Pest

S. Datskovskiy

Version 0xFA

Contents

1 Why Pest?

Pest is a *peer-to-peer* network protocol for secure real-time communication[1] between mutually-consenting parties. It is designed for decentralization of control, obstruction of eavesdropping and traffic analysis, resistance to natural and artificial interference, and mechanical simplicity – in that order.

1.1 Exodus from IRC, and Resisting "User Domestication".

Pest was originally devised in reaction against the odiously centralizing design of IRC. An IRC *relay* is typically inhabited by a multitude of casual users, who cannot communicate directly[2] with one another, and interact via the relay strictly at the mercy of a small group of administrators. The latter typically oversee a network of such relays, and determine which users may log in, use particular *handles*, create and manage particular *channels*, etc. and may temporarily delegate this authority to others.

The typical outcome of any conflict between IRC users and administrators is the expulsion of the former from the network. Even when a dispute reaches a boiling point and users begin to "emigrate" en masse, any refuge they may choose is quickly found to resemble the place from which they escaped – whether they move to another IRC network, or start a new one where the "rebels" immediately take their own turn as "tyrants" over a fresh crop of powerless users. And often enough, the escapees take to using commercial chat services – dispensing with IRC's threadbare imitation of decentralization entirely.

Interestingly, IRC suffers from a design defect which makes true decentralization impossible even if every user were to operate a personal relay: the protocol's intolerance of cyclic routing requires relays to be arranged in a *tree topology*, where every well-connected participant is a *central point of failure*. IRC offers relay operators no defense against *denial-of-service*[3] attacks; but even in the absence of attacks, its acyclic networks are fragile. Their frequent *net splits* make reliable logging of public conversations difficult, further helping commercial chat service providers to lure away IRC users.

On top of this, IRC makes no provisions for secure end-to-end communication, or in fact for any kind of censorship resistance whatsoever – the protocol trivially lends itself to detection and inspection en route, and ISPs have been known to block it. Commercial chat services increasingly market themselves with misleading, or outright *false* (but impressive to nontechnical audiences) claims concerning privacy and censorship resistance. In every case, their actual objective is *user domestication* – any commercial entity's claim to decentralization[4] is nothing more than *bait* for the gullible and the desperate.

[1] At the time of this writing, there are two prototypes which implement IRC-style chat. Planned extensions include file sharing and WWW hosting.

[2] Outside of special cases, e.g. DCC file transfer, a kludge which still requires relay mediation.

[3] IRC, like *every* other protocol built on top of TCP, is inherently vulnerable to DDOS.

[4] Or, more egregiously still, of defying governmental snoops.

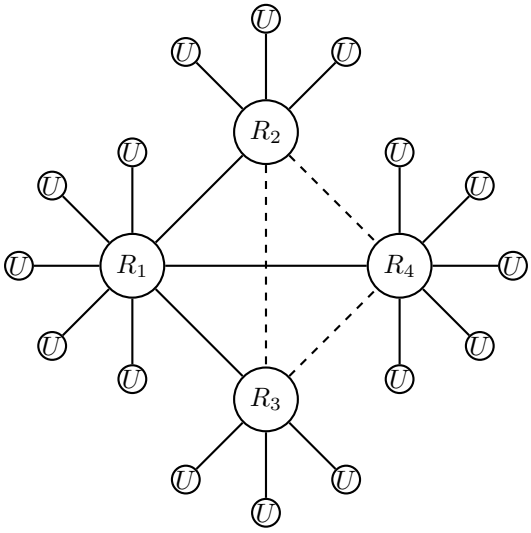


Figure 1: An IRC net with 4 relays and 16 users. Dashed lines represent prohibited cyclic connections. Failure of R_1 would split this net into 3 islands.

1.2 How Pest Differs from IRC and Other Chat Protocols.

In contrast to an IRC relay or a commercial chat server, a Pest *station* is inhabited by exactly *one* user: its *operator*. It sends and receives authenticably-encrypted UDP packets, communicating exclusively with an operator-selected set of *peer* stations known as its WOT (*web of trust.*) Each of these stations, in turn, may have its own set of peers, *ad infinitum*, forming a fully-decentralized *net* with unrestricted connection topology.

Pest does not rely on *any* centralized Internet services (in particular, it does not use DNS, NTP, or the *master-keyed* pseudo-cryptography of SSL) and offers resilience to communication disruptions, as well as compatibility with ad-hoc alternatives to traditional Internet connectivity.

The simplicity of the Pest protocol allows for multiple independent, interoperable implementations. Writing a working Pest client *ab initio* does not require knowledge of university-level mathematics. Neither does it demand the use of bulky (and invariably bug-ridden) cryptographic libraries.

1.3 Pest Nets.

Pest stations organize into *nets*. A net is formed by a group of station operators with a common interest. An operator who wishes to join a net must peer with *at least one* existing station in that net. Nets may easily and organically combine

into larger nets, or, on the contrary, undergo schismatic splits, whenever the individual station operators so desire.

1.4 Identity is Decentralized.

A prospective Pest station operator does not need to ask permission from *anyone*; and the only other people who will need to know about the mere existence of the station will be its *peers*. To join a Pest net, an operator must simply find one or more current members of that net who would like to peer with his station, and securely exchange[5] a small amount of information to establish the peering. An active Pest station may have as few as *one* peer, or as many as its hardware is able to service at the desired bandwidth capacity.

A Pest station operator may choose any *handle* he likes, so long as it does not collide with that of a peer. Importantly, one person may easily operate *multiple* Pest stations, and inhabit multiple *disjoint* nets; and may use, if he wishes, a different handle on each net.

1.5 Station Operator is Answerable Only to Peers.

Unlike `IRC` or commercial chat services, Pest does not impose a hierarchical structure of control, and therefore offers no direct equivalents to `IRC`'s *"kick"* and *"ban"*. Instead, an annoying, tedious, or habitually-spamming station operator may be rebuked by his peers; if he persists in his misbehaviour – *ignored* via Usenet-style *killfiles*; and, if he proves incorrigible – *unpeered*. Sooner or later, the malefactor will find himself where he belongs: either alone or in the company of his own kind.

1.6 Unrestricted Network Topology.

Pest broadcasts are *flood-routed* – i.e. they traverse *all* available propagation paths. Unlike `IRC` relays, Pest stations may be connected in any topology their operators prefer; concretely, loops are permitted, in the interest of decentralization. *Packet storms* are prevented via *deduplication* at the station level – rather than by prohibiting loops, or via a *spanning tree protocol*, or any other traditional routing schemes which enforce acyclic connection graphs by demanding the existence of "root nodes", "supernodes", centrally-imposed precedence tables, etc.

Pest station operators are not merely permitted, but in fact encouraged to form richly cyclic connection graphs for the highest attainable resiliency.

[5] Through a secure communication channel external to Pest – e.g. via `GPG`-encrypted mail, or over "sneakernet."

1.7 Connectionless and Medium-Agnostic.

Every Pest `Message` travels in an individual authenticably-encrypted packet. The receiver of a packet determines the identity of its sender strictly by attempting `Seal` verification against *each of the `Key`s in his `WOT`*, in random order[6]. If and when a `Seal` verification succeeds, the packet is attributed to the peer whose `Key` verified it, and the `Message` is decrypted and processed.

A Pest station is able to quickly reject malformed, spurious, corrupted, or duplicate incoming messages. At the same time, the address fields provided by the IP protocol **are not used** in the validation of a Pest packet. Consequently, Pest traffic may be easily and safely carried over an otherwise-unsecured and "addressless" *shared-everything* medium, such as radio, or via several conventional Internet connections operated in tandem[7].

Pest `Message`s carry hashes of their predecessors to enable detection of loss in transit and allow for retransmission requests. As a result, Pest is insensitive to connectivity disruptions (whether planned, accidental, or malicious), and Pest traffic may be carried reliably over lossy channels (radio, poor-quality or short-lived Internet connections, etc.)

[6] To interfere with a snoop's attempts to infer packet senders' identities – or to distinguish the act of receiving a *valid*, rather than *bogus* packet – via traffic analysis.

[7] This enables – among other things – an arbitrary degree of resistance to `DDOS`, even on a miserly budget.

2 The Philosophy of Pest: The Three "Nothings".

Pest is designed around the intent of "cutting off the oxygen" to three species of vermin: the stranger – who brings DDOS and spam; the snoop – who steals "anything not bolted down"; and the snitch – who betrays trust.

2.1 "Nothing to the Stranger."

From a Pest station's point of view, a **stranger** is *any Internet-connected machine other than a current WOT peer*. This category includes *former* peers, as well as members of Pest nets disjoint from the given station's. Unlike virtually all traditional Internet protocols, Pest *does not talk to strangers*. At all.

Strangers may, of course, *send* arbitrary packets to anywhere at all – including a Pest station. However, because a stranger (by definition) does not possess any of the Keys in that station's WOT, all such packets will be deemed *bogus* and *immediately discarded*. A bogus packet will not trigger *any kind* of response from a Pest station.

A stranger who somehow comes across a packet previously sent to a Pest station by one of its peers, and retransmits copies of it to that station (*"replay attack"*) will not succeed in flooding the station or its net with garbage or exhausting the station's machine resources: such a packet will be inexpensively determined to be a *duplicate* or *stale* and discarded as *bogus*.

A *bogus*[8] packet is one which the receiving station identifies as *martian*, *malformed*, *duplicate*, or *stale*. Such packets are *silently discarded*. Conversely, packets which are found to be neither *martian*, *malformed*, *duplicate*, nor *stale* are considered *valid*, and will be processed by the station in the order in which they were received.

2.1.1 "Martians".

An incorrectly-sized[9] packet received by a Pest station – or a correctly-sized one which does not bear a valid Seal from one of its peers is referred to as a *martian*. All such packets are silently discarded.

2.1.2 Malformed.

An incoming packet carrying a Message which violates a formatting rule is referred to as *malformed*. All such packets are silently discarded.

[8] Bogus packets do not necessarily come from strangers – they may be sent by peers; at times – deliberately.

[9] I.e. of length above or below 496 bytes.

2.1.3 Stales.

Any packet bearing a `Message` which has expired, or appears to come "from the future" is deemed stale and – even though it may be valid in every other respect – silently discarded[10].

2.1.4 Duplicates.

An incoming packet which is found to be neither *martian* nor *stale* may be deemed a **duplicate** if the `Message` it bears is found to be identical to one encountered in any *previously-received* valid packet. Duplicates are silently discarded.

