IsoGrid/IsoMesh Network Protocol Specification

Version 0.200

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

1 Licensing & Legal

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

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

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

The Protocol was released to further the following socioeconomic goals:

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

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

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

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

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

2 Introduction

This proposed specification attempts to meet the requirements of a new network layer protocol with a mesh topology.

2.1 Definitions

Term	Description
Node	A device that interacts with other devices on The IsoGrid using the IsoGrid Protocol
Link	The protocol running on a physical wire or wireless line that connects two adjacent nodes
Switch	A node with multiple links that is able to route data using the IsoGrid Protocol
Word	The atomic unit of transmission across the network. 16 bits of data, 1 bit of parity
Slot	A 1 word wide logical division of the link's bandwidth, delivered isochronously 1 word at a time. Each available slot on a link can be allocated to switch a single connection stream at any given moment.
Input Slot	A slot on an input link. Has a matching output slot on the node on the other end of the link.
Output Slot	A slot on an output link. Has a matching input slot on the node on the other end of the link.
IsoStream	A one way, End-to-end Isochronous stream of words that flows isochronously from a source to a destination across a pre-defined subset of the nodes that comprise The IsoGrid
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

2.2 Layers

There are 4 layers relevant to this specification:

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
- 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 (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 exchange credits so as to allocate scarce resources

Layer 3: Transport (EccFlow) Layer

- Defines how nodes use the network to distribute routing information
- Defines how nodes use the network to safely and reliably communicate with each other

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

IsoGrid vs. TCP/IP Comparison

Layer	TCP/IP	IsoGrid
Application	SSH, FTP, HTTP, etc.	TODO, let's not get ahead of ourselves ☺
Transport	TCP, UDP, etc.	EccFlow
Network	IP	IsoGrid
Link	Point-To-Point, Ethernet, subnet Broadcast, token ring, ATM, etc.	Point-To-Point only, Isochronous USB, ATM, P2P Ethernet
Physical	Any	Point-To-Point only: Communication mediums that have collisions aren't well suited to IsoGrid

2.3 How it works

The IsoGrid is a mesh network that supports routing of one-way isochronous streams (an *IsoStream*). 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 connection. The header that starts a connection defines the length of the *IsoStream*, and how many credits to send per word. The IsoGrid network layer (*IsoStream*) is optimized to support the IsoGrid transport layer (*EccFlow*) which fragments the data, applies a forward error correction code, and sends the data over many paths across the network.

2.4 Protocol Secondary Limitations

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

- Shared links (ex: those with collisions) aren't well suited to be used by the network (too much latency)
- A single *IsoStream* can use no more than the fastest available slot along a route
  - However, an endpoint can create multiple connections such that nearly 100% utilization with *EccFlow* is possible
- Low-rate connections have Higher Latency
- Half-Duplex links not supported
- Streams through nodes that move will have non-trivial buffering/timing requirements
  - The faster it moves, the more demanding the requirements
- TODO: Add more as new limitations are identified

3 Isochronous Word Format

The atomic unit of transmission across the IsoGrid network layer is called a *word*. A word is 16 bits of data, with an additional 1 bit for parity. The parity of a word is defined to be EVEN if all 17 bits of the word have an even number of 1 bits. The parity of a word is defined to be ODD if all 17 bits of the word have an odd number of 1 bits.

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

Valid words meant for non-*IsoStream* 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.
Data	Node Defined	MSG_EVEN	Data sent from one node to the neighbor node using a Link Layer protocol. Not part of an *IsoStream*

IsoStreamInit	IsoGrid Protocol Defined Sent by Source Node	STRM_ODD	These words initialize the payment and initial word count of an *IsoStream*. The structure is specified in this document, and the values are produced by the source node.
IsoStreamRoute	Switch Defined Sent by Source Node	STRM_ODD	These words are defined and advertised by each switch to describe each step of a route that an *IsoStream* is going to take through the network.
IsoStreamHeader	Source Defined	STRM_ODD	These words are sent by the source as a series of headers to the payload of an *IsoStream*.
IsoStreamPayload	Source Defined	STRM_ODD	These words are sent by the source as the payload of an *IsoStream*
IsoStreamFooter	Switch Defined Sent by switch	STRM_ODD	These words are sent by each switch along the route as the final parts of the *IsoStream*. They MUST match the *IsoStreamRoute* words that were originally sent to this switch when the *IsoStream* was initialized.
Lost/Corrupted *IsoStream**	*LostWord* (0x0000)	MSG_EVEN	If an *IsoStreamRoute*, *IsoStreamHeader*, or *IsoStreamPayload* word is lost or corrupted along the route through the network, that word MUST be replaced with *LostWord* to signal this fact to the destination.

