Network Working Group   L. Wood
Internet-Draft   Surrey alumni
Intended status: Experimental   W. Eddy
Expires: October 21, 2014   MTI Systems
                                                              C. Smith
                                                               Vallona
                                                           W. Ivancic
                                                                  NASA
                                                           C. Jackson
                                                                  SSTL
                                                        April 19, 2014

                  Saratoga: A Scalable Data Transfer Protocol
                         draft-wood-tsvwg-saratoga-15

Abstract

   This document specifies the Saratoga transfer protocol. Saratoga was
   originally developed to transfer remote-sensing imagery efficiently
   from a low-Earth-orbiting satellite constellation, but is useful for
   many other scenarios, including ad-hoc peer-to-peer communications,
   delay-tolerant networking, and grid computing. Saratoga is a simple,
   lightweight, content dissemination protocol that builds on UDP, and
   optionally uses UDP-Lite. Saratoga is intended for use when moving
   files or streaming data between peers which may have permanent,
   sporadic or intermittent connectivity, and is capable of transferring
   very large amounts of data reliably under adverse conditions. The
   Saratoga protocol is designed to cope with highly asymmetric link or
   path capacity between peers, and can support fully-unidirectional
   data transfer if required. Saratoga can also cope with very large
   files for exascale computing. In scenarios with dedicated links,
   Saratoga focuses on high link utilization to make the most of limited
   connectivity times, while standard congestion control mechanisms can
   be implemented for operation over shared links. Loss recovery is
   implemented via a simple negative-ack ARQ mechanism. The protocol
   specified in this document is considered to be appropriate for
   experimental use on private IP networks.

Internet-Drafts are draft documents valid for a maximum of six months
and may be updated, replaced, or obsoleted by other documents at any
time. It is inappropriate to use Internet-Drafts as reference
material or to cite them other than as "work in progress."

This Internet-Draft will expire on October 21, 2014.

Copyright Notice

Table of Contents

1. Background and Introduction

Saratoga is a file transfer and content dissemination protocol
capable of efficiently sending both small (kilobyte) and extremely
large (yottabyte) files, as well as streaming continuous content.
Saratoga was originally designed for the purpose of large file
transfer from small low-Earth-orbiting satellites. It has been used
in daily operations since 2004 to move mission imaging data files of
the order of several hundred megabytes each from the Disaster
Monitoring Constellation (DMC) remote-sensing satellites to ground
stations.

The DMC satellites, built at the University of Surrey by Surrey
Satellite Technology Ltd (SSTL), all use IP for payload
communications and delivery of Earth imagery. At the time of this
writing, in November 2013, nine DMC satellites have been launched
into orbit since 2002, five of those are currently operational in
orbit, and three more are under construction. The DMC satellites use
Saratoga to provide Earth imagery under the aegis of the
International Charter on Space and Major Disasters.

An orbital pass giving a period of visibility and connectivity
between a satellite and ground station offers an 8-12 minute time
window in which to transfer imagery files, using a minimum of an 8.1
Mbps downlink and a 9.6 kbps uplink. Newer operational DMC
satellites can use faster downlinks, capable of 20, 40, 80, 105 or
210 Mbps [Brenchley12]. Planned DMC satellites are expected to use
downlinks at up to 320 Mbps, without significant increases in uplink
rates. SSTL's TechDemoSat-1 satellite plans to use a 400 Mbps
downlink [Brenchley12]. This high degree of link asymmetry, with the
need to fully utilize the available downlink capacity to move the
volume of data required within the limited time available, motivates
much of Saratoga's design.

Further details on how these DMC satellites use IP to communicate
with the ground and the terrestrial Internet are discussed elsewhere
[Hogie05][Wood07a]. Saratoga has also been implemented for use in
high-speed private ground networks supporting radio astronomy sensors
[Wood11].

Store-and-forward delivery relies on reliable hop-by-hop transfers of
files, removing the need for the final receiver to talk to the
original sender across long delays and allowing for the possibility
that an end-to-end path may never exist between sender and receiver
at any given time. Breaking an end-to-end path into multiple hops
allows data to be transferred as quickly as possible across each
link; congestion on a longer Internet path is then not detrimental to
the transfer rate on a space downlink. Use of store-and-forward hop-
by-hop delivery is typical of scenarios in space exploration for both
near-Earth and deep-space missions, and useful for other scenarios,
such as underwater networking, ad-hoc sensor networks, and some
message-ferrying relay scenarios. Saratoga is intended to be useful
for relaying data in these scenarios.

Saratoga can optionally also be used to carry the Bundle Protocol
"bundles" intended for Delay and Disruption-Tolerant Networking (DTN)
by the IRTF DTN Research Group [RFC5050]. This has been tested from
orbit using the UK-DMC satellite [Ivancic10]. How Saratoga can
optionally function as a "bundle convergence layer" alongside a DTN
bundle agent is specified in a companion document
[I-D.wood-dtnrg-saratoga].

Saratoga contains a Selective Negative Acknowledgement (SNACK)
'holestofill' mechanism to provide reliable retransmission of data.
This is intended to correct losses of corrupted link-layer frames due
to channel noise over a space link. Packet losses in the DMC are due
to corruption introducing non-recoverable errors in the frame. The
DMC design uses point-to-point links and scheduling of applications
in order, so that the link is dedicated to one application transfer
at a time, meaning that packet loss cannot be due to congestion when
applications compete for link capacity simultaneously. In other
wireless environments that may be shared by many nodes and
applications, allocation of channel resources to nodes becomes a MAC-
layer function. Forward Error Coding (FEC) to get the most reliable
transmission through a channel is best left near the physical layer
so that it can be tailored for the channel. Use of FEC complements
Saratoga's transport-level negative-acknowledgement approach that
provides a reliable ARQ mechanism.

Saratoga is scalable in that it is capable of efficiently
transferring small or large files, by choosing a width of file offset
descriptor appropriate for the filesize, and advertising accepted
offset descriptor sizes. 16-bit, 32-bit, 64-bit and 128-bit
descriptors can be selected, for maximum file sizes of 64KiB-1 (<64
Kilobytes of disk space), 4GiB-1 (<4 Gigabytes), 16EiB-1 (<16
Exabytes) and 256 EiEiB-1 (<256 Exa-exabytes) respectively.

Earth imaging files currently transferred by Saratoga are mostly up
to a few gigabytes in size. Some implementations do transfer more
than 4 GiB in size, and so require offset descriptors larger than 32
bits. We believe that supporting a 128-bit descriptor can satisfy
many future Big Data needs, but we expect current implementations to
only support up to 32-bit or 64-bit descriptors, depending on their
application needs. The 16-bit descriptor is useful for small
messages, including messages from 8-bit devices, and is always
supported. The 128-bit descriptor can be used for moving very large
files stored on a 128-bit filesystem, such as on OpenSolaris ZFS.

As a UDP-based protocol, Saratoga can be used with either IPv4 or
IPv6. Compatibility between Saratoga and the wide variety of links
that can already carry IP traffic is assured.

High link utilization is important during periods of limited
connectivity. Given that Saratoga was originally developed for
scheduled peer-to-peer communications over dedicated links in private
networks, where each application has the entire link for the duration
of its transfer, many Saratoga implementations deliberately lack any
form of congestion control and send at line rate to maximise
throughput and link utilisation in their limited, carefully
controlled, environments. In accordance with UDP Guidelines
[RFC5405] for protocols able to traverse the public Internet, newer
implementations may perform TCP-Friendly Rate Control (TFRC)
[RFC5348] or other congestion control mechanisms. This is described
further in [I-D.wood-tsvwg-saratoga-congestion-control].

Saratoga was originally implemented as outlined in [Jackson04], but
the specification given here differs substantially, as we have added
a number of capabilities while cleaning up the initial Saratoga
specification. The original Saratoga code uses a version number of
0, while code that implements this version of the protocol advertises
a version number of 1. Further discussion of the history and
development of Saratoga is given in [Wood07b].

This document contains an overview of the transfer process and
sessions using Saratoga in Section 2, followed by a formal definition
of the packet types used by Saratoga in Section 4, and the details of
the various protocol mechanisms in Section 6.

Here, Saratoga session types are labelled with underscores around
lowercase names (such as a "_get_" session), while Saratoga packet
types are labelled in all capitals (such as a "REQUEST" packet) in
order to distinguish between the two.

The remainder of this specification uses 'file' as a shorthand for
'binary object', which may be a file, or other type of data, such as

a DTN bundle. This specification uses 'file' when also discussing
streaming of data of indeterminate length. Saratoga uses unsigned
integers in its fields, and does not use signed types.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
"SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this
document are to be interpreted as described in RFC 2119. [RFC2119]

2. Overview of Saratoga File Transfer

Saratoga is a peer-to-peer protocol in the sense that multiple files
may be transferred in both directions simultaneously between two
communicating Saratoga peers, and there is not intended to be a
strict client-to-server relationship.

Saratoga nodes can act as simple file servers. Saratoga supports
several types of operations on files including "pull" downloads,
"push" uploads, directory listing, and deletion requests. Each
operation is handled as a distinct "session" between the peers.

Saratoga nodes MAY advertise their presence, capabilities, and
desires by sending BEACON packets. These BEACONs are sent to either
a reserved, unforwardable, multicast address when using IPv4, or a
link-local all-Saratoga-peers multicast address when using IPv6. A
BEACON might also be unicast to another known node as a sort of
"keepalive". Saratoga nodes may dynamically discover other Saratoga
nodes, either through listening for BEACONs, through pre-
configuration, via some other trigger from a user, lower-layer
protocol, or another process. The BEACON is useful in many
situations, such as ad-hoc networking, as a simple, explicit,
confirmation that another node is present; a BEACON is not required
in order to begin a Saratoga session.. BEACONs have been used by the
DMC satellites to indicate to ground stations that a link has become
functional, a solid-state data recorder is online, and the software
is ready to transfer any requested files.

A Saratoga session begins with either a _get_, _put_, _getdir_, or
delete session REQUEST packet corresponding to a desired download,
upload, directory listing, or deletion operation. _put_ sessions may
instead begin directly with METADATA and DATA, without an initial
REQUEST/OKAY STATUS exchange; these are known as 'blind puts'. The
most common envisioned session is the _get_, which begins with a
single Saratoga REQUEST packet sent from the peer wishing to receive
the file, to the peer who currently has the file. If the session is
rejected, then a brief STATUS packet that conveys rejection is
generated. If the file-serving peer accepts the session, an OKAY
STATUS can be optional; the peer can immediately generate and send a

more useful descriptive METADATA packet, along with some number of
DATA packets constituting the requested file.

These DATA packets are finished by (and can intermittently include) a
DATA packet with a flag bit set that demands the file-receiver send a
reception report in the form of a STATUS packet. This DATA-driven
cycle is shown in Figure 1. The STATUS packet can include
'holestofill' Selective Negative Acknowledgement (SNACK) information
listing spans of octets within the file that have not yet been
received, as well as whether or not the METADATA packet was received,
or an error code terminating the transfer session. Once the
information in this STATUS packet is received, the file-sender can
begin a cycle of selective retransmissions of missing DATA packets,
until it sees a STATUS packet that acknowledges total reception of
all file data.

```
    AT SENDER                        AT RECEIVER
   +---------+
   |  START  |
   +---------+
       |     STATUS is processed when it arrives.
 ----->|<------------------------------\
/      |                               |
| +---------+                          |
| |  DATA   |<--------------------     |
| +---------+                     \    |
|      |    \  repeat until STATUS |   |
|      |     \ request or until end|   |
|      |      \      of DATA       /   |
|      |       --------------------     |
| +---------+                  +---------+
| |  DATA*  |----------------->| STATUS  | can include HOLESTOFILL
| +---------+ STATUS requested   +---------+ can include error code
|      |         regularly from receiver
\      /        while sending DATA packets
 ------          * request flag set
```
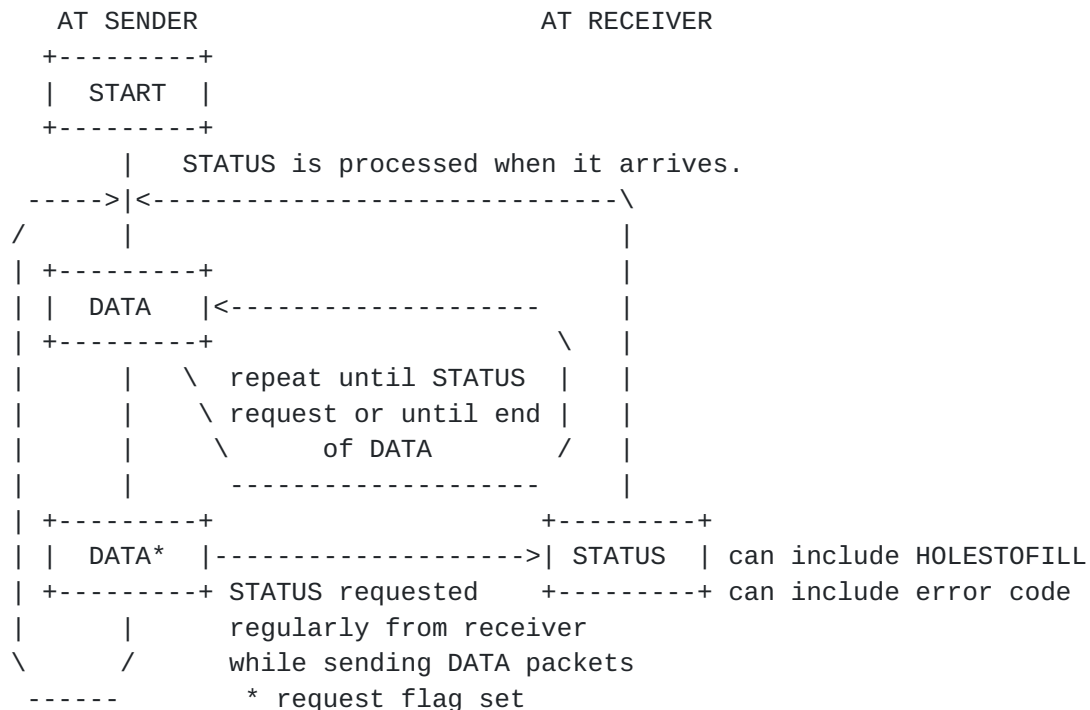
                   Figure 1: STATUS and DATA cycle

