1 bytes]
[16 byte][8 byte][0 to (2^64)-1 bytes]

NexusAdvertisement:

- <SPHINCS Signature> (41,000 octets)
- *LocatorHash* (64 octets)
- PublicKey (1056 octets)
- Optional Text location (64 octets)
- *3DGeoHash* (10 octets)
- TickBeginValid (4 octets)
- TickEndValid (4 octets)
- µPkt Type Declarations (16 + 4 + 12 octets each [TypeID + *Energy* + TypeSpecificData])
- Service Declarations: (16 + 16 + 4 + 12 octets each [ServiceID + Tag + *Energy* + ServiceSpecificData])
    - *LinkAdvertisement*
    - *InitEccFlow*
    - *InitEccFlowSegment*
- <DirectNexusLink> : List of Nexus nodes to which the local Nexus specifies a link
    - LocatorHash (64 octets)
    - Index (or are these just counted?)
- SwitchN:
    - *3DGeoHash* (10 octets)
    - LinkN:
        - Tag
        - TagBits
        - *3DGeoHash* of Destination Switch (10 octets)
        - Index of Nexus for Destination Switch
        - Flags (1 octet)
            - Direct or *EccFlowTunnel* (1 bit)
        - Lowest Supported Rate (1 octet)
        - Highest Supported Rate (1 octet)
        - Review the Slot Isochronous Standard for other requirements
        - Maximum *IsoStreamWordCount* (2 octets)
        - 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)
        - <Price>
            - Maximum *Energy* transfer expressed as 16-bit floating-point (2 octets)
            - *µPkt Energy* expressed as 16-bit floating-point (2 octets)
            - *IsoStream Energy* required per word expressed as 16-bit floating-point (2 octets)

- <TickPeriodOverride>
- <MonthOverride>
- <DayOfMonthOverride>
- <DayOfWeekOverride>
- <TimeOfDayOverride>
- <ElementExpiration>: Expresses the Tick at which the element is no longer valid

Overrides can be nested, and order matters: The first Override at a given level to apply to a given Tick takes precedence.

LinkLayerNexusAdvertisement:
Same contents as *NexusAdvertisement*. Sent via Link Layer protocol agreement. No *Energy* required because neighbors can be friendly (and avoid spamming each other).

13.1 Earth 3D Cartesian GeoHash

This coordinate system uses 10 octets to specify the general location of anything on the surface of the earth.

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
The bounds of each axis is from 0 to 2m^23=8,388,608m.
Thereafter, every third bit refers to the relevant axis, and the bounds are bisected like a normal GeoHash.
78 bits gives 500cm precision anywhere within a bounding box 2m^24 on a side centered on the Earth. This covers the surface of the Earth. The node SHOULD attempt to set the *3DGeoHash* to be accurate within 1m.
A different location mechanism needs to be defined for mobile switches like comsats, airplanes, and boats.

13.2 LocatorHash

The *NexusAdvertisement* contains the ID of the node, represented as a 64 octet *LocatorHash*. The last 10 octets form the Planet and Key Standard of the *LocatorHash*:

1. *GravityBody*: {Sun, Planet, Lagrangian point, etc.} (2 octets)
    a. *Body* (13 bits)
    b. *GravityNexus* (3 least significant bits)
2. Key Standard (8 octets)

The *Body* field specifies the gravitational body of the solar system to which the *LocatorHash* is relative to. The large gravitational bodies in the solar system are numbered (in order of decreasing mass): Sun (0), Jupiter (1), Saturn (2), Neptune (3), Uranus (4), Earth (5), Venus, Mars, Mercury, Ganymede, Titan, Calisto, Io, Moon, Europa, etc…

Each *Body* is further subdivided into one of 8 *GravityNexus* values:

Value	Meaning
0	The center of mass of the *Body*
1	The L1 Lagrangian between the *Body* and the *Body* it orbits
2	The L2 Lagrangian between the *Body* and the *Body* it orbits
3	The L3 Lagrangian between the *Body* and the *Body* it orbits
4	The L4 Lagrangian between the *Body* and the *Body* it orbits
5	The L5 Lagrangian between the *Body* and the *Body* it orbits
6	Reserved
7	Reserved

The Key Standard defines the specific standard method used for hashing and the Public Key type.
The other 54 octets provide a *KeyHash* with a meaning specific to the Key Standard.