4 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 be able to detect the start of an *IsoStream* in a slot. See section ◎ below

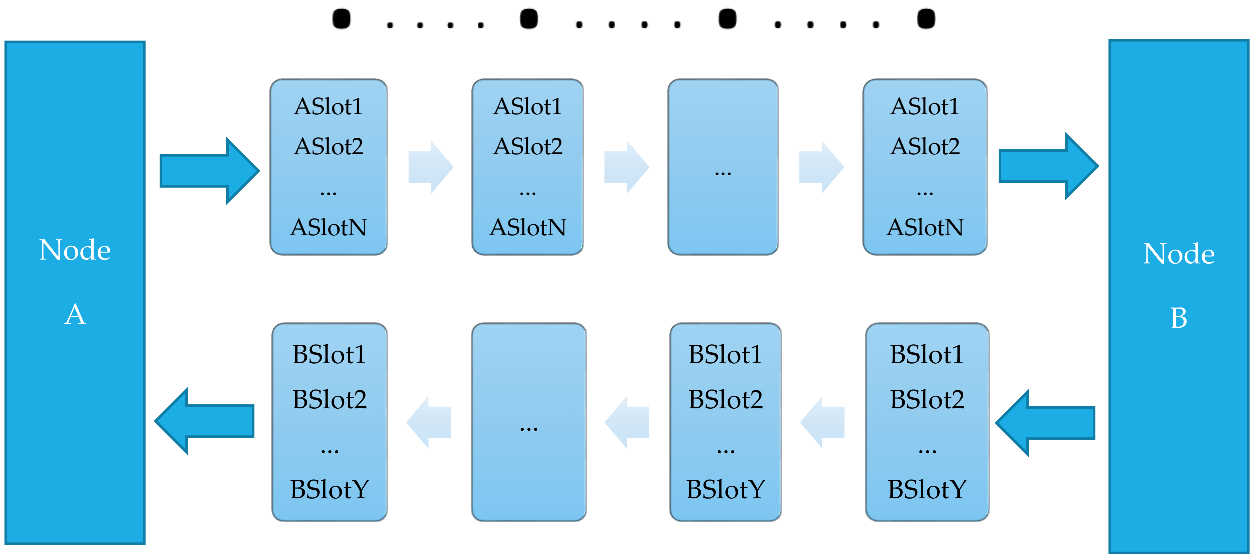

Figure 1. Link Layer frames exchanged between two neighbor nodes

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

4.1 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 / 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 some or 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 some or all of the words aggregated into a smaller number of slots, such that 2^n words arrive for a slot every frame.
- A node MUST be able to identify the slots of a valid frame that arrives on one of its links
- A node SHOULD be able to identify the slots of a valid frame even immediately following missed or corrupt frames

- A node MUST be able to assign an exact total ordering and count to every valid frame it receives
- Electronics MUST be fast and stable enough such that skipping a frame, or somehow seeing two when there was only one, MUST be extremely improbable
- A node MUST advertise the best-case and worst-case word-corruption rates of its output links
- A node MAY use error-correction codes to ensure the error rate meets the advertised value
- A frame MAY contain hashes to determine validity
- A node MUST be able to detect corruption of frames or words that occurs on its input links
    - Bit error(s) detected in a slot assigned to an *IsoStream* MUST be re-transmitted as *LostWord*.
    - 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
    - Y ordered words, Y/2 slots occur every frame, Y half-rate slots occur every other frame
- 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
- Negotiation at any time can dynamically change the size of the frame and/or change the logical allocation of slots

When a slot does not have an allocated *IsoStream*, the sending node MAY use that slot to send non-*IsoStream* data to its neighbor (1 word at a time per slot). If an *IsoStreamInit* arrives on a slot currently being used for non-*IsoStream* data, the node MUST prioritize the *IsoStream* over the non-*IsoStream* data.
Some possible non-*IsoStream* uses include (but are not limited to):

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

- Check credit statistics

Non-*IsoStream* data MAY be ignored/dropped 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.

4.2 Link Slot Synchronized Frequency Standard

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

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

4.2.1 Stationary Nodes

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

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

To illustrate how this might be implemented within a node, here are a few examples:
- Combine all the input waveforms and use that combined waveform to tune the output frequency oscillator
- This waveform feedback could also be a digital process, where the 'fullness' of buffers determines the 'waveform' phase shift.
  - Buffers filling up: Advance the waveform
  - Buffers getting empty: Retard the waveform

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

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

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

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

4.2.2 Feedback-Mediated Link Frame Rate Synchronization

The IsoGrid protocol's concept of using input frame rates to drive output frame rates is referred to as "Feedback-Mediated Link Frame Rate Synchronization" or FMLFRS.
Given a Frame Rate, a Link Latency, and the measure of the clock's short-term stability, it's possible to specify the minimum buffer required to compensate for clock drift and skew.