In the example scenario in Figure 2, a _get_ request is granted. The
reliable file delivery experiences loss of a single DATA packet due
to channel-induced errors.

```
        File-Receiver              File-Sender

     GET REQUEST --------------------->

     (indicating error/reject) <---- STATUS

                                  or

                         <------- METADATA
              <-------------------- DATA #1
         STATUS ----------------> (voluntarily sent at start)
                  (lost) <------ DATA #2
              <-------------------- DATA #3 (bit set
                                            requesting STATUS)
         STATUS ---------------->
     (indicating that range in DATA #2 was lost)
              <---------------------- DATA #2 (bit set
                                            requesting STATUS)
         STATUS ---------------->
     (complete file and METADATA received)
```

Figure 2: Example _get_ session sequence

A _put_ is similar to _get_, although once the OKAY STATUS is
received, DATA is sent from the peer that originated the _put_
request. A 'blind _put_' does not require an REQUEST and OKAY STATUS
to be exchanged before sending DATA packets, and is efficient for
long-delay or unidirectional links.

A _getdir_ request proceeds similarly, though the DATA transfer
contains a directory record with one or more directory entries,
described later, rather than a given file's bytes. _getdir_ is the
only request to also apply to directories, where one or more
directory entries for individual files is received.

The STATUS and DATA packets are allowed to be sent at any time within
the scope of a session, in order for the file-sending node to
optimize buffer management and transmission order. For example, if
the file-receiver already has the first part of a file from a
previous disrupted transfer, it may send a STATUS at the beginning of
the session indicating that it has the first part of the file, and so
only needs the last part of the file. Thus, efficient recovery from
interrupted sessions between peers becomes possible, similar to
ranged FTP and HTTP requests. (Note that METADATA with a checksum is
useful to verify that the parts are of the same file and that the
file is reassembled correctly.)

The Saratoga 'blind _put_' session is initiated by the file-sender
sending an optional METADATA packet followed by immediate DATA
packets, without requiring a REQUEST or waiting for a STATUS
response. This can be considered an "optimistic" mode of protocol
operation, as it assumes the implicit session request will be
granted. If the sender of a PUT request sees a STATUS packet
indicating that the request was declined, it MUST stop sending any
DATA packets within that session immediately. Since this type of
put is open-loop for some period of time, it should not be used in
scenarios where congestion is a valid concern; in these cases, the
file-sender should wait on its METADATA to be acknowledged by a
STATUS before sending DATA packets within the session.

Figure 3 illustrates the sequence of packets in an example _put_
session, beginning directly with METADATA and DATA as in a blind put,
where the second DATA packet is lost. Other than the way that it is
initiated, the mechanics of data delivery of a blind _put_ session
are similar to a _get_ session.

```
              File-Sender          File-Receiver

               METADATA ---------------->
               DATA  #1 ---------------->
          (transfer accepted) <---------- STATUS
               DATA  #2 ---> (lost)
           DATA  #3 (bit set ------------>
          requesting STATUS)
               (DATA #2 lost) <---------- STATUS
            DATA  #2 (bit set ------------>
          requesting STATUS)
           (transfer complete) <---------- STATUS

           Figure 3: Example PUT session sequence
```

In large-distance scenarios such as for deep space, the large
propagation delays and round-trip times involved discourage use of
ping-pong packet exchanges (such as TCP's SYN/ACK) for starting
sessions, and unidirectional transfers via these optimistic 'blind
_put_s' are desirable. Blind _puts_ are the only mode of transfer
suitable for unidirectional links. Senders sending on unidirectional
links SHOULD send a copy of the METADATA in advance of DATA packets,
and MAY resend METADATA at intervals.

The _delete_ sessions are simple single packet requests that trigger
a STATUS packet with a status code that indicates whether the file
was deleted or not. If the file is not able to be deleted for some
reason, this reason can be conveyed in the Status field of the STATUS
packet.

A _get_ REQUEST packet that does not specify a filename (i.e. the
request contains a zero-length File Path field) is specially defined
to be a request for any chosen file that the peer wishes to send it.
This 'blind _get_' allows a Saratoga peer to request any files that
the other Saratoga peer has ready for it, without prior knowledge of
the directory listing, and without requiring the ability to examine
files or decode remote file names/paths for meaningful information
such as final destination.

If a file is larger than Saratoga can be expected to transfer during
a time-limited contact, there are at least two feasible options:

(1) The application can use proactive fragmentation to create
multiple smaller-sized files. Saratoga can transfer some number of
these smaller files fully during a contact.

(2) To avoid file fragmentation, a Saratoga file-receiver can retain
a partially-transferred file and request transfer of the unreceived
bytes during a later contact. This uses a STATUS packet to make
clear how much of the file has been successfully received and where
transfer should be resumed from, and relies on use of METADATA to
identify the file. On resumption of a transfer, the new METADATA
(including file length, file timestamps, and possibly a file
checksum) MUST match that of the previous METADATA in order to re-
establish the transfer. Otherwise, the file-receiver MUST assume
that the file has changed and purge the DATA payload received during
previous contacts.

Like the BEACON packets, a _put_ or a response to a _get_ MAY be sent
to the dedicated IPv4 Saratoga multicast address (allocated to
224.0.0.108) or the dedicated IPv6 link-local multicast address
(allocated to FF02:0:0:0:0:0:0:6C) for multiple file-receivers on the
link to hear. This is at the discretion of the file-sender, if it
believes that there is interest from multiple receivers. In-progress
DATA transfers MAY also be moved seamlessly from unicast to multicast
if the file-sender learns during a transfer, from receipt of further
unicast _get_ REQUEST packets, that multiple nodes are interested in
the file. The associated METADATA packet is multicast when this
transition takes place, and is then repeated periodically while the
DATA stream is being sent, to inform newly-arrived listeners about
the file being multicast. Acknowledgements MUST NOT be demanded by
multicast DATA packets, to prevent ack implosion at the file-sender,
and instead status SNACK information is aggregated and sent
voluntarily by all file-receivers. File-receivers respond to
multicast DATA with multicast STATUS packets. File-receivers SHOULD
introduce a short random delay before sending a multicast STATUS
packet, to prevent ack implosion after a channel-induced loss, and
MUST listen for STATUS packets from others, to avoid duplicating fill

requests. The file-sender SHOULD repeat any initial unicast portion
of the transfer as multicast last of all, and may repeat and cycle
through multicast of the file several times while file-receivers
express interest via STATUS or _get_ packets. Once in multicast and
with METADATA being repeated periodically, new file-receivers do not
need to send individual REQUEST packets. If a transfer has been
started using UDP-Lite and new receivers indicate UDP-only
capability, multicast transfers MUST switch to using UDP to
accommodate them.

3. Optional Parts of Saratoga

Implementing support for some parts of Saratoga is optional. These
parts are grouped into three sections, namely useful capabilities in
Saratoga that are likely to be supported by implementations,
congestion control that is needed in shared networks and across the
public Internet, and functionality requiring other protocols that is
less likely to be supported.

3.1. Optional but useful functions in Saratoga

These are useful capabilities in Saratoga that implementations SHOULD
support, but may not, depending on scenarios:

- sending and parsing BEACONs.

- sending METADATA. However, sending and receiving METADATA is
considered extremely useful, is strongly recommended, and SHOULD be
done. A METADATA that is received MUST be parsed.

- streaming data, including real-time streaming of content of unknown
length. This streaming can be unreliable (without resend requests)
or reliable (with resend requests). Session protocols such as http
expect reliable streaming. Although Saratoga data delivery is
inherently one-way, where a stream of DATA packets elicits a stream
of STATUS packets, bidirectional duplex communication can be
established by using two Saratoga transfers flowing in opposite
directions.

- multicast DATA transfers, if judged useful for the environment in
which Saratoga is deployed, when multiple receivers are participating
and are receiving the same file or stream.

- sending and parsing STATUS messages, which are expected for
bidirectional communication, but cannot be sent on and are not
required for sending over unidirectional links.

- sending and responding to packet timestamps in DATA and STATUS
  packets. These timestamps are useful for streaming and for giving a
  file-sender an indication of path latency for rate control. There is
  no need for a file-receiver to understand the format used for these
  timestamps for it to be able to receive them from and reflect them
  back to the file-sender.

- support for descriptor sizes greater than 16 bits, for handling
  small files, is optional, as is support for descriptor sizes greater
  than 32 bits, and support for descriptor sizes greater than 64 bits.
  If a descriptor size is implemented, all sizes below that size MUST
  be implemented.

3.2. Optional congestion control

Saratoga can be implemented to perform congestion control at the
sender, based on feedback from acknowledgement STATUS packets
[I-D.wood-tsvwg-saratoga-congestion-control], or have the sender
configured to use simple open-loop rate control to only use a fixed
amount of link capacity. Congestion control is expected to be
undesirable for many of Saratoga's use cases and expected
environmental conditions in private networks, where sending as
quickly as possible or simple rate control at a fixed output speed
are considered useful.

In accordance with the UDP Guidelines [RFC5405], congestion control
MUST be supported if Saratoga is being used across the public
Internet, and SHOULD be supported in environments where links are
shared by traffic flows. Congestion control may not be supported
across private, single-flow links engineered for performance:
Saratoga's primary use case.

3.3. Optional functionality requiring other protocols

The functionality listed here is useful in rare cases, but requires
use of other, optional, protocols. This functionality MAY be
supported by Saratoga implementations:

- support for working with the Bundle Protocol for Delay-Tolerant
  Networking. Saratoga can optionally also be used to carry the Bundle
  Protocol "bundles" that is proposed for use in Delay and Disruption-
  Tolerant Networking (DTN) by the IRTF DTN Research Group [RFC5050].
  The bundle agent acts as an application driving Saratoga. Use of a
  filesystem is expected. This approach has been tested from orbit
  using the UK-DMC satellite [Ivancic10]. How Saratoga can optionally
  function as a "bundle convergence layer" alongside a DTN bundle agent
  is specified in a companion document [I-D.wood-dtnrg-saratoga].

- transfers permitting some errors in content delivered, using UDP-
Lite [RFC3828]. These can be useful for decreasing delivery time
over unreliable channels, especially for unidirectional links, or in
decreasing computational overhead for the UDP Lite checksum. To be
really usefuly, error tolerance requires that lower-layer frames
permit delivery of unreliable data, while header information is still
checked to assure that e.g. destination information is reliable.

If a file contains separate parts that require reliable transmission
without errors or that can tolerate errors in delivered content,
proactive fragmentation can be used to split the file into separate
reliable and unreliable files that can be transferred separately,
using UDP or UDP-Lite.

If parts of a file require reliability but the rest can be sent by
unreliable transfer, the file-sender can use its knowledge of the
internal file structure and vary DATA packet size so that the
reliable parts always start after the offset field and are covered by
the UDP-Lite checksum.

A file that permits unreliable delivery can be transferred onwards
using UDP. If the current sender does not understand the internal
file format to be able to decide what parts must be protected with
payload checksum coverage, the current sender or receiver does not
support UDP-Lite, or the current protocol stack only implements
error-free frame delivery below the UDP layer, then the file MAY be
delivered using UDP.