The following shows the length of a full *LocatorHash* as written for IsoGrid in Base64:
`\\12345678901234\123456789112345678921234567893123456789412345678951234567896123456789712`

13.3 Earth SPHINCS-256 Public Key Hash Skein-512

Key Standard: 0xA2FF41EBB3ED0735 (TODO: Spell something readable in Base64)

"\e\" == "EarthCenter\0xA2FF41EBB3ED0734\"

`\e\123456789112345678921234567893123456789412345678951234567896123456789712`

The *Nexus's* PublicKey MUST use SPHINCS-256, which is a Hash-based (post-quantum) egalitarian public key signature system.
The *KeyHash* MUST be the first 54 octets of the Skein-512 hash of the *Nexus's* full PublicKey. The least-significant 16 (<CONFIRM THIS>) bits of the hash MUST be zero in order to be considered valid. This forces a compute cost to creating new identities which can help prevent some types of Sybil attacks. The IsoGrid Foundation SHOULD change the Key Standard every so often to require higher numbers of trailing zeros over time: This exercises the extensibility mechanisms and also creates an ongoing cost to Sybil attacks.

13.4 Methodically Verified Routes

Routing information that has been built up hop by hop via *EccFlow* to each *Nexus* 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, switch, and link information it receives is authentic. The only uncertainty would be the possibility of widespread collusion among neighboring network participants.

13.5 Just-In-Time Routes

Building up verified routes to a far-away *Nexus* 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 to successfully perform a man-in-the-middle attack is likely smaller. Additionally, colluding nodes could agree to send routes through higher-*Energy* conspirators.
This is called a "Just-In-Time" route, because it is possible to build up routes to far away nodes with 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

13.5.1 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
SourceLocatorHash	64 octets	The LocatorHash of the start of the route
DestinationLocatorHash	64 octets	The LocatorHash of the end of the route
MaxEnergyLimit	4 octets	
MaxNodeReference	2 octets	Maximum number of times that a single node can be referenced within a route

MaxLatencyLimit	4 octets	
MinEnergyTransferLimit	4 octets	
MaxIsoStreamHeaderLimit	4 octets	
RangeRateLimit	2 octets	
MinWordCountLimit	2 octets	
MaxHopCountLimit	2 octets	

This message has three *Energy* components: A base component, *Energy* per returned complete route, and *Energy* per partial route. No *Energy* is used 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 Destination, it MUST set IsSingleDestination := 1.
If the recipient doesn't have a verified route that goes all the way to the Destination, 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.

13.5.2 MultiRouteContainer

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

Field Name	**Size**	**Description**
SourceLocatorHash	64 octets	
IsSingleDestination	1bit	1: Only a single destination node is included 0: Each route leads to a different destination
DestinationLocatorHash	64 octets	(Included only if IsSingleDestination == 1)
RouteCount	2 octets	The number of routes in the container
Routes	Variable	The list of routes:
DestinationLocatorHash	64 octets	(Included only if IsSingleDestination == 0)
RouteEnergy	8 octets	The total *Energy* required by the route
RouteLatency	4 octets	The total latency of the route
RouteEnergyTransferLimit	4 octets	The most *Energy* that will be transferred by the route
RouteRateRange	2 octets	The range of rates available in the route
RouteWordCountLimit	2 octets	The maximum word count limit 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.

13.6 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 *Energy* demand 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. A few ways nodes could be smarter about verified routing data are described here.
It's important that the chosen solution avoids any negative socio-economic effects.

13.6.1 Random Trailblazing

DEPRECATED: The 'random trailblazing' idea has been deprecated in favor of HMLM (in the section below this one).
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.