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

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

$MinBuffer = Rate * Latency * ClockStability$

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

4.2.3 Clock examples

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

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

4.2.4 Dealing with bad clocks

Since most nodes rely on their neighbors to collectively lock to the TCG frame rate, a node with a misbehaving clock will have local impacts. It could potentially cause a group of neighbors to lose link sync frequently. Since this is a local issue, it can be dealt with at the local level by the affected neighbors making the choice to stop using the bad clock as a clock synchronization source. That way, the bad clocked node alone has the consequences of the bad clock.
There is no need for a Time Cop :-)

4.2.5 Moving Nodes

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

5 Credits and Micro-Payments

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

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

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

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

Nodes MAY have different costs for different outbound links. Nodes SHOULD have their node-local credit units be scaled such that the cost of the least expensive link is as close to 1 as possible but not less than 1. If the cost (as expressed in node-local units) is less than 1, then the subtraction step becomes less precise (it MUST be rounded up to the nearest expressible unit).

Nodes MUST offer a rate at which it will exchange credits with each neighbor

Neighbor nodes MAY choose to have a maximum per-diem settlement, to provide a backstop against possible software bugs or security vulnerabilities in the system.

6 IsoGrid Headers

Each *IsoStream* has a sequence of headers that follow the routing envelope. There is no globally required header.
Each header starts with one of two things:

1. An 8 word *ProtocolIdentifier* value, where bits 15, 11, and 10 of the first word are all cleared
2. A 1-word *ProtocolAlias* value, where bit 15 is set

Bits 12:14 of these values indicate the length of the header's data section which follows directly after the *ProtocolIdentifier* or *ProtocolAlias* value:

Bit Sequence	Header Data Length	Comment
000	0	Useful for 'flag' type headers
001	2	
010	8	
011	16	
100	32	
101	64	
110	128	
111	1024	

First recognized Header defines the type of the encapsulated payload.

There is no central authority for assignment of *PrototcolIdentifier* values. Instead, anyone MAY create a cryptographically random 128 bit number, clear bit 15, 11, and 10 of the first word, and use it as a *ProtocolIdentifier*.

A *ProtocolAlias* is used by agreement between the source and destination in place of an actual *ProtocolIdentifier*. This is just to allow for decreased overhead and latency. There is no global requirement that a *ProtocolAlias* maps to any specific *ProtocolIdentifier*, and it's conceivable that some endpoints might not recognize any *ProtocolAlias* values.
The end of the header sequence is marked by a header end word:

Name	Value
HEADER_END_SOWORD	0x8000

6.1 Scenarios

To illustrate the potential usefulness of headers, here are some example scenarios:

- Flagging the method by which a payload is encrypted with Public Key Encryption
- Holding session identifiers to support symmetric encryption
- Describing individual streams of Forward-Error-Corrected data, ie. 16 streams were sent, which one is this?
- Defining the encapsulated upper layer protocol
- Describing how long the payload data is
- Etc.

7 Network Layer: Isochronous Streams

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

Source Sends	Switch A Sends	Switch B Sends	Destination Processes as Payload
*	*	*	*
SO – IsoStreamInit0	*	*	*
SO – IsoStreamInit1	*	*	*
SO – IsoStreamRoute_A	*	*	*
SO – IsoStreamRoute_A	SO – IsoStreamInit0	*	*
SO – IsoStreamRoute_A	SO – IsoStreamInit1	*	*
SO – IsoStreamRoute_B	SO – IsoStreamRoute_B	SO – IsoStreamInit0	No (IsoStreamInit)
SO – IsoStreamRoute_B	SO – IsoStreamRoute_B	SO – IsoStreamInit1	No (IsoStreamInit)
SO – IsoStreamHeader0	SO – IsoStreamHeader0	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
SE - *SetInBandSignal*	ME – *SetInBandSignal*	SE - *SetInBandSignal*	No (*SetInBandSignal*)
SO – Payload2	SO – Payload2	SO – Payload2	SO – Payload2
SO – Payload3	SO – Payload3	SO – Payload3	SO – Payload3
SO – *InBandSignal0*	SO – *InBandSignal0*	SO – *InBandSignal0*	No (*InBandSignal)*
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 – *InBandSignal1*	SO – *InBandSignal1*	SO – *InBandSignal1*	No (*InBandSignal)*
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 – *InBandSignal2*	SO – *InBandSignal2*	SO – *InBandSignal2*	No (*InBandSignal)*
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 – *InBandSignal3*	ME – *LostWord*	ME – *LostWord*	No (*InBandSignal)*
…	…	…	…
SO – Payload79	SO – Payload79	SO – Payload79	SO – Payload79
SO – Payload80	SO – Payload80	SO – Payload80	SO – Payload80
SO – Payload81	SO – Payload81	SO – Payload81	SO – Payload81
SO – *InBandSignal26*	SO – *InBandSignal26*	SO – *InBandSignal26*	No (*InBandSignal)*
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)
*	SO – IsoStreamRoute_A	SO – IsoStreamRoute_A	No (Footer)
*	SO – IsoStreamRoute_A	SO – IsoStreamRoute_A	No (Footer)