4. Packet Types

Saratoga is defined for use with UDP over either IPv4 or IPv6
[RFC0768]. UDP checksums, which are mandatory with IPv6, MUST be
used with IPv4. Within either version of IP datagram, a Saratoga
packet appears as a typical UDP header followed by an octet
indicating how the remainder of the packet is to be interpreted:

```
                    1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      UDP source port          |      UDP destination port     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          UDP length           |          UDP checksum         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Vers |Pckt Type| other Saratoga fields ... //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+//
```

Saratoga data transfers can also be carried out using UDP-Lite
[RFC3828]. If Saratoga can be carried over UDP-Lite, the

implementation MUST also support UDP. All packet types except DATA
MUST be sent using UDP with checksums turned on. For reliable
transfers, DATA packets are sent using UDP with checksums turned on.
For files where unreliable transfer has been indicated as desired and
possible, the sender MAY send DATA packets unreliably over UDP-Lite,
where UDP-Lite protects only the Saratoga headers and parts of the
file that must be transmitted reliably.

The three-bit Saratoga version field ("Ver") identifies the version
of the Saratoga protocol that the packet conforms to. The value 001
MUST be used in this field for implementations conforming to the
specification in this document, which specifies version 1 of
Saratoga. The value 000 was used in earlier implementations, prior
to the formal specification and public submission of the protocol
design, and is incompatible with version 001 in many respects.

The five-bit Saratoga "Packet Type" field indicates how the remainder
of the packet is intended to be decoded and processed:

```
+---+----------+------------------------------------------------------+
| # | Type     | Use                                                  |
+---+----------+------------------------------------------------------+
| 0 | BEACON   | Beacon packet indicating peer status.                |
| 1 | REQUEST  | Commands peer to start a transfer.                   |
| 2 | METADATA | Carries file transfer metadata.                      |
| 3 | DATA     | Carries octets of file data.                         |
| 4 | STATUS   | responds to REQUEST or DATA. Can signal list of      |
|   |          | unreceived data to sender during a transfer.         |
+---+----------+------------------------------------------------------+
```

Several of these packet types include a Flags field, for which only
some of the bits have defined meanings and usages in this document.
Other, undefined, bits may be reserved for future use. Following the
principle of being conservative in what you send and liberal in what
you accept, a packet sender MUST set any undefined bits to zero, and
a packet recipient MUST NOT rely on these undefined bits being zero
on reception.

The specific formats for the different types of packets are given in
this section. Some packet types contain file offset descriptor
fields, which contain unsigned integers. The lengths of the offset
descriptors are fixed within a transfer, but vary between file
transfers. The size is set for each particular transfer, depending
on the choice of offset descriptor width made in the METADATA packet,
which in turn depends on the size of file being transferred.

In this document, all of the packet structure figures illustrating a
packet format assume 32-bit lengths for these offset descriptor

fields, and indicate the transfer-dependent length of the fields by
using a "(descriptor)" designation within the [field] in all packet
diagrams. That is:

The example 32-bit descriptors shown in all diagrams here

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
[                            (descriptor)                       ]
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

are suitable for files of up to 4GiB - 1 octets in length, and may be
replaced in a file transfer by descriptors using a different length,
depending on the size of file to be transferred:

16-bit descriptor for short files of up to 64KiB - 1 octets in size
(MUST be supported)

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
[         (descriptor)          ]
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

64-bit descriptor for longer files of up to 16EiB - 1 octets in size
(optional)

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
[                            (descriptor)                       /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                     (descriptor, continued)                  ]
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

128-bit descriptor for very long files of up to 256 EiEiB - 1 octets
in size (optional)

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
[                            (descriptor)                       /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                     (descriptor, continued)                  /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                     (descriptor, continued)                  /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                     (descriptor, continued)                  ]
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Descriptors are used for the descriptor size less one octet, e.g.
16-bit for files up to 64KB - 1 octets in size, before switching to
the larger descriptor, e.g. using the 32-bit descriptor for a 64KB
file and larger.

For offset descriptors and types of content being transferred, the
related flag bits in BEACON and REQUEST indicate capabilities, while
in METADATA and DATA those flag bits are used slightly differently,
to indicate the content being transferred.

Saratoga packets are intended to fit within link MTUs to avoid the
inefficiencies and overheads of lower-layer fragmentation. A
Saratoga implementation does not itself perform any form of MTU
discovery, but is assumed to be configured with knowledge of usable
maximum IP MTUs for the link interfaces it uses.

4.1. BEACON

BEACON packets may be multicast periodically by nodes willing to act
as Saratoga peers, or unicast to individual peers to indicate
properties for that peer. Some implementations have sent BEACONS
every 100 milliseconds, but this rate is arbitrary, and should be
chosen to be appropriate for the environment and implementation.

The main purpose for sending BEACONs is to announce the presence of
the node to potential peers (e.g. satellites, ground stations) to
provide automatic service discovery, and also to confirm the activity
or presence of the peer.

The Endpoint Identifier (EID) in the BEACON serves to uniquely
identify the Saratoga peer. Whenever the Saratoga peer begins using
a new IP address, it SHOULD issue a BEACON on it and repeat the
BEACON periodically, to enable listeners to associate the IP address
with the EID and the peer.

Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 1|  Type   |                    Flags                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
[[               Available free space (optional)              ]]
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Endpoint identifier...                              //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+//
```

where

```
+------------+------------------------------------------------------+
| Field      | Description                                          |
+------------+------------------------------------------------------+
| Type       | 0                                                    |
| Flags      | convey whether or not the peer is ready to           |
|            | send/receive, what the maximum supported file size   |
|            | range and descriptor is, and whether and how free    |
|            | space is indicated.                                  |
| Available  | This optional field can be used to indicate the      |
| free space | current free space available for storage.            |
| Endpoint   | This can be used to uniquely identify the sending    |
| identifier | Saratoga peer, or the administrative node that the   |
|            | BEACON-sender is associated with. If Saratoga is     |
|            | being used with a bundle agent, a bundle endpoint ID |
|            | (EID) can be used here.                              |
+------------+------------------------------------------------------+
```

The Flags field is used to provide some additional information about
the peer. The first two octets of the Flags field is currently in
use. The later octet is reserved for future use, and MUST be set to
zero.

The BEACON flags field, expanding a line of flag bits with
descriptions of each flag, is as follows:

BEACON Flags

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|0|1| => Version Field: Saratoga version 1
|     |0|0|0|0|0| => Type field: BEACON Frame designation
|              |X|X| => Descriptor size
|                  |X| => Supports bundles?
|                    |X| => Supports streaming?
|                      |X|X| => Sending files
|                          |X|X| => Receiving files
|                              |X| => Supports UDP Lite?
|                                |X| => Includes free space size?
|                                  |X|X| => Freespace Descriptor
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The two highest-order bits (bits 8 and 9 above) indicate the maximum
supported file size parameters that the peer's Saratoga
implementation permits. Other Saratoga packet types contain
variable-length fields that convey file sizes or offsets into a file
-- the file offset descriptors. These descriptors may be 16-bit,
32-bit, 64-bit, or 128-bit in length, depending on the size of the

file being transferred and/or the integer types supported by the
sending peer.

The indicated bounds for the possible values of these bits are
summarized below:

```
+-------+-------+------------------------+------------------+
| Bit 8 | Bit 9 | Supported Field Sizes  | Maximum File Size |
+-------+-------+------------------------+------------------+
| 0     | 0     | 16 bits                | 2^16 - 1 octets.  |
| 0     | 1     | 16 or 32 bits          | 2^32 - 1 octets.  |
| 1     | 0     | 16, 32, or 64 bits     | 2^64 - 1 octets.  |
| 1     | 1     | 16, 32, 64, or 128 bits | 2^128 - 1 octets. |
+-------+-------+------------------------+------------------+
```

If a Saratoga peer advertises it is capable of receiving a certain
size of file, then it MUST also be capable of receiving files sent
using smaller descriptor values. This avoids overhead on small
files, while increasing interoperability between peers.

It is likely when sending unbounded streams that a larger offset
descriptor field size will be preferred to minimise problems with
offset sequence numbers wrapping. Protecting against sequence number
wrapping is discussed in the STATUS section.

```
+-----+-------+-------------------------------------------------------+
| Bit | Value | Meaning                                               |
+-----+-------+-------------------------------------------------------+
| 10  | 0     | not able to pass bundles to a local bundle agent;     |
|     |       | handles files only.                                   |
| 10  | 1     | handles files, but can also pass marked bundles to    |
|     |       | a local bundle agent.                                 |
+-----+-------+-------------------------------------------------------+
```

Bit 10 is reserved for DTN bundle agent use, indicating whether the
sender is capable of handling bundles via a local bundle agent. This
is described in [I-D.wood-dtnrg-saratoga].

```
+-----+-------+------------------------------------+
| Bit | Value | Meaning                            |
+-----+-------+------------------------------------+
| 11  | 0     | not capable of supporting streaming. |
| 11  | 1     | capable of supporting streaming.    |
+-----+-------+------------------------------------+
```

Bit 11 is used to indicate whether the sender is capable of sending
and receiving continuous streams.

```
+--------+--------+------------------------------------------------+
| Bit 12 | Bit 13 | Capability and willingness to send files       |
+--------+--------+------------------------------------------------+
| 0      | 0      | cannot send files at all.                      |
| 0      | 1      | invalid.                                       |
| 1      | 0      | capable of sending, but not willing right now. |
| 1      | 1      | capable of and willing to send files.          |
+--------+--------+------------------------------------------------+


+-------+-------+--------------------------------------------------+
| Bit   | Bit   | Capability and willingness to receive files      |
| 14    | 15    |                                                  |
+-------+-------+--------------------------------------------------+
| 0     | 0     | cannot receive files at all.                     |
| 0     | 1     | invalid.                                         |
| 1     | 0     | capable of receiving, but unwilling. Will reject |
|       |       | METADATA or DATA packets.                        |
| 1     | 1     | capable of and willing to receive files.         |
+-------+-------+--------------------------------------------------+
```

Also in the Flags field, bits 12 and 14 act as capability bits, while bits 13 and 15 augment those flags with bits indicating current willingness to use the capability.

Bits 12 and 13 deal with sending, while bits 14 and 15 deal with receiving. If bit 12 is set, then the peer has the capability to send files. If bit 14 is set, then the peer has the capability to receive files. Bits 13 and 15 indicate willingness to send and receive files, respectively.

A peer that is able to act as a file-sender MUST set the capability bit 12 in all BEACONs that it sends, regardless of whether it is willing to send any particular files to a particular peer at a particular time. Bit 13 indicates the current presence of data to send and a willingness to send it in general, in order to augment the capability advertised by bit 12.

If bit 14 is set, then the peer is capable of acting as a receiver, although it still might not currently be ready or willing to receive files (for instance, it may be low on free storage). This bit MUST be set in any BEACON packets sent by nodes capable of acting as file-receivers. Bit 15 augments this by expresses a current general willingness to receive and accept files.

```
+-----+-------+------------------------------------------------------+
| Bit | Value | Meaning                                              |
+-----+-------+------------------------------------------------------+
| 16  | 0     | supports DATA transfers over UDP only.               |
| 16  | 1     | supports DATA transfers over both UDP and UDP-Lite.  |
+-----+-------+------------------------------------------------------+
```

Bit 16 is used to indicate whether the sender is capable of sending
and receiving unreliable transfers via UDP-Lite.

```
+-----+-------+------------------------------------------------------+
| Bit | Value | Meaning                                              |
+-----+-------+------------------------------------------------------+
| 17  | 0     | available free space is not advertised in this       |
|     |       | BEACON.                                              |
| 17  | 1     | available free space is advertised in this BEACON.   |
+-----+-------+------------------------------------------------------+
```

Bit 17 is used to indicate whether the sender includes an optional
field in this BEACON packet that tells how much free space is
available. If bit 17 is set, then bits 18 and 19 are used to
indicate the size in bits of the optional free-space-size field. If
bit 17 is not set, then bits 18 and 19 are zero.