[10] Unless the station is currently expecting a `GetData` response, and the message's hash is found to equal the requested one.

2.2 "Nothing to the Snoop."

A stranger who is able to capture some or all of the packets entering or leaving a Pest station is referred to as a **snoop**.

2.2.1 No Plaintext Fields in Pest Packets.

Pest packets traveling between stations contain no unencrypted information *whatsoever*[11] – and in particular, feature no *"magic numbers"* or other fields with meaningful or predictable values of any kind which could identify them to a third party as Pest packets or reveal any information at all concerning their `Payload`s to anyone lacking the requisite `Key`.

Consequently, a Pest packet intercepted en route *conveys no useful information* to a snoop, apart from the mere fact that a particular machine had sent a string of 496 apparently-random bytes[12] to a certain other.

2.2.2 Chaff.

In the interest of thwarting traffic analysis, a Pest station will occasionally transmit rubbish packets – indistinguishable, to a snoop, from other Pest traffic – to peers, or even to randomly-generated IP addresses.

Additionally, a station *may* occasionally transmit copies of a packet keyed for a given addressee to one or more *randomly-selected other peers*, in *random order*. All recipients other than the intended addressee will harmlessly reject the *martian*; while the task of a snoop charged with determining "who is talking to whom" becomes rather unenviable.

[11] Pest `Seal`s could be considered "plaintext" information, but are computed strictly *over* ***`Ciphertext`***, using a peer-unique `HMAC` *sealing key* separate from the peer's *cipher key*; and therefore reveal nothing to a party not in possession of the requisite sealing key.

[12] A network of Pest peers may, by mutual agreement, pad their packets beyond the default size, to circumvent a (hypothethetical) ISP ban against 496-byte packets. However, steganographic techniques, necessary as they may one day become, are beyond the scope of this document.

2.3 "Nothing to the Snitch."

> *If you give me six lines written by the hand of the most honest of men, I will find something in them which will hang him.*
>
> *Cardinal Richelieu.*

In the context of Pest, a **snitch** is a *traitor* or *infiltrator* who would divulge to a third party some information he had been given in confidence by one of his Pest peers – in an attempt to implicate the latter in a *"thoughtcrime"* or scandal, or otherwise tarnish his reputation.

2.3.1 Pest Messages are Authenticable, but not Opposable.

All Pest `Message`s are *authenticable* – a station will only process an incoming message if it carries a valid `Seal` (i.e. signature) from a peer. However, they are also *repudiatable* (i.e. *non-opposable*). `Seal`s are produced using *symmetric cryptography* – a peering `Key` is *exactly the same on both sides* of a peering. Therefore, a *snitch* cannot, at any point in time, prove to anyone that he was not *himself* the author of a message he may claim to have received.

2.3.2 Caveats.

- Even though all traffic between Pest peers is encrypted, a Pest *broadcast* should usually be thought of as *public speech* – its originator can have no certain knowledge of where it may eventually propagate to. You may know your peers well, but how well do you know all of *their* peers? The same is true of a *direct* message sent to someone with whom you do not have a strong relationship of trust.
- A Pest *hearsay* should be thought of as a *rumour* – it may originate from virtually *any* participant of a given Pest net, and may claim to have been authored by virtually anyone.
- A traitorous or incompetent peer may expose your shared peering `Key` to third parties, and consequently his side of the peering may begin to suffer from "multiple personality disease". Do not hesitate to *unpeer* him!
- Peering `Key`s are only as secure as the devices they are stored in. Remember that anyone who succeeds in stealing a `Key` will be able to impersonate *either side* of the peering defined by that `Key`. Avoid exchanging Pest keys in public places or via unsecured communication channels. (Use e.g. `GPG`.) Beware of shared hosting services which may expose your keys to snoops.
- If you end up revealing a *genuinely valuable secret* to a traitor, the gods themselves cannot save you!

3 Pest Station Basics.

The *operator* of a Pest station – who may be a human or a bot – has absolute control of the station and its configuration. A station communicates *exclusively* with:

1. The operator – via the *operator console*.
2. An operator-selected set of remote *peer* stations – via ciphered and signed `UDP` packets.

3.1 Peers and Keys.

In order for a pair of Pest stations to communicate, their operators must decide to *peer* them by agreeing on a shared secret `Key`. Every packet sent by one peer to the other is enciphered and signed by the sender, and will be verified and deciphered by the receiver using this `Key`.

Additionally, the peers must establish one another's reachable `Address`es. If one or both of them has a routable, static public IP, this may be accomplished via the `AT` Command; otherwise it will take place automatically, supposing both of the peers are able to reach a given net.

If, at some future time, the operator of either station no longer wishes to continue in this relationship, he may terminate the peering unilaterally, and the two stations will then be said to have *unpeered*. A former peer is treated exactly the same as any other *stranger*.

3.2 The `WOT`.

A Pest station may have any number of peers. One or more[13] known `Key` for each peer is kept in a data structure referred to as the station's `WOT`. The operator may alter this structure at any time, and changes take effect *immediately*. The `WOT` is *never* altered by the station *except* by direct command of the operator. Each peer entry in the `WOT` also contains one or more `Handles` known to be in use by the peer.

3.3 The `AT`.

A Pest station has another data structure, the `AT` (*Address Table*), which holds the last known reachable `Address` of each `WOT` peer. The `AT` is used *exclusively* for determining where to send outgoing packets.

[13] Multiple `Key`s associated with one peer are permitted. This is convenient when phasing out an old `Key` in favour of a new one. The converse (the use of one `Key` for multiple peers) is prohibited. When addressing outgoing packets to a peer for whom multiple `Key`s are known, the one which validated the packet most recently received from that peer is to be used.

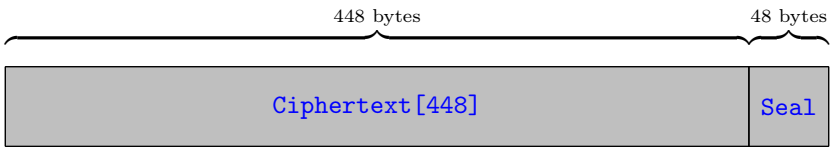
Like the WOT, the AT may also be altered by the operator at any time. Unlike the WOT, the AT is *automatically updated* by the station when a packet attributed to a given peer is received from a *new* (i.e. not currently in the AT entry for that peer) Address. The AT holds *one* entry per WOT peer.

4 The Pest Transport Protocol.

4.1 Packets.

4.1.1 Black Packet.

All Pest network traffic without exception consists of *black*[14] packets. Every black packet is precisely 496 bytes long[15], and consists of a 448-byte `Ciphertext` followed by a 48-byte `Seal`:



Such a packet is easily authenticated and deciphered by its intended addressee, but entirely meaningless to snoops – who, by definition, lack the `Key` against which the `Ciphertext` and `Seal` were created, and therefore cannot decipher or authenticate the packet, distinguish it from random noise, or craft a plausible – in whole or in part – spurious replacement.

4.1.1.1 `Ciphertext` in a black packet is produced from a 448-byte `Plaintext` via the `Serpent` cipher, keyed to the `Ciphrator` component of a `Key` known to both the sender and the addressee. The `Plaintext`, in turn, always consists of a 16-byte random `Nonce` followed by a 432-byte `Payload` – also referred to as a *red packet*:

Given as `Serpent` is used in the *Cipher Block Chaining* mode of operation, the `Nonce` provides a reasonable guarantee that no two black packets emitted by a Pest station will ever be identical, or similar in any meaningful respect, irrespective of their `Payload`s.

4.1.1.2 `Seal` in a black packet allows the addressee (and *only* the addressee) to uniquely identify the sender and verify the integrity of its `Ciphertext`. It is generated via `HMAC-384`, keyed to the `Sealer` component of the addressee's `Key` and computed over the `Ciphertext`.

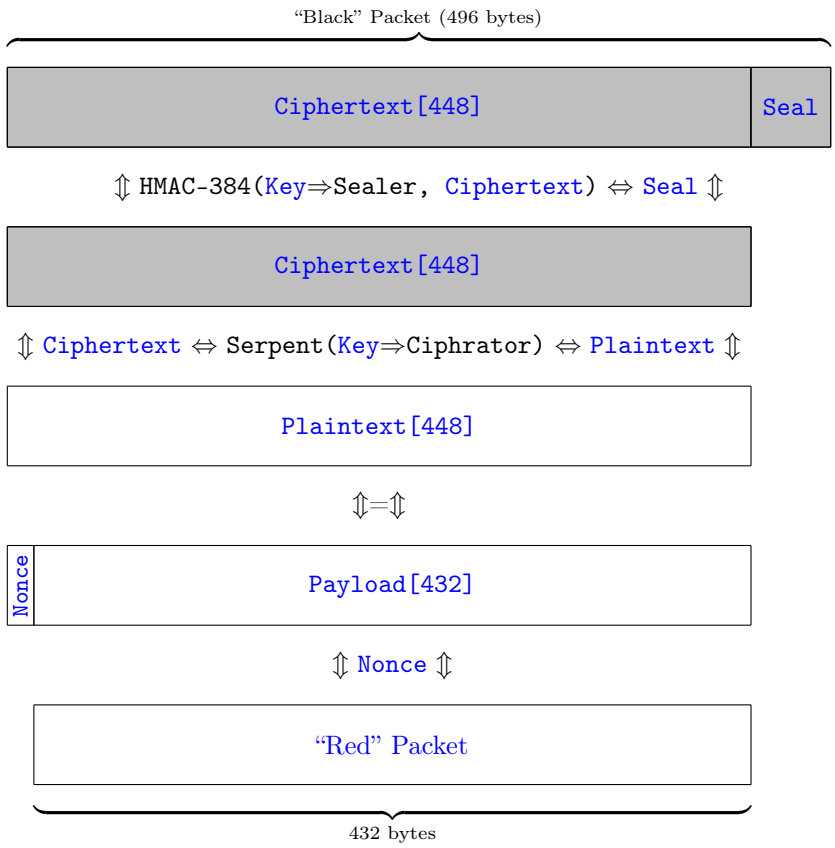
[14] Convenient shorthand borrowed from American bureaucracies, where all information is distinguished into *"black"* and *"red"*; the former – suitably encrypted, and may be safely carried via public networks, radio, unattended parcels, and the like; the latter – confidential plaintext, strictly for use in a guarded location by politically-reliable personnel.

[15] Excluding IP and UDP headers.

4.1.2 "Redding" and "Blacking".