13.6.2 HashMatchLogMap Protocol (*HMLM*)

HMLM is the de-facto standard route discovery algorithm of the IsoGrid. *HMLM* is to IsoGrid as BGP is to TCP/IP.
Some ways that *HMLM* is like BGP:

- They both provide a system for finding routes between any two nodes on the network
- They are both built on top of higher level networking protocols; *HMLM* uses *EccFlow* (the IsoGrid's Session Layer protocol); BGP uses TCP (the Internet's Transport Layer Protocol)
- Neither protocol defines the network itself but each play a critical part in the scalability of the route finding system itself

Some ways that *HMLM* and BGP are different:

- *HMLM* scales O(log(N)), where N is the number of routers/nodes, but BGP scales linearly
- BGP is hard to configure and fragile, whereas *HMLM* is largely self-configuring and self-healing
- *HMLM* allows a convenient distributed content addressable storage (CAS) system that can be used by higher level services

An IsoGrid node has two synergistic motivations to run the *HMLM* algorithms: 1) It provides a scalable means for a node to find routes across the IsoGrid on its own behalf, 2) It allows a node an economic opportunity to provide routing information and distributed CAS to paying clients.

The CAS formed by *HMLM* is critical for locating content on the IsoGrid. One of the critical features of this CAS is that it's easy and logical to find things nearby if they are there before having to look further and it should converge toward a logical path to optimize caching.

HMLM borrows and extends a lot of ideas from S/Kademlia:

- NodeIds are hashes of the node's *PublicKey*: See *LocatorHash*
- Requires a number of trailing zeros in the *LocatorHash* to be valid (don't need to transmit these zeros)
- XOR metric (*HashMatch*), placed into k-buckets
- Set *Energy* requirements for queries by K-bucket, which allows nodes to know how much *Energy* is required to query a given value
    a. This is efficient and sustainable because it's a bounded number of hops (one per k-bucket)
- Two sets of k-buckets: Local and Global
    a. Local k-bucket set only includes the C cheapest nodes
        1. First find the matchiest *LocatorHash* among the C cheapest nodes
    b. Global k-bucket set includes all peered nodes

HashMatch is a simple math operation that compares two *LocatorHash* values:

XOR(*LocatorHash*A, *LocatorHash*B) as Integer

In words, this operation finds the result of XORing the two *LocatorHash* values. Smaller is a 'better' *HashMatch* for purposes of data storage and scalable route tracking.

***HMLM* Initialization Routine:**

1. Initially, *HMLM* MUST communicate with its direct neighbor nodes to build up a route-tracked list of some number (C) of the least expensive nodes to access.
    a. Estimate the true *Energy* requirements of nodes based on how long they've been known (recently created/learned nodes are penalized)
    b. Estimate the true *Energy* requirements of nodes based on how successful they are at delivering answers to queries (nodes that fail to answer correctly are penalized)
        1. When first starting, use a known tree of lots of data to generate random, but known-valid, queries. A binary search tree of various extremely large datasets (like the Internet archive) might be a good choice
        2. After startup, grade nodes based on real queries
        3. Only follow links with reasonably *Energy* requirements
    c. Select the 256 cheapest neighbors with at least one completely unique outbound link (breadth-first to optimize for unique paths)
        1. Don't select nodes with only links to one of the other 256 nodes (or any of the nodes closer to the self node than the 256 nodes
        2. *EccFlows* to these nearby neighbors have a high overhead to provide multi-path redundancy: Usually higher latency than the direct path, and

the overhead can be high unless the nearby IsoGrid is extremely well-connected. But this is OK since this high-overhead is limited to short hops