```
        +--------+--------+-------------------------+
        | Bit 18 | Bit 19 | Size of free space field |
        +--------+--------+-------------------------+
        | 0      | 0      | 16 bits.                |
        | 0      | 1      | 32 bits.                |
        | 1      | 0      | 64 bits.                |
        | 1      | 1      | 128 bits.               |
        +--------+--------+-------------------------+
```

The free space field size can vary as indicated by a varying-size
field indicated in bits 18 and 19 of the flags field. Unlike other
offset descriptor use where the value in the descriptor indicates a
byte or octet position for retransmission, or gives a file size in
bytes, this particular field indicates the available free space in
KIBIBYTES (KiB, multiples of 1024 octets), rather than octets.
Available free space is rounded down to the nearest KiB, so
advertising zero means that less than 1KiB is free and available.
Advertising the maximum size possible in the field means that more
free space than that is available. While this field is intended to
be scalable, it is expected that 32 bits (up to 4TiB) will be most
common in use.

A BEACON unicast to an individual peer MAY choose to indicate the
free space available for use by that particular peer, and MAY

indicate capabilities only available to that particular peer,
overriding or supplementing the properties advertised to all local
peers by multicast BEACONs.

Any type of host identifier can be used in the endpoint identifier
field, as long as it is a reasonably unique string within the range
of operational deployment. This field encompasses the remainder of
the packet, and might contain non-UTF-8 and/or null characters.

4.2. REQUEST

A REQUEST packet is an explicit command to perform either a _put_,
get, _getdir_, or _delete_ session.

Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 1|  Type   |          Flags          | Request Type |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Session Id                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 variable-length File Path ...                 /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/                                                              /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/               |   null byte   |                              /
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
/    variable-length Authentication Field (optional)   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

where

```
+---------+---------------------------------------------------------+
| Field   | Description                                             |
+---------+---------------------------------------------------------+
| Type    | 1                                                       |
| Flags   | provide additional information about the requested      |
|         | file/operation; see table below for definition.         |
| Request | identifies the type of request being made; see table    |
| Type    | further below for request values.                       |
| Id      | uniquely identifies the session between two peers.       |
| File    | the path of the requested file/directory following the  |
| Path    | rules described below.                                   |
+---------+---------------------------------------------------------+
```

The Id that is used during sessions serves to uniquely associate a
given packet with a particular sessions. This enables multiple
simultaneous data transfer or request/status sessions between two
peers, with each peer deciding how to multiplex and prioritise the
parallel flows it sends. The Id for a session is selected by the
initiator so as to not conflict with any other in-progress or recent
sessions with the same host. This Id should be unique and generated
using properties of the file, which will remain constant across a
host reboot. The 3-tuple of both host identifiers and a carefully-
generated session Id field can be used to uniquely index a particular
session's state.

The REQUEST flags field, expanding a line of flag bits with
descriptions of each flag, is as follows:

REQUEST Flags

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|0|1| => Version field: Saratoga version 1
|     |0|0|0|0|1| => Type field: REQUEST Frame designation
|             |X|X| => Descriptor size
|                 |X| => Supports bundles?
|                   |X| => Supports streaming?
|                             |X| => Supports UDP Lite?
|                       Request Type field <= |X|X|X|X|X|X|X|X|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

In the Flags field, the bits labelled 8 and 9 in the figure above
indicate the maximum supported file length fields that the peer can
handle, and are interpreted exactly as the bits 8 and 9 in the BEACON
packet described above. Bits 12 and 13, and 14 and 15, indicate
capability and willingness to send and receive files, as described
above. Making a _get_ request would require that the requester is
capable and willing to receive files. The remaining defined
individual bits are as summarised as follows:

```
+-----+-------+------------------------------------------------------+
| Bit | Value | Meaning                                              |
+-----+-------+------------------------------------------------------+
| 10  | 0     | The requester cannot handle bundles locally.         |
| 10  | 1     | The requester can handle bundles.                    |
| 11  | 0     | The requester cannot receive streams.                |
| 11  | 1     | The requester is also able to receive streams.       |
| 16  | 0     | The requester is able to receive DATA over UDP       |
|     |       | only.                                                |
| 16  | 1     | The requester is also able to receive DATA over      |
|     |       | UDP-Lite.                                            |
+-----+-------+------------------------------------------------------+
```

The Request Type field is an octet that contains a value indicated
the type of request being made. Possible values are:

```
+-------+--------------------------------------------------------------+
| Value | Meaning                                                      |
+-------+--------------------------------------------------------------+
| 0     | No action is to be taken; similar to a BEACON.               |
| 1     | A _get_ session is requested. The File Path field holds      |
|       | the name of the file to be sent.                            |
| 2     | A _put_ session is requested. The File Path field           |
|       | suggests the name of the file that will be delivered only    |
|       | after an OK STATUS is received from the file receiver.       |
| 3     | A _get_ session is requested, and once received             |
|       | successfully, the original copy should be deleted. The       |
|       | File Path field holds the name of the file to be sent.       |
|       | (This get+delete is known as a 'take'.)                     |
| 4     | A _put_ session is requested, and once sent successfully,    |
|       | the original copy will be deleted. The File Path field       |
|       | holds the name of the file to be sent. (This put+delete      |
|       | is known as a 'give'.)                                       |
| 5     | A _delete_ session is requested, and the File Path field     |
|       | specifies the name of the file to be deleted.               |
| 6     | A _getdir_ session is requested. The File Path field         |
|       | holds the name of the directory or file on which the         |
|       | directory record is created.                                |
+-------+--------------------------------------------------------------+
```

The File Path portion of a _get_ packet is a null-terminated UTF-8
encoded string [RFC3629] that represents the path and base file name
on the file-sender of the file (or directory) that the file-receiver
wishes to perform the _get_, _getdir_, or _delete_ operation on.
Implementations SHOULD only send as many octets of File Path as are
needed for carrying this string, although some implementations MAY
choose to send a fixed-size File Path field in all REQUEST packets
that is filled with null octets after the last UTF-8 encoded octet of

the path. A maximum of 1024 octets for this field, and for the File
Path fields in other Saratoga packet types, is used to limit the
total packet size to within a single IPv6 minimum MTU (minus some
padding for network layer headers), and thus avoid the need for
fragmentation. The 1024-octet maximum applies after UTF-8 encoding
and null termination.

As in the standard Internet File Transfer Protocol (FTP) [RFC0959],
for path separators, Saratoga allows the local naming convention on
the peers to be used. There are security implications to processing
these strings without some intelligent filtering and checking on the
filesystem items they refer to. See also the Security Considerations
section later within this document.

If the File Path field is empty, i.e. is a null-terminated zero-
length string one octet long, then this indicates that the file-
receiver is ready to receive any file that the file-sender would like
to send it, rather than requesting a particular file. This allows
the file-sender to determine the order and selection of files that it
would like to forward to the receiver in more of a "push" manner. Of
course, file retrieval could also follow a "pull" manner, with the
file-receiving host requesting specific files from the file-sender.
This may be desirable at times if the file-receiver is low on storage
space, or other resources. The file-receiver could also use the
Saratoga _getdir_ session results in order to select small files, or
make other optimizations, such as using its local knowledge of
contact times to pick files of a size likely to be able to be
delivered completely. File transfer through pushing sender-selected
files implements delivery prioritization decisions made solely at the
Saratoga file-sending node. File transfer through pulling specific
receiver-selected files implements prioritization involving more
participation from the Saratoga file-receiver. This is how Saratoga
implements Quality of Service (QoS).

The null-terminated File Path string MAY be followed by an optional
Authentication Field that can be used to validate the REQUEST packet.
Any value in the Authentication Field is the result of a computation
of packet contents that SHOULD include, at a minimum, source and
destination IP addresses and port numbers and packet length in a
'pseudo-header', as well as the content of all Saratoga fields from
Version to File Path, excluding the predictable null-termination
octet. This Authentication Field can be used to allow the REQUEST
receiver to discriminate between other peers, and permit and deny
various REQUEST actions as appropriate. The format of this field is
unspecified for local use.

Combined get+delete (take) and put+delete (give) requests should only
have the delete carried out once the deleting peer is certain that

   the file-receiver has a good copy of the file. This may require the
   file receiver to verify checksums before sending a final STATUS
   message acknowledging successful delivery of the final DATA segment,
   or aborting the transfer if the checksum fails. If the transfer
   fails and an error STATUS is sent for any reason, the file should not
   be deleted.

   REQUEST packets may be sent multicast, to learn about all listening
   nodes. A multicast _get_ request for a file that elicits multiple
   METADATA or DATA responses should be followed by unicast STATUS
   packets with status errors cancelling all but one of the proposed
   transfers. File timestamps in the Directory Entry can be used to
   select the most recent version of an offered file, and the host to
   fetch it from.

   If the receiver already has the file at the expected file path and is
   requesting an update to that file, REQUEST can be sent after a
   METADATA advertising that file, to allow the sender to determine
   whether a replacement for the file should be sent.

   Delete requests are ignored for files currently being transferred.

4.3. METADATA

   METADATA packets are sent as part of a data transfer session (_get_,
   getdir, and _put_). A METADATA packet says how large the file is
   and what its name is, as well as what size of file offset descriptor
   is chosen for the session. METADATA packets are optional, but SHOULD
   be sent. A METADATA packet that is received MUST be parsed. A
   METADATA packet is normally sent at the start of a DATA transfer, but
   can be repeated throughout the transfer. Sending METADATA at the
   start if using checksums allows for incremental checksum calculation
   by the receiver, and is a good idea.

Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 1|  Type   |              Flags           |Sumleng|Sumtype|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Session Id                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                              /
/                                                              /
/      example error-detection checksum (128-bit MD5 shown)    /
/                                                              /
/                                                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                              /
/              single Directory Entry describing file          /
/                        (variable length)                     /
/                                                             //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-//
```

where

```
+-----------+----------------------------------------------------+
| Field     | Description                                        |
+-----------+----------------------------------------------------+
| Type      | 2                                                  |
| Flags     | indicate additional boolean metadata about a file. |
| Sumleng   | indicates the length of a checksum, as a multiple of |
|           | 32 bits.                                           |
| Sumtype   | indicates whether a checksum is present after the Id, |
|           | and what type it is.                               |
| Id        | identifies the session that this packet describes. |
| Checksum  | an example included checksum covering file contents. |
| Directory | describes file system information about the file,  |
| Entry     | including file length, file timestamps, etc.; the  |
|           | format is specified in Section 5.                  |
+-----------+----------------------------------------------------+
```

The first octet of the Flags field is currently specified for use.
The later two octets are reserved for future use, and MUST be set to
zero.

The METADATA flags field is as follows, expanding a line of flag bits
with explanations of each field:

METADATA Flags

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|0|1| => Version Field: Saratoga version 1
|     |0|0|0|1|0| => Type field: METADATA Frame designation
|             |X|X| => Descriptor
|                 |X|X| => File/bundle/stream/dir record
|                     |X| => Transfer in progress?
|                       |X| => UDP Lite permitted?
|       Checksum length in no. of 32-bit words <=|X|X|X|X|
|                     Error detection checksum type <=|X|X|X|X|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

In the Flags field, the bits labelled 8 and 9 in the figure above
indicate the exact size of the offset descriptor fields used in this
particular packet and are interpreted exactly as the bits 8 and 9 in
the BEACON packet described above. The value of these bits
determines the size of the File Length field in the current packet,
as well as indicating the size of the offset fields used in DATA and
STATUS packets within the session that will follow this packet.

```
+-------+-------+---------------------------------------------------+
| Bit   | Bit   | Type of transfer                                  |
| 10    | 11    |                                                   |
+-------+-------+---------------------------------------------------+
| 0     | 0     | a file is being sent.                             |
| 0     | 1     | the file being sent should be interpreted as a    |
|       |       | Directory Record.                                 |
| 1     | 0     | a bundle is being sent.                           |
| 1     | 1     | an indefinite-length stream is being sent.        |
+-------+-------+---------------------------------------------------+
```

Also inside the Flags field, bits 10 and 11 indicate what is being
transferred - a file, special directory record file that contains one
or more directory entries, bundle, or stream. The value 01 indicates
that the METADATA and DATA packets are being generated in response to
a _getdir_ REQUEST, and that the assembled DATA contents should be
interpreted as a Directory Record containing directory entries, as
defined in Section 5.

```
+-----+-------+---------------------------------------------------+
| Bit | Value | Meaning                                           |
+-----+-------+---------------------------------------------------+
| 12  | 0     | This transfer is in progress.                     |
| 12  | 1     | This transfer is no longer in progress, and has   |
|     |       | been terminated.                                  |
+-----+-------+---------------------------------------------------+
```

Bit 12 indicates whether the transfer is in progress, or has been
terminated by the sender. It is normally set to 1 only when METADATA
is resent to indicate that a stream transfer has been ended.

```
+-----+-------+---------------------------------------------------+
| Bit | Value | Meaning                                           |
+-----+-------+---------------------------------------------------+
| 13  | 0     | This file's content MUST be delivered reliably    |
|     |       | without errors using UDP.                         |
| 13  | 1     | This file's content MAY be delivered unreliably, or |
|     |       | partly unreliably, where errors are tolerated,    |
|     |       | using UDP-Lite.                                   |
+-----+-------+---------------------------------------------------+
```

Bit 13 indicates whether the file must be sent reliably or can be
sent at least partly unreliably, using UDP-Lite. This flag SHOULD
only be set if the originator of the file knows that at least some of
the file content is suitable for sending unreliably and is robust to
errors. This flag reflects a property of the file itself. This flag
may still be set if the immediate file-receiver is only capable of
UDP delivery, on the assumption that this preference will be
preserved for later transfers where UDP-Lite transfers may be taken
advantage of by senders with knowledge of the internal file
structure. The file-sender may know that the receiver is capable of
handling UDP-Lite, either from a _get_ REQUEST, from exchange of
BEACONs, or a-priori.