The addressee of a black packet identifies its sender simply by computing a `Seal` of its `Ciphertext` against *every* `Sealer` in his `WOT`, *in random order*[16]. If one of these is found to match the packet's `Seal`, the `Ciphertext` is deciphered with the corresponding `Ciphrator`, and the resulting red packet – now attributable to a particular peer – is processed. However, if no such match is found, the packet is deemed a *martian* and silently discarded.

The process of "redding" a black packet is illustrated below. The first step is performed against every `Key` in the receiver's `WOT`; if a matching `Seal` is found, the decipherment and extraction of the red packet takes place as shown:

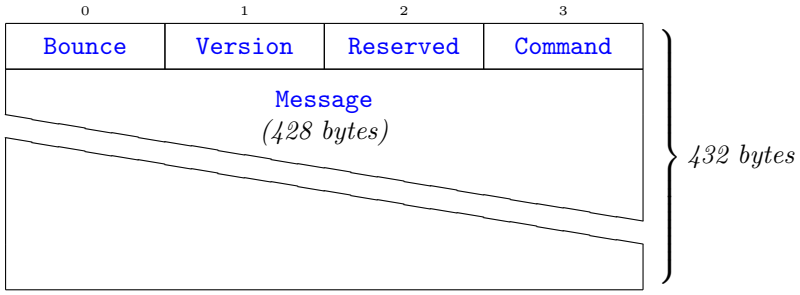


"Blacking" simply goes in reverse (bottom to top), with the main difference being that the addressee is known apriori, and so only *one* `Seal` is computed.

[16] Or – where hardware permits – in parallel. But in either case, without giving hints to snoops, via a timing side-channel, about whether any given packet was successfully verified by the receiving station.

4.1.3 Red Packet.

A `Payload[432]` – whether newly-created or deciphered from a black packet – is known as a *red packet*. It is the basic unit of all peer-to-peer communication in Pest. **Red packets are never exposed to public networks** – they comes into existence only inside a Pest station, and are always *blacked* prior to transmission. Every red packet consists of a 4-byte preamble followed by a 428-byte `Message`:



4.1.3.1 `Bounce` is an `Integer[1]` and initially set to *zero* when a red packet is originated[17]. Along with `Command`, `Bounce` is examined *immediately following* `Version` when an incoming red packet is processed. The treatment of `Bounce` depends on the supplied `Command`'s propagation type, as shown below:

Bounce	**Direct** `Command`**s**	**Broadcast** `Command`**s**
$N = 0$	Direct (**No Relay**)	Immediate *(Relayable)*
$0 < N < MaxBounce$	Malformed	Hearsay *(Relayable)*
$N \geq MaxBounce$	Malformed	Hearsay ***(No Relay)***

Combinations marked *Malformed* indicate a packet which must be silently discarded. Note that combinations marked *Relayable* simply refer to the packet being marked for relay after full validation, rather than being immediately relayed to peers.

4.1.3.2 `Version` **is the first field examined after deciphering a red packet**, and is an `Integer[1]` representing a "degrees Kelvin" (*decrementing*) version of the Pest protocol conformed to by a red packet. If the protocol version in use at a given station is known to represent a breaking change from previous versions, an incoming packet marked with a *higher* (i.e. older) version *must be silently discarded*[18].

[17] Note that a response to a `GetData` request for a *broadcast* packet *must* be sent with that packet's *original* – from the responder's point of view – `Bounce`.

[18] Unless the station is running a Pest implementation designed to correctly process packets with multiple incompatible versions of the protocol.

4.1.3.3 `Reserved` is mandatorily a `Zero[1]` in the current version of the protocol (0xFA).

4.1.3.4 `Command` is an `Integer[1]` which indicates the purpose of the red packet's `Message`. (See § 5.4.)

4.1.3.5 `Message` represents the useful cargo of a red packet. See § 4.2.

4.2 Classes of `Message`.

Every Pest `Message` occupies precisely 428 bytes. These are organized in several possible ways, depending on the *class* of the `Message`:

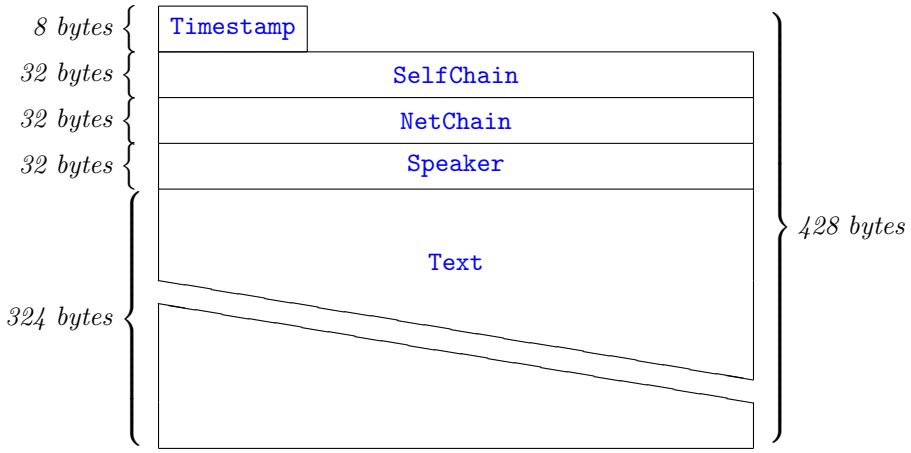
4.2.1 Chained `Message`.

Chained `Message`s transport human-readable `Text` between peers, with a claimed authorship indicated by `Speaker`, and allow for linkage with preceding Chained `Message`s via their `SelfChain` and `NetChain` hashes[19]. Only Chained `Message`s are stored in a station's `Log`, displayed to a station's *operator console*, and retrievable via `GetData`.

Currently (as of protocol version 0xFA) the following `Command` types denote Chained `Message`s:

Command	Name	Propagation
0x00	`BroadcastText`	Broadcast
0x01	`DirectText`	Direct

The layout of a Chained `Message` is shown below:



4.2.1.1 `Timestamp` is a `Time` representing the moment this `Message` came into being at its originator's station[20]. Any `Message` bearing a `Timestamp` *more than 15 minutes* away, in either direction, from the `Time` at a receiver's station at the time of its receipt, is deemed *stale* and discarded (unless it is an expected response to a `GetData` request.) **Every** Pest `Message` carries a `Timestamp`.

[19] Each such hash covers the entire 428 bytes of a preceding `Message`.

[20] Stations relaying a *broadcast* **must not** alter `Timestamp`s (or **any** other `Message` fields).

4.2.1.2 `SelfChain` normally identifies the *previous* `Message` of the given type most recently sent by the originator of this one. Note that the `SelfChain` of a `Message` **may not** refer to one with a `Timestamp` *greater than* its own; to itself; to any `Message` where `Speaker` is not equal to its own; or to any `Message` not of the same class. Any `Message` found to contain such a `SelfChain` is considered *malformed*.

4.2.1.3 `NetChain` normally identifies a previously-existing `Message` this one should be considered a logical *successor* of. Note that the `NetChain` of a `Message` **may not** refer to one with a `Timestamp` *greater than* its own[21]; to itself; or to any `Message` not of the same class. Any `Message` found to contain such a `NetChain` is considered *malformed*.

4.2.1.4 `Speaker` is an `AString[32]` representing the `Handle` in use by this `Message`'s originator. Pest `Handle`s are mandatorily pure `ASCII` to abolish the homoglyph impersonation attacks which plagued traditional chat protocols (e.g. `IRC`) that permitted the use of `UTF` in handles.

A `Handle` may not be less than 3 characters in length. It must consist strictly of alphanumeric `ASCII` characters, permitting both upper and lower case letters, and additionally including the underscore. `Message`s bearing a value of `Speaker` which does not conform to this pattern are considered *malformed* and silently discarded.

4.2.1.5 `Text` is a `UString[324]` – a human-readable text. In all cases where a `Text` in excess of 324 bytes must be sent, a series of Chained Multipart `Message`s should be used.

[21] If someone's clock is running "fast", and this results in peer stations having to *wait* to transmit, their operators will be informed via the *operator console* and know which peer to blame.

4.2.2 Chained Multipart `Message`.

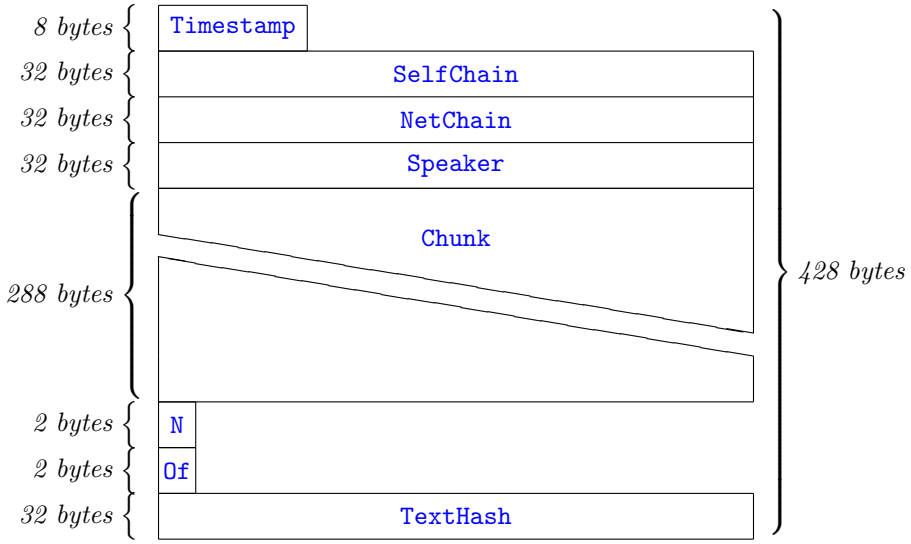
Chained Multipart `Message`s permit the piecewise transport of a `Text` in excess of the 324-byte capacity of a Chained `Message`, but in all other ways behave exactly like the latter. Every such `Message` carries a `Chunk` of `Text`; `N` – the index of the particular `Chunk`; `Of` – the total number of `Chunk`s required for reassembly; and `TextHash` – the expected hash of the reassembled `Text`. A given series of Chained Multipart `Message`s may transfer up to 65,535 chunks of an original `Text`, 288 bytes at a time, allowing for a reassembled `Text` of up to 18,874,080 bytes in length.

A station which receives a sequence of Chained Multipart `Message`s will attempt to reassemble the `Chunk`s in the specified order to obtain the original `Text`, and will verify the result against `TextHash` – or indicate to the operator, via the *operator console*, if this proved impossible.