d. Looping over each of these selected 256 neighbors, in cheapest-to-access order:
    1. Use Dijkstra's Algorithm (or similar) to calculate the cheapest paths from this neighbor to all the other tracked nodes without going through any of the other 256 selected nodes or any node closer to the self node.
    2. Mark the links that are used as the cheapest path as used, so as to ensure that the next loop uses fully independent links
e. The above operation produces a data structure that can be used to efficiently retrieve up to 256 of the cheapest link-independent routes from the source node to any other tracked node
    1. Using a PriorityQueue, this is an O(NodeCount + LinkCount) operation. Considering that the HMLM algorithm as a whole only tracks O(log(N)) nodes of the whole IsoGrid (this is a property of the *Find Spiral Best* and *Find Global Best* algorithms), this is very scalable
    2. Note that this algorithm doesn't guarantee that the routes are node-independent. An algorithm that guarantees node-independence would have to make *Energy*-tradeoffs (like slightly increasing the *Energy* requirements of the cheapest route to get a significantly-cheaper secondary routes). However, in testing a well-connected IsoGrid simulation with link-independence, these routes are statistically very likely to be node-independent.

2. *HMLM* MUST bucketize all nodes by physical location relative to the self, into four separate *BucketSets*.
    a. Every node will exist in exactly one bucket in each of the four *BucketSets* (for a total of four buckets)
    b. In the case of overlapping buckets within a *BucketSet*, a node MUST only be placed in the smallest bucket
    c. Nodes within each bucket are sorted by *HashMatch* value

- The first set of buckets is called the *SpiralBucketSet*, and is defined like so:
    a. Bucket 1 is the self *CoordinateHash*
    b. Buckets 1-8 are the 7 smallest *CoordinateHash* cubes that are closest to the self, but farther from the center of coordinate system
    c. The next 56 *CoordinateHash* cubes are the same size, and completely surround the first 8
    d. The next 56 *CoordinateHash* cubes have edges twice the size of the last set of cubes and completely surround them. This step repeats arbitrarily large, but most nodes will only keep track of less than 2k buckets
    e. The buckets are numbered in a well-known pattern, so that other nodes can ask for specific buckets of nodes
- The next three sets of buckets are the *AxialBucketSets*. The three sets of buckets are defined by the three dimensions of the coordinate system; one set of buckets for each dimension axis. The *AxialBucketSet* for the positive 'X' axis is defined below:

    a. Bucket 0 contains no nodes
    b. Bucket 1 is the self bucket
    c. Bucket 2 extends away from the self by the smallest *CoordinateHash* in the positive X axis
    d. Bucket 3 extends the bounds by the smallest *CoordinateHash* in the positive Y axis
    e. Bucket 4 extends the bounds by the smallest *CoordinateHash* in the positive Z axis
    f. Bucket 5 extends the bounds by the smallest *CoordinateHash* in the negative Y axis
    g. Bucket 6 extends the bounds by the smallest *CoordinateHash* in the negative Z axis
    h. Bucket 7 extends by doubling the size of the positive X bound
    i. Bucket 8 extends by doubling the size of the positive Y bound
    j. Bucket 9 extends by doubling the size of the positive Z bound
    k. Bucket 10 extends by doubling the size of the negative Y bound
    l. Bucket 11 extends by doubling the size of the negative Z bound
    m. Every 5 buckets repeat h-l
- The *AxialBucketSet* for the negative 'X' axis is defined below:
    a. Bucket -1 is one smallest *CoordinateHash* unit from the self on the negative X axis
    b. Bucket -2 extends away from the self by the smallest *CoordinateHash* in the negative X axis
    c. Bucket -3 extends the bounds by the smallest *CoordinateHash* in the positive Y axis
    d. Bucket -4 extends the bounds by the smallest *CoordinateHash* in the positive Z axis
    e. Bucket -5 extends the bounds by the smallest *CoordinateHash* in the negative Y axis
    f. Bucket -6 extends the bounds by the smallest *CoordinateHash* in the negative Z axis
    g. Bucket -7 extends by doubling the size of the negative X bound
    h. Bucket -8 extends by doubling the size of the positive Y bound
    i. Bucket -9 extends by doubling the size of the positive Z bound
    j. Bucket -10 extends by doubling the size of the negative Y bound
    k. Bucket -11 extends by doubling the size of the negative Z bound
    l. Every -5 buckets repeat g-k
- The full *AxialBucketSetX* consists of the union of the negative and positive 'X' axis sets
- *AxialBucketSetY* and *AxialBucketSetZ* are defined exactly the same way, but by replacing 'X'->'Y', 'Y'->'Z', and 'Z'->'X'
- *HMLM* MUST create and maintain an index of all nodes known to it, sorted by *HashMatch* value
- *HMLM* MUST create and maintain an index of all nodes known to it, sorted by how physically close the nodes are
- *HMLM* MUST create and maintain an index of all nodes known to it, sorted by how cheap to access the nodes are
- After the initial set of nodes are route-tracked, *HMLM* SHOULD use the *Find Spiral Best HashMatch Routine*