The high four bits of the Flags field, bits 28-31, are used to
indicate if an optional error-detection checksum has been included in
the METADATA for the file to be transferred. Here, bits 0000
indicate that no checksum is present, with the implicit assumption
that the application will do its own end-to-end check. Other values
indicate the type of checksum to use. The choice of checksum depends
on the available computing power and the length of the file to be
checksummed. Longer files require stronger checksums to ensure
error-free delivery. The checksum of the file to be transferred is
carried as shown, with a fixed-length field before the varying-length
File Length and File Name information fields.

Assigned values for the checksum type field are:

```
+-----------+------------------------------------------------------+
| Value     | Use                                                  |
| (0-15)    |                                                      |
+-----------+------------------------------------------------------+
| 0         | No checksum is provided.                             |
| 1         | 32-bit CRC32 checksum, suitable for small files.     |
| 2         | 128-bit MD5 checksum, suitable for larger files.     |
| 3         | 160-bit SHA-1 checksum, suitable for larger files but|
|           | slower to process than MD5.                          |
+-----------+------------------------------------------------------+
```

The length of an optional checksum cannot be inferred from the
checksum type field, particularly for unknown checksum types. The
next-highest four bits of the 32-bit word holding the Flags, bits
24-27, indicate the length of the checksum bit field, as a multiple
of 32 bits.

```
+---------------------+------------------------------------+
| Example Value (0-15) | Use                               |
+---------------------+------------------------------------+
| 0                    | No checksum is provided.          |
| 1                    | 32-bit checksum field, e.g. CRC32.|
| 4                    | 128-bit checksum field, e.g. MD5. |
| 5                    | 160-bit checksum field, e.g. SHA-1.|
+---------------------+------------------------------------+
```

For a 32-bit CRC, the length field holds 1 and the type field holds
1. For MD5, the length field holds 4 and the type field holds 2.
For SHA-1, the length field holds 5 and the type field holds 3.

It is expected that higher values will be allocated to new and
stronger checksums able to better protect larger files. These
checksums can be expected to be longer, with larger checksum length
fields.

A checksum SHOULD be included for files being transferred. The
checksum SHOULD be as strong as possible. Streaming of an
indefinite-length stream MUST set the checksum type field to zero.

It is expected that a minimum of the MD5 checksum will be used,
unless the Saratoga implementation is used exclusively for small
transfers at the low end of the 16-bit file descriptor range, such as
on low-performing hardware, where the weaker CRC-32c checksum can
suffice.

The CRC32 checksum is computed as described for the CRC-32c algorithm
given in [RFC3309].

The MD5 Sum field is generated via the MD5 algorithm [RFC1321],
computed over the entire contents of the file being transferred. The
file-receiver can compute the MD5 result over the reassembled
Saratoga DATA packet contents, and compare this to the METADATA's MD5
Sum field in order to gain confidence that there were no undetected
protocol errors or UDP checksum weaknesses encountered during the
transfer. Although MD5 is known to be less than optimal for security
uses, it remains excellent for non-security use in error detection
(as is done here in Saratoga), and has better performance
implications than cryptographically-stronger alternatives given the
limited available processing of many use cases [RFC6151].

Checksums may be privately keyed for local use, to allow transmission
of authenticated or encrypted files delivered in DATA packets. This
has limitations, discussed further in Section 8 at end.

Use of the checksum to ensure that a file has been correctly relayed
to the receiving node is important. A provided checksum MUST be
checked against the received data file. If checksum verification
fails, either due to corruption or due to the receiving node not
having the right key for a keyed checksum), the file MUST be
discarded. If the file is to be relayed onwards later to another
Saratoga peer, the metadata, including the checksum, MUST be retained
with the file and SHOULD be retransmitted onwards unchanged with the
file for end-to-end coverage. If it is necessary to recompute the
checksum or encrypted data for the new peer, either because a
different key is in use or the existing checksum algorithm is not
supported, the new checksum MUST be computed before the old checksum
is verified, to ensure overlapping checksum coverage and detect
errors introduced in file storage.

METADATA can be used as an indication to update copies of files. If
the METADATA is in response to a _get_ REQUEST including a file
record, and the record information for the held file matches what the
requester already has, as has been indicated by a previously-received
METADATA advertisement from the requester, then only the METADATA is
sent repeating this information and verifying that the file is up to
date. If the record information does not match and a newer file can
be supplied, the METADATA begins a transfer with following DATA
packets to update the file.

4.4. DATA

A series of DATA packets form the main part of a data transfer
session (_get_, _put_, or _getdir_). The payloads constitute the
actual file data being transferred.

Format

```
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |0 0 1|  Type   |                  Flags                        |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                       Session Id                              |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                                                              /
 /          Timestamp/nonce information (optional)             /
 /                                                             /
 /                                                             |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 [                     Offset (descriptor)                      ]
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                       Payload data...                        //
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+//
```

where

Field	Description
Type	3
Flags	are described below.
Id	identifies the session to which this packet belongs.
Timestamp/nonce	is an optional 128-bit field providing timing or identification information unique to this packet. See Appendix A for details.
Offset	the offset in octets to the location where the first byte of this packet's payload is to be written.

The DATA packet has a minimum size of ten octets, using sixteen-bit descriptors and no timestamps.

DATA packets are normally checked by the UDP checksum to prevent errors in either the header or the payload content. However, for transfers that can tolerate content errors, DATA packets MAY be sent using UDP-Lite. If UDP-Lite is used, the file-sender must know that the file-receiver is capable of handling UDP-Lite, and the file contents to be transferred should be resilient to errors. The UDP-Lite checksum MUST protect the Saratoga headers, up to and including the offset descriptor, and MAY protect more of each packet's payload,

depending on the file-sender's knowledge of the internal structure of
the file and the file's reliability requirements.

The DATA flags field is as follows, expanding a line of flag bits
with explanations of each field:

DATA Flags

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|0|1| => Version Field: Saratoga version 1
|     |0|0|0|1|1| => Type field: DATA Frame designation
|              |X|X| => Descriptor
|                  |X|X| => File/bundle/stream/dir record
|                      |X| => Includes timestamp?
|                         |X| => STATUS requested
|                           |X| => End of Data (EOD) set
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Bit 8	Bit 9	Type of transfer
0	0	16-bit descriptors are in use in this transfer.
0	1	32-bit descriptors are in use in this transfer.
1	0	64-bit descriptors are in use in this transfer.
1	1	128-bit descriptors are in use in this transfer.

Flag bits 8 and 9 are set to indicate the size of the offset
descriptor as described for BEACON and METADATA packets, so that each
DATA packet is self-describing. This allows the DATA packet to be
used to construct a file even when an initial METADATA is lost and
must be resent. The flag values for bits 8 and 9 MUST be the same as
indicated in any expected METADATA packet.

Bit 10	Bit 11	Type of transfer
0	0	a file is being sent.
0	1	the file being sent should be interpreted as a directory record.
1	0	a bundle is being sent.
1	1	an indefinite-length stream is being sent.

Also inside the Flags field, bits 10 and 11 indicate what is being
transferred - a file, special file that contains a Directory Records,
bundle, or stream. The value 01 indicates that the METADATA and DATA
packets are being generated in response to a _getdir_ REQUEST, and
that the assembled DATA contents should be interpreted as a Directory
Record containing directory entries, as defined in Section 5. The
flag values for bits 10 and 11 MUST be the same as indicated in the
initial METADATA packet.

```
+-----+-------+------------------------------------------------------+
| Bit | Value | Meaning                                              |
+-----+-------+------------------------------------------------------+
| 12  | 0     | This packet does not include an optional             |
|     |       | timestamp/nonce field.                               |
| 12  | 1     | This packet includes an optional timestamp/nonce     |
|     |       | field.                                               |
+-----+-------+------------------------------------------------------+
```

Flag bit 12 indicates that an optional packet timestamp/nonce is
carried in the packet before the offset field. This packet timestamp
/nonce field is always sixteen octets (128 bits) long. Timestamps
can be useful to the sender even when the receiver does not
understand them, as the receiver can simply echo any provided
timestamps back, as specified for STATUS packets, to allow the sender
to monitor flow conditions. Packet timestamps are particularly
useful when streaming. Packet timestamps are discussed further in
Appendix A.

```
+-----+-------+------------------------------+
| Bit | Value | Meaning                      |
+-----+-------+------------------------------+
| 15  | 0     | No response is requested.    |
| 15  | 1     | A STATUS packet is requested.|
+-----+-------+------------------------------+
```

Within the Flags field, if bit 15 of the packet is set, the file-
receiver is expected to immediately generate a STATUS packet to
provide the file-sender with up-to-date information regarding the
status of the file transfer. This flag is set carefully and rarely.
This flag may be set periodically, but infrequently. Asymmetric
links with constrained backchannels can only carry a limited amount
of STATUS packets before ack congestion becomes a problem. This flag
SHOULD NOT be set if an unreliable stream is being transferred, or if
multicast is in use. This flag SHOULD be set periodically for
reliable file transfers, or reliable streaming. The file-receiver
MUST respond to the flag by generating a STATUS packet, unless it
knows that doing so will lead to local congestion, in which case it
may choose to send a later voluntary STATUS message. Voluntary

STATUS packets MAY be sent if a request for one has not been made
within an appropriate time.

```
+-----+-------+-----------------------------------+
| Bit | Value | Meaning                           |
+-----+-------+-----------------------------------+
| 16  | 0     | Normal use.                       |
| 16  | 1     | The EOD End of Data flag is set.  |
+-----+-------+-----------------------------------+
```

The End of Data flag is set in DATA packets carrying the last byte of
a transfer. This is particularly useful for streams and for the rare
Saratoga implementations that do not send or receive METADATA.

Immediately following the DATA header is the payload, which consumes
the remainder of the packet and whose length is implicitly defined by
the end of the packet. The payload octets are directly formed from
the continuous octets starting at the specified Offset in the file
being transferred. No special coding is performed. A zero-octet
payload length is allowable, and a single DATA packet indicating zero
payload, consisting only of a header with the EOD flag set, may be
useful to simply elicit a STATUS response from the receiver.

The length of the Offset fields used within all DATA packets for a
given session MUST be consistent with the length indicated by bits 8
and 9 of any accompanying METADATA packet. If the METADATA packet
has not yet been received, a file-receiver that supports METADATA
MUST indicate that it has not been received via a STATUS packet, and
MAY choose to enqueue received DATA packets for later processing
after the METADATA arrives.

4.5. STATUS

The STATUS packet type is the single acknowledgement method that is
used for feedback from a Saratoga receiver to a Saratoga sender to
indicate session progress, both as a response to a REQUEST, and as a
response to a DATA packet when demanded or volunteered.

When responding to a DATA packet, the STATUS packet MAY, as needed,
include selective acknowledgement (SNACK) 'hole' information to
enable transmission (usually re-transmission) of specific sets of
octets within the current session (called "holes"). This
'holestofill' information can be used to clean up losses (or indicate
no losses) at the end of, or during, a session, or to efficiently
resume a transfer that was interrupted in a previous session.