Currently (as of protocol version 0xFA) the following `Command` types denote Chained Multipart `Message`s:

Command	Name	Propagation
0x06	`BroadcastTextM`	Broadcast
0x07	`DirectTextM`	Direct

The layout of a Chained Multipart `Message` is shown below:



4.2.2.1 `Timestamp` must be *equal* for all `Message`s carrying the `Chunk`s of the original `Text` represented by `TextHash`. Otherwise, exactly as in Chained `Message`s; see § 4.2.1.1.

4.2.2.2 `SelfChain` if `N` $\neq 1$, must be equal to the `Hash256` of the Chained Multipart `Message` carrying `Chunk N - 1`. Otherwise, exactly as in Chained `Message`s; see § 4.2.1.2.

4.2.2.3 `NetChain` If `N` $\neq 1$, must be equal to `SelfChain`. Otherwise, exactly as in Chained `Message`s; see § 4.2.1.3.

4.2.2.4 `Speaker` must be *equal* for all `Message`s carrying the `Chunk`s of the original `Text` represented by `TextHash`. Otherwise, exactly as in Chained `Message`s; see § 4.2.1.4.

4.2.2.5 `Chunk` is a `UString[288]`, and represents the `N`th chunk (from total number `Of`) of the original `Text` represented by `TextHash`. Otherwise, exactly as in Chained `Message`s; see § 4.2.1.5.

4.2.2.6 `N` is an `Integer[2]`, and represents the index (starting with *one*) of the particular `Chunk` of the original `Text` represented by `TextHash` carried in this particular `Message`. $1 \leq N \leq$ `Of`.

4.2.2.7 `Of` is an `Integer[2]`, and represents the *total number* of `Chunk`s into which the original `Text` had been split. It must be *equal* for all `Message`s carrying the `Chunk`s of a particular original `Text` represented by `TextHash`.

4.2.2.8 `TextHash` represents a `Hash256` of the complete original `Text`, prior to being split into `Chunk`s, and must be *equal* for all `Message`s carrying the `Chunk`s of a particular original `Text`. It is used by a receiving station to verify the successful reassembly of a sequence of Chained Multipart `Message`s into the intended original `Text`.

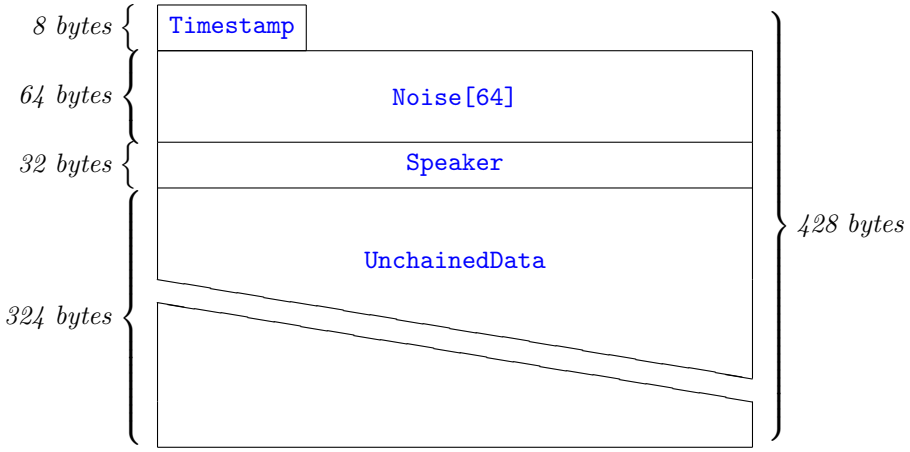
4.2.3 Unchained `Message`.

Unchained `Message`s transport non-human-readable data between peers. Such `Message`s do not allow hash linkage with other `Message`s, are not stored in a station's `Log`, are not displayed to a station's *operator console*, and are not retrievable via `GetData`. However, an Unchained `Message` still has a claimed authorship indicated by `Speaker`, and may therefore be a *broadcast*.

Currently (as of protocol version 0xFA) the following `Command` types denote Unchained `Message`s:

Command	Name	Propagation
0x02	`Prod`	Direct
0x03	`GetData`	Direct
0x04	`KeyOffer`	Direct
0x05	`KeySlice`	Direct
0xFE	`AddressCast`	Broadcast
0xFF	`Ignore`	Direct

The layout of an Unchained `Message` is shown below:



4.2.3.1 `Timestamp` is exactly like a *Chained* `Message`'s `Timestamp`. (See § 4.2.1.1.)

4.2.3.2 `Speaker` is exactly like a *Chained* `Message`'s `Speaker`. (See § 4.2.1.4.)

4.2.3.3 `UnchainedData` is a custom data structure, specific to a particular `Command`.

4.2.4 Binary `Message`.

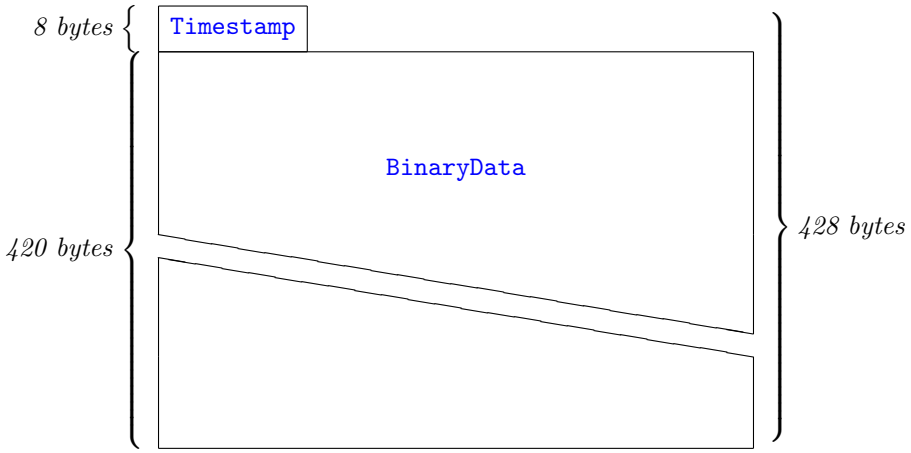
Binary `Message`s resemble Unchained `Messages`, in that they transport non-human-readable data between peers; however, they do not carry a `Speaker` field, and are therefore mandatorily *direct*.

Analogously to an Unchained `Message`, a *Binary* `Message` does not allow hash linkage with other `Message`s, is not stored in a station's `Log`, is not displayed to a station's *operator console*, and is not retrievable via `GetData`.

Currently (as of protocol version 0xFA) the following `Command` types **are reserved for** Binary `Message`s:

Command	Name	Propagation
`0x40`	`Inv`	Direct
`0x41` – `0x80`	*Reserved (Binary)*	Direct

The layout of a Binary `Message` is shown below:



4.2.4.1 `Timestamp` is exactly like a *Chained* `Message`'s `Timestamp`. (See § 4.2.1.1.)

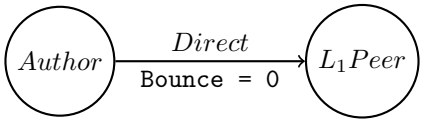
4.2.4.2 `BinaryData` is a custom data structure, specific to a particular `Command`.

5 The Pest Message.

5.1 The Three Paths of a Pest Message.

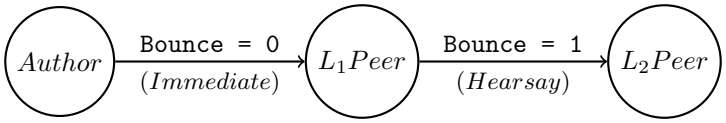
5.1.1 Direct.

The Pest equivalent of IRC's *"private message"* is called a *direct*. However, unlike IRC *PMs*, Pest *directs* are authenticably-encrypted and travel straight from the originator station to the addressee (necessarily a peer of the originator.) A *Direct* is considered *prima facie* authentic: so long as the Key of the peering has not been compromised, the addressee can be certain that the *direct* could not have been forged by a third party or altered in transit.



5.1.2 Broadcast.

The Pest equivalent of a message emitted into an IRC channel is the *broadcast*, which propagates from peer to peer until it has been seen by every reachable station[22] on a given Pest net. The receiver classifies it as either *immediate* or *hearsay*:



5.1.2.1 Immediate refers to a broadcast received *directly from its originator* (Bounce = 0). Like *direct*s, *immediates* are considered *prima facie* authentic.

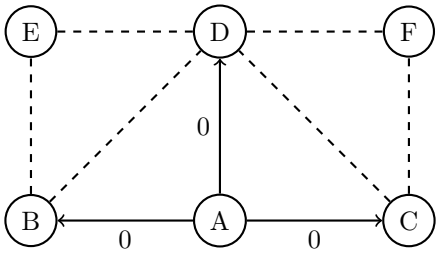
5.1.2.2 Hearsay refers to a copy of a broadcast received from anyone *other* than its originator (Bounce ≠ 0). Its authorship cannot be cryptographically verified[23] – all that is known for certain is that it was originated by *some* participant of the Pest net, and eventually found its way to the receiver via one or more of his peers. For this reason, hearsays are specially-marked when displayed to an operator. The marking lists the peers who relayed copies of the hearsay.

[22] Broadcast propagation terminates at stations where Bounce ≥ MaxBounce.

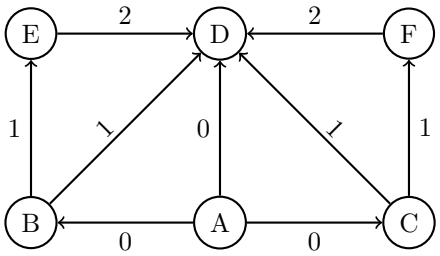
[23] If Pest used *public key* cryptography – it could be. However, no currently commonplace hardware can process usefully strong public key signatures, in *constant time*, at a rate which would give acceptable DDOS resistance. Nor is an unfragmentable UDP packet large enough to hold a 4096-bit RSA ciphertext and its signature...

5.2 Broadcast Propagation.

Suppose that in the net pictured below, station `A` originates a broadcast packet `P` and transmits *immediate* copies to its peers `B`, `C`, and `D`. How many copies of `P` could reach station `D`, and with what `Bounce`s?



Note that the shortest network path is not necessarily the shortest *temporal* path. Packets traveling over public networks are sometimes delayed or even lost in transit. In addition to the *immediate* path `A` → `D`, there are also *four* possible *hearsay* paths for `P`: `A` → `B` → `D`, `A` → `C` → `D`, `A` → `B` → `E` → `D`, and `A` → `C` → `F` → `D`. Station `D` may receive up to *five* copies of `P` – *in unknown order*.