*	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 chose to use 2 words for the *IsoStreamInit* sequence, with the *IsoStreamWordCount* set to exactly 128 words. Switch 'A' handles a 3 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. An *InBandSignal* is setup for every 4 words, and starts 3 words after the *IsoStreamSetInbandSignal* word. Switch 'A' inserts its 3 word footer (which matches the *IsoStreamRoute* is stripped off the beginning) at the end of the 128 words it sends. Switch 'B' inserts its 2 word footer at the end of the 128 words it sends. In this way, with an *IsoStream* with 128 *IsoStreamWordCount*, all nodes process exactly 128 words.
Notice in this example, that the link between Switch 'A' and Switch 'B' lost or corrupted Payload9, and the destination receives *LostWord* instead of Payload9. Also, the link between the source and Switch 'A' lost or corrupted InBandSignal3, and thus Switch 'A', Switch 'B', and the destination won't receive InBandSignal3; receiving *LostWord* instead. This rate of loss isn't expected, but shown merely as an example.

For a slot not allocated to an *IsoStream*:

- A word sent with MSG_EVEN parity indicates that the node is sending data directly to its neighbor using some Link Layer protocol
- A word sent with STRM_ODD parity indicates that the sending node is attempting to allocate the slot for an *IsoStream*.
    - This first STRM_ODD word MUST be interpreted as the first word of an *IsoStreamInit* sequence. The sequence MUST be between 1 and 19 words long, depending on the desired precision of the data values.

All attempts to start an *IsoStream* MUST begin with an *IsoStreamInit* sequence.

The *IsoStreamInit* sequence passes along two values:

1. *IsoStreamWordCount*: A floating point value describing the initial maximum number of words to be sent on the *IsoStream*.
    a. Each node MUST remember this value for the *IsoStream* on this slot
    b. Each node MUST begin counting the words of the *IsoStream* with the first word that follows a valid *IsoStreamInit* sequence
    c. This value is used to know when the node should send the *IsoStreamFooter* words.
    d. Minimum expressible value: 16 words
    e. Maximum expressible value: approx. 2^77 words

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

2. *IsoStreamPaymentCredits*: A floating point value describing the number of credits per word to pay for the *IsoStream*.
   a. The switch receiving this value MUST deterministically subtract the exact amount of credits that covers the node's advertised cost per word of handling the *IsoStream*
   b. If, after the subtraction step, the credits would be negative, the switch MUST drop the *IsoStream* initialization request.
   c. If, after the subtraction step, the value is greater than the advertised amount of credits the switch is willing to transfer per word, the switch MUST drop the *IsoStream* connection request.
   d. In all other cases, the switch MUST scale the remaining credits appropriately at the advertised exchange rate, and MUST forward the resulting value in the *IsoStreamPaymentCredits* field of the next *IsoStreamInit* sequence over an available output slot in the appropriate outgoing link.

The following table describes the meaning of the bits in these words:

Word	**Value**	**Parity**
IsoStreamInit0	14:15 (2 bits) (Number of additional *IsoStreamInit* words after IsoStreamInit1) OPEN: If the above is 0, use a very compact format for payment and assume initial word count = 1024. A floating point value representing the number of words that are being paid for: 8:13 (6 bit) exponent, biased such that the minimum expressible value is 16 0:7 (8 bit) a 9 bit multiplier (with an assumed initial 1 'hidden bit')	STRM_ODD
IsoStreamInit1	A floating point value representing the number of credits to transfer to the neighbor per word of *IsoStream*. 11:15 (5 bit) exponent, biased by 0 0:10 (11 bit) a 12 bit fraction (with an assumed initial 1 'hidden bit')	STRM_ODD
IsoStreamInit2	0:14 - 15 bits of additional precision for credits Bit 15 - extra bit (6th) for the exponent	STRM_ODD
IsoStreamInit3	0:10 (11 bits) additional precision for credits	STRM_ODD