Format

```
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |0 0 1|  Type   |             Flags             |     Status    |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                          Session Id                           |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                                                              /
 /           Timestamp/nonce information (optional)            /
 /                                                              /
 /                                                              |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 [             Progress Indicator (descriptor)                  ]
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 [                In-Response-To (descriptor)                   ]
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |               (possibly, several Hole fields)                /
 /                          ...                                 /
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

where

Field	Description
Type	4
Flags	are defined below.
Id	identifies the session that this packet belongs to.
Status	a value of 00 indicates the transfer is sucessfully proceeding. All other values are errors terminating the transfer, explained below.
Zero-Pad	an octet fixed at 00 to allow later fields to be conveniently aligned for processing.
Timestamp (optional)	an optional fixed 128-bit field, that is only present and used to return a packet timestamp if the timestamp flag is set. If the STATUS packet is voluntary and the voluntary flag is set, this should repeat the timestamp of the DATA packet containing the highest offset seen. If the STATUS packet is in response to a mandatory request, this will repeat the timestamp of the requesting DATA packet. The file-sender may use these timestamps to estimate latency. Packet timestamps are particularly useful when streaming. There are special considerations for streaming, discussed further below, to protect against the ambiguity of wrapped offset descriptor sequence numbers. Packet timestamps are discussed further in Appendix A.
Progress Indicator (descriptor)	the offset of the lowest-numbered octet of the file not yet received, and expected.
In-Response-To (descriptor)	the offset of the octet following the DATA packet that generated this STATUS packet, or the offset of the next expected octet following the highest DATA packet seen if this STATUS is generated voluntarily and the voluntary flag is set.
Holes	indications of offset ranges of missing data, defined below.

The STATUS packet has a minimum size of twelve octets, using sixteen-bit descriptors, a progress indicator but no Hole fields, and no timestamps. The progress indicator is always zero when responding to requests that may initiate a transfer.

The Id field is needed to associate the STATUS packet with the
session that it refers to.

The Progress Indicator and In-Response-To fields mark the 'left edge'
and 'right edge' of the incomplete working area where holes are being
filled in. If there are no holes, these fields will hold the same
value. At the start of a transfer, both fields begin by expecting
octet zero. When a transfer has completed successfully, these fields
will contain the length of the file.

The STATUS flags field is as follows, expanding a line of flag bits
with explanations of each field:

STATUS Flags

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|0|1| => Version Field: Saratoga version 1
|     |0|0|1|0|0| => Type field: STATUS Frame designation
|               |X|X| => Descriptor
|                   |X| => Timestamp included?
|                     |X| => METADATA received?
|                       |X| => Hole information complete?
|                         |X| => Voluntary STATUS message?
|                             Status code <= |X|X|X|X|X|X|X|X|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Flags bits 8 and 9 are set to indicate the size of the offset
descriptor as described for BEACON and METADATA packets, so that each
STATUS packet is self-describing. The flag values here MUST be the
same as indicated in the initial METADATA and DATA packets.

Other bits in the Flags field are defined as:

```
  +-----+-------+----------------------------------------------------+
  | Bit | Value | Meaning                                            |
  +-----+-------+----------------------------------------------------+
  | 12  | 0     | This packet does not include a timestamp field.    |
  | 12  | 1     | This packet includes an optional timestamp field.  |
  +-----+-------+----------------------------------------------------+
```

Flag bit 12 indicates that an optional sixteen-byte packet timestamp/
nonce field is carried in the packet before the Progress Indicator
descriptor, as discussed for the DATA packet format. Packet
timestamps are discussed further in Appendix A.

```
+-----+-------+---------------------------------------+
| Bit | Value | Meaning                               |
+-----+-------+---------------------------------------+
| 13  | 0     | file's METADATA has been received.    |
| 13  | 1     | file's METADATA has not been received.|
+-----+-------+---------------------------------------+
```

If bit 13 of a STATUS packet has been set to indicate that the
METADATA has not yet been received, then any METADATA SHOULD be
resent. This flag should normally be clear.

A receiver SHOULD tolerate lost METADATA that is later resent, but
MAY insist on receiving METADATA at the start of a transfer. This is
done by responding to early DATA packets with a voluntary STATUS
packet that sets this flag bit, reports a status error code 10, sets
the Progress Indicator field to zero, and does not include
HOLESTOFILL information.

```
+-----+-------+-----------------------------------------------------+
| Bit | Value | Meaning                                             |
+-----+-------+-----------------------------------------------------+
| 14  | 0     | this packet contains the complete current set of    |
|     |       | holes at the file-receiver.                         |
| 14  | 1     | this packet contains incomplete hole-state; holes   |
|     |       | shown in this packet should supplement other        |
|     |       | incomplete hole-state known to the file-sender.     |
+-----+-------+-----------------------------------------------------+
```

Bit 14 of a 'holestofill' STATUS packet is only set when there are
too many holes to fit within a single STATUS packet due to MTU
limitations. This causes the hole list to be spread out over
multiple STATUS packets, each of which conveys distinct sets of
holes. This could occur, for instance, in a large file _put_
scenario with a long-delay feedback loop and poor physical layer
conditions. These multiple STATUS packets will share In-Response-To
information. When losses are light and/or hole reporting and repair
is relatively frequent, all holes should easily fit within a single
STATUS packet, and this flag will be clear. Bit 14 should normally
be clear.

In some rare cases of high loss, there may be too many holes in the
received data to convey within a single STATUS's size, which is
limited by the link MTU size. In this case, multiple STATUS packets
may be generated, and Flags bit 14 should be set on each STATUS
packet accordingly, to indicate that each packet holds incomplete
results. The complete group of STATUS packets, each containing
incomplete information, will share common In-Response-To information
to distinguish them from any earlier groups.

```
+-----+-------+----------------------------------------------+
| Bit | Value | Meaning                                      |
+-----+-------+----------------------------------------------+
| 15  | 0     | This STATUS was requested by the file-sender. |
| 15  | 1     | This STATUS is sent voluntarily.             |
+-----+-------+----------------------------------------------+
```

Flag bit 15 indicates whether the STATUS is sent voluntarily or due
to a request by the sender. It affects content of the In-Response-To
timestamp and descriptor fields.

In the case of a transfer proceeding normally, immediately following
the STATUS packet header shown above, is a set of "Hole" definitions
indicating any lost packets. Each Hole definition is a pair of
unsigned integers. For a 32-bit offset descriptor, each Hole
definition consists of two four-octet unsigned integers:

Hole Definition Format

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
[              offset to start of hole (descriptor)            ]
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
[               offset to end of hole (descriptor)             ]
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The start of the hole means the offset of the first unreceived byte
in that hole. The end of the hole means the last unreceived byte in
that hole.

For 16-bit descriptors, each Hole definition holds two two-octet
unsigned integers, while Hole definitions for 64- and 128-bit
descriptors require two eight- and two sixteen-octet unsigned
integers respectively.

Holes MUST be listed in order, lowest values first.

Since each Hole definition takes up eight octets when 32-bit offset
lengths are used, we expect that well over 100 such definitions can
fit in a single STATUS packet, given the IPv6 minimum MTU. (There
may be cases where there is a very constrained backchannel compared
to the forward channel streaming DATA packets. For these cases,
implementations might deliberately request large holes that span a
number of smaller holes and intermediate areas where DATA has already
been received, so that previously-received DATA is deliberately
resent. This aggregation of separate holes keeps the backchannel
STATUS packet size down to avoid backchannel congestion.)

A 'voluntary' STATUS can be sent at the start of each session. This
indicates that the receiver is ready to receive the file, or
indicates an error or rejection code, described below. A STATUS
indicating a successfully established transfer has a Progress
Indicator of zero and an In-Response-To field of zero.

On receiving a STATUS packet, the sender SHOULD prioritize sending
the necessary data to fill those holes, in order to advance the
Progress Indicator at the receiver.

4.5.1. Errors and aborting sessions

In the case of an error causing a session to be aborted, the Status
field holds a code that can be used to explain the cause of the error
to the other peer. A zero value indicates that there have been no
significant errors (this is called a "success STATUS" within this
document), while any non-zero value means the session should be
aborted (this is called a "failure STATUS").

```
+----------------+---------------------------------------------+
| Error Code     | Meaning                                     |
| Status Value   |                                             |
+----------------+---------------------------------------------+
| 0x00           | Success, No Errors.                          |
| 0x01           | Unspecified Error.                          |
| 0x02           | Unable to send file due to resource constraints. |
| 0x03           | Unable to receive file due to resource      |
|                | constraints.                                |
| 0x04           | File not found.                             |
| 0x05           | Access Denied.                              |
| 0x06           | Unknown Id field for session.               |
| 0x07           | Did not delete file.                        |
| 0x08           | File length is longer than receiver can support. |
| 0x09           | File offset descriptors do not match expected |
|                | use or file length.                         |
| 0x0A           | Unsupported Saratoga packet type received.  |
| 0x0B           | Unsupported Request Type received.          |
| 0x0C           | REQUEST is now terminated due to an internal |
|                | timeout.                                    |
| 0x0D           | DATA flag bits describing transfer have changed |
|                | unexpectedly.                               |
| 0x0E           | Receiver is no longer interested in receiving |
|                | this file.                                  |
| 0x0F           | File is in use.                             |
| 0x10           | METADATA required before transfer can be    |
|                | accepted.                                   |
| 0x11           | A STATUS error message has been received    |
|                | unexpectedly, so REQUEST is terminated.     |
+----------------+---------------------------------------------+
```

The recipient of a failure STATUS MUST NOT try to process the
Progress Indicator, In-Response-To, or Hole offsets, because, in some
types of error conditions, the packet's sender may not have any way
of setting them to the right length for the session.

5. The Directory Entry

Directory Entries have two uses within Saratoga:

1. Within a METADATA packet, a Directory Entry is used to give
    information about the file being transferred, in order to
    facilitate proper reassembly of the file and to help the file-
    receiver understand how recently the file may have been created
    or modified.

2. When a peer requests a directory record via a _getdir_ REQUEST,
    the other peer generates a file containing a series of one or

more concatenated Directory Entry records, and transfers this
file as it would transfer the response to a normal _get_ REQUEST,
sending the records together within DATA packets. This file may
be either temporary or within-memory and not actually a part of
the host's file system itself.

Directory Entry Format

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|      Properties         [       Size (descriptor)      ]
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Mtime file modification time (using year 2000 epoch)       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Ctime file creation time (using year 2000 epoch)       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                              /
+                                                              /
/                                                              /
/           File Path (max 1024 octets,variable length)       /
/                                                        ... //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-//
```

where

```
+------------+-------------------------------------------------------+
| field      | description                                           |
+------------+-------------------------------------------------------+
| Properties | if set, bit 7 of this field indicates that the entry  |
|            | corresponds to a directory. Bit 6, if set, indicates  |
|            | that the file is "special". A special file may not    |
|            | be directly transferable as it corresponds to a       |
|            | symbolic link, a named pipe, a device node, or some   |
|            | other "special" filesystem object.  A file-sender     |
|            | may simply choose not to include these types of       |
|            | files in the results of a _getdir_ request.  Bits 8   |
|            | and 9 are flags that indicate the width of the        |
|            | following descriptor field that gives file size. Bit  |
|            | 10 indicates that the file is to be handled by        |
|            | Saratoga as a bundle, and passed to a bundle agent.   |
| Size       | the size of each file or directory in octets. This    |
|            | is a descriptor, varying as needed in each entry for  |
|            | the size of the file. For convenience in the figure,  |
|            | it is shown here as a 16-bit descriptor for a small   |
|            | file.                                                 |
| Mtime      | a timestamp showing when the file or directory was    |
|            | modified.                                             |
| Ctime      | a timestamp of the last status change for this file   |
|            | or directory.                                         |
| File Path  | contains the file's name relative within the          |
|            | requested path of the _getdir_ session, a maximum of  |
|            | 1024-octet UTF-8 string, which is null-terminated to  |
|            | indicate its end.  The File Path may contain          |
|            | additional null padding in the null termination to    |
|            | allow Directory Entries to each be allocated a fixed  |
|            | amount of space or to place an integer number of      |
|            | Directory Entries in each DATA packet for debugging   |
|            | purposes.                                             |
+------------+-------------------------------------------------------+
```

The first bit of the Directory Entry is always 1, to indicate the
start of the record and the end of any padding from previous
Directory Entries.

```
+-------+-------+--------------------+
| Bit 6 | Bit 7 | Properties conveyed |
+-------+-------+--------------------+
| 0     | 0     | normal file.       |
| 0     | 1     | normal directory.  |
| 1     | 0     | special file.      |
| 1     | 1     | special directory. |
+-------+-------+--------------------+
```

Streams listed in a directory should be marked as special. If a
stream is being transferred, its size is unknown -- otherwise it
would be a file. The size property of a Directory Entry for a stream
is therefore expected to be zero.

```
+-------+-------+-------------------------------------------------+
| Bit 8 | Bit 9 | Properties conveyed                             |
+-------+-------+-------------------------------------------------+
| 0     | 0     | File size is indicated in a 16-bit descriptor.  |
| 0     | 1     | File size is indicated in a 32-bit descriptor.  |
| 1     | 0     | File size is indicated in a 64-bit descriptor.  |
| 1     | 1     | File size is indicated in a 128-bit descriptor. |
+-------+-------+-------------------------------------------------+
```

Flag bits 8 and 9 of Properties are descriptor size flags, with
similar meaning as before, describing the size of the File Size
descriptor that follows the Properties field. When a single
Directory Entry appears in the METADATA packet, these flags SHOULD
match flag bits 8 and 9 in the METADATA header. (A smaller
descriptor size may be indicated in the Directory Entry when doing
test transfers of small files using large descriptors.)

```
+--------+------------------------------------+
| Bit 10 | Properties conveyed                |
+--------+------------------------------------+
| 0      | File really is a file.             |
| 1      | File is to be treated as a bundle. |
+--------+------------------------------------+
```

Bit 10 of Directory Entry Properties is a bundle flag, as indicated
in and matching the METADATA header. Use of Saratoga with bundles is
discussed further in [I-D.wood-dtnrg-saratoga].

```
+------+---------------------------------------------------------------+
| Bit  | Use                                                           |
| 13   |                                                               |
+------+---------------------------------------------------------------+
| 0    | This file's content MUST be delivered reliably without        |
|      | errors using UDP.                                             |
| 1    | This file's content MAY be delivered unreliably, or partly   |
|      | unreliably, where errors are tolerated, using UDP-Lite.       |
+------+---------------------------------------------------------------+
```

Bit 13 indicates whether the file must be sent reliably or can be
sent at least partly unreliably, using UDP-Lite. This matches
METADATA flag use.