TODO

5.3 `Message` Storage.

5.3.1 The `Filter`.

A station's `Filter` is a data structure which retains a hash of **every** valid incoming and originated `Message`[24], until such a time that if a duplicate of that `Message` were received, it would be deemed *stale*. Also stored along with the `Message`'s hash is its *min-`Bounce`*[25].

The `Filter` is used for *deduplication*: every otherwise-valid (i.e. not *martian*, *malformed*, or *stale*) incoming `Message` is hashed and the `Filter` is queried for said hash. If it was found to contain the hash, the `Message` is deemed a *duplicate* and discarded. However, if the `Message` was *not* originated at the station, and the Red Packet carrying the duplicate had a `Bounce` *equal or lower* to the *min-`Bounce`* associated with the hash in the `Filter`, the relayer list for that `Message` in the `Log` will be updated to include the peer who supplied the duplicate copy; and if the duplicate's `Bounce` was *lower* than the stored *min-`Bounce`*, the latter will be updated in both the `Log` and the `Filter`.

5.3.2 The `Log`.

A station's `Log` is a non-volatile store which retains every[26] valid *Chained* `Message` originated or received by the station, indexed *uniquely* by hash (and non-uniquely by `Timestamp`.) The `Time` at which a given `Message` was received (for incoming `Message`s strictly; distinctly from its internal originator-given `Timestamp`) is stored; as well as the `Command` code associated with it; as well as its *min-`Bounce`*; and also:

- In the case of a *direct*[27] or *immediate*: the identity of the peer from whom the `Message` originated.
- In the case of a *hearsay*: a list of all peers from whom a *min-`Bounce`* Red Packet containing the `Message` was received. If the `Message` has not yet *expired* and exited the `Filter`, that list may be *updated* when additional duplicate copies with a `Bounce` $\leq$ *min-`Bounce`* (including, possibly, an *immediate* copy – in which case the list will be reduced to *one* peer) are received by the station.

The `Log` may be searched or browsed in order (either chronological or by chain) via the *operator console*. Arbitrary `Message`s may be retrieved from the `Log` via their hash when processing *chain*s or servicing `GetData` requests.

[24] Naturally, along with its `Timestamp`.

[25] That is, the `Bounce` of the Red Packet containing the copy received with the *lowest* `Bounce`. For a *hearsay*: ≥ 1; for *direct*s, *immediate*s, and any originated `Message`s, this value is simply zero.

[26] Up to an operator-configured disk footprint limit.

[27] Note that only the peer to whom a *direct* was originally addressed may request its retransmission via `GetData`. It is recommended to store *direct*s separately from *broadcast*s; preferably, in such a way that they are encrypted when the station is offline.

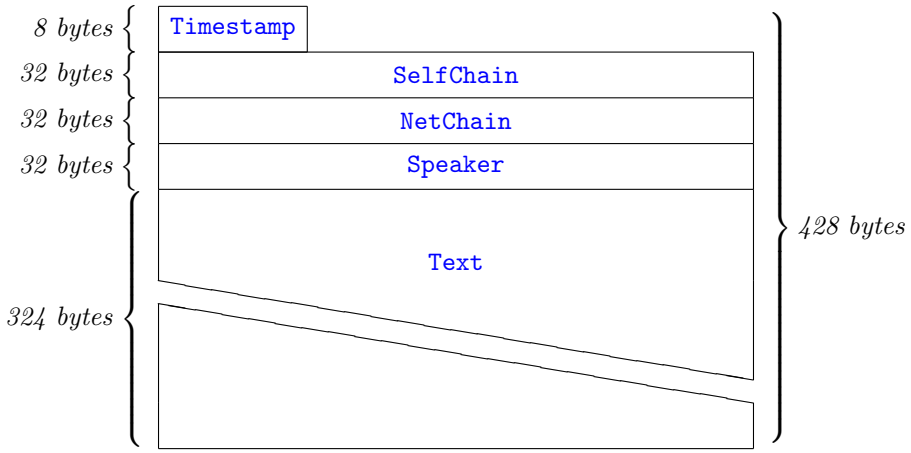
5.4 Defined `Message` Types.

As of protocol version 0xFA, the available `Command` codes for a `Message` are:

Command	Name	Propagation	Message Class
0x00	BroadcastText	Broadcast	Chained
0x01	DirectText	Direct	Chained
0x02	Prod	Direct	Unchained
0x03	GetData	Direct	Unchained
0x04	KeyOffer	Direct	Unchained
0x05	KeySlice	Direct	Unchained
0x06	BroadcastTextM	Broadcast	Chained Multipart
0x07	DirectTextM	Direct	Chained Multipart
0x08 – 0x3F	*Reserved*		
0x40	Inv	Direct	Binary
0x41 – 0x80	*Reserved (Binary)*	Direct	Binary
0x81 – 0xFD	*Reserved*		
0xFE	AddressCast	Broadcast	Unchained
0xFF	Ignore	Direct	Unchained

5.4.1 `BroadcastText`

Command	Description	Propagation	Message Class
0x00	Broadcast Text	Broadcast	Chained



A `BroadcastText` carries a human-readable `Text`. It enters the `Filter` and the `Log`, and a copy is then sent to every peer in the originator's `WOT`, and will be *broadcast* to *their* peers, and so on. (See: § 4.1.3.1, § 5.1.2.2, § 5.2.)

5.4.1.1 `Timestamp` is common to all Pest `Message`s. (See § 4.2.1.1.)

5.4.1.2 `SelfChain` is a `Hash256` of the most recent `BroadcastText` previously sent by the originator. If the originator believes that he is sending a `BroadcastText` to the net *for the first time*, `SelfChain` may be set to equal `Zero[32]`. However, in this case, every receiver of the `Message` will be warned[28] of this fact. See also § 4.2.1.2.

5.4.1.3 `NetChain` is a `Hash256` of a `BroadcastText` previously *sent or received* by the originator; by default, the most recent – known to the originator – `BroadcastText` on the net. `NetChain` may be set to equal `Zero[32]`, if and only if[29] `SelfChain` was also set to `Zero[32]`. See also § 4.2.1.3.

5.4.1.4 `Speaker` *must* match a `Handle` known by each *immediately*-receiving peer to be in use by the originator. See § 4.2.1.4.

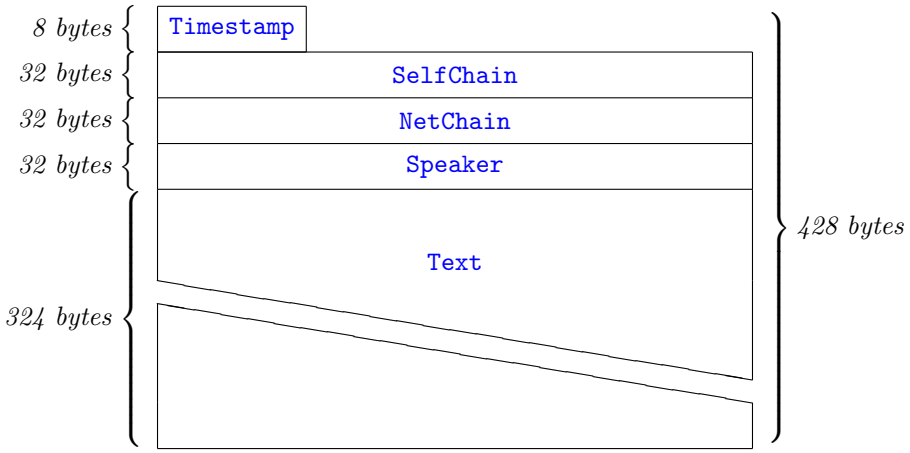
5.4.1.5 `Text` See § 4.2.1.5.

[28] In a `CLI` operator console, the `Speaker` of such a `Message` will be prefixed with an exclamation point.

[29] Version of the protocol prior to `0xFA` did not require this. Therefore if this condition is violated, a warning is to be issued to the receiving operator.

5.4.2 DirectText

Command	Description	Propagation	Message Class
0x01	Direct Text	Direct	Chained



A `DirectText` carries a human-readable `Text`, intended as a confidential communication to *one* person. It enters the `Filter` and the `Log`, and is then sent directly to *one specified addressee*, who must be a peer of the originator.

5.4.2.1 `Timestamp` is common to all Pest `Message`s. (See § 4.2.1.1.)

5.4.2.2 `SelfChain` is a `Hash256` of the most recent `DirectText` previously sent by the originator *to the given addressee*. If the originator believes that he is sending a `DirectText` to this addressee *for the first time*, `SelfChain` may be set to equal `Zero[32]`. However, in this case, the receiver of the `Message` will be warned of this fact. See also § 4.2.1.2.

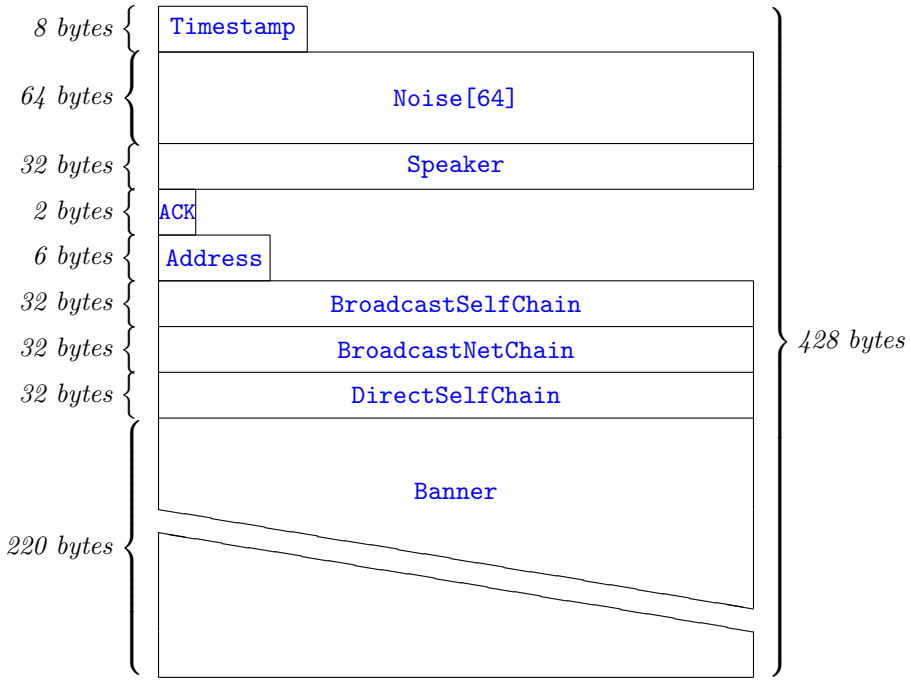
5.4.2.3 `NetChain` is a `Hash256` of a `DirectText` previously *sent or received* by the originator *to or from the given addressee*; by default, the most recent – known to the originator – such `DirectText`. `NetChain` may be set to equal `Zero[32]`, if and only if `SelfChain` was also set to `Zero[32]`. See also § 4.2.1.3.