	11:14 (4 bits) Number of additional *IsoStreamInit* words after IsoStreamInit5 Bit 15 – extra bit (7th) for the exponent	
IsoStreamInit4	16 bits of additional precision for credits	STRM_ODD
IsoStreamInit5	0:14 - 15 bits of additional precision for credits Bit 15 – Extra bit (8th, and final) for the exponent	STRM_ODD
IsoStreamInit6	16 bits of additional precision for credits	STRM_ODD
IsoStreamInit7	16 bits of additional precision for credits	STRM_ODD
IsoStreamInit8	16 bits of additional precision for credits	STRM_ODD
IsoStreamInit9	16 bits of additional precision for credits	STRM_ODD
IsoStreamInit10	16 bits of additional precision for credits	STRM_ODD
IsoStreamInit11	16 bits of additional precision for credits	STRM_ODD
IsoStreamInit12	16 bits of additional precision for credits	STRM_ODD
IsoStreamInit13	16 bits of additional precision for credits	STRM_ODD
IsoStreamInit14	16 bits of additional precision for credits	STRM_ODD
IsoStreamInit15	16 bits of additional precision for credits	STRM_ODD
IsoStreamInit16	16 bits of additional precision for credits	STRM_ODD
IsoStreamInit17	0:9 (10 bits) additional precision for credits (for a total of 256 bits of precision) 11:15 - RESERVED - MUST be 0	STRM_ODD
IsoStreamInit18	RESERVED MUST be 0	STRM_ODD

TODO: Specify a deterministic subtraction and multiplication method for variable-precision credits.

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

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

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

7.1 Source Routing

The source provides a series of route instructions to be used at each hop (Source Routing).
After an *IsoStreamInit* sequence, the source node MUST stream a series of *IsoStreamRoute* words that define the route, one switch at a time. The number of words consumed by each switch and their meaning MUST be defined and advertised by the switch.
A switch MAY advertise a requirement that the *IsoStreamRoute* be some number of words longer than the *IsoStreamInit* sequence used. This is to allow for low buffering of the *IsoStream*'s words. If the switch accepts an *IsoStreamRoute* word(s) that is shorter than the *IsoStreamInit* sequence, then the switch MUST implement a ring buffer to make up the difference and MUST queue all the future words belonging to that *IsoStream*, which adds to the advertised latency.
If an *IsoStreamRoute* is recognized by a switch, the switch MUST allocate an output slot on the required link to stream the required *IsoStreamInit* sequence. After the *IsoStreamInit* sequence is streamed to the next node, the switch MUST mark the input slot as an *Active IsoStream*.

Let's say there's 1,000 words of *IsoStreamRoute* (500 hops, 2 words each hop)
Effect of 1 word headers on latency at 1Mwps when travelling a 1,000 hop route:
    (2 words / 1,000,000 wps) * 1,000 hops = 2 w / 1,000 wps = 2ms

7.2 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 the *IsoStream*.
When the previously specified *IsoStreamWordCount* is reached, the switch MUST send the required *IsoStreamFooter* words, matching the exact number and value of *IsoStreamHeader* words that were consumed by this switch in the *IsoStreamRoute* sequence that activated this *IsoStream*.

7.3 In-band Signals

An *IsoStreamSetInBandSignal* (MSG_EVEN) announces to all switches along a route the initial timing and frequency of the *InBandSignal*. The *IsoStreamSetInBandSignal* word defines:

- The timing for the initial *InBandSignal* word
- The separation (in words) between each subsequent *InBandSignal* word

All IsoGrid switches MUST note these two values and decode future *InBandMessages* within the *InBandSignal*. The destination endpoint MUST note these two values such that it can distinguish the *IsoStream* words from the *InBandSignal* words. An *InBandMessage* within the *InBandSignal* COULD target a single switch, many switches, or all switches. It doesn't seem useful to have an

InBandMessage target the destination endpoint; that's what the *IsoStream* is for. An *IsoStreamSetInBandSignal* that arrives while an existing *InBandSignal* is active MUST be ignored: There can only be one *InBandSignal* within an *IsoStream*. Each *InBandMessage* MUST have a single IsoGrid Header. An *InBandMessagePreamble* word MUST be sent before each *InBandMessage*. *InBandMessage* words MUST have STRM_ODD parity. The *InBandMessagePreamble* word has MSG_EVEN parity. If a switch recognizes an *InBandMessagePreamble* word within the *InBandSignal*, it MUST read the next word of the *InBandSignal* as the first word of the next *InBandMessage*. A valid *InBandSignal* can only begin with a MSG_EVEN word: If the first word of the *InBandSignal* has STRM_ODD parity, the node MUST assume that the *InBandSignal* has ended (or never existed).