Undefined or unused flag bits of the Properties field default to
zero. Bit 0 is always 1, to indicate the start of a Directory Entry.
In general, bits 1-7 of Properties are for matters related to the
sender's filesystem, while bits 8-15 are for matters related to
transport over Saratoga.

It may be reasonable that files are visible in Directory Entries only
when they can be transferred to the requester - this may depend on
e.g. having appropriate access permissions or being able to handle
large filesizes. But requesters only capable of handling small files
MUST be able to skip through large descriptors for large file sizes.
Directory sizes are not calculated or sent, and a Size of 0 is given
instead for directories, which are considered zero-length files.

The "epoch" format used in file creation and modification timestamps
in directory entries indicates the unsigned number of seconds since
the start of January 1, 2000 in UTC. The times MUST include all leap
seconds. Using unsigned 32-bit values means that these time fields
will not wrap until after the year 2136.

Converting from unix CTime/MTime holding a time past January 1, 2000
but with the traditional 1970 epoch means subtracting the fixed value
of 946 684 822 seconds, which includes the 22 leap seconds that were
added to UTC between 1 January 1970 and 1 January 2000. A unix time
before 2000 is rounded to January 1, 2000.

A file-receiver should preserve the timestamp information received in
the METADATA for its own copy of the file, to allow newer versions of
files to propagate and supercede older versions.

6. Behaviour of a Saratoga Peer

This section describes some details of Saratoga implementations and
uses the RFC 2119 standards language to describe which portions are
needed for interoperability.

6.1. Saratoga Sessions

Following are descriptions of the packet exchanges between two peers
for each type of session. Exchanges rely on use of the Id field to
match responses to requests, as described earlier in Section 4.2.

6.1.1. The _get_ Session

1. A peer (the file-receiver) sends a REQUEST packet to its peer
    (the file-sender). The Flags bits are set to indicate that this
    is not a _delete_ request, nor does the File Path indicate a
    directory. Each _get_ session corresponds to a single file, and

fetching multiple files requires sending multiple REQUEST packets
and using multiple different Session Ids so that responses can be
differentiated and matched to REQUESTs based on the Id field. If
a specific file is being requested, then its name is filled into
the File Path field, otherwise it is left null and the file-
sender will send a file of its choice.

2. If the _get_ request is rejected, then a STATUS packet containing
    an error code in the Status field is sent and the session is
    terminated. This STATUS packet MUST be sent to reject and
    terminate the session. The error code MAY make use of the
    "Unspecified Error" value for security reasons. Some REQUESTs
    might also be rejected for specifying files that are too large to
    have their lengths encoded within the maximum integer field width
    advertised by bits 8 and 9 of the REQUEST.

3. If the _get_ request is accepted, then a STATUS packet MAY be
    sent with an error code of 00 and an In-Response-To field of
    zero, to indicate acceptance. Sending other packets (METADATA or
    DATA) also indicates acceptance. The file-sender SHOULD generate
    and send a METADATA packet. A METADATA packet that is received
    MUST be parsed. The sender MUST send the contents of the file or
    stream as a series of DATA packets. In the absence of STATUS
    packets being requested from the receiver, if the file-sender
    believes it has finished sending the file and is not on a
    unidirectional link, it MUST send the last DATA packet with the
    Flags bit set requesting a STATUS response from the file-
    receiver. The last DATA packet MUST always have its End of Data
    (EOD) bit set. This can be followed by empty DATA packets with
    the Flags bits set with EOD and requesting a STATUS until either
    a STATUS packet is received, or the inactivity timer expires.
    All of the DATA packets MUST use field widths for the file offset
    descriptor fields that match what the Flags of the METADATA
    packet specified. Some arbitrarily selected DATA packets may
    have the Flags bit set that requests a STATUS packet. The file-
    receiver MAY voluntarily send STATUS packets at other times,
    where the In-Response-To field MUST set to zero. The file-
    receiver SHOULD voluntarily send a STATUS packet in response to
    the first DATA packet.

4. As the file-receiver takes in the DATA packets, it writes them
    into the file locally. The file-receiver keeps track of missing
    data in a hole list. Periodically the file sender will set the
    ack flag bit in a DATA packet and request a STATUS packet from
    the file-receiver. The STATUS packet can include a copy of this
    hole list if there are holes. File-receivers MUST send a STATUS
    packet immediately in response to receiving a DATA packet with
    the Flags bit set requesting a STATUS.

5. If the file-sender receives a STATUS packet with a non-zero
    number of holes, it re-fetches the file data at the specified
    offsets and re-transmits it. If the METADATA packet has not been
    received, this is indicated by a bit in the STATUS packet, and
    the METADATA packet can be retransmitted. The file-sender MUST
    retransmit data from any holes reported by the file-receiver
    before proceeding further with new DATA packets.

6. When the file-receiver has fully received the file data and any
    METADATA packet, then it sends a STATUS packet indicating that
    the session is complete, and it terminates the session locally,
    although it MUST persist in responding to any further DATA
    packets received from the file-sender with 'completed' STATUSes,
    as described in Section 4.5, for some reasonable amount of time.
    Starting a timer on sending a completed STATUS and resetting it
    whenever a received DATA/sent 'completed' STATUS session takes
    place, then removing all session state on timer expiry, is one
    approach to this.

Given that there may be a high degree of asymmetry in link bandwidth
between the file-sender and file-receiver, the STATUS packets should
be carefully generated so as to not congest the feedback path. This
means that both a file-sender should be cautious in setting the DATA
Flags bit requesting STATUSes, and also that a file-receiver should
be cautious in gratuitously generating STATUS packets of its own
volition. When sending on known unidirectional links, a file-sender
cannot reasonably expect to receive STATUS packets, so should never
request them.

6.1.2. The _getdir_ Session

A _getdir_ session to obtain a Directory Record proceeds through the
same states as the _get_ session. Rather than transferring the
contents of a file from the file-receiver to the file-sender, a set
of records representing the contents of a directory are transferred
as a file. These records can be parsed and dealt with by the file-
receiver as desired. There is no requirement that a Saratoga peer
send the full contents of a directory listing; a peer may filter the
results to only those entries that are actually accessible to the
requesting peer.

Any file system entries that would normally be contained in the
directory records, but that have sizes greater than the receiver has
indicated that it can support in its BEACON, MUST be filtered out.

6.1.3. The _delete_ Session

1. A peer sends a REQUEST packet with the bit set indicating that it
    is a deletion request and the path to be deleted is filled into
    the File Path field. The File Path MUST be filled in for
    delete sessions, unlike for _get_ sessions.

2. The other peer replies with a feedback STATUS packet whose Id
    matches the Id field of the _delete_ REQUEST. This STATUS has a
    Status code that indicates that the file is not currently present
    on the filesystem (indicated by the 00 Status field in a success
    STATUS), or whether some error occurred (indicated by the non-
    zero Status field in a failure STATUS). This STATUS packet MUST
    have no Holes and 16-bit width zero-valued Progress Indicator and
    In-Response-To fields.

   If a request is received to delete a file that is already deleted, a
   STATUS with Status code 00 and other fields as described above is
   sent back in acknowledgement. This response indicates that the
   indicated file is not present, not the exact action sequence that led
   to a not-present file. This idempotent behaviour ensures that loss
   of STATUS acknowledgements and repeated _delete_ requests are handled
   properly.

6.1.4. The _put_ Session

   A _put_ session proceeds as a _get_ does, except the file-sender and
   file-receiver roles are exchanged between peers. In a _put_ a PUT
   REQUEST is sent.

   However, in a 'blind _put_', no REQUEST packet is ever sent. The
   file-sending end senses that the session is in progress when it
   receives METADATA or DATA packets for which it has no knowledge of
   the Id field.

   If the file-receiver decides that it will store and handle the _put_
   request (at least provisionally), then it MUST send a voluntary (ie,
   not requested) success STATUS packet to the file-sender. Otherwise,
   it sends a failure STATUS packet. After sending a failure STATUS
   packet, it may ignore future packets with the same Id field from the
   file-sender, but it should, at a low rate, periodically regenerate
   the failure STATUS packet if the flow of packets does not stop.

6.2. Beacons

   Sending BEACON packets is not required in any of the sessions
   discussed in this specification, but optional BEACONs can provide
   useful information in many situations. If a node periodically

generates BEACON packets, then it should do so at a low rate which
does not significantly affect in-progress data transfers.

A node that supports multiple versions of Saratoga (e.g. version 1
from this specification along with the older version 0), MAY send
multiple BEACON packets showing different version numbers. The
version number in a single BEACON should not be used to infer the
larger set of protocol versions that a peer is compatible with.
Similarly, a node capable of communicating via IPv4 and IPv6 MAY send
separate BEACONs via both protocols, or MAY only send BEACONs on its
preferred protocol.

If a node receives BEACONs from a peer, then it SHOULD NOT attempt to
start any _get_, _getdir_, or _delete_ sessions with that peer if bit
14 is not set in the latest received BEACONs. Likewise, if received
BEACONs from a peer do not have bit 15 set, then _put_ sessions
SHOULD NOT be attempted to that peer. Unlike the capabilities bits
which prevent certain types of sessions from being attempted, the
willingness bits are advisory, and sessions MAY be attempted even if
the node is not advertising a willingness, as long as it advertises a
capability. This avoids waiting for a willingness indication across
long-delay links.

6.3. Upper-Layer Interface

No particular application interface functionality is required in
implementations of this specification. The means and degree of
access to Saratoga configuration settings, and session control that
is offered to upper layers and applications, are completely
implementation-dependent. In general, it is expected that upper
layers (or users) can set timeout values for session requests and for
inactivity periods during the session, on a per-peer or per-session
basis, but in some implementations where the Saratoga code is
restricted to run only over certain interfaces with well-understood
operational latency bounds, then these timers MAY be hard-coded.

6.4. Inactivity Timer

In order to determine the liveliness of a session, Saratoga nodes may
implement an inactivity timer for each peer they are expecting to see
packets from. For each packet received from a peer, its associated
inactivity timer is reset. If no packets are received for some
amount of time, and the inactivity timer expires, this serves as a
signal to the node that it should abort (and optionally retry) any
sessions that were in progress with the peer. Information from the
link interface (i.e. link down) can override this timer for point-to-
point links.

The actual length of time that the inactivity timer runs for is a
matter of both implementation and deployment situation. Relatively
short timers (on the order of several round-trip times) allow nodes
to quickly react to loss of contact, while longer timers allow for
session robustness in the presence of transient link problems. This
document deliberately does not specify a particular inactivity timer
value nor any rules for setting the inactivity timer, because the
protocol is intended to be used in both long- and short-delay
regimes.

Specifically, the inactivity timer is started on sending REQUEST or
STATUS packets. When sending packets not expected to elicit
responses (BEACON, METADATA, or DATA without acknowledgement
requests), there is no point to starting the local inactivity timer.

For normal file transfers, there are simple rules for handling
expiration of the inactivity timer during a _get_ or _put_ session.
Once the timer expires, the file-sender SHOULD terminate the session
state and cease to send DATA or METADATA packets. The file-receiver
SHOULD stop sending STATUS packets, and MAY choose to store the file
in some cache location so that the transfer can be recovered. This
is possible by waiting for an opportunity to re-attempt the session
and immediately sending a STATUS that only lists the parts of the
file not yet received if the session is granted. In any case, a
partially-received file MUST NOT be handled in any way that would
allow another application to think it is complete.

The file-sender may implement more complex timers to allow rate-based
pacing or simple congestion control using information provided in
STATUS packets, but such possible timers and their effects are
deliberately not specified here.