5.4.2.4 `Speaker` *must* match a `Handle` known by the addressee to be in use by the originator. See § 4.2.1.4.

5.4.2.5 `Text` See § 4.2.1.5.

5.4.3 Prod

Command	Description	Propagation	Message Class
0x02	Prod	Direct	Unchained



Prod Messages aid Pest stations in NAT penetration (via AddressCast) and chain synchronization (via GetData). They also allow a station operator to share an arbitrary string with his peers (e.g. advertising his particular Pest implementation, a WWW site URL, etc.)

5.4.3.1 Timestamp is common to all Pest Messages. (See § 4.2.1.1.)

5.4.3.2 Speaker is *ignored*, as this type of Message is *direct*, and so the addressee is able to unambiguously identify the originator.

5.4.3.3 ACK is an Integer[2] with permitted values of 0: indicating that an answer to this Prod, consisting of a Prod from the addressee is requested; or 1: indicating that this Prod is an answer to one previously received from the addressee.

5.4.3.4 Address is the Address currently found in the sender's AT entry for the addressee. It is identical to the one to which the sender intends to transmit

the packet bearing this message. This allows a station trapped behind a `NAT` to learn its publicly-routable `Address`, for use with `AddressCast`.

5.4.3.5 `BroadcastSelfChain` is the sender's latest *broadcast* `SelfChain`.

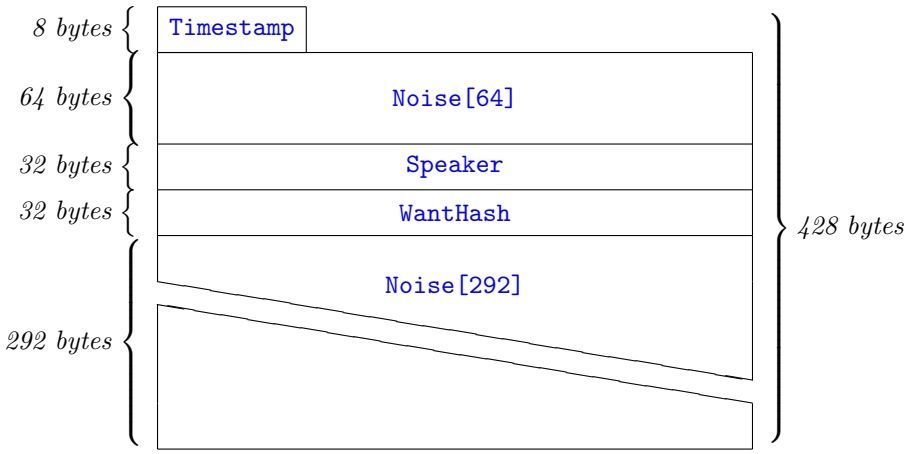
5.4.3.6 `BroadcastNetChain` is the sender's latest *broadcast* `NetChain`.

5.4.3.7 `DirectSelfChain` is the sender's latest *direct* `SelfChain` with *this* addressee.

5.4.3.8 `Banner` is a `UString[220]`, and is an arbitrary human-readable description of the sender's Pest station. This string may be set by an operator via the `BANNER` command.

5.4.4 GetData

Command	Description	Propagation	Message Class
0x03	GetData	Direct	Unchained



A GetData carries a WantHash identifying a previously-existing *Chained* Message being requested for retransmission. A BroadcastText may be requested by any peer, but a DirectText may only be retransmitted to the peer *to whom it was originally addressed*.

A station issues a GetData upon encountering any SelfChain or NetChain (including in a Prod) for which no corresponding Message exists in the Log. The WantHash is added to a non-volatile data structure: AskedFor. At all times when AskedFor is non-empty, it is queried for the Hash256 of every incoming *Chained* Message, at the point immediately prior to the latter being subjected to the *staleness* test. If a match occurs, the matching AskedFor entry is removed, and the received Message is processed without regard to *staleness* (as it may well be stale.) It enters the Log and the Filter, and its SelfChain and NetChain are queried against the Log. Additional GetDatas are then issued, if required. If no response to a GetData appears within GetDataWait milliseconds, it is reissued, after the same interval, at most GetDataTries times. A GetData triggered by a BroadcastText is sent to *each* of the station's peers, *in random order*. A BroadcastText returned via GetData is *not* propagated to peers.

5.4.4.1 Timestamp

is common to all Pest Messages. (See § 4.2.1.1.)

5.4.4.2 Speaker

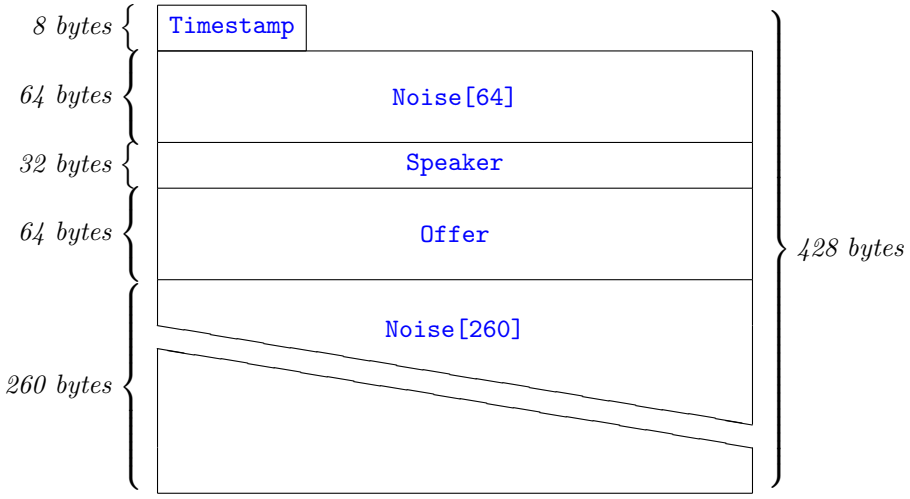
is *ignored*, as this type of Message carries no text, and, since it is *direct*, its addressee is able to unambiguously identify the originator.

5.4.4.3 WantHash

is a Hash256 which identifies the *Chained* Message the originator is asking for.

5.4.5 `KeyOffer`

Command	Description	Propagation	Message Class
`0x04`	Key Offer	Direct	Unchained



A `KeyOffer` carries a Rekeying `Offer`, i.e. a hash of a proposed `xor` `Slice` for creating a replacement peering `Key`. The Rekeying procedure begins with the participants exchanging `Offer`s in order to demonstrate that their `Slice`s were generated *independently of one another*. A `KeyOffer` may be sent to *initiate* a Rekeying, or as a *response* to a `KeyOffer` previously received from the addressee.

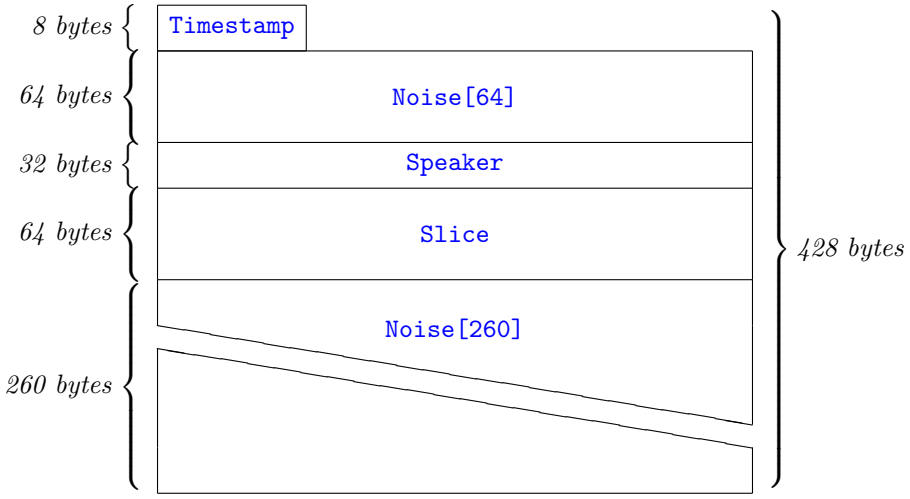
5.4.5.1 `Timestamp` is common to all Pest `Message`s. (See § 4.2.1.1.)

5.4.5.2 `Speaker` is *ignored*, as this type of `Message` is *direct*, and so the addressee is able to unambiguously identify the originator.

5.4.5.3 `Offer` is equal to `Hash512`(`Slice`).

5.4.6 KeySlice

Command	Description	Propagation	Message Class
0x05	Key Slice	Direct	Unchained



A KeySlice carries a proposed Slice for the calculation of a replacement Key. In the Rekeying procedure, after the participants have exchanged Offers and determined that they were not identical, each peer reveals his Slice to the other. The procedure concludes successfully if and only if each of the Slices is in fact found to hash to its respective previously-sent Offer. The participants calculate a new Key, equal to the xor of:

- The Key previously shared by the participants, with which the Rekeying exchange was carried out.
- Each participant's Slice.

5.4.6.1 Timestamp is common to all Pest Messages. (See § 4.2.1.1.)

5.4.6.2 Speaker is *ignored*, as this type of Message is *direct*, and so the addressee is able to unambiguously identify the originator.

5.4.6.3 Slice is a Noise[64], and must be verified by the addressee to hash to the Offer that had been sent previously. If this is found to be false, the addressee will consider the Rekeying process to have been aborted.

5.4.7 `BroadcastTextM`

Command	Description	Propagation	Message Class
`0x06`	Broadcast Text Multipart	Broadcast	Chained Multipart

A `BroadcastTextM` is similar to a `BroadcastText`, but carries a `Chunk` of a multi-part `Text`; reassembly is attempted upon the successful receipt of all required `Chunk`s. See: Chained Multipart (§ 4.2.2).