Name	**Value**
ISO_CONN_SET_IN_BAND_SIGNAL_MEWORD	A floating point value describing the period of the *InBandSignal*: 0:2 (3 bit) a 3 bit fraction (assumed initial 1 'hidden bit' if all zeros) 3:6 (4 bit) exponent, biased by 3 (such that the minimum expressible value is exactly 1) 7:11 (5 bits) exponent, no bias This value describes the number of *IsoStream* words between this word and the first *InBandSignal* word A value of zero means that the next word is the first *InBandSignal* word. 12:15 (4 bits) - BITWISE_NOT([0:3] XOR [4:7] XOR [8:11])
IN_BAND_MESSAGE_PREAMBLE_MEWORD	0xFFFF
IN_BAND_SIGNAL_NO_MESSAGE_SOWORD	0x0000

All IsoGrid switches MUST support decoding the following InBandMessages:

- *GetIsoStreamUtilizationFactor*
- *ChangeIsoStreamRate*
- *SetNewIsoStreamWordCount*
- *EndInBandSignalNow*
- *EndInBandSignalFuture*

7.3.1 GetIsoStreamUtilizationFactor

TODO: Used by higher level protocols to determine the utilization factor and hop count of the most congested and least congested link along a route: 0xFFFF means fully utilized, 0 means it's nearly unused.

7.3.2 ChangeIsoStreamRate

TODO: This may be too hard to implement.

This *InBandMessage* allows source nodes to set a future point where the switch MUST modify the word frequency of the *IsoStream*. This message MUST be ignored if it's not immediately preceded by an *InBandMessagePreamble*. The message has a 4 word payload. The first word MUST contain a simple 16-bit integer defining the number of words of *InBandSignal* that will follow the end of this *InBandMessage*. The third word MUST contain the new word rate. The second and fourth word MUST duplicate the first and third (respectively), and MUST be ignored otherwise. Using multiple *ChangeIsoStreamRate* messages allows for redundancy to best ensure that both endpoints and every switch along the route agree on the precise word where the rate changes.

7.3.3 SetNewIsoStreamWordCount

This *InBandMessage* sets a new *IsoStreamWordCount*. This message MUST be ignored if it's not immediately preceded by an *InBandMessagePreamble*. This message allows source nodes to start off with smaller *IsoStream* word counts, and then dynamically increase them if they continue to work well. A *SetNewIsoStreamWordCount* with a word count shorter than the previous word count MUST be ignored.
All IsoGrid switches MUST recognize 0x9000 as a *ProtocolAlias* of {0x1000, 0xDC30, 0x4396, 0x11E5, 0xB970, 0x0800, 0x200C, 0x9A66} which refers to this *SetNewIsoStreamWordCount InBandMessage*. The first word of the 2 word payload MUST be a normal *IsoStreamWordCount*, but with an extra two bits for the exponent (total 8), leading to the maximum expressible word count being roughly the number of atoms in the observable universe. NOTE: Perhaps 7 bits for the exponent is enough, and using the extra bit for the fraction is a better use of the bit?
The second word MUST be identical to the first.
TODO: Can this use a better FEC?

7.3.4 EndInBandSignalNow

This *InBandMessage* allows source nodes to end an *InBandSignal*. This message MUST be ignored if it's not immediately preceded by an *InBandMessagePreamble*. The message has no payload. If a switch decodes this *InBandMessage*, it MUST stop decoding this particular *InBandSignal*.
All IsoGrid switches MUST recognize 0x8000 as a *ProtocolAlias* of {0x0000, 0x305F, 0xEFD7, 0x4598, 0x934E, 0x9A0E, 0x5544, 0xD954} which refers to this *EndInBandSignalNow InBandMessage*. This message is fast and efficient, but possibly risky on noisy links.

7.3.5 EndInBandSignalFuture

This *InBandMessage* allows source nodes to set a future end to the *InBandSignal*. This message MUST be ignored if it's not immediately preceded by an *InBandMessagePreamble*. The message has a two word payload. The first word MUST contain a simple 16-bit integer defining the number of words of *InBandSignal* that will follow the end of this *InBandMessage*. The second word MUST duplicate the first, and MUST be ignored otherwise. Using multiple *EndInBandSignalFuture* messages allows for redundancy to best ensure that both endpoints and every switch along the route agree on the precise word where the *InBandSignal* ends.
All IsoGrid switches MUST recognize 0x9001 as a *ProtocolAlias* of {0x8001, 0xEFBD, 0x26E2, 0x4AD7, 0x9C74, 0x19B7, 0xF9E3, 0xEA59} which refers to this *EndInBandSignalFuture InBandMessage*.

7.4 Packets

TODO: Perhaps packets aren't really necessary now that we have *EccFlow*?

The global IsoGrid standard has a practically infinite packet size limit and Maximum Transmission Unit (MTU) of ~2^136 words.