    a. This step is an *Energy* saving feature to attempt to get reasonable independent routes when running the *Find Global Best HashMatch Routine*
- After the first run of the *Find Spiral Best HashMatch Routine* would have completed, *HMLM* MUST use the *Find Global Best HashMatch Routine*

HMLM Find Spiral Best HashMatch Routine - HMLM SHOULD:

1. Start with Bucket 2 of the *SpiralBucketSet* and count route-tracked nodes
    a. Once 25% of the route-tracked nodes have been counted, go to the next step
2. For each empty Bucket, ask a random node in a random physically adjacent bucket for its top Y best *HashMatch* nodes within its buckets that have a good (50% or more) overlap with the empty Bucket(s)
3. Add each retrieved node into the appropriate bucket(s) within the *SpiralBucketSet* and begin tracking routes to it if it meets the criteria for doing so
4. Set a (very small) daily *Energy* allowance for attempting to find nodes within each Spiral bucket
5. After the first run of this routine, attempt to schedule the finding of new nodes for the time of day when it would be cheapest to do so
6. Set a (small) initial *Energy* allotment for attempting to find nodes within each Spiral bucket
7. (MAY) Set a zero *Energy* allowance for attempting to find nodes within Spiral buckets that exist entirely within the molten interior of the *Body* (if it exists)
8. (MAY) Set an extremely small (or zero) *Energy* allowance for attempting to find nodes within Spiral buckets that exist entirely outside of the *Body's* atmosphere
9. (MAY) Decide to reduce the daily *Energy* allowance for a bucket if it continues to fail to find a node within the bucket (exponential back-off)

HMLM Find Global Best HashMatch Routine

1. *HMLM* MUST choose among the first X nodes of its main *HashMatch* index of route-tracked nodes and ask each node individually for its top Y best *HashMatch* nodes.
    a. The local *HMLM* MUST sort these by *HashMatch*
    b. If any of the new nodes would be in the closest k-bucket (by *HashMatch*) of all route-tracked nodes, and aren't already tracked, the *HMLM* SHOULD find 'reasonable' routes to the new node(s)
    c. As in S/Kademlia, the number of nodes in the Global k-bucket best-match set is limited to some reasonably small number (like 256). If a newly found node is a better *HashMatch* than the other limited number of Global tracked nodes, the node SHOULD stop tracking routes to nodes that fall outside the limit
    d. When choosing among the first X nodes, *HMLM* SHOULD choose randomly; weighting by *HashMatch*
    e. *HMLM* MAY consider looking at only a few of the first X nodes in a given round, postponing looking at the other nodes until later rounds
    f. The node MUST choose and advertise values that the node uses to define what it means to be a 'reasonable' route:

i. A cap on the amount of latency per meter
ii. A lower limit on the amount of bandwidth the routes are capable of handling
iii. A minimum *Energy* transfer ability
iv. A maximum *Energy* per meter for data transit
v. A maximum *Energy* for route-endpoint nodes in order to learn route data
vi. A maximum *Energy* for route-endpoint nodes in order to relay data
vii. A required range of word rates
viii. The *MaxWordCount* of *IsoStream* connection support
ix. The size of the *IsoStreamHeader* normalized to bits per meter

g. *HMLM* MAY set a total *Energy* budget for finding nodes and routes for each round
h. The local *HMLM* MUST ask the newly route-tracked node individually for its top Y best *HashMatch* nodes, repeating (a), (b), and (c) until there are no more new nodes (with 'reasonable' routes) to add to the top percentage of route-tracked nodes
    i. If *HMLM* has a *Energy* budget, it should perform a depth first search regarding this repetition