6.5. Streams and wrapping

When sending an indefinite-length stream, the possibility of offset
sequence numbers wrapping back to zero must be considered. This can
be protected against by using large offsets, and by the stream
receiver. The receiver MUST separate out holes before the offset
wraps to zero from holes after the wrap, and send Hole definitions in
different STATUS packets, with Flag 14 set to mark them as
incomplete. Any Hole straddling a sequence wrap MUST be broken into
two separate Holes, with the second Hole starting at zero. The
timestamps in STATUS packets carrying any pre-wrap holes should be
earlier than the timestamp in later packets, and should repeat the
timestamp of the last DATA packet seen for that offset sequence
before the following wrap to zero occurred. Receivers indicate that
they no longer wish to receive streams by sending Status Code 0C.

6.6. **Completing file delivery and ending the session**

   The sender infers a completely-received transfer from the reported
   receiver window position. In the final STATUS packet sent by the
   receiver once the file to be transferred has been completely
   received, bit 14 MUST be 0 (indicating a complete set of holes in
   this packet), there MUST NOT be any holestofill offset pairs
   indicating holes, the In-Response-To and Progress Indicator fields
   contain the length of the file (i.e. point to the next octet after
   the file), and the voluntary flag MUST be set. This 'completed'
   STATUS may be repeated, depending on subsequent sender behaviour,
   while internal state about the transfer remains available to the
   receiver.

   Because METADATA not mandatory for implementations, the file receiver
   may not know the length of a file if METADATA is never sent. The
   sender MUST set the EOD End of Data flag in each DATA packet that
   sends the last byte of the file, and SHOULD request a STATUS
   acknowledgement when the EOD flag is set. If METADATA has been sent
   and the EOD comes earlier than a previously reported length of a
   file, an unspecified error 0x01, as described below, is returned in
   the STATUS message responding to that DATA packet and EOD flag. If a
   stream is being marked EOD, the receiver acknowledges this with a
   Success 0x00 code.

7. **Implementation Development**

   There is a mailing list for discussion of Saratoga and its
   implementations. Contact Lloyd Wood for details. Further
   information is at: http://saratoga.sourceforge.net/

8. **Security Considerations**

   The design of Saratoga provides limited, deliberately lightweight,
   services for authentication of session requests, and for
   authentication or encryption of data files via keyed metadata
   checksums. This document does not specify privacy or access control
   for data files transferred. Privacy, access, authentication and
   encryption issues may be addressed within an implementation or
   deployment in several ways that do not affect the file transfer
   protocol itself. As examples, IPSec may be used to protect Saratoga
   implementations from forged packets, to provide privacy, or to
   authenticate the identity of a peer. Other implementation-specific
   or configuration-specific mechanisms and policies might also be
   employed for authentication and authorization of requests.
   Protection of file data and meta-data can also be provided by a
   higher-level file encryption facility. If IPsec is not required, use
   of encryption before the file is given to Saratoga is preferable.

Basic security practices like not accepting paths with "..", not
following symbolic links, and using a chroot() system call, among
others, should also be considered within an implementation.

Note that Saratoga is intended for single-hop transfers between
peers. A METADATA checksum using a previously shared key can be used
to decrypt or authenticate delivered DATA files. Saratoga can only
provide payload encryption across a single Saratoga transfer, not
end-to-end across concatenated separate hop-by-hop transfers through
untrusted peers, as checksum verification of file integrity is
required at each node. End-to-end data encryption, if required, MUST
be implemented by the application using Saratoga.

9. IANA Considerations

IANA has allocated port 7542 (tcp/udp) for use by Saratoga.

saratoga   7542/tcp   Saratoga Transfer Protocol
saratoga   7542/udp   Saratoga Transfer Protocol

IANA has allocated a dedicated IPv4 all-hosts multicast address
(224.0.0.108) and a dedicated IPv6 link-local multicast addresses
(FF02:0:0:0:0:0:0:6c) for use by Saratoga.

10. Acknowledgements

Developing and deploying the on-orbit IP-based infrastructure of the
Disaster Monitoring Constellation, in which Saratoga has proven
useful, has taken the efforts of hundreds of people over more than a
decade. We thank them all.

We thank James H. McKim as an early contributor to Saratoga
implementations and specifications, while working for RSIS
Information Systems at NASA Glenn. We regard Jim as an author of
this document, but are prevented by the boilerplate five-author limit
from naming him earlier.

We thank Stewart Bryant, Dale Mellor, Cathryn Peoples, Kerrin Pine,
Abu Zafar Shahriar and Dave Stewart for their review comments.

Work on this specification at NASA's Glenn Research Center was funded
by NASA's Earth Science Technology Office (ESTO).

11. A Note on Naming

Saratoga is named for the USS Saratoga (CV-3), the aircraft carrier
sunk at Bikini Atoll that is now a popular diving site.

12. References

12.1. Normative References

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768,
           August 1980.

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321,
           April 1992.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
           Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3309] Stone, J., Stewart, R., and D. Otis, "Stream Control
           Transmission Protocol (SCTP) Checksum Change", RFC 3309,
           September 2002.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO
           10646", STD 63, RFC 3629, November 2003.

12.2. Informative References

[Brenchley12]
           Brenchley, M., Garner, P., Cawthorne, A., Wisniewska, K.,
           and P. Davies, "Bridging the Abyss - Agile Data Dwonlink
           Solutions for the Disaster Monitoring Constellation",
           Small Satellites Systems and Services (4S) Symposium,
           European Space Agency, Portoroz, Slovenia, June 2012.

[Hogie05] Hogie, K., Criscuolo, E., and R. Parise, "Using Standard
           Internet Protocols and Applications in Space", Computer
           Networks, Special Issue on Interplanetary Internet, vol.
           47, no. 5, pp. 603-650, April 2005.

[I-D.wood-dtnrg-saratoga]
           Wood, L., McKim, J., Eddy, W., Ivancic, W., and C.
           Jackson, "Using Saratoga with a Bundle Agent as a
           Convergence Layer for Delay-Tolerant Networking", draft-
           wood-dtnrg-saratoga-14 (work in progress) , April 2014.

[I-D.wood-tsvwg-saratoga-congestion-control]
           Wood, L., Eddy, W., and W. Ivancic, "Congestion control
           for the Saratoga protocol", draft-wood-tsvwg-saratoga-
           congestion-control-05 (work in progress) , April 2014.

[Ivancic10]
          Ivancic, W., Eddy, W., Stewart, D., Wood, L., Northam, J.,
          and C. Jackson, "Experience with delay-tolerant networking
          from orbit", International Journal of Satellite
          Communications and Networking, Special Issue on best
          papers of the Fourth Advanced Satellite Mobile Systems
          Conference (ASMS 2008), vol. 28, issues 5-6, pp. 335-351,
          September-December 2010.

[Jackson04]
          Jackson, C., "Saratoga File Transfer Protocol", Surrey
          Satellite Technology Ltd internal technical document ,
          2004.

[RFC0959] Postel, J. and J. Reynolds, "File Transfer Protocol", STD
          9, RFC 959, October 1985.

[RFC3828] Larzon, L-A., Degermark, M., Pink, S., Jonsson, L-E., and
          G. Fairhurst, "The Lightweight User Datagram Protocol
          (UDP-Lite)", RFC 3828, July 2004.

[RFC5050] Scott, K. and S. Burleigh, "Bundle Protocol
          Specification", RFC 5050, November 2007.

[RFC5348] Floyd, S., Handley, M., Padhye, J., and J. Widmer, "TCP
          Friendly Rate Control (TFRC): Protocol Specification", RFC
          5348, September 2008.

[RFC5405] Eggert, L. and G. Fairhurst, "Unicast UDP Usage Guidelines
          for Application Designers", BCP 145, RFC 5405, November
          2008.

[RFC6151] Turner, S. and L. Chen, "Updated Security Considerations
          for the MD5 Message-Digest and the HMAC-MD5 Algorithms",
          RFC 6151, March 2011.

[Wood07a] Wood, L., Ivancic, W., Hodgson, D., Miller, E., Conner,
          B., Lynch, S., Jackson, C., da Silva Curiel, A., Cooke,
          D., Shell, D., Walke, J., and D. Stewart, "Using Internet
          Nodes and Routers Onboard Satellites", International
          Journal of Satellite Communications and Networking,
          Special Issue on Space Networks, vol. 25, no. 2, pp.
          195-216, March/April 2007.

   [Wood07b]   Wood, L., Eddy, W., Ivancic, W., Miller, E., McKim, J.,
               and C. Jackson, "Saratoga: a Delay-Tolerant Networking
               convergence layer with efficient link utilization",
               International Workshop on Satellite and Space
               Communications (IWSSC '07) Salzburg, September 2007.

   [Wood11]   Wood, L., Smith, C., Eddy, W., Ivancic, W., and C.
               Jackson, "Taking Saratoga from space-based ground sensors
               to ground-based space sensors", IEEE Aerospace Conference
               Big Sky, Montana, March 2011.

Appendix A. Timestamp/Nonce field considerations

   Timestamps are useful in DATA packets when the time that the packet
   or its payload was generated is of importance; this can be necessary
   when streaming sensor data recorded and packetized in real time. The
   format of the optional timestamp, whose presence is indicated by a
   flag bit, is implementation-dependent within the available fixed-
   length 128-bit field. How the contents of this timestamp field are
   used and interpreted depends on local needs and conventions and the
   local implementation.

   However, one simple suggested format for timestamps is to begin with
   a POSIX time_t representation of time, in network byte order. This
   is either a 32-bit or 64-bit signed integer representing the number
   of seconds since 1970. The remainder of this field can be used
   either for a representation of elapsed time within the current
   second, if that level of accuracy is required, or as a nonce field
   uniquely identifying the packet or including other information. Any
   locally-meaningful flags identifying a type of timestamp or timebase
   can be included before the end of the field. Unused parts of this
   field MUST be set to zero.

   There are many different representations of timestamps and timebases,
   and this draft is too short to cover them in detail. One suggested
   flag representation of different timestamp fields is to use the least
   significant bits at the end of the timestamp/nonce field as:

```
+--------+-------------------------------------------------------+
| Status | Meaning                                               |
| Value  |                                                       |
+--------+-------------------------------------------------------+
| 00     | No flags set, local interpretation of field.          |
| 01     | 32-bit POSIX timestamp at start of field indicating   |
|        | whole seconds from epoch.                             |
| 02     | 64-bit POSIX timestamp at start of field indicating   |
|        | whole seconds elapsed from epoch.                     |
| 03     | 32-bit POSIX timestamp, as in 01, followed by 32-bit  |
|        | timestamp indicating fraction of the second elapsed.  |
| 04     | 64-bit POSIX timestamp, as in 02, followed by 32-bit  |
|        | timestamp indicating fraction of the second elapsed.  |
| 05     | 32-bit timestamp giving seconds elapsed since the 2000 |
|        | epoch, as in file timestamps. This option is likely only |
|        | useful for very slow links.                           |
+--------+-------------------------------------------------------+
```

Other values may indicate specific epochs or timebases, as local
requirements dictate. There are many ways to define and use time
usefully.

Echoing timestamps back to the file-sender is also useful for
tracking flow conditions. This does not require the echoing receiver
to understand the timestamp format or values in use. The use of
timestamp values may assist in developing algorithms for flow control
(including TCP-Friendly Rate Control
[I-D.wood-tsvwg-saratoga-congestion-control]) or other purposes.
Timestamp values provide a useful mechanism for Saratoga peers to
measure path and round-trip latency.

Authors' Addresses

   Lloyd Wood
   University of Surrey alumni
   Sydney, New South Wales
   Australia

   Email: L.Wood@society.surrey.ac.uk

      Wesley M. Eddy
      MTI Systems
      MS 500-ASRC
      NASA Glenn Research Center
      21000 Brookpark Road
      Cleveland, OH 44135
      USA

      Phone: +1-216-433-6682
      Email: wes@mti-systems.com

      Charles Smith
      Vallona Networks
      7 Wattle Crescent
      Phegans Bay, New South Wales 2256
      Australia

      Phone: +61-404-05-8974
      Email: charlesetsmith@me.com

      Will Ivancic
      NASA Glenn Research Center
      21000 Brookpark Road, MS 54-5
      Cleveland, OH 44135
      USA

      Phone: +1-216-433-3494
      Email: William.D.Ivancic@grc.nasa.gov

      Chris Jackson
      Surrey Satellite Technology Ltd
      Tycho House
      Surrey Space Centre
      20 Stephenson Road
      Guildford, Surrey GU2 7YE
      United Kingdom

      Phone: +44-1483-803803
      Email: C.Jackson@sstl.co.uk