Packets CAN be sent over a normal *IsoStream:* This is referred to as an *IsoPacket*. This method requires that:

1. The sender MUST have an Isochronous Link available over which it can send the *IsoPacket*
2. The sender MUST have available credits to pay for the *IsoPacket* to reach its destination
3. The sender MUST know the full isochronous path to the destination prior to sending the *IsoPacket*

*IsoPacket*s have the same low latency as a normal *IsoStream*.

Packets MAY be sent from node to node using any Link-Layer protocol. Links MAY use Geo-Location routing, offer bounties for successful delivery, or other routing strategies. These methods have larger latency, but don't have the strict *IsoPacket* requirements and might be useful for edge nodes without access to dedicated Isochronous Links (like Internet-of-Things devices and sensors).

8 Transport 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 transport layer protocol (*EccFlow*). However, 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 a set of *EccStreams*. Each *EccStream* is presented to the application layer as a series of packets.

8.1 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 you segment the data evenly, that just increases the likelihood that a link failure will cause data loss. However, 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. 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.

8.2 Stream Options

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

Async sub-option	**Description**
Reliable Ordered	Lost packets are resent, and later packets are held in a destination buffer until the lost packet arrives.
Reliable Unordered	Lost packets are resent, and delivered out of order
Unreliable Raw	All recognizable packets are delivered to the application layer, even if they contain bad data (and don't request resend)
Unreliable Dropping	If enough IsoConn streams lose data, drop the packet (and don't request resend)

Isochronous Streams:

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

Asynchronous Streams Flow Control:

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

Other EccFlow Responsibilities:

- Building up and tearing down streams to try to keep the buffer no more than half full
- Deciding the cost/benefit of additional ring buffer vs. paying for additional link redundancy
- Maintaining a mutually acceptable credit balance between the two nodes

8.3 Congestion Control

TCP contains Congestion Avoidance and Fairness algorithms. Instead of relying on endpoints implementing a 'fair' TCP (which breaks down with malicious nodes), IsoGrid requires endpoints to pay for their use of the *IsoStream* (network) layer. So *EccFlow* doesn't have fairness algorithms.
IsoGrid doesn't exhibit the problem of congestive collapse, because active *IsoStreams* are able to use their slot even in the presence of 100% load, and new *IsoStreamInit* requests MUST be dropped unless they have a higher priority. Higher priority *IsoStreamInit* 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).

8.4 EccFlow Initialization

Initially, data MUST be sent in blocks of 111w (222B).
Minimum number of EccBlocks is 7 (for a total minimum useful size of 665w)
Generate 7x 16w (256bit) cryptographically random numbers, use #1 as a symmetric key to run an AES-GCM Authorization/Encryption of the data (do not send this symmetric key on a network). Place the 6 others into a list called KeyPart.
XOR all 6x numbers in KeyPart together to produce a 7th 16w number, which should be inserted randomly into the KeyPart list (making a total of 7 in the list).
Generate a 96bit (6w) IV for the algorithm. The IV is used as the SetID for the *InitialErrorCorrectionCodedFlow*.
Send numbers 2-9 (8 total) along with the ciphertext, like so in the following *EccBlocks*:

1. KeyPart[0] is sent in words 0-15
2. KeyPart[1] is sent in words 16-31
3. KeyPart[2] is sent in words 32-47
4. KeyPart[3] is sent in words 48-63
5. KeyPart[4] is sent in words 64-79
6. KeyPart[5] is sent in words 80-95
7. KeyPart[6] is sent in words 96-110 & 0

The first 7 EccBlocks each contain 95w of ciphertext. Any remaining EccBlocks contain 111w of ciphertext.

InitialErrorCorrectionCodedFlow: Encoded first with *EccBlock* and then using Reed-Solomon (255,223) on each *EccBlock*.

- SetID (6w) - The unique IV of the EccFlow algorithm
- PartID (1B) PartCount (1B)
- CodeBytes (1B per part)

The first byte of every *EccBlock* given to the Reed-Solomon encoder/decoder is hardwired to 0.
Desired redundancy can be targeted between 0-32 parity bytes per block.
The *EccFlow* implementation MUST try to send different parts down different routes with as few duplicated nodes between routes as is cost effective.
The service MUST use the same encryption key to encrypt reply blocks. For the replies, it MUST use the same IV except the MSB is flipped. Implicitly, all replies go down the *EccFlow* in the reverse direction.
The *EccFlow* between two nodes constitutes an arbitrarily long term bi-directional session.

8.5 Denial of service attack against EccFlow

An attacker could notice an *InitialErrorCorrectionCodedFlow* and start spoofing SetID, PartID, and PartCount with invalid parity bits; this could end up confusing the decoder to the point where it gives up.
Mitigation: Once an *EccFlow* is initialized, the server will only accept new *IsoStreams* as being part of the *EccFlow* via privately agreed random-looking tags. So the attack is only feasible during the initial setup of the *EccFlow*. Trying to block all initial communication to a server can be made arbitrarily expensive (by advertising a higher cost, and dropping requests that don't pay enough).