2. *HMLM* SHOULD use some mechanism to forget about old nodes that haven't been heard from in a day or so

Different *HMLMs* could choose different definitions of 'reasonable' routes, which could provide different optimizations for service quality. For example, there could be 'High', 'Medium', and 'Low' latency *HMLMs*: Endpoint nodes could choose to look for new nodes and routes via these different *HMLMs* based on the needs of the endpoint node. It is also possible for an *HMLM* to keep multiple sets of nodes and routes with different service qualities.

Once each day, *HMLM* SHOULD run the *Find Global Best HashMatch Routine*

Once a day, *HMLM* SHOULD run the *Find Spiral Best HashMatch Routine*

13.6.3 Breadcrumbs for HashMatch

One idea to (seriously) reduce the toll of link info distribution on the network, is to require nodes to keep up a full (256?) set of unique multi-path *Breadcrumb* trails to each of the X best global *HashMatched* nodes. This also makes long-distance *μPkts* possible with very little overhead.
This might have scalability issues if none of the long-distance *Breadcrumb* trails are shared. Obvious solutions to this scalability problem require a sharing mechanism that result in logarithmic use of HW *Breadcrumbs* the further away from the source a switch is.

14 Distributed Data Storage

14.1 Immutable Data - Content Addressable Storage (CAS)

The IsoGrid system should distribute data such that it's likely that there is a copy nearby and easy to find. Typically, a CAS is addresses content via its hash, and as such will distribute data very randomly through the network.

In the IsoGrid's CAS, A data set is fully and securely identified by 72 bytes:

1. Total amount of data stored in the set (8 B)
2. Skein-512 hash of data (64 B)

Notably, this construction leads the CAS to only be able to store immutable data: There is no way to store dynamic data in a CAS repository. On the other hand, data in a CAS is self-certifying, making validation easy without a trusted authority.

Any IsoGrid node can offer a CAS repository service in which it holds a data set in hopes of being able to charge for retrieval later. A client might want to ensure the data is still retrievable and provide *Energy* incrementally for such a service. To do so, it can provide *Energy* to the repository to perform a proof of memory, which keeps the repository holding the data for longer time periods. The repository and client could agree to exponentially increase the length of time (at least up to some reasonable point).

Say, a node wishes to publish static webpage data. It would do so by placing the data on a CAS repository nearby the producer. As part of storing the data, the CAS repository MUST also provide for the placement of a *LocatorHash* to itself within a *LocatorHashTree* on the 8 best *HashMatch* nodes in the Key Standard. If the publisher would like the data to be accessible natively in other Key Standards, then the publisher MUST place the data in a CAS repository within each the other Key Standards. A CAS repository MAY support multiple Key Standards at once. The *LocatorHash* values MUST be stored within the *LocatorHashTree* for at least the amount of time payed for by the repository. The node holding the *LocatorHashTree* MUST accept all *LocatorHash* searches for the *Energy* advertised by the node. The node's advertised *Energy* of a *LocatorHash* search is the same for all *LocatorHash* values.
When a client node retrieves the data, it may choose to route that data through a repository nearer to itself. This nearby repository can choose to cache the data for other local nodes to use. The client may also choose to be a repository itself. Whenever a node becomes a repository for a data set, it SHOULD add its *LocatorHash* to the best *HashMatched LocatorHashTree* just like above.