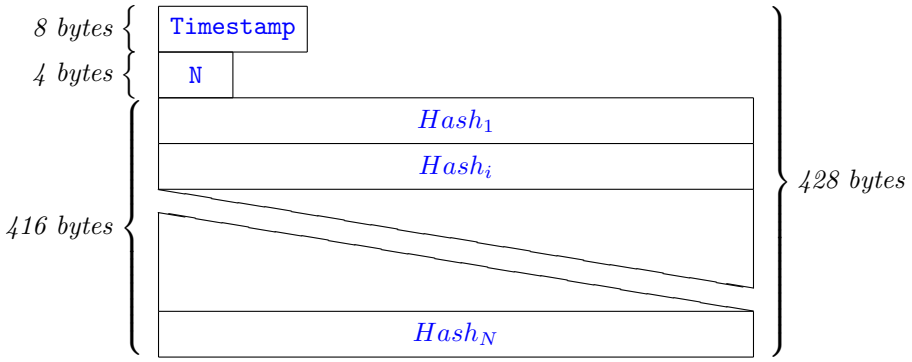
5.4.8 `DirectTextM`

Command	Description	Propagation	Message Class
`0x07`	Direct Text Multipart	Direct	Chained Multipart

A `DirectTextM` is similar to a `DirectText`, but carries a `Chunk` of a multipart `Text`; reassembly is attempted upon the successful receipt of all required `Chunk`s. See: Chained Multipart (§ 4.2.2).

5.4.9 `Inv`

Command	Description	Propagation	Message Class
0x40	Inventory	Direct	Binary



An `Inv` supplies a peer with a list of hashes identifying at least one and up to 13 arbitrary Chained `Message`s, in *descending order* of `Timestamp`. Only the hashes of such `Message`s as the addressee may retrieve via `GetData` (i.e. `BroadcastText`, and `DirectText` originally addressed to him) may appear in the `Inv`. The receiver of an `Inv` *may* check these hashes against his `Log`, and *may* issue `GetData` requests to obtain `Message`s found to be missing in the latter. The circumstances under which a station emits an `Inv` are currently *unspecified*.

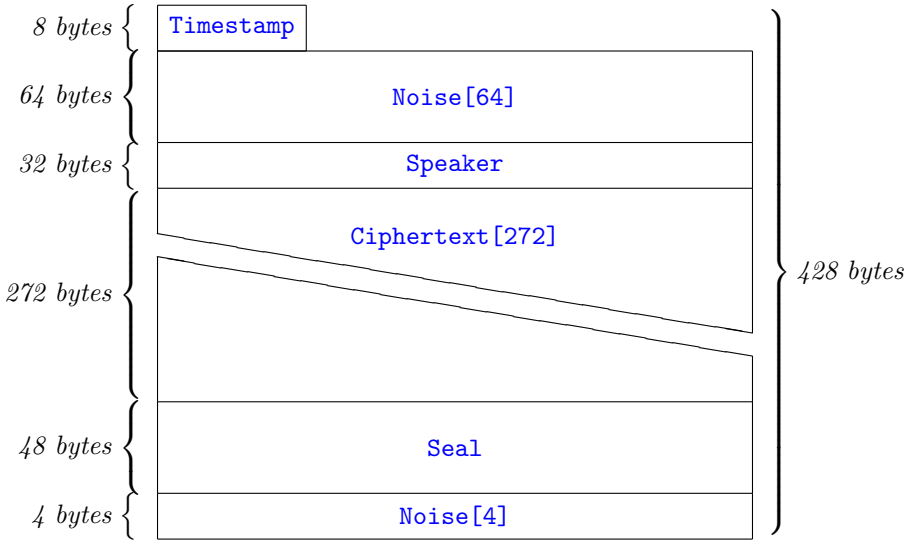
5.4.9.1 `Timestamp` is common to all Pest `Message`s. (See § 4.2.1.1.)

5.4.9.2 `N` is an `Integer[4]`, and signifies the number of `Message` hashes supplied in this `Inv`. $1 \leq N \leq 13$.

5.4.9.3 $Hash_i$ is a `Message` hash. All unused entries in the `Inv` (i.e. $N < i \leq 13$) must be set to `Noise[32]`.

5.4.10 AddressCast

Command	Description	Propagation	Message Class
0xFE	Address Cast	Broadcast	Unchained



Address Casts are periodically *broadcast* by a station at all times when there are *cold* peers in its WOT. A cold peer is one from whom no valid packets have been received for at least ColdTime milliseconds, or for whom at least one Key is known but no AT entry currently exists.

Similarly to a black packet, each Address Cast carries a Ciphertext and a Seal, both of which are Keyed to the *target* peer. The latter, while at a given time not directly reachable by the sender of the Address Cast – may be present on the net and able to receive *broadcast*s.

The ciphered payload of an Address Cast consists of an Address at which the sender would like to be reached by the target. Incoming Address Casts are only deciphered when the receiver has at least one cold peer, and strictly on a best-effort basis (i.e. when the CPU would otherwise be idle.)

5.4.10.1 Timestamp

is common to all Pest Messages. (See § 4.2.1.1.)

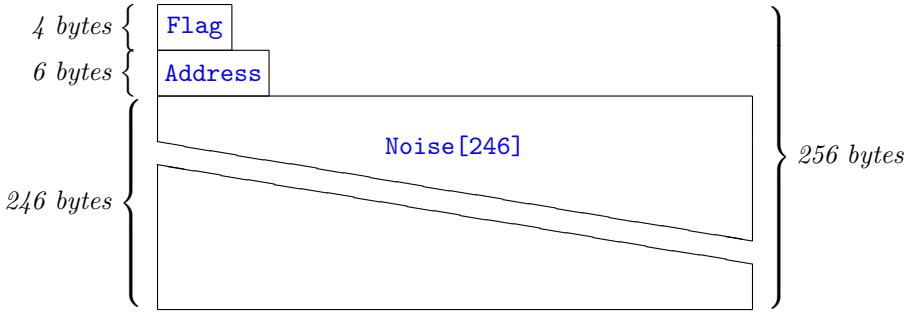
5.4.10.2 Speaker

must match a Handle known by each *immediately*-receiving peer to be in use by the originator. See § 4.2.1.4. The *target* of the Address Cast will attempt to decode it using every Key known to be in use by the peer associated with this Handle, supposing the latter is in fact found in his WOT, and that peer is currently *cold*.

5.4.10.3 Ciphertext

is a Ciphertext[272], Keyed to the *target*.

5.4.10.4 `Seal` is a `Seal`, `Key`ed to the *target*. The receiver of an Address Cast will attempt to decode it, on a best-effort basis, via a process resembling black packet intake: a `Seal` verification is attempted against the `Sealer` belonging to each `Key` of the peer identified by `Speaker`, if the latter is *cold*, *in random order*. If a match is found, the `Ciphertext` will be deciphered with that `Key`'s `Ciphrator`. A successful Address Cast decipherment yields a `Plaintext[272]`. After discarding the `Nonce`, what remains is a `Payload[256]`, also referred to as a *Red Address Cast*:



5.4.10.5 `Flag` is a `Integer[4]`:

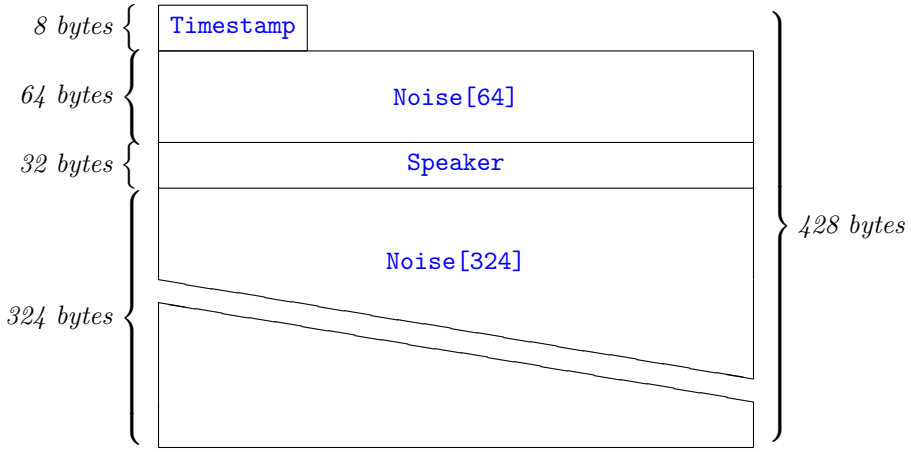
- If `Flag` $= 0$: the originator is requesting a single `Prod` to be sent by the *target* to the supplied `Address`.
- If `Flag` $= 1$: the originator believes that he is trapped behind a *symmetric* `NAT`, and is requesting a *port hammering session*: a sequence of `Prod`s to be sent by the *target* to the `IP` specified in the supplied `Address`, each with a *randomly*-selected 16-bit port. Hammering is requested to begin precisely `HammerWait` milliseconds after the `Timestamp` of the Address Cast – so that it can be carried out *from both directions*. The originator only proceeds with the hammering if the target confirms the request by sending an Address Cast to the originator during the `HammerWait` interval. A `HammerShots` number of random ports is tried. If a valid packet is received from the peer being hammered, the hammering terminates, as the peer is no longer cold. Incoming requests for hammering sessions are ignored when such a session is in progress.
- If `Flag` > 1 : the `Message` is discarded.

5.4.10.6 `Address` is an `Address` at which the originator of the Address Cast would like to be reached by the target. A target which successfully processes a Red Address Cast from a *cold*[30] peer will execute an equivalent of the `AT` Command, and henceforth use the supplied `Address` for all outgoing packets intended to reach the originator.

[30] A Red Address Cast found to have been originated by a *warm* (i.e. not *cold*) peer is ignored.

5.4.11 Ignore

Command	Description	Propagation	Message Class
0xFF	Ignore	Direct	Unchained



An *Ignore* Message resets the ColdTime interval with respect to a given peering, but otherwise is simply ignored by the receiver. Ignores also serve as *chaff* to stymie traffic analysis by snoops, and to maintain port forwarding state for a station that is operating behind a NAT. At least one Ignore must be transmitted to each WOT peer, *in random order*, every IgnorePeriod milliseconds. Ignores may also be sent under other, unspecified circumstances.

5.4.11.1 Timestamp

is common to all Pest Messages. (See § 4.2.1.1.)

5.4.11.2 Speaker

is *ignored*.

6 Operator Console

TODO

7 Rekeying

TODO

8 NAT Penetration

TODO

A Appendix.