9 Path Determination

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

This section covers the standardized *LinkAdvertisement* protocol that facilitates node and link advertisements.

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

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

When a node decides to change one or more links, the node SHOULD send the updated *LinkAdvertisement* to each remote node that it knows how to reach for a reasonable cost.

NodeAdvertisement:

- NodeID (8w)
- 3D Cartesian Geohash (12w)
- Optional Text location (32w)
- Maximum supported IsoConnInitX (1B)
- Service Declaration Count (4w)
- Service Declarations: (8w + 8w + 2w + 6w each [ServiceID + Tag + Cost + ServiceSpecificData])
    - LinkAdvertisement
    - InitErrorCorrectionCodedFlow
    - ErrorCorrectionCodedFlow (resume)

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

LinkAdvertisement:
NodeID (8w)
LinkCount (4w)
LinkN:

- Tag (8w)
- Lowest Supported Rate (1B) Highest Supported Rate (1B)
- Review the Slot Isochronous Standard for other requirements
- Maximum *IsoStreamWordCount* (1w)
- Minimum *InBandSignal* period, expressed just like in *IsoStreamSetInBandSignal* (1B)
- Count of buffer size, expressed as an exponent (1B)
- Minimum supported WordCount(*IsoStreamRoute*) - WordCount(*IsoStreamInit*) (1B)
- Worst Case Latency (2w)
- Maximum credit transfer per word (2w)
- Link cost (2w)
- Exchange Rate (2w)

NodeAdvertisement of DestinationN (variable size)

Later Maybe add:
Beginning valid time (4w)
Expiration time (2w)

9.1 3D Cartesian Geohash

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

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

9.2 Scalability Analysis

The above node and link advertisement plan are likely to work for a long time (while contiguous IsoGrid node counts are low). Each node is likely to be able to keep up with 100,000 nodes or more. If the number of contiguous nodes on an IsoGrid rises high, or the cost of sending advertisements rises too high, an additional protocol will need to be developed. Given the economic incentive to using such a protocol, it is likely to arise soon enough.

If a truly distributed solution can't be found/implemented, an overlapping/redundant decentralized solution should suffice and is likely to be able to avoid any negative socio-economic effects. For example, nodes could use the 3D Geohash to ask one or more remote routing champion nodes, which have specialized in finding the optimized remote routes near a specific location. Asking multiple champion nodes at random makes cheating much harder.

TODO: Specify in more detail a decentralized or distributed solution with no negative socio-economic effects.

10 Bootstrapping Plan

In the early stages of implementing the IsoGrid, there won't be any widely-deployed native-IsoGrid services. The only practical use of the IsoGrid during the early stages would be as a gateway to the existing Internet: The IsoGrid Protocol Stack will need to act as a reasonably inexpensive alternative to traditional Internet Service Providers. Two of the biggest costs of traditional Internet Service Providers are: 1) infrastructure for connectivity over the last-mile to

customers, and 2) customer acquisition costs. With a local IsoGrid, last-mile infrastructure is provided by the customer. Customer acquisition costs might be significantly reduced because the IsoGrid is likely to spread from neighbor to neighbor by word-of-mouth, precisely for the purpose of getting cheaper internet access. The early adopters are likely to be willing to start the IsoGrid before financial benefits are clear, due to being dissatisfied with their existing ISP options (or lack thereof). Once a small IsoGrid is started, the connected participants have a strong financial incentive to connect up more of their neighbors.
Once a significant portion of the population of developed countries start using the IsoGrid for Internet access, the hardware and software costs will get much lower. Having a mesh topology allows for implementations with very simple initial setup and maintenance. When these begin to appear, the IsoGrid will spread to developing countries, providing cheap, scalable, and dependable connectivity to the world.

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

10.2 Network Management

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

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

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

11 Undefined Higher-Level Services and Protocols

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

Low Latency Game Streaming

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

Distributed Storage

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

Alarm Systems

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

IsoGrid Internet Service Providers

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

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

EccFlow to Data-Center based personal VPN

A client that has created many *IpVpn* sessions to various dual-homed service 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.

Link Tunnels

Tunneling an IsoGrid link across the IsoGrid between distant nodes should be efficient, with low overhead. This could make for quicker, easier, and more reliable long distance *IsoStream*. This effectively reduces the hop-count on an *IsoStream* that traverses the tunnel. Should be easy to layer this on top of an *IsochEccStream* within an *EccFlow*.

IP Transit

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

Large Scale, High-Precision Timing

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

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