LocatorHashTree is a variant of a binary 'splay' tree that holds a set of *LocatorHash* values sorted by GeoHash. The variance is that the 'splay' operation only occurs when a *LocatorHash* is added to (or refreshed within) the list. A searcher wants to know both recently refreshed and nearby *LocatorHash* values. So searching this list and returning all nodes visited along the search path gives a very convenient mechanism for a first-order sort of 'recent' and a second order sort of 'nearby', getting more nearby and less recent as the list is traversed. Also convenient, is that the retrieved list isn't likely to be very long (as the splay tree is statistically likely to be balanced, and a balanced binary tree provides logarithmic node traversals to get to the leaves).

14.2 Mutable Data – GetNodeInfoFromLocatorHash

Mutable data must be retrieved via the routing system by using the *LocatorHash* to retrieve a *NodeInfo* object that refers to the source of this mutable data. The *LocatorHash* of the server needs to be known prior to retrieving the mutable data. Using the routing system in this way creates a distributed hash table of *NodeInfo* objects. IsoGrid borrows the idea of Self-Certified names from SFS:

SFS addresses remote files like so:

/sfs/<Location>:<NodeID>

where <Location> is the network address of the server, and:

<NodeID> = hash(<Location> + PublicKey)

The name of an SFS file path certifies the server directly. The client can verify the public key offered by the host before securing traffic via a key exchange. SFS instances use a global namespace where name allocation is fully distributed to the endpoint nodes via cryptographic means.

In the IsoGrid implementation of the same idea, <Location> and <NodeID> both exist within the *LocatorHash* of the IsoGrid service node being named. Given that *HMLM* ensures that the node will have an *EccFlow* with all the nodes with similar *LocatorHash* values, this provides logical, secure, and convenient places for the node to store a mutable *NodeInfo* object that describes the server. A storing node MUST validate that this mutable *NodeInfo* object comes from the described node before relaying it to any clients that try to lookup the name; this prevents denial-of-service attacks that try to flood the repository with bogus *NodeInfo* objects.

A *NodeInfo* object consists of the following data:

- NodeAdvertisement
- Maybe all InboundLinkAdvertisements (variable size)
- And/Or perhaps a file index?
- Signature (41,000 Bytes)

TODO: Specify a 'GetNodeInfoFromLocatorHash' SessionProtocolId

15 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, performance, and efficiency 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.

15.1 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 async *EccFlowSession* is created that will be used by the client to send packets via the server node directly to the target IP network (which may have a gateway to the IP Internet). A second *EccFlowSession* 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 *Energy* to be able to send 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.

15.2 Network Management

Initially, the early adopters of the IsoGrid will have to manage their own networking equipment and handle *Energy* 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 data 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.

15.3 Micro-Transfers of Energy Within a Computer

The IsoGrid relies on micro-transfers of *Energy* to fairly allocate the resources required to transmit *μPkts* and streams. There could be an obvious desire to have multiple network services and/or multiple network clients each maintain independent *Energy* ledgers, even when operating in the same PC. One obvious solution to this problem is to implement a software

IsoGrid switch that conforms to the basic IsoGrid open Protocol within a PC. It also seems obvious and likely that such support might grow to include HW acceleration of such a SW switch. Another obvious solution would be to create hypervisor/virtualization of a HW switch.

16 Undefined Higher-Level Services and Protocols

There are several 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 to allow the overall system more flexibility over time.

Distributed Naming

We should separate the two reasons for name resolution: 1) canonical naming for programs to reference other programs, and 2) simple names for humans to communicate addresses to others. For the canonical names, the programmer doesn't need high-value (easy to remember) names, since the software can have the name encoded within it. In this case, the name may include ownership rights, since there's more than enough 256-bit names to go around. This is standardized for the IsoGrid with the concept of the *LocatorHash*.

However, the simple names should not be 'owned'. For example, if I wanted to name myself 'tree', this doesn't mean the rest of the world (or universe) is obliged to refer all those who ask for 'tree' to me. Instead of ownership, those that do the naming should pay everyone else for the privilege of monopolizing that name. Even with this payment, there should not be an 'ownership' right, merely a lease.

The problem of finding services by a simple 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.

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 excellently 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