A.1 Fundamental Data Types.

The following data types are used in Pest, and must be encoded and decoded as described here:

A.1.1 `Zero`

A `Zero[N]` consists of precisely `N` bytes, each of which is mandatorily equal to *zero*. (If, upon `Message` processing, any byte in a field defined as `Zero` is found to be non-zero, the `Message` is *malformed*, and *must be silently discarded*.)

A.1.2 `Integer`

An `Integer[N]` is an *unsigned fixed-point integer*, occupying precisely `N` bytes, mandatorily in *little-endian* order when applicable. `N` may be equal to 1, 2, 4, or 8.

A.1.3 `Noise`

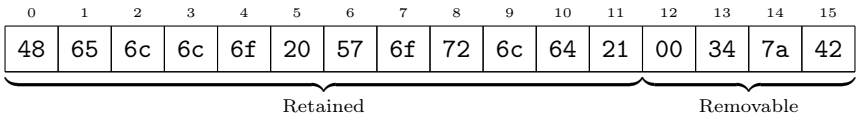
A `Noise[N]` consists of precisely `N` bytes of *uniformly-distributed* entropic noise, obtained from an auditable hardware TRNG where feasible.

A.1.4 `Time`

A `Time` consists of an `Integer[8]`, obtained from a Pest station's 64-bit *monotonic epoch clock* at the time of `Message` encoding; traditionally defined as: *"a 64-bit unsigned fixed-point number, in seconds relative to 00:00:00 January 1, 1970, UTC"*.

A.1.5 `AString`

An `AString[N]` occupies precisely `N` bytes, and contains a *seven-bit-clean* (pure `ASCII`, i.e. no byte exceeds `0x7F`) string, *at most* `N` characters long. If, after encoding the string, there is unused space at the end of the field, the *first* unused byte is set to zero, and any remaining bytes – to `Noise`[31]. For example, here is one possible encoding of `Hello World!` into an `AString[16]`:

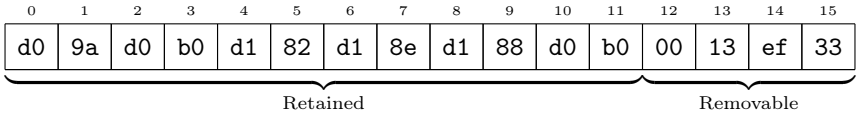


[31] This is done to maximize the entropy of a Pest `Message` – to frustrate *known-plaintext* attacks against the cipher and signature schemes.

Upon Message processing, for all purposes *other than hashing, storage, and rebroadcasting*, an AString[N] field is treated as an AString[N-i] where i is the number of removable trailing bytes, if any (starting with the zero) found in the original. (If a *leading* zero was found, N-i will equal zero, i.e. the string is considered empty.) After removing trailing bytes, the string is verified to be seven-bit-clean; if this is found to be false, the Message containing it is *malformed* and *must be silently discarded*.

A.1.6 UString

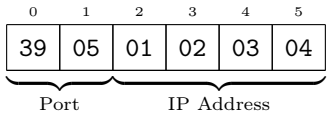
A UString[N] occupies precisely N bytes, and contains a validly UTF8-encoded string, *at most* N characters long. If, after encoding the string, there is unused space at the end of the field, the *first* unused byte is set to zero, and any remaining bytes – to Noise. For example, here is one possible encoding of Катюша into a UString[16]:



Upon Message processing, for all purposes *other than hashing, storage, and rebroadcasting*, a UString[N] field is treated as a UString[N-i] where i is the number of removable trailing bytes, if any (starting with the zero) found in the original. (If a *leading* zero was found, N-i will equal zero, i.e. the string is considered empty.) After removing trailing bytes, the string is validated per UTF8. A Message found to contain an invalid UString is *malformed* and *must be silently discarded*.

A.1.7 Address

An Address occupies precisely 6 bytes, and consists of an Integer[2] *port* followed by four Integer[1] which represent a publicly-routable IPv4 *IP address*. The latter are laid out in order of *descending significance*. For example, the IP address 1.2.3.4 and port 1337 will be encoded as:



A Message containing an Address which is determined *not* to be publicly-routable is *malformed*, and *must be silently discarded*.

A.1.8 Key

A Key occupies precisely 64 bytes, which consist of a Noise[32] Sealer (an HMAC-384 signing key, used to create and verify Seals) followed by a Noise[32] Ciphrator (a Serpent cipher key, used to encipher and decipher Ciphertexts.) The two components are generated independently of one another, and *may not be identical.*[32]

A Key is the secret shared by a pair of Pest peers, and under no circumstances revealed – in whole or in part – to any third party (including *any of their other peers.*) It enables the peer relationship from *both directions*: the question of which peer had sent a given packet is answered *strictly* by attempting verification of its Seal against *every* Sealer known to the receiver. After a successful verification, the Ciphrator corresponding to the Sealer which verified the packet's Seal is used to decipher its Ciphertext.

On every occasion when a Key must be handled by a human, it is represented in Base-64 format. For example, the following Key:

Sealer:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
d8	d7	b0	96	29	7b	08	40	1c	ac	b9	4b	26	12	5a	5f
56	ce	85	83	33	85	bc	d7	e6	cf	d4	3d	81	97	33	7c
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Ciphrator:

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
53	4e	bc	bb	b2	ab	f0	63	2a	7a	7d	f8	a7	a5	09	a0
55	52	99	6a	18	e0	20	62	0b	ac	f7	00	1f	6a	08	e8
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

... will be encoded in Base-64 format as:

```
2Newli17CEAcrLlLJhJaX1bOhYMzhbzX5s/UPYGXM3xTTry7sqvwYyp6ffinpQmgVVKZahjgIGILrPcAH2oI6A==
```

A.1.9 Ciphertext

A Ciphertext[N] occupies precisely N bytes, and is an output of the Serpent symmetric cipher, operated in *Cipher Block Chaining (CBC)* mode, with the cipher key being the Ciphrator component of a peer's Key. N must be a multiple of 16 (the block size of Serpent). The Plaintext corresponding to a given Ciphertext[N] also occupies precisely N bytes.

[32] A conforming implementation of Pest will *not* accept a Key containing identical Sealer and Ciphrator components, and immediately warn the operator.

A.1.10 Nonce

A Nonce consists of a Noise[16] and is mandatorily present at the start of a Plaintext. It is not used in any way following decryption; its purpose is to prevent the emission of identical Ciphertext and Seal in the event of a given Plaintext being ciphered and sealed more than once with a particular Key, and, more generally, to increase the resistance of the cipher and signature schemes used in Pest to *known-plaintext attacks*.

A.1.11 Seal

A Seal is a signature produced via the traditional HMAC-384 function. It occupies precisely 48 bytes. Seals are produced and verified using the Sealer component of a peer's Key, exclusively over a Ciphertext.

A.1.12 Plaintext

A Plaintext[N] occupies precisely N bytes, and consists of a Nonce followed by a Payload which occupies N-16 bytes. N must be a multiple of 16 (the block size of Serpent) and greater than or equal to 32. A Plaintext is the result of the decipherment of, or suitable for the creation of, a Ciphertext.

A.1.13 Payload

A Payload[N] occupies precisely N bytes, and is the useful cargo of a Plaintext. N must be a multiple of 16 (the block size of Serpent), and greater than or equal to 16.

A.1.14 Hash256

A Hash256 is an output of the traditional SHA-256 function, and occupies precisely 32 bytes.

A.1.15 Hash512

A Hash512 is an output of the traditional SHA-512 function, and occupies precisely 64 bytes.

A.2 Knobs

The Pest protocol refers to certain numeric constants, which may be adjusted to better fit the needs of particular use cases. This section contains a complete list of such constants, along with a *recommended default value* for each. Generally speaking, `Knobs` should not be turned away from their default values without a good reason. Altering `Knobs` may affect your station's interoperability with its peers.

A.2.1 `MaxBounce`

If an incoming *broadcast red packet*'s `Bounce` $\geq$ `MaxBounce`, the packet is processed by the receiver but *not* automatically relayed to peers (a peer *may*, however, explicitly request the `Message` via `GetData`.) `MaxBounce` allows participants in large Pest nets to limit their interaction to a group bounded by Dunbar's Number. `MaxBounce` is an `Integer[1]`, i.e. $0 \leq N \leq 255$. Recommended default value: **7**.

A.2.2 `GetDataWait`

If a `GetData` request is not satisfied (i.e. a `Message` with the requested `Hash256` arrives from *any* of the peers to whom the particular `GetData` was issued) within `GetDataWait` milliseconds after a particular attempt, an additional `GetData` request will be issued. The attempt will be made at most `GetDataTries` times. Ordinary station operation continues during a `GetDataWait` interval – nothing is blocked by it. `GetDataWait` is an `Integer[2]`. Recommended default value: **2500**.

A.2.3 `GetDataTries`

A `GetData` request triggered by a particular incoming `Message` will be attempted at most `GetDataTries` times. `GetDataTries` is an `Integer[2]`. Recommended default value: **7**.

A.2.4 `ColdTime`

A peer from whom no valid packets have been received for at least `ColdTime` milliseconds is considered *cold*. `AddressCast Message`s are to be generated for cold peers strictly. Attempts to decode an `AddressCast Message` are to be undertaken if and only if a station presently has at least one cold peer. `ColdTime` is an `Integer[4]`. Recommended default value: **30000**.

A.2.5 AddrCastPeriod

An attempt to re-establish communication with a *cold* peer via the AddressCast mechanism will be repeated no sooner than AddrCastPeriod milliseconds after the previous such attempt. AddrCastPeriod must be greater than or equal to ColdTime. AddrCastPeriod is an Integer[4]. Recommended default value: **60000**.

A.2.6 IgnorePeriod

An Ignore is sent to every peer every IgnorePeriod milliseconds, to maintain "warmth", and preserve routing table state in case the station or a given peer is behind a NAT. IgnorePeriod is an Integer[4]. Recommended default value: **8000**.

A.2.7 HammerWait

An Address Cast may request a *port hammering* session, to commence precisely HammerWait milliseconds after the supplied Timestamp. HammerWait is an Integer[4]. Recommended default value: **10000**.

A.2.8 HammerShots

A *port hammering* session requested via Address Cast will consist of Prods sent to the IP in the supplied Address, but with a sequence of randomly-generated 16-bit *port* numbers. A HammerShots number of such attempts is made in response to one such request. (See § A.1.7.) HammerShots is an Integer[4]. Recommended default value: **10000**.